KIP-878: Internal Topic Autoscaling for Kafka Streams

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-14318*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

The fundamental unit of work in Kafka Streams is the StreamTask, with a 1:1 mapping from task to partition (technically partition number as in the case of e.g. joins, one task may process two partitions from two different input topics). Because of this, the number of partitions of the user input topics represent an upper limit on the parallelism and ability to scale out. Currently this is effectively a permanent upper limit as Streams assumes the partition count is fixed, and will shut itself down if any changes are detected as the internal topics will no longer match the partitioning of the user topics. The only way to increase the parallelism in today's world is by stopping Streams and manually repartitioning not just the external user topics but also any internal topics (such as repartitions and changelogs). Besides being massively inconvenient and requiring a long pause in processing, the internal topics are generally to be managed by Streams itself and may not all be known to the user.

Therefore, we intend to introduce automatic and live scaling out by expanding the partition count to match any increase in the partitions on the external user topics. Users can simply repartition the topics they manage, and Streams will detect this and scale itself to match without any downtime.

One of the main challenges to supporting repartitioning within Streams is the fact that partitioning generally (and by default) depends on the current number of partitions. Thus, any change to the partition count will result in keys being routed to a new/different partition number, effectively losing all the state it had built up so far. To ensure correctness of results in the face of changing partition count, there are two options: (a) manually reprocess **all** data in the application's history and migrate it to new topics, or (b) ensure that keys continue to be routed to the same partition number, regardless of changes to the total partition count. To keep things lightweight the autoscaling feature introduced by this KIP will *not* attempt to repartition historical data or state, which can be tackled in a future KIP if there is demand. Instead, this KIP intends to solve the partition scaling problem for the subset of applications that are able to tolerate partitions being added, such as those which are **stateless** or **statically partitioned**.

Public Interfaces

The only public facing changes in this KIP will be the introduction of two new metrics and configs related to autoscaling. The first config is a simple feature flag that users can turn on (or off) to control whether Streams will automatically react to partition expansions upstream by expanding the internal topics to match. When this is *not* enabled, if input topics have their partition count changed in a way that is incompatible with the current application state (as discussed above), Streams will log the details and send the INCOMPLETE_SOURCE_TOPIC_METADATA error code to all clients as is the case today.

In addition, there will be a new timeout config added for users to bound the maximum time that Streams will spend attempting to retry partition autoscaling, in case of failures or other edge cases. This is discussed in more detail below.

StreamsConfig.java

```
// default to false, priority LOW
public static final String PARTITION_AUTOSCALING_ENABLED_CONFIG = "partition.autoscaling.enabled";
private static final String PARTITION_AUTOSCALING_ENABLED_DOC = "Enable autoscaling the partitions of internal
topics which are managed by Streams. If an internal topic's partition count depends on an upstream input topic
(or topics), then expanding the number of partitions on the input topic(s) will result in the internal topic(s)
automatically being expanded to match.";

// default to 15 minutes, priority LOW
public static final String PARTITION_AUTOSCALING_TIMEOUT_MS_CONFIG = "partition.autoscaling.timeout.ms";
private static final String PARTITION_AUTOSCALING_TIMEOUT_MS_DOC = "The maximum amount of time in milliseconds
that Streams will attempt to retry autoscaling of internal topic partitions.";
```

We will also provide three new **INFO** metrics as a means of monitoring the status and completion of autoscaling events. The first two will be subtopology-level counters for the current and expected/target number of partitions (and thus tasks/parallelism):

`current-subtopology-parallelism`

`expected-subtopology-parallelism` (only reported when feature flag is turned on)

- type = stream-subtopology-metrics
- subtopology-id=[subtopologyId] (also called groupId sometimes)

The other metric will track failures where Streams has given up on retrying at the client level. This can be monitored to send an alert when Streams is struggling to successfully add partitions, for example due to a quota violation, and indicates an issue that should be investigated further.

`num-autoscaling-failures` (only be reported when the feature flag is turned on.)

- type = stream-client-metrics
- client-id=[clientId]

Proposed Changes

As discussed above, the autoscaling feature can be enabled for any kind of application and allow Streams to handle partition expansion of its input topics. When upstream topics change in partition count, this can/will change the expected number of partitions for downstream repartition topics and changelogs. Normally upon startup, these internal topics are created during the first rebalance: during (any) rebalance, Streams will process the topology to resolve the intended number of partitions for all topics and create them topics if necessary. If those topics already exist, we validate their partition count instead. With this KIP, when discrepancies are found during the validation step, we will branch the handling logic on the feature flag: when enabled, Streams will now attempt to "fix" any such discrepancies by requesting the internal topics be expanded to match the intended number of partitions. **Note that we will still trigger an error if internal topics are found to be over-partitioned**, that is, because you can only increase the partition count of a topic and not decrease, finding internal topics with **more** than the expected number indicates a user error/misconfiguration and is not something Streams can fix.

We can rely on the fact that a change in the partitions of any topic the StreamThread is subscribed to is guaranteed to trigger a rebalance – this will occur when the metadata refreshes, so we should begin autoscaling within the ***metadata.max.age.ms*** of a partition change. In the case of source nodes with more than one source topic, multiple rebalances may be triggered if these topics are expanded out of sync with enough time between them. **In the case of multiple source topics, Streams will begin autoscaling internal topics when the *first* source topic is expanded.**

Of course, it's always possible for something to go awry and cause the topic scaling to fail: request timeouts, internal errors, and quota violations should all be anticipated. To handle such failures, Streams will automatically retry until the operation is successful or it reaches the maximum allowed retry time. This will be configurable for users via the new ***partition.autoscaling.timeout.ms*** config, which will start counting after the first failure (rather than when the autoscaling attempt began). The timeout will be reset after any success, so in the case of interleaving success and failure we would still only time out if there is no progress at all over the full timeout that's configured.

If the *Admin#createPartitions* call does fail, Streams will schedule a followup rebalance to re-attempt the autoscaling. This is because the API is not atomic and may succeed for some topics but not others, so we want to refresh the cluster metadata and refine the set of topics to expand when retrying. If the timeout has been exceeded, a warning will be logged instead of trying to create partitions and Streams will continue on. Note that if successful, we will need to trigger a final rebalance after all internal topics have been expanded in order to assign those partitions from the updated cluster metadata. The final followup will be scheduled for 10 seconds after the successful request, as the javadocs for *Admin#createPartitions* call out that

> *It may take several seconds after this method returns success for all the brokers to become aware that the partitions have been created.*

The final change of note is the two new metrics. The `num-autoscaling-failures` is self-explanatory, incrementing each time the ***partition.autoscaling.timeout.ms*** is exceeded and Streams has to stop retrying. The `subtopology-parallelism` metric will reflect the current number of partitions/tasks, and thus the degree of parallelism, for each subtopology. **Note: users should roll this up and take the maximum value across all clients**, since only the group leader will process any expansions that occur and the new partitions will not necessarily be noticed by all clients/threads. The point of this metric is to provide visibility into the status of autoscaling and let users know when Streams has finished successfully expanding all internal topics, since they should wait for this before sending any records to the new input topic partitions.

Compatibility, Deprecation, and Migration Plan

Since this is a new feature there are no compatibility concerns or need for a migration plan. Note that the feature flag can be enabled & disabled again at will, without impact to operations.

Test Plan

We plan to add a new system test to the Streams suite for this feature. This will be fairly straightforward and cover starting up a Streams app, changing the number of partitions for external topics (following the procedure laid out above), validating that the internal topics have expanded to match the new partition count, and finally verifying that the application has been producing correct/expected results.

The application in question should cover multiple sub-topologies and include any operators that have special dependencies and/or effects related to partitioning, such as: a Repartition, a Join, and a ForeignKeyJoin. We might also want to cover applications built via the PAPI.

Rejected Alternatives

1. As mentioned briefly in the 'Motivation' section, an alternative to enforcing the static partitioning condition would be to implement some kind of manual data migration via historical reprocessing. While this would enable the feature to be used for all applications, it would have extreme downsides and make the feature itself far less useful by requiring long downtimes, as well as being incredibly resource heavy. Thus we feel it's more practical to introduce the best version of the feature by imposing a constraint: static partitioning.
2. To prevent users from shooting themselves in the foot, this KIP chose not to use a feature flag in the initial proposal, and instead to expose the feature via a special static partitioner interface to discourage turning on the feature in applications that aren't compatible with autoscaling. However since this is an advanced feature anyways, we decided to trust advanced users to know what they are doing and allow full control over this feature.
3. Considered including a new config for a default partitioner, but this turned out to be non-trivial and was extracted for a followup PR