

Discussion: Topology Optimizations for Footprint Reduction

Background and Motivation


The persistent storage footprint of a Kafka Streams application contains the following aspects:

1. The internal topics created on the Kafka cluster side.
2. The materialized state stores on the Kafka Streams application instances side.

There have been some questions about reducing these footprints, especially since many of them are not necessary. For example, there are redundant internal topics, as well as unnecessary state stores that takes up space but also affect performance. When people are pushing Streams to production with high traffic, this issue would be more common and severe. Reducing the footprint of Streams have clear benefits of Kafka Streams operations.

Proposal

There are a few tasks that we have summarized so far, under

 Unable to render Jira issues macro, execution error.

(sorted in terms of

ROI):

1. **Reduce #.state stores for KTable, hence #. Changelog topics.** Today we always “physically” materialize a KTable when user give a queryable name, but we can consider only “logically” materialize a KTable, for example for KTables generated from filter / mapValues / etc.

Post KIP-182 would be a good timing to do this task, as we have layered the appropriate public APIs for this, so the code refactoring for this project would be simpler. It would be a medium investment with high return project.

2. **Reduce #.repartition topics.** The repartition topics for aggregations and joins may be duplicated, i.e. containing exactly the same data as other topics. We can refactor our DSL translation mechanism from operator-by-operator to some global policy with topology optimizations to remove such duplicates. Example JIRA: [KAFKA-4601](#)

The investment of this sub-task may be high due to DSL translation refactoring, but this big project can also come with big returns since topology optimization can also benefit KSQL.


3. **Reduce #.changelog topics.** The source topics of the KTable can be also used as its changelog topics, so we do not need to have a duplicate changelog topic. Today we already do that for external source topics, but not for internal repartition topics.

The investment of this task is low, but its impact may also be low since this scenario would be less common.

4. **Reduce #.state stores for KStream, hence #. Changelog topics.** Today the only place that we would require a state store for KStream is stream-stream joins, however it is sub-optimal to use a windowed-kv-store for such scenarios. What's more is that if the joining KStreams are directly from some Kafka topics, then the changelog topic for these state stores would just be the duplicate of theses source topics of the Stream, since the store is append-only; even if there is no direct Kafka topics for the joining KStreams, we can still consider using the “indirect” source topic and apply the operators on-the-fly when resuming the joins.

If we remove the changelog topics completely for state stores of stream-stream joins, we need to consider how to restore the state from other topics (could be the source topic / repartition topic, etc); furthermore we can replace the current implementation of stream-stream joins. For

example:

 Unable to render Jira issues macro, execution error.

This would be a medium to high investment, and the return is unknown since we are not sure how common is stream-stream join in practice.

I'd like to proposal a very high-level solution for this general improvement. The key is to avoid “one-patch-per-optimization” approach, which would make the parsing logic very complicated over time. And I feel it is important to discuss and communicate this principle sooner than later in our team as well as to the community as we have observed people starting to pick some of them up.

Here is the high-level proposal:

1. Internal Streams Builder would keep an internal object that keep track of the currently built-in-progress topology. This is to allow them to go beyond the independent "one-operator-at-time" building process.
2. When parsing a new operator, comparing its specifications (the involved processor, the required queryable state, whether it is logging enabled, etc) with the current built-in-progress topology, and decide to add new processor / modify existing processor, with building decisions like adding new / re-using state stores (in the KTableXXProcessorSupplier), adding new / reusing internal topics, etc.
3. Admittedly such decisions may not be optimally made statically without taking workloads etc, but for the first version I think it is acceptable with hard-written rules; but we'd better capture this rule-based framework in a single class / a few classes that can be independently interfaced with its calling classes and be extended to more complicated optimization methods in the future.