# KIP-95: Incremental Batch Processing for Kafka Streams



## Status

**Current state**: *Under Discussion*

**Discussion thread**: [DISCUSS] KIP-95: Incremental Batch Processing for Kafka Streams

**JIRA**: ⚠ Unable to render Jira issues macro, execution error.

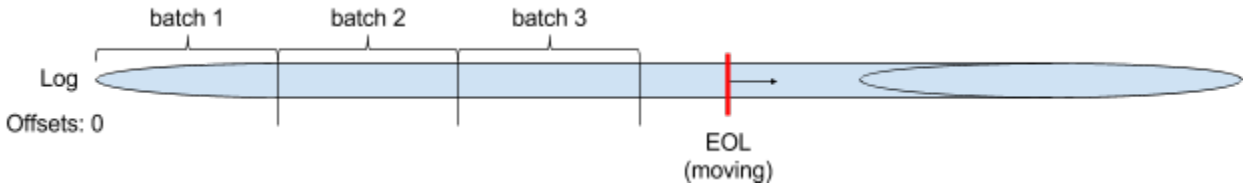
**Released:** TDB

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

## Motivation

Kafka has the vision to unify stream and batch processing with the log as central data structure (ground truth). With this KIP, we want to enlarge the scope Kafka Streams covers, with the most basic batch processing pattern: incremental processing.

By incremental processing, we refer to the case that data is collected for some time frame, and an application is being started periodically to process all the newly collected data so far, similar to a “batch job” in Hadoop. For example, some data pipeline creates a new file of collected data each hour. Thus, whenever a new file is available, a new batch job is started to process the file. To carry over this scenario to Kafka, producers would continuously write data into a topic, and the user want to schedule a recurring “batch” job, that processes everything written “so far” (here, “EOL” stands for “end-of-log”):




As the vision is to unify batch and stream processing, a regular Kafka Streams application will be used to write the batch job. Because currently only continuous queries are supported via Kafka Streams, we want to add an “auto stop” feature that terminate a stream application when it has processed all the data that was newly available at the time the application started. Thus, on each restart the application will resume where it stopped before, thus effectively chopping the log into finite chunks that are being processed one after another.

## Public Interfaces

To really unify batch and stream processing, the program itself should not be changed. Thus, we only add a new configuration parameter for Kafka Streams (i.e. `StreamsConfig`) that is used to enable “auto stop”:

- parameter name: `autostop.at`
- possible values: `"eol"`
- default: `null` (i.e. disabled)

We disable "auto stop by default to be backward compatible.

# Proposed Changes

## Determining stopping point: End-Of-Log

We want to stop the application when it reaches end-of-log, which we define as the current high watermark, where “current” refers to the time when the application was (re)started. Using the high watermark gives an offset for each input partition that collectively define the end of the input data for the current run. The challenge is as follows: If we pick up the high watermark on startup we must avoid that on failure restart we pick up a new high watermark, which might have moved ahead if new messages were written to an input topic in between.

There would be two issues if we pick up a new high watermark on failure restart:

- First, we would not stop at the point the user expects the application to stop itself.
- Second, there might be the rare case in which an application never terminates, because it keeps failing, and thus the watermark keeps moving and the application never reaches the watermark (even if it does make progress in each run).

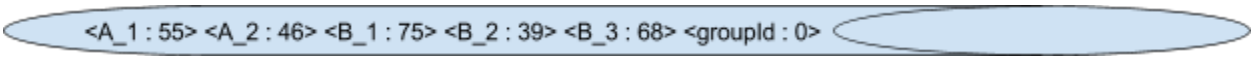
To guard against this issue, we want to introduce a Streams metadata topic (single partitioned and log compacted; one for each application-ID) that contains the required stop offset information. On startup the information is written and on failure the information is recovered. At a clean shutdown, we “clear” the metadata topic.

### The startup algorithm would be as follows:

Only for consumer group leader:

1. On `#assign()`, check the metadata topic for end-offsets
2. If no end-offsets are found (this is the initial startup case)
    a. Get the high watermarks of all input topic partitions and write them into the metadata topic
    (to make this failsafe, we need to write all or nothing; to guard against failure, we write a **completed marker** message after the last offset information -- in (1), end-offsets are only valid if we find the **completed marker** and no following marker tombstone -- see below)
3. If end-offsets are found (this is the restart on failure case)
    a. do nothing

Because there will be just one leader, this operation is single-threaded and thus safe. After application startup, the metadata topic would contain the following data (assuming we have two input topics A and B with 2 and 3 partitions, respectively). Message format is `key=topic_partition` and `value=stop-offset`. The **completed marker** uses the applicationId (i.e., *groupId)* as key and a value of zero (value must not be null as it is not a tombstone).



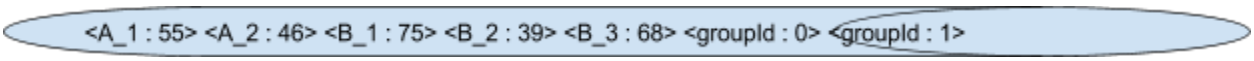

Startup and termination for all group members (this happens after `#assign()`):

1. Read the whole metadata topic to get stop-offsets for all partitions
(because the metadata topic must be consumed by all instances, we need to assign the topic’s partitions manually and do not commit offsets -- we also need to `seekToBeginning()` each time we consume the metadata topic)
2. If end-offset is reached, commit the last offset and stop processing this partition (i.e., pause the partition)
3. If no partitions are left to be processed, terminate yourself cleanly
    a. If application is the last instance (last running member of the group, i.e., the application will also be the group leader and will know if it is the only running instance), write finished marker (`key=applicationId` and `value=1`)

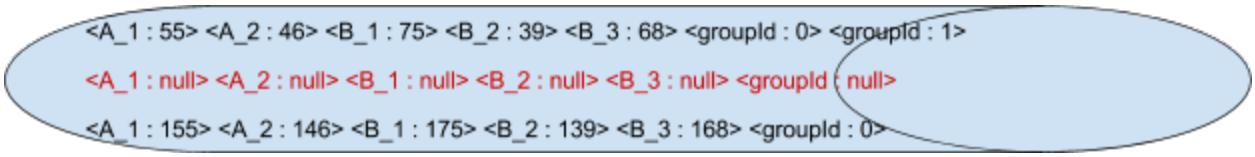
## Restart in failure-case and clean shutdown:

If an error occurs and partitions are assigned, no special action is required, as all partitions stop-offsets are known anyway. In instance termination case, a rebalance will happen eventually. However, the application instance which gets the finished partitions assigned, will not process any more data for those partitions as the latest committed offset matches the stop-offset.

After the last instance did terminate itself, the metadata topic will have one more record written.



The writing order into the metadata topic ensures, that we shut down cleanly. All writes happens in a single thread (either leader or last running instance) thus there is no race condition. To ensure cleanup, on application startup, we write tombstone messages for all previously read data. For example, after a second startup of the application the metadata topic will be as follows (assuming that 100 record got written to each partition in the meantime):

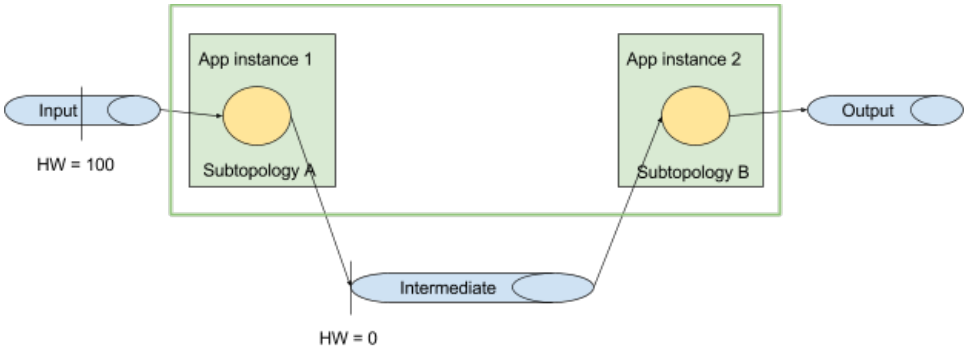



Before the new high watermark are written, we write tombstones (red messages) for each message of the previous run. Log compaction ensures that the topic gets cleaned up eventually. Intermediate log compaction cannot mess with running application because we only write tombstones on new application startup “deleting” messages from a previous run. Thus, even if something goes wrong writing tombstones, we are always in a consistent state.

## Handling Intermediate Topics

Intermediate topic need to be handled differently than input topics:

1. assume you have a topology with two sub-topologies that are connected via `through()` (i.e., an intermediate user topic) or an internal repartitioning topic
2. the application consumes a single input topic with one partition
3. the intermediate topic also has one partition
4. the application is started with two instances
5. this will result in a task assignment such that each application instance hosts one sub-topology, as shown below




6. on startup, the high watermark could be 100 for the input topic, and it will be 0 for the empty intermediate topic
7. because the intermediate topic is basically an input topic for subtopology B, subtopology B would process all data up to high watermark zero, i.e., will not process anything and shut down immediately (if we handle intermediate topics the same way as input topics)
8. thus, the user will not see any output data in the output topic
9. on the second application run, input high watermark might be 200 and intermediate high watermark might be 60 (depending on how many records subtopoloy A wrote in the first run)
10. thus, subtopology B would now process the data from 0 to 60 (but still not process anything from the current run)
11. if we have more subtopologies like this, each one processes data from "the run before", the output topic might get data quite late (i.e., for N subtopologies, the result of the first run, would be written to the output topic in the N-th application run)

Thus, we need to handle intermediate topics differently and exclude them from the high watermark shutdown approach. The problem is, that we do not know the offset for which subtopology B should stop processing, because this offsets depends on the number of output records subtopology A writes into the intermediate topic. On the other hand, in contrast to input topics, subtopology B cannot read “too much” data, because the intermediate topic will only contain the data subtopology A did write while processing up to its input topics’ stop-offsets. Thus, intermediate topics can just be consumed without any worries about “reading too much”.

Hence, subtopology B can just terminate, if there is no more new data in the intermediate topic. To guard against an early shutdown (it could happen that subtopolgy A does not produce data fast enough and the intermediate topic is “empty” in between even if subtopology A will write more data later on), we also exploit the metadata topic. Subtopology B processes data as long as it did not get the final offset of the intermediate topic from the metadata topic. For this, subtopology A must write the last write offsets of the intermediate topics into the metadata topic before shutting down (see next paragraph for more details). To make subtopology B pick up the written offsets, it must consume from the metadata topic continuously.

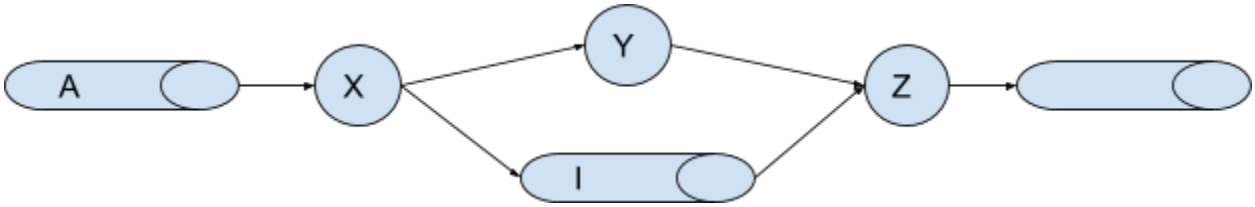
The metadata topic would look like this after shutdown (with input topic A and intermediate topic I -- this is actually not the final design but illustrates the idea -- c.f. next paragraph *"Multiple producers and deadlock issue"*):

When instance #1 running subtopoloy A processed all 100 input records, it writes the stop-offset for the intermediate topic and terminates itself. Instance #2 will process all 60 messages before `poll()` will return no more message. Eventually, instance 2 receives stop-offset 60 (from metadata topic) and also terminates. This pick up of the high watermark can happen safely any time (i.e., before all data got processed or after). Because instance #2 is the last running member of the group, it will also write the final end marker.

In failure case, message `<I_1 : 60>` might get written again to make sure the message is there.

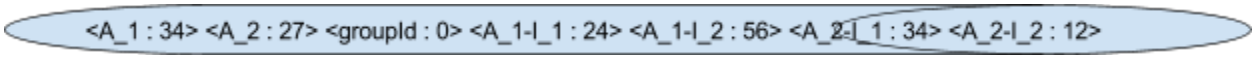
### Multiple producers and deadlock issue

The first problem in the case of multiple producers. Because intermediate topics are used for data repartitioning, all producer instances might write data to all partitions. Thus, if the first producer finishes processing and would write stop-offsets, those offsets might not be correct. A straightforward idea would be to let only the last producer that finishes processing write the intermediate topic stop-offsets (similar to the idea that the last instance writes the final “done marker”). However, this might end up in a deadlock. Assume we have a topology DAG as follows and run two instances of the application:




After both instances of operator X consumed their assigned partitions from input topic A up to the stop-offsets, neither of them will terminate because they are running in the same thread as an operator instance Z and Z cannot terminate as it did not receive the end-offset of the intermediate topic. To resolve this issue, each operator instance X needs to write “done notifications” into the metadata topic. This done notification encode *intermediate partitions and input topic partitions* that got completely processed (for the key) including the last written offset (for the value). For our two examples from above we would get (for the simple example from the beginning of the section)


and (for the deadlock example, assume A and I do both have two partitions)




For the deadlock example, operator instance X#1 processed input partitions A_1 and the last offset is it write to partitions I_1 was 24 while the last write offset for I_2 was 56. The second operator instance X#2 processed input partitions A_2 and reports write offset for I_1 as 34 and for I_2 as 12. Operator Z can receive all this metadata and will pick the largest offset reported for the intermediate partitions over all input partitions. Independent of the number of producers, there will be one message per input partition per intermediate partition and thus, operator Z knows that the highest reported offset for partition P is the final/correct one, if it received an end-offset for each partition of input topic A.

## Switching between Streaming and Batch mode

If a user wants to switch an application between (default) streaming and batch mode, we need to take special care of this.

**Batch Mode to Streaming Mode**

For this scenario, we delete the metadata topic containing stop-offsets to avoid that another switch from Streaming mode to Batch mode could pick up stall end-offsets (the user might have terminated an application running in Batch mode before the application did terminate itself cleanly, thus, switching back to batch mode mode be handles as failure restart if metadata topic is found). For the actual processing we just ignore the metadata topic and run "forever".

**Streaming Mode to Batch Mode**

As we clean the metadata topic when switching from Batch mode to Streaming mode, there is nothing special we need to do.

For the case, that an application gets started in Batch mode and the user wants to terminate the application manually, with the goal to restart with new end-offsets, we need to provide tooling to delete the currently stored end-offsets from the metadata topic. Otherwise, restart would be handled as failure restart because the application did not terminate itself in a clean manner. We can extend the Application Reset tool with an additional flag `--delete-stop-offsets` that enables the user to delete the metadata topic.

## Compatibility, Deprecation, and Migration Plan

This change is backward compatible, because "auto stop" is disabled by default.

## Test Plan

Integration tests should cover this feature. Focus will be on failure/restart scenarios to ensure application does resume and stop correctly if an instance goes down at any point in time. Need to test with different DAGs to ensure it works for all cases:

- Deadlock example DAG
- Multi-stage DAG (i.e., more than two subtopologies)

Tests with different number of partitions and threads required, too.

# Rejected Alternatives

1. Use best effort and pick up new high watermark on failure.
    - Discarded because of bad user experience (user expectation not met) and possible never terminating application.
2. Use topic embedded markers within intermediate topics to propagate “stop notification” downstream.
    - Discarded because this would only work for internal topics, but intermediate topics could also be user topics and we cannot “mess /pollute” (with) user topics.
3. Provide an external tool, that collect the current high watermark that the user runs before starting the application. Feed those high watermarks into the application as config parameter.
    - Discarded because clumsy to use and we need metadata topic anyway to handle intermediate topics.