KIP-557: Add emit on change support for Kafka Streams

Status

Current state: ~~Adopted~~ Accepted but temporarily reverted (see KAFKA-12508)

Discussion thread: *Voting Thread*

JIRA:

⚠ Unable to render Jira issues macro, execution error.

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

Note: We will now refer to no-ops as idempotent updates in this KIP. No-ops tend to sound generic, so we will use "idempotent updates" as our new terminology.

Motivation

Currently, in Kafka Streams, we support two modes of processing:

- emit on update
- emit on window close

The first one has been present for the majority of Kafka Stream's inception, when results are forwarded regardless . The second option has been added by implementing the so-called "suppression" operator which collects results throughout the window, and extending into the grace period. However, one mode of processing should've been implemented long ago, and that is *emit on change.* As mentioned in the JIRA, in many cases, the user would process a prior result to yield a new one, yet, this new result is forwarded to a downstream processor even though the operation performed had been effectively a idempotent update. This is problematic in certain circumstances where there is a high number of such operations, leading to an enormous number of useless results being forwarded down the topology.

This KIP will also discuss other possible reporting strategies for Kafka Streams and weigh its pros and cons. To be mindful of the KIP's scope, it is our intention that we restrict any changes to operations being performed on KTables.

Reporting Strategies

Reporting strategies are basically just another word for models of emission. When we emit results, and how we determine if we do it is discussed here.

1. Emit on change: The objective of this KIP is to implement this reporting strategy. The primary benefit of such an emission model is that it will drop idempotent updates, so it will be superior to emit on update in that traffic can be significantly reduced in high idempotent update traffic situations.
2. Emit on update: It has already been implemented by Kafka Streams. Basically, as long as the content or processed result is not empty, we will return the processed result. That could however lead to high idempotent update traffic, which is the primary motivation for adding emit on change.
3. Emit on window close: This has been implemented some time ago with the suppress operator which allows users to combine results in a window, and then we emit only the final result of the active window.
4. Periodic Emission: This emission model is unique in that it is solely dependent on time to emit a given result. Essentially, every time a fixed time interval has passed, then a result will be emitted.

For the above emission models, two have already been implemented, and we are aiming to implement another in this KIP. Periodic emission will likely not be added for now, as there has been little user demand for such a feature.

Proposed Behavior Changes

Definition: "idempotent update" is one in which the new result and prior result, when serialized, are identical byte arrays.

Note: an "update" is a concept that only applies to Table operations, so the concept of an "idempotent update" also only applies to Table operations. See https://kafka.apache.org/documentation/streams/developer-guide/dsl-api.html#streams_concepts_ktable for more information.

Given that definition, we propose for Streams to drop idempotent updates in any situation where it's possible and convenient to do so. For example, any time we already have both the prior and new results serialized, we may compare them, and drop the update if it is idempotent.

Note that under this proposal, we can implement idempotence checking in the following situations:
1. Any aggregation (for example, KGroupedStream, KGroupedTable, TimeWindowedKStream, and SessionWindowedKStream operations):

⚠ Unable to render Jira issues macro, execution error.

2. Any source KTable operation:

⚠ Unable to render Jira issues macro, execution error.

3. Repartition operations, when we need to send both prior and new results:

⚠ Unable to render Jira issues macro, execution error.

Metrics

The following metric will be added to record the number of idempotent updates dropped:

idempotent-update-skip : (Level 3 - Per Processor Node) DEBUG (rate | total)

Description: This metric will record the number of idempotent updates (no-ops) that have been dropped. This will be only attached to processor nodes where their operations already provides the old result. In other words, it will include repartition operations and any aggregations.

Note:

- The rate option measures the average amount of idempotent records dropped in a one second interval.
- The total option will just give a raw count of the number of records dropped. we have decided that we will forward aggregation results if and

Design Reasoning

With the current default model of emission, we are forwarding the processed results downstream regardless if it has changed or not. After some discussion, there are a couple of points that I would like to emphasize:

1. There has been some thoughts as to only coupling emit on change with the suppress operator. However, that approach has been abandoned in favor of a more extensive change which would impact almost all KTable operations supported by Kafka Streams. Our justification is that idempotent update changes should be suppressed even before they are forwarded downstream, as the duplication of these useless results has the potential to explode across multiple downstream nodes once it has been forwarded. The expected behavior is the following:
   a. Any operations that have resulted in a idempotent update would be discarded. Exceptions would be made for non-materialized KTables for performance reasons. Some might have concerns about a KTable constructed through aggregations. In this case, if a idempotent update result is produced, how do we determine which timestamp to use? To keep things simple, we have decided that we will drop aggregation results if and only if the timestamp and the value had not changed.
2. About configurations. As noted, we intend that Kafka Streams be as simple as possible when it comes to configurations. In that case, this KIP proposes that instead, emit-on-change becomes the new behavior of Kafka Streams. Since `emit-on-change` is more efficient (idempotent updates shouldn't really be sent in the first place), there is a strong case for no additional config that allows the user to "opt-out" of this change (and remain with emit-on-update).

The other crucial aspect of our proposal is why we restrict ourselves to dropping idempotent updates for operations that does not have the old result already loaded. To check for idempotence, both the old and new result has to be available. However, when the old result is unavailable, then we will have to procure that result again somehow. Unfortunately, that would exact an extra performance cost (which can be quite expensive).

Effects on Stream Time

In certain situations, idempotent updates are used to advance stream time quite aggressively. If these updates are dropped, it is possible that stream time advancement will start to stall, having some effect on operations which rely on stream time for functionality. After some consideration, we have decided that the current proposal is still satisfactory. The components which can be affected is listed below:

1. Stream aggregation (both windowed or not) will not be effected. As emphasized in the behavior changes, the current arrangement means that any operations which will advance stream time will still be emitted (as we recall earlier, operations are dropped only if the timestamp has *not* changed.)
2. Windowed operations are a primary user of stream time, so they have the largest potential to be impacted in someway by this KIP. For background, In order to advance stream time far enough to close a window or push it out of retention, the new records must have timestamps that are in later windows, which means that they are updates to *new* keys, which means they would not be suppressed as idempotent. In retrospect, we can conclude that at worst the maximum amount of stream time which can be delayed by dropped updates would be one window. Interestingly enough, this isn't too bad, since it is not in the contract to emit results at the earliest time possible (just some time after the window closes).
3. The last one is Suppression. One of the primary usages of suppression is to limit the rate of record emission, and at the same time, there should be a consistent rate of records being forwarded by the suppression operator. If we drop idempotent updates, it is possible that the buffer would grind to a halt since stream time is no longer advanced, but this isn't as big an issue as it seems. If this is the case, the user can simply remove the suppression operator from the topology (and allow the dropping of idempotent updates to fulfill the same role as the suppression buffer).

Future Work

Most would note that a pitfall with this KIP would be that we don't drop idempotent updates for stateless operations where the old result is not immediately available. As we have iterated, procuring/storing the old result in a stateless operation is not viable. In the KIP's current state, the only way to drop idempotent updates from stateless operations will be to materialize it. That of course would also have its own costs. One possible solution is instead to store the SHA256 hash of the serialized byte array instead of the whole result for stateless operations.

In this case, the amount of extra memory used would be minimized, and we would only need to compare the hash codes for equality to see if two byte arrays are equal. The main problem with this approach is that there will still be some extra I/O from inserting and reading elements of a store. For now, we have decided not to implement this feature as part of the proposal mainly because of this reason. If users report that they materialize a large number of stateless operations for the sole purpose of dropping updates, then this feature will be considered as a reasonable alternative.

Compatibility, Deprecation, and Migration Plan

If the user has logic that involves actively filtering out idempotent update operations, then they will be able to remove this code in a future release once this KIP has been integrated into Kafka Streams. For now, this KIP will aim to add a minimum number of configurations, and avoid adding APIs. If the user has any logic regarding timestamps, there might be a chance that their program will be affected. This is specific however to the individual user, and behavior will have to be changed accordingly.

Rejected Alternatives

Equality Checking using Object#equals()

We have decided to abandon using equals() to check equality for now. There are concerns that this will not be a good way to reliably compare two values without errors.

Configuration for Opting Out

For configurations, we have decided that an opt-out config is not necessary. Emit-on-change as a whole is intended solely as a performance optimization when it comes to Kafka Streams. The user should have little reason not to want this change, particularly since this KIP should not have a severe impact on performance (it is possible in implementation that we will only check if two values are equal in KTables where prior results has already been loaded).

Emit-on-change only for Suppress Operator

Also, we have decided that we should not restrict only dropping idempotent updates to the suppress operator. It is better to allow this behavior across most KTable API, rather than just one. This provides both flexibility for the user as well as more granular control over which idempotent updates we can drop and not drop.

Secondary Approaches

There is a possibility where we can support emit-on-change for all operations.

There is more than one way to yield the prior result. After all, we can obtain it from an upstream processor. For most operations, we can forward downstream both the old and new results of the upstream processor. In this case, the same operation will be performed twice. However, each operation can be very expensive. Performing it twice will in other words has the potential to incur horrendous performance hits. It might be that this is not a serious issue, but it is of significant concern.

The main bottleneck for emit-on-change for stateless operations is really how to load the prior result. That does not necessarily have to be done. Earlier in our discussion, we have talked about using a hash code as a way of uniquely identifying the results, and then comparing those hash codes. But as noted, hash codes can vary across JVMs, and it is not a requirement that the programs on different runs return the same hash code for the exact same object. Instead, we can consider using some method distinct from `Object#hashCode()`. There is the possibility here that we can add a configuration for allowing emit-on-change for stateless operations. If emit-on-change is enabled, then we can use some method defined by the user i.e. `generateUniqueObjectId (V result)` returning a 32-bit or 64-bit integer as an id – this method which will have stricter constraints than a normal hash code function. These generated ids would be used as is the hash codes described in the Implementation section below. We store these ids instead, and compare these for equality.

This potentially can work, but the user must implement the provided method correctly. This must be stressed in further documentation.

Forwarding the Old Result With the New

There had been some considerations, that for stateless operations, we would forward the old result along with the new one. Then, the stateless operation would be performed twice, once on the old, and once on the new. We compare these two new results to check if they are equal. The drawback with this approach is that stateless operations cannot be assumed to be inexpensive. Performing these operations twice would not work (especially since we are working to optimize Kafka Streams).