KIP-904: Kafka Streams - Guarantee subtractor is called before adder if key has not changed

Status

Current state: Accepted

Discussion thread: here

Vote thread: here

JIRA: KAFKA-12446

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

Motivation

During `KTable.groupBy`, we write events into an internal repartition topic. Since users can potentially change the grouping key inside `groupBy` , we send separate events for the oldKey-oldValue and the newKey-newValue to downstream processor nodes, where they will be subtracted/added from the corresponding aggregate associated with the old/new key respectively.

However, sending the oldKey-oldValue and the newKey-newValue as separate events is not strictly necessary when the grouping key does not change since both events will end up going to the same downstream processor node. In fact, doing so creates two challenges for users:

1. Firstly, the resulting KTable (i.e. the result of `KTable.groupBy(???).aggregate(???)`) can briefly be in an “inconsistent” state where the oldValue has been “subtracted” from the aggregate for the key but the newValue has not yet been “added” to the aggregate of the key because each event (oldValue, newValue) is processed separately.

   This transient inconsistency can be problematic in many situations e.g. if we are joining a KStream with a KTable. It is entirely possible that a KStream record could join with the KTable in a state where the subtractor has been executed but the adder has not yet been executed for a KTable update event. The consequences of this could be significant depending on the use case.

   Users can get around this issue today by dropping down to the Processor API level, but it would be better if users didn't have to do this just to handle this edge-case. Ideally, the Kafka Streams DSL should handle this edge-case.

2. Secondly, if users fail to configure their producers correctly to avoid reordering during `send()` , it’s entirely possible the newValue may be sent (and added to the aggregate) before the oldValue is sent (and subtracted from the aggregate). This means that if the user’s `adder` and `subtractor` functions are non-commutative, the resulting aggregate is in a permanently “inconsistent” state.

   An example of a useful non-commutative operation would be aggregating records into a `Set` :

   ```
   .aggregate(
     initializer = Set.empty[Animal],
     adder = (zooKey, animalValue, setOfAnimals) -> setOfAnimals + animalValue,
     subtractor = (zooKey, animalValue, setOfAnimals) -> setOfAnimals - animalValue
   )
   ```

   Consider the situation where this operator receives the following sequence of events:

   ```
   new KeyValueTimestamp<>("zoo1", "tiger", 8L)
   new KeyValueTimestamp<>("zoo1", "tiger", 9L)
   ```

The first event would trigger the initializer followed by the adder function, resulting in the aggregate for key="zoo1" to be `Set("tiger")` . So far so good. The second event would trigger both the adder and the subtractor functions. The end-result here depends on the order in which the adder and the subtractor functions are executed. If the subtractor ends up being called before the adder, the resulting aggregate would remain unchanged `Set("tiger")` . However, if the adder is called before the subtractor you would end up with an empty `Set()` !

Although it can be easy to miss, this non-deterministic behaviour is actually known and documented in the Kafka docs:

When subsequent non-null values are received for a key (e.g., UPDATE), then (1) the subtractor is called with the old value as stored in the table and (2) the adder is called with the new value of the input record that was just received. The order of execution for the subtractor and adder is not defined.

Instead of leaving the order of execution entirely undefined, I think we can do better specifically in the case where the key has not changed. Having a defined order of execution for this case, irrespective of how a user has configured their producer settings, makes it easier for users to reason about the semantics of `KGroupedTable.aggregate` and encourages wider use of this method beyond just commutative adder /subtractor functions.

Proposed Changes

As we already mentioned, if the grouping key has not changed, the oldValue and newValue events are guaranteed to be sent to the same processor node. Instead of sending them as two separate events, we should combine them and send them as a single event to the relevant downstream node. The subtractor and adder functions can then be executed (in that order) and the KTable state can be updated in a single “atomic” operation. In this way, we are able to remove the possibility of a transient inconsistent state.

In addition, sending the oldValue and newValue in the same event ensures that they can’t be re-ordered relative to each other irrespective of how a user has configured the underlying producer, thus eliminating the possibility that the adder may be called before the subtractor, and therefore eliminating that source of inconsistency as well (in the case of non-commutative subtractor/adder functions).

The exact logic we need to implement then is:

- Detect if the grouping key has changed or not.
- If the grouping key has not changed, send only a single event containing both the oldValue and the newValue to the downstream node.
- If the grouping key has changed, continue with the old behavior i.e. send two events, one for the oldKey with the oldValue, and one for the newKey with the newValue.

It is **not** necessary to review this as part of the KIP process but for those who are interested, here is an early pull request with the proposed changes.

Affected Public Interfaces

KGroupedTable interface

As a result of the proposed change, all the methods exposed via the `KGroupedTable` interface (i.e. `aggregate` , `reduce` , and `count`) will no longer produce inconsistent states in cases where the grouping key has not changed, transient or otherwise, reflecting the improved semantics of a single "atomic" update. For example, consider the following simple topology:

```
StreamsBuilder builder = new StreamsBuilder();

Serde<String> stringSerde = Serdes.String();

builder
.table(input, Consumed.with(stringSerde, stringSerde))
// key is not changed
.groupBy(KeyValue::pair, Grouped.with(stringSerde, stringSerde))
.count()
.toStream()
.to(output);

Topology topology = builder.build();
```

If we were to run the following inputs through this topology:

```
new KeyValueTimestamp<>("1", "", 8L)
new KeyValueTimestamp<>("1", "", 9L)
```

We would see the following messages being written to the output topic in the current version of Kafka Streams:

```
new KeyValueTimestamp<>("1", 1L, 8)
new KeyValueTimestamp<>("1", 0L, 9) // transient state of KTable
new KeyValueTimestamp<>("1", 1L, 9)
```

In contrast, running it with the changes proposed in this KIP would yield the following messages in our output topic:

```
new KeyValueTimestamp<>("1", 1L, 8)
new KeyValueTimestamp<>("1", 1L, 9)
```

Notice how the intermediate `new KeyValueTimestamp<>("1", 0L, 9)` message is missing! This is because a single "atomic" update operation has occurred and so we no longer see this transient state.

`Change<T>` serialization format

An important part of the proposed logic is being able to send the old and new value for the same key as a single event. This can be accomplished by sending them as a single `Change<T>(T newValue, T oldValue)` instance. However, we will need to make changes to the `ChangedSerializer` and `ChangedDeserializer`classes which currently only support serializing/deserializing instances of `Change<T>(T newValue, T oldValue)` where either the oldValue **or** the newValue are present but not both. This unfortunately means that we need to evolve the serialization scheme, and while this is not considered a public interface, this does have significant implications for the migration plan and is the main reason for doing this KIP. Currently the serialization scheme is of the form:

```
{BYTE_ARRAY oldValue}{BYTE newOldFlag=0}
{BYTE_ARRAY newValue}{BYTE newOldFlag=1}
```

As you can see, there is no format where both the oldValue and the newValue are present. We can extend the serialization scheme to support this in a backwards compatible fashion as follows:

```
{BYTE_ARRAY oldValue}{BYTE newOldFlag=0}
{BYTE_ARRAY newValue}{BYTE newOldFlag=1}
{INT newDataLength}{BYTE_ARRAY newValue}{BYTE_ARRAY oldValue}{BYTE newOldFlag=2}
```

Making this change in a backwards compatible fashion is important to ensure a migration path for upgrades without any production outage (see the next section for details).

Users should implement `.equals` method for key

Another important part of the proposed logic is detecting if the key has changed. In order for us to be able to do this, we depend on a correct implementation of the `.equals` method on the key, which users will need to implement.

Note however that this is not a strict new requirement for users; if users fail to implement the `.equals` method, then they should *generally* get the old behaviour of sending the oldValue and the newValue as two separate messages to the repartition topic i.e. nothing breaks. In this way, our changes can be considered backwards compatible with existing code where the key type does not implement the `.equals` method. There is one edge-case where this does not hold true, as follows.

Since the default `.equals` implementation for an `Object` is by reference, if a user's `groupBy` returns the same reference for the key, then the oldKey and the newKey will naturally `.equals` each other. This will result in a single event being sent to the repartition topic. This change in behaviour should be considered a "bug-fix" rather than a "breaking change" as the semantics of the operation remain unchanged, the only thing that changes for users is they no longer see transient "inconsistent" states. In the worst case, users in this situation will need to update any strict tests that check specifically for the presence of transient "inconsistent" states.

Compatibility, Deprecation, and Migration Plan

Upgrading from any older version without any production downtime is possible but users will need to execute two rolling bounces. This is required to safely handle the underlying serialization format changes discussed in the previous section.

1. In the first rolling bounce, we replace the byte code (i.e. swap the jars), set the config `upgrade.from="older version"` (possible values are "0.10.0" - "3.4"), and then bounce each instance to upgrade it.

   The `upgrade.from="older version"` config will ensure we are still writing out only the old serialization format until all instances are on the new byte code, at which point we can be sure that all instances in the group will be able to successfully deserialize the new format if we were to start writing it.
2. The second rolling bounce is to simply remove the `upgrade.from="older version"` config and bounce each instance for it to begin writing in the new serialization format.

Alternatively a simpler, offline upgrade is also possible. If users are willing to accept a temporary production outage, then the upgrade can be done by:

1. Stopping all old application instances.
2. Starting the new application instances with the updated code i.e. the new jar file. The application should resume work without any problems.

Downgrading is only possible with two rolling bounces and requires special attention:

1. The first rolling bounce is to set the config `upgrade.from="older version we are rolling back to"` and then bounce each instance.

   It's important to wait in this state for a few minutes and make sure that the application has finished processing any "in-flight" messages (i.e. those written into any repartition topics) which will have been written in the new serialization format. Once we revert back to the previous version of the byte code in the next step, Kafka Streams will no longer know how to deserialize messages in the new serialization format and will throw an exception.
2. The second rolling bounce is to replace the byte code (i.e. swap the jars), remove the `upgrade.from` config, and bounce each instance.

Test Plan

The change will be covered with unit tests.

Rejected Alternatives

Do nothing i.e. preserve the current state of affairs.