KIP-695: Further Improve Kafka Streams Timestamp Synchronization

Status

Current state: Adopted

Discussion thread: https://lists.apache.org/thread.html/rfc80d0c7856d8758e4fe7fbfd0662d36e45dcfba4b1a8d0ee44d0a38%40%3Cdev.kafka.apache.org%3E

Vote thread: https://lists.apache.org/thread.html/r201c3bcc3ec048da044078ca231481bb223de31f6be15afe3162c5a8%40%3Cdev.kafka.apache.org%3E

JIRA: **KAFKA-10091** - Getting issue details... STATUS

POC: https://github.com/apache/kafka/pull/9616

Motivation

When Streams is processing a task with multiple inputs, each time it is ready to process a record, it has to choose which input to process next. It always takes from the input for which the next record has the least timestamp. The result of this is that Streams processes data in timestamp order. However, if the buffer for one of the inputs is empty, Streams doesn't know what timestamp the next record for that input will be.

Streams introduced a configuration "max.task.idle.ms" in KIP-353 to address this issue (https://cwiki.apache.org/confluence/display/KAFKA/KIP-353%3A+Improve+Kafka+Streams+Timestamp+Synchronization).

The config allows Streams to wait some amount of time for data to arrive on the empty input, so that it can make a timestamp-ordered decision about which input to pull from next.

However, this config can be hard to use reliably and efficiently, since what we're really waiting for is the next poll that *would* return data from the empty input's partition, and this guarantee is a function of the poll interval, the max poll interval, and the internal logic that governs when Streams will poll again.

In the situation of a task with two (or more) inputs, there are three use cases:

UC1: I do not care about selecting among the inputs in order. I never want to wait to fetch data from empty inputs as long as I have at least one non-empty input. I just want to process records from any input as soon as they are available.

UC2: I would like to try and select among the inputs in order, but if some of the partitions are empty *on the brokers* , I don't want to wait for the producers to send more data.

UC3: I would like to try and select among the inputs in order, but if some of the partitions are empty *on the brokers* , I would like to wait some configured amount of time.

Proposed Solution

Streams Config change

We propose to alter the max.task.idle.ms config to guarantee to wait for fetches from empty partitions with nonzero lag before actualizing the timeout. This preserves the meaning of the config (how long to wait for more data to arrive for empty partitions before proceeding), but changes its behavior. Rather than starting to count against the timeout as soon as we discover an empty local buffer, we will only count against the timeout when the lag is zero. If the lag is not known, then we have to wait until the metadata is updated, and if the lag is non-zero, then we wait to poll the data. Thus, the configuration controls how long to wait for new data to be produced on the topic, regardless of fetch dynamics between the consumer and the brokers.

With this proposal:

- UC2 can be satisfied with a configuration value of "max.task.idle.ms: 0", since it means they will do at least one extra fetch for their empty partitions, and if those partitions are still empty, they proceed right away.
- UC3 can be satisfied with any positive integer value for "max.task.idle.ms". They will wait the configured amount of time while also being sure they actually fetch from the brokers before enforcing the timeout.

Note that UC1 is not satisfiable with this alone, so we also propose to add an extra config value of "-1", which indicates that the task idling feature itself should be disabled. So:

- UC1 can be satisfied with the configuration value of "max.task.idle.ms: -1", which indicates that Streams will never wait to buffer extra data, but always choose from what it already has buffered locally.

Consumer change

To support the desired semantics, Streams needs a programmatic way to check its lag on specific partitions. All of the current mechanisms for checking lag or end offsets will perform a remote call to the brokers, which is unnecessary in this case. Introducing a remote call to check lag would severely impact throughput in the case where we find that our local buffer is empty because the lag is zero. Also, such a check would delay our next call to poll(), so it would also harm throughput in the case where the lag is nonzero.

Streams only needs to know the lag of specific currently assigned partitions, which is already locally known inside the Consumer's internal data structures. This is how the client-side lag metric is implemented. Therefore, we propose to expose this local lag metadata in a new Consumer API method, `currentLag()`, which would only return a value when the lag is known locally.

Public Interfaces Affected

- Change "max.task.idle.ms" to accept "-1" as a flag to disable task idling entirely
- Change the semantics of the default of "max.task.idle.ms" ("0") so that Streams will deterministically fetch all available partitions before deciding to proceed with enforced processing.
- Add `Consumer#currentLag(): OptionalLong`, which is set when the lag is known and unset when no lag is known for any reason.

Proposed Changes

We propose to alter the config value space of "max.task.idle.ms" in the following way:

- Current:
  - negative integer: disallowed
  - 0 (default): indicates that Streams will never wait to buffer empty partitions before choosing the next record by timestamp
  - positive integer: indicates the amount of wall-clock time on the client to wait for more data to arrive on empty partitions (but no guarantee that the client will actually fetch again before the timeout expires).
- Proposed:
  - negative integer other than -1: disallowed
  - -1: indicates that Streams will never wait to buffer empty partitions before choosing the next record by timestamp
  - 0 (default): indicates that Streams will not wait for more data to be produced to empty partitions (partitions that are not buffered locally and also have zero lag). Doesn't count the time Streams has to wait to actually fetch those partitions' lags.
  - positive integer: Indicates the amount of time Streams will wait for more data to be produced to empty partitions. Doesn't count the time Streams has to wait to actually fetch those partitions' lags.

We also propose the following addition to the Consumer interface:

```
/**
 * Get the consumer's current lag on the partition.
 * Returns an "empty" {@link OptionalLong} if the lag is not known,
 * for example if there is no position yet, or if the end offset is not known yet.
 *
 * <p>
 * This method uses locally cached metadata and never makes a remote call.
 *
 * @param topicPartition The partition to get the lag for.
 *
 * @return This {@code Consumer} instance's current lag for the given partition.
 *
 * @throws IllegalStateException if the {@code topicPartition} is not assigned
 **/
OptionalLong currentLag(TopicPartition topicPartition);
```

Compatibility, Deprecation, and Migration Plan

Streams Config Change

After this change, Streams will offer more intuitive join semantics by default at the expense of occasionally having to pause processing if we haven't gotten updated lag metadata from the brokers in a while. This is typically not expected, since the brokers send back as much metadata as they have available in each fetch response, so we do not anticipate a default performance degradation under normal circumstances.

ConsumerRecords#currentLag() addition

Since this is a new method, there are no backward compatibility concerns. In the future, we may wish to expose locally-cached metadata in a more general way, in which case, this method could be deprecated and removed.

Performance

As a pre-emptive verification on the performance implications of this change, I ran our internal benchmarks on my POC branch and found that the measured throughput across all operations is within the 99% confidence interval of the baseline performance of trunk. I also deployed our internal soak test from my POC branch, which includes a join operation, and I observe that the throughput of that soak cluster is identical to the soak for trunk.

This result is to be expected, since the semantics improved here would only kick in for Join/Merge operations where Streams is processing faster than it can fetch some partitions on average. I would expect Streams to catch up to the fetches occasionally, but not on average.

It’s also worth noting that we have seen increasing numbers of users complaining of incorrect join results due to the current implementation. Even if the new implementation showed a modest drop in performance, I would advocate for correct results over top performance by default.

As a final safeguard, I'd note that the configuration value "-1" completely opts out of the new behavior and should avoid any potential performance drawbacks.

Rejected Alternatives

Streams Config Change

We could instead propose a new config with the desired semantics and deprecate the existing one. We judged it to be simpler for both users and maintainers to keep the config the same, since the semantics are so close to the existing ones. We are taking the perspective that the current "semantics" are really just a bugged implementation of the semantics we desired to provide in the first place.

Exposing fetched partitions in ConsumerRecords

We considered simply exposing which partitions have been fetched within the poll in ConsumerRecords. Either by adding empty-fetched partitions in `partitions()`, or by adding a new method to expose the partitions that had been fetched. The theory was that when the consumer gets an empty fetch response back from the brokers, it means that the partition in question has no more records to fetch. However, it turns out that the broker may also return empty responses for other reasons, such as:

- Quota enforcement
- Request/response size limits
- etc. Essentially, it's within the broker's contract that it can return no data for any reason.

Adding metadata to Consumer#poll response

We initially proposed to add metadata to the poll response object, ConsumerRecords and return early from the poll if we get a metadata-only response. The latter part of the proposal was a behavior change that caused system test failures, since the system tests rely on an assumption that when poll returns, either it would contain a record or the test has already timed out. That failure brought this proposal to the attention of more people and raised new concerns about the safety of this change. Most notably, it was pointed out that a user might manually assign a partition and then call `poll(Long. MAX_VALUE)`, which they would reasonably expect should *only* return with records available. This objection resulted in us altering the proposal to its current form, introducing the `currentLag` API instead.

Archived proposal

While analyzing possible implementations, we have determined that the current Consumer API does not offer a deterministic way to know whether a specific partition has been fetched. Right now, callers can only observe when a poll returns some data for a partition. If poll does not return data from a partition, it might have fetched and received no records, or it might not have fetched (yet). Therefore, we also prose to alter `Consumer#poll` to supply this information.

In the current API, `Consumer#poll` returns a `ConsumerRecords` instance, which is an `Iterable` collection of `ConsumerRecord` instances, and which also offers a `ConsumerRecords#partitions` method to list the partitions that are represented in the results.

Internally, when the consumer handles fetch responses from the brokers, it also receives metadata about the end offset of the partition(s), which it uses to update its metrics, including specifically a metric indicating the current lag. Theoretically, this could supply the information we need, except that we don't know how old those metrics are. To provide the desired semantics, we would like to use the lag information only if it is fresh, but the existing metrics may be arbitrarily stale.

To overcome these challenges, we propose to expose this fetched metadata in a new method on ConsumerRecords: `ConsumerRecords#metadata(): Map<TopicPartition, Metadata>`, where `Metadata` includes the `receivedTime`, `position`, `lag`, `beginningOffset`, and `endOffset` information.

Streams will be able to use this new method by maintaining internal flags of which partitions have been fetched, what the lag was at each fetch, and when the fetches took place. Essentially, each time we get a response back from poll(), we will persist the receivedTimestamp and lag for each partition. Then, when it comes time to decide whether we are ready to process, if we don't have data buffered for a partition, then we can consult this local metadata to see what the lag is. If the lag is missing, we would wait to fetch the metadata. If the lag is greater than zero, then we know there is data on the brokers, and we should wait to poll it before processing anything. If the lag is zero, then we can apply the config to idle the desired amount of time before enforcing processing.

We propose the following additions to the ConsumerRecords interface:

```
public class ConsumerRecords<K, V> implements Iterable<ConsumerRecord<K, V>> {

    ...

    public static final class Metadata {
        /**
         * @return The timestamp of the broker response that contained this metadata
         */
        public long receivedTimestamp()

        /**
         * @return The next position the consumer will fetch
         */
        public long position()

        /**
         * @return The lag between the next position to fetch and the current end of the partition
         */
        public long lag()

        /**
         * @return The current first offset in the partition.
         */
        public long beginningOffset()

        /**
         * @return The current last offset in the partition. The determination of the "last" offset
         * depends on the Consumer's isolation level. Under "read_uncommitted," this is the last successfully
         * replicated offset plus one. Under "read_committed," this is the minimum of the last successfully
         * replicated offset plus one or the smallest offset of any open transaction.
         */
        public long endOffset()
    }

    ...

    /**
     * Get the updated metadata returned by the brokers along with this record set.
     * May be empty or partial depending on the responses from the broker during this particular poll.
     * May also include metadata for additional partitions than the ones for which there are records in this
{@code ConsumerRecords} object.
     */
    public Map<TopicPartition, Metadata> metadata()

    ...

}
```