KIP-812: Introduce another form of the `KafkaStreams.close()` API that forces the member to leave the consumer group

Status

Current state: Adopted (3.3.0)

Discussion thread: here

Vote thread: here

JIRA: here

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

Motivation

(Sophie Blee-Goldman wrote most of this in the issue description. And as that is rich enough, I just copied parts of that)

Context

In Kafka Streams, when an instance is shut down via the close() API, we intentionally skip sending a LeaveGroup request. We decided to do so because the shutdown is often not due to a scaling-down event but some temporary closure, such as during a rolling bounce. In cases where the instance is expected to start up again shortly after, we originally wanted to avoid that member's tasks from being redistributed across the remaining group members since this would disturb the stable assignment and could cause unnecessary state migration and restoration. We also hoped to limit the disruption to just a single rebalance, rather than forcing the group to rebalance once the member shuts down and then again when it comes back up. So it's an optimization for the case in which the shutdown is temporary.

Problem

Above optimization makes sense for the cases of temporary closure.

But as this optimization is applied to all other cases too, when we want to permanently close a `KafkaStreams` instance, we have no handy way to make the member immediately leave the consumer group. We have to wait for the `session.timeout` mechanism to force the member to leave the consumer group.

This situation is more critical given the recent increase in default `session.timeout` to 45s, since that's a long time to go without noticing that a consumer has indeed permanently left the group.

Public Interfaces

```
package org.apache.kafka.streams;

public class KafkaStreams implements AutoCloseable {
    public void close()  // Already exist
    private boolean close(final long timeoutMs)  // Already exist
    public synchronized boolean close(final Duration timeout) throws IllegalArgumentException  // Already exist
    public synchronized boolean close(CloseOptions options) throws IllegalArgumentException  // *This one will
be added

         public static class CloseOptions {
                private Duration timeout = Duration.ofMillis(Long.MAX_VALUE);
            private boolean leaveGroup = false;

             public CloseOptions timeout(Duration timeout)
            public CloseOptions leaveGroup(boolean leaveGroup)
        }
}
```

Proposed Changes

We introduce another form of the `KafkaStreams.close()` method that forces the member to leave the consumer group, to be used in event of actual scale down rather than a transient bounce.

Compatibility, Deprecation, and Migration Plan

The proposal is backward-compatible because it only adds new method and does not change any existing methods.

This would be considered as an optimization for some cases of `KafkaStreams` instance closure. And there will be no impact till users start using this new method to optimize their applications.

Rejected Alternatives

There was another option to achieve the same purpose: letting a member leave the consumer group in every case, including quick bounce. With this approach, there will be unexpected side effects which guozhang Wang figures out. In short, if we make the bouncing member leave the consumer group, we have to move the tasks on that shutdown instance to others immediately, which would start restoring the states. So it's still valuable to let the bouncing member temporarily be closed without leaving the group so that we can rebalance the tasks only after the instance comes back and get the tasks back to the restarted instances and hence no task migration.

> You can find more details on the original comment.

I think the validity of this option depends on how light the consumer group rebalancing logic is.

R1: The reasons why we can say the rebalance is light:

- We've introduced light rebalancing logics, including "The incremental cooperative rebalancing protocol."
  - https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/#incremental-cooperative-rebalancing-protocol

R2: The reasons why we can say the rebalance can be heavy:

- We give users full permission how the consumer group rebalances tasks. With this decision, even though we've introduced new styles of lightweight rebalancing logic, there could be a heavy rebalance logic adopted by users at any chance.
  - https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/#consumer-groups-and-rebalance-protocol

It seems R2 can be applied more commonly. So I reject this option for now.