KIP-54 - Sticky Partition Assignment Strategy

Status

Current state: Accepted

Discussion thread: link

JIRA: KAFKA-2273

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

⚠ Changes introduced in this KIP could potentially be superseded by changes to consumer rebalancing protocol.

⚠ There was a bug in the partition assignment algorithm as described in this KIP that was fixed in KIP-341.

Motivation

In certain circumstances the round robin assignor, which produces better assignments compared to range assignor, fails to produce an optimal and balanced assignment of topic partitions to consumers. KIP-49 touches on some of these circumstances. In addition, when a reassignment occurs, none of the existing strategies consider what topic partition assignments were before reassignment, as if they are about to perform a fresh assignment. Preserving the existing assignments could reduce some of the overheads of a reassignment. For example, Kafka consumers retain pre-fetched messages for partitions assigned to them before a reassignment. Therefore, preserving the partition assignment could save on the number of messages delivered to consumers after a reassignment. Another advantage would be reducing the need to cleanup local partition state between rebalances.

Public Interfaces

This assignment strategy, which is implemented for the new consumer, would add a *StickyAssignor* class that can be used as *org.apache.kafka.clients.consumer.StickyAssignor* for the value of the consumer property *partition.assignment.strategy*. It would not affect the default value of this consumer property.

Proposed Changes

Add a Sticky Assignor option to the potential assignment strategies of the new consumer. The Sticky Assignor serves two purposes.

First, it guarantees an assignment that is as balanced as possible, meaning either:

- the numbers of topic partitions assigned to consumers differ by at most one; or
- if a consumer A has 2+ fewer topic partitions assigned to it compared to another consumer B, none of the topic partitions assigned to A can be assigned to B.

When starting a fresh assignment, the Sticky Assignor would distribute the partitions over consumers as evenly as possible. Even though this may sound similar to how round robin assignor works, the second example below shows that it results in a more balanced assignment.

Second, during a reassignment the Sticky Assignor would perform the reassignment in such a way that in the new assignment,

1. topic partitions are still distributed as evenly as possible, and
2. topic partitions stay with their previously assigned consumers as much as possible.

Of course, the first goal above takes precedence over the second one. This means it is possible that a few topic partitions cannot remain assigned to the same consumer and have to switch to another consumer in order to guarantee the most balanced assignment possible.

With the Sticky Assignor, the reassignment is performed by

1. preserving all the existing partition assignments
2. removing all the partition assignments that have become invalid due to the change that triggers the reassignment
3. assigning the unassigned partitions in a way that balances out the overall assignments of partitions to consumers
4. further balancing out the resulting assignment by finding the partitions that can be reassigned to another consumer towards an overall more balanced assignment.

Upon each partition assignment calculation, the full partition assignments are preserved for each consumer (as user data). The embedded protocol used by consumer groups opting the sticky partition assignment would be

```
ProtocolName => "sticky"

ProtocolMetadata => Version Subscription UserData
  Version => int16
  Subscription => [Topic]
    Topic => string
  UserData => CurrentAssignments
    CurrentAssignments => [Topic [Partition]]
           Topic => string
                Partition => int32
```

This is specially helpful when a consumer group leader, who is in charge of performing the partition assignment, dies and the leadership has to be given to some other group member. In such circumstances, the new leader has access to the most recent partition assignment and can easily take over the rebalance that is triggered when the former leader disappears, as well as future rebalances.

Example 1

Suppose there are three consumers C_0, C_1, C_2, four topics t_0, t_1, t_2, t_3, and each topic has 2 partitions, resulting in partitions t_0p_0, t_0p_1, t_1p_0, t_1p_1, t_2p_0, t_2p_1, t_3p_0, t_3p_1. Each consumer is subscribed to all four topics.

The assignment with both sticky and round robin assignors results in

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_1, t_3p_0
C_1	t_0p_1, t_2p_0, t_3p_1
C_2	t_1p_0, t_2p_1

Now, let's assume that consumer C_1 is removed and a reassignment occurs. The round robin assignor would produce

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_0, t_2p_0, t_3p_0
C_2	t_0p_1, t_1p_1, t_2p_1, t_3p_1

The sticky assignor, on the other hand, would result in

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_1, t_3p_0, t_2p_0
C_2	t_1p_0, t_2p_1, t_0p_1, t_3p_1

preserving 5 of the previous assignments (unlike the round robin assignor which preserves only 3).

Example 2

There are three consumers C_0, C_1, C_2, and three topics t_0, t_1, t_2, with 1, 2, and 3 partitions, respectively. Therefore, the partitions are t_0p_0, t_1p_0, t_1p_1, t_2p_0, t_2p_1, t_2p_2. C_0 is subscribed to t_0; C_1 is subscribed to t_0, t_1; and C2 is subscribed to t_0, t_1, t_2.

The round robin assignor would result in the following assignment:

Consumer	Assigned Topic Partitions
C_0	t_0p_0
C_1	t_1p_0
C_2	t_1p_1, t_2p_0, t_2p_1, t_2p_2

which is not as balanced as the assignment produced by the sticky assignor:

Consumer	Assigned Topic Partitions
C_0	t_0p_0
C_1	t_1p_0, t_1p_1
C_2	t_2p_0, t_2p_1, t_2p_2

Now, if consumer C_0 is removed, these two assignors would produce the following assignments.

Round Robin (preserves 3 partition assignments):

Consumer	Assigned Topic Partitions
C_1	t_0p_0, t_1p_1
C_2	t_1p_0, t_2p_0, t_2p_1, t_2p_2

Sticky (preserves 5 partition assignments):

Consumer	Assigned Topic Partitions
C_1	t_1p_0, t_1p_1, t_0p_0
C_2	t_2p_0, t_2p_1, t_2p_2

Not only the sticky assignor preserves more assignments, it also results in a more balanced assignment.

Example 3

There are two consumers C_0, C_1, and two topics t_0, t_1, with 2 partitions each. Therefore, the partitions are t_0p_0, t_0p_1,t_1p_0, t_1p_1. Both consumers are subscribed to both topics.

The range, round robin, and sticky assignors all result in the following assignment:

Consumer	Assigned Topic Partitions

C_0	t_0p_0, t_1p_0
C_1	t_0p_1, t_1p_1

Now, if consumer C_2 subscribed to both topics comes on board, the range assignor would produce the following (which is not a fair assignment):

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_0
C_1	t_0p_1, t_1p_1
C_2	

The round robin assignor would produce this (which is not the stickiest assignment):

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_1
C_1	t_0p_1
C_2	t_1p_0

The sticky assignor would move one of the partitions to the new consumer producing something like this (preserving 3 partition assignments):

Consumer	Assigned Topic Partitions
C_0	t_0p_0, t_1p_0
C_1	t_0p_1
C_2	t_1p_1

The Algorithm

The inputs to the sticky partition assignment algorithm are

- *partitionsPerTopic*: topics mapped to number of their partitions
- *subscriptions*: mapping of consumer to subscribed topics
- *currentAssignment*: preserved assignment of topic partitions to consumers calculated during the previous rebalance

The sticky partition assignment algorithm works by defining and maintaining a number of data structures

- potential consumers of each topic partition (*partition2AllPotentialConsumers*)
- topic partitions each consumer can consume from (*consumer2AllPotentialPartitions*)
- the current consumer of each topic partition (*currentPartitionConsumer*)
- sorted list of topic partitions (in ascending order) by how many consumers can consumer from them (*sortedAllPartitions*)
- partitions that don't have a consumer (either because they are just created, or because their consumer no longer exists) and need new assignment (*unassignedPartitions*)
- sorted list of consumers (in ascending order) based on how many partitions are currently assigned to them (*sortedCurrentSubscriptions*)
- re-assignable partitions, or those with 2 or more potential consumers (*reassignablePartitions*)
- partitions that cannot be assigned to another consumer (*fixedPartitions*)
- current assignment of (fixed) consumers whose assignment cannot change (*fixedAssignments*)

The algorithm includes these main steps

1. Update *currentAssignment* by removing consumers or partitions that no longer exist.

2. Distribute *unassignedPartitions* among existing consumers: For each unassigned partition start from the consumer with fewest number of assignments and if the consumer can consume from this partition assign the partition to the consumer and update the necessary data structures. After this distribution all partitions that can be assigned to some consumer are, in fact, assigned. There maybe some that have no subscriber and remain unassigned (which is fine).
3. Find consumers whose topic partition assignment cannot change and remove them from *currentAssignment* (to limit the scope of the problem).
4. From partitions that can potentially be reassigned (*sortedAllPartitions)* start from one with fewest possible consumers and if it can go to another consumer to improve balance move it to that other consumer. Continue until all partitions are gone through this reassignment or we reach a balance (defined below).
5. If reassignments occurred in previous step and we don't reach a better overall balance (for example, we get an equally balanced assignment) we revert the assignment in favor of stickiness since we could not improve fairness.
6. Re-add fixed assignments to currentAssignment.

We call a consumer partition assignment *balanced* when

- The minimum and maximum number of partitions assigned to any consumer differs by at most one; or
- No topic partition can be moved from its current consumer to another of its potential consumers and improve the overall *balance score* at the same time.
  The overall *balance score* of a topic partition assignment is defined as the sum of assigned partitions size difference of all consumer pairs. A perfectly balanced assignment (with all consumers getting the same number of partitions) has a balance score of 0. Lower balance score indicates a more balanced assignment.

Note that due to the complex nature of the problem and the heuristics (e.g. steps 2 and 4) applied in the algorithm above we expect to find an optimum (and not necessarily the best) sticky partition assignment.

Alternatives Yet to be Considered

- The sticky partition assignment algorithm described above, as mentioned earlier in this KIP, favors fairness over stickiness (we may call it the *fair yet sticky* or *stickiest fair* strategy). Therefore, some partitions may change their consumer towards a fair assignment. This strategy is supposedly more complex to implement due to complex nature of calculating the most balanced assignment.
  An alternative strategy is to implement it so that stickiness take precedence and all previously assigned partitions are preserved; while the new consumers or partitions that need to be assigned do so towards the fairest possible assignment. We may call this alternative the *sticky yet fair* or *f airest sticky* strategy. This strategy would be more efficient as it exhibits less complexity and variance. After all, the previous valid assignments remain as they are and only a few new assignments have to be calculated.
  The fair strategy discussed in KIP-49 considers fairness only and does not take into consideration the partition assignments before the rebalance. Fairness only can also be supported as part of this KIP by switching off stickiness and always starting from an empty *currentAssignment*.

Notes

- KAFKA-2019: This JIRA describes an issue with the round robin assignor of the old consumer, which stems from multiple threads that each (old) consumer may have and how the threads of each consumer are assigned first before assigning partitions to other consumer threads. Since the new consumer is single threaded there is no such problem in its round robin strategy. It simply considers consumers one by one for each partition assignment, and when one consumer is assigned a partition, the next assignment starts by considering the next consumer in the list (and not from the start of the list again not from the same consumer that was just assigned). This removes the possibility of the issue reported in KAFKA-2019 surfacing in the new consumer's round robin assignor. In the sticky strategy this issue does not exist either, since every time an assignment is about to happen it starts with the consumer with least number of assignments. So there is no scenario in which a consumer is repeatedly assigned partitions as in KAFKA-2019 (unless that consumer is lagging behind other consumers on the number of partitions assigned).

Compatibility, Deprecation, and Migration Plan

This proposal would add a new option to existing assignment strategies of the new consumer. The only impact is to those new consumers that have some partition cleanup code in their `onPartitionsRevoked()` callback listeners. That cleanup code is rightfully placed in that callback listener because the consumer has no assumption or hope of preserving any of its assigned partitions after a rebalance when it is using range or round robin assignor. The listener code would look like this:

```
class MyOldRebalanceListener {

  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (TopicPartition partition: partitions) {
      commitOffsets(partition);
      cleanupState(partition);
    }
  }

  void onPartitionsAssigned(Collection<TopicPartition> partitions) {
```

```
    for (TopicPartition partition: partitions) {
      initializeState(partition);
      initializeOffset(partition);
    }
  }
}
```

One of the advantages of the sticky assignor is that, in general, it reduces the number of partitions that are actually de-assigned from a consumer. Because of that, consumers now can do their cleanup more efficiently. Of course, they still can perform the partition cleanup in the `onPartitionsRevoked()` listener, but they can be more efficient and make a note of their partitions before and after the rebalance, and do the cleanup after the rebalance on the actual partitions they lost (which is normally not a whole lot). The code snippet below clarifies this point.

```
class MyNewRebalanceListener {
  Collection<TopicPartition> lastAssignment = Collections.emptyList();

  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (TopicPartition partition: partitions)
      commitOffsets(partition);
  }

  void onPartitionsAssigned(Collection<TopicPartition> assignment) {
    for (TopicPartition partition: difference(lastAssignment, assignment)
      cleanupState(partition);

    for (TopicPartition partition: difference(assignment, lastAssignment)
      initializeState(partition);

    for (TopicPartition partition: assignment)
      initializeOffset(partition);

    this.lastAssignment = assignment;
  }
}
```

Looking back at Example 2, when Consumer C_0 is removed a rebalance occurs and with the round robin assignment the remaining consumers would perform partition clean up on all of their partitions before the rebalance (a total of 5). With the sticky assignor and the change proposed above, since both C_1 and C_2 preserve all their assigned partitions there would be no need to do any cleanup.

Rejected Alternatives

- Having the consumer group leader store the calculated topic partition assignment in an internal topic for other consumers to retrieve in case of a leadership change. It was decided that passing the calculated assignments as user data to all consumers after each rebalance is a more viable option.