KIP-480: Sticky Partitioner

Status

Current state: Accepted *(vote thread)*

Discussion thread: *here*

JIRA: *here*

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

Motivation

The size of record batches from the producer to the broker plays a role in latency. Smaller batches lead to more requests and queuing as well as higher latency. This means in general, even when linger.ms is set to zero, a larger batch size will decrease latency. In the case where linger.ms is turned on, low throughput often injects latency into the system because if there are not enough records to fill a batch, the batch will not be sent until linger.ms. Finding a way to increase the size of batches to trigger a send before linger.ms will decrease latency even more.

Currently, in the case where no partition and no key is specified, the default partitioner partitions records in a round-robin fashion. That means that each record in a series of consecutive records will be sent to a different partition until we run out of partitions and start over again. While this spreads records out evenly among the partitions, it also results in more batches that are smaller in size. It seems like it would be better to have all the records go to a specified partition (or a few partitions) and be sent together in a larger batch.

The sticky partitioner attempts to create this behavior in the partitioner. By “sticking” to a partition until the batch is full (or is sent when linger.ms is up), we can create larger batches and reduce latency in the system compared to the default partitioner. Even in the case where linger.ms is 0 and we send right away, we see improved batching and a decrease in latency. After sending a batch, the partition that is sticky changes. Over time, the records should be spread out evenly among all the partitions.

Netflix had a similar idea and created a sticky partitioner that selects a partition and sends all records to it for a given period of time before switching to a new partition.

A different approach is to change the sticky partition when a new batch is created. The idea is to minimize mostly empty batches that might occur on an untimely partition switch. This approach is used in the sticky partitioner presented here.

Public Interfaces

The sticky partitioner will be part of the default partitioner, so there will not be a public interface directly.

There is one new method exposed on the partitioner interface.

```
  /**
  * Executes right before a new batch will be created. For example, if a sticky partitioner is used,
  * this method can change the chosen sticky partition for the new batch.
  * @param topic The topic name
  * @param cluster The current cluster metadata
  * @param prevPartition The partition of the batch that was just completed
  */
default public void onNewBatch(String topic, Cluster cluster, int prevPartition) {
}
```

The method onNewBatch will execute code right before a new batch is created. The sticky partitioner will define this method to update the sticky partition. This includes changing the sticky partition even when there will be a new batch on a keyed value. Test results show that this change will not significantly affect latency in the keyed value case.

The default of this method will result in no change to the current partitioning behavior for other user-defined partitioners. If a user wants to implement a sticky partitioner in their own partitioner class, this method can be overridden.

Proposed Changes

Change the behavior of the default partitioner in the no explicit partition, key = null case. Choose the “sticky” partition for the given topic. The “sticky” partition is changed when the record accumulator is allocating a new batch for a topic on a given partition.

These changes will slightly modify the code path for records that have keys as well, but the changes will not affect latency significantly.

A new partitioner called UniformStickyPartitioner will be created to allow sticky partitioning on all records, even those that have non-null keys. This will mirror how the RoundRobinPartitioner uses the round robin partitioning strategy for all records, including those with keys.

Compatibility, Deprecation, and Migration Plan

- No compatibility, deprecation, migration plan required.
- Users can continue to use their own partitioners--if they want to implement a sticky partitioner, they can use the onNewBatch(String topic, Cluster cluster) method to implement the feature, if they don’t want to use the feature, behavior will be the same.
- Existing users of the default partitioner for non-keyed, not set partitioned values should see either the same or decreased latency and cpu usage

Test Results

Testing the performance for this partitioner was done through the trogdor ProduceBench test. Modifications were made to allow for non-keyed, non-partitioned values as well as the ability to prevent batches from being flushed by the throttle. Besides waiting on linger.ms and filling the batch, a batch can be sent through a flush.Tests were done with and without flushing.

The tests were done using Castle on AWS m3.xlarge instances with SSD. Latency is measured as time to producer ack.

Tests run with these specs unless otherwise specified:

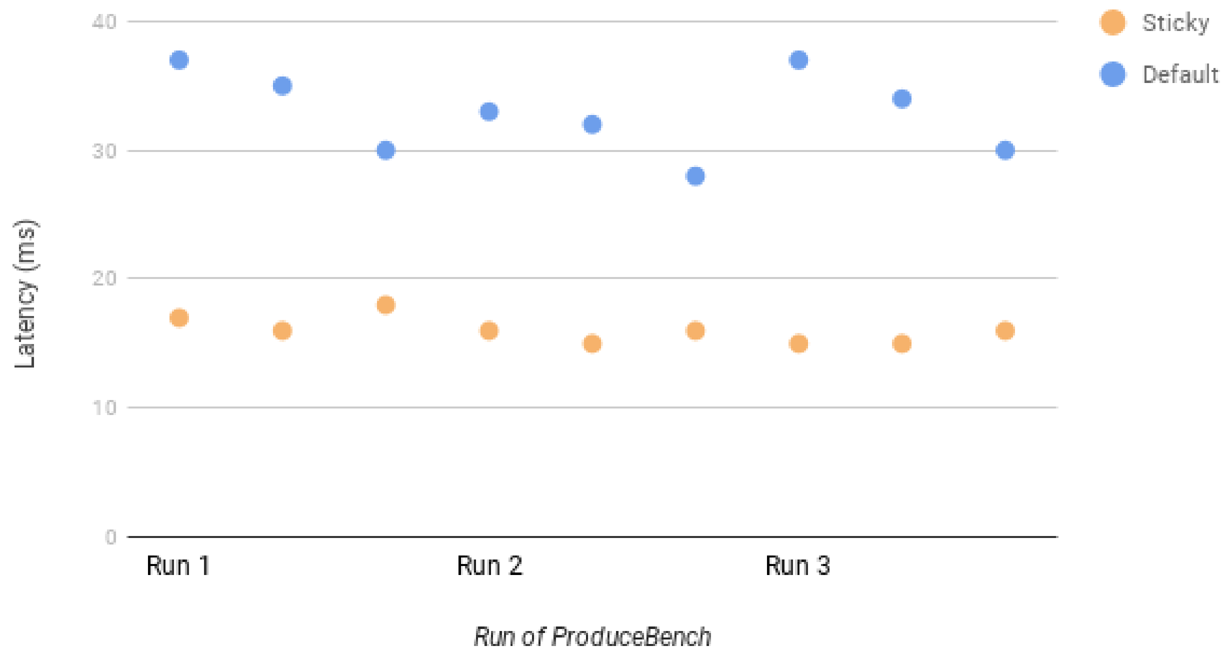
EC2 Instance	m3.xlarge
Disk Type	SSD
Duration of Test	12 minutes
Number of Brokers	3
Number of Producers	3
Replication Factor	3
Active Topics	4
Inactive Topics	1
Linger.ms	0
Acks	All
keyGenerator	{"type": "null"}
useConfiguredPartitioner	True
No Flushing on Throttle (skipFlush)	True

For more details, there is an example spec here.

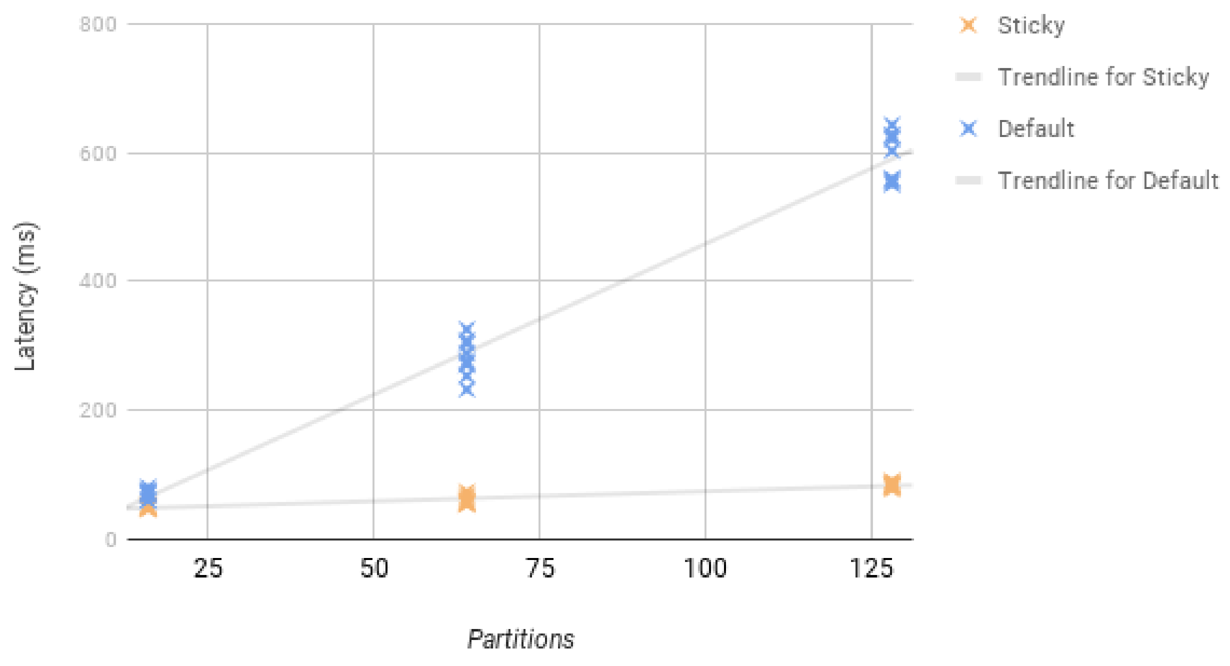
In general, this partitioner often saw latency cut in half compared to the current code. In the worst case, the sticky partitioner performed on-par with the default code.

One trend was to see more benefit as partitions increased. Still, with 16 partitions, there was a clear benefit observed. On throughput of 1000 msg/sec, the latency was still about half of the default.

p99 Latency: 3 Producers, 1,000 msg/sec, 16 partitions (No Flush)



p99 Latency: 3 Producers, 10,000 msg/sec (No Flush)

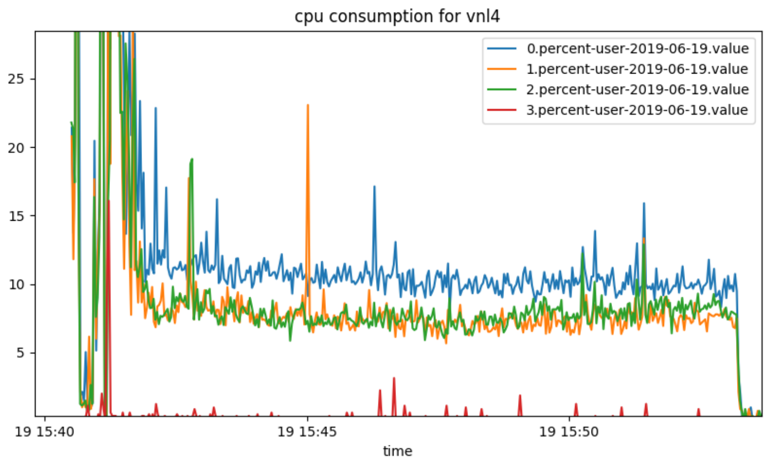


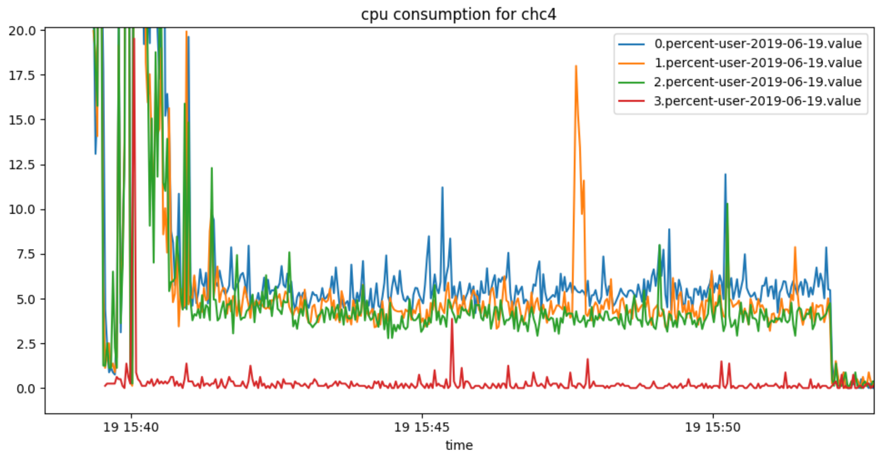
Another trend observed, especially in the flushing case, was latency to decrease more as the number of messages sent increased from low to medium throughput. The benefit relies in part on the ratio of messages per second to partitions.

Finally, there was a clear benefit in the case where linger.ms was not zero and throughput was low enough for the default code to need to wait on linger.ms. For example, a run with 1 producer, 16 partitions, and 1000 msg/sec, as well as linger.ms = 1000 saw the following a p99 latency of 204 for the sticky partitioner compared to 1017 for the default. This is approximately the latency and is due to batches not having to wait for linger.ms.

More performance tests comparing the sticky partitioner to the default partitioner can be found here.

Aside from latency, the sticky partitioner also sees decreased cpu utilization compared to the default code. In the cases observed, the sticky partition's nodes often saw up to 5-15% reduction in CPU utilization (for example from 9-17% to 5-12.5% or from 30-40% to 15-25%) compared to the default code's nodes.





(Vnl is the default case and Chc is the sticky partitioner. This is the result of a 1 producer, 16 partition, 10,000 msg/sec no flush case.)

More comparisons of CPU utilization can be found here.

Rejected Alternatives

Configurable sticky partitioner:

- Testing showed that the sticky partitioner performed just as well or better than the default in both cpu utilization and latency. Making the sticky partitioner a configurable feature means that some users may miss out on this beneficial feature

Partitions changes based on time

- Time before change would vary based on throughput, would need to be set for different circumstances, throughput may not be consistent