KIP-794: Strictly Uniform 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

KIP-480: Sticky Partitioner introduced a UniformStickyPartitioner and made it the default paritioner. It turned out that despite being called UniformStickyPartitioner, the sticky partitioner is not uniform in a problematic way: it actually distributes more records to slower brokers and can cause "runaway" problems, when a temporary slowness of a broker skews distribution such that the broker gets more records and becomes slower because of that, which in turn skews distribution even more, and the problem is perpetuated.

The problem happens because the "stickiness" time is driven by the new batch creation, which is reciprocal to broker latency - slower brokers drain batches slower, so they get more of the "sticky" time than faster brokers, thus skewing the distribution. The details of the scenario are described well here.

Suppose that we have a producer writing to 3 partitions with linger.ms=0 and one partition slows down a little bit for some reason. It could be a leader change or some transient network issue. The producer will have to hold onto the batches for that partition until it becomes available. While it is holding onto those batches, additional batches will begin piling up. Each of these batches is likely to get filled because the producer is not ready to send to this partition yet.

Consider this from the perspective of the sticky partitioner. Every time the slow partition gets selected, the producer will fill the batches completely. On the other hand, the remaining "fast" partitions will likely not get their batches filled because of the `linger.ms=0` setting. As soon as a single record is available, it might get sent. So more data ends up getting written to the partition that has already started to build a backlog. And even after the cause of the original slowness (e.g. leader change) gets resolved, it might take some time for this imbalance to recover. We believe this can even create a runaway effect if the partition cannot catch up with the handicap of the additional load.

We analyzed one case where we thought this might be going on. Below I've summarized the writes over a period of one hour to 3 partitions. Partition 0 here is the "slow" partition. All partitions get roughly the same number of batches, but the slow partition has much bigger batch sizes.

```
Partition TotalBatches TotalBytes TotalRecords BytesPerBatch RecordsPerBatch
0         1683         25953200   25228        15420.80      14.99
1         1713         7836878    4622         4574.94       2.70
2         1711         7546212    4381         4410.41       2.56
```

After restarting the application, the producer was healthy again. It just was not able to recover with the imbalanced workload.

This is not the only problem; even when all brokers are uniformly fast, with linger.ms=0 and many brokers, the sticky partitioner doesn't create batches as efficiently. Consider this scenario, say we have 30 partitions, each has a leader on its own broker.

1. a record is produced, partitioner assigns to partition1, batch becomes ready and sent out immediately

2. a record is produced, partitioner sees that a new batch is created, triggers reassignment, assigns to partition2, batch becomes ready and sent out immediately
3. a record is produced, partitioner sees that a new batch is created, triggers reassignment, assigns to partition3, batch becomes ready and sent out immediately

and so on. (The actual assignment is random, but on average we'd rotate over all partitions more or less uniformly.) Then it repeats the whole loop once again (the pattern will be the same because we allow 5 in-flight), and again, while it's doing that, the first on the first broker may complete, in which case, a single record batch may be ready again and so on. This is probably not that big of a deal when the number of brokers is small (or to be precise, the number of brokers that happen to host partitions from one topic), but it's good to understand the dynamics.

So in some sense, the UniformStickyPartitioner is neither uniform nor sufficiently sticky.

Public Interfaces

Changed Default Configuration

partitioner.class would have the default value **null**. When partitioner.class isn't explicitly to set to a custom partitioner class, the producer uses partitioning logic implemented in `KafkaProducer` . The `DefaultPartitioner` and `UniformStickyPartitioner` are deprecated, instead of setting partitioner.class=UniformStickyPartitioner, the partitioner.class shouldn't be set and **partitioner.ignore.keys** configuration should be set to 'true'.

New Configuration

partitioner.adaptive.partitioning.enable. The default would be 'true', if it's true then the producer will try to adapt to broker performance and produce more messages to partitions hosted on faster brokers. If it's 'false', then the producer will try to assign partitions randomly.

partitioner.availability.timeout.ms. The default would be 0. If the value is greater than 0 and adaptive partitioning is enabled, and the broker cannot accept a produce request to the partition for **partitioner.availability.timeout.ms** milliseconds, the partition is marked as not available. If the value is 0, this logic is disabled. This configuration is ignored if adaptive partitioning is disabled via setting **partitioner.adaptive.partitioning.enable** to 'false'.

partitioner.ignore.keys. The default would be 'false', if it's 'false' then the producer uses the message key (if specified) to pick a partition, if it's 'true' the producer doesn't use message keys to pick a partition, even if it's specified.

Note that the new configuration applies to the partitioning algorithm that is used when **partitioner.class** is not specified. They don't have any effect if a custom partitioner is used.

Proposed Changes

This is the default partitioning logic for messages without keys, that substitutes the corresponding logic implemented in the default partitioner. The default partitioning logic is going to be implemented in `KafkaProducer` itself. `KafkaProducer` would check if the **partitioner.class** is set to **null** and implement the default partitioning logic in that case. If **partitioner.ignore.keys** is set to 'true', then even messages that have keys would be uniformly distributed among partitions.

Uniform Sticky Batch Size

Instead of switching partitions on every batch creation, switch partitions every time **batch.size** bytes got produced to partition. Say we're producing to partition 1. After 16KB got produced to partition 1, we switch to partition 42. After 16KB got produced to partition 42, we switch to partition 3. And so on. We do it regardless of what happens with batching or etc. just count the bytes produced to a partition. This way the distribution would be both uniform (there could be small temporary imbalance) and sticky even if linger.ms=0 because more consecutive records are directed to a partition, allowing it to create better batches.

Let's consider how the batching is going to be different with a strictly uniform sticky partitioner and linger.ms=0 and 30 partitions each on its own broker.

1. a record is produced, partitioner assigns to partition1, batch becomes ready and sent out immediately
2. a record is produced, partitioner is still stuck to partition1, batch becomes ready and sent out immediately
3. same thing
4. --
5. --
6. a record is produced, partitioner is still stuck to partition1, now we have 5 in-flight, so batching begins

The batching will continue until either an in-flight batch completes or we hit the **batch.size** bytes and move to the next partition. This way it takes just 5 records to start batching. This happens because once we have 5 in-flight, the new batch won't be sent out immediately until at least on in-flight batch and keeps accumulating records. With the current solution, it takes 5 x number of partitions to have enough batches in-flight so that new batch won't be sent immediately. As the production rate accelerates, more records could be accumulated while 5 batches are already in-flight, thus larger batches are going to be used for higher production rates to sustain higher throughput.

If one of the brokers has higher latency the records for the partitions hosted on that broker are going to form larger batches, but it's still going to be the same amount records sent less frequently in larger batches, the logic automatically adapts to that.

To summarize, the uniform sticky partitioner has the following advantages:

1. It's uniform, which is simple to implement and easy to understand. Intuitively, this is what users expect.
2. It creates better batches, without adding linger latency on low production rate but switching to better batching on high production rate.

3. It adapts to higher latency brokers, using larger batches to push data, keeping throughput and data distribution uniform.
4. It's efficient (logic for selecting partitions doesn't require complex calculations).

From the implementation perspective, the partitioner doesn't have enough information to calculate the number of bytes in the record, namely:

- Headers
- Compression info
- Batch headers info (one partition could reside on a faster broker and get more smaller batches, we need to account for that to achieve uniformity)
- Record overhead (due do var int optimization). Not sure if it matters, but seems to be easier to implement than to prove that it doesn't.

This information can be easily collected in the `RecordAccumulator` object so the `KafkaProducer` class would calculate the partition information in the `RecordAccumulator` unless a custom partitioner class is explicitly set.

Adaptive Partition Switching

One potential disadvantage of strictly uniform partition switching is that if one of the brokers is lagging behind (cannot sustain its share of throughput), the records will keep piling in the accumulator, and will eventually exhaust the buffer pool memory and slow down the production rate to match the capacity of the slowest broker. To avoid this problem, the partition switch decision can adapt to broker load.

The queue size of batches waiting to be sent is a direct indication of broker load (more loaded brokers would have longer queue). Partition switching taking into account the queue sizes when choosing next partition. The probability of choosing a partition is proportional to the inverse of queue size (i.e. partitions with longer queues are less likely to be chosen).

In addition to queue size - based logic, **partitioner.availability.timeout.ms** can set to a non-0 value, in which case partitions that have batches ready to be sent for more than **partitioner.availability.timeout.ms** milliseconds, would be marked as not available for partitioning and would not be chosen until the broker is able to accept the next ready batch from the partition.

Adaptive partition switching can be turned off by setting **partitioner.adaptive.partitioning.enable** = false.

From the implementation perspective, the partitioner doesn't know anything about queue sizes or broker readiness (but that information can be gathered in the `RecordAccumulator` object), so so the `KafkaProducer` class would execute partitioning logic in the `RecordAccumulator` unless a custom partitioner is explicitly set.

Note that these changes do not affect partitioning for keyed messages, only partitioning for unkeyed messages.

Compatibility, Deprecation, and Migration Plan

- `DefaultPartitioner` and `UniformStickyPartitioner` will be deprecated, they'll behave the same way as they are today.
- Users that don't specify a custom partitioner would get the new behavior automatically.
- Users that explicitly specify `DefaultPartitioner` or `UniformStickyPartitioner` would get a deprecation warning, but see no change of behavior. They would need to update the configuration correspondingly to get the updated behavior (remove **partitioner.class** setting and optionally set **partitioner.ignore.keys** to 'true' if replacing `UniformStickyPartitioner`).
- `Partitioner.onNewBatch` will be deprecated.

Test Results

Test Setup

3 kafka brokers on a local machine, one topic with 3 partitions, RF=1, one partition on each broker:

```
    Topic: foo    Partition: 0    Leader: 1    Replicas: 1    Isr: 1    Offline:
    Topic: foo    Partition: 1    Leader: 0    Replicas: 0    Isr: 0    Offline:
    Topic: foo    Partition: 2    Leader: 2    Replicas: 2    Isr: 2    Offline:
```

Kafka-0 has 20ms sleep injected for each produce response.

The settings of Kafka producer are default other than the explicitly mentioned.

Test Series 1

```
bin/kafka-producer-perf-test.sh --topic foo --num-records 122880 --print-metrics --record-size 512 --throughput
2048 --producer.config producer.properties
```

The test produces ~120K records 512 bytes each (total of ~60MB) with the throughput throttle of 2048 rec/sec (1MB/sec)

Summary

Partitioner Config	Throughput	Avg Latency	P99 Latency	P99.9 Latency
Old DefaultPartitioner	0.84 MB/s	4072.74 ms	10992 ms	11214 ms

partitioner.adaptive.partitioning.enable=false	1 MB/s	49.92 ms	214 ms	422 ms
New logic with all default settings	1 MB/s	40.06 ms	154 ms	220 ms
partitioner.availability.timeout.ms=5	1 MB/s	36.29 ms	150 ms	184 ms

Old DefaultPartitioner (current implementation)

```
122880 records sent, 1724.873666 records/sec (0.84 MB/sec), 4072.74 ms avg latency, 11246.00 ms max latency, 3870 ms 50th, 10221 ms 95th, 10992 ms 99th, 11214 ms 99.9th.
```

The current implementation favors the slowest broker and manages to handle ~0.85 MB/s, so the latency grows over time.

```
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 46826262.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 9207276.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 9004193.000
```

The Kafka-0 broker takes ~5x more bytes than the other 2 brokers, becoming the bottleneck for the cluster and potentially skewing downstream data distribution.

New Uniform Partitioner (partitioner.adaptive.partitioning.enable=false)

```
122880 records sent, 2043.198484 records/sec (1.00 MB/sec), 49.92 ms avg latency, 795.00 ms max latency, 8 ms 50th, 150 ms 95th, 214 ms 99th, 422 ms 99.9th.

producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 22237614.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 21606034.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 22152273.000
```

All brokers take roughly the same load. The slowest broker isn't overloaded anymore, so the clusters is pulling 1MB/sec and latencies are more under control.

New Default Partitioner (all settings are default)

```
122880 records sent, 2045.477245 records/sec (1.00 MB/sec), 40.06 ms avg latency, 817.00 ms max latency, 7 ms 50th, 141 ms 95th, 154 ms 99th, 220 ms 99.9th.

producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 19244362.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 23589010.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 23321818.000
```

Here the adaptive logic sends less data to the slower broker, so faster brokers can take more load and the latencies are better. The data distribution isn't skewed too much, just enough to adjust load to broker capacity.

New Default Partitioner with partitioner.availability.timeout.ms=5

```
122880 records sent, 2044.218196 records/sec (1.00 MB/sec), 36.29 ms avg latency, 809.00 ms max latency, 6 ms 50th, 138 ms 95th, 150 ms 99th, 184 ms 99.9th.

producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 17431944.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 25781879.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 22977244.000
```

Here adaptive logic is more responsive to latency and sends even less data to the slower broker, which increases the mix of data processed by faster brokers. Note that it took me a few experiments to find a good value that made a difference, so this confirms the design decision that this logic should be off by default an only turned on after tuning to a specific configuration and workload.

Test Series 2

In this test series, the throughput throttle is increased to 2MB/sec.

```
bin/kafka-producer-perf-test.sh --topic foo --num-records 122880 --print-metrics --record-size 512 --throughput 4096 --producer.config producer.properties
```

The test produces ~120K records 512 bytes each (total of ~60MB) with the throughput throttle of 4096 rec/sec (2MB/sec)

New Uniform Partitioner (partitioner.adaptive.partitioning.enable=false)

```
122880 records sent, 3789.317873 records/sec (1.85 MB/sec), 426.24 ms avg latency, 2506.00 ms max latency, 8 ms 50th, 2065 ms 95th, 2408 ms 99th, 2468 ms 99.9th.

producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 22396882.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 21652393.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 21355134.000
```

Here the brokers take the same load, looks like the slowest broker is maxed out, so the cluster can only take 1.85MB/s and latencies grow.

New Default Partitioner (all settings are default)

```
122880 records sent, 4078.327249 records/sec (1.99 MB/sec), 34.66 ms avg latency, 785.00 ms max latency, 4 ms
50th, 143 ms 95th, 167 ms 99th, 297 ms 99.9th.

producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-0}  : 14866064.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-1}  : 25176418.000
producer-node-metrics:outgoing-byte-total:{client-id=perf-producer-client, node-id=node-2}  : 25581116.000
```

Adaptive logic manages to redistribute the load to faster brokers, to sustain the 2MB/sec throughput and the latencies are stable. Note that here a bigger skew in the distribution (vs. 1MB/sec throttle) was made to adjust load to broker capacity, showing that adaptive logic does just enough skew to keep latencies stable.

Rejected Alternatives

KafkaProducer to Explicitly Check for DefaultPartitioner

KafkaProducer could explicitly check if `DefaultPartitioner` or `UniformStickyPartitioner` is used and execute partitioning logic in the `RecordAccumulator` instead.

This approach was rejected because it was deemed that it would be confusing to have partitioner implementations that are ignored.

Partitioner.partition to return -1

The Partitioner.partition method could return -1 to indicate that a default partitioning decision should be made by the producer itself. Now, Partitioner.partition is required to return a valid partition number. This departs from the paradigm that partitioning logic (including the default partitioning logic) is fully encapsulated in a partitioner object, this encapsulation doesn't work well anymore as it requires information that only producer (Sender, RecordAccumulator) can know (such as queue sizes, record sizes, broker responsiveness, etc.). See the rejected alternatives section for an attempt to preserve encapsulation of default partitioning logic within a paritioner object.

When the producer gets -1 from the partitioner, it calculates the partition itself. This way the custom paritioner logic can continue to work and the producer would use the partition that is returned from the paritioner, however if the paritioner just wants to use default partitioning logic, it can return -1 and let the producer figure out the partition to use.

This also seems to be more future proof than trying to preserve (partial) encapsulation of partitioning logic within default partitioner, because if in the future we support additional signals, we can just change the logic in the producer and don't need to extend the partitioner interface to pass additional info.

This approach was rejected because it was deemed that this interface change wouldn't benefit other partitioners.

Callbacks in Partitioner.partition

As an alternative to allowing to return -1 from the Partitioner.partition method to indicate that the producer should execute default partitioning logic, it was considered to provide a callback interface that could feed information from producer back to the partitioner, as the following:

```
public interface Partitioner extends Configurable, Closeable {

    /**
     * Callbacks from the producer
     */
    interface Callbacks {
       /**
         * Get record size in bytes.  The keyBytes and valueBytes may present skewed view of the number
         * of bytes produced to the partition.  In addition, the callback takes into account the following:
         *  1. Headers
         *  2. Record overhead
         *  3. Batch overhead
         *  4. Compression
         *
         * @param partition The partition we need the record size for
         * @return The record size in bytes
         */
       int getRecordSize(int partition);

        /**
         * Calculate the partition number.  The producer keeps stats on partition load
         * and can use it as a signal for picking up the next partition.
         *
         * @return The partition number, or -1 if not implemented or not known
         */
       default int nextPartition() {
           return -1;
       }
    }

   // ... <skip> ...

   /**
     * Compute the partition for the given record.
     *
     * @param topic The topic name
     * @param key The key to partition on (or null if no key)
     * @param keyBytes The serialized key to partition on( or null if no key)
     * @param value The value to partition on or null
     * @param valueBytes The serialized value to partition on or null
     * @param callbacks The record size and partition callbacks (see {@link Partitioner#Callbacks})
     * @param cluster The current cluster metadata
     */
    default int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes,
                          Callbacks callbacks, Cluster cluster) {
        return partition(topic, key, keyBytes, value, valueBytes, cluster);
    }

   // ... <skip> ...
}
```

The getRecordSize callback method is needed to calculate the number of bytes in the record, the current information is not enough to calculate it accurately. It doesn't have to be 100% precise, but it needs to avoid systemic errors that could lead to skews over the long run (it's ok if, say, compression rate was a little bit off for one batch, it'll converge over the long run) and it should roughly match the batch size (e.g. we have to apply compression estimates if compression is used, otherwise we'll systemically switch partition before batch is full). See also the comments in the code snippet.

The nextPartition callback method effectively delegates partition switching logic back to producer.

This was an attempt to preserve the role separation between core producer logic and partitioner logic, but in reality it led to complicated interface (hard to understand the purpose without digging into implementation specifics and not really useful for other custom producers) and the logic that is logically tightly coupled (hard to understand partitioner logic without understanding producer logic and vice versa) but physically split between partitioner and core producer.

After doing that we realized that the desired encapsulation of default partitioning logic within default partitioner was broken anyway, so we might as well hoist the default partitioning logic into producer and let the default partitioner just inform the producer that the default partitioning logic is desired. Hoisting the logic into producer was also slightly more efficient, as the split logic required multiple lookups into various maps as it transitioned between producer and partitioner, now (with returning -1) lookup is made once and the logic runs in one go.