Keys to Faster Sampling in Dataflow

Ben Chambers, former Cloud Software Engineer
Rafael Fernandez, Cloud Engineering Manager

Editor's Note: Ben Chambers made the majority of the contributions to this post and white paper prior to moving on to other opportunities. He was a long-time Googler, and remains a strong contributor to the Apache Beam project.

In this whitepaper we show you how to improve the performance of a useful operation: Selecting a sample of elements on Cloud Dataflow. The ability to select such a sample is useful on its own, and the techniques used to improve its performance are generally applicable to other algorithms you might want to use with Cloud Dataflow.

Selecting a sample of elements in a `PCollection` is useful for diagnosing problems with your pipeline. You may look at the final results to verify they are correct, or inspect intermediate results to make sure each part of your pipeline is behaving correctly. The Apache Beam SDK includes `Sample.fixedSizeGlobally(...)` for taking such a sample.

You can make the built-in operation faster by spreading (in parallel) the sampling across multiple keys. This approach to improving performance by increasing parallelism is a generally useful strategy within Dataflow.

Next, we'll build a composite transform for producing a stratified sample that preserves the distribution of a specific property in the data. For example, we can produce a sample of US demographic data that ensures each state is (approximately) represented in the sample in the same proportion it was represented in the original data. We also ensure that elements belonging to "outlier keys" that would normally not be included in the sample have a chance to show up, which is useful for debugging problems that only manifest on these outliers.

We'll also show by example how composite transforms allow us to package this functionality and reuse it by building on the improved global sampling while creating the stratified sampling.

Sampling may be applied to many different kinds of data. For this post we use a high-volume (10 billion) of relatively small (100-byte) elements, which is similar to processing log messages or event logs. We'll be looking at producing a sample of 1,000 to 500,000 elements (1MB to 500MB). All of the pipelines are executed using autoscaling with a maximum of 128 workers.

Step 0: Built-in Baseline

The Beam Java SDK includes an implementation of distributed reservoir sampling. In the basic definition of reservoir sampling, you store *k* items that are your sample and as each datum is considered, it is added to a sample or not according to a random chance that decreases as you progress through the data set.

There are multiple ways to implement this technique in a distributed manner. In the SDK's implementation — `Sample.fixedSizeGlobally(k)` — the reservoirs from each worker are the accumulators of a `CombineFn`. As elements are added on each worker, they are assigned a random weight. The accumulator is limited to the top *k* elements according to their weights. They are computed separately on each worker, then gathered and merged. We're going to use the same approach, but build it ourselves so that we can make changes and improvements.

The basic idea is to use a `CombineFn` configured as follows:

1. The accumulator (the reservoir) is a max-heap with a bounded size of *k*.
2. Adding an element computes a random sampling weight *w*.
3. Once the maximum size of *k* is reached, adding new elements first removes the previous largest number.
4. Merging elements takes the *k* elements with the largest weights across all the accumulators.

See Appendix 1 and 2 for some helper classes which we need `SampleElement<T>` for representing an element paired with a random number and `BoundedHeap<T>` for a heap of bounded size ordered by that random number. We then use these to implement a `StaticallySizedSampleFn` which is a `CombineFn` that uses the `BoundedHeap<T>` as an accumulator to collect a sample of a fixed size.

Given the `CombineFn`, computing a sample of a fixed size is relatively straightforward. It uses `Combine.globally(...)` to apply the `StaticallySizedSampleFn`, and then unpacks the resulting `Iterable` of elements.

```
private static class FixedSizeGlobally<T>
    extends PTransform<PCollection<T>, PCollection<T>> {
  private final int sampleSize;

  public FixedSizeGlobally(int sampleSize) {
    this.sampleSize = sampleSize;
  }

  @Override
  public PCollection<T> expand(PCollection<T> input) {
    return input
        .apply(Combine.globally(new StaticallySizedSampleFn<>(sampleSize)))
        .apply(Flatten.iterables());
  }
}
```

Below is a table of pipeline run time and vCPU hours for our random dataset, including sample sizes of between 1,000 and 500,000 elements.

Note that for any more than 50,000 elements, the job fails after running for some time. The first stage of processing — generating all of the random numbers and writing the partial accumulator — is successfully completed on all the workers. The second stage — combining all of the partial accumulators to compute the final result — runs out of memory.

Sample Size	Total Execution Time (hours)	Total Worker Time (vCPU hour)
1,000 elements	9m12s	6.786 vCPU hours
5,000 elements	10m51s	10.757 vCPU hours
10,000 elements	13m46s	16.761 vCPU hours
50,000 elements	Failed after 1h18m47s	126.725 vCPU hours
100,000 elements	Failed after 2h10m49s	235.555 vCPU hours
500,000 elements	Failed after 9h33m27s	1,170.584 vCPU hours

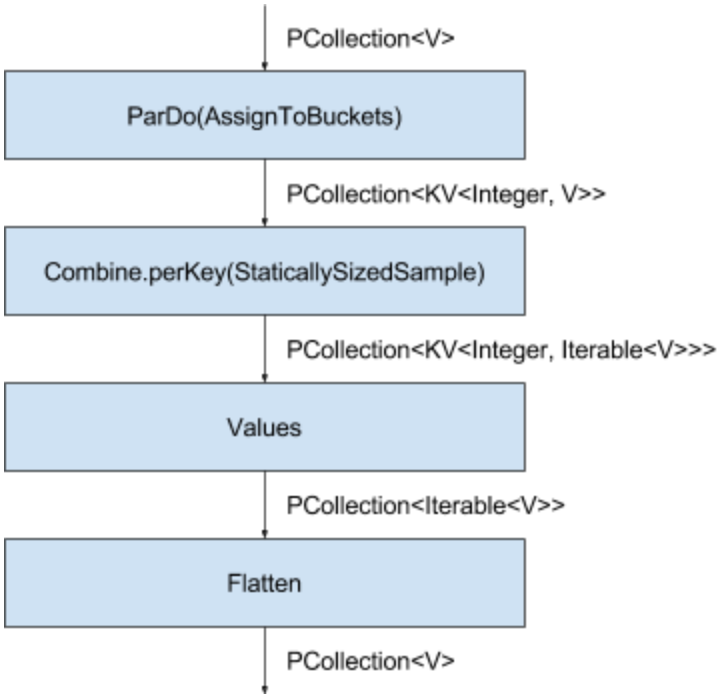
As you can see the process is quite time-consuming, and takes much longer as the sample size increases. Performing the `Combine.globally(...)` step, which computes the sample, requires single-threaded execution to produce a single result: the *n* sample elements. At large sample sizes, the pipeline runs out of memory and fails.

Step 1: Fixed Bucketing

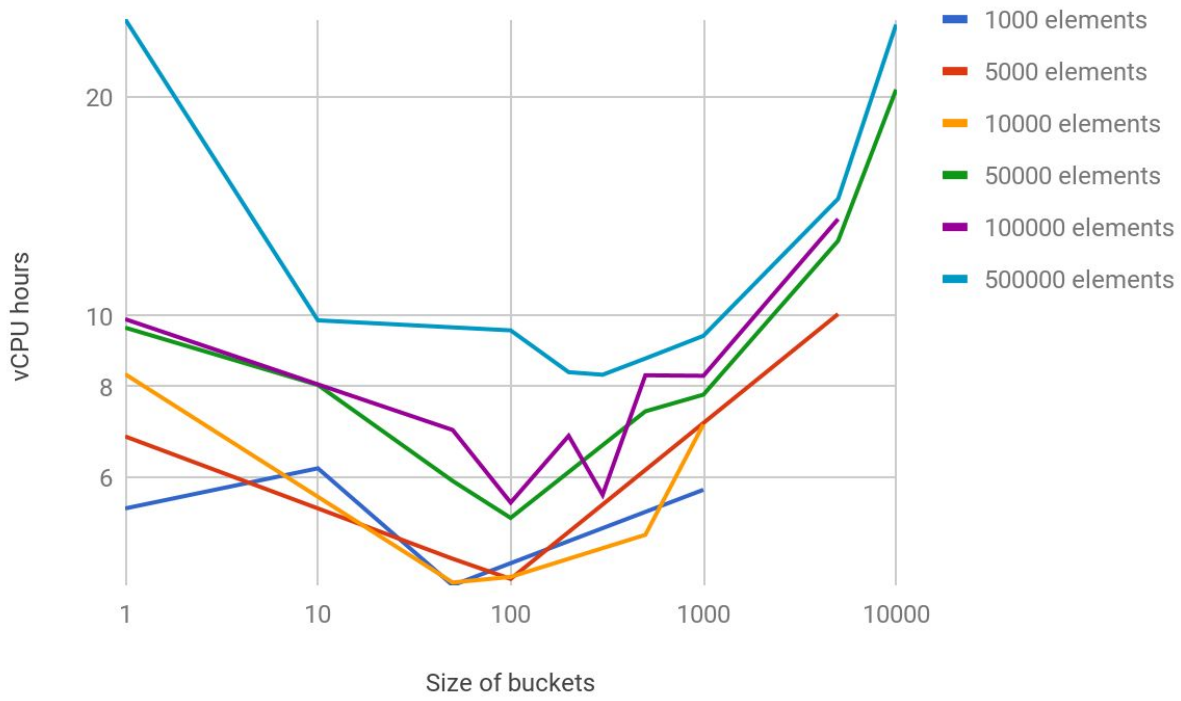
One way to sample more efficiently is to spread the sampling out across multiple buckets. This preliminary step reduces the size of the accumulator and allows each bucket to compute the result on a different worker. For instance, if we randomly divide our input between *k* keys, and then take an *m*-element sample within each key, we get to simultaneously spread the second stage of processing across multiple workers and also reduce the size of the accumulator.

One downside of bucketing like this is that we may produce fewer than *n* sample elements if the total input set isn't significantly larger than the desired sample size. As an extreme case, consider what happens if we are trying to compute a 200-element sample of a dataset with 200 elements. With the global `Combine` we would produce all 200 elements as the sample. If we are using 2 buckets of 100 elements each, we may randomly assign 105 elements to one bucket and 95 elements to the other. The first bucket will produce a 100 element sample of those 105 elements, and the second bucket will produce a 95 element sample. So we end up with only 195 elements in the sample.

Implementing fixed bucketing requires two changes to our previous transform—first we need to introduce a `DoFn` that can be used to assign each element to a random bucket, and then we need to change the `Combine.globally(...)` to a `Combine.perKey(...)`.



```
private static class FixedSizeBuckets<T>
    extends PTransform<PCollection<T>, PCollection<T>> {

  private final int numBuckets;
  private final int bucketSize;

  public FixedSizeBuckets(int numBuckets, int bucketSize) {
    this.numBuckets = numBuckets;
    this.bucketSize = bucketSize;
  }

  @Override
  public PCollection<T> expand(PCollection<T> input) {
    return input
        .apply(ParDo.of(new AssignToFixedBucketsDoFn<>(numBuckets)))
        .apply(Combine.perKey(new StaticallySizedSampleFn<>(bucketSize)))
        .apply(Values.create())
        .apply(Flatten.iterables());
  }
}

public static class AssignToFixedBucketsDoFn<V>
    extends DoFn<V, KV<Integer, V>> {

  private final int numBuckets;
  private transient int index

  public AssignToFixedBucketsDoFn(int numBuckets) {
    this.numBuckets = numBuckets;
  }

  @Setup
  public void startBundle() {
    index = ThreadLocalRandom.current().nextInt(numBuckets);
  }

  @ProcessElement
  public void processElement(ProcessContext c) throws Exception {
    index = (index + 1) % numBuckets;
    c.output(KV.of(index, c.element()));
  }
}
```



Sample Size	Bucket Size	Total Execution Time (hours)	Total Worker Time (vCPU hours)
1000	1	9m24s	5.434
1000	10	8m43s	6.171
1000	100	9m30s	4.259
1000	1000	10m07s	5.765
5000	1	9m38s	6.827
5000	50	10m06s	4.633
5000	100	9m30s	4.346
5000	5000	10m22s	10.063
10000	1	9m46s	8.316
10000	50	9m40s	4.297
10000	100	9m33s	4.374
10000	500	9m34s	4.998
10000	1000	9m25s	7.094
50000	1	10m21s	9.639

50000	10	9m44s	8.037
50000	50	11m19s	5.929
50000	100	10m11s	5.273
50000	500	9m47s	7.393
50000	1000	9m31s	7.794
50000	5000	11m47s	12.695
50000	10000	16m18s	20.504
50000	50000	1h09m54s	116.613
100000	1	10m18s	9.902
100000	50	8m30s	6.969
100000	100	9m31s	5.536
100000	200	9m15s	6.84
100000	300	9m53s	5.673
100000	500	9m54s	8.288
100000	1000	9m55s	8.274
100000	5000	11m44s	13.601
500000	1	17m37s	25.626
500000	10	10m54s	9.865
500000	100	10m06s	9.556
500000	200	9m56s	8.37
500000	300	10m01s	8.302
500000	500	10m02s	8.741
500000	1000	10m51s	9.391
500000	5000	12m47s	14.503
500000	10000	18m47s	25.198

Note that the rows where the number of divisions is 1 correspond approximately to the baseline case above.

Dataflow includes some optimizations that makes Combine operations more efficient. Specifically, it will perform partial local combining on all the workers before sending the results to a single worker to produce the final result.

This example demonstrates an interesting property. Spreading the work across more than one

key significantly improves performance, as it improves parallelism and makes the accumulators a more manageable size. Spreading the work across too many keys reduces the benefits of partial local combining, we can only combine partial results for the same key.

It seems like buckets of size 100 elements are generally pretty good for this data set. It ensures there are enough buckets to parallelize the work without producing too many accumulators or allowing the accumulators to be too large.

Another interesting property of this bucketing is that while using more buckets increases the parallelism, it also increases the number of elements necessary to ensure that all buckets are full. An under-filled bucket will lead to a sample that is smaller than desired. For our intended use there will be significantly more input elements than desired sample elements, so there is no cause for concern.

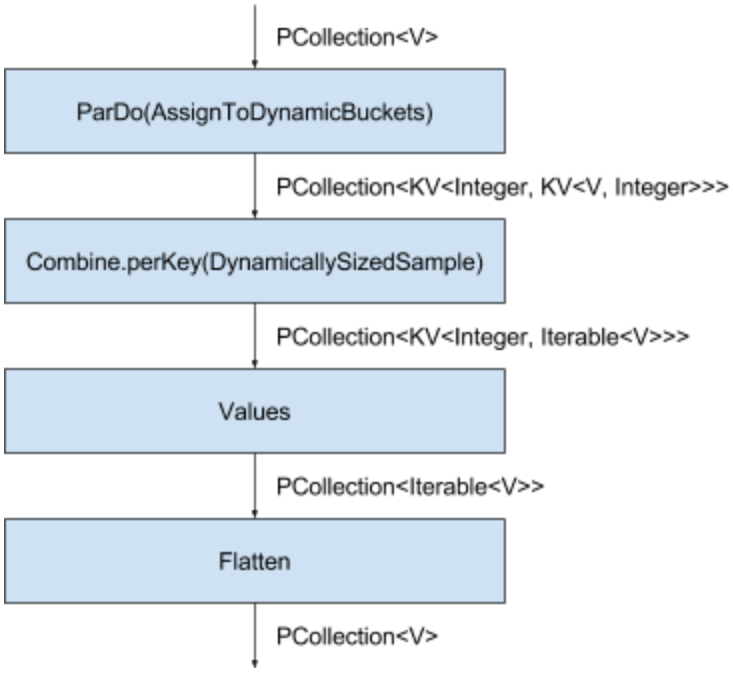
Step 2: Dynamic Bucketing

From the previous experiments, we've learned that there is a specific sample size within the buckets that performs the best — in this case it is 100 elements. Building on this, we're going to make a version of global sampling that takes the desired sample size and the maximum bucket size. Unlike the previous case where we took the number of buckets and the bucket size, this allows us to choose a bucket size and apply this bucketing to produce any sample size.

This requires we switch to a dynamically sized bucket — for example, if we want a 250 element sample with maximum bucket sizes of 100 elements we need to use two buckets of size 100 and one bucket of size 50.

Implementing this is a bit tricky — we need to pass the bucket size into the `CombineFn`. It would be most natural to make it a part of the key (for instance `KV.of(bucketIndex, bucketSize)`), but the `CombineFn` API does not allow accessing the key from within the `CombineFn`. So instead, we pass the bucket size as part of the value.

At first, this may be concerning because we are adding the bucket size to **every value** passed to the `CombineFn`. However, thanks to the partial local combining mentioned previously, these values will be incorporated into a partial accumulator before being transmitted between workers.



We present the corresponding code below. This code re-uses the `BoundedHeap` and also depends on a new `DynamicallySizedSampleFn` — shown in Appendix 3 — which is a `CombineFn` that uses a dynamically configured size for the heap. It is very similar to the `StaticallySizedSampleFn` we used earlier.

```
private static class RandomSample<T>
    extends PTransform<PCollection<T>, PCollection<T>> {

  private final int sampleSize;
  private final int maxBucketSize;

  public RandomSample(int sampleSize, int maxBucketSize) {
    this.sampleSize = sampleSize;
    this.maxBucketSize = maxBucketSize;
  }

  @Override
  public PCollection<T> expand(PCollection<T> input) {
    return input
        .apply(ParDo.of(new AssignToDynamicBucketsDoFn<>(
            sampleSize, maxBucketSize)))
        .apply("Sample each bucket",
           Combine.perKey(new DynamicallySizedSampleFn<>()))
        .apply(Values.create())
        .apply(Flatten.iterables());
  }
}

/** A holder for the assignment of an element to a bucket. */
public static class BucketAssignment {

  private final int bucketIndex;
  private final int bucketSize;

  private BucketAssignment(int bucketIndex, int bucketSize) {
    this.bucketIndex = bucketIndex;
    this.bucketSize = bucketSize;
  }

  public int bucketIndex() {
    return bucketIndex;
  }

  public int bucketSize() {
    return bucketSize;
```

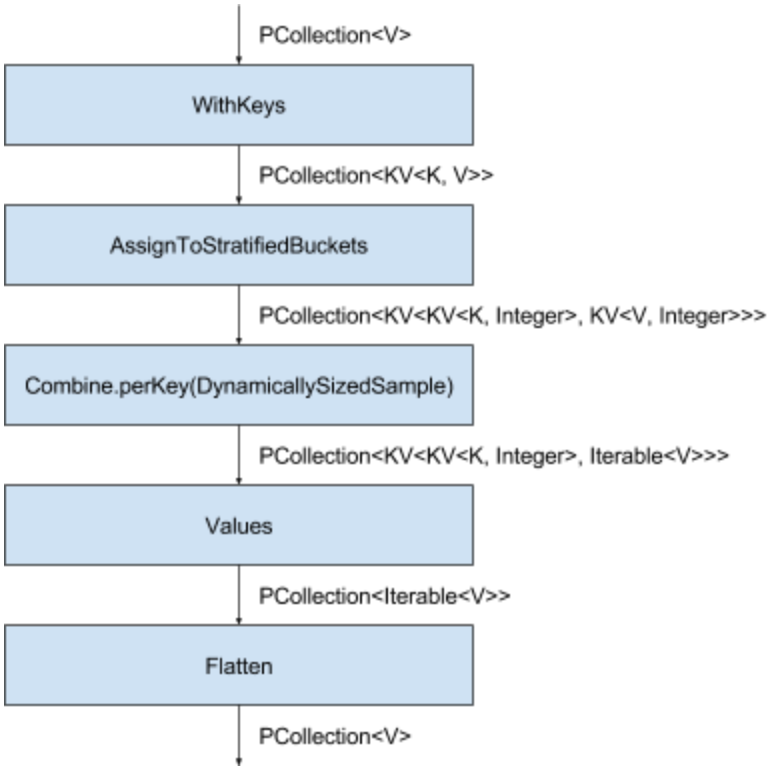
```
  }
}

/**
 * Given a random position in the range {@code [0..., sampleSize)} return
 * an assignment of that position into buckets of size {@code maxBucketSize}.
 */
@VisibleForTesting
static BucketAssignment assignBucket(
    int assignedPosition, int sampleSize, int maxBucketSize) {
  int assignedBucket = assignedPosition / maxBucketSize;
  // The size of this bucket is either maxBucketSize or
  // sampleSize % maxBucketSize if it is the final (remainder) bucket.
  int remainderSize = sampleSize % maxBucketSize;
  boolean isRemainderBucket = assignedPosition > (sampleSize - remainderSize);
  int bucketSize = isRemainderBucket ? remainderSize : maxBucketSize;
  return new BucketAssignment(assignedBucket, bucketSize);
}

public static class AssignToDynamicBucketsDoFn<V>
    extends DoFn<V, KV<Integer, KV<V, Integer>>> {

  private final int sampleSize;
  private final int maxBucketSize;
  private transient int index;

  public AssignToDynamicBucketsDoFn(int sampleSize, int maxBucketSize) {
    this.sampleSize = sampleSize;
    this.maxBucketSize = maxBucketSize;
  }

  @Setup
  public void setup() {
    index = ThreadLocalRandom.current().nextInt(sampleSize);
  }

  @ProcessElement
  public void processElement(ProcessContext c) throws Exception {
    index = (index + maxBucketSize) % sampleSize;
    BucketAssignment bucket = assignBucket(index, sampleSize, maxBucketSize);
    c.output(KV.of(bucket.bucketIndex(),
        KV.of(c.element(), bucket.bucketSize())));
  }
}
```

Step 3: Stratified Random Sampling

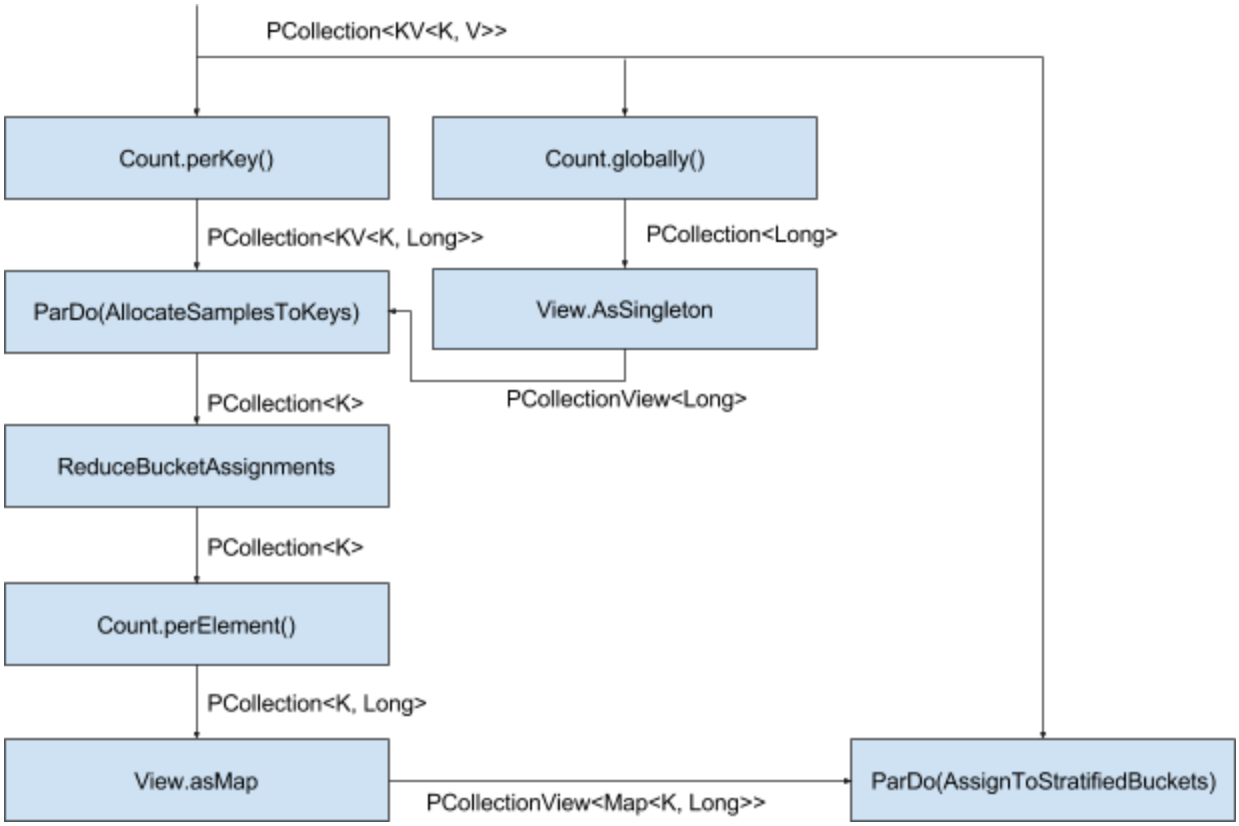
Now that we understand how to more efficiently produce a global sample by first dividing it across some random keys, we are equipped to investigate how to produce a more informative sample for debugging purposes.

Often, a data set has several “kinds” of elements. For debugging, it is useful to get a sample that contains some of each kind. Ideally, this would be proportional to how frequently each kind of element appears in the total data set — a stratified sample.

For our debugging purposes, we make one small change. No matter how infrequently a kind of element is, we would like at least one element of the sample to be of that kind. This ensures that potential outliers are represented, at the risk of producing a less representative sample.



Notice again that the actual sampling transform didn’t change significantly. The only differences are that we first assign keys to the input elements using a function to extract the kind of element for stratification. We also use a new composite transform to assign the elements to stratified buckets. Also note that we continue to use the assigned key and an integer for each bucket’s key.



The `AssignToStratifiedBuckets` transform is a composite transform that makes use of the existing sampling. A general outline of the approach:

1. Determine how many total elements there are in the data set using `Count.globally()`. This is made available to a later ParDo as a side-input, by using `View.asSingleton()`
2. Determine how many of each kind of element they are using `Count.perKey()`.
3. For each kind of element, allocate it to one or more buckets based on its frequency. This uses the count-per-key as a main input and the global count as a side input.
4. We may have assigned too many samples, so we apply our previous dynamic bucketing as a composite to potentially reduce the number of samples.
5. Determine how many samples should be taken for each key using `Count.perElement()`. Use `View.asMap()` to make that available as a side-input.
6. Use a ParDo to assign buckets to each element. This takes the previously computed map as a side-input, allowing it to determine how many samples should be taken for each element.

```
private static class StratifiedRandomSample<K, V>
    extends PTransform<PCollection<V>, PCollection<V>> {
  private final int sampleSize;
  private final int maxBucketSize;
  private final SerializableFunction<V, K> keyFn;
```

```
  public StratifiedRandomSample(
      int sampleSize, int maxBucketSize, SerializableFunction<V, K> keyFn) {
    this.sampleSize = sampleSize;
    this.maxBucketSize = maxBucketSize;
    this.keyFn = keyFn;
  }

  @Override
  public PCollection<V> expand(PCollection<V> input) {
    final PCollection<KV<K, V>> keyedInput = input.apply(WithKeys.of(keyFn));

    return keyedInput
        .apply("Assign Stratified Buckets",
            new AssignValuesToStratifiedBuckets<>(sampleSize, maxBucketSize))
        .apply("DynamicSample Each Bucket",
            Combine.perKey(new DynamicallySizedSampleFn<>()))
        .apply(Values.create())
        .apply(Flatten.iterables());
  }
}


/**
 * Given a collection of {@code KV<K,V>} pairs, produce a collection where
 * each key is divided into an arbitrary bucket and each value includes a
 * bucket size.
 *
 * <p>Specifically, given {@code KV.of("key", "value")}, returns
 * {@code KV.of(KV.of("key", randomBucket), KV.of("value", bucketSize))}.
 */
private static class AssignValuesToStratifiedBuckets<K, V>
    extends PTransform<PCollection<KV<K, V>>,
    PCollection<KV<KV<K, Integer>, KV<V, Integer>>>> {

  private final int sampleSize;
  private final int maxBucketSize;

  private AssignValuesToStratifiedBuckets(
      int sampleSize, int maxBucketSize) {
    this.sampleSize = sampleSize;
    this.maxBucketSize = maxBucketSize;
  }

  @Override
  public PCollection<KV<KV<K, Integer>, KV<V, Integer>>> expand(
      PCollection<KV<K, V>> input) {
    // Figure out how many total rows there are
```

```
    final PCollectionView<Long> numberOfRows = input
        .apply("Count total rows", Count.globally())
        .apply(View.asSingleton());

    final PCollectionView<Map<K, Long>> itemsPerKey = input
        // Count how many rows each key has
        .apply("Count rows per key", Count.perKey())
        // Allocate samples to each key based on the ratio of data with
        // that key. Even infrequent keys are assigned at least one element
        // of the sample.
        .apply("Allocate samples to each key",
            ParDo.of(new AllocateSamplesDoFn<K>(sampleSize, numberOfRows))
                .withSideInputs(numberOfRows))
        // If there are very many distinct keys, the allocated samples
        // may exceed the actual desired bucketSize for the sample.
        // Since the number of allocated keys is likely close to the desired
        // sample size, we shouldn't use our sampling algorithm to reduce the
        // set because it would be likely to have underfilled buckets.
        .apply(new ReduceBucketAssignments<>(sampleSize))
        // Then we count how many samples each key has allocated to it
        .apply("Count DynamicSample Allocation", Count.<K>perElement())
        // And create a PCollectionView
        .apply(View.<K, Long>asMap());

    return input
        // Assign each item to a bucket. The number of buckets for a
        // given key is determined by the number of samples allocated
        // to the key, and the maximum bucket bucketSize.
        .apply(ParDo.of(
            new AssignToStratifiedBucketsDoFn<K, V>(maxBucketSize, itemsPerKey))
            .withSideInputs(itemsPerKey));
  }
}

/**
 * DoFn that maps {@code KV<K, V>} elements to a
 * bucket (within the key) and the size of the bucket.
 *
 * <p>Requires the sample-size per key as a side-input.
 */
public static class AssignToStratifiedBucketsDoFn<K, V>
    extends DoFn<KV<K, V>, KV<KV<K, Integer>, KV<V, Integer>>> {

  private final int maxBucketSize;
  private final PCollectionView<Map<K, Long>> itemsPerKey;
  private transient ThreadLocalRandom random;
```

```
  public AssignToStratifiedBucketsDoFn(
      int maxBucketSize, PCollectionView<Map<K, Long>> itemsPerKey) {
    this.maxBucketSize = maxBucketSize;
    this.itemsPerKey = itemsPerKey;
  }

  @StartBundle
  public void startBundle() {
    random = ThreadLocalRandom.current();
  }

  @ProcessElement
  public void processElement(ProcessContext c) throws Exception {
    Long sampleSizeLong = c.sideInput(itemsPerKey).get(c.element().getKey());
    if (sampleSizeLong == null) {
      return;
    }

    int sampleSize = (int) (long) sampleSizeLong;
    int assignedPosition = random.nextInt(sampleSize);
    BucketAssignment bucket = assignBucket(
        assignedPosition, sampleSize, maxBucketSize);
    c.output(KV.of(
          KV.of(c.element().getKey(), bucket.bucketIndex()),
          KV.of(c.element().getValue(), bucket.bucketSize())));
  }
}

private static class AllocateSamplesDoFn<K> extends DoFn<KV<K, Long>, K> {
  private final int sampleSize;
  private final PCollectionView<Long> numberOfRows;

  public AllocateSamplesDoFn(
      int sampleSize, PCollectionView<Long> numberOfRows) {
    this.sampleSize = sampleSize;
    this.numberOfRows = numberOfRows;
  }

  @ProcessElement
  public void processElement(ProcessContext c) throws Exception {
    long keyRows = c.element().getValue();
    long totalRows = c.sideInput(numberOfRows);
    long samples = getNumAllocatedSamples(keyRows, totalRows, sampleSize);

    for (int i = 0; i < samples; i++) {
      c.output(c.element().getKey());
    }
```

```
  }
}

/**
 * Compute the number of samples that should be allocated to a given key.
 *
 * Rounds up so that even outlier keys receive one allocated sample.
 *
 * @param keyRows The number of rows in the data set with this key.
 * @param totalRows The number of total rows in the data set.
 * @param sampleSize The number of desired rows in the sample.
 * @return The number of rows that should be allocated to this key in the sample.
 */
@VisibleForTesting static long getNumAllocatedSamples(
  long keyRows, long totalRows, long sampleSize) {
  // Always round up. This ensures that outliers (which represent less than
  // one full sample) still have a chance to appear, and also ensures that
  // we choose enough samples.
  return (long) Math.ceil(keyRows * 1.0 * sampleSize / totalRows);
}

private static class ReduceBucketAssignments<K>
    extends PTransform<PCollection<K>, PCollection<K>> {

  private static final Logger LOG =
LoggerFactory.getLogger(AllocateSamplesDoFn.class);

  private final int sampleSize;
  public ReduceBucketAssignments(int sampleSize) {
    this.sampleSize = sampleSize;
  }

  @Override
  public PCollection<K> expand(PCollection<K> input) {
    return input
      .apply(WithKeys.<Void, K>of((Void) null)
          .withKeyType(new TypeDescriptor<Void>() {}))
      .apply(GroupByKey.create())
      .apply(ParDo.of(new DoFn<KV<Void, Iterable<K>>, K>() {
        @ProcessElement
        public void processElement(ProcessContext c) {
          Iterator<K> keyIterator = c.element().getValue().iterator();
          try {
            for (int i = 0; i < sampleSize; i++) {
              c.output(keyIterator.next());
            }
          } catch (NoSuchElementException e) {
```

```
            LOG.warn("Not enough samples allocated.");
          }
        }
      }));
    }
  }
```

As before, we experimented with buckets of varying sizes. We also experiment with varying the number of keys. We again find that using 100 element buckets produces good results by balancing the size of the accumulator with the amount of parallelism.

Sample Size	Bucket Size	Number of Keys	Total Elapsed Time (hours)	Total Worker Time (vCPU hours)
10000	1	5	26m17s	41.893
10000	1	10	26m20s	43.44
10000	1	25	26m35s	43.293
10000	100	5	22m20s	34.341
10000	100	10	23m22s	36.489
10000	100	25	23m52s	37.222
10000	1000	5	27m27s	44.741
10000	1000	10	23m54s	37.286
10000	1000	25	23m51s	35.874
10000	5000	5	28m29s	44.516
10000	5000	10	22m47s	36.21
10000	5000	25	23m54s	36.171
50000	1	5	25m54s	42.278
50000	1	10	26m41s	43.43
50000	1	25	41m06s	72.672
50000	100	5	23m55s	36.969
50000	100	10	24m10s	37.245
50000	100	25	24m04s	37.133
50000	1000	5	24m20s	38.56
50000	1000	10	24m40s	38.092
50000	1000	25	24m01s	37.25
50000	5000	5	26m17s	42.033

50000	5000	10	30m10s	49.296
50000	5000	25	25m18s	39.931
100000	1	5	26m43s	43.454
100000	1	10	27m02s	45.837
100000	1	25	31m00s	53.77
100000	100	5	36m46s	63.839
100000	100	10	24m12s	37.576
100000	100	25	24m00s	37.806
100000	1000	5	23m37s	36.913
100000	1000	10	24m19s	38.441
100000	1000	25	23m43s	37.546
100000	5000	5	26m10s	42.353
100000	5000	10	26m05s	42.776
100000	5000	25	26m03s	42.453
500000	1	5	35m23s	61.376
500000	1	10	37m12s	66.064
500000	1	25	39m12s	70.259
500000	100	5	25m16s	40.221
500000	100	10	27m13s	44.206
500000	100	25	25m38s	41.766
500000	1000	5	25m29s	40.238
500000	1000	10	25m49s	40.551
500000	1000	25	25m35s	41.535
500000	5000	5	27m48s	44.953
500000	5000	10	38m04s	63.882
500000	5000	25	28m28s	46.43

Conclusion

We demonstrated how to introduce additional parallelism to random sampling as a way of improving pipeline performance. The same approaches may be useful in writing your own pipelines. We also demonstrated how to build more sophisticated sampling from simpler parts by reusing transforms. The preceding approaches may both be useful when writing your own pipelines.

We also provided a re-usable approach for stratified random sampling which should be helpful for taking a peek at the contents of a `PCollection` for debugging purposes.

Appendix 1: SampleElement, BoundedHeap and some Coders

```
/** An element paired with a random value used for comparison. */
private static class SampleElement<T> implements Comparable<SampleElement<T>> {
  private final int value;
  private final T element;

  public SampleElement(int value, T element) {
    this.value = value;
    this.element = element;
  }

  @Override
  public int compareTo(SampleElement<T> o) {
    return Integer.compare(o.value, this.value);
  }
}

/** The coder for {@code SampleElement<T>} uses the coder for {@code T}. */
private static class SampleElementCoder<T>
    extends CustomCoder<SampleElement<T>> {

  private final Coder<Integer> intCoder = BigEndianIntegerCoder.of();
  private final Coder<T> elementCoder;

  public SampleElementCoder(Coder<T> elementCoder) {
    this.elementCoder = elementCoder;
  }

  @Override
  public void encode(SampleElement<T> value, OutputStream outStream)
      throws IOException {
    intCoder.encode(value.value, outStream);
    elementCoder.encode(value.element, outStream);
  }

  @Override
  public SampleElement<T> decode(InputStream inStream) throws IOException {
    int value = intCoder.decode(inStream);
    T element = elementCoder.decode(inStream);
    return new SampleElement<>(value, element);
  }
```

```
}

/** A heap that stores a bounded number of {@link SampleElement elements}. */
static class BoundedSample<T> {

  /**
   * A list in which smallest key at the front for quick merging.
   *
   * <p>Only one of asList and asQueue may be non-null.
   */
  private List<SampleElement<T>> asList;

  /**
   * A queue with largest random key at the head, for quick addition.
   *
   * <p>Only one of asList and asQueue may be non-null.
   */
  private PriorityQueue<SampleElement<T>> asQueue;

  /** The maximum sampleSize of the heap. */
  private int maximumSize;

  private BoundedSample(int maximumSize,
      PriorityQueue<SampleElement<T>> asQueue,
      List<SampleElement<T>> asList) {
    this.maximumSize = maximumSize;
    this.asQueue = asQueue;
    this.asList = asList;
  }

  public static <T> BoundedSample<T> fromSortedList(
      int maximumSize, List<SampleElement<T>> asList) {
    return new BoundedSample<>(maximumSize, null, asList);
  }

  public List<SampleElement<T>> sortedList() {
    if (maximumSize == 0) {
      return Collections.emptyList();
    }

    if (asList == null) {
      List<SampleElement<T>> reverseList = new ArrayList<>(maximumSize);
      while (!asQueue.isEmpty()) {
        reverseList.add(asQueue.poll());
      }
      asList = Lists.reverse(reverseList);
      asQueue = null;
```

```
    }
    return asList;
  }

  public static <T> BoundedSample<T> fromSamples(
      Iterable<BoundedSample<T>> samples) {
    BoundedSample<T> result = null;
    for (BoundedSample<T> sample : samples) {
      if (sample.getMaximumSize() != 0) {
        if (result == null) {
          result = sample;
        } else {
          for (SampleElement<T> element : sample.sortedList()) {
            if (!result.maybeAddInput(element)) {
              break;
            }
          }
        }
      }
    }
    return result;
  }

  public static <T> BoundedSample<T>create() {
    return new BoundedSample(0, null, null);
  }

  public static <T> BoundedSample<T>create(int maximumSize) {
    return new BoundedSample(
      maximumSize, new PriorityQueue<>(maximumSize), null);
  }

  private boolean maybeAddInput(SampleElement<T> element) {
    if (maximumSize == 0) {
      return false;
    }

    if (asQueue == null) {
      asQueue = new PriorityQueue<>(asList);
      asList = null;
    }

    if (asQueue.size() < maximumSize) {
      asQueue.add(element);
      return true;
    } else if (element.value < asQueue.peek().value) {
      asQueue.poll();
```

```
      asQueue.add(element);
      return true;
    }

    return false;
  }

  public boolean maybeAddInput(int randomInt, T value) {
    if (maximumSize == 0) {
      return false;
    }

    if (asQueue == null) {
      asQueue = new PriorityQueue<>(asList);
      asList = null;
    }

    if (asQueue.size() < maximumSize) {
      asQueue.add(new SampleElement<T>(randomInt, value));
      return true;
    } else if (randomInt < asQueue.peek().value) {
      asQueue.poll();
      asQueue.add(new SampleElement<T>(randomInt, value));
      return true;
    }

    return false;
  }

  public int getMaximumSize() {
    return maximumSize;
  }

  public void setMaximumSize(int maximumSize) {
    Preconditions.checkState(this.maximumSize == 0);
    Preconditions.checkState(this.asQueue == null && this.asList == null);
    this.maximumSize = maximumSize;
    this.asQueue = new PriorityQueue<SampleElement<T>>(maximumSize);
  }

  Iterable<T> unsortedOutput() {
    if (asQueue == null && asList == null) {
      return Collections.emptyList();
    } else {
      Iterable<SampleElement<T>> iterable = asQueue == null ? asList : asQueue;
      return Iterables.transform(iterable, new Function<SampleElement<T>, T>() {
        @Nullable
```

```
        @Override
        public T apply(@Nullable SampleElement<T> input) {
          return input.element;
        }
      });
    }
  }
}

/**
 * A {@link Coder} for {@link BoundedSample}.
 */
private static class BoundedSampleCoder<T>
    extends CustomCoder<BoundedSample<T>> {

  private final Coder<Integer> sizeCoder = VarIntCoder.of();
  private final Coder<List<SampleElement<T>>> listCoder;

  public BoundedSampleCoder(Coder<T> elementCoder) {
    listCoder = ListCoder.of(new SampleElementCoder(elementCoder));
  }

  @Override
  public void encode(BoundedSample<T> value, OutputStream outStream)
      throws IOException {
    sizeCoder.encode(value.maximumSize, outStream);
    if (value.maximumSize != 0) {
      listCoder.encode(value.sortedList(), outStream);
    }
  }

  @Override
  public BoundedSample<T> decode(InputStream inStream) throws IOException {
    int size = sizeCoder.decode(inStream);
    if (size == 0) {
      return BoundedSample.create();
    } else {
      return BoundedSample.fromSortedList(size, listCoder.decode(inStream));
    }
  }
}
```

Appendix 2: Fixed Size Sampling CombineFn

```
/**
  * {@code CombineFn} that computes a fixed-size sample of a
  * collection of values.
  *
  * @param <T> the type of the elements
  */
 public static class StaticallySizedSampleFn<T>
     extends CombineFn<T, BoundedSample<T>, Iterable<T>> {

   private final Random rand = new Random();
   private final int size;

   private StaticallySizedSampleFn(int size) {
     this.size = size;
   }

   @Override
   public BoundedSample<T> createAccumulator() {
     return BoundedSample.create(size);
   }

   @Override
   public BoundedSample<T> addInput(BoundedSample<T> accumulator, T input) {
     accumulator.maybeAddInput(rand.nextInt(), input);
     return accumulator;
   }

   @Override
   public BoundedSample<T> mergeAccumulators(
       Iterable<BoundedSample<T>> accumulators) {
     return BoundedSample.fromSamples(accumulators);
   }

   @Override
   public Iterable<T> extractOutput(BoundedSample<T> accum) {
     return accum.unsortedOutput();
   }

   @Override
   public Coder<BoundedSample<T>> getAccumulatorCoder(
       CoderRegistry registry, Coder<T> inputCoder) {
     return new BoundedSampleCoder<>(inputCoder);
```

```
    }

    @Override
    public Coder<Iterable<T>> getDefaultOutputCoder(
        CoderRegistry registry, Coder<T> inputCoder) {
      return IterableCoder.of(inputCoder);
    }
  }
```

Appendix 3: Dynamic Sampling CombineFn

```
/**
   * {@code CombineFn} that computes a fixed-bucketSize sample of a
   * collection of values.
   *
   * @param <T> the type of the elements
   */
  public static class DynamicallySizedSampleFn<T>
      extends CombineFn<KV<T, Integer>, BoundedSample<T>, Iterable<T>> {

    private final Random rand = new Random();

    private DynamicallySizedSampleFn() {
    }

    @Override
    public BoundedSample<T> createAccumulator() {
      return BoundedSample.create();
    }

    @Override
    public BoundedSample<T> addInput(
        BoundedSample<T> accumulator, KV<T, Integer> input) {
      if (accumulator.getMaximumSize() == 0) {
        accumulator.setMaximumSize(input.getValue());
      }
      accumulator.maybeAddInput(rand.nextInt(), input.getKey());
      return accumulator;
    }

    @Override
    public BoundedSample<T> mergeAccumulators(
```

```
        Iterable<BoundedSample<T>> accumulators) {
      return BoundedSample.fromSamples(accumulators);
    }

    @Override
    public Iterable<T> extractOutput(BoundedSample<T> accum) {
      return accum.unsortedOutput();
    }

    @Override
    public Coder<BoundedSample<T>> getAccumulatorCoder(
        CoderRegistry registry, Coder<KV<T, Integer>> inputCoder) {
      KvCoder<T, Integer> kvCoder = (KvCoder) inputCoder;
      return new BoundedSampleCoder<>(kvCoder.getKeyCoder());
    }

    @Override
    public Coder<Iterable<T>> getDefaultOutputCoder(
        CoderRegistry registry, Coder<KV<T, Integer>> inputCoder) {
      KvCoder<T, Integer> kvCoder = (KvCoder) inputCoder;
      return IterableCoder.of(kvCoder.getKeyCoder());
    }
  }
```