Developer's Guide to Prometheus Native Histograms

Carrie Edwards

Resources

Design Doc
PromCon talk by Ganesh
Talk by Beorn
Talk by Ganesh

Overview of Native Histograms

Introduction

Native histograms are a data structure in Prometheus that store a distribution of data points. They are similar to the originally implemented Prometheus histograms (classic histograms) in many ways, but help to address several problems that the originally implemented Prometheus histograms (classic histograms) were facing.

Histograms are often used to calculate quantile values to determine important information, such as finding the percentage of requests that had a response time of more than a certain number of milliseconds. However, the accuracy of a quantile estimation depends on the bucket layout, and how well the buckets of the histogram are defined. Proper bucket boundaries can be difficult to determine, as it depends on the type of data being represented in the histogram, and what the expected ranges of values are. This often requires experimentation before getting it right.

With the originally implemented classic histograms in Prometheus:

- The bucket boundaries need to be predefined before instrumentation
- The layout of the buckets have to be compatible across aggregated metrics and across the range of rate calculation, which was difficult to determine, and could potentially require changing the bucket layout fairly frequently
- If the bucket boundaries of interest changed at any point, then the bucket layouts have to be changed. Each time the bucket boundaries change, it necessary to re-instrument and re-deploy everywhere for the new bucket boundaries to persist
- Buckets were expensive to store, so most users only define a small number of buckets. With a smaller number of total buckets, the ranges of values stored in each bucket have to be fairly precise to prevent high degrees of inaccuracy in quantile estimations
- Because buckets were expensive to store, empty buckets wasted a lot of space

Native histograms address these issues because of a key difference in how bucket boundaries are determined. Instead of requiring configuration during instrumentation, knowledge about the expected values or needing to pre-determine the bucket boundaries like classic histograms, native histogram use a logarithmic bucket layout with predefined boundaries. The buckets have an exponential growth factor that is determined by a set scale, and the resolution/precision can be adjusted. Native histograms are easier to aggregate with each other in both time and space, and the quantile and percentage estimations are improved. Additionally, native histograms are lower cost and much more efficient to store.

There are two types of native histograms: integer histograms, and float histograms. Integer histograms contain integer counts of values, while float histograms contain counts with floating point precision. Float histograms can therefore store fractional counts of values. Each native histogram represents a single time series.

Structure

Native histograms are structured and encoded in a way that optimizes efficiency of storage, and allows for easier aggregation. Here is the basic structure and properties:

- CounterResetHint
    - Used to detect if the observation count of any bucket goes down. If this occurs in either the overall histogram's count field, a decrease of a value in any individual bucket, or in the zero count, then the entire histogram is considered to be a counter reset
- Schema
    - A number that is used to define the relationship between the boundaries of the buckets
- Zero Threshold
    - The boundaries of values that will considered to be zero (and thus placed in the zero bucket, rather than in the positive or negative buckets)
- ZeroCount
    - A bucket that holds the count of values that fall within the zero threshold
- Count
    - Total number of observations that have been placed in the zeroCount, positiveBuckets and negativeBuckets
- Sum
    - The sum of all of the values placed in the positiveBuckets and negativeBuckets
- Positive and Negative Spans
    - Represent the layouts of positive and negative buckets
- Positive and Negative Buckets
    - Represent the counts of values. Buckets have fixed boundaries that are based on the schema

Properties

Native histograms use an exponential growth function to calculate bucket boundaries. The exponential growth function is based on the schema, which determines how the buckets will scale. The bucket boundaries will change depending on which schema is used; the larger the schema, the more narrow the buckets are.

In terms of encoding native histograms, a sparse representation of the buckets is used for improved storage efficiency. This means that the boundaries of each bucket are not encoded in the histogram directly. Instead, three properties provide the context needed to determine the bucket boundaries and the count each bucket contains: schema, spans, and bucket counts.

Schema

A native histogram's schema is used to determine the boundaries of the buckets, and allows for defining the precision/resolution of the buckets. A lower resolution has larger buckets (larger gap between the lower and upper boundaries of each bucket), and a higher resolution has smaller buckets (smaller gap between the lower and upper boundaries of each bucket). The higher the resolution, the higher the precision.

Currently accepted values for the schema are integers between -4 and 8. The integer schema corresponds to a predefined table of bucket boundaries defined here, so the only thing required is to specify a factor that you want to multiply each bucket boundary by; there is no need to specify the exact boundaries of the buckets.

The central idea behind the schema and how it relates to the bucket layout is that we always start with a bucket with an upper boundary of 1. This is referred to as Bucket 0. Given the baseline of starting at an upper boundary of 1, then each upper boundary of the following bucket will be calculated by multiplying the previous bucket's upper boundary by 2^(2^-n), where n refers to the schema value.

The larger the schema (8 being the highest possible value currently), the higher the resolution, and the smaller the distance between the lower and upper boundary of each bucket.

With a schema of 0, using the formula 2^(2^-0) results in 2^1. A bucket boundary of 2^1 means that, starting at bucket 0 with an upper boundary of 1, we can calculate bucket 1's upper boundary as:

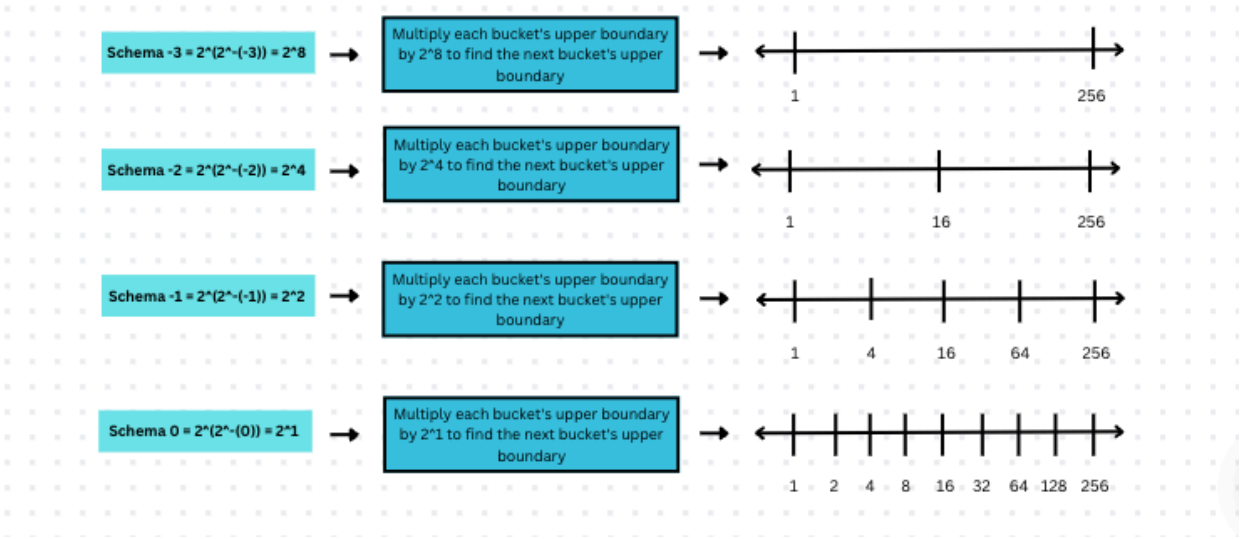
1 * 2^1 = 2, meaning bucket 1 has a lower boundary of 1 (because that was the previous bucket's upper boundary), and an upper boundary of 2.

To calculate bucket 2's upper boundary, we take bucket 1's upper boundary and multiple it by 2^1:

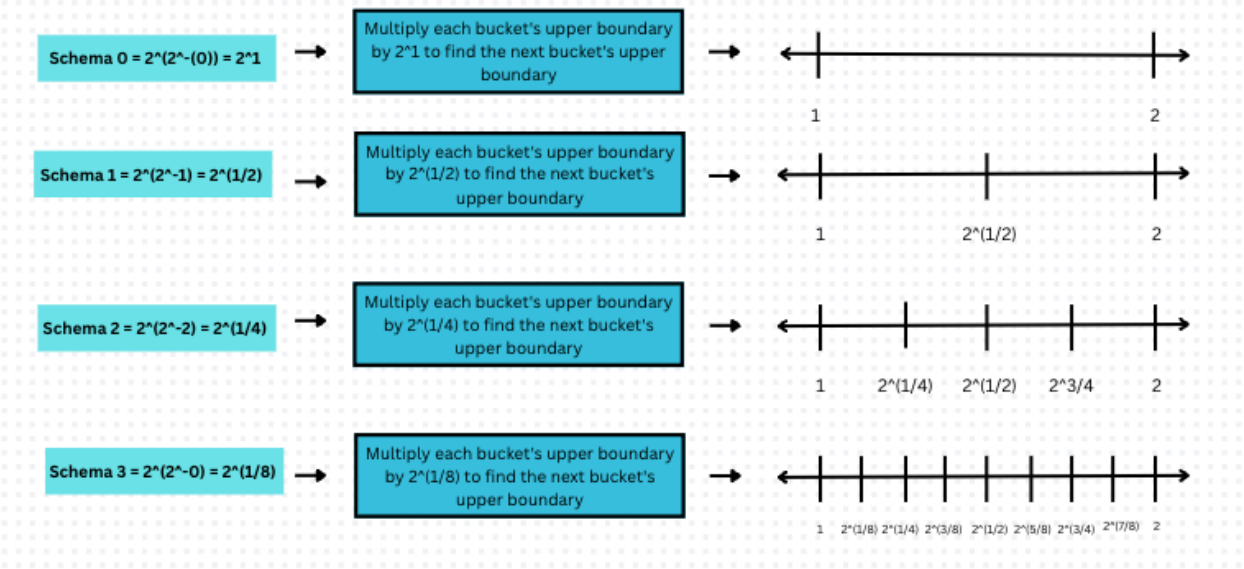
2 * 2^1 = 4, meaning bucket 2 has a lower boundary of 2, and an upper boundary of 4.

And so on.

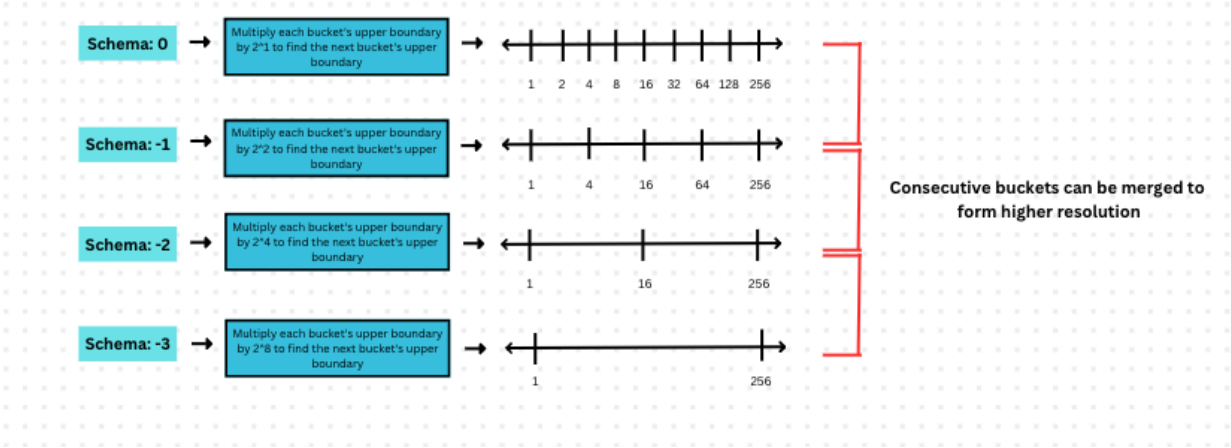
The smallest schema value produces the widest buckets. As the schema value is increased, the width of the buckets decrease:



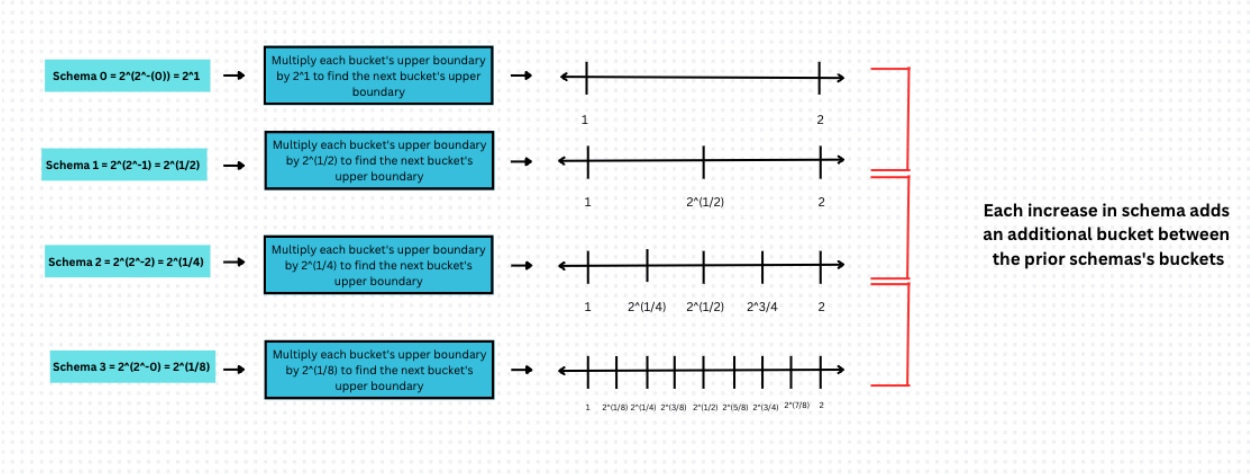
As the schema value increases, the width of the buckets decrease, which increases the precision:



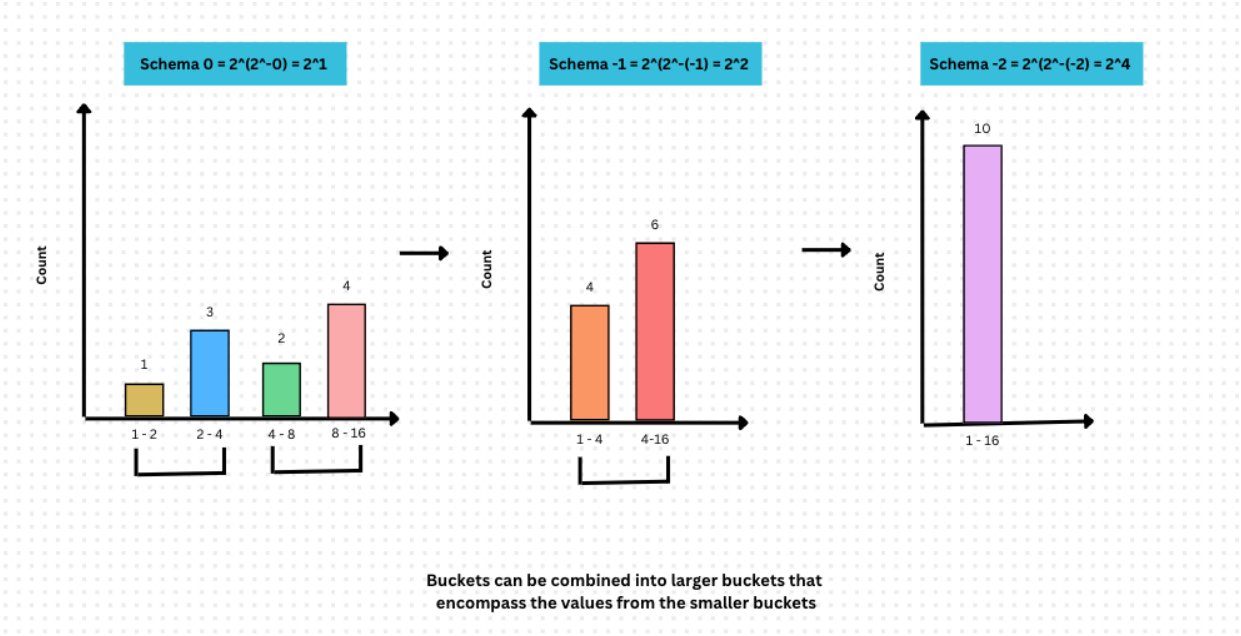
The reasoning for this bucket layout is that there are common buckets between the different bucket layouts with the different schemas. This allows for the possibility of moving from a higher resolution layout (smaller gaps between lower and upper boundaries of each bucket) to a lower resolution layout (larger gaps between the lower and upper boundaries of each bucket).



With schemas above 0, each increase in schema adds an additional bucket between the boundaries in the previous schema's bucket layout:

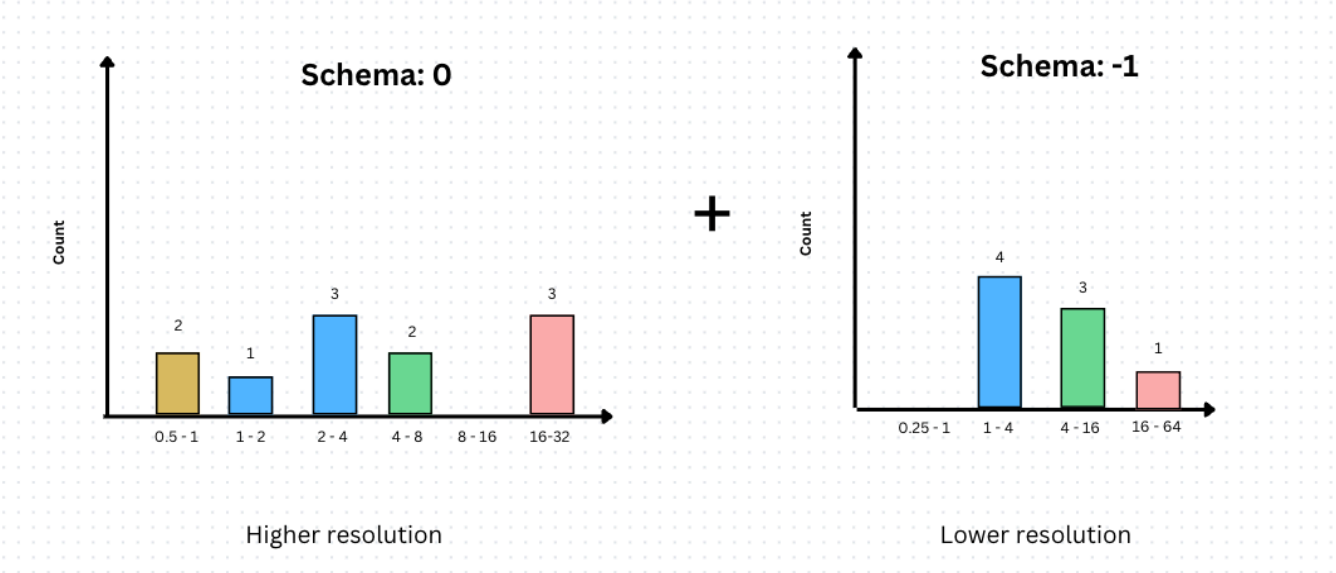


This property is generally referred to as perfect subsetting, and it allows for changing a histogram from a higher resolution to a lower resolution:



Besides being able to lower the resolution of a single histogram, this property also allows for the ability to combine two histograms with different resolutions. However, it is only possible to convert from a higher resolution to a lower resolution (i.e making the buckets larger, not smaller).

Consider adding together the following two histograms, one with a schema value of 0 and the other with a schema value of -1:



In order to combine them, the lower resolution (wider buckets) must be used. The resulting histogram will then have the lower resolution, with a schema of -1.

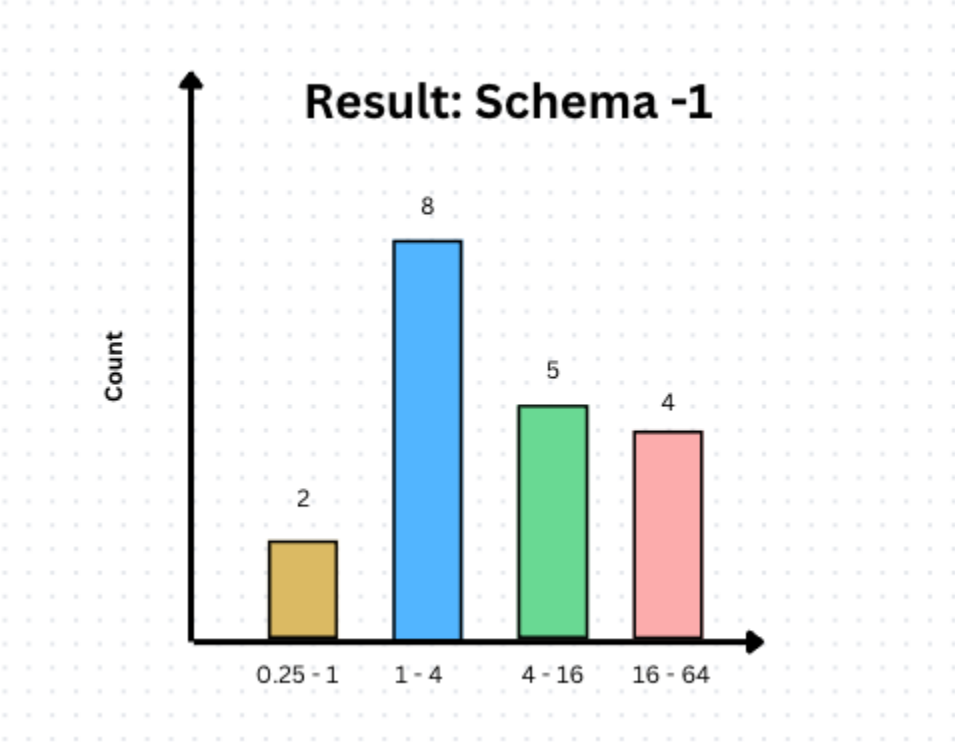
For the first histogram, the first bucket has a count of 2 within the range of 0.5 and 1. The second histogram has a count of 0 within the range 0.25 - 1. Since the second histogram's bucket is wider, we can add the counts between both into the 0.25 - 1 bucket.

Then, the first histogram has a count of 1 in the 1 - 2 bucket and a count of 3 in the 2 - 4 bucket. The second histogram has a count of 4 in the 1 - 4 bucket. We can combine the two buckets of the first histogram into a count of 4, then add that to the count in the 1 - 4 bucket of histogram 2, for a total count of 8.

Then, the first histogram has a count of 2 in the 4 - 8 bucket, and a count of 0 in the 8 - 16 bucket. The second histogram has a count of 3 in the 4 - 16 bucket. We can combine the two buckets of the first histogram into a count of 2, then add that to the second histogram's count of 3, for a total count of 5 in the 4 - 16 bucket.

Finally, the first histogram has a count of 3 in the 16 - 32 bucket, and no further buckets. The second histogram has a count of 1 in the 16 - 64 bucket. We can combined these values for a total of 4 in the 16 - 64 bucket.

The resulting histogram looks like this:



This property is useful when adding, subtracting, multiplying or dividing native histograms.

Spans

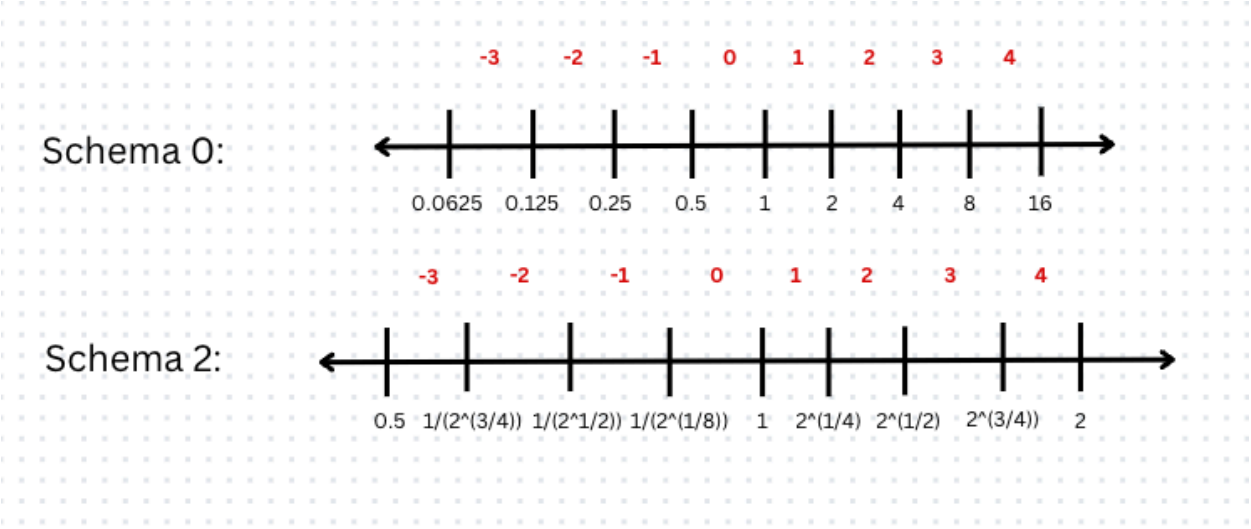
Spans are used as an efficient way of representing the layout of the buckets in the native histogram. They depict buckets that are non-empty, and this prevents having to store details about empty buckets. This is helpful to prevent the performance and memory issue of storing empty buckets. The data type is defined as follows:

```
type Span struct {
   Offset int32
   Length uint32
}
```

The offset represents:

- The starting index for the first span (which can be negative)
- The gap to the previous span (for every span after the first). In this case, it must be a positive integer

The starting index of the bucket for the first span can be negative because of how bucket indices are defined: bucket 0 always has an upper boundary of 1. Depending on the schema, there are other buckets before bucket 0.



Because the 0th bucket must always have an upper boundary of 1, there can be other buckets that have an index below 0 but are larger than the range of the ZeroCount bucket.

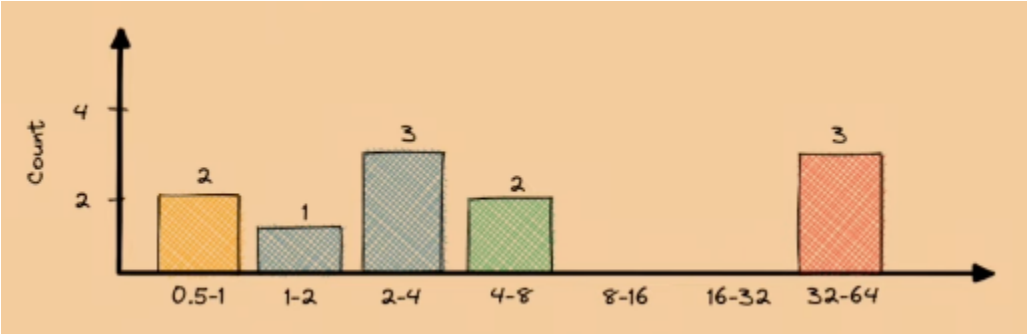
The length represents how many consecutive buckets there are in this span, starting at the bucket index.

There are both positive and negative spans to correspond to the counts in the positive and negative buckets.

Here are some examples of what spans look like:

[ ]Span{
    {Offset: 0, Length: 4},
    {Offset: 2, Length: 1}
}

This indicates that starting at index 0, there are 4 buckets that contain values. Then there is a gap of 2 buckets, then there is another used bucket.



[ ]Span{
    {Offset: -2, Length: 2},
    {Offset: 2, Length: 3}
}

This indicates that there are 2 buckets, starting at index -2, then a gap of 2, then there are 3 more buckets that are populated with values.

Buckets

In a native histogram, buckets are represented as a list of numbers (integers for integer histograms, and floats for float histograms). There are two separate lists for positive and negative buckets. Negative buckets are used to store negative values; however, the boundaries of negative buckets are expressed as absolute values.

There is a major difference between how buckets are represented between regular and float histograms. For float histograms, the values in the buckets are absolute counts, and must be 0 or positive. In integer histograms, the values in the buckets are deltas between buckets (i.e. a

bucket list of [1, 2, -2] would correspond to a bucket count of 1 in the first bucket, 3 in the second bucket, and 1 in the third bucket).

Buckets are spaced logarithmically, and the bucket boundaries are guaranteed at powers of 2. The indexing of the buckets is based around the idea that the 0th bucket must have an upper boundary of 1 (or a lower boundary of -1 for negative buckets). This means that, depending on the schema/bucket layout, it is possible for the bucket index to be a negative number.

Note that when referring to a bucket boundary, it is referring to the upper boundary of the bucket. The lower boundary can be determined by the upper boundary of the previous bucket.

There is also a separate bucket, represented as ZeroCount, that holds the counts of values that are at or close to 0. The range of values that will be included in the zero bucket is defined by the ZeroThreshold.

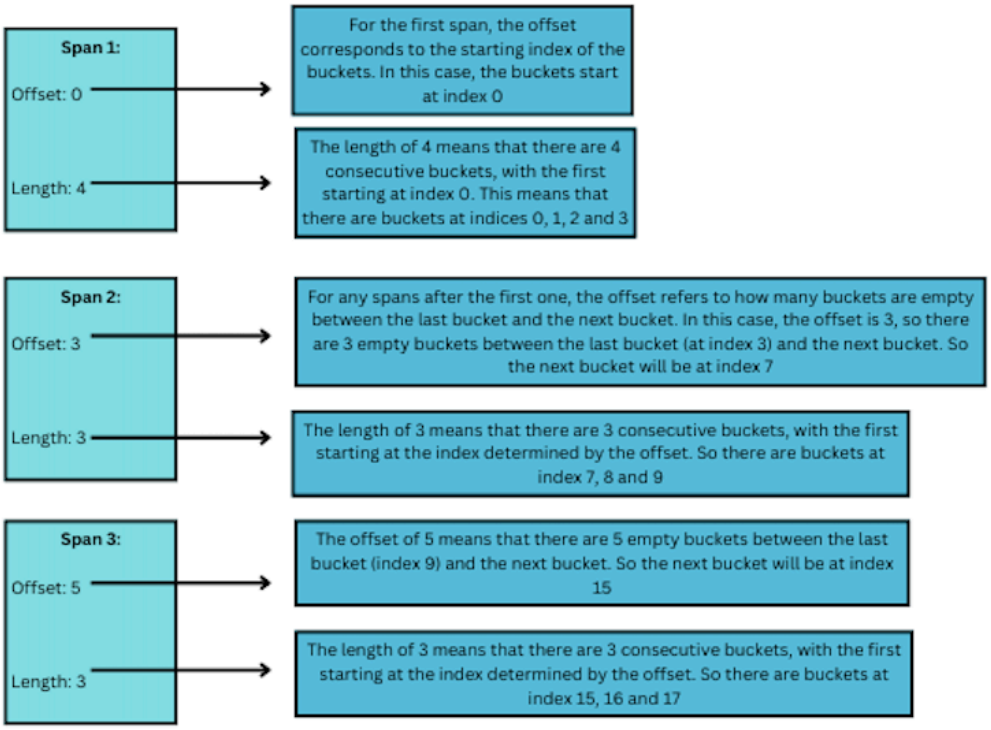
The indices of the buckets are defined in the spans. Here is an example of how this works:

Suppose that we have a native histogram with a list of positive spans:

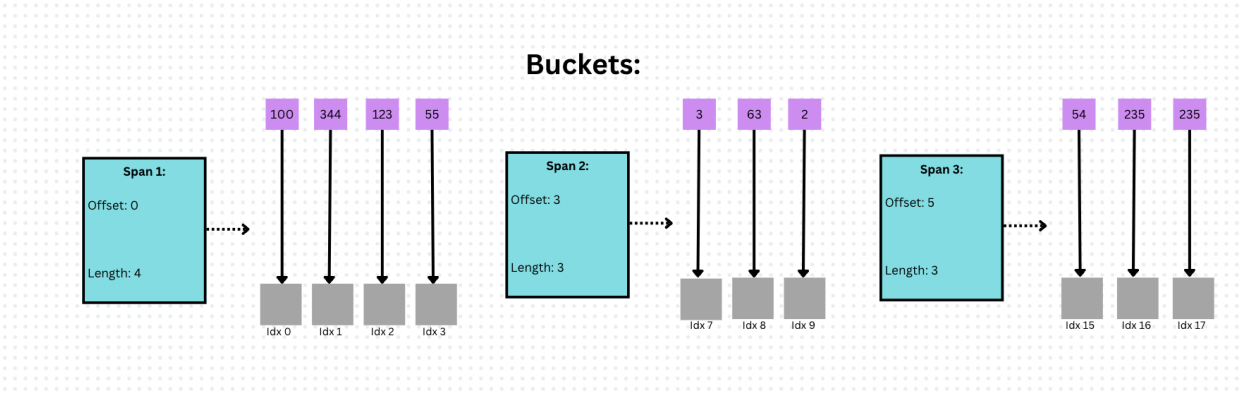
PositiveSpans:
{
    {Offset: 0, Length: 4},
    {Offset: 3, Length: 3},
    {Offset: 5, Length: 3},
}

And we have a list of positive buckets:

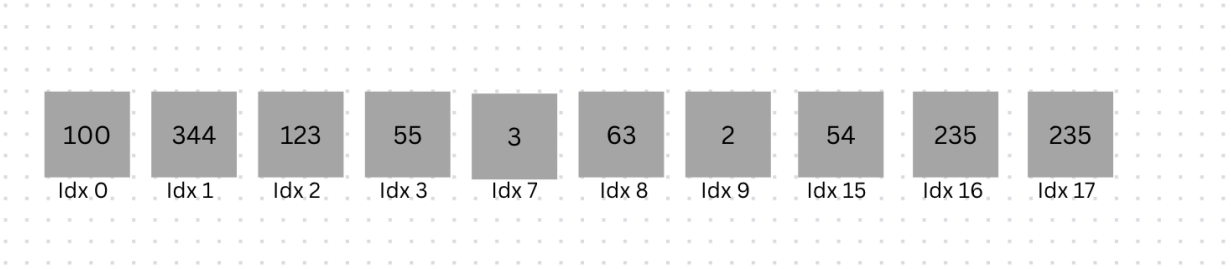
PositiveBuckets: [100, 344, 123, 55, 3, 63, 2, 54, 235, 33]



This shows how the spans' offsets and lengths are mapped to the bucket indices:



Resulting in the following buckets:



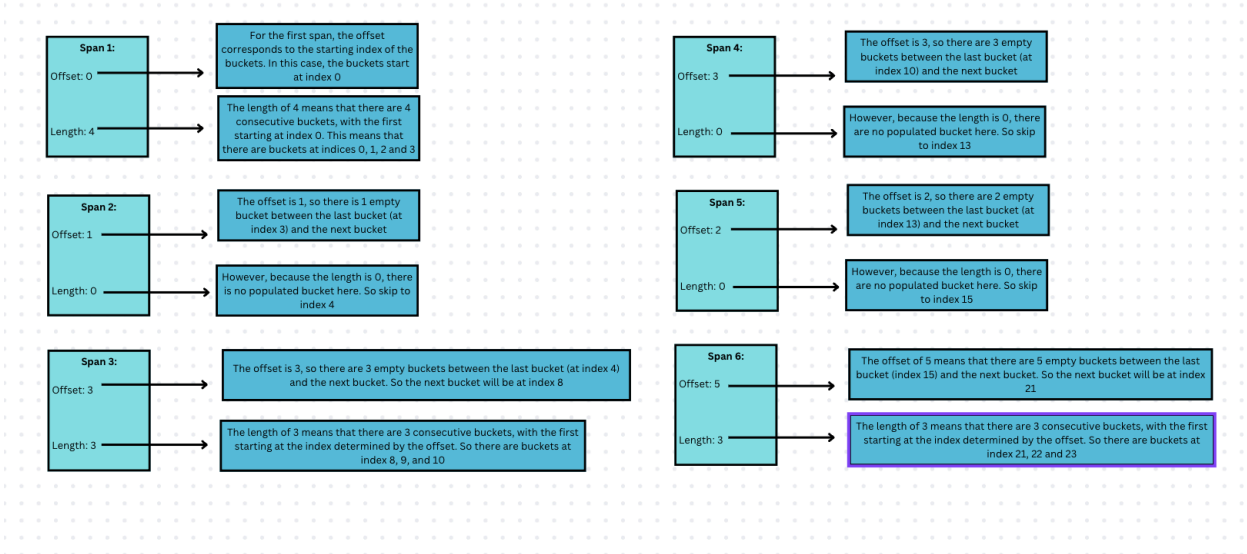
It is also possible for spans to have an offset above 0, but a length of 0. This corresponds to buckets that don't contain any values. This can happen when adding and subtracting histograms. The buckets are mapped in the same way: use the offsets and lengths to determine how many empty buckets there are between populated buckets.

Consider the same example, except there are some spans with a length of 0:

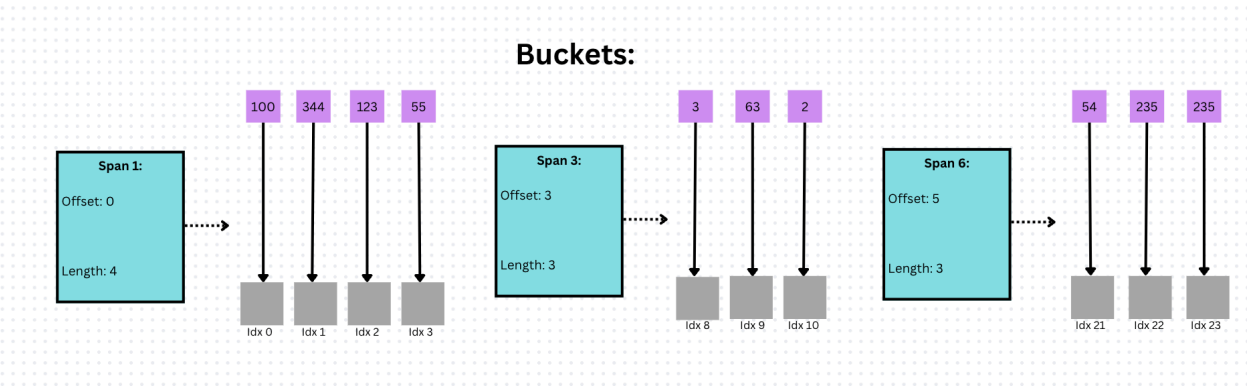
PositiveSpans:
{
    {Offset: 0, Length: 4},
    {Offset: 1, Length: 0},
    {Offset: 3, Length: 3},
    {Offset: 3, Length: 0},
    {Offset: 2, Length: 0},
    {Offset: 5, Length: 3},
}

The list of positive buckets remains the same:

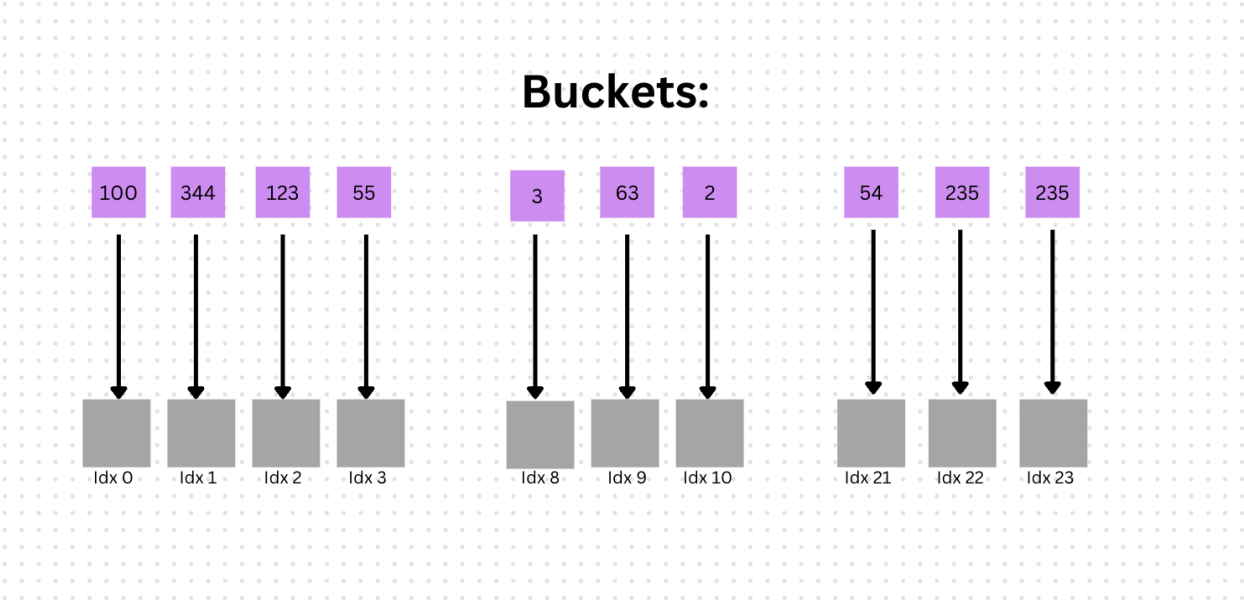
PositiveBuckets: [100, 344, 123, 55, 3, 63, 2, 54, 235, 33]



This shows how the spans' offsets and lengths are mapped to the bucket indices, and how empty buckets being represented in the spans are ignored:



Resulting in the following buckets:



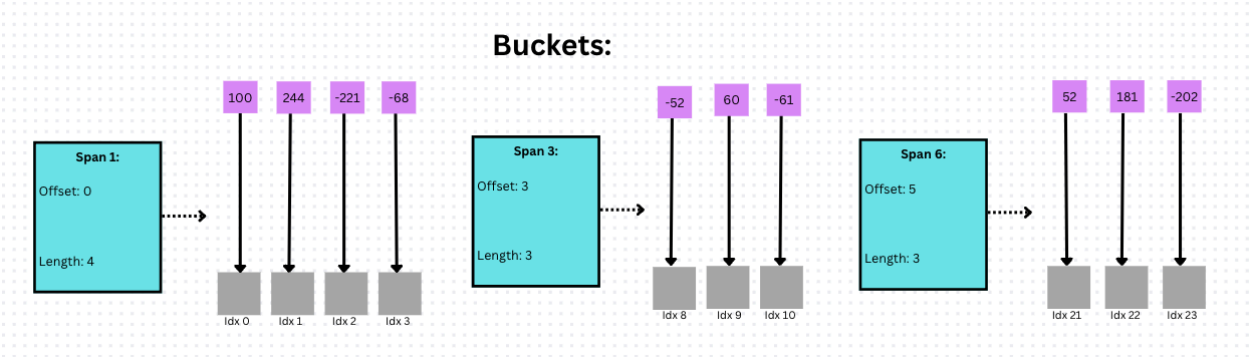
Integer histograms

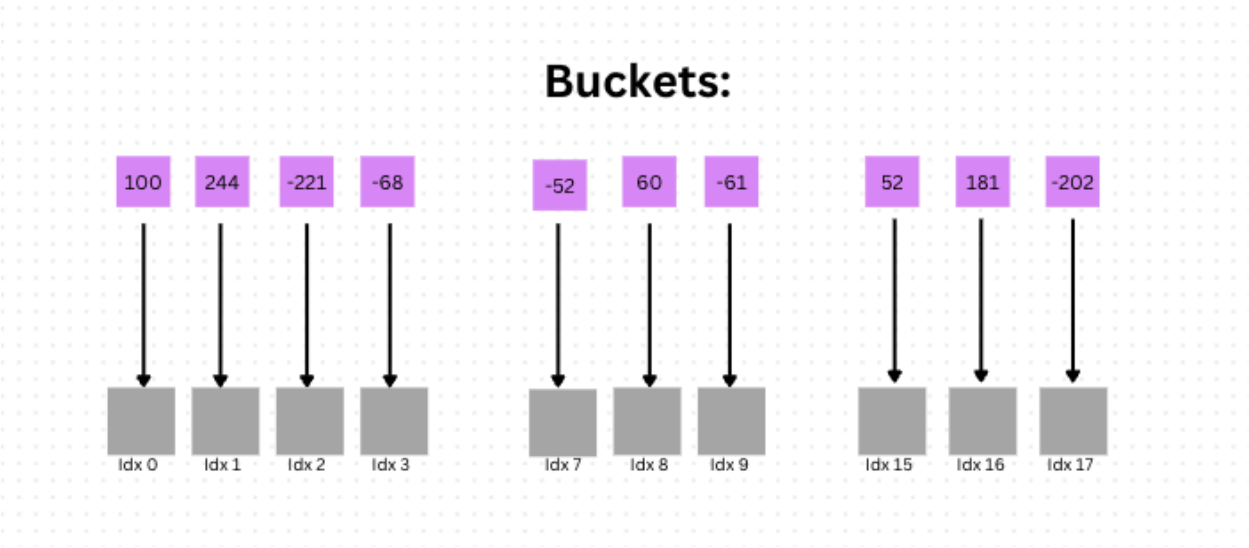
For integer histograms, bucket values are stored differently, because they are represented as the delta between the buckets.

In the example above, the buckets would be described as:

[100, 244, -221, -68, -52, 60, -61, 52, 181, -202]

The logic of the spans and list of bucket counts is the same, other than the counts being stored as deltas. So if the same spans were representing a integer histogram's buckets with the same absolute counts, the result histogram's bucket counts would be represented as:





This can also be written as:

Bucket 0: 100
Bucket 1: [Bucket 0] + 244 = 344
Bucket 2: [Bucket 1] - 221 = 123
Bucket 3: [Bucket 2] - 68 = 55
Bucket 4: [Bucket 3] - 52 = 3
Bucket 5: [Bucket 4] + 60 = 63
Bucket 6: [Bucket 5] - 61 = 2
Bucket 7: [Bucket 6] + 52 = 54
Bucket 8: [Bucket 7] + 181 = 235
Bucket 9: [Bucket 8] - 202 = 33

Codebase layout

Model

The models of native histograms exist here:

- prometheus/model/histogram
    - generic_histogram.go - contains structures and functions that are shared between both regular and float native histograms, such as the the exponential bounds related that are mapped to the schema value, base code for bucket iterators, and bucket compaction
    - histogram.go - contains definition of Histogram, and functions related to it, including more specific iterators for integer buckets, and functions for converting a integer histogram into a float histogram
    - float_histogram.go - contains definition of FloatHistogram, and functions related to it, including more specific iterators for float buckets, and functions for adding, subtracting, multiplying and dividing float histograms

PromQL

- prometheus/promql
    - functions.go - contains functions for performing operations on various data types, including histograms
    - quantile.go - contains functions for calculating quantiles of histograms, as well as helper functions

TODO: add information about TSDB / storage of native histograms

Deep-dive into the code

Schema and Bucket boundaries

The integer schema is used to find the upper boundary of a bucket. The predetermined boundaries are defined here.

getBound()

To find the boundary of a particular bucket, the getBound function is used. Bucket boundaries are defined by the upper boundary; the upper boundary is found by calling the getBound function on the bucket's index, while the lower boundary can be found by calling the getBound function on the bucket's index-1.

The predefined boundaries are defined as a 2D array. A bucket's boundary is found by indexing the 2D array with the schema and the fractional index based on the bucket index.

```
func getBound(idx, schema int32) float64 {  8 usages  new *
    if schema < 0 {
        exp := int(idx) << -schema
        if exp == 1024 {
            // This is the last bucket before the overflow bucket
            // (for ±Inf observations). Return math.MaxFloat64 as
            // explained above.
            return math.MaxFloat64
        }
        return math.Ldexp( frac: 1, exp)
    }

    fracIdx := idx & ((1 << schema) - 1)
    frac := exponentialBounds[schema][fracIdx]
    exp := (int(idx) >> schema) + 1
    if frac == 0.5 && exp == 1025 {
        // This is the last bucket before the overflow bucket (for ±Inf
        // observations). Return math.MaxFloat64 as explained above.
        return math.MaxFloat64
    }
    bound := math.Ldexp(frac, exp)
    return bound
}
```

Attempting to break down this code:

- `fracIdx := idx & ((1 << schema) - 1)`:
    - This takes the bucket index and keeps only the lower schema bits using bitwise AND operator and bit masking. The result essentially isolates the fractional portion of the bucket index
- `frac = exponentialBounds[schema][fracIdx]`
    - This code looks up the value in exponentialBounds, which is a 2D array where the first index refers to the schema and the second index is the fractional portion of the bucket index.
- `exp = (int(idx) >> schema) + 1`
    - This shifts the integer portion of the bucket index to the right using the bits from the schema, which is equivalent to reducing the magnitude of the index. Incrementing by 1 moves it to the next bucket
- `math.Ldexp(frac, exp)`

- Performs the mathematical formula of `frac * 2^exp` which results in returning the previous bucket boundary * 2^(2^-n)

In summary, this code uses bit manipulation to isolate the fractional portion of the bucket index, retrieves the correct multiplier for the boundary from the predefined exponentialBounds array, calculates a reduction factor and then calculates the bucket's upper boundary according to the normal of boundary * 2^(2^-n)

Buckets

When storing the native histogram, buckets are represented as a list of integers (integer histograms) or floats (float histograms), and combined with the schema and list of spans, more details about the bucket layouts can be determined. But when performing various operations involving the buckets, buckets are represented by a Bucket type:

```
type Bucket[BC BucketCount] struct {  56 usages  new *
    Lower, Upper                   float64
    LowerInclusive, UpperInclusive bool
    Count                          BC

    // Index within schema. To easily compare buckets that share the same
    // schema and sign (positive or negative). Irrelevant for the zero bucket.
    Index int32
}
```

BucketCount is either a float or an integer depending on the type of native histogram. The same is true for Index.

The index is used to easily compare buckets that share the same schema and sign (positive or negative).

The Lower and Upper variables are used to refer to the upper and lower boundaries of the bucket. LowerInclusive and UpperInclusive refer to whether the bucket boundaries are inclusive or not, with inclusive creating a closed interval, and non-inclusive creating an open interval.

There are several different bucket iterators to iterate over the buckets. Both regular and float native histograms share a baseBucketIterator, and then there are other types of iterators, such as regularBucketIterator and floatBucketIterator, which use the baseBucketIterator but allow for the handling of the different data types for fields in the two different histogram types.

```
type baseBucketIterator[BC BucketCount, IBC InternalBucketCount] struct {
    schema  int32
    spans   []Span
    buckets []IBC

    positive bool // Whether this is for positive buckets.

    spansIdx   int    // Current span within spans slice.
    idxInSpan  uint32 // Index in the current span. 0 <= idxInSpan < span.Length.
    bucketsIdx int    // Current bucket within buckets slice.

    currCount IBC   // Count in the current bucket.
    currIdx   int32 // The actual bucket index.
}
```

For integer histograms, there are also:

- PositiveBucketIterator and NegativeBucketIterator, for iterating over all positive and all negative buckets, respectively. PositiveBucketIterator starts at the zero bucket and iterates up, and NegativeBucketIterator starts the the zero bucket and iterates down
- CumulativeBucketIterator - currently only used for testing. It iterates over a cumulative view of the buckets. It only works on integer histograms that do not have negative buckets

For float histograms, there are also:

- PositiveBucketIterator and NegativeBucketIterator, for iterating over all positive and all negative buckets, respectively. PositiveBucketIterator starts at the zero bucket and iterates up, and NegativeBucketIterator starts the the zero bucket and iterates down
- PositiveReverseBucketIterator - iterates over all of the positive buckets, starting at the highest bucket and iterating down towards zero
- NegativeReverseBucketIterator - iterates over all of the negative buckets, starting at the lowest bucket and iterating up towards zero
- AllBucketIterator - iterates over all buckets (negative, zero, and positive), starting at the lowest bucket and going up. This is useful for calculating the quantile of the histogram, if the p value is less than 0.5.
- AllReverseBucketIterator - iterates over all buckets (negative, zero, and positive), starting at the highest bucket and going down. This is useful when calculating quantiles with a p value of 0.5 or higher for increased efficiency

When iterating through buckets, it is important to note:

- spansIdx - refers to the index of the current span being processed amongst the native histograms positive or negative spans that is currently being processed
- idxInSpan - refers to the current index being processed within the current span
- bucketsIdx - refers to the index of the current bucket that is being processed within the current current index of the current span

The basic functions of the baseBucketIterator are

- At() - which returns the bucket at the current index
- Next() - advances the iterator by one
- getBound function - returns the upper bound of the bucket.
- compactBuckets - trims empty buckets at the beginning and end of spans, merges consecutive spans, and split spans that contain consecutively empty buckets

Next()

All bucket iterators have a Next() function, which advances the iterator by one. More specifically, it keeps track of the index of the current bucket, within the current span, within the list of all spans (either positive or negative). It updates the index of the current bucket (bucketIdx), updates the index in the current span (idxInSpan), if needed, and updates the spansIdx (if needed).

For example, if considering the following list of positive spans:

<insert example>

Additionally, if changing between a higher resolution to a lower resolution, the Next() function will merge buckets together to fit into the new schema/resolution (see examples in Schema overview section).

At()

The At() function simply returns the current bucket. The current bucket is updated by the Next() function.

compactBuckets()

This function is shared between both regular and float histograms. It contains code to distinguish between the counts contained in the buckets of integer histograms (which are deltas and not absolute counts) versus float histograms (which are absolute values). Integer histograms and float histograms have their own Compact() functions, which call this base function with the boolean denoting whether the bucket counts are delta or absolute values.

When histograms are added, subtracted, multiplied or divided, sometimes the resulting histogram contains empty buckets. This is because if the receiving histogram does not have

buckets that are contained within the other histogram, these buckets are still inserted into the receiving histogram.

This function handles the merging of buckets and handling of empty buckets, including removing all buckets at the start and end of spans.

Histogram aggregation functions

TODO: include information on the following functions

Add()

See TODO

Sub()

See TODO

Mul()

See TODO

Div()

See TODO

Differences with Mimir Native Histograms

There are some differences between how Native Histograms are implemented in Mimir versus how they are implemented in Prometheus.

- In Prometheus, the bucket counts for both regular and float histograms are stored in the PositiveBuckets and Negative buckets fields. integer histograms store the bucket counts as a list of integers, and float histograms store bucket counts as a list of floats. In Mimir histograms, there is not a differentiation between float and integer histograms: there is just one histogram type. Each Mimir Histogram has variables for

PositiveDeltas/NegativeDeltas and PositiveCounts/NegativeCounts. When representing a integer histogram, the PositiveDeltas/NegativeDeltas fields are used, and when representing a float histogram, the PositiveCounts/NegativeCounts fields are used.