FLEXIBLE COMPUTATION OF MULTIDIMENSIONAL HISTOGRAMS

Samriddhi Singla
Computer Science and Engineering
University of California, Riverside
ssing068@ucr.edu

Ahmed Eldawy
Computer Science and Engineering
University of California, Riverside
eldawy@ucr.edu

ABSTRACT

Histograms are a popular method to understand spatial data distribution and help facilitate query optimization, approximate query processing, and load balancing. The most common computation method for histograms requires two passes over the data, first to compute the data domain and then to compute the histogram values. This gem presents an alternative method that can compute an approximate histogram in one pass over the data, which makes it more suitable for incremental computation and streaming applications where two passes over the data are not possible. It can also be more efficient if reading the input data is costly, e.g., requires decompression. The key algorithm that makes the one-scan algorithm possible is for merging two non-aligned histograms.

1 Introduction

Numerous query optimization and data summarization methods use histograms. The histogram is used mainly for selectivity estimation [1], which can be used to balance the load across partitions [2, 3, 4] or to choose an efficient algorithm by the query optimizer. The histogram also gives a summary of the data that can be used for approximate query processing [5].

Computing multi-dimensional histograms normally requires two scans of the data. The first scan computes the minimum bounding box (MBB) of the data and the bucket boundaries while the second scan assigns the data points to their respective buckets. Some techniques assume knowledge of the input space, e.g., the entire world, to avoid the first scan, but this is not always known, especially for multi-dimensional histograms that include various attributes. For some applications, two scans of the data could be very expensive or sometimes impossible, e.g., streaming applications.

In this gem, we provide an algorithm and a reference implementation for histogram computation that enables the computation of approximate multidimensional histograms in only one scan. The key idea is to compute a partial histogram for each batch of the data and then merge these batches to compute the final histogram. For example, in the case of streaming applications, a partial histogram can be computed for each batch of data that arrives and then merged with the existing histogram for the previous data. The challenge is that the partial histograms are not necessarily aligned since each batch might cover a different region of the input, i.e., each batch could have a different MBB. To address this challenge, we propose a sort-merge-like algorithm that can merge two non-aligned histograms. We use this merge algorithm as a building block for the flexible computation of multidimensional histograms.

Applications for this new algorithm are numerous. First, for regular batch processing systems, this new algorithm can save time by scanning the data only once, which would be more efficient for big data. Second, in big data systems that store the data in fixed-size blocks, e.g., in Hadoop Distributed File System (HDFS) and in Log-structured-merge trees (LSM) [6], the partial histogram can be attached to each fixed block, and then the histograms can be merged lazily when accessed. Third, in continuous and micro-batch streaming applications when the input domain is not known, the proposed technique can be used to update the histogram and occasionally expand its topology, i.e., range and number of buckets, without having to recompute it entirely.

Below, Section 2 describes the proposed algorithm. Section 3 gives a brief experimental evaluation on real datasets. Section 4 gives a final conclusion.

2 Computing Histograms

In this section, we first define the terminology that we use for histogram. For clarity, we start by providing an algorithm for merging non-aligned *one-dimensional* histograms. After that, we show how to easily extend that algorithm for multidimensional histograms.

2.1 Definitions

Definition 1 (Point) *A d-dimensional point p is represented by a tuple in the space $p \in \mathbb{R}^d$. The individual dimensions are denoted by $p_1, p_2, \cdots, p_d$.*

Definition 2 (Bounding Box) *A bounding box is a multidimensional range defined by two corner points l and u such that $l_k \leq u_k \forall\ 1 \leq k \leq d$. In this paper, we assume that the bounding box is inclusive of the lower corner and exclusive of the upper corner. In other words, a point $p \in BB \Leftrightarrow l_k \leq p_k < u_k \forall k \in [1, d]$.*

Definition 3 (Histogram) *A histogram H is defined by a bounding box $H.BB$ and a number of partitions along each dimension, $H.n \in \mathbb{N}^d$. The histogram H divides the range $H.BB$ into N buckets, $N = \prod_{k \in d} n_k$. Each bucket has a unique ID, $i \in \mathbb{Z}^d$ such that $0 \leq i_k < n_k$.*

Definition 4 (Bucket Bounding Box) *A bucket i in a histogram H has a bounding box defined by two corner points $l(H, i)$ and $u(H, i)$ that can be calculated as follows where the k subscript denotes the k^{th} dimension of each corner.*

$$\begin{aligned} l(H, i_k)_k &= \frac{H.l_k(H.n_k - i_k) + H.u_k \cdot i_k}{H.n_k} \\ u(H, i_k)_k &= l(H, i_k + 1)_k \end{aligned} \tag{1}$$

Definition 5 (Bucket Value) *Each bucket i in a histogram H is associated with a real number, denoted $H.v[i] \in \mathbb{R}$, that represents its value in the histogram. Typically, this value represent the number of input records that are inside the bucket bounding box.*

Definition 6 (Histogram Merging Problem) *The problem of histogram merging involves two histograms, a source histogram S and a target histogram T. The problem is to update the bucket values in the target histogram to reflect the values in the source histogram.*

In the histogram merging problem, if the two histograms are perfectly aligned, i.e., have the same bounding box BB and number of partitions n, then the problem is trivial since there is a one-to-one-correspondence between buckets. If the histograms are not aligned, then we show how to merge them in the following part.

2.2 Merging Non-aligned One-dimensional Histogram

The implemented algorithm can work with any number of dimensions, but for the sake of simplicity, we first explain the method to merge histograms having only one dimension. We then extend the algorithm to multiple dimensions in the following section.

This algorithm merges two non-aligned histograms based on an idea similar to that of a sort-merge algorithm. Histograms are divided along each dimension into buckets, and each of these buckets have a bounding box defined by two corner points $l(H, i)$ and $u(H, i)$ along each dimension. The range defined by these corner points along each dimension is mutually exclusive for any two buckets in the same histogram. These corner points also impose a row-order-like sort on the histogram buckets. For example, in case of a 1-D Histogram, as shown in Figure 1, the first bucket for the source histogram S has corner points $l(S, 0) = 4$ and $u(S, 0) = 6$, which have a value less than the respective $l(S, i)$ and $u(S, i)$ of the other buckets. The algorithm uses this sort order to assign a identifier to each bucket as well as to iterate over buckets in both the source and target histograms. It starts by comparing the first buckets for both histograms and then only the index of the bucket that has a lower value of upper corner point u is incremented. This reduces the number of buckets compared, and contributes to the efficiency of the algorithm. It works on the assumption that the data is uniformly distributed in each bucket and hence uses overlap volume as a weight to assign values from the bucket in source histogram to that in the target histogram. If a bucket in the target histogram overlaps with that in the source histogram, the value assigned to the target bucket will be the product of the overlap volume and source bucket value.

We present a pseudo-code in Algorithm 1 that can be used to merge two non-aligned 1-D histograms. Since there is only one dimension, we do not use subscripts to indicate the dimension. Below, we define terms used in Algorithm 1 in respect to a 1-D histogram.

Bucket Index In this implementation, we store bucket values as a one dimensional array. The number of buckets in a 1-D histogram is equal to the number of partitions. Hence, we can access each bucket value in this array with an index equal to its partition number.

$$BucketIndex(H, i) = i \tag{2}$$

Overlap The overlap for a source bucket and target bucket is ratio of the common length between the two buckets to the length of the source bucket and can be computed as:

$$Overlap(i, j) = \frac{Min(u(S, i), u(T, j)) - Max(l(S, i), l(T, j))}{u(S, i) - l(S, i)} \tag{3}$$

The Algorithm 1 starts by initializing iterators, i and j, to the first partition in source and target histograms, respectively as can be seen in **lines 1 - 2**. Since, the histograms have only one dimension, these iterators to the partitions can also be used to iterate over the buckets. The while loop in **lines 3- 10** allows the simultaneous iteration over buckets in both the histograms. The loop terminates when all the partitions in either of the histograms have been accessed. The first step in the loop is to calculate the *overlap* between the current source and target buckets. Once this is calculated, a value equal to the product of the *overlap* and the value of the source bucket $S.v[i]$ is added to the value of the target bucket. After this, the iterator to either the source or target partitions needs to be incremented. This is decided based on the current buckets' corner point u. The iterator belonging to the partition whose bucket has a lower u is incremented.

Algorithm 1: MergeNonAlignedHistograms(1-D)

Input: S: Source Histogram, T: Target Histogram

```
1  i = 0
2  j = 0
3  while i < S.n and j < T.n do
4      overlap = (Min(u(S,i),u(T,j)) − Max(l(S,i),l(T,j))) / (u(S,i) − l(S,i))
5      if overlap > 0 then
6          T.v[j] += overlap × S.v[i]
7      if u(S,i) < u(T,j) then
8          i++
9      else
10         j++
```

We now walkthrough the pseudo-code in Algorithm 1 using the example in Figure 1. The algorithm takes as input a source histogram S and target histogram T. As can be seen in the figure, the source histogram S and the target histogram T have five partitions/buckets each. The buckets for the source histogram have values assigned to them as can be seen in the figure, while those for target histogram will be calculated using Algorithm 1.

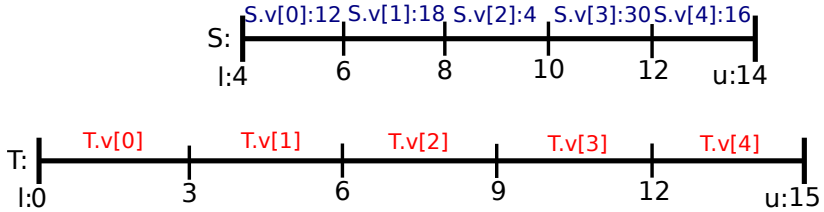


Figure 1: Example for 1-D Histograms

According to **lines 1 - 2**, we start at partition 0 for both source histogram and target histogram ($i = 0, j = 0$). The bucket values for target histogram are $\{T.v[0] = 0, T.v[1] = 0, T.v[2] = 0, T.v[3] = 0, T.v[4] = 0\}$. In the first iteration, we calculate the overlap between source bucket 0 and target bucket 0, which is equal to zero as can be seen in Figure 1. Therefore, the value assigned to $T.v[0] = 0$. Now we need to increment iterator to either source or target partitions. The target bucket has a $u(T, 0) = 3$, which is less than $u(S, 0) = 6$ for the source bucket. Hence, the iterator to the target partition is incremented. The partitions in the next iteration are: $i = 0$ and $j = 1$. In this iteration, the source bucket overlaps completely with the target bucket, hence $T.v[1] = 12$. Also, since both buckets have the same corner point u, iterator to the target partition is incremented. This goes

on and in the last iteration, the current partitions are $i = 4$ and $j = 4$, and the target histogram values are $\{T.v[0] = 0, T.v[1] = 12, T.v[2] = 20, T.v[3] = 32, T.v[4] = 0\}$. Since target bucket 4 completely overlaps with the source bucket 4, $T.v[4] = 16$. Since all partitions in both the histograms have been accessed, the while loop terminates. The target bucket values at this point are $\{T.v[0] = 0, T.v[1] = 12, T.v[2] = 20, T.v[3] = 32, T.v[4] = 16\}$, as shown in Figure 2.

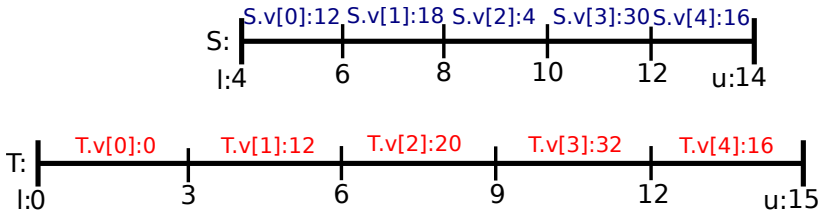


Figure 2: Final values for 1-D Histograms example

2.3 Implementation for Multidimensional Histograms

In this section, we extend the algorithm described in the previous section to work with multidimensional histograms. There are some key differences in how partitions and buckets are accessed in the implementation, which are described below.

Accessing Partitions and Buckets In multidimensional histograms, there are n_k partitions across each dimension k. Unlike a 1-D histogram, bucket i in a multidimensional histogram is identified by a combination of partition numbers, $i = \{i_1, i_2, \ldots, i_d\}$, where i_k is the partition number along the k^{th} dimension. Therefore, in the implementation for merging non-aligned multidimensional histograms, we use d-dimensional arrays to store the iterators for the partitions.

Bucket Index In this implementation, we store bucket values as a one dimensional array. We map the bucket ID i to an array index as follows:

$$BucketIndex(H, i) = \sum_{k=d}^{1} \left(i_k \times \prod_{m=k-1}^{1} n_m \right) \tag{4}$$

Overlap The overlap for a source bucket and target bucket is the ratio of the common volume between the two buckets to the volume of the source bucket and can be computed as:

$$Overlap = \prod_{k=1}^{d} \frac{Min(u(S, i_k)_k, u(T, j_k)_k) - Max(l(S, i_k)_k, l(T, j_k)_k)}{u(S, i_k)_k - l(S, i_k)_k} \tag{5}$$

Advancing Iterators: Another change for multi-dimensional histograms is how the iterators, i and j, advance after each iteration. The key idea behind the algorithm remains the same, which is to use the inherent row-major-like sorting of the buckets to iterate over them. We first increment the lowest dimensions, either $i[1]$ or $j[1]$, similar to the one-dimensional case. If we reach the last bucket of either histogram, then we reset the bucket indexes and increment the next dimension $k = 2$ in the same way. The algorithm terminates when we reach the last bucket of either histogram in the last dimension $k = d$.

The pseudo-code in Algorithm 2 takes as input a source histogram S, a target histogram T and the number of dimensions, d. It starts by initializing the all the elements of the d-dimensional arrays i and j that contain partition numbers for all dimensions to zero in **lines 1- 2**. The while loop in **lines 4- 20** loops over the number of dimensions and ensures that the partition numbers are incremented along each dimension. It terminates when all the buckets have been accessed. The loop starts by calculating the *overlap* between the current bucket in the source and target histogram. It then calculates the bucket index for source and target histograms, which is required to access bucket values of the source and target histograms. The value added to the current bucket in the target histogram is equal to the product of *overlap* and the value of the current bucket in the source histogram. The inner while loop in **lines 11- 20**, ensures that partition number only across one dimension is incremented in an iteration of the outer while loop at **lines 4- 20**. It compares the corner point u along the current dimension k for both the source and target histograms. The partition number along the dimension k for the histogram with a lower corner point u is incremented by one. If the partition number along the

current dimension k for either of the histograms exceeds the number of partitions along it, it is set to zero for both histograms. The current dimension is then incremented by one and inner while loop is repeated. The inner while loop terminates when a partition number is successfully incremented to a valid partition number. The nested while loops ensure that all partition numbers for dimensions 1 to $k-1$ are iterated over before that for the current dimension is incremented by setting the current dimension $k = 1$ at **line 9** before the start of the inner while loop.

Algorithm 2: MergeNonAlignedHistograms

Input: S: Source Histogram, T: Target Histogram, d: number of dimensions

```
1  i:Array[d] = {0}
2  j:Array[d] = {0}
3  k = 1
4  while k ≤ d do
5      overlap = getOverlap(S, i, T, j)
6      TbucketIndex = getBucketIndex(T, j)
7      SbucketIndex = getBucketIndex(S, i)
8      T.v[TbucketIndex] += overlap × S.v[SbucketIndex]
9      k = 1
10     dimensionReset = false
11     do
12         if u(S, i_k)_k < u(T, j_k)_k then
13             i[k]++
14         else
15             j[k]++
16         if i[k] ≥ S.n or j[k] ≥ T.n then
17             i[k] = j[k] = 0
18             k++
19             dimensionReset = True
20     while k ≤ d and dimensionReset;
```

3 Experimental Results

To put our method to the test, we implemented it in Spark and tested it with several big spatial datasets. The implementation can be found at: `https://bitbucket.org/eldawy/beast/` under the class *UniformHistogram*. Below, we describe the algorithms used in the evaluation. We also give a brief description of the datasets and then we present the results.

3.1 Algorithms

We used the two-pass algorithm as a baseline and implemented two alternative methods based on the proposed algorithm, namely, one-pass and $1\frac{1}{2}$-pass. Both algorithms first compute a set of partial histograms for each partition in the input.

Algorithm 3 shows the one-pass algorithm which begins with the first partial histogram (Line 1) and iterates over the remaining partial histograms (Line 2) and merge them one-by-one into the final histogram. If the partial histogram is completed contained within the final histogram, it is directly merged using the proposed method (Line 4). Otherwise, a new empty histogram is created with a bounding box that covers both histograms (Line 6) and they are both merged into it.

Algorithm 4 shows the pseudo-code of the $1\frac{1}{2}$-pass merge algorithm. It starts by computing the bounding box of all histograms in Line 1 which is then used to initialize the final histogram in Line 2. Since this final histogram contains all partial histogram, the loop will directly merge them into the final histogram using the proposed merge method in Line 4.

3.2 Datasets

We tested the proposed methods on four real datasets as detailed in Table 1. The number of partitions is an indicator of how many times partial histograms need to be merged.

Algorithm 3: One-pass Merge of Partial Histograms

Input: $\mathcal{H}$: List of partial histograms
Output: FH: One final histogram

```
FH = H.remove(0)
for PH ∈ H do
    if PH.BB ⊆ FH.BB then
        MergeNonAlignedHistograms(PH,FH)
    else
        MergedBB = FH.BB ∪ PH.BB
        MH = new histogram with MergedBB
        MergeNonAlignedHistogram(FH,MH)
        MergeNonAlignedHistogram(PH,MH)
        FH = MH
```

Algorithm 4: $1\frac{1}{2}$-pass Merge of Partial Histograms

Input: $\mathcal{H}$: List of partial histograms
Output: FH: One final histogram

```
MergedBB = ∪_{H∈H} H.BB
FH = new histogram with MergedBB
for PH ∈ H do
    MergeNonAlignedHistograms(PH,FH)
```

3.3 Results

Figure 3 shows the results of both the running time and the accuracy. In Figure 3(a), we show the overall running time on a 12 node Spark cluster. In general, the one-pass and $1\frac{1}{2}$-pass algorithms are faster than the two-pass algorithm. However, we would like to note that the main advantage is the ability to compute the histogram in a flexible way, e.g., for streaming applications, where the two-pass method is not even applicable. We also note that the $1\frac{1}{2}$-pass method is generally as fast as the one-pass method but it could be significantly slower when the number of partitions is very large, e.g., more than 3,000 partitions in LinearWater dataset.

To measure the accuracy of the calculated histograms, we use the two-pass algorithm as the baseline, hence, it always has an error of zero. The error is calculated based on the following formula:

$$error = \frac{\sum_{0 \le i < H_2.N} |H_2.bucket[i] - H_*.bucket[i]|}{\sum H_2.bucket} \tag{6}$$

Where H_2 is the two-pass histogram (baseline) and H_* is an approximate histogram. The formula computes the sum of absolute errors (SAE) and normalizes it by the sum of all values.

Figure 3(b) shows the error of the proposed method. First, we can see a big variance from 2% error up-to 30% error. One of the main driving factors of the error is the number of partitions which controls the number of non-aligned merge operations that need to be done. The error accumulates as each non-aligned partition is merged into the final histogram. We also note that the $1\frac{1}{2}$-pass algorithm is generally more accurate since all partial histograms are merged directly into the final histogram and the final histogram is never changed.

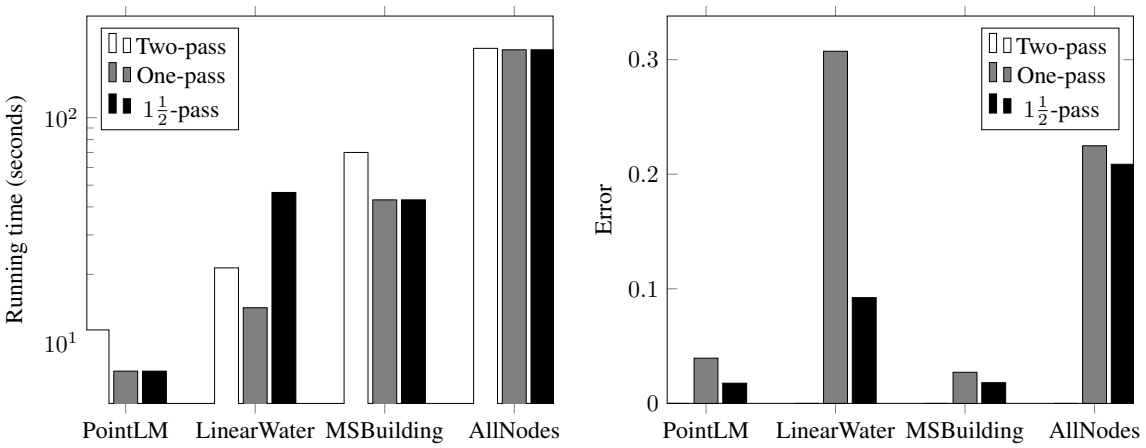
One idea that can be applied to improve the accuracy is to use larger partial histograms that could be more accurate. Then, after all the partial histograms are merged, the final histogram can then be adjusted to the desired size. An interesting problem is how big the partial histograms should be to achieve a desired accuracy without significantly slowing doing the algorithm. We leave this problem for a future work.

4 Conclusion and Future Work

This gem presented an algorithm for merging non-aligned histograms which can be used to efficiently calculate approximate multi-dimensional histograms incrementally and for streaming applications. We presented two reference implementations, one-pass merge and one-half-pass merge, and showed their efficiency and accuracy with an experi-

Table 1: Datasets used in the experimental evaluation

Dataset	Format	Size	Num Partitions
PointLM [7]	Shapefile	152 MB	58
LinearWater [8]	Shapefile	5.5 GB	3233
MSBuildings [9]	GeoJSON	28.5 GB	287
AllNodes [10]	CSV	168 GB	1599



(a) Running Time (b) Error

Figure 3: Experimental Results

mental evaluation. It was observed that these two implementations are equally efficient but one-half-pass merge is more accurate. In the future, we will study how to improve the accuracy of the result by using larger partial histograms.

5 Acknowledgements

This work is supported in part by Agriculture and Food Research Initiative Competitive Grants no. 2019-67022-29696 and 2020-69012-31914 from the USDA National Institute of Food and Agriculture.

References

[1] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMoD Record*, volume 27, pages 448–459. ACM, 1998.

[2] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

[3] Tin Vu and Ahmed Eldawy. R*-Grove: Balanced Spatial Partitioning for Large-scale Datasets. *Frontiers in Big Data*, 2020.

[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. In *ACM SIGMOD Record*, volume 30, pages 211–222. ACM, 2001.

[5] A. B. Siddique and Ahmed Eldawy. Experimental evaluation of sketching techniques for big spatial data. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2018), Carlsbad, CA, USA, October 11-13*, page 522, Carlsbad, CA, October 2018.

[6] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. Lightweight cardinality estimation in lsm-based systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 841–855, New York, NY, USA, 2018. ACM.

[7] US Census Bureau. Point landmark, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?TIGER2018/POINTLM&d`.

[8] US Census Bureau. Linear hydrography, 2019. Retrieved from UCR-STAR https://star.cs.ucr.edu/?TIGER2018/LINEARWATER&d.

[9] Microsoft. Computer generated building footprints in all 50 us states., 2020. Retrieved from UCR-STAR https://star.cs.ucr.edu/?MSBuildings&d.

[10] Ahmed Eldawy and Mohamed F. Mokbel. All points on the map as extracted from openstreetmap., 2019. Retrieved from UCR-STAR https://star.cs.ucr.edu/?OSM2015/all_nodes&d.