Abstract—The amount of data generated and stored in cloud systems has been increasing exponentially. The examples of data include user generated data, machine generated data as well as data crawled from the Internet. There have been several frameworks with proven efficiency to store and process the petabyte scale data such as Apache Hadoop, HDFS and several NoSQL frameworks. These systems have been widely used in industry and thus are subject to several research. The proposed data processing techniques should be compatible with the above frameworks in order to be practical. One of the key data operations is deriving equi-depth histograms as they are crucial in understanding the statistical properties of the underlying data with many applications including query optimization. In this paper, we focus on approximate equi-depth histogram construction for big data and propose a novel merge based histogram construction method with a histogram processing framework which constructs an equi-depth histogram for a given time interval. The proposed method constructs approximate equi-depth histograms by merging exact equi-depth histograms of partitioned data by guaranteeing a maximum error bound on the number of items in a bucket (bucket size) as well as any range on the histogram.

Index Terms—approximate histogram, merging histograms, big data, log files

1 INTRODUCTION

The data generated and stored by enterprises are in the orders of terabytes or even petabytes [1], [2], [3]. We can classify the source of the data in the following groups: machine generated data (a.k.a logs), social media data, transactional data and data generated by medical and wearable devices. Processing the produced data and deriving results are critical in decision making and thus the most important competitive power for the data owner. Therefore, handling such big datasets in an efficient way is a clear need for many institutions. Hadoop MapReduce [4], [5] is a big data processing framework that has rapidly become the standard method to deal with data bombarding in both industry and academia [3], [6], [7], [8], [9], [10]. The main reasons of such strong adoption are the ease-of-use, scalability, failover and open-source properties of Hadoop framework. After the wide distribution, many research works (from industry and academia) have focused on improving the performance of Hadoop MapReduce jobs in many aspects such as different data layouts [8], [11], [12], join algorithms [13], [14], [15], high-level query languages [3], [7], [10], failover algorithms [16], query optimization techniques [17], [18], [19], [20], and indexing techniques [6], [21], [22].

In today’s fast-paced business environment, obtaining results quickly represents a key desideratum for Big Data Analytics [3]. For most applications on large datasets, performing careful sampling and computing early results from such samples provide a fast and effective way to obtain approximate results within the predefined level of accuracy. The need for approximation techniques grow with the size of the data sets and most of the time they shed a light to make fast decisions for the businesses. General methods and techniques for handling complex tasks have room to improve in both MapReduce systems and parallel databases. For example, consider a web site, such as a search engine, consists of several web server hosts; user queries (requests) are collectively handled by these servers (using some scheduling protocol); and the overall performance of the web site is characterized by the latency (delay) encountered by the users. The distribution of the latency values is typically very skewed, and a common practice is to track some particular quantiles, for instance, the 95th percentile latency. In this context, one can ask the following questions:

- What is the 95th percentile latency of a single web server for the last month?
- What is the 95th percentile latency of the entire web site (over all the servers) for Christmas seasons?
- How latency affected the user behavior for the holiday seasons?

The Yahoo website, for instance, handles more than 43 million hits per day [23], which translates to 40000 requests per second. The Wikipedia website handles 30000 requests per second at peak, with 350 web servers. While all three questions relate to computing of statistics over data, they have different technical nuances, and often require different algorithmic approaches as accuracy can be traded for performance.

One way to obtain statistical information from data is histogram construction. Histograms summarize the whole data and give information about distribution of the data. Moreover, the importance of histogram increases when the size of the data is huge. Since, histograms are very useful and are efficient ways to get quick information about data distribution, they are highly used in database systems for query optimization, query result estimation, approximate query answering, data mining, distribution fitting, and parallel data partitioning [24]. One of the most used histogram
types in database systems is the equi-depth histogram. The equi-depth histogram is constructed by finding boundaries that split the data into a predefined number of buckets containing equal number of tuples. More formally, \( \beta \)-bucket equi-depth histogram construction problem can be defined as follows: given a data set with \( N \) tuples, find the boundary set \( B = b_1, b_2, \ldots, b_{\beta-1} \) that splits the sorted tuples into \( \beta \) buckets, each of which has approximately \( N/\beta \) tuples.

In this paper, we propose a framework to compute equi-depth histograms on-demand (dynamic) from the precomputed histograms of the partitioned data. In order to do so, we propose a histogram merging algorithm giving a user specified error bound on the bucket size. In particular, we merge \( T \)-bucket histograms to build a \( \beta \)-bucket histogram for the underlying data of size \( N \) and give a mathematical proof showing \( 2\beta/T \) error rate on the bucket size and as well as any range on the histogram. In our framework, users specify \( T \) and \( \beta \), we compute \( T \)-bucket histograms for each partition, and a query asking for a histogram of any subset of the partitions. Then, the framework computes the \( \beta \) histogram on-demand from the offline computed histograms of the partitions. In real life systems, the precomputation is done incrementally (i.e., daily, hourly or monthly) such as logs and database transactions.

Our contribution can be summarized as follows:

- We proposed a novel algorithm to build an approximate equi-depth histogram for a union of partitions from the sub-histograms of the partitions.
- We theoretically and experimentally showed that the error percentage of a bucket size is bounded by a predefined error set by the user (i.e., \( \varepsilon_{\text{max}} \)).
- We theoretically and experimentally showed that the error percentage of a range size is bounded by a predefined error set by the user (i.e., the same above \( \varepsilon_{\text{max}} \)).
- We implemented our algorithm on Hadoop and demonstrated how to apply it to practical real life problems.

The rest of the paper is organized as follows: In Section 2, we introduce the histogram construction problem for big data. We give some background information about cloud computing environments in Section 3. In Section 4, an in-depth explanation of the proposed method takes place. The details of implementation on Hadoop MapReduce framework is given in Section 5. In Section 6, related works are summarized. Evaluation methodology and experimental results are discussed in Section 7 and finally we conclude the paper with Section 8.

## 2 Problem Definition

In this section, we motivate the problem with a practical example and then formally define it. Machine generated data, also known as logs, is automatically created by machines. Logs contain list of activities of machines. In general, logs are stored daily in W3C format [25]. When we consider a web server, requests from web applications and responses from the server are written to log files. There are several actors deriving intelligence from these logs. For example; operations engineers derive operational intelligence (response times, errors, exceptions etc.) and business analyst derives business intelligence (top requested pages, top users, click through rates etc.). In the context of web applications, the need to analyze clickstreams has increased rapidly, and in order to answer the demand, businesses build log management and analytical tools. A typical internet business may have thousands of web servers logging every activity on the site. In addition, they have ETL processes incrementally collecting, cleaning and storing the logs in a big data storage (i.e., This is usually Hadoop and its storage HDFS). This work-flow is demonstrated in Figure 1. The amount of data to ETL and to run analytics on is huge and has been increasing rapidly. Most of the time, customers would be happy to trade accuracy for performance as they need a quick intelligence to make fast decisions.

One quick and reliable way to understand the statistics about the underlying data is using equi-depth histograms. In the paper, we outline a framework computing on-demand histograms of the data for any time interval for the above scenario. In web servers, daily logs are kept instantly. At the end of the day, all the log files belonging to that day are concatenated in a single log file and it is pushed to HDFS. As soon as the new log file is available in the HDFS, an exact equi-depth histogram is built and stored in the HDFS in a new summary file by the Summarizer Job. This means that the equi-depth histogram of each daily log is stored. Then, if a histogram for any time interval is requested (for example histogram for the last month), the Merger Job fetches an equi-depth histogram of each sub-histogram and merges them using the proposed merging algorithm explained in the following sections. We also provide an error rate on the histogram in order to increase the confidence. Although we motivate our framework with logs, it can be applied without loss of generality to any structured data where we need a histogram such as database transactions, etc...

After motivating and showing the need, we can formulate the problem we are solving as follows:

**Problem Definition:** Given \( k \) partitions, \( P_1, P_2, \ldots, P_k \), and their respective \( T \)-bucket equi-depth histograms, \( H_1, H_2, \ldots, H_k \), build a \( \beta \)-bucket equi-depth histogram \( H^* \) where \( \beta \leq T \) over \( P_1, P_2, \ldots, P_k \) where \(|P_1| + |P_2| + \ldots + |P_k|\) is equal to \( N \) and \( B_1, B_2, \ldots, B_\beta \) are the buckets of \( H^* \) such that:

- The size of any bucket \( B_i \) is \((N/\beta) \pm \varepsilon_{\text{max}}\) where \( \varepsilon_{\text{max}} < 2\beta/T \times (N/\beta) \).
- The size of any range spanning \( m \) buckets \( B_i \)
through $B_j$ is $m \times (N/\beta) \pm \varepsilon_{\text{max}}$ where $\varepsilon_{\text{max}} < 2\beta/T \times (N/\beta)$.

3 BACKGROUND

In this section, some background information is given about cloud computing environments such as Distributed File System (DFS) [26], MapReduce (MR) [27], Pig [28], and Hive [29].

Distributed File System (DFS): DFS is a generalized name of distributed, scalable, and fault-tolerant file systems such as Google FS [26] and Hadoop DFS [30]. In particular, we address the HDFS in this paper. In HDFS, large files are divided into small chunks and these small chunks are replicated and stored in multiple machines named DataNodes. The replication process ensures that HDFS is fault-tolerant. The metadata of the stored files such as name, replication count, file chunk locations, etc. are indexed in NameNode which is another machine. Clients read and write files to HDFS by interacting with the NameNode and the DataNodes.

The overall system architecture of HDFS is seen in Figure 3. In the figure, the NameNode takes place at the top and the DataNodes at the bottom. The replicas of the file chunks are labeled with the same numbers. The NameNode can interact with the DataNodes to maintain the file system by controlling the health and balancing the loads of the DataNodes. If there is a problem in a DataNode, the NameNode detects the problematic DataNode and replicates the file chunks in that DataNode to other DataNodes. Clients can also interact with the whole HDFS to read existing files and write new files to HDFS.

MapReduce (MR): MapReduce is a programming model for processing huge datasets which is especially resided in distributed file systems. Besides, MapReduce framework is the combination of the components which executes submitted MapReduce tasks by managing all resources and communications among the cluster while providing for fault tolerance and redundancy. In this paper, we specifically handle the Hadoop MapReduce framework.

A MapReduce task consist of Mappers and Reducers. The Mapper has a method called Map which gets $(key, value)$ pairs as input and emits $(key, value)$ pairs as intermediate output. The intermediate output is shuffled and sorted by a component of the MapReduce framework at the Sort and Shuffle phase and all $(key, value)$ pairs are grouped and sorted by keys at this phase. The output of the Sort and Shuffle phase is $(key, [value_1, value_2, ...])$ pairs and this is the input of the Reduce method which is in the Reducer. After the Reduce method finishes it’s job, it also emits $(key, value)$ pairs as final output of the MapReduce task. In some cases, a Combiner is also included in MapReduce tasks which is often the same with the Reducer. The Combiner has a Combine method which combines the output of the Map method to decrease network traffic.

The summary picture of the MapReduce framework is given in Figure 4. In the figure, input to a Mapper is read from HDFS. The output of the Mappers goes through the Sort and Shuffle phase and Reducers get the sorted and shuffled data and process it and write the output to HDFS, again.

Apache Pig: Pig is a platform for processing big data with query programs written in a procedural language called Pig Latin. Query programs are translated into MapReduce tasks and the tasks are run over MapReduce framework. The queries can be written by using both existing and user defined functions. Thus, Pig is an extensible platform and users can create their own functions.

Apache Hive: Hive is another platform for storing and processing large datasets like Pig. Hive has its own SQL-like declarative querying language named as HiveQL. HiveQL also supports custom user defined Map/Reduce tasks in queries.
In this section, we explain our approximate equi-depth histogram construction method in detail. In the first part of the method, exact equi-depth histograms of data partitions are constructed. This part is done offline with a well-known straightforward histogram construction algorithm. In the second and the important part of the method, equi-depth histograms are merged to construct an approximate equi-depth histogram over the partitions. One important feature is that the constructed histogram comes with maximum error bound on both size each bucket and size of any bucket range.

In the following part of the section, merging part of the method is explained with an example and then the algorithm of the merging is given and the section is concluded with maximum error bound theorems and their proofs.

A $T$-bucket equi-depth histogram $H$ for a set of values $P$ (may be called a partition) can be described as an increasing sequence of numbers, which represents the boundaries. Each pair of consecutive boundaries defines a bucket, and the size of this bucket is the number of values between its boundaries, where inclusive at the front and exclusive at the end (except the last bucket). Last bucket size also includes the last boundary. For an exact equi-depth histogram, size of each bucket is the same and equals and exactly total number of values divided by total number of buckets. On the other hand, bucket sizes of an approximate equi-depth histograms may not be equal.

We express a $T$-bucket equi-depth histogram as $H = \{(b_1, s_1), (b_2, s_2), \ldots, (b_i, s_i), \ldots, (b_{T-1}, s_{T-1}), (b_T, 0)\}$, where $b_i$ indicates the $i_{th}$ boundary and the $s_i$ indicates the $i_{th}$ bucket size for exact histograms (the approximate size of the $i_{th}$ bucket for approximate histograms), for the rest of the paper.

Let us have two example value sets, $P_1$ and $P_2$, which are \{2, 4, 5, 6, 7, 10, 13, 16, 18, 20, 21, 25\} and \{3, 9, 11, 12, 14, 15, 17, 19, 22, 23, 24, 26, 27, 29, 30\}. According to the value sets, $|P_1|$ and $|P_2|$ which represent number of values in each set, equal to 12 and 15, respectively. 3-bucket histogram of $P_1$ is $H_1 = \{(2, 4), (7, 4), (18, 4), (25, 0)\}$ and $P_2$ is $H_2 = \{(3, 5), (15, 5), (24, 5), (30, 0)\}$ and graphical representation of them are given in Figures 5 and 6.

First bucket of $H_1$ contains the first four values, \{2, 4, 5, 6\}, the second bucket contains four values, \{7, 10, 13, 16\}, and the third (also the last) bucket contains the last four values \{18, 20, 21, 25\}. For $H_2$, first bucket has five values,
Definition

The exact equi-depth histogram for the union of the histograms and let bucket count of final histogram, total number of values \( \{3, 9, 11, 12, 14\} \), the second bucket contains five values, \( \{15, 17, 19, 22, 23\} \), and the last bucket has five values, \( \{24, 26, 27, 29, 30\} \). Let us define a \( s(i, H) \) function which denotes the size of the \( i^{th} \) bucket of the equi-depth histogram \( H \), and a \( S(i, H) \) function which denotes the cumulative size of all buckets from the first to the \( i^{th} \) bucket of \( H \), that is,

\[
S(i, H) = s(1, H) + s(2, H) + \cdots + s(i, H)
\]

(1)

Then, the convention assures that \( S(i, H) = i \times |P|/T \), for all \( i \leq T \), where \( |P| \) is the number of values and \( T \) is the number of buckets. Considering \( H_1 \), \( s(1, H_1) = s(2, H_1) = s(3, H_1) = 4 \). For cumulative sizes, \( S(1, H_1) = 4 \), \( S(2, H_1) = 8 \), and \( S(3, H_1) = 12 \). Bucket sizes of \( H_2 \) is \( s(1, H_2) = s(2, H_2) = s(3, H_2) = 5 \) and cumulative sizes are \( S(1, H_2) = 5 \), \( S(2, H_2) = 5 \), and \( S(3, H_2) = 5 \).

Let us define two more functions, \( a(i, H) \) and \( A(i, H) \), which are the \( i^{th} \) approximate bucket size and the \( i^{th} \) cumulative bucket size for approximate equi-depth histograms, respectively. By writing an approximate version of Equation 1 we get the following equation:

\[
A(i, H) = a(1, H) + a(2, H) + \cdots + a(i, H)
\]

(2)

Lastly, let us define a range function \( R(i, j, H) \) that gives the sum of sizes of buckets which starts from the \( i^{th} \) bucket up to the \( j^{th} \) bucket, both inclusive. The formal definitions are given in the following formulas for both exact bucket sizes and approximate bucket sizes.

\[
R_a(i, j, H) = s(i, H) + s(i + 1, H) + \cdots + s(j, H)
\]

\[
R_e(i, j, H) = a(i, H) + a(i + 1, H) + \cdots + a(j, H)
\]

(3) (4)

The definitions given belove are summarized in Table 1.

Since we completed the definitions for conventions, we start to explain the merging process in detail. We have exact 3-bucket equi-depth histograms \( H_1 \) and \( H_2 \) given in Figures 5 and 6 for the example value sets \( P_1 \) and \( P_2 \) where \( P_1 = \{2, 4, 5, 6, 7, 10, 13, 16, 18, 20, 21, 25\} \) and \( P_2 = \{3, 9, 11, 12, 14, 15, 17, 19, 22, 23, 24, 26, 27, 29, 30\} \). The total number of values \( N \) is equal to \( |P_1| + |P_2| = 12 + 15 = 27 \) and let bucket count of final histogram, \( \beta \), be 3. As seen in the histograms \( H_1 \) and \( H_2 \), \( H_1 \) has a boundary sequence of \( 2, 7, 18, 25 \) and each \( H_1 \) bucket has \( 12/3 = 4 \) values, \( H_2 \) has \( 3, 15, 24, 30 \) and bucket size of \( 15/3 = 5 \). In Figure 7, we show \( H_1 \) and \( H_2 \) on the same plot, so therein, we clearly see the overall sequence of boundary values, which is \( 2, 3, 7, 15, 18, 24, 25, 30 \). Although the desired final number of buckets \( \beta \) may be chosen to be any number less than or equal to 3, we drive the example merging for \( \beta = 3 \).

Let us name the calculated pre-histogram (after first assembling of \( H_1 \) and \( H_2 \)) as \( H^0 \), final merged approximate equi-depth histogram resulted from our method as \( H^* \), and exact equi-depth histogram for the union of value sets \( P_1 \) and \( P_2 \) as \( H^e \). Briefly, we start with assembling \( H_1 \) and \( H_2 \) in an initial pre-histogram \( H^0 \) and we merge consecutive buckets of \( H^0 \) while the merged bucket size is greater than or equal to the exact bucket size \( N/\beta \) till the remaining number of buckets is equal to the desired number \( \beta \). For \( \beta = 3 \), exact bucket sizes \( N/\beta \) should be equal to \( 27/3 = 9 \) and it can be presented as \( s(1, H^e) = s(2, H^e) = s(3, H^e) = 9 \). The cumulative bucket sizes for \( H^e \) are \( S(1, H^e) = 9 \), \( S(2, H^e) = 18 \), and \( S(3, H^e) = 27 \). Now, we shall examine the creation of the pre-histogram \( H^0 \). The boundaries of \( H^0 \) are \( 2, 3, 7, 15, 18, 24, 25, 30 \) which are shortly the sorted boundaries of \( H_1 \) and \( H_2 \). Since \( H^0 \) has \( (T + 1) \times 2 = (3 + 1) \times 2 = 8 \) boundaries, it has \( 8 - 1 = 7 \) buckets. The important part of the creation is approximation of bucket sizes of \( H^0 \). Before the approximation of bucket sizes, we should determine approximate cumulative bucket sizes of \( H^0 \) and then we are able to calculate the approximate bucket sizes from the definition.

![Figure 5](image1.png)

Fig. 5. A sample equi-depth histogram \( H_1 \) with 3 buckets, based on data \( \{2, 4, 5, 6, 7, 10, 13, 16, 18, 20, 21, 25\} \).

![Figure 6](image2.png)

Fig. 6. Another sample equi-depth histogram \( H_2 \) with 3 buckets, which represents data \( \{3, 9, 11, 12, 14, 15, 17, 19, 22, 23, 24, 26, 27, 29, 30\} \).

![Figure 7](image3.png)

Fig. 7. Example equi-depth histograms given in Figures 5 (orange-dashed) and 6 (cyan-solid) are coupled together, with boundary sequence \( \{2, 3, 7, 15, 18, 24, 25, 30\} \).

| Symbol | Definition |
|--------|------------|
| \( H^0 \) | Pre-histogram after first assembling of \( k \) exact equi-depth histograms, \( H_1, H_2, \ldots, H_k \) |
| \( H^* \) | Final approximate equi-depth histogram after bucket merging operations of \( H^0 \) |
| \( H^e \) | Exact equi-depth histogram for the union of \( k \) value sets, \( P_1, P_2, \ldots, P_k \) |
| \( s(i, H) \) | The \( i^{th} \) bucket size of the equi-depth histogram \( H \) |
| \( a(i, H) \) | The \( i^{th} \) approximate bucket size of the approximate equi-depth histogram \( H \) |
| \( S(i, H) \) | The \( i^{th} \) cumulative size of the equi-depth histogram \( H \) |
| \( A(i, H) \) | The \( i^{th} \) approximate cumulative size of the equi-depth histogram \( H \) |
| \( R(i, j, H) \) | Sum of bucket sizes starting from the \( i^{th} \) bucket up to the \( j^{th} \) bucket (both inclusive) of the equi-depth histogram \( H \) |
of the cumulative bucket size function \(A(i, H)\). The approximate cumulative bucket sizes are calculated by presuming that all values in each bucket are at the beginning boundary of the bucket. For example, let us consider the first bucket of \(H_1\). In this bucket, we have values 2, 4, 5, and 6 and we suppose that all these values are at the point 2. By using this supposition, any cumulative bucket size is easily determined by summing the bucket size of the histogram which holds the next boundary and the previous cumulative bucket size starting with 0. Thus, since the first boundary of \(H^0\) is 2 and this boundary is the first boundary of \(H_1\), the first cumulative approximate bucket size \(A(1, H^0)\) is equal to \(s(1, H_1) = 4\). The second cumulative approximate bucket size \(A(2, H^0)\) is equal to \(s(1, H_2) + A(1, H^0) = 5 + 4 = 9\) because the next coming \(H^0\) boundary is 3 and it is the first boundary of \(H_2\). After the boundary 3, the next \(H^0\) boundary is 7 and it is the second boundary of \(H_1\). Therefore, the third cumulative approximate bucket size \(A(3, H^0)\) is equal to \(s(2, H_1) + A(2, H^0) = 4 + 9 = 13\). The remaining approximate bucket sizes are calculated in the same way and \(A(4, H^0), A(5, H^0), A(6, H^0)\), and \(A(7, H^0)\) are 18, 22, 27, and 27, respectively. Now, we are able to calculate approximate bucket sizes. Approximate size of the first bucket \(a(1, H^0)\) relying between boundary 2 and 3 is directly equal to \(A(1, H^0)\) and it is 4. The second approximate bucket size \(a(2, H^0)\) is equal to \(A(2, H^0) − A(1, H^0) = 9 − 4 = 5\). Similarly, \(a(3, H^0) = A(3, H^0) − A(2, H^0) = 13 − 9 = 4\), \(a(4, H^0) = 5\), \(a(5, H^0) = 4\), \(a(6, H^0) = 5\), and \(a(7, H^0) = 0\). Graphical representation of created \(H^0\) is given in Figure 8.

Next, we merge the buckets of \(H^0\) until the remaining bucket count is equal to \(\beta\). We use approximate cumulative bucket size instead of approximate bucket size to decrease the division error while merging. The merging process starts with the first bucket of \(H^0\). First of all we compare the first cumulative bucket size of \(H^0\), \(A(1, H^0)\), with the first cumulative bucket size of exact (ideal) histogram, \(S(1, H^c)\), and we see that \(A(1, H^0)\) is less than \(S(1, H^c)\). We continue comparing the next cumulative bucket size of \(H^0\), \(A(2, H^0)\), with again the first cumulative bucket size of exact (ideal) histogram, \(S(1, H^c)\), and we now see that \(A(2, H^0)\) is equal to \(S(1, H^c)\). Again, we continue comparing. This time, we see that \(A(3, H^0)\) is greater than \(S(1, H^c)\). Therefore, the buckets starting from the first bucket to the third bucket except the third one (because the result of the previous comparison is equality) would be merged and this merged bucket would be the first bucket of the final merged approximate histogram, \(H^*\). The resulting new bucket size would be \(A(2, H^0)\) because the new merged bucket is the first bucket of \(H^*\). Then, we are going to create the second bucket of \(H^*\). For this creation, we continue comparing cumulative bucket sizes starting from the first not merged bucket number with the second cumulative bucket size of \(H^c\). We see that \(A(3, H^0)\) is less than \(S(2, H^c)\). Next comparison is between the next cumulative of \(H^0\) and again the second cumulative of \(H^c\). This time equality is seen. We continue comparing the next cumulative of \(H^0\), \(A(5, H^0)\) with \(S(2, H^c)\). At this point, \(A(5, H^0)\) is greater than \(S(2, H^c)\). Thus, we merge the buckets starting from the third one to the fifth one again except the fifth one and the created new bucket would be the second bucket of \(H^*\). This merging process would end when the remaining bucket count is equal to \(\beta\) and we get \(H^* = \{(2, 9), (7, 9), (18, 9), (30, 0)\}\) as seen in Figure 9(a). For comparison, \(H^c\) is given in Figure 9(b).

The generalization of this method for merging more than 2 histograms is now easy after the one given above. Let us have \(k\) value sets \((P_1, P_2, ..., P_k)\) and their summaries (T-bucket equi-depth histograms, \(H_1, H_2, ..., H_k\)) to be merged. The merging process for the general case starts with the creation of an initial pre-histogram, \(H^0\). This can be done with sorting all boundary values coming from summaries and determining approximate bucket sizes in the same way with the one described above. The calculated histogram \(H^0\) has \(k \times (T + 1)\) boundaries and thus \(k \times (T + 1) − 1\) buckets. The rest of the merging method is exactly the same with the case when we have only two histograms. That is, we combine consecutive buckets of \(H^0\) by comparing the cumulative bucket sizes of \(H^0\) with cumulative sizes of exact histogram, \(H^c\), until \(\beta\) buckets remain.

Algorithm 4 shows the pseudocode of the explained method above. The algorithm takes \(T\)-bucket equi-depth histograms of \(k\) value sets, total number of values, \(N\), which
Algorithm 1: Equi-depth Histogram Merging

Input: $H_1, H_2, \ldots, H_k$: $k$ equi-depth histograms each with $T$ buckets, $N$: total number of values, $\beta$: desired bucket count of final histogram

Output: $H^*$: an approximate equi-depth histogram with $\beta$ buckets.

1. $b \leftarrow \{\text{sorted boundaries of } H_1, H_2, \ldots, H_k\}$
2. $s \leftarrow \{\text{bucket sizes calculated as described}\}$
3. $H^0 \leftarrow \text{CREATEHISTOGRAM}(b, s)$
4. $H^* \leftarrow H^0$
5. last ← 1; next ← 1; current ← 1
6. remaining ← $k(T + 1) - 1$
7. while remaining $> \beta$ do
8. while $A(\text{next}, H^0) \leq \text{current} \times N/\beta$ do
9. next ← next + 1
10. MERGBUCKETS(last, next − 1, $H^*$)
11. last ← next; current ← current + 1
12. remaining ← remaining − (next − 1 − last)
13. return $H^*$

is the sum of all sizes of value sets, and desired bucket count of final histogram, $\beta$ as inputs and constructs and returns final approximate $\beta$-bucket equi-depth histogram, $H^*$. Lines 1 through 5 of the algorithm is performed for the creation of the initial pre-histogram, $H^0$. First, boundaries of input histograms are sorted at Line 1 and then bucket sizes are calculated according to the above example at Line 2. The subroutine CREATEHISTOGRAM called at Line 3 simply creates a histogram from given boundary and bucket size sets and at that line $H^0$ is created from $b$ and $s$. The created $H^0$ has $k \times (T + 1)$ boundaries and $k \times (T + 1) - 1$ buckets. After creation of $H^0$, it would be copied to $H^*$ at Line 4. Once $H^0$ is created and copied to $H^*$, required buckets are combined on $H^*$ considering ideal bucket size, $N/\beta$. The main while loop iterates until the remaining number of buckets is equal to $\beta$. The inner while loop given in Lines 8 and 9 seeks for the next feasible point of buckets to combine at each iteration of the main loop. When such a point is found, we apply MERGBUCKETS subroutine which combines buckets from last to next − 1, both inclusive, on $H^*$ as shown in Line 10. Notice that MERGBUCKETS merges buckets according to the first state of bucket indexes.

For the asymptotic performance of the algorithm, sorting boundaries is likewise merging $k$ sorted lists and it can be done in $O(Tk \log k)$. Bucket sizes and CREATEHISTOGRAM subroutine can both run in $O(Tk)$ at Lines 2 and 3. For the inner loop, the increment at Line 9 can be performed at most $\beta$ times. The number of iterations for the main loop changes with the decrease in remaining bucket counts. Observe that the decrease is equal to the inner loop iteration number and MERGBUCKETS subroutine takes the same time with the inner loop for each main loop iteration. Considering this observation, total time required for the main loop is $O(Tk)$. Consequently, the initial sorting dominates the rest of the algorithm, and the algorithm runs in $O(Tk \log k)$-time.

Let us debug the algorithm line by line for the two example 3-bucket equi-depth histograms $H_1$ and $H_2$ given in Figures 5 and 6. Recall that $H_1$ and $H_2$ are histograms of value sets $P_1$ and $P_2$. Therefore $N$ is equal to $|P_1| + |P_2| = 12 + 15 = 27$. Let $\beta$ is equal to 3. We know $H^0 = \{ (2, 4), (3, 5), (7, 4), (15, 5), (18, 4), (24, 5), (25, 0), (30, 0) \}$ from the given detailed explanation above. In addition, the start state of $H^*$ is the same as $H^0$. The variables last, next, and current is equal to 1 and remaining is calculated as $k(T + 1) - 1 = 2(3 + 1) - 1 = 7$. Because the remaining is greater than $\beta$ at current state, we enter the main loop. For the inner loop, $A(1, H^0)$ and $A(2, H^0)$ is less or equal to current $\times N/\beta$ which is $1 \times 27/3 = 9$ but $A(3, H^0)$ is greater than 9. Hereby, inner while loop 2 times and next would be 3. Then, MERGBUCKETS subroutine merges the buckets 1 and 2 of $H^*$. The illustration of $H^*$ is shown in Figure 10 after merging. The variables last and current are updated after the execution of MERGBUCKETS is finished and last would be 3 and current would be 2. The remaining variable, keeping the remaining bucket number of $H^*$, would be 6 after the calculation is done at Line 12. The main loop finishes after $H^*$ has $\beta$ buckets and execution of the algorithm ends with returning the created $H^*$.

Now, we discuss the error bounds of the output histogram $H^*$. The following two theorems and their proofs verify the error bounds on bucket sizes and the sum of any range of bucket sizes of $H^*$.

**Theorem 1.** Let $H_1, H_2, \ldots, H_k$ be $T$-bucket equi-depth histograms of value sets $P_1, P_2, \ldots, P_k$, and $H^*$ be the approximate $\beta$-bucket equi-depth histogram where $\beta \leq T$ constructed by the algorithm. Then, the size of any bucket $a(i, H^*)$ is $(N/\beta) \pm \varepsilon_{\text{max}}$ where $\varepsilon_{\text{max}} < 2\beta/T \times (N/\beta)$.

**Proof 1.** Recall that the calculations of bucket sizes of $H^0$ depends on supposition that all values in each bucket are at the beginning boundary of the bucket and $H^*$ is some-buckets-merged version of $H^0$. Now consider an $i^{th}$ bucket between the $i^{th}$ and the $i + 1^{th}$ boundaries (boundaries may be any of the two consecutive boundaries of $H^0$) of $H^*$ illustrated in Figure 11. As seen in the figure, all of the values in the buckets divided by the $i^{th}$ boundary may stay at the right hand side of the boundary in contrast to our assumption and all values in the buckets divided by the $i + 1^{th}$ boundary may stay at the left hand side of the boundary. In this case, $a(i, H^*)$ gets the maximum value. Vice versa, $a(i, H^*)$ gets the minimum value in the case that all possible values in the divided buckets stay out of the $i^{th}$ bucket in contrast.
Proof 2. The following calculation shows the maximum value of \( a(i, H^*) \).

\[
a(i, H^*)_{\text{max}} = C + |P_1|/T + |P_2|/T + \cdots + |P_k|/T
\]

where \( C \) is constant which is the sum of the sizes of the buckets relying completely in the \( i^{th} \) bucket and \( |P_1|, |P_2|, \ldots, |P_k| \) is the size of sets. Adding and subtracting \( |P_2|/T \) and \( |P_k|/T \) to the equation, we get the following equation.

\[
a(i, H^*)_{\text{max}} = C + (|P_1| + |P_2| + \cdots + |P_k|)/T
\]

And \( a(i, H^*)_{\text{max}} \) is equal to \( C \) because no additional values are located in the \( i^{th} \) bucket except the constant ones. Once \( a(i, H^*)_{\text{max}} \) and \( a(i, H^*)_{\text{min}} \) are determined, \( \varepsilon_{\text{max}} \) would be the difference between them.

\[
\varepsilon_{\text{max}} = a(i, H^*)_{\text{max}} - a(i, H^*)_{\text{min}}
\]

\[
< C + 2N/T - C
\]

\[
< 2N/T
\]

The following equation shows another expression of \( \varepsilon_{\text{max}} \) in terms of exact (ideal) bucket size \( N/\beta \).

\[
\varepsilon_{\text{max}} = 2N/T
\]

\[
< 2N/\beta T
\]

\[
< 2\beta T \times (N/\beta)
\]

Theorem 2. Let \( H_1, H_2, \ldots, H_k \) be \( T \)-bucket equi-depth histograms of value sets \( P_1, P_2, \ldots, P_k \), and \( H^* \) be the approximate \( \beta \)-bucket equi-depth histogram where \( \beta \leq T \) constructed by the algorithm. Then, the size of any range spanning \( m \) buckets, \( R_p(i, i + m, H^*) \), is \( m \times (N/\beta) \pm \varepsilon_{\text{max}} \) where \( \varepsilon_{\text{max}} < 2\beta T \times (N/\beta) \).

Proof 2. Let us start with proving the error bound of range size of two consecutive buckets. Figure 12 shows this case. There are two consecutive buckets and three boundaries \( b_i, b_{i+1}, \) and \( b_{i+2} \), the middle one \( b_{i+1} \) splits the two buckets. Notice that the intersected buckets by \( b_{i+1} \) completely rely in the range of the two buckets and this means that the sizes of these buckets are added as a constant to the range size \( R_p(i,i+1,H^*) \). As a result, this proof turns into the proof of error bound of bucket size given in Proof 1 and Equation 2 also proves Theorem 2. The general case -spanning ranges includes more than two buckets- can also transform into a single bucket problem in the same way with the case with two buckets.

According to Theorems 1 and 2 users can bound the maximum bucket size error of final \( \beta \)-bucket approximate equi-depth histogram \( H^* \) by selecting appropriate bucket numbers \( \beta \) and \( T \). For example, let us calculate \( T \), number of buckets of equi-depth histograms of data partitions kept in the summary files, in terms of \( \beta \) for getting final merged histograms, the maximum bucket size errors of which do not exceed 5% of the ideal bucket size \( (N/\beta) \). If we use Equation 8, we can find the minimum number of buckets \( T \) needed to satisfy the 5% error condition as follows.

\[
\varepsilon_{\text{max}} < 2\beta T \times (N/\beta) \leq 0.05(N/\beta)
\]

\[
40\beta \leq T
\]

Consequently, the required bucket size \( T \) should be at least 40 times \( \beta \) which is the desired number of buckets of constructed histograms using our method.

5 Implementation With Hadoop Map-Reduce

In this section, we explain the implementation details of our histogram processing framework on Hadoop MapReduce. The framework consists of two main MapReduce jobs. One of them is named as Summarizer which runs offline and is scheduled for summarizing the new coming data to HDFS. The Summarizer constructs a \( T \)-bucket equi-depth histogram of the data. After summarizing, the resulting equi-depth histograms are stored in HDFS. The second job, Merger, is run on-demand according to users’ requests. Its duty is to merge the related summaries from HDFS by considering user requests and to construct the final \( \beta \)-bucket approximate equi-depth histogram.
When the data is too big to handle in a single-node database, the data is distributed to multi-nodes. One of the well-known distributed data storage frameworks is Hadoop Distributed File System (HDFS) and the data processing framework of the stored data in the HDFS is Hadoop MapReduce. The histogram construction of such distributed data is not well-studied and there is less work on histogram creation of distributed data than the ones on undistributed data. One of the adapted methods for constructing approximate histogram is tuple-level sampling. Okcan et al. proposed a tuple-level sampling based algorithm to construct approximate equi-depth histograms for distributed data to improve processing theta-joins using MapReduce. The algorithm works as follows. In the map section of a MapReduce Job, a predefined number of tuples are selected randomly by scanning the whole data and outputted. The tuples are sorted and sent to the reducer. The reducer of the job determines and outputs the boundaries of equi-depth histograms. In a method for approximate wavelet histogram construction for big data using MapReduce is proposed and an improved sampling method -ignoring low frequent sampled keys in splits- is given. The drawback of such histogram construction algorithms of distributed data using tuple-level sampling is that scanning the whole data is a time consuming process. Another approximate histogram construction method is proposed in [33]. This method also uses a sampling method named two-phase sampling which samples the whole data at block-level and constructs the approximate histogram and calculates the error. If the error is not in the desired error boundary, the additional sampling size needed is calculated and histogram construction process is repeated. The insufficiency of this method is that histogram is rebuilt for every new data and it requires a customized MapReduce framework. In this paper, we propose a novel approximate equi-depth histogram construction method with a log histogram monitoring framework that users can query the daily stored log files for their equi-depth histogram. In the proposed method, a MapReduce Job is scheduled to summarize the daily stored log files which means that the exact equi-depth histogram of each log file is constructed and stored in corresponding summary files and another MapReduce Job merges the summaries of intended log files for approximate equi-depth histogram construction.
7 Experimental Results

In this section, we will describe how we tested the proposed method. The method was implemented on Hadoop MapReduce framework and was tested on two different datasets. One of them is synthetic data with 155 million of tuples created by using Gumbel distribution for skewness to represent the response of the method for skewed data. The other one is 295 GB uncompressed real data which is taken from hourly page view statistics of Wikipedia. The data consists of approximately 5 billion tuples which belong to January 2015 and each tuple has 4 columns which are language, pagename, pageviews and pagesize. We used pagesize for histogram construction. The proposed method (merge) is compared with corrected tuple level random sampling (tuple). By definition, bare tuple level random sampling collects tuples randomly and constructs histogram with collected tuples but doing so does not work well when the data is sparse at the edges. Therefore, we fix this problem by including the edge values to the collected tuples by default. As mentioned in Subsection 2, histogram construction of the data coming from daily logs is an important issue and tuple level random sampling method is also unfavorable for constructing a histogram of a given time interval. Hereby, sampling stage of tuple level is run offline to compare time spendings.

All equi-depth histograms are build with bucket size of 254 as used by Oracle as default bucket size for histogram enhancement. We fundamentally run two types of tests to represent the effectiveness of the proposed method in terms of boundary and bucket size error and run time. The first test represents the results according to T changes which is daily exact histogram bucket size for the proposed method and sample size for tuple level sampling. The second test represents the run time efficiency of histogram construction for the changes in a given time interval.

Approximate histograms may have two types of error. One of them is that approximate histograms may not have the same bucket boundaries with the exact ones and the other is that bucket sizes of the approximate histograms may deviate from the exact ones. The former error is named as boundary error ($\mu_b$) and defined as follows:

$$\mu_b = \frac{B}{v_{max} - v_{min}} \sqrt{\frac{1}{B+1} \sum_{i=1}^{B+1} [b(i, H*) - b(i, H)]^2}.$$  \hspace{1cm} (9)

where $B$ is the bucket size, $v_{max}$ and $v_{min}$ are maximum and minimum values in relation $R$ respectively, and the function $b(i, H)$ is the $i^{th}$ value of a given histogram $H$. $\mu_b$ is the standard deviation of boundary errors normalized with respect to the mean boundary length $(v_{max} - v_{min})/B$. The latter error is named as size error ($\mu_s$) and formulated as follows:

$$\mu_s = \frac{B}{N} \sqrt{\frac{1}{B} \sum_{i=1}^{B} [s(i, H*) - s(i, H)]^2}.$$  \hspace{1cm} (10)

where $N$ is the total number of elements in relation $R$ and function $s(i, H)$ is the size of the $i^{th}$ bucket of a given histogram $H$. $s(i, H)$ is equal to the mean bucket size $N/B$ for all $i$ values in the range of 1 to $B$ if the given histogram $H$ is an equi-depth histogram. $\mu_s$ is the standard deviation of bucket size errors normalized with respect to the mean bucket size $N/B$.

7.1 Effect of $T$

Changing $T$ value effects the accuracy of the constructed approximate equi-depth histogram. The first experiment is run to show the effects of $T$ changes on both the proposed method and tuple level sampling. Figures 14 and 15 show the error graphs of the approximate equi-depth histograms constructed using the proposed method and tuple level sampling method. According to the graph in Figure 14a the constructed histogram by using merge method for real data is at least 2 times more accurate in terms of boundary error $\mu_b$ than the one constructed using tuple method. Moreover, the $\mu_b$ error for tuple method is not consistent because of the randomness and the construction process must be repeated for consistency and this is not convenient because of the run time. For example, let us consider the graph of $\mu_b$ against $T$ given in Figure 14a Notice that the $\mu_b$ error for tuple method is not consistent. The expected result is that $\mu_b$ should decrease while $T$ increases. On the other hand, it is clearly seen from the graphs of merge method in Figures 14a and 15a that $\mu_b$ is a non-decreasing function of $T$. The reason of this consistency is
the maximum error bound of merge method described in Section 4 in detail. The mean running times for all methods are given in Table 2. Run times for merging daily summaries and samplings are nearly the same. But required time for the summarization stage of merge method is more than the time for offline sampling stage of tuple method. The reason of this time difference is that summarization is exact histogram construction and exact histogram construction for each data partition requires a complete MapReduce Job with a mapper and a reducer and the data comes from each mapper subjected to shuffle and sort phase. On the other hand, tuple level random sampling does not require a reducer because the randomly selected tuples would be stored directly without sorting. Because of this difference, summarization of each day for real data takes approximately 12 minutes while sampling takes 4 minutes. Although this time efficiency of tuple method, all the daily summarizations and samplings are done offline, it makes merge method convenient for real life applications.

7.2 Effect of given time interval

Histogram constructions according to the given time interval effect deviation of approximate histograms from the exact ones and running times. We compared merge method with tuple method with an experiment for 5 different time intervals (1 day, 1 week, 2 weeks, 3 weeks, and 1 month). T value is taken to be $B \times 254 \times 2^n$ for real data and to be $B \times 254 \times 2^7$ for skewed data. All daily exact histogram and samplings are computed offline. In Figures 16 and 17, graphs of error metrics against time intervals are given and it is clearly seen from the graphs that the proposed method produces more sensible histograms than the ones produced using tuple level random sampling. In particular, a real data histogram constructed using merge method has at least 10 times less boundary error than the one constructed using tuple method in Figure 16a. Besides, again the consistency is an issue for tuple method as seen in Figures 16a and 17a. The graphs of running time against number of days are given in Figure 18. More specifically, the compared methods (merge and tuple) run in nearly the same time duration except offline parts.

8 Conclusion

In this paper, we proposed a novel approximate equi-depth histogram construction method by merging precomputed exact equi-depth histograms of data partitions. The method is implemented on Hadoop to demonstrate how it is applied to real life problems. The theoretical calculations and the experimental results showed that both the bucket size errors and total size error of any bucket range are bounded by
a predefined error set by a user in terms of $T$ and $\beta$. In particular, the experimental results run on both real and synthetic data show that the constructed histograms using the proposed method (merge) are more accurate than the tuple level random sampling (tuple) with a cost of offline run time. In addition to the proposed merged based histogram construction method, we also proposed a novel histogram processing framework for the daily stored log files. This framework is crucial for fast histogram construction over a subset of a list of partitions on demand. The time complexity and the incrementally updated nature of the proposed method makes it practical to be applied over real life problems.

### Table 2
Mean Running Times of Monthly Equi-depth Histogram Construction

| Method                      | Real Data | Skewed Data |
|-----------------------------|-----------|-------------|
| Exact Hist. Construction    | 24358     | 582         |
| Tuple with Online Sampling  | 6169      | 68          |
| Summarizing for Each Day    | 725       | 18          |
| Merging of Daily Summaries | 117       | 73          |
| Sampling for Each Day       | 223       | 18          |
| Merging of Daily Samplings  | 113       | 71          |

**Fig. 17.** Graphs of error metrics against merged # of days for skewed data

**Fig. 18.** Graphs of running time against merged # of days

---

**ACKNOWLEDGMENTS**

We would like to thank Enver Kayaaslan for his help for presentation.

**REFERENCES**

1. D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, “Stateful bulk processing for incremental analytics,” in Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010, pp. 51–62.
2. A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at facebook,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 1013–1020.
3. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, “Hive-a petabyte scale data warehouse using hadoop,” in Data Engineering (ICDE), 2010 IEEE 26th International Conference on. IEEE, 2010, pp. 996–1005.
4. A. S. Foundation. (2008) Apache hadoop. [Online]. Available: https://hadoop.apache.org/.
5. J. Dean and S. Ghemawat, “Mapreduce: a flexible data processing tool,” Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.
6. J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, “Hadoop++: making a yellow elephant run like a cheetah (without it even noticing),” Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 515–529, 2010.
7. A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayana-murthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, “Building a high-level dataflow system on top of map-reduce: the pig experience,” Proceedings of the VLDB Endowment, vol. 2, no. 2, pp. 1414–1425, 2009.
