Shi YJ, Meng XF, Wang F *et al.* HEDC++: An extended histogram estimator for data in the cloud. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(6): 973–988 Nov. 2013. DOI 10.1007/s11390-013-1392-7

HEDC++: An Extended Histogram Estimator for Data in the Cloud

Ying-Jie Shi[1] (史英杰), Xiao-Feng Meng[1] (孟小峰), *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*
Fusheng Wang[2,3], and Yan-Tao Gan[1] (干艳桃)

[1] *School of Information, Renmin University of China, Beijing 100872, China*

[2] *Department of Biomedical Informatics, Emory University, Atlanta 30322, U.S.A.*

[3] *Department of Mathematics and Computer Science, Emory University, Atlanta 30322, U.S.A.*

E-mail: {shiyingjie, xfmeng}@ruc.edu.cn; fusheng.wang@emory.edu; ganyantao19901018@163.com

Received December 2, 2012; revised May 3, 2013.

Abstract With increasing popularity of cloud-based data management, improving the performance of queries in the cloud is an urgent issue to solve. Summary of data distribution and statistical information has been commonly used in traditional databases to support query optimization, and histograms are of particular interest. Naturally, histograms could be used to support query optimization and efficient utilization of computing resources in the cloud. Histograms could provide helpful reference information for generating optimal query plans, and generate basic statistics useful for guaranteeing the load balance of query processing in the cloud. Since it is too expensive to construct an exact histogram on massive data, building an approximate histogram is a more feasible solution. This problem, however, is challenging to solve in the cloud environment because of the special data organization and processing mode in the cloud. In this paper, we present HEDC++, an extended histogram estimator for data in the cloud, which provides efficient approximation approaches for both equi-width and equi-depth histograms. We design the histogram estimate workflow based on an extended MapReduce framework, and propose novel sampling mechanisms to leverage the sampling efficiency and estimate accuracy. We experimentally validate our techniques on Hadoop and the results demonstrate that HEDC++ can provide promising histogram estimate for massive data in the cloud.

Keywords histogram estimate, sampling, cloud computing, MapReduce

1 Introduction

The cloud data management system provides a scalable and highly cost-effective solution for large-scale data management, and it is gaining much popularity these days. Most of the open-source cloud data management systems, such as HBase①, Hive[1], Pig[2], Cassandra② and others now attract considerable enthusiasm from both the industry and academia. Compared with the relational DBMS (RDBMS) with sophisticated optimization techniques, the cloud data management system is newly emerging and there is ample room for performance improvement of complex queries[3]. As the efficient summarization of data distribution and statistical information, histograms are of paramount importance for the performance improvement of data accessing in the cloud. First of all, histograms provide reference information for selecting the most efficient query execution plan. A large fraction of queries in the cloud are implemented in MapReduce[4], which integrates parallelism, scalability, fault tolerance and load balance into a simple framework. For a given query, there are always different execution plans in MapReduce. For example, in order to conduct log processing which joins the reference table and log table, four different MapReduce execution plans are proposed in [5] for different data distributions. However, how to select the most efficient execution plan adaptively is not addressed, which could be guided by histogram estimation. Secondly, histograms contain basic statistics useful for load balancing. Load balance is crucial to the performance of query in the cloud, which is normally processed in parallel. In the processing framework of MapReduce, output results of the mappers are parti-

Regular Paper

This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 61070055, 91024032, 91124001, the Fundamental Research Funds for the Central Universities of China, the Research Funds of Renmin University of China under Grant No. 11XNL010, and the National High Technology Research and Development 863 Program of China under Grant Nos. 2012AA010701, 2013AA013204.

① http://hbase.apache.org/, October 2012.

② http://cassandra.apache.org/, October 2012.

tioned to different reducers by hashing their keys. If data skew on the key is obvious, then load imbalance is brought into the reducers and consequently results in degraded query performance. Histograms constructed on the partition key can help to design the hash function to guarantee load balance. Thirdly, in the processing of joins, summarizing the data distribution in histograms is useful for reducing the data transmission cost, which is among the scarce resources in the cloud. Utilizing the histogram constructed on a join key can help prevent sending the tuples that do not satisfy the join predicate to the same node[6]. In addition, histograms are also useful in providing various estimates in the cloud, such as query progress estimate, query size estimate and results estimate. Such estimates play an important role in pre-execution user-level feedback, task scheduling and resource allocation. However, it can be too expensive and impractical to construct a precise histogram before the queries due to the massive data volume. In this paper, we propose a histogram estimator to approximate the data distribution with desired accuracy.

Constructing approximate histograms is extensively studied in the field of single-node RDBMS. However, this problem has received limited attention in the cloud. Estimating the approximate histogram of data in the cloud is a challenging problem, and a simple extension to the traditional work will not suffice. The cloud is typically a distributed environment, which brings parallel processing, data distribution, data transmission cost and other problems that must be accounted for during the histogram estimate. In addition, data in the cloud is organized based on blocks, which could be a thousand times larger than that of traditional file systems③. The block is the transmission and processing unit in the cloud, and block-based data organization can significantly increase the cost of tuple-level random sampling. Retrieving a tuple randomly from the whole dataset may cause the transmission and processing of one block. A natural alternative is to adopt all the tuples in the block as the sampling data. However, correlation of tuples in one block may affect the estimate accuracy. To effectively build statistical estimators with blocked data, a major challenge is to guarantee the accuracy of the estimators while utilizing the sampling data as efficiently as possible. Last but not least, the typical batch processing mode of tasks in the cloud does not match the requirements of histogram estimate, where "early returns" are generated before all the data is processed.

In this paper, we propose an extended histogram estimator called HEDC++, which is significantly extended from our previous work called HEDC[7]. According to the rule in which the tuples are partitioned into buckets, histograms can be classified into several types, such as the equi-width histogram, equi-depth histogram, and spline-based histogram[8]. HEDC provides approximation method only for equi-width histogram, which is easy to maintain because its bucket boundary is fixed. Actually when summarizing the distribution for skewed data, equi-depth histogram provides more accurate approximation[8]. The main issue of equi-depth histogram approximation is to estimate the bucket boundary, which cannot be transformed into the estimate of functions of means just like the equi-width histogram. HEDC++ develops corresponding techniques for equi-depth histogram estimate, which include the sampling unit design, the error bounding and sampling size bounding algorithm, and the implementation methods over MapReduce. HEDC++ supports the equi-width and equi-depth histogram approximation for data in the cloud through an extended MapReduce framework. It adopts a two-phase sampling mechanism to leverage the sampling efficiency and estimate accuracy, and constructs the approximate histogram with desired accuracy. The main contributions include:

1) We extend the original MapReduce framework by adding a sampling phase and a statistical computing phase, and design the processing workflow for the approximate histogram construction of data in the cloud based on this framework.

2) We model the construction of equi-width and equi-depth histograms into different statistical problems, and propose efficient approximation approaches for their estimates in the cloud.

3) We adopt the block as the sampling level to make full use of data generated in the sampling phase, and we prove the efficiency of block-level sampling for estimating histograms in the cloud.

4) We derive the relationship of required sample size and the desired error of the estimated histogram, and design the sampling mechanisms to investigate the sample size adaptive to different data layouts, which leverage the sampling efficiency and estimate accuracy.

5) We implement HEDC++ on Hadoop and conduct comprehensive experiments. The results show HEDC++'s efficiency in histogram estimating, and verify its scalability as both data volume and cluster scale increase.

The rest of the paper is organized as follows. In Section 2 we summarize related work. In Section 3 we introduce the main workflow and architecture of HEDC++. Section 4 discusses the problem modeling and statistical issues, which include the sampling mechanisms and histogram estimate algorithms. The

③http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html, November 2012.

implementing details of HEDC++ are described in Section 5, and we also discuss how to make the processing incremental by utilizing the existing results. The performance evaluation is given in Section 6, followed by the conclusions and future work in Section 7.

2 Related Work

Histogram plays an important role in cost-based query optimization, approximate query and load balancing, etc. Ioannidis[9] surveyed the history of histogram and its comprehensive applications in the data management systems. There are different kinds of histograms based on the constructing way. Poosala *et al.*[8] conducted a systematic study of various histograms and their properties. Constructing the approximate histogram based on sampled data is an efficient way to reflect the data distribution and summarize the contents of large tables, which is proposed in [10] and studied extensively in the field of single-node data management systems. The key problem has to be solved when constructing approximate histogram is to determine the required sample size based on the desired estimation error. We can classify the work into two categories by the sampling mechanisms. Approaches in the first category adopt uniform random sampling[11-13], which samples the data with tuple level. Gibbons *et al.*[11] focused on sampling-based approach for incremental maintenance of approximate histograms, and they also computed the bound of required sample size based on a uniform random sample of the tuples in a relation. Surajit *et al.*[12] further discussed the relationship of sample size and desired error in equi-depth histogram, and proposed a stronger bound which lends to ease of use. [13] discusses how to adapt the analysis of [12] to other kinds of histograms. The above approaches assume uniform random sampling. Approaches of the second category construct histograms through block-level sampling[12,14]. [12] adopts an iterative cross-validation based approach to estimate the histogram with specified error, and the sample size is doubled once the estimate result does not arrive at the desired accuracy. [14] proposes a two-phase sampling method based on cross-validation, in which the sample size required is determined based on the initial statistical information of the first phase. This approach reduces the number of iterations to compute the final sample size, and consequently processing overhead. However, the tuples it samples is much bigger than the required sample size because cross-validation requires additional data to compute the error. All the above techniques focus on histogram estimating in the single-node DBMS, and adapting them to the cloud environment requires sophisticated considerations.

There is less work on constructing the histogram in the cloud. Authors of [6] focused on processing theta-joins in the MapReduce framework, and they adopted the histogram built on join key to find the "empty" regions in the matrix of the cartesian product. The histogram on the join key is built by scanning the whole table, which is expensive for big data. Jestes *et al.*[15] proposed a method for constructing approximate wavelet histograms over the MapReduce framework, their approach retrieves tuples from every block randomly and sends the outputs of mappers at certain probability, which aims to provide unbiased estimate for wavelet histograms and reduce the communication cost. The number of map tasks is not reduced because all the blocks have to be processed, and the sum of start time of all the map tasks cannot be ignored. Though block locality is considered during the task scheduling of MapReduce framework, there exist blocks that have to be transferred from remote nodes to the mapper processing node. So we believe there is room to reduce this data transmission cost. In the previous work, we proposed a histogram estimator for data in the cloud called HEDC[7], which only focuses on estimating the equi-width histograms. The equi-depth histogram is another type of histograms widely used in data summarizing for query optimization, and its estimation technique is very different from that of the equi-width histogram. In this paper, we extend HEDC to HEDC++, and propose novel method for equi-depth histogram estimate in the cloud.

3 Overview of HEDC++

Constructing the exact histogram of data involves one original Map-Reduce job, which is shown in the solid rectangles of Fig.1. To construct the equi-width histogram, the mappers scan data blocks and generate a bucket ID for every tuple. Then the bucket ID is used as the key of the output key-value pairs, and all the pairs belonging to the same bucket are sent to the same reducer. At last, the reducers combine the pairs in each bucket of the histogram. For the equi-depth histogram, the mappers and combiners compute the number of items of every column value, and the column value is set to be the output key of pairs sent to the reducers. All the pairs are sorted by the key in the reducers, and the separators of the buckets are determined. Generating the exact histogram requires full scan of the whole table, which is expensive and costs long time to complete in the cloud.

HEDC++ constructs the approximate histogram with desired accuracy to significantly reduce the time it takes to get efficient data summarization over large datasets. However, providing approximate results with statistical significance in the cloud requires to solve

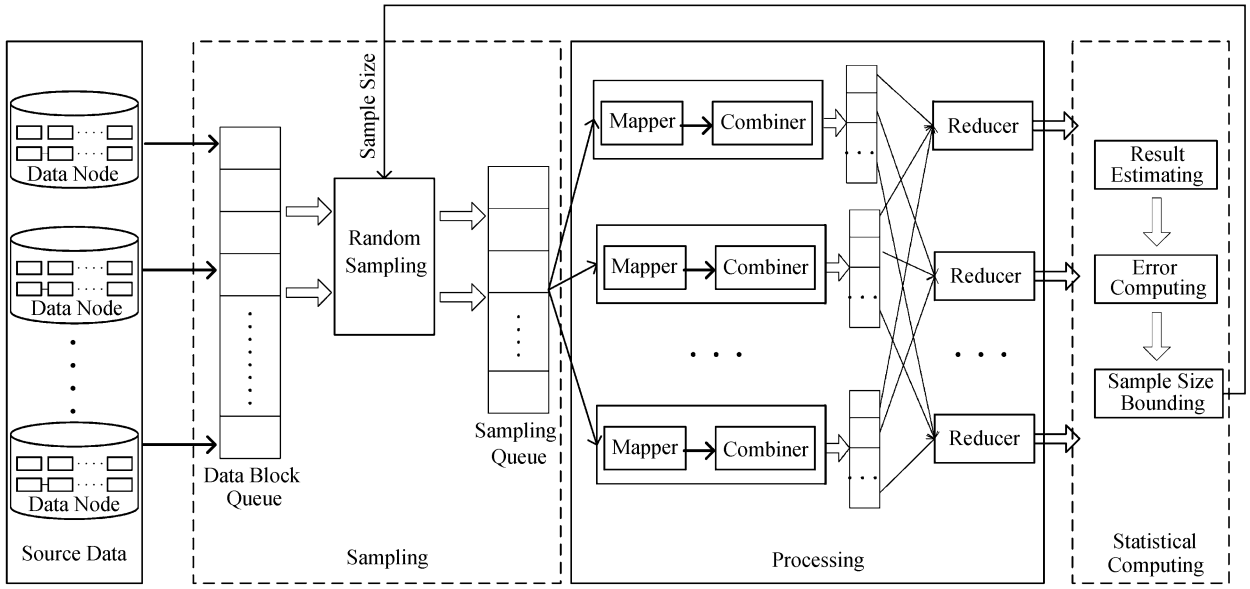

Fig.1. Architecture of HEDC++.

two challenging problems. First, data distributed in the cluster should be randomly sampled to guarantee the accuracy of estimated results. For reasonably large datasets, sampling at the tuple level is expensive, while sampling at the block level brings in data correlation. We design a block-level sampling algorithm that adapts to the correlations of tuples during one block. Secondly, the amount of samples should be determined according to the given estimate accuracy. We compute a bound on the sampling size without any assumptions on the data layout.

In order to satisfy the above requirements of histogram estimating in the cloud, we make novel extensions to the original MapReduce framework. Fig.1 illustrates the main workflow of HEDC++, and our extensions are identified in the dotted rectangles. The data files are organized into blocks and distributed in the cloud, and the block IDs are put into a scheduling queue to be arranged to map tasks. We add a sampling module on this scheduling queue, and a specified number of blocks are retrieved randomly from this queue as the MapReduce job's input. These sample blocks are processed by the following MapReduce jobs, which conduct different functions for equi-width and equi-depth histograms. However, the output result of the MapReduce job is just the histogram computed directly on the sampled data. So we add a statistical computing module after the reducers. During this module, we estimate the histogram of the whole data, and compute the error of the estimated histogram. If the error is less than the predefined error, then the estimated histogram is returned to the user. Otherwise, we compute a bound of the sample size required to meet the predefined accuracy based on the data layout. Then additional sample data required is retrieved by the sampling module and processed by the following MapReduce job. In the statistical computing module, the outputs are merged with the previous job's results, and the final estimated histogram is returned. Also we make the computing incremental and avoid the replicated computing at this phase. The details of sampling and statistical computing are discussed in the following sections.

4 Statistical Issues

In this section we model the estimate of equi-width and equi-depth histograms into different statistical problems, and discuss the statistical issues based on the characteristics of data storage and processing in the cloud. First we give the problem definition and notations.

4.1 Preliminaries

Consider a table T with t tuples, the data are organized into N blocks in the cloud, with block size B. The attribute of interest X is distributed over domain D. Then the problem is defined as follows:

Definition 1. *Set V to represent the value set of T on attribute $X(V \subseteq D)$, given a value sequence $s_1, s_2, \ldots, s_{k+1} \in V$, then the histogram contains k buckets $B_1, B_2, \ldots, B_k$, where $B_i = \{v | s_i \leqslant v < s_{i+1}\}$. The histogram estimate is to compute the bucket size $|B_i|$ based on sampled data from T.*

According to the rules in which the separators are determined, the histograms can be classified into different categories. The separators of the equi-width histo-

gram divide the value domain into buckets with equal width, set P_i to represent the proportion of tuples belonging to B_i, and we can get $|B_i| = N \times B \times P_i$. Consider the tuples of one bucket as a category, then estimating the equi-width histogram can be modeled as estimating the proportion of the different categories in the table. In the equi-depth histograms, the separators divide the value domain into buckets with equal height, so the proportion of every bucket is the same. The separators s_1 and s_{k+1} represent the boundaries of the value domain, and the remaining separators can be considered as the quantiles. Estimating the equi-depth histogram can be modeled as estimating the quantiles of the dataset.

In order to measure the difference between the approximation histogram constructed on the samples and the histogram constructed on the whole data, various error metrics are proposed in the literature. Set h_i to represent the exact size of bucket B_i, and set $\tilde{h_i}$ to represent the estimated size of B_i. The height-variance error[12] is defined as:

$$err\_height = \frac{k}{NB}\sqrt{\frac{1}{k}\sum_{i=1}^{k}(\tilde{h_i} - h_i)^2}. \quad (1)$$

It is the standard deviation of the estimated bucket size from the actual number of elements in each bucket, normalized with respect to the average width of buckets. This error metric is used for both the equi-width and equi-depth histograms. For equi-depth histograms, $\tilde{h_i}$ represents the size of the bucket determined by the estimated separators on the real dataset. And h_i is equal to the perfect bucket size NB/k. The max-boundary error[16] is special for the equi-depth histogram, which is defined as:

$$err\_boundary = \max_i \left\{\frac{k}{domain\_width}|\widetilde{s_i} - s_i|\right\}, \quad (2)$$

where $\widetilde{s_i}$ is the estimated separator for s_i, the max-boundary error measure the maximum error across all the separators, normalized with respect to the average of the bucket width. The exact error of an approximate histogram should be computed based on the precise histogram, which is gotten after the processing of the whole data. This error metric provides important reference information for evaluating the approximate results and determining the required sample size during the estimation processing. However, the exact histogram cannot be gotten at the time when the processing is far from completion. We propose algorithms to bound the error based on the sampled data for both equi-width and equi-depth histograms, and the details are described in Subsection 4.3.

4.2 Sampling Unit

Sampling is a standard technique for constructing approximate summaries of data, and most of the studies adopt the uniform random sampling. However, the tuple-level true random sampling can be very inefficient in the cloud. Recall that data in the cloud is organized into blocks and the block is the unit of data transmitting in the network. Picking tuple-level random sampling from data in such an organization is very expensive. In the worst case, retrieving n tuples may cause a full scan of n blocks. Secondly, the block is also the processing unit of the MapReduce framework, and one block is processed by a map task. The startup time of map tasks cannot be ignored when the number of tasks is big. One alternative approach for solving these problems is to sample using block as the unit. We prove in Theorem 1 that with the same data transmission cost, block-level sampling will provide more accurate estimated equi-width histogram than tuple-level sampling in the cloud.

Theorem 1. *Set $\tilde{P_{ib}}$ to represent the proportion of B_i obtained from a simple random sample of n blocks, and $\tilde{P_{it}}$ to represent the proportion of B_i obtained from a simple random sample of n tuples. Further let the estimated bucket size $\tilde{h_{ib}} = N \times B \times \tilde{P_{ib}}$, and $\tilde{h_{it}} = N \times B \times \tilde{P_{it}}$. Then both $\tilde{h_{ib}}$ and $\tilde{h_{it}}$ are unbiased estimates of the bucket size for the equi-width histogram, and the variance of $\tilde{h_{ib}}$ is equal to or less than that of $\tilde{h_{it}}$: $VAR(\tilde{h_{ib}}) \leqslant VAR(\tilde{h_{it}})$.*

Proof. The block-level random sampling has the same spirit with cluster sampling in the statistical terms[17], with the cluster size equal to the block size. Set P_{ij} to represent the proportion of elements belonging to B_i in the j-th block, then the proportion of B_i is: $P_i = \frac{1}{N} \times \sum_{j=1}^{N} P_{ij}$. According to the properties of proportion estimate on cluster sampling, we can get $\tilde{P_{ib}} = \frac{1}{n} \times \sum_{j=1}^{n} P_{ij}$ is the unbiased estimate of P_i, consequently $\tilde{h_{ib}}$ is the unbiased estimate of the bucket size. The variance of $\tilde{P_{ib}}$ is:

$$VAR(\tilde{P_{ib}}) = \frac{N-n}{N^2 n}\sum_{j=1}^{N}(P_{ij} - P_i)^2. \quad (3)$$

The tuple-level sampling is a final uniform sampling. According to the characteristic of proportion estimate, the sample proportion $\tilde{P_{it}}$ is the unbiased estimate of the population proportion, and the variance of $\tilde{P_{it}}$ is:

$$VAR(\tilde{P_{it}}) = \frac{NB-n}{NB-1} \times \frac{P_i(1-P_i)}{n}. \quad (4)$$

Consequently, the design effect *deff* is:

$$\frac{VAR(\tilde{h_{ib}})}{VAR(\tilde{h_{it}})} = \frac{N^2B^2\,VAR(\tilde{P_{ib}})}{N^2B^2\,VAR(\tilde{P_{it}})} = \frac{NB-nB}{NB-n} \times \frac{\sum_{j=1}^{N}(P_{ij}-P_i)^2}{NP_i(1-P_i)}. \quad (5)$$

While $B \geqslant 1$, then we can get:

$$NB - nB \leqslant NB - n. \quad (6)$$

The numerator of the right part in (5) can be derived as:

$$\begin{aligned}\sum_{j=1}^{N}(P_{ij}-P_i)^2 &= \sum_{j=1}^{N}(P_{ij}^2 - 2P_iP_{ij} + P_i^2)\\ &= \sum_{j=1}^{N}P_{ij}^2 - 2NP_i^2 + NP_i^2\\ &= \sum_{j=1}^{N}P_{ij}^2 - NP_i^2. \end{aligned} \quad (7)$$

P_{ij} is the proportion of tuples belonging to bucket B_i in the j-th block, so $0 \leqslant P_{ij} \leqslant 1$, then according to (7) we can get:

$$\sum_{j=1}^{N}(P_{ij}-P_i)^2 \leqslant \sum_{j=1}^{N}P_{ij} - NP_i^2 = NP_i - NP_i^2. \quad (8)$$

According to (6) and (8), we can conclude that:

$$\frac{VAR(\tilde{h_{ib}})}{VAR(\tilde{h_{it}})} \leqslant 1. \quad (9)$$

□

Theorem 1 reflects the efficiency of block-level sampling for equi-width histograms. However, estimating the equi-depth histogram is different from the problem of estimating the proportion. The height of every bucket is almost the same in the equi-depth histogram, and the main problem is to estimate the boundary of every bucket, which can be considered as the quantiles of the population. We prove in Theorem 2 that with the same transmission cost, the block-level sampling will also provide more accurate estimated results for the equi-depth histogram.

Theorem 2. *Set var_b^2 to represent the height-variance error of the estimated equi-depth histogram obtained from a simple random sample of n blocks, and var_t^2 to represent the height-variance error of the estimate results over the simple random sample of n tuples. Then the mean value of var_b^2 is equal to or less than that of var_t^2: $E(var_b^2) \leqslant E(var_t^2)$.*

Proof. The boundary of bucket B_i is determined by two p-quantiles: Y_i and Y_{i+1}. We set α_{ij} to represent the proportion of records in the j-th block that belongs to B_i according to its boundary, and set σ_i^2 to represent the variance of α_{ij} of every block. Then the mean value of the variance error is:

$$\begin{aligned}E(var_b^2) &= \frac{k}{N^2B^2}\sum_{i=1}^{k}E[(\widetilde{h_i}-h_i)^2]\\ &= \frac{k}{N^2B^2}\sum_{i=1}^{k}\sigma^2_{\widetilde{h_i}}\\ &= \frac{k}{N^2B^2}\sum_{i=1}^{k}n\times B^2\sigma_i^2\\ &= \frac{nk}{N^2}\sum_{i=1}^{k}\sigma_i^2. \end{aligned} \quad (10)$$

The tuple-level sampling can be considered as the special case of block-level sampling with block size of 1, so the variance error of estimate obtained from the tuple-level sampling with n tuples can also be computed through the above equation:

$$E(var_t^2) = \frac{nk}{N^2}\sum_{i=1}^{k}\sigma_{it}^2. \quad (11)$$

However, the variance σ_{it}^2 in the tuple-level sampling of n tuples is equal to or greater than that of the block-level sampling of n blocks, so we can get the conclusion that $E(var_b^2) \leqslant E(var_t^2)$. □

In order to efficiently utilize the data gotten during the course of the sampling, we adopt the block-level random sampling to estimate both the equi-width and equi-depth histograms in HEDC++. However, the estimate accuracy based on block-level sampling is influenced by the data layout during one block. If data during every block is randomly stored, then retrieving one block randomly from the data files is equal to retrieving B tuples randomly. On the other hand, if data in a block has correlation associated with attribute X, the sample size required to get the estimate result with given accuracy will be bigger than that of sampling on data with random layout. In the following subsection, we bound the error of the estimated histogram, and compute the relationship of required sample size and data correlation under a given error.

4.3 Bounding the Error and Sample Size

The main problem of equi-width histogram estimation is different from that of equi-depth histogram: the former focuses on estimating the histogram height of every bucket with fixed bucket boundary, while the latter focuses on estimating the boundary of every

bucket with fixed height. We propose different bounding methodologies for these two kinds of histograms.

4.3.1 Bounding Method for Equi-Width Histogram

In the previous subsection, we have modeled the equi-width histogram estimation as estimating the proportions of different buckets in the table. Given a bucket B_i, we construct a random variable X_{ij}, where $X_{ij} = P_{ij}$. The data blocks are of the same size, then the average of random variables in the population μ_i is the exact bucket proportion: $P_i = \mu_i = \frac{1}{N}\sum_{j=1}^{N} X_{ij}$. Consequently the problem can be transformed into estimating the average value of X_{ij} over all the blocks in the table. We use σ_i^2 to represent the variance of random variable X_{ij}: $\sigma_i^2 = \frac{1}{N}\sum_{j=1}^{N}(X_{ij}-\mu_i)^2$. σ_i^2 reflects how evenly the elements of bucket B_i are distributed over the blocks, and it can also reflect the correlation of data during one block to some extent. If the tuples are fairly distributed among the blocks, then the correlations of tuples during one block are small, and σ_i^2 will be small. And the opposite is also true.

During the sampling phase of HEDC++, blocks are randomly drawn from the data file without bias. Given bucket B_i, every block corresponds to a proportion value X_{ij}. So after the sampling phase we get a sample set $S = \{X_{i1}, X_{i2}, \ldots, X_{in}\}$ of size n, during which the random observations are independently and identically distributed (i.i.d.). According to the Central Limit Theorem (CLT) for averages of i.i.d. random variables, for large n the estimated proportion $\tilde{P}_i$ approximately obeys a normal distribution with mean value μ_i and variance σ_i^2/n. We construct a random variable Z by standardizing $\tilde{P}_i$: $Z = \frac{\tilde{P}_i - P_i}{\sigma_i/\sqrt{n}}$. Then according to the property of normal distribution, Z approximately obeys a standard normal distribution. Given a confidence level p, denote by z_p the p-quantile of the standard normal distribution, we have: $P\{|Z| \leqslant z_p\} \approx p$. It means that with probability p, we have:

$$|\tilde{P}_i - P_i| \leqslant \sigma_i z_p/\sqrt{n}. \tag{12}$$

Recall the relationship of bucket size and the bucket's proportion, we can get:

$$|\tilde{h_i} - h_i| \leqslant NB\sigma_i z_p/\sqrt{n}. \tag{13}$$

According to the height-variance error metric definition in (1), the bound of the error can be computed through:

$$err_{\text{bound}} = z_p k\sqrt{\frac{1}{kn}\sum_{i=1}^{k}\sigma_i^2}. \tag{14}$$

From (14), we can observe the elements that influence the estimate error of histogram. The error is directly proportional to the square root of $\sum_{i=1}^{k}\sigma_i^2$, which reflects the data correlation for constructing the histogram. Also it is inversely proportional to the square root of sample size. In most cases, the variance of the population σ_i^2 is not available, we adopt the variance of the sampled data $\tilde{\sigma_i}^2$ to compute err_{bound}, where $\tilde{\sigma_i}^2 = \frac{1}{n}\sum_{j=1}^{n}(X_{ij}-\tilde{\mu_i})^2$. According to the property of simple random sampling, $\tilde{\sigma_i}^2$ is a consistent estimate of σ_i^2[17]. If err_{bound} computed based on the existing sampling data is bigger than the required error err_{req}, we will have to retrieve more blocks. The extra sample size required for the desired error err_{req} is computed in a conservative way by assigning err_{req} to the error bound. According to the relationship between error bound and the sample size, we can compute the extra sample size:

$$b = \frac{err^2 - err_{\text{req}}^2}{err^2}n, \tag{15}$$

where err reflects the height-variance error of the estimated histogram over the initial sample data and is computed based on the data layout.

4.3.2 Bounding Method for Equi-Depth Histogram

As mentioned in the previous subsection, the estimate of equi-depth histogram can be modeled as quantile estimate. The estimate of quantiles is different from the estimation of functions of means, which is similar to the equi-width histogram estimate. In order to analyze and discuss the bounding method more clearly, we adopt the max-boundary error for the equi-depth histogram, which can reflect the quality of the estimate result more directly. Our bounding method can also be extended to the height-variance error with manageable modifications. The separator s_i is the p_i-quantile Y_{p_i} of the whole dataset, where $p_i = \frac{i-1}{k}$. The first separator s_1 is set to be a fixed value small enough to contain the minimum value of the domain in the first bucket, and the last separator s_{k+1} is set in the similar way.

After the sampling phase, HEDC++ retrieves n blocks randomly from the data file. According to the analysis of [18], the p_i-quantile of the sample data $\widetilde{Y_{p_i}}$ is the unbiased estimate for Y_{p_i}. In order to bound the error, we have to estimate the confidence interval for the quantile. However, developing the confidence interval for quantiles directly is very difficult, Woodruff inverted the usual confidence interval for the distribution function[19], and we adopt the method. Set $[Y_L, Y_H]$ to represent the confidence interval for Y_{p_i}, and set p_i' to represent the percentage of items in the sample data less than Y_{p_i}, then we can get: $P(Y_L < Y_{p_i} < Y_H) = P(L < p_i' < H)$. L is the percentage of items in the sample data less than Y_L, and H has the similar meaning. So we first compute the interval of $[L, H]$, and then transfer it to $[Y_L, Y_H]$.

Let α_{ij} be the percentage of items less than Y_{p_i} in the j-th block, and σ_i be the variance of α_{ij}. Though σ_i has different meaning from the variance defined in the equi-width histogram of the previous subsection, it can also quantify the effect of data layout during one block to the estimation. Woodruff gave the interval of $[L_i, H_i]$ based on cluster sampling as $[p_i - \delta, p_i + \delta]$[19], where

$$\delta^2 = \frac{B^2 n(N-n)}{N}\sigma_i^2. \quad (16)$$

Actually, the p_i-quantile Y_{p_i} is not known, so the estimated quantile $\widetilde{Y_{p_i}}$ is used to compute the variance σ_i^2. According to the max-boundary error metric definition for the equi-depth histogram in (2), we propose the estimate form of error bound:

$$err_{\text{bound}} = \max_i \left\{ \frac{k}{domain\_width}(Y_{Hi} - Y_{Li}) \right\}. \quad (17)$$

If err_{bound} computed based on the existing sampling data is bigger than the required error err_{req}, we will have to retrieve more blocks for the estimate. The computation of extra sample size b of the equi-depth histogram is more difficult than that of the equi-width histogram, because relationship of error bound and the sampling size cannot be computed directly. Deciding the exact extra sample size requires estimating the density function of Y_{pi}, which is a notoriously difficult statistical problem. In this paper, we assume that the sampling can reflect enough details of the density function, and adopt a conservative method to compute the extra sample size. We first compute the required confidence interval of p_i' based on err_{req}. The required left bound of the quantile is $Y_{\text{req}L} = \widetilde{Y_{p_i}} - err_{\text{req}} \times \frac{NB}{k} \times \frac{\widetilde{Y_{p_i}} - Y_L}{Y_H - Y_L}$, and the percentage of tuples smaller than $Y_{\text{req}L}$ is set to δ_L. We compute δ_H in the similar way, and choose $\delta_{\text{req}} = \max\{\delta_L, \delta_H\}$ as the required interval of the percentage. Based on (16), we can compute the extra sample size:

$$b = \frac{\delta^2 - \delta_{\text{req}}^2}{\delta_{\text{req}}^2} n. \quad (18)$$

4.4 Adaptive Sampling Method

HEDC++ adopts a sampling mechanism adaptive to the data correlation, which is illustrated in Algorithm 1. The input of Algorithm 1 includes two variables: the desired error err_{req} and the initial sample size r. We assume that the desired error is specified in terms of the height-variance error metric and the max-boundary metric for equi-width and equi-depth histograms respectively. The initial sample size is the theoretical size required to obtain estimate result with error err_{req} assuming uniform random sampling, which can be computed before the processing based on the analysis in [13]. We can conclude from (14) and (16) that in order to get the estimate result of the same accuracy, it requires more sampling blocks from data with correlated layout during one block than the random layout. After picking $\lceil \frac{r}{B} \rceil$ blocks, HEDC++ estimates the histogram based on the sampled data (lines 1 and 2). The estimation is implemented through one or two MapReduce jobs, and the details will be described in the next section. Then the error bound err is computed based on the analysis in the previous subsection (line 3), which is executed in the "Statistical Computing" module of HEDC++. If err is equal to or less than the desired error, it means that the estimate results satisfy the requirements and they are returned (lines 4 and 5). Otherwise, the extra required sample size b is computed based on the data layout in the initial samples (line 7). Then b blocks are retrieved from the data files and they are combined with the sampled blocks in the first phase (line 8). At last the final estimated histogram is computed on the combined sample set and returned (lines 9 and 10).

Algorithm 1. Adaptive Sampling Algorithm

Input: desired error in the estimated histogram: err_{req}; required sample size on the random layout: r

1 $S = \lceil \frac{r}{B} \rceil$ blocks randomly retrieved from the data file;
2 $\tilde{H} = HistogramEstimate(S)$;
3 $err = ErrorBound(S)$;
4 **if** $err \leqslant err_{\text{req}}$ **then**
5 return $\tilde{H}$;
6 **else**
7 $b = SamsizeBound(S)$;
8 $S = S \cup b$ blocks retrieved randomly from data file;
9 $\tilde{H} = HistogramEstimate(S)$;
10 return $\tilde{H}$;
11 **end**

During the adaptive sampling method, the extra sample size required for the desired error err_{req} is computed in a conservative way by assigning err_{req} to the error bound. Both the error bounds of equi-width and equi-depth histograms are computed based on the data layout. When the correlation of data grouped into one block is bigger, the error we bound will be larger and it requires more extra sample data. HEDC++ determines the required sample size adaptively according to the data layout.

5 Implementing over MapReduce

In this section we describe the implementing details of HEDC++ over the Hadoop MapReduce framework.

Hadoop[④] is one of the most popular open source platforms that support cloud computing. A Hadoop installation consists of one master node and many slave nodes. The master node, called JobTracker, is responsible for assigning tasks to the slave nodes and detecting the execution status of tasks. The slave node, called TaskTracker, executes tasks actually and reports status information to the JobTracker through heartbeat. In order to construct approximate histogram on big data, we make some necessary extensions to the original MapReduce framework. Though demonstrated on Hadoop, HEDC++ can also be implemented to other MapReduce platforms with straightforward modifications.

5.1 Extensions to the Original MapReduce

Building the exact histogram can be implemented through one original MapReduce job, however, constructing the approximate histogram has to meet two special requirements. The first requirement is to access the data blocks in a random way. During the original MapReduce framework, the data file are divided into a lot of splits, which are put into a scheduling queue. The size of a split is the same as the size of one block by default, and in this paper we assume adopting this default configuration to describe our solutions more clearly. However, our announcements and methods still work when the split size is not equal to the block size. The task scheduler on the JobTracker schedules every split in the queue to a task tracker. The scheduling is executed sequentially from the head of the queue, which is averse to the random sampling requirement of histogram estimate. In this paper, we make some extensions to the task scheduler by adding a shuffle module just before the scheduling. After the splits are put into the scheduling queue, we shuffle the splits and provide a random permutation of all the elements in the queue. Then all the splits are organized in a random way, and scheduling the first n splits sequentially is equal to sampling n blocks randomly from the data files (without replacement).

The second requirement is that after the sampled data is processed by the MapReduce job, a statistical computing module is needed, which estimates the data size of every bucket and computes the extra sample size required based on the outputs of all reducers. However, during the original MapReduce framework, the outputs of every reducer are written into a separate file at the end of a MapReduce job. Then these output files can be processed by the following MapReduce jobs in parallel. We add a merge module after the reduce tasks, which conducts the statistical computing by merging all the output files.

The approximation of the equi-width histogram focuses on estimating the bucket height with fixed boundaries, while the approximation of the equi-depth histogram focuses on estimating the separators of buckets with equal bucket height. The implementation of estimating these two histograms are described in the following subsections.

5.2 Function Design of Equi-Width Histogram

In this subsection, we describe the functions of the MapReduce jobs to construct the approximate equi-width histogram. When designing the functions, we take the following rules into consideration. First, network bandwidth is a kind of scarce resource when running MapReduce jobs[4], so we design the functions to decrease the quantity of intermediate results. We add combine functions after the mappers, which pre-aggregate the key-value pairs sent to the reducers. Secondly, during the MapReduce processing of estimating histograms, the number of map tasks is much larger than that of reducer tasks. So arranging more work to the map tasks helps increase the parallelism degree and reduce the execution time. During the statistical computing module, the estimation and sample size bounding require several statistical parameters, and we try to compute these parameters as early as possible by arranging more computing in the map function.

Approximating the equi-width histogram with specified error in HEDC++ involves one or two MapReduce jobs depending on the data layouts. The processings of the two jobs are the same except the reduce function. The first job processes the initial sampling data. If the error satisfies the specified requirement, then the estimated results are returned, else extra sampling data is needed and the extra data is processed in the second job. The map function is depicted in Algorithm 2. In this paper we assume the existence of a *getBucketID*() function, which computes the bucket ID based on the value of the column associated with the histogram (line 1). For every tuple in the block, the bucket ID is specified as the output key (line 2). The output value is a data structure called Twodouble, which contains two numbers of double type. The first double is used to compute the variable's mean value, and the second double is used to compute the variance in the reduce function. During the map function, the first double is set to 1 for every tuple, and the second double is set to zero (lines 3 and 4).

④http://hadoop.apache.org/, October 2012.

Algorithm 2: Map Function
Input: tuple t
Output: text *key*, twodouble *value*
1 $bucketID = getBucketID\ (t)$;
2 *key*.set(*bucketID*);
3 *value*.set(1, 0);
4 *output*.collect(*key*, *value*);

In order to reduce the cost of intermediate data transmission in the shuffle phase, we define a combine function, which is shown in Algorithm 3. The values belonging to the same bucket in the block are accumulated (lines 1∼4) and the proportion of every bucket in the block is computed (line 5). The first double of the output value is specified as the proportion of the bucket in this block, and the second double is specified as the square of the proportion (line 6). After the combine function, all the values belonging to the same bucket are sent to the same reducer. During the reduce function described in Algorithm 4, the two doubles *sum* and *quadraricsum* of the same bucket in the value list are accumulated respectively (lines 4∼8). If the reducer belongs to the second MapReduce job, it means that data processed in the first job is not enough to construct an approximate histogram with the specified error. In order to save the compute resource and estimation time, we make the processing incremental by utilizing the output results of the reducers in the first job. Suppose the data size of the first job is n_1, and the data size needed to estimate the histogram with the specified error is n. HEDC++ samples $n_2 = n - n_1$ blocks in the second job, and processes them through the map function. The sum of all the proportions for bucket i of the blocks is: $\sum_{j=1}^{n} X_{ij} = \sum_{j=1}^{n_1} X_{ij} + \sum_{j=n_1+1}^{n_1+n_2} X_{ij}$. Also the accumulation of the square sum is: $\sum_{j=1}^{n} X_{ij}^2 = \sum_{j=1}^{n_1} X_{ij}^2 + \sum_{j=n_1+1}^{n_1+n_2} X_{ij}^2$. The incremental computing is implemented through lines 9∼12. Then the estimated histogram size and variance are computed (lines 13 and 14).

Algorithm 3: Combine Function
Input: text *key*, iterator (twodouble) *values*
Output: (text *key*, twodouble *value*′)
1 **while** *values*.hasNext() **do**
2 twodouble *it* = *values*.getNext();
3 *sum*+ = *it*.get_first();
4 **end**
5 *prop* = *sum*/*B*;
6 *value*′.set(*prop*, *prop* ∗ *prop*);
7 *output*.collect(*key*, *value*′);

After all the reducers of the first job complete, the merge module collects all the output results of the reducers and conducts the statistical computing to bound the error and sample size. The error bound is computed through equation (line 14) based on the sum of variance of all the buckets. If the error is bigger than the specified error, then the number of extra sampling blocks is computed through (15).

Algorithm 4: Reduce Function
Input: text *key*, iterator (twodouble) *values*
Output: size estimate of bucket h_i, proportion variance σ_i^2
1 //n: number of blocks processed
2 //*sum*′: sum of the variables in the last job
3 //*quadraticsum*′: quadratic sum of the variables in the last job
4 **while** *values*.hasNext() **do**
5 twodouble *it* = *values*.getNext();
6 *sum*+ = *it*.get_first();
7 *quadraticsum*+ = *it*.get_second();
8 **end**
9 **if** the second job **then**
10 *sum* = *sum* + *sum*′;
11 *quadraticsum* = *quadraticsum* + *quadraticsum*′;
12 **end**
13 $h_i = N * B * sum/n$;
14 $\sigma_i = quadraticsum/n - sum * sum/n * n$;

5.3 Function Design of Equi-Depth Histogram

Different from the implementation of the equi-width histogram, the approximation of the equi-depth histogram over MapReduce is much more difficult. It requires two or three MapReduce jobs according to the data layout: the first job and the second job process the initial sampling data, and the third job processes extra sampling data if required. These three jobs can be classified into two kinds: the separator estimate job (*sepJob*) and the variance estimate job (*varJob*). The first job and the third job are *sepJob*s that compute the percentage of items less than every column value and then estimate the separators. The second job is *varJob*, which computes the variance of percentages of separators among the blocks. It is used to bound the error and compute the extra sample size.

The map function of *sepJob* scans all the sampled blocks. The value of the column of interest is set to be the output key, and the output value is set to be 1 for every key-value pair. In order to reduce the data transmission cost and the burden of the reducers, a combiner is designed to accumulate the number of items equal to a given column value. After the shuffle phase, all the pairs of the same column value are sent to the same reducer. The *sepJob* reduce function is depicted in Algorithm 5. For a given column value, the number

of items owning the same column value are computed (lines 3∼6). In order to estimate the separators, we have to compute the percentage of items with the column value less than the given value. So we accumulate the number of items with column value less than the column value being processed (line 7). If the executing job is the third job, then we accumulate the number of items with the results of the first job, which makes the processing and computing incremental (lines 8∼13). If the processing column value appeared in the first job, then we accumulate the stored results. Otherwise, it means that the processing column value is new, and there is no corresponding result for it in the first job. We adopt the result of the first column value in the first job which is less than the new column value (line 10). The column value is arranged as the output key. The output value is the accumulative number of items which is less than the key (lines 14 and 15). During the merge module of *sepJob*, the percentage of items less than every column value is computed, and the separators are estimated based on the percentages.

Algorithm 5: Reduce Function(*SepJob*)
Input: text *key*, iterator (long) *values*
Output: (text *key*, long *value′*)
1 //*sum′*: sum of the variables in the last job
2 //*sum_acc*: sum of the variables in the last iteration
3 **while** *values*.hasNext() **do**
4 *it* = *values*.getNext();
5 *sum*+ = *it*;
6 **end**
7 *sum_acc* = *sum* + *sum_acc*;
8 **if** the third job **then**
9 **if** *sum′*==0 **then**
10 *sum′* = *getAdjSum*();
11 **end**
12 *sum_acc* = *sum_acc* + *sum′*;
13 **end**
14 *value′*.set(*sum_acc*);
15 *output*.collect(*key*, *value′*);

After the first job, the initial separators are estimated, and the next job is to compute the necessary statistical parameters to bound the error based on the separators. The map function of *varJob* scans the sampled blocks, and determines the bucket for every item based on the separators. The output key is set to be the right separator of the bucket, and the value is set to 1. Actually in the implementation of HEDC++, we make some extensions to the original file input mechanism of MapReduce, and adopt the output results of every combine function of *sepJob* as the mapper's input. The extensions are aimed to reduce the computation and I/O cost, and then reduce the running time of the estimate.

A combiner is also designed after the mapper to compute the percentage of items less than every separator for each block, which is depicted in Algorithm 6. The number of items during the bucket with the given separator as the right boundary is accumulated (lines 2∼5). In order to compute the number of items less than the separator, we also have to accumulate the results from the last iteration (line 6). The separator is arranged as the output key. The output value is a structure containing two numbers of doubles, which includes the percentage of items less than the given separator and the square of the percentage (lines 7∼9). Then the reduce function described in Algorithm 7 computes the percentage variance among all the blocks for every separator. Given a separator value, the sum of *prop*s and their squares are accumulated (lines 2∼6). The variance is computed through the mean values of the percentage and its square (line 7).

Algorithm 6: Combine Function(*varJob*)
Input: text *key*, iterator (long) *values*
Output: (text *key*, twodouble *value′*)
1 //*sum_acc*: sum of the variables in the last iteration
2 **while** *values*.hasNext() **do**
3 *it* = *values*.getNext();
4 *sum*+ = *it*;
5 **end**
6 $sum_{acc} = sum_{acc} + sum$;
7 $prop = sum_{acc}/B$;
8 *value′*.set(*prop*, *prop* ∗ *prop*);
9 *output*.collect(*key*, *value′*);

Algorithm 7: Reduce Function (*varJob*)
Input: text *key*, iterator (twodouble) *values*
Output: proportion variance σ_i^2
1 //*n*: number of blocks processed
2 **while** *values*.hasNext() **do**
3 twodouble *it* = *values*.getNext();
4 *sum*+ = *it*.get_first();
5 *quadraticsum*+ = *it*.get_second();
6 **end**
7 $\sigma_i = quadraticsum/n - sum * sum/n * n$;

After the reduce function of *varJob*, all the separators are decided, and the output results are put together to the statistical computing module. We bound the error of the estimate results based on the methods described in Subsection 4.3.2. If the error is bigger than the required error, then we decide the extra sample size

in a conservative way, and the third job is triggered. After the third job, the separator estimate is improved with the extra sample data and used as the estimate of the equi-depth histogram.

6 Performance Evaluation

In this section, we evaluate the performance of HEDC++ in terms of sample size required and the running time to get an estimate with specified error. We compare our adaptive sampling approach in HEDC++ against two other sampling methods by evaluating their performances on datasets with different data correlations and block sizes. We also evaluate the scalability of HEDC++ from two aspects: the data size and cluster scale. All the experiments are implemented on Hadoop 0.20.2.

6.1 Experiment Overview

Our experiment platform is a cluster of 11 nodes connected by a 1 gigabit Ethernet switch. One node serves as the namenode of HDFS and jobtracker of MapReduce, and the remaining 10 nodes act as the slaves. Every node has a 2.33 G quad-core CPU and 7 GB of RAM, and the disk size of every node is 1.8 T. We set the block size of HDFS to 64 M, and configure Hadoop to run two mappers and one reducer per node.

We adopt two metrics to evaluate the performance: sample size and running time. Sample size is computed by the histogram estimate algorithm, which represents the number of tuples needed to be sampled to get the histogram estimate with a specified error. Running time is the time cost to get the final estimate. We compare the performance of HEDC++ with two histogram estimate methods called TUPLE and DOUBLE, which adopt different sampling mechanisms. TUPLE conducts a tuple-level random sampling, which is similar to the sampling method in [15]. It accesses all the blocks in the data file and retrieves tuples randomly from every block. DOUBLE adopts a block-level sampling method. Its difference from HEDC++ is the sample size computing method, which is an iterative approach originates from [12]. If the existing sampled data is not enough to complete the estimate, DOUBLE repeatedly doubles the sample size and executes the estimation.

The dataset we adopt in the experiment is the page traffic statistics of Wikipedia hits log. It contains seven months of hourly pageview statistics for all articles in Wikipedia, and includes 320 GB of compressed data (1 TB uncompressed)[⑤]. Every tuple in the dataset contains four columns: *language*, *page name*, *page views* and *page size*. During the experiment of estimating the equi-width histogram, we choose *language* as the key column to construct histogram to reflect the data distribution on 10 languages. So the number of buckets of the histogram is $k = 10$. We set the confidence level of bounding the error to 95%, and set the specified error to 0.05. During the experiment of the equi-depth histogram estimate, we choose *page size* as the key column, which provides reference information for the result size estimate of the range query on the *page size* column. The number of buckets k is also set to 10, and its required error is also set to 0.05.

6.2 Effect of Data Correlation

In this subsection, we evaluate the performances of three approaches on datasets with different correlations. We change the data layout of the real dataset to make different degrees of correlations. We adopt a metric C to measure the data correlation, which has the similar spirit of the *cluster degree* in [14]. According to the analysis of (14) and (16), $var\_sum = \sum_{i=1}^{k} \sigma_i^2$ reflects the data correlation in blocks. var_sum is maximized when the tuples are fully ordered by the column of interest, and is minimized when the data layout is random. We normalize the metric with respect to var_sum for $C = 1$ when the correlation is the biggest, and $C = 0$ when tuples are laid randomly.

Fig.2 illustrates the sample size and running time of three approaches on different data correlations for EWH (equi-width histogram), and Fig.3 depicts the corresponding results for EDH (equi-depth histogram). The effect of data correlation for these two kinds of histograms has similar trends. For the approaches with block-level sampling, the sample size computed is always the number of blocks. In order to compare these three approaches conveniently, we show the number of tuples in the experimental results. We can see that the data correlation does not affect the required sample size of TUPLE. This is because that TUPLE conducts a random sampling on every block in the data files, which is equal to conduct a tuple-level random sampling on the whole data. Though the sample size of TUPLE is the smallest, its running time is longer than HEDC++ and DOUBLE. Two reasons may explain the results. First, TUPLE has to access all the blocks during the sampling phase. For blocks that are not in the local disk of the TaskTracker, data transmission is needed, and this transmission cost of TUPLE is the same as that of processing all the data. Secondly, the number of mapper tasks in TUPLE is also the same as that of processing all the data, and the sum of map tasks' start

[⑤] http://aws.amazon.com/datasets/2596, November 2012.

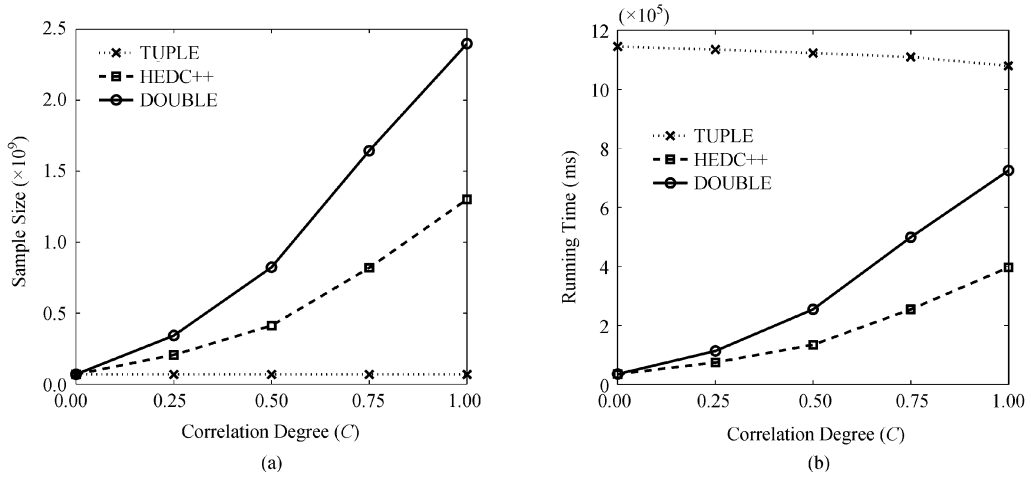

Fig.2. Effect of data correlation for EWH. (a) Sampling size. (b) Running time.

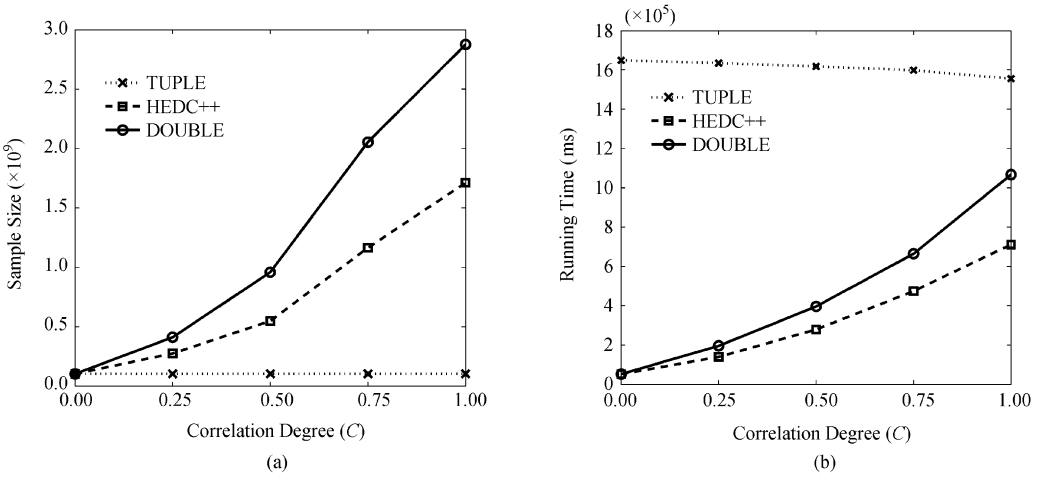

Fig.3. Effect of data correlation for EDH. (a) Sampling size. (b) Running time.

time is not reduced. When the degree of correlation is equal to zero, the three approaches retrieve almost the same sample size, and HEDC++ and DOUBLE pick a little more tuples because their sampling unit is the block. As the degree of correlation increases, the sample size of DOUBLE is larger than that of HEDC++. For $C = 0.5$ of the equi-width histogram, DOUBLE requires the sample size twice larger than HEDC++, and this is the worst case for DOUBLE. In addition, DOUBLE has to increase the sample size and process the extra sampled data iteratively, so the time it costs is more than HEDC++. HEDC++ can determine the sample size adaptively according to the data layout and completes the histogram estimate within acceptable time.

6.3 Effect of Block Size

The default block size of Hadoop is 64 M. During the practical applications, the block size can be changed to adapt to different data sizes. In this subsection, we evaluate the effect of different block sizes on the estimator's performance. We adopt the real Wikipedia hit log data directly in this experiment without any changes. For the real dataset, log records within the same hour are ordered by the language, so the data correlation is between the random layout and the fully ordered layout. The results are shown in Fig.4 and Fig.5, we run the three approaches with five block sizes: 16 M, 32 M, 64 M, 128 M and 256 M. The block size does not affect the sample size of TUPLE because of its sampling level. Given the data file size, bigger block size results in less blocks, then less map tasks, and the sum of the start time of all the map tasks will be shorter. However, the processing time of every map task gets longer because of bigger blocks. So there is a tradeoff between the running time of TUPLE and the block size. In our real dataset, logs during one hour are stored in a data file, and the data size of every hour is about 65 M. So the correlation degrees of block size 16 M and 32 M are

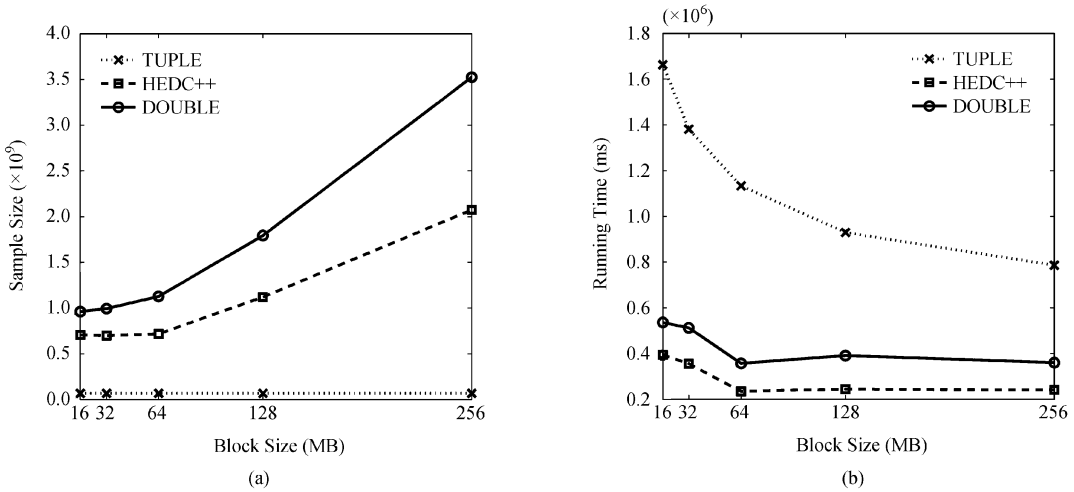

Fig.4. Effect of block size for EWH. (a) Sampling size. (b) Running time.

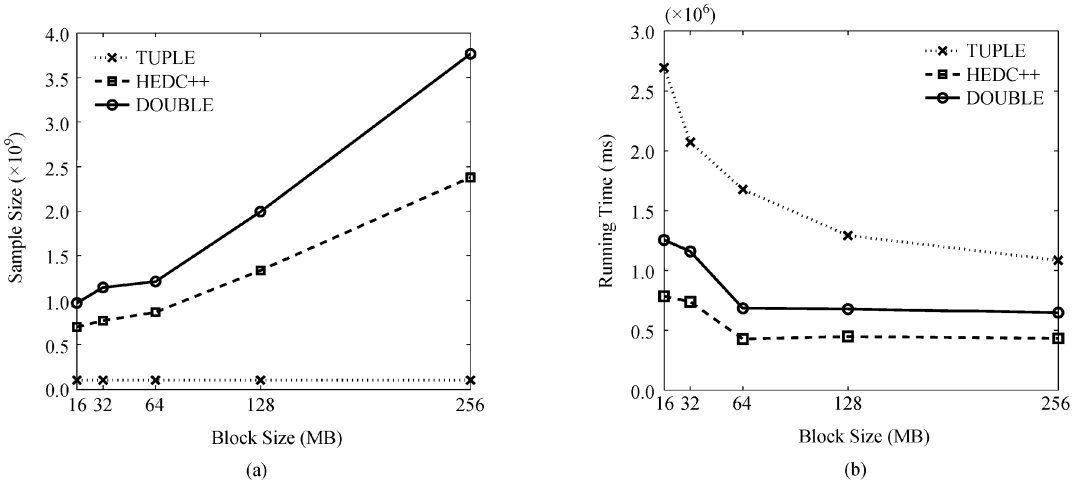

Fig.5. Effect of block size for EDH. (a) Sampling size. (b) Running time.

much bigger than the other three block sizes, and more sample blocks are required for HEDC++ and DOUBLE. When the block size is bigger than 32 M, the data correlation decreases gradually, so the required sample blocks of HEDC++ and DOUBLE decrease correspondingly. However, the sample tuples increase as the block size increases, and the processing time of every map task gets longer. So there is also a tradeoff between the block size and the running time of the estimator with block-level sampling. We can see that in this experiment, when the block size is 64 M, the running time of HEDC++ and DOUBLE gets shortest. In general, it costs less running time for HEDC++ to construct the approximate histograms than TUPLE and DOUBLE.

6.4 Scalability Evaluation

We evaluate the scalability by varying the data scale and node number of the testbed cluster for both the equi-width and equi-depth histogram estimate. We provide the results of either equi-width or equi-depth histogram for one experiment, since the results are similar for them. Fig.6 shows the required sample size and running time of the estimators on a 10-node cluster with different data sizes. For HEDC++ and DOUBLE, the sample size does not increase directly proportional to the data size. According to (14), the error bound is not effected directly by the data size. It is associated with the variance sum of all the buckets. The variance sum does not change a lot as the data size increases in our dataset, so the data size needed to be sampled does not grow significantly as the data size increases. The running time increases a little because the sampling time gets a little longer as the data size increases. For TUPLE, though the sample size maintains almost constant, the running time still increases because more blocks have to be accessed as the data size increases.

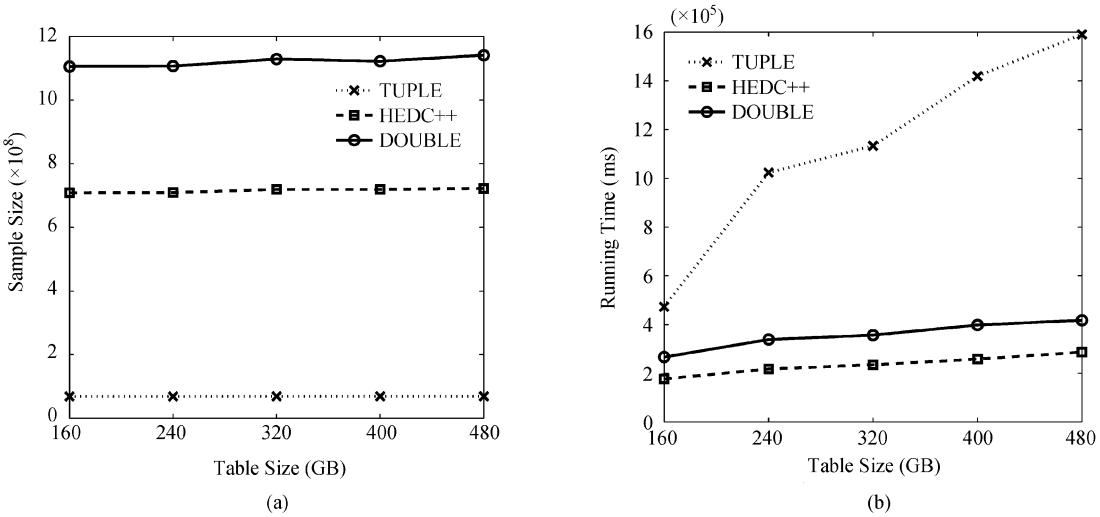

Fig.6. Scale-up with data size for EWH. (a) Sampling size. (b) Running time.

Given the dataset to be processed, the required sample size does not change as the cluster scale changes, so we only show the running time when varying the number of nodes in the cluster. The experimental results of the equi-depth histogram are illustrated in Fig.7. We can see that as the number of nodes increases, the running time of these three approaches decreases, and the speedup originates from the speedup of MapReduce processing. We can conclude from the results that HEDC++ has scalability for both data size and cluster scale.

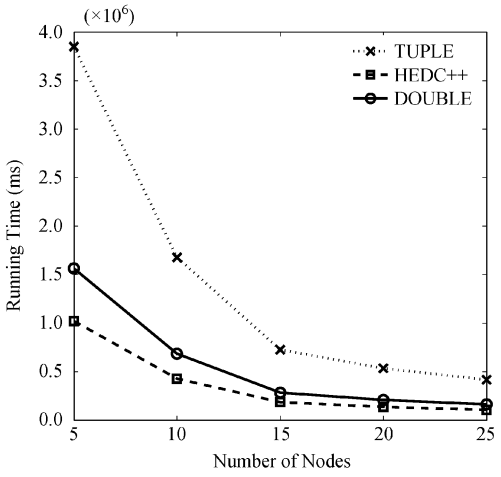

Fig.7. Scale-up with cluster scale for EDH.

7 Conclusions and Future Work

In this paper, we proposed an extended histogram estimator called HEDC++, which focuses on estimating the equi-width and equi-depth histograms for data in the cloud. This problem is challenging to solve in the cloud for two reasons: 1) Data in the cloud is always organized into blocks, which makes it inefficient to conduct uniform random sampling, so the estimator should leverage the sampling efficiency and estimation accuracy; 2) The typical processing mode of the cloud is batch processing, which is adverse to the requirements of estimator to return "early results". We designed the processing framework of HEDC++ on extended MapReduce, and proposed efficient sampling mechanisms which include designing the sampling unit and determining the sampling size adaptive to the data layout. The experimental results of our techniques on Hadoop show that HEDC++ can provide efficient histogram estimate for data of various layouts in the cloud.

For future work, we will research on the histogram estimate for dynamic data and real-time data in the cloud. As tuples in the dataset are changing continuously, the precomputed approximate histograms become outdated and should be updated correspondingly. We will explore techniques to propagate the updates to histograms to make the summarization effective without affecting the performance of cloud databases.

References

[1] Thusoo A, Sarma J, Jain N *et al.* Hive: A warehousing solution over a Map-Reduce framework. In *Proc. the 35th Conference of Very Large Databases* (*VLDB2009*), August 2009, pp.1626-1629.

[2] Olston C, Reed B, Srivastava U *et al.* Pig latin: A not-so-foreign language for data processing. In *Proc. the ACM Int. Conf. Management of Data* (*SIGMOD2008*), June 2008, pp.1099-1110.

[3] Abadi D J. Data management in the cloud: Limitations and opportunities. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009, 32(1): 3-12.

[4] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In *Proc. the 6th Symposium on Opearting Systems Design and Implementation* (*OSDI2004*), December 2004, pp.137-150.

[5] Blanas S, Patel J, Ercegovac V *et al.* A comparison of join algorithms for log processing in MapReduce. In *Proc. the ACM Int. Conf. Management of Data* (*SIGMOD2010*), June 2010, pp.975-986.
[6] Okcan A, Riedewald M. Processing theta-joins using MapReduce. In *Proc. the ACM International Conference on Management of Data* (*SIGMOD2011*), June 2011, pp.949-960.
[7] Shi Y J, Meng X F, Wang F S *et al.* HEDC: A histogram estimator for data in the cloud. In *Proc. the 4th Int. Workshop on Cloud Data Management* (*CloudDB2012*), Oct. 29-Nov. 2, 2012, pp.51-58.
[8] Poosala V, Ioannidis Y E, Haas P J, Shekita E J. Improved histograms for selectivity estimation of range predicates. In *Proc. the ACM International Conference on Management of Data* (*SIGMOD1996*), June 1996, pp.294-305.
[9] Ioannidis Y E. The history of histograms (abridged). In *Proc. the 29th Conference of Very Large Databases* (*VLDB2003*), September 2003, pp.19-30.
[10] Piatetsky-Shapiro G, Connell C. Accurate estimation of the number of tuples satisfying a condition. In *Proc. the ACM International Conference on Management of Data* (*SIGMOD1984*), June 1984, pp.256-276.
[11] Gibbons P B, Matias Y, Poosala V. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems*, 2002, 27(3): 261-298.
[12] Chaudhuri S, Motwani R, Narasayya V. Random sampling for histogram construction: How much is enough? In *Proc. ACM International Conference on Management of Data* (*SIGMOD1998*), June 1998, pp.436-447.
[13] Chaudhuri S, Motwani R, Narasayya V. Using random sampling for histogram construction. Technical Report, Microsoft, http://citeseerx.ist.psu.edu/showciting?cid=467221, 1997.
[14] Chaudhuri S, Das G, Srivastava U. Effective use of block-level sampling in statistics estimation. In *Proc. ACM International Conference on Management of Data* (*SIGMOD2004*), June 2004, pp.287-298.
[15] Jestes J, Yi K, Li F F. Building wavelet histograms on large data in MapReduce. In *Proc. the 37th International Conference of Very Large Databases* (*VLDB2011*), August 29-September 3, 2011, pp.109-120.
[16] Mousavi H, Zaniolo C. Fast and accurate computation of equi-depth histograms over data streams. In *Proc. the 14th International Conference on Extending Database Technology* (*EDBT2011*), March 2011, pp.69-80.
[17] Cochran W G. Sampling Techniques. John Wiley and Sons, 1977.
[18] Francisco C A, Fuller W A. Quantile estimation with a complex survey design. *The Annals of Statistics*, 1991, 19(1): 454-469.
[19] Woodruff R S. Confidence intervals for medians and other position measures. *Journal of the American Statistical Association*, 1952, 47(260): 635-646.

Ying-Jie Shi received the B.S. degree from Shandong University, Jinan, in 2005, and M.S. degree from Huazhong University of Science and Technology, Wuhan, in 2007, both in computer science and technology. She is currently a Ph.D. candidate of Renmin University of China, Beijing. Her research interests include cloud data management and online aggregations of big data.

Xiao-Feng Meng is a full professor at School of Information, Renmin University of China, Beijing. He received a B.S. degree from Hebei University, M.S. degree from Renmin University of China, Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, all in computer science. He is currently the vice dean of School of Information, Renmin University of China. He is the secretary general of Database Technique Committee of the China Computer Federation (CCF DBTC). His research interests include Web data management, Cloud data management, mobile data management, XML data management, flash-aware DBMS, privacy protection in mobile Web, and social computing. He has published over 100 papers in refereed international journals and conference proceedings including IEEE TKDE, VLDB, SIGMOD, ICDE, EDBT, etc. He has served on the program committee of SIGMOD, ICDE, CIKM, MDM, DASFAA, etc., and the editorial board of Journal of Computer Science and Technology (JCST) and Frontiers of Computer Science (FCS).

Fusheng Wang is an assistant professor in the Department of Biomedical Informatics, adjunct assistant professor in the Department of Mathematics and Computer Science, and a senior research scientist at the Center for Comprehensive Informatics, Emory University, U.S.A. Prior to joining Emory University, he was a research scientist at Siemens Corporate Research from 2004 to 2009. He received his Ph.D. in computer science from University of California, Los Angeles, in 2004. His research interests include big data management, biomedical imaging informatics, data integration, spatial and temporal data management, sensor data management and processing, natural language processing, and data standardization. He has published over 70 research papers in international conferences, journals and book chapters. He received the best paper award at ICDCS 2011. He has served as program committee for nearly 20 international conferences and workshops. He was the program chair of the first Extremely Large Databases Conference at Asia, Beijing, 2012, program co-chair of the 4th International Workshop on Cloud Data Management (CloudDB 2012), and co-chair of CloudDB 2013.

Yan-Tao Gan received the B.S. degree in computer science and technology from Renmin University of China in 2012. She is currently a master candidate of Renmin University of China. Her research interests include cloud data management and online aggregations of big data.