HEDC: A Histogram Estimator for Data in the Cloud

Yingjie Shi[1] Xiaofeng Meng[1] Fusheng Wang[2] Yantao Gan[1]

[1]School of Information, Renmin University of China, Beijing, China
[2]Department of Biomedical Informatics, Emory University, Atlanta, USA

[1]{shiyingjie,xfmeng,ganyantao19901018}@ruc.edu.cn [2]fusheng.wang@emory.edu

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 database 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 plan, and generate basic statistics useful for guaranteeing the load balance of query processing in the cloud. Since it is too expensive to construct the exact histogram on massive data, building the 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, a Histogram Estimator for Data in the Cloud. We design a 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.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query processing*

General Terms

Algorithms, Design, Performance

Keywords

histogram estimate, sampling, cloud computing, MapReduce

1. INTRODUCTION

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 HBase[1], Hive[20], Pig[17], Cassandra[2] and others now attract considerable enthusiasm from both the industry and academia. Compared to the relational DBMS with sophisticated optimization techniques, cloud data management system is newly emerging and there is ample room for performance improvement of complex queries[6]. As the efficient summarization of data distribution and statistical information, histogram is 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[12], 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, [7] proposed four different MapReduce execution plans applicable to different data distributions. However, how to select the most efficient execution plan adaptively is not addressed, and histogram can offer great guidance to this problem. Secondly, histograms contain basic statistics useful for load balancing. Load balance is crucial to the performance of query in the cloud, which is always processed in parallel. In the processing framework of MapReduce, output results of the mappers are partitioned 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 help to design the hash function to guarantee load balance. Thirdly, in the processing of joins, summarizing the data distribution in histogram is useful for reducing the data transmission cost, which is one of the scarce resources in the cloud. Utilizing the histogram constructed on join key can help prevent sending the tuples that do not satisfy the join predicate to the same node[16]. In addition, histograms are also useful in providing various estimates in the cloud, such as query progress estimate, query size estimate and results estimate, etc. Such estimate plays an important role in pre-execution user-level feed back, task scheduling and resource allocation. However, it is too expensive and impractical to construct a precise histogram before the queries due to the massive data volume in the cloud. 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 classic work will not suffice. Cloud is typically a distributed environment, it brings parallel processing, data dis-

CloudDB'12, October 29, 2012, Maui, Hawaii, USA.

tribution, 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 systems[3]. Block is the transmission and the processing unit in the cloud, and block based data organization increases 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, the correlation of tuples in one block may affect the estimate accuracy. To effectively build statistical estimators with blocked data, the 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 a histogram estimator called HEDC, which focuses on constructing approximate equi-width histograms for data in the cloud. According to the rule in which the tuples are partitioned into buckets, histograms can be classified into several types, such as equi-width histogram, equi-depth histogram, spline-based histogram, etc[19]. Equi-width histogram is one of the histograms that are used commonly and easy to maintain. HEDC approximates the histogram of 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 of our work include:

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

2. We adopt 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;

3. 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;

4. We implement HEDC on Hadoop and conduct comprehensive experiments, the results show HEDC's efficiency in histogram estimating, and verity it's scalability as both data volume and cluster scale increase.

The rest of the paper is organized as follows. In Section2 we summarize the related work in the area. In Section3 we introduce the main workflow and architecture of HEDC. Section4 discusses the problem modeling and statistical issues, which includes the sampling mechanism and histogram estimate algorithm. The implementing details of HEDC is described in Section5, we also discuss how to make the processing incremental by utilizing the existing results. The performance evaluation is given in Section6, followed by the conclusions and future work.

2. RELATED WORK

Histogram plays an important role in cost-based query optimization, approximate query and load balancing, etc. [14] 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, [19] 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, it's proposed in [18] 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 works into two categories by the sampling mechanisms. Approaches in the first category adopted uniform random sampling[13, 10, 9], which samples the data with tuple level. [13] focused on sampling-based approach for incremental maintenance of approximate histograms, and it also computed the bound of required sample size based on a uniform random sample of the tuples in a relation. Surajit et al. further discussed the relationship of sample size and desired error in equi-depth histogram, they proposed a stronger bound which lends to ease of use[10]. [9] discussed how to adapt the analysis of [10] to other kinds of histograms. The above approaches assumed uniform random sampling. Approaches of the second category constructed histograms through block-level sampling[10, 8]. [10] adopted an iterative cross-validation based approach to estimate the histogram with specified error, the sample size was doubled once the estimate result didn't arrive at the desired accuracy. [8] proposed a two-phase sampling method based on cross-validation, in which the sample size required was determined based on the initial statistical information of the first phase. This approach reduced the number of iterations to compute the final sample size, and consequently processing overhead. However, the tuples it sampled may be 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, adapting them to the cloud environment requires sophisticated considerations.

There is less work on constructing the histogram in the cloud. Authors of [16] focused on processing theta-joins in the MapReduce framework, 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 was built by scanning the whole table, which is expensive for big data. Authors of [15] proposed a method for constructing approximate wavelet histograms over the MapReduce framework. Approach of this paper retrieved tuples from every block randomly and sent the outputs of mappers at certain probability, which aimed 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 can not be ignored. Though block locality is considered during the task scheduling of MapReduce framework, there exist blocks that have to be transferred from remote node to the mapper processing node, we believe there is room to reduce this data transmission cost.

3. OVERVIEW OF HEDC

Constructing the exact histogram of data involves one original Map-Reduce job, which is shown in the solid rectangles of Figure1. 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, 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. 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 accu-

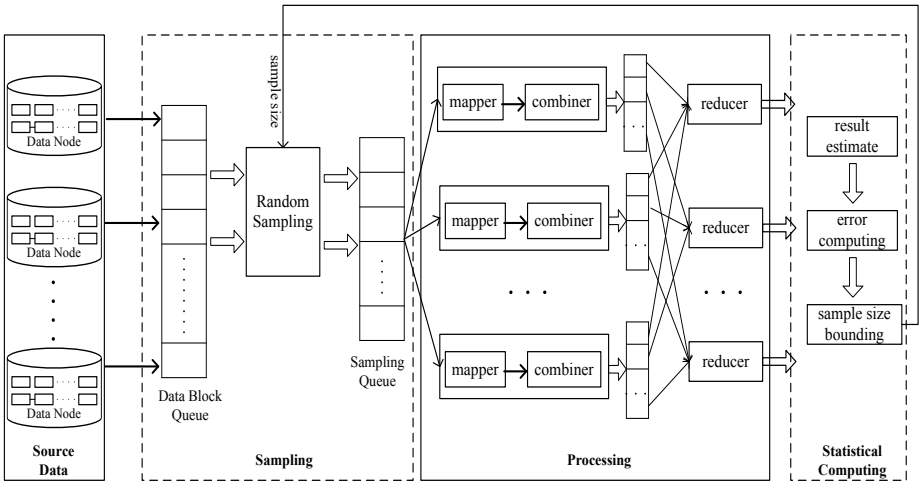


Figure 1: Architecture of HEDC

racy to significantly reduce the time it takes to get efficient data summarization over large data sets. However, providing approximate results with statistical significance in the cloud requires to solve the following challenging problems. First, data distributed in the cluster should be randomly sampled to guarantee the accuracy of estimated results. For reasonably large data sets, sampling at tuple level is expensive, while sampling at block level brings in data correlation. We design a block-level sampling algorithm that adapts to 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. Figure1 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, a specified number of blocks are retrieved randomly from this queue as the MapReduce job's input. These sampled blocks are processed by the map tasks, which compute the bucket ID for every tuple and partition the output key-value pairs based on bucket ID. We also design the combine function executed on the partitioned output to aggregate values of the same bucket, so the intermediate results transmitted in the network is further reduced. All the key-value pairs belonging to the same bucket are sent to the same reducer, and the reducer computes the bucket size of the histogram. However, the output result 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. Else, 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 second MapReduce job. During the statistical computing module, the outputs are merged with the first 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 histogram estimate into a statistical problem, 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 definition is as follows:

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

The histogram summarizes the data distribution on attribute X, set P_i to represent the proportion of tuples belonging to Bi, we can get $|B_i| = N * B * P_i$. Consider the tuples of one bucket as a category, then estimating the histogram can be modeled as estimating the proportion of the different categories in the table.

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. To measure the difference between the approximate histogram and the exact histogram, we adopt the standard error metric in the literature, which is defined as:

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

It's the standard deviation of the estimated bucket size from the actual number of elements in each bucket, normalized with respect to the mean bucket size. 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 an algorithm to bound this error based on the sampled data, the details are described in Section4.3.

4.2 Sampling Unit

Sampling is a standard technique for constructing approximate summaries of data, and most of the works 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, also the block is the unit of data transmitting in the network. Picking tuple-level random sampling from data in such organization is very expensive. In the worst case, retrieving n tuples may cause a full scan of n blocks. Secondly, block is also the processing unit of MapReduce framework, 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 Theorem1 that with the same data transmission cost, block-level sampling will provide more accurate estimated 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 $\tilde{h_{ib}} = N*B*\tilde{P_{ib}}$, and $\tilde{h_{it}} = N*B*\tilde{P_{it}}$. Then both $\tilde{h_{ib}}$ and $\tilde{h_{it}}$ are unbiased estimate of the bucket size, and the variance of $\tilde{h_{ib}}$ is equal or less than that of $\tilde{h_{it}}$: $VAR(\tilde{h_{ib}}) \leq VAR(\tilde{h_{it}})$.*

PROOF. The block-level random sampling has the same spirit with cluster sampling in the statistical terms[11], with the cluster size equal to the block size. Set P_{ij} to represent the proportion of elements belonging to B_i in the jth block, then the proportion of B_i is: $P_i = \frac{1}{N}*\sum_{j=1}^{N} P_{ij}$. According to the properties of proportion estimate on cluster sampling, we can get $\tilde{P_{ib}} = \frac{1}{n}*\sum_{j=1}^{n} P_{ij}$ 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:

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

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 population proportion, and the variance of $\tilde{P_{it}}$ is:

$$VAR(\tilde{P_{it}}) = \frac{NB-n}{NB-1}*\frac{P_i(1-P_i)}{n} \tag{3}$$

Consequently, the design effect $deff$ is:

$$\frac{VAR(\tilde{h_{ib}})}{VAR(\tilde{h_{it}})} = \frac{N^2B^2VAR(\tilde{P_{ib}})}{N^2B^2VAR(\tilde{P_{it}})} = \frac{NB-nB}{NB-n}*\frac{\sum_{j=1}^{N}(P_{ij}-P_i)^2}{NP_i(1-Pi)} \tag{4}$$

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

$$NB-nB \leq NB-n. \tag{5}$$

The numerator of the right part in equality(4) can be derived as:

$$\begin{aligned}\sum_{j=1}^{N}(P_{ij}-P_i)^2 &= \sum_{j=1}^{N}(P_{ij}^2-2P_iP_{ij}+P_i^2)\\ &= \sum_{j=1}^{N}P_{ij}^2-2NP_i^2+NP_i^2 = \sum_{j=1}^{N}P_{ij}^2-NP_i^2\end{aligned} \tag{6}$$

P_{ij} is the proportion of tuples belonging to bucket B_i in the jth block, so $0 \leq P_{ij} \leq 1$, then according to equality(6) we can get:

$$\sum_{j=1}^{N}(P_{ij}-P_i)^2 \leq \sum_{j=1}^{N}P_{ij}-NP_i^2 = NP_i-NP_i^2 \tag{7}$$

According to inequality (5) and (7), we can conclude that:

$$\frac{VAR(\tilde{h_{ib}})}{VAR(\tilde{h_{it}})} \leq 1 \tag{8}$$

□

In order to efficiently utilize the data gotten during the course of the sampling, we adopt the block-level random sampling to estimate histogram in HEDC. However, the estimate accuracy based on block-level sampling is directly 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 during 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 standard error of the estimated histogram, and compute the relationship of required sample size and data correlation under a given standard error.

4.3 Bounding the Error and Sample Size

In the previous subsection, we have modeled the 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 reflects 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 $S = \{X_{i1}, X_{i2}, ..., X_{in}\}$ of size n, during which the random observations are independently and identically distributed(iid). According to the Central Limit Theorem(CLT) for averages of iid 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: $P\{|Z| \leq z_p\} \approx p$. It means that with probability p, we have:

$$|\tilde{P_i}-P_i| \leq \sigma_i z_p/\sqrt{n} \tag{9}$$

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

$$|\tilde{h_i}-h_i| \leq NB\sigma_i z_p/\sqrt{n} \tag{10}$$

According to the error metric definiton of estimated histogram in equation(1), the bound of the error can be computed through:

$$err_{bound} = z_p k\sqrt{\frac{1}{kn}\sum_{i=1}^{k}\sigma_i^2} \tag{11}$$

From equation(11), 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 the $error_{bound}$, 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[11].

4.4 Adaptive Sampling Method

HEDC adopts a sampling mechanism adaptive to the data correlation, which is illustrated in Algorithm1. The input of Algorithm1 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 standard error metric defined in Section4.1. The initial sample size is the theoretical size required to obtain estimate result with error err_{req} assuming uniform random sampling, it can be computed before the processing based on the analysis in [9]. We can conclude from equation(11) 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(1-2). The estimation is implemented through one MapReduce job, and the details will be described in the next section. Then the error bound err is computed based on the analysis in Section4.3(3), which is executed in the "Statistical Computing" module of HEDC. If err is equal or less than the desired error, it means that the estimate results satisfies the requirements and they are returned(4-6). Otherwise, the extra required sample size b is computed based on the data layout in the initial samples(8), we will introduce the computation method later. Then b blocks are retrieved from the data files and they are combined with the sampled blocks in the first phase(9). At last the final estimated histogram is computed on the combined sample set and returned(10-11).

Algorithm 1: Adaptive Sampling Algorithm

```
   input: The desired error in estimated histogram: err_req
          The required sample size on the random layout: r
 1 S=⌈r/B⌉ blocks randomly retrieved from the data file;
 2 H̃=HistogramEstimate(S);
 3 err=ErrorBound(S);
 4 if err ≤ err_req then
 5     return H̃;
 6 end
 7 else
 8     b=SamsizeBound(S);
 9     S=S∪b blocks retrieved randomly from data file;
10     H̃=HistogramEstimate(S);
11     return H̃;
12 end
```

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:

$$b = \frac{err^2 - err_{req}^2}{err^2}\lceil \frac{r}{B} \rceil \tag{12}$$

err reflects the standard error of the estimated histogram over the initial sample data, it is computed based on the data layout. When the correlation of data grouped into one block is bigger, then err will be larger, and it requires more 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 the JobTracker, is responsible for assigning tasks to the slave nodes and detecting the execution status of tasks. The slave node, called the 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 modification.

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 to 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 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, an 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 ending 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.

5.2 Function Design

In this subsection, we describe the functions of the MapReduce jobs to construct the approximate histogram. When designing the functions, we take the following rules into consideration. First, network bandwidth is scarce resource when running MapReduce jobs[12], 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 more 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, we try to compute these parameters as early as possible by arranging more computing in the map function.

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);
```

As described in Section4.4, constructing the approximate histogram with specified error in HEDC involves one or two MapReduce jobs depending on the data layouts. The processings of these 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 Algorithm2. 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(1). For every tuple in the block, the bucket ID is specified as the output key(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(3-4).

In order to reduce the cost of intermediate data transmission in the shuffle phase, we define a combine function, which is shown in Algorithm3. The values belonging to the same bucket in the block are accumulated(1-4) and the proportion of every bucket in the block is computed(5). The first double of the output value is specified the proportion of the bucket in this block, and the second double is specified the square of the proportion(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 Algorithm4, the two doubles *sum* and *quadraricsum* of the same bucket in the value list are accumulated respectively(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 *line* $9-12$. Then the estimated histogram size and variance is computed(13-14).

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(11) 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 are computed through equation(12).

Algorithm 4: Reduce Function

```
   input : Text key, Iterator (Twodouble) values
   output: size estimate of bucket_i 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 σ_i = quadraticsum/n − sum*sum/n*n;
```

6. PERFORMANCE EVALUATION

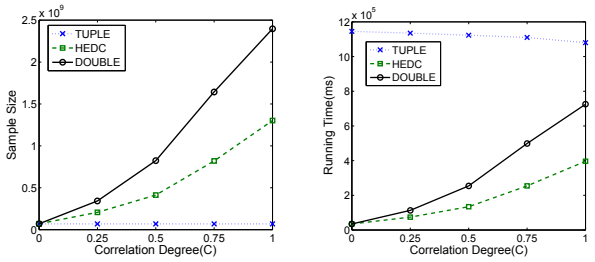
In this section, we evaluate the performance of HEDC in terms of sample size and the running time to get an estimate with specified error. We compare our adaptive sampling approach in HEDC against two other sampling methods by evaluating their performances on datasets with different data correlations 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 1Gbit 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 2.33G quad-core CPU and 7GB of RAM, and the disk size of every node is 1.8T. We set the block size of HDFS to 64M, and configure Hadoop to run 2 mappers and 1 reducer per node.

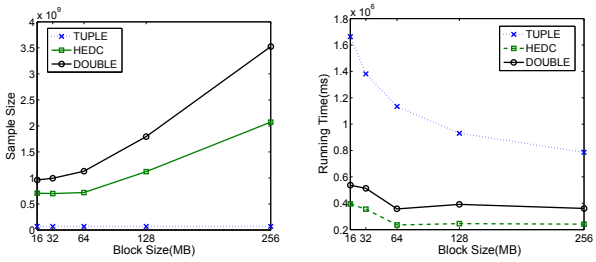
We adopt two metrics to evaluate the performance: *sample size* and *running time*. *Sample size* is computed by the histogram estimate algorithm, it represents the number of tuples needed to be sampled to get the histogram estimate with a specified error, and *running time* is the time cost to get the final estimate. We compare the performance of HEDC with two histogram estimate methods called *TUPLE* and *DOUBLE*, they 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 [10]. If the initial 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 7 months of hourly pageview statistics for all articles in Wikipedia, and includes 320GB of compressed data(1TB uncompressed)[5]. Every tuple in the dataset contains four columns: language, page name, page views and page



(a) Sampling Size (b) Running Time

Figure 2: The Effect of Data Correlation



(a) Sampling Size (b) Running Time

Figure 3: The Effect of Block Size

size. We choose *language* as the key column to construct histogram to reflect the data distribution on ten languages, so the number of buckets of the histogram is k=10. We set the confidence level of bounding the error at 95%, and set the specified error at 0.05.

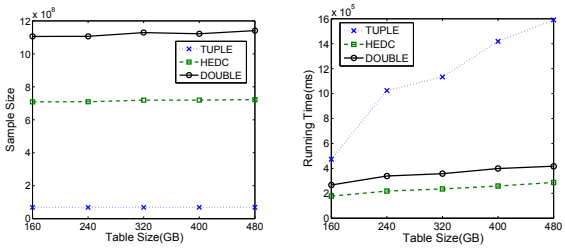
6.2 Effect of Data Correlation

In this section, 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 [8]. According to the analysis of equation11, $var\_sum=\sum_{i=1}^{k}\sigma_i^2$ reflects the data correlation in blocks. *var_sum* is maximized when the tuples are fully ordered by the language, 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 biggest, and C=0 when tuples are laid randomly.

Figure2 illustrates the sample size and running time of three approaches on different data correlations respectively. 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 experiment results. We can see that the data correlation doesn't 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 will 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 to that of processing all the data. Secondly, the number of mapper task in TUPLE is also the same to that of processing all the data, and the sum of map task's start time is not reduced. When the degree of correlation is equal to zero, the three approaches retrieve almost the same sample size, HEDC and DOUBLE pick a little more tuples because their sampling unit is block. As the degree of correlation increases, the sample size of DOUBLE is larger than that of HEDC. For C=0.5, DOUBLE requires sample size 2 timer 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 larger 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 64M. 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



(a) Sampling Size (b) Running Time

Figure 4: Scale-up with Data Size

sizes on the estimator's performance. We adopt the real Wikipedia hit log data directly in this experiment without any changing. For the real dataset, log records within the same hour are ordered by language, so the data correlation is between the random layout and the fully ordered layout. The results are shown in Figure3, we run the three approaches with five block sizes: 16M, 32M, 64M, 128M and 256M. The block size doesn't 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 start time sum of all the map tasks will be shorter. However, the processing time of every map task gets longer because of bigger block. 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, the data size of every hour is about 65M. So the correlation degrees of block size 16M and 32M are much bigger than other three block sizes, and it requires more sample blocks for HEDC and DOUBLE. When the block size is bigger than 32M, the data correlation decreases gradually, so the required sample blocks of HEDC and DOUBLE decreases 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 64M, the running time of HEDC and DOUBLE get 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. Figure4 shows the required sample size and running time of the estimators on a 10-node cluster with different data size. For HEDC and DOUBLE, the sample size doesn't increase in directly proportional to the data size. According to equation(11), the error bound isn't effected directly by the data size, it's associated with the variance sum of all the buckets. The

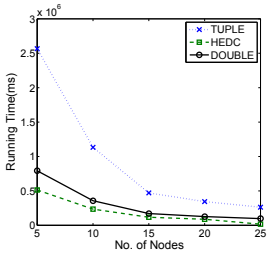


Figure 5: Scale-up with Cluster Scale

variance sum doesn't change a lot as the data size increases in our dataset, so the data size needed to be sampled doesn't 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. Given the dataset to be processed, the required sample size doesn't change as the cluster scale changes, so we only show the running time when varying the number of nodes in the cluster in Figure5. We can see that as the number of nodes increases, the running time of these three approaches decreases, 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.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a novel histogram estimator called HEDC, which focuses on estimating the histograms for data in the cloud. This problem is challenging to solve in the cloud for two reasons: i) 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; ii) the typical processing mode of the cloud is batch processing, which is adverse to the requirements of estimator to return "early results". We design the processing framework of HEDC on extended MapReduce, and propose efficient sampling mechanisms which include designing the sampling unit and determining the sampling size adaptive to the data layout. The experiment results of our techniques on Hadoop show that HEDC can provide efficient histogram estimate for data of various layouts in the cloud.

In the future work, we will research on histogram estimate in the cloud from two aspects. First, we focus on equi-width histogram's estimate in this paper, and will investigate the estimate of equi-depth histogram, max-diff histograms and other types of histograms that are also used widely in data summarizing. One of the problems to be solved of these histograms' estimate is to approximate the bucket range based on the sampling data. On the other hand, when tuples in the dataset are modified, then the precomputed approximate histograms are 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.

8. ACKNOWLEDGMENTS

This research was partially supported by the grants from the Natural Science Foundation of China (No. 91024032, 91124001, 61070055, 60833005), the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (No. 11XNL010, 10XNI018), National Science and Technology Major Project (No. 2010ZX01042-002-003), the US National Library of Medicine (No. R01LM009239).

9. REFERENCES

[1] HBase. Available at http://hadoop.apache.org/hbase/.

[2] Cassandra. Available at http://incubator.apache.org/cassandra/.

[3] HDFS. Avaiable at http://hadoop.apache.org/hdfs/.

[4] Hadoop. Available at http://hadoop.apache.org/.

[5] Wikipedia page traffic statistics. Avaiable at http://aws.amazon.com/datasets/2596.

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

[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD 2010 Conference Proceedings*, pages 975–986, June 2010.

[8] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD 2008 Conference Proceedings*, pages 287–298, June 2004.

[9] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Using random sampling for histogram construction. *Microsoft Research Report*, 1997.

[10] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998 Conference Proceedings*, pages 436–447, June 1998.

[11] W. G. Cochran. *Sampling Techniques*. John Wiley and Sons, New York, 1977.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI 2004 Conference Proceedings*, pages 137–150, Dec 2004.

[13] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems*, 27(3):261–298, 2002.

[14] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB 2003 Conference Proceedings*, pages 19–30, August 2003.

[15] J. Jestes, K. Yi, and F. Li. Building wavelet histograms on large data in mapreduce. In *VLDB 2011 Conference Proceedings*, pages 109–120, August 2011.

[16] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD 2011 Conference Proceedings*, pages 949–960, June 2011.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD 2008 Conference Proceedings*, pages 1099–1110, June 2008.

[18] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD 1984 Conference Proceedings*, pages 256–276, June 1984.

[19] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD 1996 Conference Proceedings*, pages 294–305, June 1996.

[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, and R. M. Pete Wyckoff. Hive: a warehousing solution over a map-reduce framework. In *VLDB 2009 Conference Proceedings*, pages 1626–1629, August 2009.