You Can Stop Early with COLA: Online Processing of Aggregate Queries 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

Cloud-based data management systems are emerging as scalable, fault-tolerant, and efficient solutions to manage large volumes of data with cost effective infrastructures, and more and more data analysis applications are migrated to the cloud. As an attractive solution to provide a quick sketch of massive data before a long wait of the final accurate query result, online processing of aggregate queries in the cloud is of paramount importance. This problem is challenging to solve because of the large block based data organization and distributed processing mode in the cloud. In this paper, we present COLA, a system for Cloud Online Aggregation to provide progressive approximate answers for both single tables and joined multiple tables. We develop an online query processing algorithm for MapReduce to support incremental and continuous computing of aggregations on joins which minimizes the waiting time before an acceptable estimate is achieved. We formulate a statistical foundation that supports block-level sampling for single-table online aggregations and effective estimation of approximate results and confidence intervals of statistical significance. We also develop a two-phase stratified sampling method to support multi-table aggregations to improve the approximate query answers and speed up the convergence of confidence intervals. We implement COLA in Hadoop, and our experiments demonstrate that COLA can deliver reasonable precise online estimates within a time period two orders of magnitude shorter than that used to produce exact answers.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

online aggregation, MapReduce, cloud computing

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

1. INTRODUCTION

In the past decade, massive data has been produced in various applications, including online transaction data, Web access logs, sensor data, scientific data, etc. Distilling the meaning from these massive data has never been in such urgent demand, for example, exploring users' behaviors and interests will provide decision support, and deep analysis of scientific data is commonly conducted to support scientific discoveries. These applications are difficult to support in traditional databases, due to the massive volumes of data, the complexity and diversities of queries. Meanwhile, cloud based data management systems provide highly efficient and cost effective solutions for massive data analytics, and applications are gradually migrated to the cloud environment.

Aggregations are among the most common query types for data analysis, which often scan enormous amount of tuples to generate summary and statistical results. Though query processing in the cloud is normally paralleled, due to the data volumes involved, an aggregation query often takes a long time to return the final result, and users may have to wait a long period of time before the whole query is executed to generate the precise answer. Since many large-scale aggregation queries are used to get a sketch or "big picture" of the data, one promising approach to address this problem is online aggregation (OLA) [14], which can continuously provide "early returns" with estimated confidence intervals. As more data is processed, the estimate is progressively refined and the confidence interval is narrowed until the precise result is produced. While OLA has been studied for RDBMS and streaming data management systems [10, 11, 16], OLA emerges as a new research area for the cloud and is of paramount importance due to the unique characteristics of the cloud. First, OLA makes it possible to save cost on the pay-as-you-go cloud cost mode. Reducing query processing time will lead to immediate cost saving and is highly desired by users. For example, obtaining a result with 95% confidence level in much less time could be very attractive. Secondly, queries in the cloud is always composed of many sub-tasks executing on different nodes, OLA could help to increase the parallelism degree and resource utilization by processing parallel subtasks in online mode. Thirdly, OLA could reduce or eliminate performance bottleneck caused by the slowest nodes. Since the cloud is typically a heterogeneous environment consisting of many nodes with diverse hardware setup and performance, the execution time of a query is significantly influenced by the sub-tasks in the slowest nodes. With an OLA system, the data flow between sub-tasks is pipelined and the estimate result is refined continuously, thus the performance "tail" effect could be alleviated or eliminated.

Though online aggregation techniques have been extensively studied for single-site relational database, there are many challenges to

adapt them to the cloud. First of all, the distributed environment of the cloud brings up the problems of processing concurrency, data distribution, data skew and other issues that must be accounted for during query processing and statistical computing. In particular, for aggregations over joined multiple tables in such environment, how to process queries in online mode to minimize the time before a confident estimate is obtained requires sophisticated considerations. In addition, data in the cloud is typically organized and processed in blocks, which could be thousands times larger than those in traditional file systems [9, 1]. In the statistics estimate procedure in OLA, uniform-random sampling is of theoretical significance. However, taking a uniform-random sample can be inefficient for distributed data in such organization. For example, if data is blocked on an attribute associated with the aggregation, in the worst case scenario, each sampling step can be no faster than a full scan of the data. To effectively build statistical estimators with blocked data, a major challenge is to guarantee the accuracy of the estimators while leveraging sampling as efficiently as possible. Last but not least, the typical batch processing mode of the cloud does not match the requirements of online aggregation processing, where "early returns" are generated before all the data is processed. For example, for MapReduce [8], the entire output of each map and reduce task is materialized to a local file before it can be consumed by the next stage. The operators cannot begin until their precursor operators finish.

Motivated by the requirements and challenges, we propose and develop a system COLA for Cloud Online Aggregation to bridge the gap of online aggregation and the cloud. Our contributions include:

1. We propose a system architecture for online processing of aggregate queries in the cloud. COLA retrieves random samples from distributed data, and provides progressive approximate answers for not only single table aggregation but also multi-table aggregation, which is largely unaddressed in previous work.
2. We develop an online query processing algorithm for MapReduce to support incremental and continuous computing of aggregations on joins, and minimize the waiting time before an acceptable estimate is achieved. The algorithm repartitions tuples to different reducers based on their join keys, and conducts ripple joins on the reducers.
3. We formulate a statistical framework for supporting online aggregation in the cloud. We design a block-level sampling method for single-table online aggregations to make best use of transmitted sampling data, and develop an estimation algorithm to provide approximate results and confidence interval of statistical significance.
4. We propose a two-phase stratified sampling method for multi-table aggregations. This method much improves the approximate query answers and speeds up the convergence of confidence intervals for join aggregations by defining the sampling strata based on data repartition.
5. We implement COLA in Hadoop, and our experiments demonstrate that COLA can deliver reasonable precise online estimates within a time period two orders of magnitude shorter than that used to produce exact answers.

The rest of this paper is organized as follows. In Section 2, we present the related work. In Section 3, we describe the system architecture and data flow of COLA. In Section 4, we analyze the sampling unit and the sampling mechanism for online aggregation on single tables, and describe the estimate and confidence interval algorithm over MapReduce. In Section 5, we describe the online processing algorithms of aggregate queries over multiple tables, including the two-phase stratified sampling method and the statistical computing algorithms on table joins. Experiment results are presented in Section 6, followed by conclusions.

2. RELATED WORK

In general, our work in this paper is related to two fields: online aggregation and data sampling. Online aggregation was first proposed in [14], which focuses on single-table queries involving "group by" aggregations. The work in [10] improves the approach in [14] by providing the large-sample and deterministic confidence interval computing methods in the case of single-table and multi-table queries. The query processing and estimate algorithms for OLA were studied in the context of joins over multi-tables [11, 16, 15]. A family of join algorithms called ripple joins were presented in [11]. Ripple joins are based on the traditional block nested-loops and hash joins, which are designed to minimize the time until an acceptably precise estimate result is available. The work in [16] extends the original ripple joins to speed up the convergence by parallelizing the query processing and sampling. However, it can not provide statistical guarantee when the exact data distribution is unknown or the memory overflows. The work in [15] tries to make query estimate and maintain probabilistic confidence bounds no matter whether the inputs fit in memory or not. The approach combines the traditional sort-merge join and ripple join algorithms, and adds a shrink phase to the query processing which runs concurrently with the merge phase to update the estimate result. The online aggregation is extended in [21] to the distributed environment maintained in a distributed hash table network. All the work above is in the context of traditional databases. Nowadays, online aggregation research is renewed in the context of cloud computing, and some studies have been conducted based on MapReduce [7, 19]. Hadoop Online Prototye(HOP) [7] pipelines the MapReduce processing of Hadoop, which allows posterior operators consume the output of precursor operators before the precursor operators complete. HOP can provide the original snapshots of the MapReduce jobs at data dependent intervals, and it supports OLA by scaling up the snapshots with the job progress without any confidence bounds of the query estimate. A Bayesian framework based approach is used to implement OLA over MapReduce [19]. The approach considers the correlation between aggregate value of each block and the processing time, takes into account the scheduling time and processing time of each block as observed data during the estimate processing. The approach focuses on single table aggregate query containing one MapReduce job, without considering aggregate queries over joined multiple tables, which contain several MapReduce jobs.

Data sampling is important for online processing of aggregate queries, and much work has been done in the DBMS field. There are two levels of sampling unit in the existing sampling techniques: row-level sampling [17, 18, 3] and page-level or block-level sampling [5]. Row-level sampling provides true uniform-randomness, which is the basis of many approximate algorithms. However, row-level sampling can be very expensive because data is always clustered by blocks or pages. The block-level sampling is more efficient, but is prone to errors when generating statistics. The work in [5] analyzes the impact of block-level sampling on statistic estimation for histogram and distinct-value estimate, and proposes the corresponding statistical estimators with block-level samplings. A bi-level sampling scheme is proposed in [12], which combines the row-level and page-level sampling methods. All the above work is in the field of single-site DBMS. In the field of distributed DBMS, the work in [20] compares the accuracy and efficiency of different sampling methods for query size estimation in the parallel DBMS,

by using stratified random sampling and simple random sampling with the unit of row-level and page-level. Stratified sampling is well used in the online aggregation of distributed environment [21, 16]. In the context of online aggregation in the cloud, existing work of OLA over MapReduce assumes random sampling [7, 19], and no special sampling techniques have been proposed.

3. OVERVIEW OF COLA

In COLA, we are addressing two major challenges: how to produce incremental and continuous aggregation computing to provide early returns in cloud computing architecture, and how to provide effective sampling and estimation of confidence of early returned results. We first present an overview of the architecture and workflow of COLA, as shown in Figure 1. Data in the cloud is organized into blocks and distributed over different nodes. COLA first retrieves samples continuously from the data nodes and then sends them to the online query processing executor. The query executor processes the samples through MapReduce jobs and reports the intermediate results to the estimator. The estimator computes the approximate aggregation result and confidence interval based on the intermediate results, and presents it to the users. As the intermediate results are updated progressively, the estimator continues to refine the approximate results. Once users are satisfied with the early result, they can stop the query early before its completion.

Implementing online aggregation in the cloud requires extensions to the traditional MapReduce framework. First, online aggregation needs to provide "early returns", so it requires that the intermediate results of all operators should be pipelined during the query processing. However, the traditional MapReduce framework is batch-oriented, which means that a consuming operator cannot start until its producing operators finish. HOP [7] is a modified version of the Hadoop MapReduce framework to allow data to be pipelined within a MapReduce job and between jobs. We implement COLA based on extending the pipelining techniques of HOP, and set different pipeline granularities for different tasks. Secondly, in order to make efficient estimates, the online queries should process sample data instead of sequential data stream from data nodes. We add samplers in two phases of the MapReduce framework. The first sampler is arranged before the first job's map functions in the job series, and it samples data directly from the source data. We also add samplers in the shuffle phase to allow reducers perform sampling on mappers' outputs, and this sampler can be switched on or off depending on the query type.

COLA supports online aggregation processing over both single tables and multiple table joins. To our best knowledge, COLA is the first work on studying online aggregation for joins over MapReduce. During the online aggregation on single table which involves only one MapReduce job, we only adopt the sampler before map functions. To make more accurate estimates under certain data transmission and I/O cost, we take data block as the random sampling unit, and conduct the statistical computing based on aggregation results of the blocks. We implement the online processing of join aggregations through combining the repartition join [4] and ripple join [11] in the MapReduce framework. We propose a two-phase stratified sampling mechanism, which involves both the samplers before the map function and in the shuffle phase. The implementation details of COLA are presented in the following sections.

4. ONLINE AGGREGATION ON A SINGLE TABLE

In this paper, we focus our discussions on SUM and COUNT queries, while other aggregations such as AVG, VARIANCE and STD_DEV can be implemented through some extensions. For a table R containing |R| tuples, we consider aggregate queries of the form:

SELECT op(exp(t_{ij})), col FROM R WHERE predicate
GROUP BY col;

In the query, *op* is the operation of SUM or COUNT, *exp* is an arithmetic expression of the attributes in R, *predicate* is an arbitrary predicate involving the attributes, and *col* is one or more columns in R. When *op* equals to COUNT, we assume that *exp* reduces to the SQL "*" identifier. t_{ij} represents the $j-th$ tuple in block i, a random variable is constructed: $X_{ij} = |R| * exp_p(t_{ij})$. Given the result group k, $exp_p(t_{ij})$ equals to $exp(t_{ij})$ if t_{ij} satisfies the predicate clause and belongs to group k, or equals to zero otherwise. If *op* is COUNT, then $exp_p(t_{ij})$ equals to 1 when t_{ij} satisfies the predicate clause, and the procedure is the same to SUM. We set X_{ij} in table R to be the population, and the mean value of the population is the final aggregation result. Thus estimating the query result is transformed to the problem of estimating the mean value of the population. In the MapReduce framework, the aggregation query on single table requires one job. We present in this section the sampling method and statistical computation of online aggregation for single tables over MapReduce.

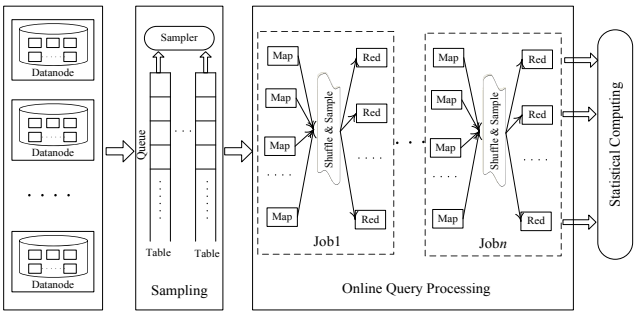


Figure 1: Data Flow of COLA

4.1 Sampling Methods

Sampling provides random access to the population data, and a majority of statistical estimates are based on the "uniform random sample" of tuples in the table. The uniform random sampling (or simple random sampling in statistical terms) means that the possibility of every tuple to be selected is equal. However, the tuple-level simple random sampling is not efficient in the cloud environment. Data in the cloud is organized into blocks, which is the unit of data transmission between nodes and disk I/O in the local node. In addition, a block is also the processing unit of MapReduce, which allocates one block to a mapper with consideration of locality. Retrieving a simple random sample of size b for one mapper may cause the transmission of b blocks in the network at worst, so it will be very expensive to conduct the tuple-level simple random sampling for online aggregation in the cloud. The alternative is to adopt the block as the sampling unit. However, in this case it is less clear how to guarantee the quality of the resulting estimate, since tuples within one block may have some correlation if data is clustered by the sampling attribute. In this section we analyze the variance-error of random sampling on both tuple level and block level, and clarify in Theorem 1 that the block-level sampling can produce more accurate estimate than tuple-level sampling with the same data transmission cost in the cloud.

THEOREM 1. *Set $\tilde{\mu}_{blk}$ and $\tilde{\mu}_{tpl}$ are used to represent the mean values of simple random sample data on block level and tuple level respectively. VAR($\tilde{\mu}_{blk}$) denotes the variance error of estimate ob-*

tained from block-level sampling of n block, and VAR($\tilde{\mu}_{tpl}$) denotes the estimate variance error from tuple-level sampling of n tuples. Then both $\tilde{\mu}_{blk}$ and $\tilde{\mu}_{tpl}$ are unbiased estimate of the aggregation query, and under the same data transmission cost,VAR($\tilde{\mu}_{blk}$) <= VAR($\tilde{\mu}_{tpl}$).

PROOF. Set S^2 to be the variance of X_{ij} in the population, and set S_b^2 to be the variance among the blocks. The simple random sampling on block level is called *cluster sampling* in statistical terms, with the cluster size equal to block size B. We define two types of mean values: the mean value per block: $\bar{X}=\sum x_i/N$, and the mean value per tuple: $\bar{\bar{X}}= \sum x_i/NB$, where N represents the number of blocks in the population, and x_i represents the sum of elements in block i. According to the characteristics of the cluster sampling, we can conclude that $\tilde{\mu}_{blk}$ is the unbiased estimate of the mean value of the population. The correlation of data in one block can be measured by the intra-cluster correlation coefficient ρ, which is computed from S and S_b: $\rho = \frac{S_b^2-S^2}{(B-1)S^2}$. The variance error of $\tilde{\mu}_{blk}$ can be computed through the correlation coefficient:

$$\begin{aligned} VAR(\tilde{\mu}_{blk}) &= \frac{N-n}{nNB}S^2[1+(B-1)\rho] \\ &= \frac{N-n}{nNB}S^2[1+(B-1)\frac{S_b^2-S^2}{(B-1)S^2}] = \frac{N-n}{nNB}S_b^2 \quad (1) \end{aligned}$$

The variance error of $\tilde{\mu}_{tpl}$ is:

$$VAR(\tilde{\mu}_{tpl}) = \frac{1-n/NB}{n}S^2 = \frac{(NB-n)S^2}{nNB} \quad (2)$$

Under the simple random sampling, the finite population corrections is negligible, thus we can compute the design effect based on equation (1) and (2):

$$\begin{aligned} deff &= \frac{VAR(\tilde{\mu}_{blk})}{VAR(\tilde{\mu}_{tpl})} = \frac{S_b^2}{BS^2} \\ &= \frac{\sum(x_i-\bar{X})^2/(N-1)B}{B\sum(x_{ij}-\bar{\bar{X}})^2/NB-1} = \frac{\sum(x_i-\bar{X})^2}{B\sum(x_{ij}-\bar{\bar{X}})^2} \quad (3) \end{aligned}$$

Since $(x_i-\bar{X}) = (x_{i1}-\bar{\bar{X}})+(x_{i2}-\bar{\bar{X}})+...+(x_{iB}-\bar{\bar{X}})$, we have:

$$\begin{aligned} \sum_{i=1}^{N}(x_i-\bar{X})^2 &= \sum_{i=1}^{N}\sum_{j=1}^{B}(x_{ij}-\bar{\bar{X}})^2 + 2\sum_{i=1}^{N}\sum_{j<k}^{B}(x_{ij}-\bar{\bar{X}})((x_{ik}-\bar{\bar{X}}) \\ &<= B\sum(x_{ij}-\bar{\bar{X}})^2 \quad (4) \end{aligned}$$

According to equation (3) and inequation (4), we can get the conclusion: $deff$<=1 □

COLA conducts the block-level random sampling in the first sampler mentioned in Section 3, it is implemented in the map task scheduling procedure of the first job. After the input file is split according to the block size, all the mappers corresponding to the file splits are added to the scheduling queue. Each time a map task is requested by the job scheduler, COLA retrieves a mapper from the scheduling queue through simple random sampling, and one block is sampled as the input data.

4.2 Estimation and Confidence Interval Computing

After the block-level sampling procedure, we can get n independent blocks with size B. Given a sampled block B_i, and set $exp_p(B_i) = \sum_{j=1}^{B} exp_p(t_{ij})$, the estimation of aggregation result is given by

$$\tilde{\mu}_{blk} = \frac{1}{nB}\sum_{i=1}^{n}\sum_{j=1}^{B}X_{ij} = \frac{1}{n}\sum_{i=1}^{n}N*exp_p(B_i) \quad (5)$$

According to the previous equation, the final aggregation query result can also be considered as the average of Y_i, where $Y_i = N*exp_p(B_i)$.The sampled blocks are retrieved in random order, so observations of Y_i are identically distributed and independent to each other. According to the analysis of [14] based on CLT(central limited theorem), the average of Y_i in samples approximately obeys the normal distribution. We can obtain the corresponding formulas to compute the half-width interval with specified confidence level $100p\%$:$\varepsilon_n = z_p\sigma_n/\sqrt{n}$, where z_p is the p-quantile in the standard normal distribution, and σ_n is the standard deviation of n variables in the sample. Then the final $100p\%$ confidence interval of the aggregation result is $[\tilde{\mu}_{blk}-\varepsilon_n, \tilde{\mu}_{blk}+\varepsilon_n]$.

4.3 Online Procedure over MapReduce

When implementing the aggregation online processing over MapReduce, we have following considerations. First, we try to arrange more computation work on the mapper functions. Generally speaking, there are more mappers in the MapReduce job for aggregation queries, so arranging more work on mappers will help increase the degree of parallelism. Secondly, we design the data flow to reduce the transmission cost in the shuffle phase. Thirdly, we make the computation of aggregation estimation and confidence interval of each iteration incremental to reduce the repetitive work.

The online aggregation processing on a single table involves one MapReduce job. In the map function shown in Algorithm 1, tuples of every block are filtered according to the predicate and transformed into key-value pairs. The key is the group column value of tuple t(2), and the value is equal to $exp_p(t)$ which is defined at the beginning of Section 4(3). In order to reduce the burden of reducers and the transmission cost in the shuffle phase, the combine function is executed after the map function as shown in Algorithm 2. The values belongs to the same group of this block are accumulated (1-4), and the output value is a structure containing two double data called "Twodouble". The first data is used to compute the variable's mean value, and the second data is used to compute the variance in the reduce function (5-6).

Algorithm 1: Map Function

```
   input  : tuple t
   output: Text Key, Twodouble value
1  if t satisfies the predicate then
2      key.set(t.col);
3      value.set(exp_p(t),0);
4  end
5  output.collect(key, value);
```

After the combine function, all the values belonging to the same group are transmitted to the same reducer. The reduce function is executed each time the estimate is invoked, so it is important to make the computing process incremental and make use of the results of the previous reduce function. The reduce function is described in Algorithm 3. After the accumulation of new values in the iteration (4-8), we compute the total sum and variance by adding the sum and quadratic sum computed in the last iteration (9-11). At last, the aggregation estimate and the half-width of confidence interval of each group is computed (12-13).

Algorithm 2: Combine Function

input : Text key, Iterator (Twodouble) values
output: (Text key, Twodouble value')

```
while values.hasNext() do
    Twodouble it = values.getNext();
    sum+ = it.get_first();
end
value'.set(sum, sum*sum);
output.collect(key, value');
```

Algorithm 3: Reduce Function

input : Text key, Iterator (Twodouble) values
output: aggregation estimate μ, confidence interval half-width ε

```
// n: number of tuples processed by the reducer
// sum_i: sum of the variables in the last iteration
// quadraticsum_i: quadratic sum of the variables in the last iteration
while values.hasNext() do
    Twodouble it =values.getNext();
    sum+ = it.get_first();
    quadraticsum+ = it.get_second();
end
sum = sum + sum_i;
quadraticsum = quadraticsum + quadraticsum_i;
variance = quadraticsum/n − sum*sum/n*n;
μ = sum/n;
ε = Zp*sqrt(variance)/sqrt(n);
```

5. ONLINE AGGREGATION OVER JOINED MULTI-TABLES

We propose the join algorithm that will support online processing for multi-table join aggregations in the form:

SELECT op(exp($t_{1i}, t_{2j}, ..., t_{km}$)) FROM R1,R2,...,Rk
WHERE predicate GROUP BY col;

In the query expression, op is the aggregate operator as COUNT or SUM. exp, $predicate$ and col represent the same meanings with those in the single table aggregation query, except that they involve the attributes of multi-relations. In this paper, we focus on the online processing of equal joins, which are most commonly used in real applications. Suppose we want to perform an equijoin on two relations R and S on attributes R.c and S.d, and conduct an aggregation on the join results. Next we discuss the procedures on processing join aggregation, the sampling mechanism and the estimation algorithm.

5.1 Online Processing of Join Aggregation

Different from aggregation query on a single table which involves only one MapReduce job, the aggregation on joins involves multiple MapReddue jobs. There are several processing methods of implementing offline join queries over MapReduce, such as repartition join, broadcast join, semi-join, etc [4]. While offline query processing aims to minimize the time to query completion, online query processing attempts to provide precise estimate quickly, in a smooth and continuous way. The broadcast join and semi-join implemented on MapReduce have to compute global distribution information of tables before the join processing, which is adverse to the incremental computing of online aggregation. In the context of online aggregation, ripple join is proposed to support cumulative update of the join result in the single-node environment [11]. We design the online processing of join aggregation by combining repartition join and ripple join.

In the context of offline processing, the repartition join on the previous example involves two MapReduce jobs. The main workflow of repartition join is depicted in the solidline rectangles of Figure 2. In $Job1$, mappers conduct the filtering on two tables and arrange the join key as the key of output results. In the shuffle phase, all the tuples whose join keys belonging to the same hash bucket are sent to the same reducer. At last, the reducers execute the local joins. Every output file of $Job1$ is sent to $Job2$ as the input file of mappers. The mappers set the group-by column as the output key to the shuffle phase, and reducers execute the aggregation function.

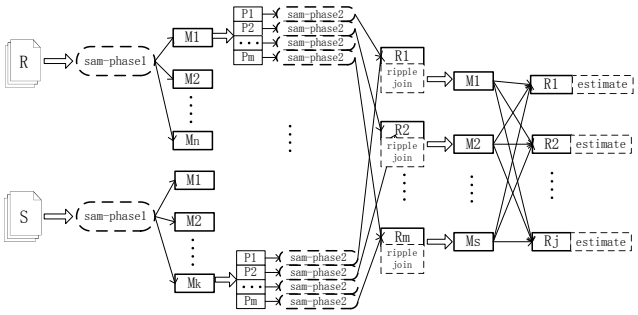


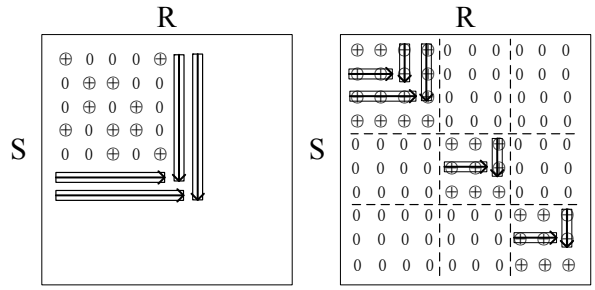
Figure 2: The Online Procedure of Join Aggregation

In order to support online aggregation, we extend the traditional repartition join framework, which is depicted in the dotted rectangles of Figure 2. First, we add samplers before some of the operators, which will provide random sampling data as the input. In addition, COLA allows the dataflow pipelined during the query processing, so in the reduce function of $Job1$, data emitted continuously from all the mappers are ripple joined. Thirdly, we add an estimate module in the reduce function of $Job2$ to provide the estimate result and confidence interval of each group. The sampling mechanisms and the estimate algorithm are described in the following sections.

5.2 Sampling Issues

5.2.1 Why Stratified Sampling?

Ripple join is proposed to support join online aggregation in the single-node environment [11], and it is the mostly used method in online aggregation of joins until recently. Let $exp_p(r_i, s_j)$ equal to $exp(r_i, s_j)$ if (r_i, s_j) satisfies the joint predicate condition, else set it to zero. We use $\oplus$ to represent $exp(r_i, s_j)$, and the ripple join result can be signified by a matrix shown in Figure 3(a). In each sampling step, the ripple join retrieves random samples from both R and S, and joins them with the previously-seen tuples and with each other. Then it computes the estimate result and confidence interval based on these join results by taking all the elements in the two-dimensional matrix as the population. However, this sampling method will not be efficient in the context of MapReduce repartition join for the following reasons. First, the repartition join processing is distributed to different nodes, and conducting a centralized sampling may introduce load imbalance for these nodes, which will result in inefficiency for both the query processing and estimation. Secondly, the ripple join sampling considers the whole space of two tables' cartesian product as the statistical population. When the number of tuples satisfying the join predicate is small or there are many groups in the output, the estimation results will be less accurate and the interval convergence will be slow. In particular, if the join key is the key or candidate key of one table, then there can only be one tuple satisfying the join predicate and contributing to the aggregation result in one column or one row of the matrix. Last, in the MapReduce repartition join, tuples that owing the same



(a) The original ripple join sampling (b) The stratified sampling

Figure 3: Sampling Procedure of Join

join key are partitioned to one reducer. However, the original ripple join does not efficiently utilize this data distribution.

Based on the previous analysis, we distribute the sampling procedure into different nodes, which has the similar spirit to the *stratified sampling* in the statistical term [6]. Set R_h and S_h represent the tuples sent from the two tables to the hth reducer of $Job1$ respectively, and the join keys of R_h and S_h belong to the same range. We can get $R \bowtie S = \bigcup R_h \bowtie S_h$. We define the tuples of each table sent to each reducer of $Job1$ as the *sampling stratum*. Given a reducerID or a stratumID h, stratified sampling data from different tables is ripple joined, and we define these joined results as *part h*. Under this sampling mechanism, we consider the union of all the joined results of each part as the population X. As shown in Figure 3(b), all the non-zero results are clustered together based on the join keys of the corresponding strata. The stratified sampling efficiently utilizes the data repartition in MapReduce join, and it can be easily distributed over multiple reducers. Also it improves the estimate accuracy and accelerates the convergence speed by excluding much of the zeroes in the cartesian product.

5.2.2 Two-Phase Stratified Sampling

Stratified sampling provides a distributed sampling mechanism for join online aggregation over MapReduce. Suppose there are m reducers in $Job1$, the population size is $N = \sum_{h=1}^{m} |R_h * S_h|$. We set the random variable X_{ij} as $X_{ij} = |N| * exp_p(r_i, s_j)$, where $(r_i, s_j) \in \bigcup(R_h * S_h)$, then the join aggregation result is the mean value of all the random variables in the population. However, applying stratified sampling directly on the repartition join confronts a big challenge: the stratum to which a tuple belongs is unknown until all the data has been processed by the shuffle phase of $Job1$. As a result, we cannot get the stratum distribution or stratum size before the completion of the join aggregation query, which is the base of conducting stratified sampling and computing the estimation.

Our general approach is to conduct a *two-phase sampling* mechanism, also called *double sampling* [6]. As shown in Figure 2, the first phase of sampling is arranged before map functions of $Job1$, and the first sampler mentioned in Section 4 is executed from R and S respectively. These sampled tuples are processed by the mappers and then repartitioned to reducers depending on the join keys. After the map functions we can get the first samples from R and S with size n'_r and n'_s respectively, which are used to estimate the stratum distribution of the population.

The second phase of sampling is conducted based on the first sample before the reduce function in $Job1$, and it produces distributed stratified random samples of size n_r and n_s from R and S respectively. After the map function, each mapper sorts the output key-value pairs and partitions them into different parts consumed by the following reducers. Algorithm4 illustrates the second-phase sampling from one table executed by reducer h before the reduce function. Each reducer executes random sampling from all the mappers' output results, and the size of sample from each mapper is proportional to the cardinality of its output results partitioned to this reducer (5-6). The samples from each mapper's output are collected by reducer h (7) and they produce a random sample of a stratum in the output of first-phase sampling from one table.

Algorithm 4: Second-Phase Sampling

```
   input  : int n_hr, int m
   output: sample set S_hr
1  // n_hr: number of required samples of reducer h from table r
2  // m: number of mapper's output files
3  i=1;
4  for i <= m do
5        w_ir = n'_hri / n'_hr ;
6        s_ir = w_ir * n_hr;
7        S_hr=S_hr ⋃ {s_ir random samples from the mth mapper's output file};
8  end
9  return S_hr;
```

THEOREM 2. *In each step of the second phase sampling, after all the reducers of Job1 conduct Algorithm4 from one table, the collected samples produce a stratified random sample on the first samples of this table.*

PROOF. After the first-phase sampling and the mapper functions of $Job1$, the output results of each mapper are partitioned into m parts, which correspond to the m reducers, and the samples from the first phase are divided into m disjoint strata. In the output results of mappers, the size of data that should be consumed by reducer h is n'_{hr}, and n'_{hri} represents the size of data consumed by reducer h from table r in the output of mapper i. Under the algorithm Second-Phase Sampling, reducer h takes random sampling from each mapper's output, and each tuple of partition h in mapper i is retrieved by the probability $1/n'_{hri}$. Reducer h picks mapper i with probability n'_{hri}/n'_{hr}. So given the stratum h of table r, every tuple has the same probability $\frac{1}{n'_{hri}} * \frac{n'_{hri}}{n'_{hr}} = \frac{1}{n'_{hr}}$ to be picked. □

5.3 Estimate and Confidence Computation

5.3.1 Estimate of Join Aggregation Results

After the two-phase sampling, we get $n_{hr} * n_{hs}$ samples from part h of population X. Estimating the join aggregation result is equal to estimating the mean value $\bar{Y}$ of all the variables in population X. Let $\bar{Y_h}$ to be the mean value of part h in population X, $\bar{y'_h}$ to be the mean value of part h in the first samples, and $\bar{y_h}$ to be the mean value of part h in the second samples. The weight of each part ω_h is estimated without bias by: $\omega'_h = n'_{hr} * n'_{hs} / \sum_{h=1}^{m} n'_{hr} * n'_{hs}$. We use the following formulas to compute the estimate of $\bar{Y}$:

$$\bar{y} = \sum_{h=1}^{m} \omega'_h * \bar{y_h} \tag{6}$$

THEOREM 3. *The estimated mean $\bar{y} = \sum_{h=1}^{m} \omega'_h * \bar{y_h}$ is an unbiased estimate of the join aggregation result $\bar{Y}$, that is $E(\bar{y}) = \bar{Y}$.*

PROOF. According to Theorem 2, the second sampling is a stratified random sampling from the first sample. As analyzed in [11],

the estimated mean of part h in the second sampling $\bar{y_h}$ is the unbiased estimate for the samples of part h in the first phase $\bar{y_h'}$, that is $E(\bar{y_h}) = \bar{y_h'}$. We can compute the expectation of $\bar{y}$ as follows:

$$E(\bar{y}) = E_1(E_2(\bar{y})) = E_1(E_2(\sum_{h=1}^{m} \omega_h' \bar{y_h})) = E_1(\sum_{h=1}^{m} \omega_h \bar{y_h'}) = \bar{Y}$$

□

5.3.2 Confidence Interval Computation

Under the two-phase stratified sampling, given stratum h, tuples from R_h and S_h are randomly retrieved. We assume that the sampling is performed with replacement, and the error of this assumption is negligible because we only sample a small fraction from the big table data. Then the samples from R_h and S_h can be viewed as independent and identically distributed observations. According to the analysis of [13], the conclusion of [11] can be extended to the repartition join situation to show that, as the sample size becomes large, the estimation $\bar{y}$ approximately obeys normal distribution with mean value $\bar{Y}$ and variance $\sigma^2 = var(\bar{y})$. The confidence interval is $[\bar{y}-\varepsilon, \bar{y}+\varepsilon]$, and ε is computed by: $\varepsilon = z_p \sigma_n / \sqrt{n}$, where σ_n^2 is the consistent estimator of σ^2 over the samples. We have computed the estimated mean value in the previous section, then the left work is to compute the variance.

Set $R_h(n_{hr})$ and $S_h(n_{hs})$ represent the tuples sampled from R and S in stratum h of the second sample, where n_{hr} and n_{hs} represent the sample size from R_h and S_h respectively. Note that the samples $R_h(n_{hr})$ and $S_h(n_{hs})$ of each stratum are mutually independent, so the estimated mean value $\bar{y_h}$ is independent of $\bar{y_l}$ for $h \neq l$. Let s_h^2 represent the variance of part h in the second sample, then the consistent estimated variance of $\bar{y}$ can be computed through the following formulas:

$$\sigma_n^2 = \sum_{h=1}^{m} \omega_h'^2 * s_h^2 \tag{7}$$

Next we discuss how to compute the variance s_h^2. In fact, variables in part h are not statistical independent, because they are the cross-products of tuples in $R_h(n_{hr})$ and $S_h(n_{hs})$. According to the argument [11], the variance of samples in part h is given by $s_h^2 = \sigma^2(R_h(n_{hr}))/n_{hr} + \sigma^2(S_h(n_{hs}))/n_{hs}$. Denote by $\mu(r, R_h(n_{hr}))$ the average of $exp_p(r,s)$ over all $s \in S_h(n_{hs})$, then $\sigma^2(R_h(n_{hr}))$ is the variance of $\mu(r, R_h(n_{hr}))$ over all $r \in R_h(n_{hr})$. $\sigma^2(S_h(n_{hs}))$ has the similar definition as $\sigma^2(R_h(n_{hr}))$.

5.4 Implementation over MapReduce

As mentioned in Section 5.1, online join aggregation of two tables involves two MapReduce jobs: the first job conducts the join, and the second job conducts the aggregation. After the shuffle phase of *Job*1, all the tuples of the same join key are partitioned to one reducer. The reduce function processes samples of one stratum from all the tables. It not only implements the ripple join, but also computes the mean value and variance of the corresponding part, as shown in Algorithm 5. The input value of the reduce function is a structure called "Threedouble": *tableMark* identifies which table the value is retrieved from, *exp_col* represents the column value associated with the arithmetic expression *exp*, and *group_col* is the group-by column value. We adopt a two-dimensional array *outRes* to store three statistical elements for each group in the stratum: the sum of $exp_p(r,s)$, the quadratic sum of $sum(r,R_h)$ and $sum(s,S_h)$. $sum(r,R_h)$ represents the sum of $exp_p(r,s)$ over all $s \in S_h$ and $sum(s,S_h)$ has the similar definition. The tuple values from table R and S in one value list own the same join key.

Algorithm 5: Reduce Function of Job1

```
   input  : Text key, Iterator<Threedouble> values
            //Threedouble includes: tableMark, exp_col, group_col
   output: (Text group_key,Threedouble)
            //Threedouble includes: stratumID, meanValue, variance
 1 // G: group number
 2 Double[G][3] outRes;
 3 //each row of outRes includes: sum,quadSum_R,quadSum_S
 4 while values.hasNext() do
 5     create buffer B_R and B_S for values from R and S respectively ;
 6     Int[G][|B_S|] sums; //sum of exp(r,s) over all r ∈ B_R
 7     while B_R.hasNext() do
 8         Int[G] sumr; // sum of exp(r,s) over all s ∈ B_S
 9         r= B_R.getNext();
10         while B_S.hasNext() do
11             s=B_S.getNext();
12             group = (r.getGroup()==null) ? s.getGroup() :
               r.getGroup();
13             t=exp(r,s);
14             sumr[group]+=t;
15             sums[group][s]+=t;
16         end
17         for (i=1;i<=G;i++) do
18             outRes[i][1]+=sumr[i];
19             outRes[i][2]+=sumr[i]*sumr[i];
20         end
21     end
22     for (i=1;i<=G;i++) do
23         for (j=1;j<=|B_S|;j++) do
24             outRes[i][3]+= sums[i][j]* sums[i][j];
25         end
26     end
27 end
28 for (i=1;i<=G;i++) do
29     var_R=E(outRes[i][2]/n_hs^2) - E^2(outRes[i][1]/n_hs);
30     var_S=E(outRes[i][3]/n_hr^2) - E^2(outRes[i][1]/n_hr);
31     S_h^2 = var_R/n_hr + var_S/n_hs;
32     meanValue = outRes[i][1] / (n_hr * n_hs);
33     h = partID;
34     outvalue.add(Threedouble(h,meanValue,S_h^2));
35     output.collect(i, outvalue);
36 end
```

Two buffers B_R and B_S are created based on *tableMark*, and they contain the values from table R and S respectively. The join is implemented by computing cartesian product of B_R and B_S through a nest loop (7-21). Each time tuple r and s are joined, the group which $exp_p(r,s)$ belongs to is retrieved (12), and the $sum(r,R_h)$ and $sum(s,S_h)$ of this group is increased by $exp_p(r,s)$(14-15). Once the accumulation of $sum(r,B_R)$ is complete, the first values and second values in outRes of all the groups are updated (17-20). The third values of outRes[G][3] are updated when the nest loop ends (22-26). The part variance is computed based on formulas given in the previous subsection (29-31), and also the mean value of the part is computed (32). The output key of reduce function is set to the group key, and the mean value and variance of the part are encapsulated into the output value with partID (33-35).

The input key for mappers in *Job*2 is the group key, and the input value is a Threedouble structure, which includes the partID h, part mean value $\bar{y_h}$ and part variance S_h^2 of each group. Map function in *Job*2 does not process the records, and after the shuffle phase, all the statistics belonging to the same group are sent to one reducer. The result estimation and the interval computation are implemented in the reduce function of *Job*2, as shown in Algorithm 6. The aggregation result is estimated through equation (6) in the previous

subsection (5). The consistent estimated variance of aggregation result is computed according to equation (7) of the previous subsection (6). Then the confidence interval half-width is computed based on the estimated variance (8).

Algorithm 6: Reduce Function of Job2

input : Text key, Iterator<Threedouble> values
output: aggregation estimate $\bar{y}$, confidence interval half-width ε

```
1 while values.hasNext() do
2     it=values.getNext();
3     ȳ_h = it.getSecond();
4     S_h = it.getThird();
5     ȳ+ = ω'_h * ȳ_h;
6     VAR(ȳ)=VAR(ȳ) + ω'^2_h * S^2_h;
7 end
8 ε = Zp * sqrt(VAR(ȳ))/sqrt(n);
```

6. PERFORMANCE EVALUATION

We evaluate the performance of COLA in terms of estimate accuracy, convergence speed and scalability. We also study the influence of data correlation and sampling unit on the performance. All the experiments are performed on Hadoop 0.19.2. We first conduct experiments over a real world dataset to evaluate COLA's estimate accuracy and interval convergence speed, then we run experiments to evaluate the influences of data correlation and sampling unit on the performance. Last we evaluate the scalability of COLA.

6.1 Experiment Overview

The testbed is established on 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 slave nodes. Each node has a 2.33G quad-core CPU and 7GB of RAM, and the disk size of each node is 1.8TB. We set the block size of HDFS to 64MB, and configure Hadoop to run 2 mappers and 1 reducer per node.

In the experiment, we analyze the page traffic statistics of Wikipedia hits log. The dataset we use contains 7 months of hourly pageview statistics for all articles in Wikipedia, with 320GB of compressed data (1TB uncompressed) [2]. Based on the original traffic data of Wikipedia, we construct two tables: *visit_log* and *page_size*. Table *visit_log* contains hits log of Wikipedia pages with three columns: pagename, language and pageviews, while table *page_size* contains the page size information with three columns: pagename, language and pagesize. Data sizes of table *visit_log* and *page_size* are 300GB and 30GB respectively, and all the data files are stored in HDFS. We test aggregation queries on a single table and multi-tables with example queries shown next.

Q1=
```
SELECT SUM(pageviews),language FROM visit_log
GROUP BY language
```

Q2=
```
SELECT SUM(pageviews), visit_log.language FROM
visit_log, page_size WHERE visit_log.pagename=pa-
ge_size.pagename AND visit_log.language=page_size
.language AND page_size.pagesize>=5000 GROUP BY
visit_log.language
```

During the online processing of the above queries, COLA returns estimate result with confidence interval of every group continuously. In the experiment, we set the confidence level to 95%, and update the confidence interval at every 2% increment of the query progress.

6.2 Performance over Real Data

In this experiment, we run the two queries on the original wikipedia dataset. The accuracy of estimated aggregation result is measured by *relative_error*, which is computed post-hoc by: $relative\_error = \frac{|estimateValue - actualValue|}{actualValue}$. The convergence speed of confidence interval is reflected through *avgResTime*, which is the average time elapsed to get the estimate result with *relative_error* less than 1% and *relative_interval* less than 5%. The *relative_ interval* is defined as: $\frac{half-width\ of\ confidence\ interval}{estimate\ result}$.

Figure 4 and Figure 5 illustrate the update of *relative_error* and *relative_ interval* as the online aggregation of Q1 proceeds. There are ten kinds of languages analyzed in the experiment, and here we show the results of English and Polish. English is the most frequently visited language, while Polish belongs to the languages that are less frequently visited. We also show the estimate aggregation results of English without sampling on the source data. As Figure 5 shows, the confidence intervals become narrow as the online aggregation continues. Figure 4 shows that the relative errors of both English and Polish become less than 1.5% when the second estimate result is obtained at the query progress of 4%. However, the relative error of English without sampling on the source data is huge, and the average relative error of all the estimate results reaches 4.9%. So the estimate result and confidence interval computed without sampling is of no statistical significance. The relative interval of English reaches 4.4% when the query progress is at 6%, while the relative interval of Polish becomes less than 5% when its progress arrives at 12%, which is 2.8 times longer than that of English. Among the source data, log records of English and Polish account for 41% and 3.9% respectively, so the selectivity of English is much higher than that of Polish. The experiments demonstrate that COLA provides reasonable estimate within acceptable time period even for data with low selectivity.

The performance of COLA on join aggregations is illustrated in Figure 6 and Figure 7. We show the results of English and Polish with the two-phase stratified sampling, and also the results of English under the un-stratified sampling. Under the un-stratified sampling, the *relative_error* keeps at least 1% larger than that of the two-phase stratified sampling before the progress reaches 75%. This is because two-phase stratified sampling utilizes the data distribution after the repartition and excludes many zeroes from the population. The *relative_ interval* under two-phase stratified sampling also shrinks faster - it becomes less than 5% at the progress of 8% for English. The progress has to reach 24% for *relative_ interval* of English to be less than 5% under the unstratified sampling.

Table1 shows the avgResTime of COLA for English and Polish, and it also shows the time consumed to complete the query to produce a precise result of COLA and HOP. Q1 runs for 1532.9s before completing the query on HOP without online aggregation, and it takes 216.5s to obtain a reasonably precise result of Polish with low selectivity, the longest time among all the language groups. Returning the precise result of Q2 also saves a significant amount of time compared to completing it without online aggregation. We can also see that the time spent to complete queries on COLA is about 1% longer than the time spent on HOP without online aggregation, which demonstrates COLA has minimal overhead.

6.3 Effect of Data Correlation

During the online aggregation of single tables, a data block is adopted as the sampling unit and the statistical computing unit. In this experiment, we evaluate the effect of data correlation on the estimate of single table aggregation with COLA. For the real dataset, log records within the same hour are ordered by language. We

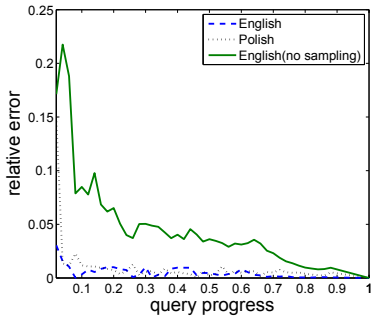


Figure 4: Relative Error(Q1)

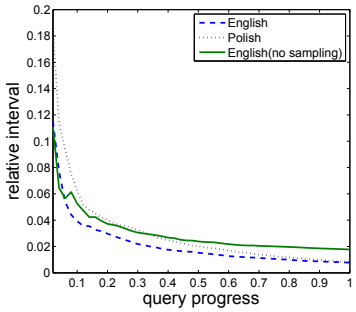


Figure 5: Relative Interval Width(Q1)

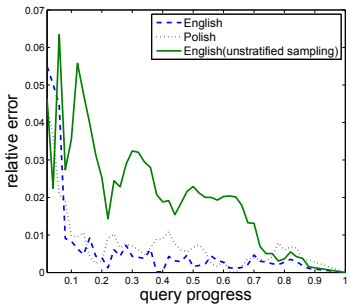


Figure 6: Relative Error(Q2)

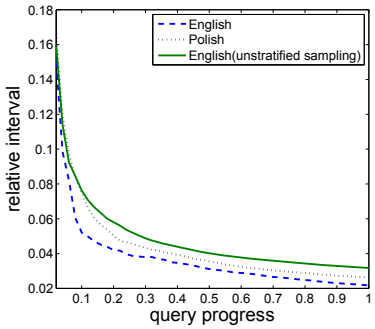


Figure 7: Relative Interval Width(Q2)

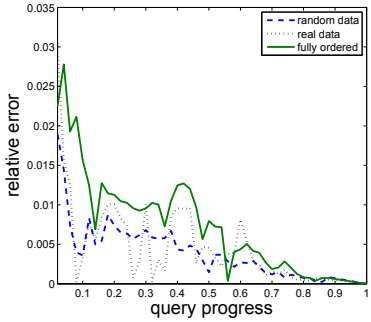


Figure 8: Correlation Effect on Error

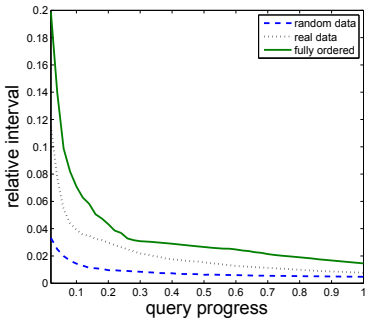


Figure 9: Correlation Effect on Interval

Table 1: Overhead of COLA

time(s)	English avgResTime	Polish avgResTime	COLA Complete	HOP Complete
Q1	131.9	216.5	1546.3	1532.9
Q2	219.9	279.3	2783.5	2761.4

change the data layout of real dataset to generate two extreme degrees of correlation: records in the whole data file are laid randomly or are ordered by language. The real data layout falls between the two. Figure 8 and 9 illustrate COLA's performance on estimating the results of English over different data layouts. At 10% of the query process, COLA provides estimate with relative error less than 1.5% for the fully ordered data. And for real data and random data, the relative error becomes less than 1% after the query progress reaches 8%. However, the relative error on real data fluctuates more widely than that on random data. This is a consequence of the fact that the variance of block aggregation on real data is bigger than the random data. The block aggregation on fully ordered data – the worst case of data layout – has the biggest variance, and the relative confidence interval becomes less than 5% after the query has processed 14%.

6.4 Effect of Block Sizes

In this experiment, we test the performance of COLA with different data block sizes. A block is the sampling unit of the sampler before map function, and it is also the processing unit in the MapReduce framework. We run Q1 and Q2 on COLA with four block sizes: 32MB, 64MB ,128MB and 256MB respectively, and compare the avgResTimes of the English group. As Figure 10 shows, queries running on the data with block size 64MB takes the shortest time to get the reasonable precise estimate. Since one block is processed by a single mapper task, a bigger block size will result in less mapper tasks of a MapReduce job. Also since every map task involves startup time, for large input data files, less number of map tasks will cost less startup time and help to reduce the total execution time of jobs. On the other hand, samples retrieved from bigger data blocks will introduce larger variances, and will result in slow interval convergence. Thus their is a tradeoff between the query execution speed and the convergence speed of different block sizes.

6.5 Scalability Evaluation

We evaluate the scalability of COLA by varying data sizes and node numbers, and the metric is also avgResTime of Q1 and Q2. Here node number represents the the number of slaves of the cluster. Figure 11 illustrates the performance of COLA running on the cluster with 10 nodes with different data sizes of table *visit_log*: 75GB, 150GB, 225GB and 300GB. From the results we can see that the avgResTime of 150GB is slightly shorter than that of 75GB, and after that the avgResTime increases with low growth rate. We run Q1 and Q2 on 5-node, 10-node, 15-node and 20-node configurations respectively, and the results are shown in Figure 12. As the number of nodes increases, the parallelism of tasks gets higher and more records are sampled at each step, thus the avgResTimes of both Q1 and Q2 decrease. This demonstrates the scalability of COLA for large datasets.

7. CONCLUSIONS

Online aggregation in the cloud makes it possible to save cost by taking acceptable approximate early answers. There are critical requirements to support online aggregation in the cloud, including the statistical methods to support sampling of data and confidence estimation of approximate results, and the query processing framework to support incremental and continuous computing of aggregations

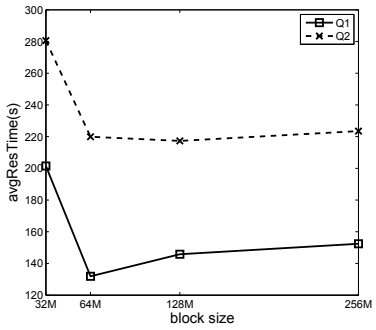


Figure 10: Effect of Block Sizes

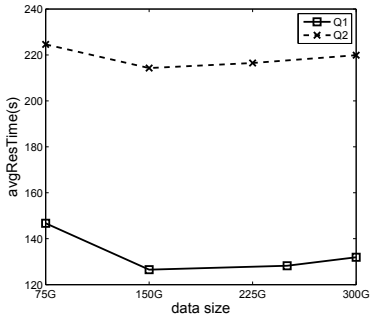


Figure 11: Scale-up with Data Size

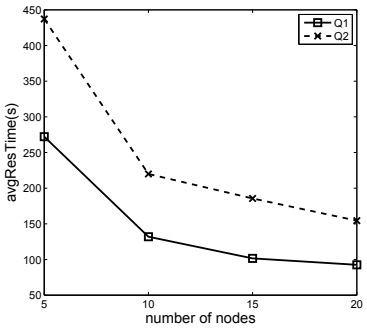


Figure 12: Scale-up with Node Number

on single tables or multiple tables. There are major challenges due to the mismatch of the data storage and computing model of the cloud and the unique requirements of online aggregation. COLA provides a promising framework to bridge the gap to support online processing of aggregate queries in the cloud. Major contributions of COLA include: a system architecture to support online aggregation by extending the HOP framework, statistical methods to support effective sampling and estimation of approximate results, online query processing algorithms to support aggregates on both single tables and joined multiple tables, and highly efficient performance based on the framework. To our best knowledge, COLA is the first work on studying online aggregation for joins in the cloud. Our experiments demonstrate that COLA can produce acceptable approximate answers within a time period two orders of magnitude shorter compared to those to produce exact results. We believe our methods are fundamental and can be extended to support other types of online analytical queries.

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

[2] Wikipedia Page Traffic Statistics. Avaiable at http://aws.amazon.com/datasets/2596.

[3] G. Antoshenkov. Random sampling from pseudo-ranked b+ trees. In *VLDB 1992 Conference Proceedings*, pages 375–382, August 1992.

[4] 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.

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

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

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD 2010 Conference Proceedings*, pages 1115–1118, June 2010.

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

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP 2003 Conference Proceedings*, pages 29–43, October 2003.

[10] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM 1997 Conference Proceedings*, pages 51–63, August 1997.

[11] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999 Conference Proceedings*, pages 287–298, June 1999.

[12] P. J. Haas and C. Koenig. A bi-level bernoulli scheme for database sampling. In *SIGMOD 2004 Conference Proceedings*, pages 275–286, June 2004.

[13] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *JCSS*, 52(3):550–569, February 1996.

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD 1997 Conference Proceedings*, pages 171–182, May 1997.

[15] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *SIGMOD 2005 Conference Proceedings*, pages 563–574, June 2005.

[16] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *SIGMOD 2002 Conference Proceedings*, pages 252–262, June 2002.

[17] F. Olken and D. Rotem. Random sampling from b+ trees. In *VLDB 1989 Conference Proceedings*, pages 269–277, August 1989.

[18] F. Olken and D. Rotem. Random sampling from database files: A survey. In *SSDBM 1990 Conference Proceedings*, pages 92–111, April 1990.

[19] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *VLDB 2011 Conference Proceedings*, pages 1135–1145, August 2011.

[20] S. Seshadri and J. F. Naughton. Sampling issues in parallel database systems. In *EDBT 1992 Conference Proceedings*, pages 328–343, March 1992.

[21] S. Wu, S. Jiang, B. C. Ooi, and K. L. Tan. Distributed online aggregation. In *VLDB 2009 Conference Proceedings*, pages 443–454, August 2009.