Wang YX, Luo JZ, Song AB *et al.* Partition-based online aggregation with shared sampling in the cloud. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(6): 989–1011 Nov. 2013. DOI 10.1007/s11390-013-1393-6

Partition-Based Online Aggregation with Shared Sampling in the Cloud

Yu-Xiang Wang (王宇翔), Jun-Zhou Luo (罗军舟), *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*
Ai-Bo Song* (宋爱波), *Member*, *CCF*, *ACM*, and Fang Dong (东　方), *Member*, *CCF*, *ACM*

School of Computer Science and Engineering, Southeast University, Nanjing 211189, China

E-mail: {lsswyx, jluo, absong, fdong}@seu.edu.cn

Received December 1, 2012; revised May 7, 2013.

Abstract Online aggregation is an attractive sampling-based technology to response aggregation queries by an estimate to the final result, with the confidence interval becoming tighter over time. It has been built into a MapReduce-based cloud system for big data analytics, which allows users to monitor the query progress, and save money by killing the computation early once sufficient accuracy has been obtained. However, there are several limitations that restrict the performance of online aggregation generated from the gap between the current mechanism of MapReduce paradigm and the requirements of online aggregation, such as: 1) the low sampling efficiency due to the lack of consideration of skewed data distribution for online aggregation in MapReduce, and 2) the large redundant I/O cost of online aggregation caused by the independent job execution mechanism of MapReduce. In this paper, we present OLACloud, a MapReduce-based cloud system to well support online aggregation for different data distributions and large-scale concurrent query processing. We propose a content-aware repartition method with a fair-allocation block placement strategy to increase the sampling efficiency and guarantee the storage and computation load balancing simultaneously. We also develop a shared sampling method to share the sampling opportunities among multiple queries to reduce redundant I/O cost. We also implement OLACloud in Hadoop, and conduct an extensive experimental study on the TPC-H benchmark for skewed data distribution. Our results demonstrate the efficiency and effectiveness of OLACloud.

Keywords cloud, MapReduce, partition, online aggregation, shared sampling

1 Introduction

Big data and big data analytics play an important role in today's fast-paced data-driven businesses[1]. The common characteristic of real-life applications is that they often have to deal with a tremendous amount of data to derive useful information. Performing analytics and delivering exact query results on such large volumes of data can be computationally expensive (long time for processing)[2] and resource intensive[3]. In general, overloaded systems and high delays are incompatible with a good user experience, and the early approximate answers that are accurate enough are often of much greater value to users than tardy exact results[4]. One commonly-used technique to handle this problem is online aggregation (OLA)[5], which aims to give response to large-scale aggregation queries with a statistically valid estimate to the final result early. The basic idea behind OLA is to compute an approximate result against the random samples and refine the result as more samples are received. In this way, users can terminate the running queries prematurely if an acceptable answer can be arrived at quickly.

With the development of cloud computing, processing OLA in some MapReduce-oriented[6] cloud systems such as Hadoop①, Hyracks[7], etc. has become indispensable due to the massive volumns of data involved make the original cloud framework often take a long time to return the final result, such as some preliminary work in [8-11]. Given the massive scale of data, the MapReduce framework needs to horizontally partition the original data into equal-size blocks and assign them to data nodes in a random manner for storage. When OLA is deployed for query processing, each query will

Regular Paper

This work is supported by the National Basic Research 973 Program of China under Grant No. 2010CB328104, the National Natural Science Foundation of China under Grant Nos. 61070161, 61202449, 61320106007, the National High Technology Research and Development 863 Program of China under Grant No. 2013AA013503, the Specialized Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20110092130002, the Jiangsu Provincial Key Laboratory of Network and Information Security under Grant No. BM2003201, the Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant No. 93K-9, and the Shanghai Key Laboratory of Scalable Computing and Systems of China under Grant No. 2010DS680095.

*Corresponding Author

①http://hadoop.apache.org, Dec. 2012.

be executed as an independent MapReduce job, which initializes a larger number of map and reduce tasks for sample collection, statistical computation and accuracy estimation, and continuously provides early results with estimated confidence intervals that are progressively refined as more samples are processed until the estimate to the final result satisfies the user's exception.

The main benefits of running OLA jobs in such MapReduce-oriented cloud systems are: 1) it makes the original platform much more flexible by providing a fast and effective way to obtain approximate results within the prescribed level of accuracy rather than the accurate results generated from the completely process model, which can significantly improve the analytic performance against the massive size of the data; 2) it reduces the economic cost of users on typical pay-as-you-go cloud systems, that is an user can save money by monitoring the estimated result and killing the computation early once sufficient accuracy has been obtained; 3) it increases the overall throughput of the cluster since the released resources of early terminated OLA jobs can be delivered to the other running OLA jobs immediately, which helps to increase the parallelism degree and resource utilization.

OLA is very suitable for cloud environment; however, there are also several limitations that affect the overall OLA performance due to the inefficiencies that arise from MapReduce's rigid execution model. We conclude such inefficiencies as follows. In general, MapReduce job input data is loaded into the files in a distributed file system (DFS) wherein each file is divided into smaller blocks, then all of blocks are re-organized as the logic chunks called input splits. Each input split corresponds to a map slot. Note that the MapReduce-oriented system as Hadoop executes all jobs under the assumption that all input splits must be processed for producing the desired exact result[12]. According to this assumption, the block size is the key factor that affects the performance, since the completion time of a job is determined by the last finished tasks (it is not unusual that large blocks always take longer to process[8]). In this way, the normally used size-aware partition manner will be the better candidate for the implementation of DFS, in which the block contains the same amount of data, thus reducing possible skew of tasks when processing in parallel and improving the performance.

However, this assumption is not always true for OLA. This is because OLA is a sampling-based method to obtain approximate results from a subset of data rather than the exact one against the whole dataset, so that the sampling efficiency becomes the major factor that affects the performance especially when the data distribution is skewed. Given a skewed dataset, it is characterized by the presence of outlier tuples that are significantly different from the rest in terms of their low frequency. For the queries focusing on these outliers, there may be a majority of blocks that do not contain any tuples that satisfy the query predicate (called relevant tuples) or contain a small amount of such tuples. And the low sampling efficiency comes in several flavors: 1) for the blocks containing fewer relevant tuples, the probability of drawing sufficient relevant tuples as samples may be relatively small so that there will be few or no relevant tuples in the sample during the initial stage of OLA, leading to large error for the accuracy estimation, and 2) for the blocks without any relevant tuples, they also need to be scanned to get the random samples but contribute little to the accuracy estimation. Compared with the needs, such tasks do more unpromising work and consume excess resources, while the wasted task slots could have served other jobs, improved their sampling efficiency and helped them to finish earlier if necessary.

Irrespective of the low sampling efficiency, the redundant disk I/O cost is another factor affecting the OLA performance. Note that, the MapReduce cluster is usually built in a shared environment for different applications, so that there are many concurrent queries with the overlapped predicate that will be submitted by different users. In the case of running OLA in the original MapReduce framework, however, all of these overlapped queries are composed of many sub-tasks that are responsible for sampling from the overlapped input splits and calculating their own results independently even though the collected samples can be reused for other queries, which means a certain overlapped split may be accessed repeatedly by different tasks for sampling, leading to large redundant disk I/O cost and affecting the performance of OLA significantly.

In light of the rapid movement to the big data analytics, running OLA in cloud becomes increasingly important. However, the above limitations indicate the existing MapReduce framework cannot completely stimulate the potential of OLA and illustrate several optimization possibilities, so that an obvious question emerges: If we set out from the start to build a parallel processing model to serve as a target for running OLA in the cloud, what should that model look like? In this paper, we present the design and implementation of OLACloud based on the original MapReduce framework with some necessary extensions, which is our response to the aforementioned question.

To address the problem arising out of the low sampling efficiency especially for skewed data distribution, we exploit a content-aware repartition method. The original relation is divided into blocks according to at-

tributes. In this way, each query will conduct part of map tasks for the blocks overlapped with the predicate rather than all, increasing the sampling efficiency due to the relatively higher probability of drawing the relevant tuples, and the map slot for those blocks without any relevant tuples can be delivered to other queries which really need it.

In the case of content-aware repartition, however, the default random block allocation strategy may not obtain the desired allocation instance due to the following reasons: 1) the block size is no longer equal, that is some blocks may be very large and others may be very small, and it very likely leads to unbalancing for storage among data nodes if there is no special care is taken during the allocation, and 2) the computation load for blocks is related to their attribute intervals, rather than size-aware manner in which the computation load for each block is equal and independent, so that it may also lead to unbalancing for computation if we over-emphasize on the storage-load balancing but ignore the factor of attribute intervals. For the purpose of dealing with this derivative problem to reduce the side effects of random block allocation to our content-aware repartition method, we propose a block placement strategy called fair-allocation by carefully coupling storage and computation load to make a tradeoff between these two issues.

For the problem of large redundant disk I/O cost caused by the independent query processing model of MapReduce framework, we propose the *shared sampling* method that the samples collected by one-pass scan can be delivered to other overlapped queries efficiently, which can significantly reduce the redundancy disk I/O cost.

Our Contributions. The main contributions of this paper are summarized as follows:

• We propose an online aggregation system in the cloud called OLACloud, which is tailored for MapReduce framework, to improve the overall performance for running OLA in cloud.

• We exploit a content-aware repartition method to optimize the sampling efficiency, and present a fair-allocation block placement strategy, which is suitable for our content-aware repartition method, to guarantee the storage and computation load balancing efficiently.

• We derive a probabilistic model of block allocation to discuss the fault-tolerance property of OLACloud and the original MapReduce framework, and demonstrate the availability and effectiveness of our OLACloud.

• We present the query processing scheme with a shared sampling strategy in OLACloud, which illustrates how the shared samples are collected for multi-queries accuracy estimation, to reduce the redundant disk I/O cost for overlapped queries.

• We implement OLACloud in the modified version of Hadoop called HOP, and conduct extensive experiments to demonstrate the efficiency and effectiveness of our OLACloud.

The remainder of this paper is organized as follows. In the next section, we give an overview of our OLACloud. In Section 3, we describe the content-aware repartition method with the fair-allocation block placement strategy, and discuss the fault-tolerance property of our OLACloud. Section 4 presents the query processing scheme with the shared sampling strategy and describes the details of the statistical computation and accuracy estimation. Section 5 describes the implementation of our OLACloud. In Section 6, we introduce the experimental setup and report results of the experimental evaluation. Finally, we review related work in Section 7 and conclude the paper in Section 8.

2 Big Picture: Data Flow of OLACloud

In our implementation of OLACloud, we choose Hadoop Online Prototype (HOP)[10] as a natural candidate for the underlying query processing engine. This is because the fact that the batch-oriented original MapReduce framework cannot keep pace with the requirement of the interactive OLA processing model. HOP is a modified version of the original MapReduce framework, which is proposed to construct a pipeline between Map and Reduce so that the reduce task could start immediately as long as any Map output is generated. Such pipeline property can help to support OLA by returning the early approximate result of the query, and scaling up such result with the query progress.

In this section, we describe the overall architecture of OLACloud. Fig.1 illustrates the data flow of OLACloud, which consists of two steps: 1) content-aware repartition with fair-allocation strategy, and 2) OLA query processing with shared sampling.

The first step can be seen as a pre-processing of our OLACloud, which is implemented by two functional components: content-aware repartition and fair-allocation. This is motivated by the observation that the performance of online aggregation is actually determined by the data distribution rather than data size[2,13]. Given an input file has already been loaded into the HDFS (Hadoop Distributed File System), the task of such pre-processing is to reorganize the original file in the granularity of blocks according to the attributes, that is the block boundaries are created such that each block has the uniform intervals of partitioning columns. In this way, the outlier tuples within the identical intervals can be gathered to the same blocks, which increases the sampling efficiency due to the relatively higher probability of drawing the relevant tuples.

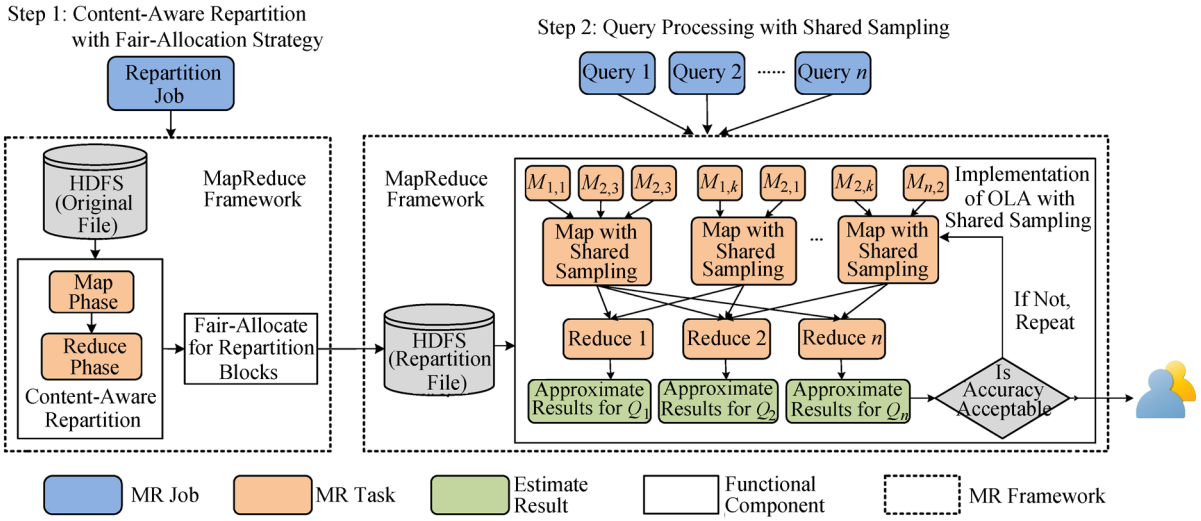


Fig.1. Data flow of OLACloud.

And each query will only run against the blocks that overlap with the query range instead of the whole dataset, so that the unpromising map tasks can be avoided and the occupied map slots can be delivered to other queries, increasing the overall sampling efficiency of all queries. For the content-aware partition, we propose a block placement strategy called fair-allocation, which replaces the default random strategy, to guarantee the storage and computation load balancing for our content-aware repartition method (as described in Subsection 3.2).

On the other hand, the second step is implemented by the component called shared sampling to support the essential procedures of OLA such as sample collection, statistic computation and accuracy estimation. The multiple queries are decomposed into a series of map tasks initially. And we can reuse the samples retrieved by one task to evaluate a number of queries rather than each query retrieves its own samples if there has potential dependency among these map tasks. As shown in Fig.1, OLACloud collects a batch of query jobs and analyzes the sharing opportunities among the queries in the granularity of task and groups the shared tasks together to form a new grouped map task, in which the samples collected are reused for accuracy estimation of each involved query. The reduce phase estimates the approximate results for the query jobs once the reducer receives a sufficient map output (a pipeline model). If the accuracy obtained is unsatisfactory, the above reduce process is repeated by taking the latest map output which is aggregated with the previous approximate results to make a new estimate for higher accuracy. The final result is returned when a desired accuracy is reached and the users can stop the query early before its completion.

3 Content-Aware Repartition with Fair-Allocation Strategy

Now we are ready to discuss the solutions for the first issue mentioned in the introduction. We first exploit a content-aware repartition method to optimize the sampling efficiency. Then we propose a fair-allocation strategy, instead of the default random block allocation method, to make the content-aware repartition method further efficient by guaranteeing both the computation and storage load balancing. Moreover, we note that the specific designed block allocation method will affect the original fault-tolerance, so we finally derive a probabilistic model of block allocation to discuss the fault-tolerance property of our fair-allocation, verifying its availability.

3.1 Content-Aware Repartition

Our content-aware repartition method can be implemented quite efficiently by making an extra normal MapReduce job called repartition job. The basic idea behind the repartition job is to scramble the tuples in the original input blocks and combine the appropriate tuples together to form a set of new blocks according to the attributes, that is the block boundaries are created such that each new block has the uniform intervals of partitioning columns. There are three major issues need to be considered firstly: 1) how to determine the partitioning columns, 2) how to determine the number of intervals for each partitioning column, and 3) how to execute the repartition job efficiently in MapReduce framework.

Towards the first issue, a straightforward method is to consider all columns in the relation as the partitioning columns. This method is scalable with regard to all

kinds of queries focusing on different columns, but suffering from the curse of dimensionality (result in a large number of blocks which increases the system overhead for block management). An alternative approach, which we adopt in this paper, is to determine the partitioning columns based on the query workload. A query training set can be collected as representative of the workload. The idea of this scheme is to partition the columns which are frequently accessed by the queries in the workload and update the partitioning columns periodically if the access characteristics of workload change (the details of the update operation will be discussed in Subsection 3.2).

For the second issue, we introduce a parameter called partition size to stand for the number of intervals for each partitioning column. A possible method to determine partition size is to customize the partition size for each partitioning column based on its own skewness couple with the underling cluster size. However, this method requires the system to have some priori understanding of the data distribution and the system configuration. In this paper, to facilitate the implementation of OLACloud, we take the partition size as an input parameter that is predefined by users according to their experience, and the calculation of the theoretical optimal value will be discussed in our future work.

After the partitioning columns and partition size are determined, we are now ready to introduce how the repartition job executes efficiently in MapReduce framework. Algorithm 1 shows the configuration of this repartition job, in which there are two input parameters: the value range of columns $\mathcal{R}$ and the partition size of columns $\mathcal{S}$, which are defined as follows.

Algorithm 1: Repartition Job
Input: ValueRange $\mathcal{R}$, PartitionSize $\mathcal{S}$
1 BlockSet $\mathcal{B}$ = `getBlockSet (`$\mathcal{R}$`, `$\mathcal{S}$`)`;
2 JobConf *repartition* = new JobConf(*repartition*.class);
3 `//initialize the repartition job, including map and reduce class and the in/output format, etc.`
4 DefaultStringifier.store(*repartition*, $\mathcal{B}$, "*blockSet*)";
5 JobClient.`runJob`(*repartition*);

Definition 1 (Value Range of Columns $\mathcal{R}$ and Partition Size of Columns $\mathcal{S}$). *Given a dataset with n partitioning columns $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$, the value range of columns is denoted by $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, where R_i represents the value range of C_i. And the partition size of columns is denoted by $S = \{S_1, S_2, \ldots, S_n\}$, where S_i represents the partition size of C_i (C_i will be partitioned into S_i intervals).*

In our implementation, $\mathcal{S}$ is predefined by the user, and the parameter $\mathcal{R}$ is available in the meta-data. Based on the two parameters, we can obtain the block set denoted by $\mathcal{B}$ (line 1), and store this variable in JobConf to support the map logic (line 3). To simplify the presentation, we define $\mathcal{B}$ as follows.

Definition 2 (Block Set $\mathcal{B}$). *Let each column C_i be uniformly partitioned into S_i intervals according to the existing range of values in each C_i (e.g., for each column C_i, we partition it into S_i intervals with equal size $\frac{R_i}{S_i}$), denoted by $\mathcal{I}_i = \{I_i^1, I_i^2, \cdots, I_i^{S_i}\}$. Thus, $\mathcal{B} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$, where $|\mathcal{B}| = \prod_{i=1}^{n} S_i$. Each block is identified by a certain $B_i \in \mathcal{B}$.*

The map and reduce logic are shown in Algorithm 2 and Algorithm 3 respectively. The map phase takes the original HDFS file as input with the default InputFormat type called TextInputFormat and evaluates the scanned tuple on the partitioning columns to decide which is the destination block it belongs to (line 3 of Algorithm 2).

Algorithm 2: Repartition: Map Logic
Input: LongWritable *key*, Text *value*
Output: OutputCollector⟨LongPair, Text⟩ *outputKV*
1 BlockSet $\mathcal{B}$ = DefaultStringifier.`load`(*repartition*, "*blockSet*", BlockSet.class);
2 **for** $\forall\langle k_i, v_i\rangle$ in input **do**
3 *block*ID=`getBlockID` (*value*, $\mathcal{B}$);
4 *rank* = *random*.`nextLong()`;
5 *outputKV*.`collect`(⟨*blockID*, *rank*⟩, *value*);
6 **end**

Algorithm 3: Repartition: Reduce Logic
Input: LongPair *key*, Iterator⟨Text⟩ *values*
Output: OutputCollector⟨Text, NullWritable⟩ *outputKV*
1 **while** *values*.`hasNext()` **do**
2 *outputKV*.`collect`(*values*.`next()`, null);
3 **end**

In our implementation of repartition job, we hope the output of each reducer has the great randomness of tuples so that sequentially scanning the repartition block gives rise to a stream of random samples with lower I/O cost rather than random sampling from the repartition block. This is because the fact that the sampling performs worse than simply scanning the whole block[14] since completely random disk access can be five orders of magnitude slower than sequential access[15]. Based on such motivation, each mapper uses the block identifier combined with a random Long integer *rank* to form a new structure called LongPair as the output key, while the tuple as the output value (lines 4∼5 of Algorithm 2). In the shuffle phase, we use a customized hash function to make each reducer dedicated for a unique block identifier. And we override the compareTo method of LongPair, which is invoked

by each reducer automatically before the reduce operation, to realize the secondary sort according to *rank*, leading to the great randomness of input pairs for each reducer. Then, each reducer receives such input pairs and collects them into the output file based on the default OutputFormat type called TextOutputFormat.

Moreover, there is another important issue needs to be considered before the content-aware partition manner exerts its effectiveness, that is how to determine the relevant blocks for a given query Q. In this paper, we adopt the method proposed in [13] called NRB-T (Nested Red-black Tree) to solve the problem (interested readers may find the details of NRB-T in [13]). Such NRB-T can be considered as an index of blocks and the core idea behind NRB-T is to associate each partition with its intervals in a hierarchical model. Each node of our NRB-T comprises four annotations: *Partitions*, *Interval*⟨*low, High*⟩, *Max* and *Pointer*. The *Max* annotation records the maximum *High* value across both its subtrees. And the *Interval* and *Max* values are used during the probe phase.

In our implementation, we deploy it as a component of HDFS. And we parse the query conditions into a set of probe intervals to lookup the overlapped blocks from NRB-T. Given the matching results by probing NRB-T, our OLACloud then checks BlockMaps to obtain the corresponding metadata of each relevant block to support the query processing.

3.2 Update Strategy

In our implementation of content-aware repartition job, the partitioning columns are determined based on a carefully collected query workload. And our method cannot always guarantee the effectiveness if the current workload changes a lot in the next moment, so we need an efficient update strategy to adjust the repartition instance if necessary. The update strategy of OLACloud is designed based on the fact that the updates to the data warehouse system are usually performed in a batch mode. We collect all the updates and commit them periodically. During the update period, our system will stop processing any queries and update the data as follows:

Case 1: *Incremental Update*. If the current configuration of repartition job, such as partitioning columns and partition size, can afford to the new query workload, then we upload the update data into the same directory with original data, conduct the same repartition job only for the update data and insert the corresponding block identify into the *Partition* scope of existing NRB-T node simultaneously.

Case 2: *Repartition for All Data*. If the new query workload is different from the previous one, which means the current configuration of repartition job cannot keep pace with the changes of query requirement, then we need an overall repartition for both the original and the update data to satisfy the new query workload. We firstly upload the update data into the same directory with original data, and then take the overall data as the input of the repartition job to partition them according to the new configuration. Finally we also need to construct a new NRB-T for the overall repartition job.

3.3 Fair-Allocation Strategy

After the content-aware repartition is completed, OLACloud needs to allocate the blocks to the appropriate data nodes. The traditional MapReduce framework adopts the random block allocation strategy for the size-aware partition method, which can guarantee the storage and computation load balancing to a certain degree. And there are also several other well-known placement strategies proposed in [16]. Suppose there are N data nodes, then these placement strategies can be represented as 1) round-robin method: iterate over blocks in some order and assign them to each data node in turn, 2) range method: split the blocks into N disjoint chunks and assign all blocks within a chunk to a data node, or 3) chunk-cyclic method: split the blocks into M regular chunks of N blocks each and iterate over the blocks of a chunk in some predefined order and assign them to each of the N data nodes in turn.

However, all of above strategies are not suitable for our content-aware repartition manner, since they do not fully consider the factors of block size and the correlation between blocks, leading to the load unbalancing for both storage and computation. For example, consider a dataset partitioned by a content-aware method which consists 16 blocks (with the block size in the center) labeled 1 to 16 in the row major order, the assignment instances of the above mentioned three methods are shown in Fig.2, where the round-robin and range methods have well storage balancing than chunk-cyclic (lower variance of the block size) but with poorer computation distribution (denoted by the blocks with dotted lines) for the given two queries Q_1 and Q_2. The problems resulted from such storage and computation load unbalancing can be summarized as: 1) the waste of storage resources, and 2) the reduction of query processing parallelism, which impacts the performance of OLACloud.

In this paper, we present a fair-allocation block placement strategy, which suits for our content-aware repartition method, to guarantee the storage and computation balancing by carefully coupling block size and correlation between blocks. The block size is the factor

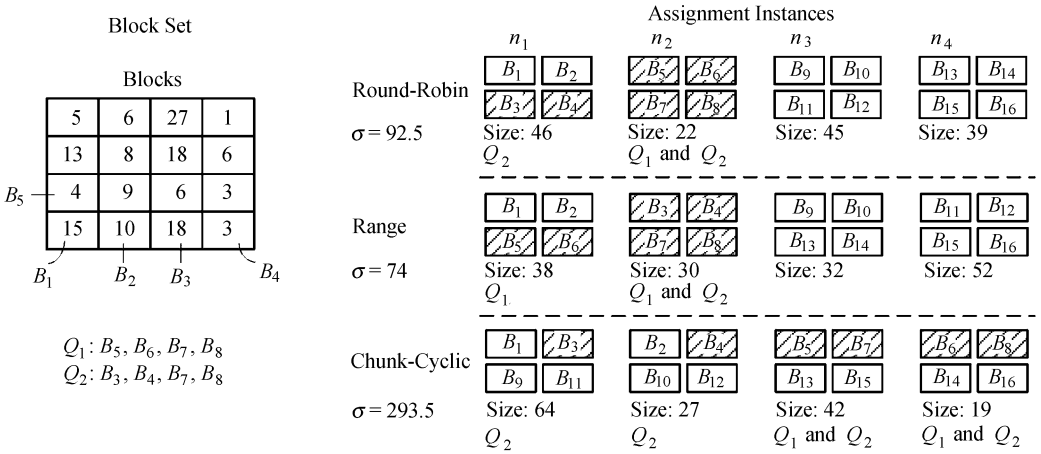


Fig.2. Assignment instances of round-robion, range and chunk-cyclic methods.

considered for the storage balancing, and we use the variance of storage consumption in each data node as the metric to measure the storage balancing; while the meaning of correlation between blocks is that some blocks will be accessed together for a set of queries. If we place these blocks in the same data node, then more map tasks of the same queries will be processed in this node, increasing the computation load of this node and reducing the parallelism degree of these queries. In order to quantify the correlation between blocks, we introduce the concept of "block distance" (D) as the metric to measure the computation balancing. And we also introduce "correlated queries" (C) to simplify the presentation. We take a 2-dimensional (2D) dataset as an example to show the definitions of D and C as follows.

Definition 3 (Block Distance). *Let B_i and B_j denote two blocks generated by our content-aware partition manner. Then, we use the number of blocks belong to the minimum bounding rectangle (MBR) of B_i and B_j as the distance denoted by $D(B_i, B_j) = |MBR(B_i, B_j)|$.*

The MBR which is also known as bounding box or envelope, is an expression of the maximum extents of a 2D object, which is typically used in the data structure of R-tree to group and represent the nearby objects. A simple example of MBR is shown in Fig.3. Given a set of blocks generated from our repartition job, in which the block distance of (B_1, B_6) and (B_6, B_{15}) is the number of blocks belongs to $MBR(B_1, B_6)$ (red rectangle) and $MBR(B_6, B_{15})$ (blue rectangle) respectively.

B_{13}	B_{14}	B_{15}	B_{16}
B_9	B_{10}	B_{11}	B_{12}
B_5	B_6	B_7	B_8
B_1	B_2	B_3	B_4

Fig.3. Example of block distance.

Definition 4 (Correlated Query). *Consider two blocks B_i and B_j, if the query range of a given query overlaps with or fully contains all $D(B_i, B_j)$ blocks, we say this query is the correlated query for B_i and B_j. And all correlated queries for B_i and B_j are denoted by $C(B_i, B_j)$.*

Without loss of generality, we assume the queries with different predicates have the same probability of occurrence. And this assumption is reasonable in practice, e.g., the random queries are often used to evaluate the effectiveness of a given system. Consider a special case that three blocks B_i, B_j and B_k in the same row based on the row major order, where $i < j < k$. Based on the above definitions, we have $D(B_j, B_k) < D(B_i, B_k)$ and $|C(B_j, B_k)| > |C(B_i, B_k)|$ since $\forall Q \in C(B_i, B_k)$ also belongs to $C(B_j, B_k)$, but $\exists Q \in C(B_j, B_k)$ may not in $C(B_i, B_k)$. Thus, we note that the smaller $D(B_j, B_k)$ indicates the higher correlation between B_j and B_k, that is B_j and B_k may be accessed together for the broader set of correlated queries with higher probability since $|C(B_j, B_k)| > |C(B_i, B_k)|$.

Therefore, a natural idea for the purpose of computation balancing would be to place the blocks with higher correlation into different data nodes, distributing the computation load among all data nodes. Given a set of data nodes N^* and two blocks B_i and B_j with $D(B_i, B_j) \leqslant |N^*|$, an obvious way to obtain the better parallelism of the queries in $C(B_i, B_j)$ is to place all blocks in $MBR(B_i, B_j)$ to different nodes preferably, which means the data node containing block B_j with $D(B_i, B_j) \leqslant |N^*|$ will have the lower priority for assigning B_i to it and we prefer to select the data node containing the block B_j with $D(B_i, B_j) > |N^*|$ as the candidate node. However, the larger $D(B_i, B_j)$ is not always good for the final assignment. If two blocks with the lower correlation are placed together, then the

probability of placing two blocks with higher correlation together will be increased. As shown in Fig.4, the first assignment is optimal, where B_5 is assigned to n_3. But in the second assignment, if we assign B_5 to n_4, then B_6, B_7 and B_8 will be placed to n_1, n_2 and n_3 respectively, reducing the parallelism of the queries. Therefore, the data node containing the block B_j with the smaller $D(B_i, B_j) > |N^*|$ will have the higher priority for assignment.

Based on the above description we conclude the principles for placing a given block B_i as follows, where *avgdis* is a variable used to measure the correlation between B_i and a certain data node.

1) Data node N_i with $avgdis(B_i, N_i) > |N^*|$ has the higher priority than the ones with $avgdis(B_i, N_i) \leqslant |N^*|$.

2) For $\forall N_i \in N^*$ with $avgdis(B_i, N_i) > |N^*|$, the smaller $avgdis(B_i, N_i)$ presents the higher priority for assignment.

3) For $\forall N_i \in N^*$ with $avgdis(B_i, N_i) \leqslant |N^*|$, the larger $avgdis(B_i, N_i)$ presents the higher priority for assignment.

We are now ready to discuss how *avgdis* is calculated, as shown in Algorithm 4. For each data node N_i, we calculate the distance between B_i and $\forall B_j \in B^*_{n_i}$ (line 4, where $B^*_{n_i}$ indicates the set of blocks in data node N_i), denoted by $\mathcal{D} = \{D_1, D_2, \ldots, D_{|B^*_{n_i}|}\}$. $\mathcal{D}$ can be classified into three types: 1) $\mathcal{D}$ contains all $D > |N^*|$, 2) $\mathcal{D}$ contains all $D \leqslant |N^*|$, and 3) $\mathcal{D}$ contains some $D \leqslant |N^*|$ and others $D > |N^*|$. In the implementation of our fair-allocation, we adopt a simple greedy strategy to assign each B_i to the data node N_i with the minimal side effects to $\forall B_j \in B^*_{n_i}$, so that we only consider the blocks with $D \leqslant |N^*|$ for *avgdis* calculation in both type 2 and type 3. We use the variables *large* and *less* to record all $D > |N^*|$ and the other $D \leqslant |N^*|$ respectively (lines 5∼8). For the first type of $\mathcal{D}$, we use the average value of *large* as *avgdis* (line 10). While we use the average value of *less* as *avgdis* for type 2 and type 3 (line 11).

Algorithm 5 shows the idea of fair-allocation strategy by considering the factor of block size (storage variance) and correlation between blocks (*avgdis*). Given a node set N^* and a block set B^*, we first calculate the storage variance (*var*) of all possible placement instances for $\forall B_i \in B^*$, then record the pair of (N_i, var) in the variable *sRank* and sort the data nodes according to the variance in an ascending order (lines 4∼5, 9).

Algorithm 4: AvgDistance
Input: Block B_i, Node N_i, int $|N^*|$
Output: double *avgdis*

```
1   large, less ← ∅;
2   B*_{n_i} = getAllBlocks(N_i);
3   for ∀B_j ∈ B*_{n_i} do
4       distance = MBR(B_i, B_j);
5       if distance > |N*| then
6           large.add(distance);
7       end
8       else less.add(distance);
9   end
10  if less = ∅ then avgdis = large.getAverage();
11  else avgdis = less.getAverage();
12  return avgdis;
```

Algorithm 5: FairAllocation
Input: NodeSet N^*, BlockSet B^*, double w_s, double w_c
Output: HashTable *allocation*

```
1   allocation, sRank, cRank, aRank ← ∅;
2   for ∀B_i ∈ B* do
3       for ∀N_i ∈ N* do
4           var = getStroageVar(B_i, N_i);
5           sRank.put(N_i, var);
6           avgdis = AvgDistance(B_i, N_i, |N*|);
7           cRank.put(N_i, avgdis);
8       end
9       sRank.sort(var);
10      cRank.sort(avgdis, Comparator);
11      for ∀N_i ∈ N* do
12          sIndex = sRank.getIndex(N_i);
13          cIndex = cRank.getIndex(N_i);
14          fIndex = sIndex × w_s + cIndex × w_c;
15          fRank.put(N_i, fIndex);
16      end
17      fRank.sort(fIndex);
18      allocation.put(B_i, fRank.getFirst());
19  end
20  return allocation;
```

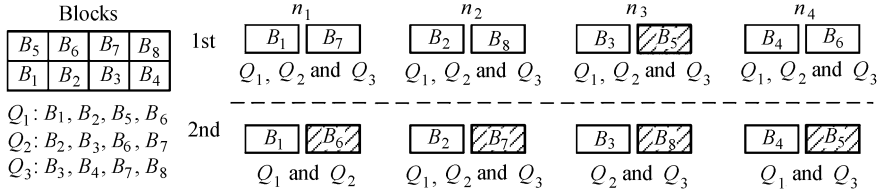


Fig.4. Example of placing the lowest correlation blocks together.

sRank indicates the priority of a data node is selected as the candidate node for storage balancing, and the smaller index of *sRank* represents the higher priority. On the other hand, for each data node N_i, we calculate *avgdis* between B_i and $\forall B_j \in N_i$, and then record the pair of $(N_i, avgdis)$ in variable *cRank* and sort the data nodes according to *avgdis* by an overwritten comparator, which is implemented to satisfy the placement principles mentioned above (lines 6∼7, 10). *cRank* indicates the priority of a data node is selected as the candidate node for computation balancing, and the smaller index of *cRank* represents the higher priority.

In fair-allocation, we use the index of *sRank* and *aRank* denoted as *sIndex* and *cIndex* respectively as the normalized parameters to measure the final priority of each node, denoted as *fIndex* (lines 11∼16), by fully coupling the storage and computation factors as the following formula.

$$fIndex = sIndex \times w_s + cIndex \times w_c,$$

where w_s and w_c are the weights assigned to *sRank* and *cRank* and $w_s + w_c = 1$. Through the adjustment of w_s and w_c, fair-allocation strategy can obtain much more flexibility for different placement requirements.

Then, we record the pair of $(N_i, fIndex)$ in *fRank* and rank the data nodes according to *fIndex* in an ascending order (line 17). Smaller *fIndex* represents higher priority of a data node N_i being selected as a candidate node for B_i to allocate, and we use the variable *allocation* to record the destination data node for $\forall B_i \in B^*$ (line 18). After all B_i are processed, OLACloud then allocates each block to the corresponding data node according to *allocation*. Fig.5 shows the fair-allocation instance ($w_s = w_c = 0.5$) of the example shown in Fig.2. Compared with the other methods, our fair-allocation strategy can guarantee the computation balancing with an acceptable storage variance.

3.4 Fault-Tolerance of Fair-Allocation with Replicas

Fault-tolerance is an important problem in cloud environment, a straight way to handle it in original MapReduce framework is to create replicas and allocate them randomly, and then restart the failed tasks with the replicas in other data nodes. A natural idea for fair-allocation with replicas is to allocate all replicas according to the variable *fIndex*, which can guarantee the storage and computation load balancing in a strict way. However, this method will result in a poor fault-tolerance since the number of candidate data nodes for each block is relatively smaller.

In order to gain the great benefit of fair-allocation and guarantee an acceptable fault-tolerance, we calculate *fIndex* only based on the first replica (also called primary replica) of all blocks, and allocate the primary replica based on such *fIndex*, while the other replicas do not be considered in the calculation of *fIndex* and are allocated in the random way as the original MapReduce framework. In another word, only the primary replica of blocks in a given data node is used for the calculation of *fIndex* in our implementation. When OLACloud is processing, we prefer to assign tasks to the data node with primary replicas, which can maximize the performance advantage of fair-allocation for computation load balancing as much as possible (discussed in Section 5).

We derive a simple probabilistic model of block allocation to compare the fault tolerance of our fair-allocation strategy and the default random strategy in original MapReduce framework. The model notations are summarized in Table 1. Suppose that the cluster consists of n data nodes for storage and computation. And we store s blocks in these data nodes, that is B_i for $1 \leqslant i \leqslant s$. Let r be the desired number of replicas and I_{ij} an indicator for the event that data node j stores a

Table 1. Symbol Description

Symbol	Definition
n	Number of data nodes
f	Number of failed data nodes
s	Number of blocks
r	Replication factor
B_i	Block i, $i \in \{1, \ldots, s\}$
I_{ij}	Indicator for the event that data node j stores a replica of block B_i
P	Probability of data loss for any block
P_i	Probability of data loss for B_i

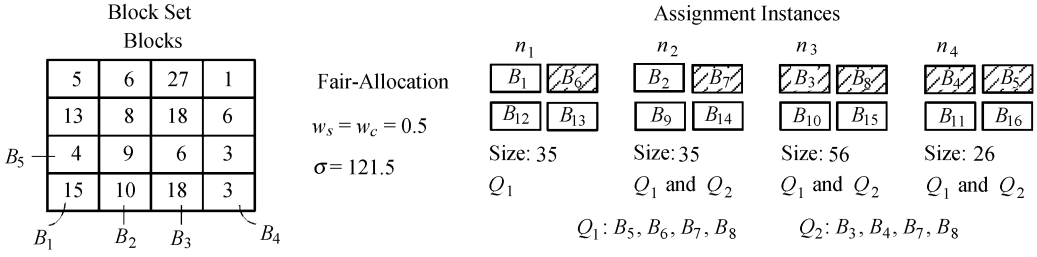


Fig.5. Result of fair-allocation.

replica of block B_i, where $\sum_j I_{ij} = r$ for all i. For simplicity, we also assume that each node has sufficient disk space to store all the blocks assigned to it. And we take the probability of data loss as the metric to analyze the fault-tolerance when f ($f \geqslant r$) data nodes of the cluster fail.

Random Strategy. We are now ready to discuss the fault-tolerance of random strategy. Note that, the vector $\boldsymbol{I}_{i*} = (I_{i1}, \ldots, I_{in})$ indicates for each node whether it stores a replica of block B_i. By default, replicas are randomly distributed to each set of r data nodes with equal probability. Thus, $\boldsymbol{I}_{i*}$ has the distribution based on the analysis in [17]:

$$P_r(\boldsymbol{I}_{i*} = \mathcal{I}) = \begin{cases} \binom{n}{r}^{-1}, & \text{if } \sum_j \mathcal{I}_j = r, \\ 0, & \text{otherwise,} \end{cases}$$

where $\mathcal{I} \in \{0,1\}^n$. Note that, B_i is lost only when all r replicas are located on the set of failing data nodes, that is $\sum_{j=1}^{f} I_{ij} = r$. Therefore, we can calculate the probability of data loss for each B_i as:

$$P_i = P_r\Big(\sum_{j=1}^{f} I_{ij} = r\Big) = \binom{f}{r} \Big/ \binom{n}{r}.$$

Thus, P can be computed as follows:

$$P = 1 - \prod_{i=1}^{s}(1 - P_i) = 1 - \left[1 - \binom{f}{r} \Big/ \binom{n}{r}\right]^s.$$

Fair-Allocation. The analysis for fair-allocation is similar, but the exact distribution of $P_r(\boldsymbol{I}_{i*} = \mathcal{I})$ is difficult to compute since 1) the placement of each block depends on the blocks which have been processed rather than the completely independent placement of random strategy, and 2) the value of *fIndex* for each node cannot be calculated without the information of block size. Therefore, we only consider the relaxation for the computation of $P_r(\boldsymbol{I}_{i*} = \mathcal{I})$:

• The first block B_1 has $\binom{n}{r}$ opportunities to place the r replicas such that each placement instance has the same probability of $\binom{n}{r}^{-1}$ as the case of random strategy.

• The primary replica of each B_i ($i \in [2, \min\{n, s\}]$) is randomly placed into one of the $n - i + 1$ "empty data nodes". Such "empty" can be explained as these $n - i + 1$ data nodes are not considered in the calculation of *fIndex* for B_i since they have not contained any primary replicas yet. And the other $r - 1$ replicas are placed into the rest $n - 1$ data nodes randomly (except the node containing the primary replica of B_i). Each block will have $\binom{n-i+1}{1} \cdot \binom{n-1}{r-1}/r$ placement instances after eliminating the replicative one.

• For the remaining blocks, only the data node with the smallest *fIndex* can be selected as the candidate for allocation. Without loss of generality, there may be several data nodes that have the smallest *fIndex* simultaneously, however, the number of these data nodes is hard to determine. This is because *fIndex* cannot be calculated without the information of block size, so that the number of placement instance cannot be computed exactly. In the worst case, we suppose there is only one data node has the smallest *fIndex*, which means the primary replica can only be assigned to a certain data node, and the rest $r - 1$ replicas have $\binom{n-1}{r-1}$ opportunities for placement.

Thus, $\boldsymbol{I}_{i*}$ has approximate distribution as:

$$P_r(\boldsymbol{I}_{i*} = \mathcal{I}) = \begin{cases} \binom{n}{r}^{-1}, & \text{if } i = 1, \\ [\binom{n-i+1}{1} \cdot \binom{n-1}{r-1}/r]^{-1}, & \text{if } 2 \leqslant i \leqslant m, \\ \binom{n-1}{r-1}^{-1}, & \text{if } m < i \leqslant s, \\ 0, & \text{otherwise,} \end{cases}$$

where $m = \min\{n, s\}$. And the probability of data loss P for fair-allocation can be calculated by the following formula.

$$P = 1 - \left[1 - \binom{f}{r} \Big/ \binom{n}{r}\right] \times \prod_{i=2}^{m} \left[1 - \left(\frac{\binom{f}{r} \cdot r}{\binom{n-i+1}{1} \times \binom{n-1}{r-1}}\right)\right] \times \left[1 - \binom{f}{r} \Big/ \binom{n-1}{r-1}\right]^{s-m}.$$

Interpretation. We set $n = 40$, $r = 3$, $s \in [20, 1000]$ and $f \in [4, 15]$. Fig.6 shows that the probability of data loss of our relaxation fair-allocation is slightly higher than the random strategy when $s = 20$, and larger s makes such weakness much more obvious. This is because the placement of the random strategy is completely independent so that the number of candidate data nodes for each block in fair-allocation is much less than the random strategy, which increases the value of $P_r(\boldsymbol{I}_{i*} = \mathcal{I})$ and leads to a higher P. In order to reduce the data loss probability of fair-allocation, we increase the number of replica. Fig.7 shows the example for $s = 20$, $s = 50$, $s = 100$ and $s = 1\,000$, which indicates the worst case of fair-allocation for $r = 4$ can obtain the acceptable data loss probability for $r = 3$. Overall, our fair-allocation does not have a negative effect on data loss when we set the replica factor to 4, which means it trades storage for fault-tolerance, but can make OLA much more efficient (significantly improve the performance of OLA).

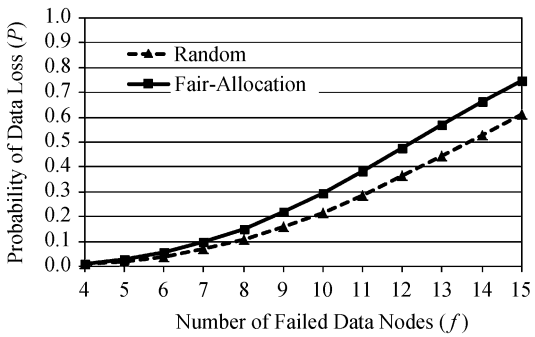


Fig.6. Comparison of fault-tolerance for $r = 3$.

4 Query Processing with Shared Sampling

4.1 Query Processing Scheme

In order to overcome the second limitation mentioned in Section 1, we propose a shared sampling strategy to retrieve unbiased samples, which can be reused to support statistical calculation and accuracy estimation for multiple queries. Our shared sampling strategy has motivation similar to the work in [18], and extends it by adding a customized sample management component to satisfy the requirements of multiple OLA-queries processing. The basic idea behind the shared sampling strategy is to combine the appropriate queries together and collect samples for them by one-pass disk scan.

The query processing scheme with shared sampling strategy can be split into two phases. In the first phase, OLACloud does not execute the incoming query immediately but collects the queries and analyzes the potential shared opportunities. Then the set of incoming queries are combined as a grouped *dynamic* MapReduce job. The reason we call it dynamic is that each grouped map task needs to be dynamically configured based on such shared information to be responsible for shared queries processing. In the second phase, the grouped dynamic MapReduce job is initialized and submitted to JobTracker for parallel processing. Each grouped map task builds a public sample buffer (PSB) to manage the random samples drawn from the input block, and collects samples from PSB rather than disk for all shared queries to compute the corresponding statistics, reducing the redundant disk I/O cost. And each reduce task collects the statistics from the map phase to calculate the approximate result and estimate the query accuracy.

Algorithm 6 illustrates the general idea of query processing with shared sampling strategy. Given a set of incoming queries $\mathcal{Q}$, OLACloud collects a subset of queries denoted by $\mathcal{Q}_{thd}$ with the size of *threshold* (line 2), and analyzes the potential sharing opportunities among the queries of $\mathcal{Q}_{thd}$, denoted as *share* (line 3). There are two major issues need to be considered in the generation of *share*: 1) which queries need to be combined for sharing, and 2) how many samples need to be collected from a certain grouped map task for each involved query.

Towards the first issue, we define the queries with the overlapped predicates as the candidates for sharing. Consider the query set $\mathcal{Q}_{thd} = \{Q_1, Q_2, \ldots, Q_n\}$ with

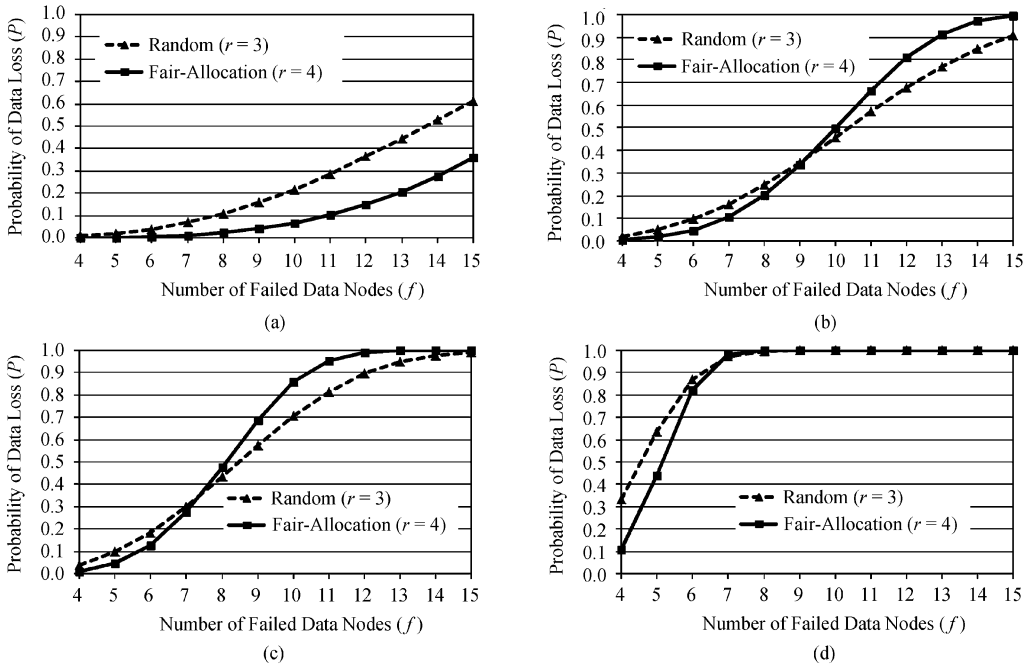


Fig.7. Comparison of fault-tolerance for different s. (a) $s = 20$. (b) $s = 50$. (c) $s = 100$. (d) $s = 1000$.

Algorithm 6: QueryProcess
Input: QuerySet $\mathcal{Q}$
1 **while** $\mathcal{Q}$.`hasNextQuery()` **do**
2 QuerySet $\mathcal{Q}_{thd}$ = `collect`($\mathcal{Q}$, *threshold*);
3 ShareInfo *share* = new `ShareInfo`($\mathcal{Q}_{thd}$);
4 EstimateInfo *estimate* = new `EstimateInfo` (*confidence*, *errRate*, *column*, *type*);
5 JobConf *OLAjob* = new `JobConf`(*OLA*.class);
6 DefaultStringifier. `store`(*OLAjob*, *share*, "*share*");
7 DefaultStringifier. `store`(*OLAjob*, *estimate*,"*eInfo*");
8 JobClient.`runJob`(*OLAjob*);
9 **end**

the predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, and the input file F with the block set $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$. The overlapped queries for each $B_i \in \mathcal{B}$ can be obtained by checking the predicate set $\mathcal{P}$, denoted by $\mathcal{Q}^i_{olp} = \{Q_x \in \mathcal{Q}_{thd} | B_i \in (P_x \cap \mathcal{B})\}$. Based on the traditional MapReduce framework, $\forall Q_x \in \mathcal{Q}^i_{olp}$ contains a map task M^i_x to process B_i; while in our OLACloud, all of $\{M^i_x | Q_x \in \mathcal{Q}^i_{olp}\}$ are combined to form a new grouped map task M^i_{com}.

For the second issue, we adopt the sampling method which shares the similar motivation with the *distributed sampling* proposed in [19] to decide the sample size for each query involved in M^i_{com} and guarantee the samples collected for each query are unbiased, which means the sample size collected from each block is proportional to its cardinality. For example, we consider the query $Q_x \in \mathcal{Q}_{thd}$ with predicate P_x, then $\forall B_i \in (P_x \cap \mathcal{B})$ provides k^x_i samples for Q_x, where $k^x_i = \frac{|B_i|}{\sum_{i=1}^{|P_x \cap \mathcal{B}|} |B_i|} \times k$, $k = \sum_i k^x_i$ is the total required sample size in each iteration of OLA, which is defined by the user. For $\forall B_i \in \mathcal{B}$, we record the queries of $\mathcal{Q}^i_{olp}$ with their required sample size $K = \{k^x_i | Q_x \in \mathcal{Q}^i_{olp}\}$ as a tuple in *share*, which is stored as a parameter of JobConf (line 6).

Moreover, we also need to record the user defined confidence, error rate, and the aggregate column and type for $\forall Q_x \in Q_{thd}$ as an item in the accuracy parameter called *eInfo* to support the accuracy estimation (line 7).

4.2 MapReduce Implementation

4.2.1 Queries for Single Relation

We are now ready to discuss the Map logic for single relation queries as shown in Algorithm 7. Given a block B_i, the corresponding grouped map task M^i_{com} loads *share* from *OLAjob*, which can be implemented by overwriting the configure function in MapReduceBase, and extracts the item for B_i into the variable *qInfo*, which is used to accomplish the sample collection and statistic calculation, with the key $Q_x \in \mathcal{Q}^i_{olp}$ and values $k^x_i \in K$ (lines 1∼2). Afterwards, M^i_{com} adapts the shared sampling strategy to build a PSB, then all the queries of $\mathcal{Q}^i_{olp}$ will collect samples from PSB rather than disk, reducing the redundant disk I/O cost. Given a block B_i and the overlapped query set $\mathcal{Q}^i_{olp}$, there are two major issues need to be considered for shared sampling firstly, that is 1) how to construct the PSB for all queries of $\mathcal{Q}^i_{olp}$, and 2) how to efficiently update the PSB to suit for the progress of $\forall Q_x \in \mathcal{Q}^i_{olp}$.

Algorithm 7: Map Logic
Input: LongWritable *key*, Text *value*
Output: OutputCollector⟨IntWritable, Text⟩ *outputKV*
1 ShareInfo *share* = `DefaultStringifier.load`(*OLAjob*, "*share*", ShareInfo.class);
2 Hashtable *qInfo* = `getQueryInfo`(*share*);
3 *PSB*.`init()`;
4 *pointers*.`init()`;
5 **for** $\forall\langle k_i, v_i\rangle$ in input **do**
6 *PSB*.`update`($\langle k_i, v_i\rangle$, *pointers*);
7 **for** $\forall Q_x \in$ *qInfo*.`keys()` **do**
8 **if** *PSB*.`hasEnoughSample`(Q_x) **then**
9 *samples* = `getSamples`(*qInfo*.`get`(Q_x), *PSB*);
10 *stats* = `getStats`(*samples*, Q_x);
11 *pointers*.`update()`;
12 *outputKV*.`collect`(Q_x.*qID*, ⟨*stats*, this.*taskID*⟩);
13 **end**
14 **end**
15 **end**

Consider the first issue, we initialize the PSB with the control parameter *pointers*, which is used to realize the management of PSB, in lines 3∼4. In our implementation, *pointers* includes the pointers to the first and last samples that have been visited by all queries of $\mathcal{Q}^i_{olp}$ denoted as *first* and *last* respectively, and the pointer to the next sample needed to be visited by a given $Q_x \in \mathcal{Q}^i_{olp}$ is denoted as $cur(Q_x)$ (also called progress pointer). Note that, the set of progress pointers $\{cur(Q_1), cur(Q_2), \ldots, cur(Q_n)\}$ is used to control the sampling procedure for all queries of $\mathcal{Q}^i_{olp}$, and the interval [*first*, *last*] indicates the samples in PSB that need to be updated, where $last = \min\{cur(Q_1), cur(Q_2), \ldots, cur(Q_n)\}$.

For the second issue, the combined map task M^i_{com} adopts an aggressive method to update PSB for each incoming input $\langle k_i, v_i\rangle$ (add sample to PSB if it is not full and swap out the expired sample if *last*>*first*) (line

6), and collect samples to calculate the statistics denoted as *stats* (the details of statistics calculation are discussed in Subsection 4.3) for $\forall Q_x \in \mathcal{Q}^i_{olp}$ if PSB has enough samples (k^x_i) for Q_x (lines 9∼10). And the corresponding progress pointers need to be updated along with the sample consumption (line 11). Or alternatively, we also can define an input buffer to collect a set of input key/value pairs, then update the PSB and collect samples to calculate statistics in a batch model to reduce the number of operations. Finally, the output collected with the key is the query ID of Q_x and the value is combined by *stats* and the task ID of M^i_{com} (line 12).

The reduce phase is responsible for collecting statistics from map tasks and calculating the estimate of the final query result. In our implementation, each reduce task is only responsible for a certain Q_x, and Algorithm 8 illustrates the logic of reduce phase. Given a certain Q_x, the reduce task R_x loads the accuracy parameter *eInfo* from *OLAjob* and initializes a variable *container* to classify the input *values* from all involved M^i_{com} (lines 1∼4).

Algorithm 8: Reduce Logic

Input: IntWritable *key*, Iterator⟨Text⟩ *values*

Output: OutputCollector⟨IntWritable, Text⟩ *outputKV*

```
1  EstimateInfo eInfo = DefaultStringifier
   load(OLAjob, "eInfo", EstimateInfo.class);
2  Hashtable container = new Hashtable();
3  while values.hasNext() do
4      update (container, values.next());
5      if container.isAvailable() then
6          unistats = uniStats(container, unistats);
7          result = estimate(unistats);
8          if
           isAccuracy(result, eInfo, unistats)
           then
9              outputKV.collect(key, ⟨result, "accept"⟩);
10         else
11             outputKV.collect(key, result);
12         end
13     end
14 end
```

For each M^i_{com}, there is a corresponding item in *container* to record the statistics of Q_x with the key $M^i_{com}.taskID$ and value $\{stats_1, \ldots, stats_n\}$, where $stats_j$ indicates the statistics calculated in the j-th iteration of OLA. Afterwards, R_x needs to check *container* to make sure it is available for accuracy estimation, which means *container* has contained the statistics for all M^i_{com}. Then R_x can calculate the unified statistics (*unistats*) in an incremental way, that is *unistats* of the j-th iteration can be computed by aggregating all $stats_j$ in *container* with the previous *unistats* of the $(j-1)$-th iteration (line 6). And the approximate result can be computed by the function estimate based on *unistats* (line 7). Afterwards, R_x invokes isAccuracy to conduct the accuracy estimation based on the parameters in *eInfo* such as the aggregate column and aggregate type, confidence and error rate. Based on the result of isAccuracy, R_x collects the output as the pair of (*key*, ⟨*result*, "*accept*"⟩) if *result* satisfies the users expectation, otherwise the output is constructed as (*key*, *result*) (lines 8∼11).

4.2.2 Queries for Multi-Relations

For queries involving multi-relations, there are two approaches that can be applied. We can precompute the join result and store such result as a regular file in HDFS, then adopt the method used in the single relation queries to estimate the final result. Or alternatively, we can get samples from each relation and calculate the estimates in the fly by one MapReduce job. Note that, the first approach trades storage for accuracy, while the second approach sacrifices some accuracy (also satisfies the accuracy requirement) for flexibility. In our OLACloud, we implement both approaches and configure the second one as the default approach.

The map logic of the default approach is similar to Algorithm 7, but it has some differences that: 1) Such map logic does not need to calculate the statistics of random samples from each mapper since the statistics are related to the samples from both two relations. And the task of this map logic is only to collect unbiased samples and assign them to the corresponding reducer for further processing. 2) Such map logic needs to redesign the structure of output key/value pair as $(\langle Q_x.qID, rTag\rangle, samples)$ to satisfy the requirement of shared sampling, where *rTag* indicates the relation that *samples* are drawn from. Note that, the output with the same *qID* would be delivered to the same reducer. And we modify the default GroupingComparator class to make sure that the input from both relations are in the same group, which is convenient for the statistics calculation in the reduce logic. Algorithm 9 shows the reduce logic for multi-relations query.

There are several parameters defined in Algorithm 9, e.g., *rPre* and *sPre* indicates the samples have been received from the relation R and S respectively, while *rCur* and *sCur* from the relation R and S respectively denotes as the incoming samples need to be processed. Firstly, each reducer must classify the incoming samples into *rCur* and *sCur* (lines 4∼10). And then, the partial statistics are computed as *statsPP*, *statsPC*,

Algorithm 9: Reduce Logic for Multi-Relations Query
Input: LongPair *key*, Iterator⟨Text⟩ *values*
Output: OutputCollector⟨IntWritable, DoubleTriple⟩ *outputKV*

```
1   static Vector rPre, sPre
2   Vector rCur, sCur;
3   EstimateInfo eInfo = DefaultStringifier.load(OLAjob,
    "eInfo", EstimateInfo.class);
4   while values.hasNext() do
5       if values.next.getTag == "R" then
6           rCur.add(values.next);
7       else
8           sCur.add(values.next);
9       end
10  end
11  if
    rPre.equals(null)&&sPre.equals(null)
    then
12      statsPP = getStats(rPre, sPre);
13  else
14      statsPC = getStats(rPre, sCur);
15      statsCP = getStats(rCur, sPre);
16      statsCC = getStats(rCur, sCur);
17  end
18  stats = merge(statsPP, statsPC, statsCP, statsCC);
19  statsPP = stats;
20  rPre = rPre.addAll(rCur);
21  sPre = sPre.addall(sCur);
22  result = estimate(stats);
23  if isAccuracy(result, eInfo, stats) then
24      qID = key.getFirst();
25      outputKV.collect(qID, ⟨result, "accept"⟩);
26  else
27      outputKV.collect(qID, result);
28  end
```

statsCP and *statsCC* (lines 11∼17), where *statsPP* indicates the statistics that calculated from *rPre* and *sPre*, and the other partial statistics have similar definitions. In order to calculate the partial estimate, we also need to combine these partial statistics together to form a final statistic denoted as *stats*, which is used to the approximate result calculation and accuracy estimation (lines 18∼22). Finally, each reducer collects the output as the pair of (*qID*, ⟨*result*, "*accept*"⟩) if the *result* satisfies the users expectation.

4.3 Statistical Estimation

The goal of online aggregation is to provide efficient, accurate estimates and their confidence intervals which are updated regularly. Let ε', c be the running error bound and confidence respectively. ε' and c give the probabilistic estimate of approximate result v' which means exact result v lies in the interval $[v'-\varepsilon', v'+\varepsilon']$ with probability c. We can say the approximate result achieves the user expectation, if $\varepsilon' \leqslant v' \times e$ (e is the predefined error rate). Note that, the confidence c and error rate e are the parameters of *eInfo* in Algorithm 6 and v' is the approximate *result* calculated by the function `estimate` in Algorithm 8 and Algorithm 9.

In this subsection, we take SUM, COUNT, AVG as examples to show how estimates and confidence intervals for the single relation query can be obtained in our OLACloud. Consider a typical single relation query Q_x such as:

SELECT *op*(*expression*) FROM R WHERE *predicate*.

Given the block set $\mathcal{B}$, S indicates the sample set collected from the relevant block set $\{P_x \cap \mathcal{B}\}$ and each $B_i \in \{P_x \cap \mathcal{B}\}$ provides $|S_i|$ samples, where $\sum_{i=1}^{|P_x\cap\mathcal{B}|} |S_i| = |S|$. Each map task calculates *stats* according to the aggregate column in *eInfo*, including the sum and count of S_i' that is $sum(S_i') = \sum_{s_j \in S_i'} s_j$ and $count(S_i') = |S_i'|$, where $S_i' \subseteq S_i$ indicates the sample set that satisfies the query predicate. Then, the reduce task can calculate the value of $\sum_{s_i \in S} \exp_p(s_i)$ as follows, which is part of the parameters in *unistats*.

$$\sum_{s_j \in S} \exp_p(s_j) = \begin{cases} \sum\limits_{i=1}^{|P_x\cap\mathcal{B}|} sum(S_i'), & \text{if } op = sum, \\ \sum\limits_{i=1}^{|P_x\cap\mathcal{B}|} count(S_i'), & \text{if } op = count. \end{cases}$$

Then the estimated aggregate *result* can be calculated in the reduce task as follows (processed by `estimate` function in Algorithm 8).

$$v'_{s|c} = \frac{T}{|S|} \times \sum_{s_j \in S} \exp_p(s_j), \quad v'_a = \frac{v'_s}{v'_c}, \tag{1}$$

where the variable $T = \sum_{B_i \in \{P_x\cap\mathcal{B}\}} |B_i|$ indicates the total number of tuples in the relevant block set of Q_x, and $\exp_p(s_i)$ equals s_i for SUM and 1 for COUNT if s_i satisfies the predicate, and 0 otherwise.

Besides the estimated aggregate results, the reduce task also needs the corresponding variances of these results to calculate the error bound for accuracy estimation. To simplify the computation of variance, we apply the computational formula of variance.

$$\sigma^2(X) = E(X^2) - E(X)^2.$$

Each grouped map task M_x^i maintains the quadratic sum of samples $X_i^2 = \sum_{s_j \in S_i'} s_j^2$ in *stats* too, which is used to calculate the variances. Reduce task collects

all X_i^2 and calculates $\sum_{i=1}^{|P_x \cap \mathcal{B}|} X_i^2$, which is the other parameter in *unistats*. In the function `isAccuracy`, the input parameter of *result* and *unistats* are used to calculate the variance firstly according to the aggregate type in *eInfo* as follows.

$$\sigma_a^2 = \frac{\sum_{i=1}^{|P_x \cap \mathcal{B}|} X_i^2}{\sum_{i=1}^{|P_x \cap \mathcal{B}|} count(S_i')} - (v_a')^2,$$

$$\sigma_s^2 = \frac{\sum_{i=1}^{|P_x \cap \mathcal{B}|} X_i^2}{|S|} \times T^2 - (v_s')^2,$$

$$\sigma_c^2 = \frac{\sum_{i=1}^{|P_x \cap \mathcal{B}|} count(S_i')}{|S|} \times T^2 - (v_c')^2.$$

Afterwards, the function isAccuracy needs to compute the error bound ε' based on the variance above and the input parameter of *eInfo*. Based on Central Limit Theorem, $\frac{\sqrt{|S|} \times (v'-v)}{\sigma}$ is distributed approximately as a standardized normal distribution when $|S|$ is "large enough", where $\frac{\sigma^2}{|S|}$ is the variance of v'. Given a predefined confidence c, ε' can be computed by the following formula:

$$P\{|v'-v| \leqslant \varepsilon'\} \approx 2\Phi\Big(\frac{\varepsilon'\sqrt{|S|}}{\sigma}\Big) - 1, \tag{2}$$

where $P\{|v'-v| \leqslant \varepsilon'\}$ is the predefined confidence c. If $\varepsilon' \leqslant v' \times e$, then we can say *result* is acceptable to the user.

For the case of the multi-relations query such as:

SELECT *op*(*expression*) FROM R, S WHERE *predicate*,

given the block sets $\mathcal{B}_r$, $\mathcal{B}_s$ for relation R and S respectively, $S_r = rPre$ ($S_s = sPre$) indicates the sample set that has been collected from the relevant block set $\{P_x \cap \mathcal{B}_r\}$ ($\{P_x \cap \mathcal{B}_s\}$). Each reduce task needs to calculate the partial statistics such as *statsCC*. For example, *statsCC* includes the sum and count of S_{cc}' that is $sum(S_{cc}') = \sum_{s_i \in S_{cc}'} s_i$ and $count(S_{cc}') = |S_{cc}'|$, where $S_{cc}' \subseteq S_{cc}$ indicates the sample set that satisfies the query predicate and $S_{cc} = rCur \times sCur$. Then, the reduce task can calculate the value of $\sum_{s_i \in S_r \times S_s} \exp_p(s_i)$ by adding these four partial statistics together, which is part of the parameters in *stats*. Then the estimated aggregate *result* can be calculated by (1) with T and $|S|$ being replaced by $T_r \times T_s$ and $|S_r| \times |S_s|$ respectively, where $T_r = \sum_{B_i \in \{P_x \cap \mathcal{B}_r\}} |B_i|$ and $|S_{\text{join}}| = |S_r| \times |S_s|$.

Moreover, the partial statistics such as *statsCC* of each reducer also maintains the quadratic sum of samples denoted as $X_{cc}^2 = \sum_{s_i \in S_{cc}'} s_i^2$. Then the reducer can calculate the value of X_{join}^2 by adding all the quadratic sums from each of the partial statistics. Therefore, the variance of samples for each relation can be computed by (3)∼(5), and the variance of the joined samples can be calculated by (6).

$$\sigma_a^2(R) = \sigma_a^2(S) = \frac{X_{\text{join}}^2}{|S_{\text{join}}'|} - (v_a')^2, \tag{3}$$

$$\sigma_s^2(R) = \sigma_s^2(S) = \frac{X_{\text{join}}^2}{|S_r| \times |S_s|} \times (T_r \times T_s)^2 - (v_s')^2, \tag{4}$$

$$\sigma_c^2(R) = \sigma_c^2(S) = \frac{S_{\text{join}}'}{|S_r| \times |S_s|} \times (T_r \times T_s)^2 - (v_c')^2, \tag{5}$$

$$\sigma_{\text{join}}^2 = \frac{\sigma^2(R)}{|S_r|} + \frac{\sigma^2(S)}{|S_s|}, \tag{6}$$

where S_{join}' indicates the sample set that satisfies the query predicate and we have $S_{\text{join}}' = S_{pp}' + S_{pc}' + S_{cp}' + S_{cc}'$. The definitions of S_{pp}', S_{pc}', S_{cp}' are similar to that of S_{cc}', and $S_{pp} = rPre \times sPre$, $S_{pc} = rPre \times sCur$, $S_{cp} = rCur \times sPre$. Finally, each reducer invokes the function `isAccuracy` to calculate the error bound ε' by (2) based on the variance above and the input parameter of *eInfo*.

5 OLACloud Implementation in Hadoop-Hop

We have used Hadoop-hop-0.2, which is currently based on Hadoop-0.19.2, to implement our OLACloud prototype and run experiments on a virtual cluster with 31 nodes (2 VCPU + 10 GB of main memory + 100 GB disks for each node) from SEUCloud (Southeast University Cloud Platform), which supports data processing applications of the whole university, such as AMS (Alpha Magnetic Spectrometer) experiment. SEUCloud consists of a compute system and a storage system, each of these compute and storage nodes has a separate 1 Gb/s Ethernet and 40 Gb/s Infiniband link respectively to different switches. The compute system contains 252 IBM H22 blade servers, 8 IBM X3850 X5 4-Way SMP servers and 2 IBM X3850 X5 8-Way SMP servers. While the storage is set up by 16 IBM X3650 M3 servers which attach an IBM DS5300 storage array via 8 Gpbs fiber channels.

Fig.8 illustrates the architecture of OLACloud. The system consists of one master node and many slave nodes. A client submits jobs to the master node, where the JobTracker daemon is initialized to manage the lifecycle of these jobs. For each submitted job, the master node schedules a number of parallel tasks to run on slave nodes by Task Scheduler. Every slave node has a TaskTracker daemon process to communicate with the master node and manage each task's execution. In order to accommodate the requirements of online aggregation applications, we make several changes to the basic Hadoop MapReduce framework. First, OLACloud

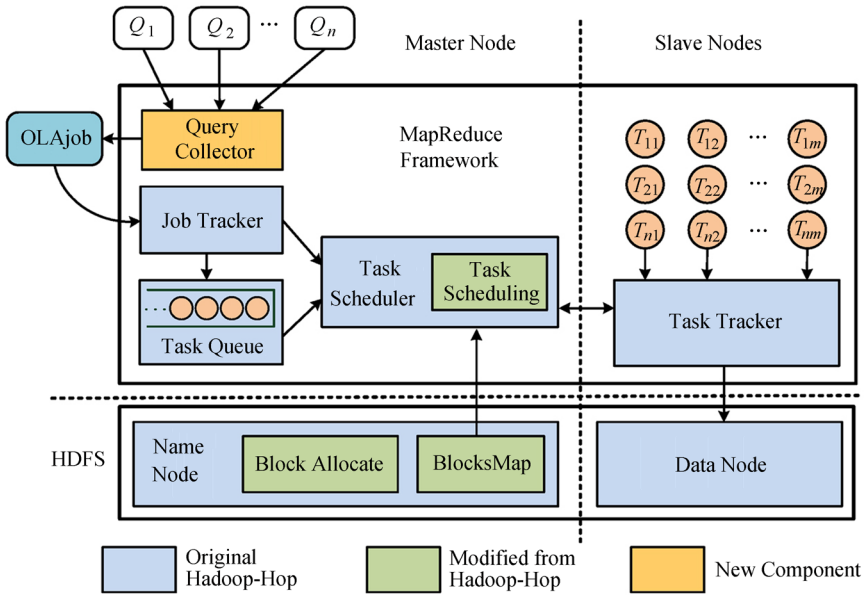


Fig.8. OLACloud framework.

assigns the blocks based on the fair-allocation strategy which is implemented by modifying the Block Allocate module. And we also build our NRB-T index in DataNode to support the relevant block determination. Second, OLACloud uses BlockMaps to maintain the map from a block to its metadata, which is extended to include the data node that stores the primary replica. Third, OLACloud adopts a modified version of the delay scheduling[20], which prefers to schedule the task to the data node with the primary replica, to improve the performance of fair-allocation. Forth, OLACloud contains a new module called Query Collector that collects the incoming queries in a system buffer, then OLAjob can combine *threshold* queries into a shared MapReduce job for processing.

6 Experimental Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness and study the performance characteristics of our OLACloud under different degrees of skew in the input data and under different data sizes. A modified TPC-H toolkit[21] is employed to generate skewed datasets with Zipf distribution, which is determined by the Zipf parameter z (z varies over 0, 1.2, 1.6, where 0 represents the uniform distribution), derived from LINEITEM and ORDER tables as our test data. The scale factor is varied over 10, 20, 40, 100. Table 2 summarizes the properties of the generated datasets.

In our experiments, we generate queries based on the Single Table Template (T_1) and Multi-Table Template (T_2) as follows.

Table 2. Properties of Datasets

Scale	Size (GB)	Number of Rows (Million)
10	7.8	66
20	15.8	132
40	37.6	264
100	78.7	660

T_1: SELECT sum(C_i)|count(C_i)|avg(C_i) FROM LINEITEM
WHERE [$l\_discount > x$ and $l\_discount < x + y$] |
[$l\_quantity > x$ and $l\_quantity < x + y$] |
[$l\_extendedprice > x$ and $l\_extednedprice < x + y$],

T_2: SELECT sum(C_i)|count(C_i)|avg(C_i) FROM LINEITEM L, ORDERS O
WHERE $L.orderkey = O.orderkey$ &
[$l\_discount > x$ and $l\_discount < x + y$] |
[$l\_quantity > x$ and $l\_quantity < x + y$] |
[$l\_extendedprice > x$ and $l\_extednedprice < x + y$]|
[$o\_totalprice > x$ and $o\_totalprice < x + y$].

Note that, parameter x is some random value that belongs to the value range of C_i and parameter y varies from 10% to 90% of the value range of C_i for each x.

The aggregate type (AVG, COUNT, SUM) for each query is randomly selected during the query generation process. Each experiment executes 100 queries for 5 times to remove any side effect. The default error rate e and confidence c used for the accuracy estimation are 0.01 and 95% respectively; the default partition size (ps) and query collection threshold (qct) are 4 and 20 respectively. The default parameters of fair-allocation

are set as $w_s = w_c = 0.5$, which means the computation and storage load balancing have equal importance.

For comparison purposes, we implemented six OLA methods for different partition manners, block allocation strategies and sampling methods. We adopt four blocks placement strategies, that is random (RM), round-robin (RR), range (RG) and fair-allocation (FA). In the case of size-aware partition manner, we implement one method that allocates the blocks in the random strategy and draws the samples without sharing, called size-RM. For the content-aware partition manner, we partition the LINEITEM and ORDER table based on the columns in "WHERE" predicates (the default partition size for each partitioning column is 4) and implement four methods for random, round-robin, range and fair-allocation respectively, denoted by content-RM, content-RR, content-RG and content-FA. Moreover, content-FA-share denotes the fair assignment strategy with shared sampling.

6.1 Effect of Data Size and Distribution

In this experiment, we vary the data size from 10 G to 100 G and evaluate the performance of four basic methods for different data distributions, which include Hadoop-complete, size-RM, content-FA and content-FA-share. Note that, Hadoop-Complete denotes a method which returns the precise results. We set c, e, ps and qct be the default values, and $w_s = w_c = 0.5$.

Figs. 9(a), 9(c), 9(e) represent the result of template 1, while Figs. 9(b), 9(d), 9(f) represent the result of template 2. We note that both results show a similar trend, but the joined one takes much more time to process since there are more tasks to be initialized in the map phase and the join operation in the reduce phase takes extra more time. Figs. 9(a)∼9(f) indicate that the performance of Hadoop-complete decreases with the increase of data size, but the OLA-based methods are scalable with regard to the data size. And the time cost is much lower than Hadoop-complete. This is because the OLA-based methods only need a small sample set to calculate the approximate results rather than all of the dataset. Among the OLA-based methods, content-FA performs better than size-RM. This is expected as the sampling efficiency is improved by our content-aware repartition method, leading to the early acceptable results. And we also note that the performance of shared method content-FA-share is better than the one with-

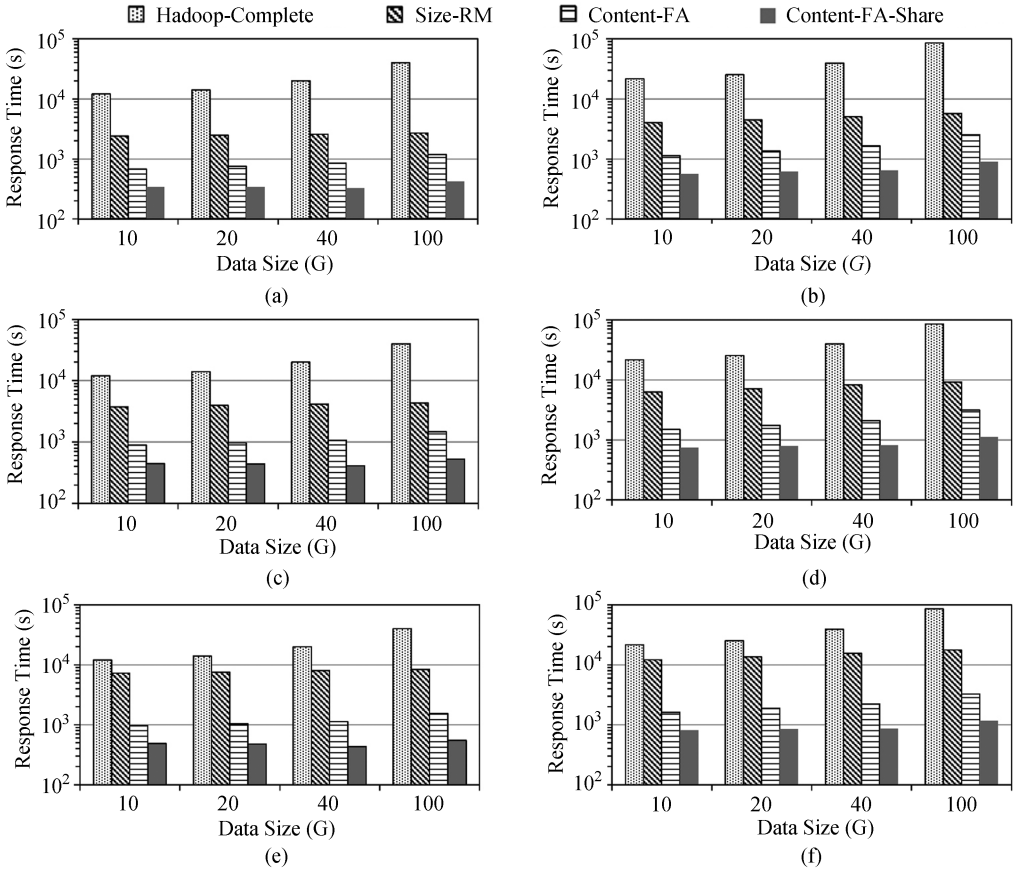


Fig.9. Effect of data size. (a) Uniform T_1. (b) Uniform T_2. (c) Zipf-1.2 T_1. (d) Zipf-1.2 T_2. (e) Zipf-1.6 T_1. (f) Zipf-1.6 T_2.

out sharing due to the redundant disk I/O can be significantly decreased by our shared sampling strategy.

On the other hand, in order to study the effect of data distribution in detail, we compare the performance of a 100 G dataset for different data distributions in Fig.10. Note that, besides the Hadoop-complete method, the content-aware methods are also scalable with regard to the data distribution. While the performance of size-RM is significantly decreased along with the much more skewed data distribution.

6.2 Effect of Block Placement Strategies

In this experiment, we evaluate the performance of content-aware online aggregation methods for different block placement strategies, that are random, range, round-robin and fair-allocation. All the parameters are set to be the default values.

As shown in Figs. 11(a) and 11(b), content-FA outperforms the content-RM, content-RG and content-RR methods. This is because of the fair-allocation strategy can guarantee a good computation load balancing. But the performance improvement of template 2 is not so obviously since our fair-assignment strategy does not consider the block correlation between different relations. Figs. 11(c) and 11(d) illustrate the load distribution in the storage and computation against the Zipf-1.6 dataset for these content-aware methods, and the result of other data distribution has the same trend since the load balancing is actually determined by the block placement strategy rather than data distribution.

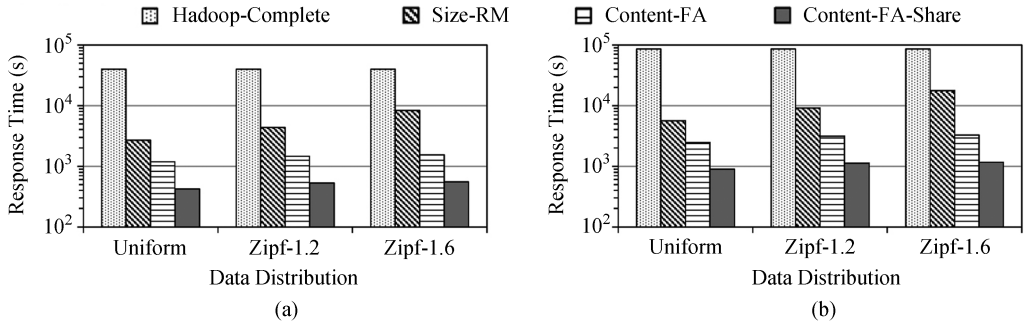


Fig.10. Effect of data distribution. (a) 100 G T_1. (b) 100 G T_2.

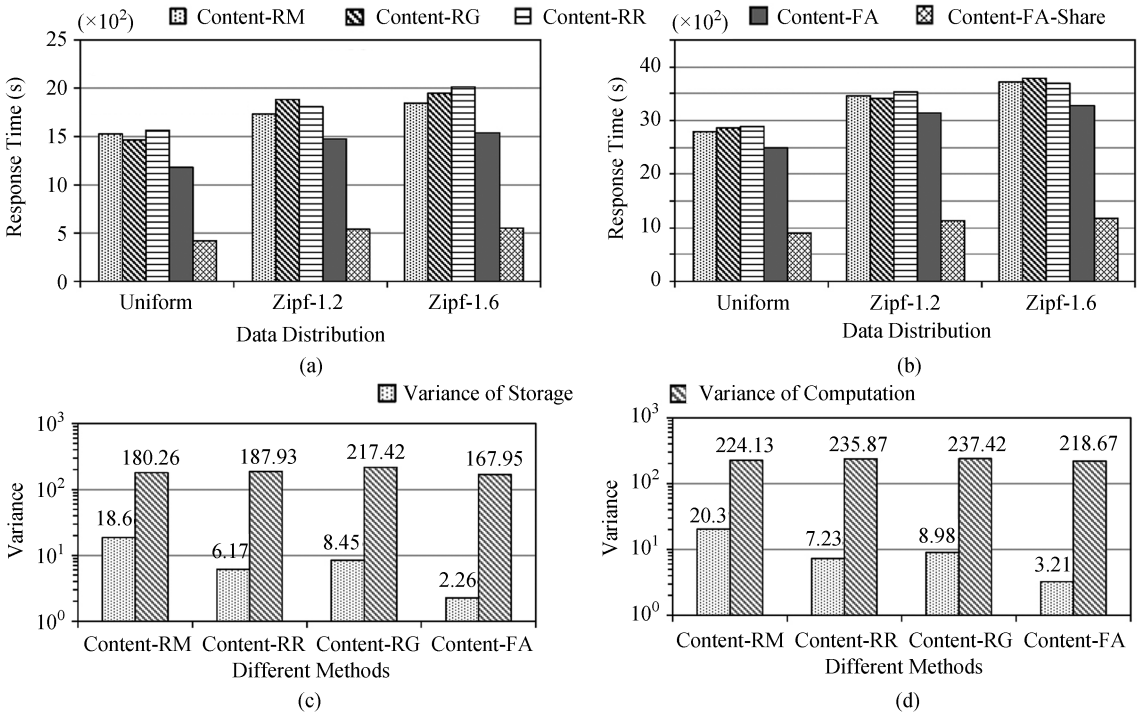


Fig.11. Comparison of different partition manners and placement strategies. (a) Comparison for response time (T_1). (b) Comparison for response time (T_2). (c) Comparison for load distribution (T_1). (d) Comparison for load distribution (T_2).

We use the variance of storage consumption (storage consumption in each data node is measured in GB), and the variance of number of queries processed in each node as the metrics. Note that, our content-FA strategy can effectively balance both the storage and computation load among data nodes.

On the other hand, the parameters w_s and w_c are used to adjust the preference between storage balancing and computation balancing, so that we show the effect of these two parameters in Figs. 12(a) and 12(b). Note that, the larger w_c indicates more computation load balancing and the response time is decreased along with the increase of w_c. While larger w_s indicates higher storage load balancing but with poorer performance. Therefore, we should choose appropriate w_s and w_c in the actual configuration of OLACloud to guarantee the storage and computation load balancing by fully considering the real system storage and computational capability.

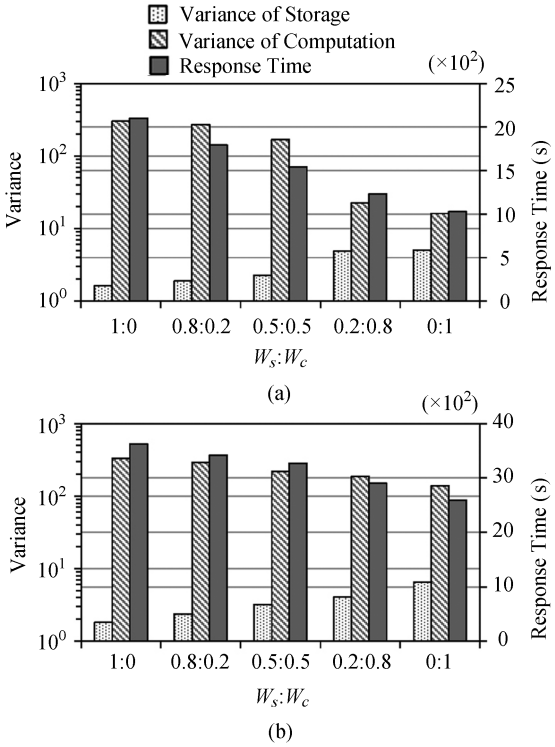


Fig.12. Effect of w_s and w_c. (a) Result for T_1. (b) Result for T_2.

6.3 Effect of Partition Size

In this test, we show the effect of partition size for content-FA. To facilitate the discussion, each column has the same partition size (ps) which varies from 3 to 8.

Through Table 3 and Table 4, we note that the results for different data distributions show the similar trend, that is the response time reduces with the increase of ps at first when ps is relatively small, but this performance improvement disappears and the response time starts growing with the continued increase of ps. This is expected as the sampling efficiency is the major factor that affects the performance at first, so that when we increase the value of ps, the query can prune much more unneeded data due to the fine-grained partition, improving the sampling efficiency. However, larger ps indicates more blocks, which also means that more query tasks will be initialized for processing. Take a single partitioning column with the range of $[1, 100]$ as an example, the partitioning column is divided as $\{[1, 50], [51, 100]\}$ for $ps = 2$ and $\{[1, 25], [26, 50], [51, 75], [76, 100]\}$ for $ps = 4$ respectively, and given the query predicate is $23 \leqslant attribute \leqslant 85$, then the number of query tasks for each ps is 2 and 4 respectively. More query tasks lead to more system overhead such as the task initialization time, which eliminates the positive effect of larger ps.

Table 3. Effect of Partition Size (T_1)

Partition Size (ps)	Response Time (s)		
	Uniform	Zipf-1.2	Zipf-1.6
3	1 449.8	2 142.3	2 933.2
4	1 178.7	1 477.4	1 543.8
5	1 296.6	1 403.5	1 494.4
6	1 485.2	1 625.1	1 420.3
7	1 744.6	1 950.2	1 883.4
8	1 909.6	2 157.1	2 192.2

Table 4. Effect of Partition Size (T_2)

Partition Size (ps)	Response Time (s)		
	Uniform	Zipf-1.2	Zipf-1.6
3	2 938.1	4 229.8	9 345.2
4	2 497.4	3 130.1	3 270.8
5	2 837.9	3 516.9	3 074.5
6	3 299.9	4 017.5	3 497.7
7	3 928.5	4 617.8	4 214.1
8	4 733.1	5 357.1	5 504.4

On the other hand, we note that the performance turning point for different data distributions is different. For Zipf-1.6 of template 1, the response time continues to reduce until ps equals 6, and the value for Zipf-1.2 and uniform is 5 and 4 respectively. This can be explained as that the effect of low sampling efficiency is more crucial for much more skewed data distribution, and more skewed data distribution indicates that we need larger ps to prune much more unneeded data. Through this experiment, we find that much larger ps is not always optimal, and we should choose appropriate ps by taking account of the additional system overhead caused by large ps.

6.4 Effect of Query Collection Threshold

In this test, we study the effect of query collection threshold (*qct*), which is an important parameter that controls the degree of shared sampling and decides the performance of content-FA-share. We vary *qct* from 10 to 100 and Fig.13 demonstrates the results of different data distributions with the default parameters.

As shown in the figure, three curves have the similar trend that is the performance of content-FA-share is improved at first until *qct* reaches a specific value, then the performance is degraded along with the increase of *qct*. This can be explained as our shared sampling strategy affects the overall performance from three aspects, including two positive cases as well as one negative case, that is: 1) reducing the redundant disk I/O cost, 2) reducing the initialization time of the query tasks, and 3) extending the execution time of each query task. For the case of smaller *qct*, the performance improvement caused by 1) and 2) is larger than the degradation results from 3), while this situation is reversed for the case of larger *qct*.

On the other hand, the optimal *qct* for uniform case is larger than the optimal value for skewed cases. Note that, the block size in skewed case is relatively smaller than the uniform one (since the high frequency tuples often appear in the minority of blocks due to the characteristic of Zipf distribution), leading to a relatively lower sampling cost. Therefore, the performance improvement of 1) for skewed data distribution is not so obviously as the uniform case, resulting an early performance degradation.

6.5 Effect of Error Rate and Confidence

Given a predefined confidence, smaller error rate indicates more approximate result we can obtain. And larger confidence gives higher probability that the accurate result is bounded by the estimated error bound.

In this experiment, we vary the predefined error rate and confidence respectively to examine the performance of the three methods, that is content-FA-share, content-FA and size-RM. The predefined error rate ranges from 0.01 to 0.05. Figs. 14(a) and 14(b) show all methods have the similar trend that much more samples are needed to gain the higher precision for smaller predefined error rate, which leads to longer processing time. On the other hand, the predefined confidence ranges from 82% to 98% to show the effect of confidence. Figs. 14(c) and 14(d) show that along with the increase of confidence, much more samples are received to update the estimators (explained by (13)), which leads to a longer processing time. Moreover, we note that the performance curve of our content-aware methods for both templates in all figures are relatively smooth and scalable with skewed data distribution.

6.6 Accuracy of Estimation

In this test, we only present the result of content-FA for template 1 and zipf1.6 as the results of two templates for different data distributions show similar trend due to the CLT is used in estimation. We use the average real error rate of queries as the metric to show the effect of error rate and confidence on the accuracy of content-FA (real error rate is calculated as $\frac{|v-v'|}{v}$, and v is computed by Hadoop-complete). We vary the predefined confidence from 82% to 98% and set the error rate to 0.01. The result depicted in Fig.15(a) shows that the average real error rate is always lower than the predefined error rate, which means most of queries have gained a good approximate result. In addition, the average real error rate also decreases with the increase of confidence. This is expected as higher confidence leads to more samples are received to gain a better estimation. Moreover, we vary the predefined error rate from 0.01 to 0.05 and set the confidence to 95%. As shown in Fig.15(b), the estimation is quite accurate because the average error rate is always lower than the predefined error rate (the real error rate is under the baseline).

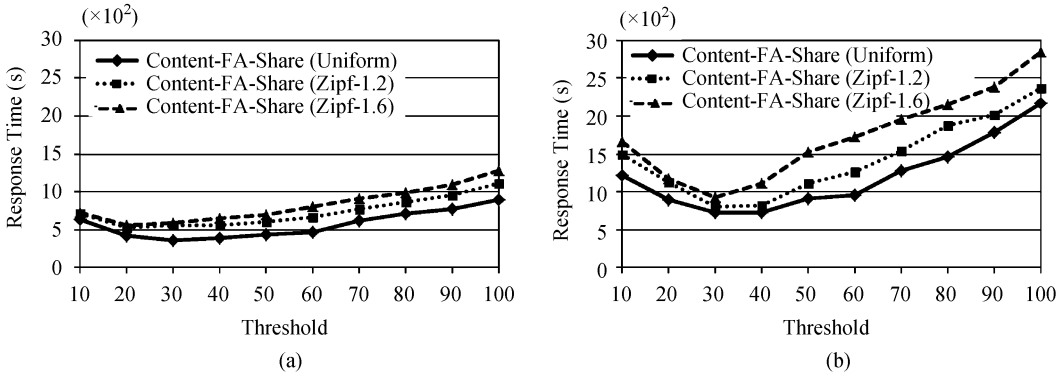


Fig.13. Effect of query collection threshold. (a) Result for T_1. (b) Result for T_2.

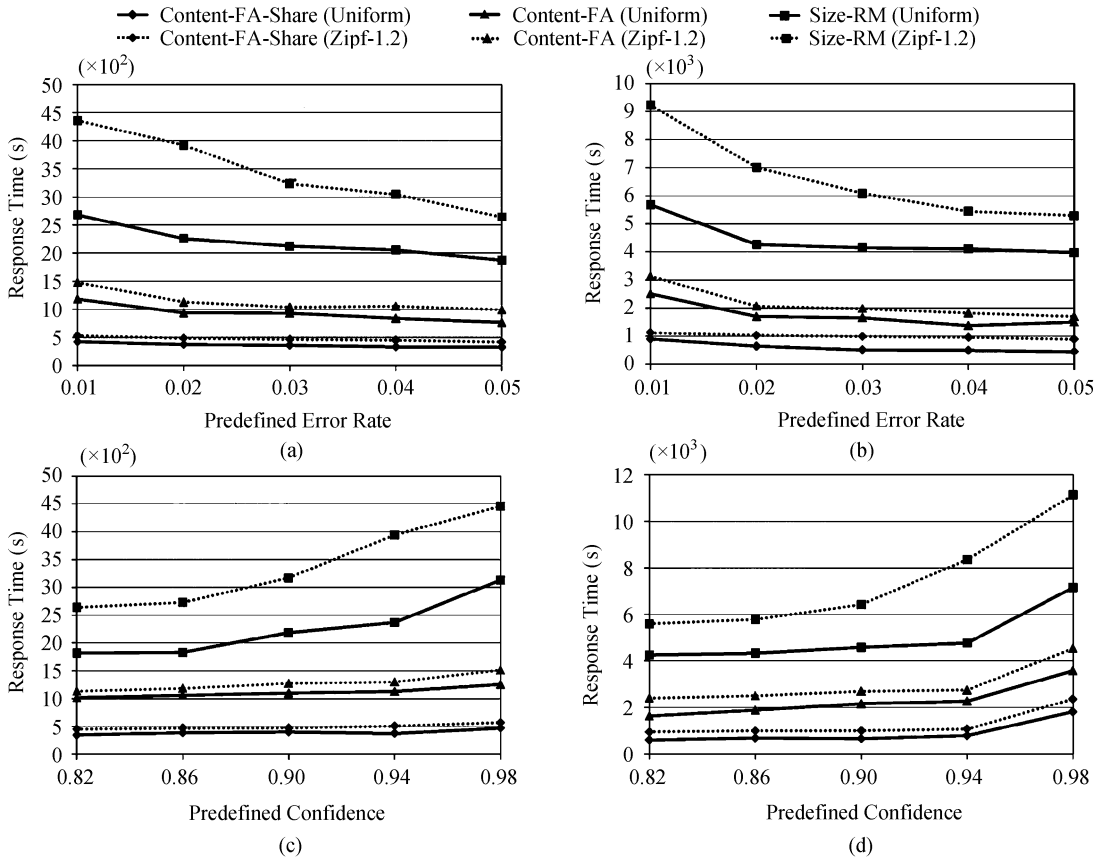


Fig.14. Effect of error rate and confidence. (a) Effect of error rate (T_1). (b) Effect of error rate (T_2). (c) Effect of confidence (T_1). (d) Effect of confidence (T_2).

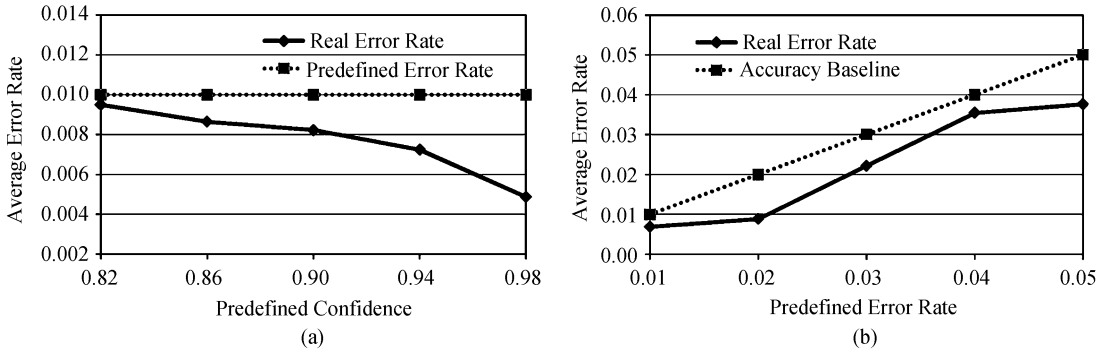


Fig.15. Accuracy of estimation. (a) Accuracy for different confidences. (b) Accuracy for different error rates.

7 Related Work

In many real applications, such as OLAP, aggregation queries are used widely and frequently. However, calculating exact results for these queries incurs long response time, and is not always required.

To response queries in a short processing time with acceptable results, online aggregation was first proposed in [5] to provide a time-accuracy tradeoff for aggregation queries. The approximate answer within a running confidence interval is produced during the early stages of query processing and gradually refined until it satisfies the users expectation. The running confidence interval tells users the estimated proximity of each running aggregation query to its final result. In [22], Haas illustrated how the central limit theorems, simple bounding arguments and the delta method can be used to derive formulas for both large-sample and deterministic confidence intervals. To support join operation for online aggregation, Hass and Hellerstein in-

troduced a novel join methods called ripple joins in [23]. But the convergence of ripple joins may be slow when memory overflows. To handle this problem, hash ripple join algorithm was proposed in [24], which combines parallelism with sampling to speed convergence and also maintains good performance in the presence of memory overflow. However, all researches in [5, 22-24] are focused on single query processing rather than multi query optimization. Therefore, Wu *et al.* proposed a new online aggregation system called COSMOS to process multiple aggregation queries efficiently[2]. COSMOS organizes queries into a dissemination graph to exploit the dependencies across queries, and the partial answers can be reused by the linked queries. In addition, Wang *et al.* present a partition-based online aggregation called POAS[13] to overcome the side effect of skewed data distribution and further improve the query performance.

In fact, these centralized online aggregation methods or systems cannot be extended to distributed manner easily; therefore the well designed distributed online aggregation systems were proposed along with the development of P2P and cloud computing[8−11,19]. Wu *et al.* extended the online aggregation to a P2P context where sites are maintained in a DHT network[19], which maintains synopses that can be reused by different queries. However, this P2P-based distributed online aggregation will transfer all the samples among processing nodes to guarantee the load balancing, generating a lot of network traffic. In addition, [9-10] demonstrate a modified version of Hadoop MapReduce framework that supports online aggregation, but they only implement the lightweight one, which returns the query progress without any precision estimation. [8] proposes a new online aggregation system that supports MapReduce job based on the open source project Hyracks[7], which discusses a Bayesian framework for producing estimates and confidence intervals for online aggregation. Although this method can allow users to see how accuracy of the result to the real final result, it uses a complex estimation method, which is hard to be implemented in the MapReduce framework, and the additional estimation module would add significant accidental complexity, restricting the overall performance. The authors of [11] formulated a statistical foundation that supports block-level sampling for single-table online aggregation and develops a two-phase stratified sampling method to support multi-table online aggregation.

However, there are several limitations that restrict the performance of online aggregation due to the gap between the current general mechanism of cloud framework and the requirements of OLA, and none of the above papers focuses on such problem to improve the performance of OLA in the cloud that we have discussed in this paper. In order to support the actual online aggregation rather than the query progress and stimulate the potential of OLA in the cloud to improve the estimate performance without large accidental complexity, we studied how to design and implement a parallel processing model to serve as a target for running online aggregation in cloud.

8 Conclusions

In this paper, we proposed a processing model called OLACloud with a fine granularity interactive mechanism, in which a content-aware partition manner is exploited to increase the computation resource utilization. A fair-allocation block placement strategy was adopted to guarantee the storage and computation load balancing and a share sampling method was used to reduce the redundant disk I/O cost. Finally, we evaluated OLACloud on the TPC-H benchmark for skew data distribution. The results demonstrate the efficiency and effectiveness of our approach. In our future work, we plan to expand our shared sampling strategy to support the computation sharing, which is another sharing opportunity to further improve the online aggregation performance by reducing the redundant statistical computation cost.

References

[1] Herodotou H, Lim H, Luo G *et al.* Starfish: A self-tuning system for big data analytics. In *Proc. the 15th CIDR*, Apr. 2011, pp.261-272.

[2] Wu S, Ooi B C, Tan K L. Continuous sampling for online aggregation over multiple queries. In *Proc. the 2010 International Conference on Management of Data* (*SIGMOD*), June 2010, pp.651-662.

[3] Chaudhuri S, Das G, Datar M *et al.* Overcoming limitations of sampling for aggregation queries. In *Proc. the 17th Int. Conf. Data Engineering*, Apr. 2001, pp.534-544.

[4] Laptev N, Zeng K, Zaniolo C. Early accurate results for advanced analytics on MapReduce. *PVLDB*, 2012, 5(10): 1028-1039.

[5] Hellerstein J M, Haas P J, Wang H J. Online aggregation. *ACM SIGMOD Record.*, 1997, 26(2): 171-182.

[6] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008, 51(1): 107-113.

[7] Borkar V, Carey M, Grover R *et al.* Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. the 27th International Conference on Data Engineering*, Apr. 2011, pp.1151-1162.

[8] Pansare N, Borkar V R, Jermaine C *et al.* Online aggregation for large MapReduce jobs. *PVLDB*, 2011, 4(11): 1135-1145.

[9] Böse J H, Andrzejak A, Högqvist M. Beyond online aggregation: Parallel and incremental data mining with online mapreduce. In *Proc. MDAC*, Apr. 2010, Article No.3.

[10] Condie T, Conway N, Alvaro P *et al.* Online aggregation and continuous query support in MapReduce. In *Proc. the 2010 International Conference on Management of Data*, June 2010, pp.1115-1118.

[11] Shi Y, Meng X, Wang F *et al.* You can stop early with COLA: Online processing of aggregate queries in the cloud. In *Proc. the 21st ACM International Conference on Information and Knowledgy Management*, Oct. 29-Nov. 2, 2012, pp.1223-1232.

[12] Grover R, Carey M J. Extending MapReduce for efficient predicate-based sampling. In *Proc. the 28th International Conference on Data Engineering*, Apr. 2012, pp.486-497.

[13] Wang Y, Luo J, Song A, Jin J H, Dong F. Improving online aggregation performance for skewed data distribution. In *Proc. Database Systems for Advanced Applications*, Apr. 2012, pp.18-32.

[14] Chaudhuri S, Das G, Srivastava U. Effective use of block-level sampling in statistics estimation. In *Proc. the 2004 International Conference on Management of Data*, June 2004, pp.287-298.

[15] Jacobs A. The pathologies of big data. *Communications of the ACM*, 2009, 52(8): 36-44.

[16] Soroush E, Balazinska M, Wang D. Arraystore: A storage manager for complex parallel array processing. In *Proc. the 2011 International Conference on Management of Data*, June 2011, pp.253-264.

[17] Eltabakh M Y, Tian Y, Ozcan F *et al.* CoHadoop: Flexible data placement and its exploitation in Hadoop. *PVLDB*, 2011, 4(9): 575-585.

[18] Nykiel T, Potamias M, Mishra C *et al.* MRShare: Sharing across multiple queries in MapReduce. *PVLDB*, 2010, 3(1/2): 494-505.

[19] Wu S, Jiang S, Ooi B C *et al.* Distributed online aggregations. *PVLDB*, 2009, 2(1): 443-454.

[20] Zaharia M, Borthakur D, Sen Sarma J *et al.* Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. the 5th European Conference on Computer System*, Apr. 2010, pp.265-278.

[21] Chaudhuri S, Narasyrra V. Program for tpc-d data generation with skew. Technical Report, ftp://ftp.research.microsoft.com/pub/user./viveknar/tpcdskew, Dec. 2012.

[22] Haas P J. Large-sample and deterministic confidence intervals for online aggregation. In *Proc. the 9th International Conference on Scientific and Statistical Database Management*, Aug. 1997, pp.51-62.

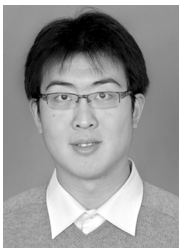
[23] Haas P J, Hellerstein J M. Ripple joins for online aggregation. *ACM SIGMOD Record*, 1999, 28(2): 287-298.

[24] Luo G, Ellmann C J, Haas P J *et al.* A scalable hash ripple join algorithm. In *Proc. the 2002 International Conference on Management of Data*, June 2002, pp.252-262.

Yu-Xiang Wang received the B.S. degree in software engineering from Tianjin University, China, in 2008. He is currently a Ph.D. student in the School of Computer Science and Engineering, Southeast University, Nanjing. His current research interests include cloud computing, big data processing and query optimization.

Jun-Zhou Luo received the M.S. and Ph.D. degrees in computer science from Southeast University, Nanjing, in 1992 and 2000, respectively. He is currently a professor and the dean of the School of Computer Science and Engineering, Southeast University. His current research interests include next generation network architecture, cloud computing, network security, and wireless network.

Ai-Bo Song received the M.S. degree from Shandong University of Science and Technology, Qingdao, the Ph.D. degree in computer application technology from Southeast University, Nanjing, in 1996 and 2003, respectively. He is currently an associate professor in the School of Computer Science and Engineering, Southeast University. His current research interests include cloud computing, big data processing, and grid computing.



Fang Dong received the B.S. and M.S. degrees in computer science from Nanjing University of Science and Technology, China, in 2004 and 2006, respectively; and received his Ph.D. degree in computer science from Southeast University in 2011. He is currently a lecturer in School of Computer Science and Engineering, Southeast University. His current research interests include cloud computing, task scheduling, and big data processing.