This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

An I/O Efficient Distributed Approximation Framework Using Cluster Sampling

Xuhong Zhang, Jun Wang, Shouling Ji, Jiangling Yin, Rui Wang, Xiaobo Zhou and Changjun Jiang

Abstract—In this paper, we present an I/O efficient distributed approximation framework to support approximations on arbitrary sub-datasets of a large dataset. Due to the prohibitive storage overhead of caching offline samples for each sub-dataset, existing offline sample-based systems provide high accuracy results for only a limited number of sub-datasets, such as the popular ones. On the other hand, current online sample-based approximation systems, which generate samples at runtime, do not take into account the uneven storage distribution of a sub-dataset. They work well for uniform distribution of a sub-dataset while suffer low I/O efficiency and poor estimation accuracy on unevenly distributed sub-datasets.

To address the problem, we develop a distribution aware method called *CLAP* (cluster sampling based approximation). Our idea is to collect the occurrences of a sub-dataset at each logical partition of a dataset (storage distribution) in the distributed system, and make good use of such information to enable I/O efficient online sampling. There are three thrusts in CLAP. First, we develop a probabilistic map to reduce the exponential number of recorded sub-datasets to a linear one. Second, we apply the *cluster sampling with unequal probability theory* to implement a distribution-aware method for efficient online sampling for a single or multiple sub-datasets. Third, we enrich CLAP support with more complex approximations such as ratio and regression using bootstrap based estimation beyond the simple aggragation approxiamtions. Forth, we add an option in CLAP to allow users specifying a target error bound when submitting an approximation job. Fifth, we quantitatively derive the optimal sampling unit size in a distributed file system by associating it with approximation costs and accuracy. We have implemented CLAP into Hadoop as an example system and open sourced it on GitHub. Our comprehensive experimental results show that CLAP can achieve a speedup by up to 20$\times$ over the precise execution.

Index Terms—Approximation, Cluster sampling, Sub-dataset, Storage distribution, Hadoop.

1 Introduction

Advances in sensing, networking and storage technologies have led to the generation and collection of data at extremely high rates and volumes. Large corporations, such as Google, Amazon and Facebook produce and collect terabytes of data with respect to click stream or event logs in only a few hours [33, 30]. Despite the fact that today's warehouse-scale computers (*WSC* in brief) supply enormous data processing capacity, getting an *ad-hoc* query answer from a large scale dataset remains challenging. In addition, the energy cost involved in such WSC applications becomes non-negligible. To attack the problem, recent years have seen a trend to promote approximate computing in distributed big data analytic frameworks [16, 13, 4, 25, 18]. Approximate computing allows for faster execution on a much smaller sample of the original data by sacrificing accuracy to a reasonable extent. An approximation process often involves two basic phases: sample preparation and results estimation. Preparing representative samples from distributed storage systems efficiently is essential to approximation systems.

Creating a single offline uniform sample of the whole dataset to serve all possible queries is impractical. The reason is that most of user's queries are only interested in a subset data of the whole dataset and the single uniform sample may contain little data relevant to the user's desirable subset. In practice, users always specify a predicate or filter condition in their query, which makes the workload unpredictable. The predicate can be a particular location, topic, time, *etc*. We refer the collection of data related to certain events or features as a **sub-dataset**. Offline based sampling approaches such as [4] do not handle well with the dynamic workload. For example, creating these uniform samples compromises locality on physical storage, and renders disk random access to individual data records. This introduces huge I/O overhead. In addition, new data is continuously appended to the system. Unfortunately, updating the offline samples triggers a new scan of the whole dataset, which is prohibitive in a realistic operation.

To enable approximation queries on arbitrary sub-datasets, there is a need for developing an **online and I/O efficient sampling** method. The main challenge is to minimize the total size of accessed data and its associative I/O overhead subject to a given error bound. Cluster sampling with equal probability as demonstrated in ApproxHadoop [13] is one potential solution. It samples data by cluster — multiple consecutive stored data records. In this paper, we refer to the term **cluster** as **segment**. Equipped with a large segment size, the number of random I/O could be dramatically reduced, resulting in a low I/O overhead. Here segment is

- *X. Zhang, J. Wang, J. Yin, and R. Wang are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, 32816. E-mail: {xhzhang, jwang, jyin}@eecs.ucf.edu*
- *S. Ji is with the College of Computer Science, Zhejiang University, Hangzhou, China. E-mail: sji@zju.edu.cn*
- *X. Zhou is with the Department of Computer Science, University of Colorado, Colorado Springs, USA. E-mail: xzhou@uccs.edu*
- *C. Jiang is with Tongji University, Shanghai Shi 200000, China. E-mail: cjjiang@tongji.edu.cn*

Corresponding authors: X. Zhang, J. Wang {xhzhang, jwang}@eecs.ucf.edu

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

also viewed as data block in storage systems. If the sub-dataset is uniformly distributed over the whole dataset, sampling each segment with equal probability will suffice. Unfortunately, in many real-life cases, sub-datasets bear strong localities. They spread unevenly over the segments of a whole dataset, and concentrate on a few segments of the whole dataset. A common phenomenon [15, 24, 29] is that, a small portion of the whole dataset contains most records of this sub-dataset, while other portions host few records belonging to this sub-dataset, as illustrated in Figure 1. Without knowledge of this skewed storage distribution, systems like ApproxHadoop may sample a large portion of the whole dataset, but obtain little sample data belonging to the queried sub-dataset. On the other hand, even if we collect enough samples of a sub-dataset, it may produce a result with a large variance. The reason is explained in the beginning of Section 3. In summary, the system could suffer from *inefficient sampling and large variance*

Interestingly, we realize one untapped method which employs the sub-dataset distribution information to enforce I/O efficient sampling approximation. More specifically, we developed a distribution-aware online sampling system called CLAP. In CLAP, we developed a probabilistic map (SegMap) to capture the occurrences of a sub-dataset at each logical partition of a dataset. SegMap is able to reduce the number of sub-datasets to be recorded from a factor of 2^f to f, where f stands for the total number of columns in a table. CLAP samples at a configurable segment level in contrast to the traditional HDFS block level, as using a HDFS block as the default sampling unit is not always optimal. In CLAP, we quantify the optimal segment size by relating it to approximation accuracy and cost. When sampling segments for a sub-dataset, each segment is assigned an inclusion probability proportional to the sub-dataset's occurrences in the segment. This unequal inclusion probability design saves I/O in two aspects. First, it allows CLAP to sample more segments where the queried sub-dataset concentrates, which avoids loading large amount of irrelevant data from disk. Moreover, the unequal inclusion probabilities enable CLAP to avoid over-representing or under-representing a segment unit when we compute the approximation result, which leads to a better accuracy. A better accuracy indicates that for the same error bound, our method needs to sample much less number of segments, which further saves I/O. In a real world, CLAP can take a sampling ratio or an error bound as input from users, and calculate an approximation answer accompanied by meaningful error bounds relative to the precise result. CLAP is open sourced on GitHub and can be supplied to users as a plug-in jar application (**https://github.com/zhangxuhong/SubsetApprox**). Unlike other systems, it adopts a non-intrusive approach which makes no modification to the core of Hadoop such as the scheduler.

While we have implemented CLAP into Hadoop, many of our basic research contributions are not specific to Hadoop, and applicable to other shared nothing frameworks such as Spark. Our comprehensive experimental results indicate that CLAP can significantly reduce application execution delay. For example, the evaluation results on a 121GB Amazon product review dataset and a 111GB TPC-H dataset conclude that, CLAP can achieve a speedup of 8.5× and 20× over the precise execution, respectively, if users are willing to tolerate less than 1% error with 99% confidence. Compared with existing systems, CLAP is more flexible than BlinkDB and more efficient than ApproxHadoop.

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to develop a probabilistic map (SegMap) to reduce the exponential number of recorded sub-datasets to linear, which makes capturing the storage distributions of arbitrary sub-datasets feasible.
- To the best of our knowledge, we are the first to develop a distribution aware online sampling method to efficiently sample data for **a single sub-dataset or multiple sub-datasets** over distributed file systems by applying cluster sampling with unequal probability theory.
- To the best of our knowledge, we are the first to employ configurable sampling unit size and quantitatively derive the optimal size of a sampling unit in distributed file systems by associating it with approximation costs and accuracy.
- We show how **close-form or bootstrap** based estimation theories can be used to **estimate sample size** and compute error bounds for approximations in MapReduce-like systems.
- We implement CLAP into Hadoop as a non-intrusive, plug-in jar application and conduct extensive comparison experiments with existing systems.

2 Background

In statistics, there are three commonly used sampling methods: uniform sampling, stratified sampling and clu-ster sampling. Three examples are shown in Figure 2. Applying these methods to sampling in distributed file systems will involve different costs. Taking HDFS for example, we assume that the content of a large file in HDFS is composed of millions or billions of records. The most straight forward method is uniform sampling, which samples at the record level and randomly pick a subset of all the records. However, given an extremely large number of records, this method is too expensive to be employed for online sampling, as uniform sampling requires a full scan of the whole dataset. For stratified sampling, if we know that most of the queries are on the "City" column, stratified sampling will first group the dataset according to the unique values in the "City" column and then create a random sample for each group. The cost of stratified sampling is one or multiple full scans of the dataset depending on the specific implementation. The advantage of stratified sampling is to ensure that rare groups are sufficiently represented, which may be missed in the uniform sampling (ATL city). A more efficient online sampling method is cluster sampling, which samples at cluster level. The cluster used in current systems [13, 25, 12] is HDFS block and each block is usually sampled with equal probability. Randomly sampling a list of clusters avoids the full scan of the whole dataset.

3 System Design

Figure 3 shows the overal architecture of CLAP. At the input stage of Hadoop, CLAP will first retrieve the sub-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

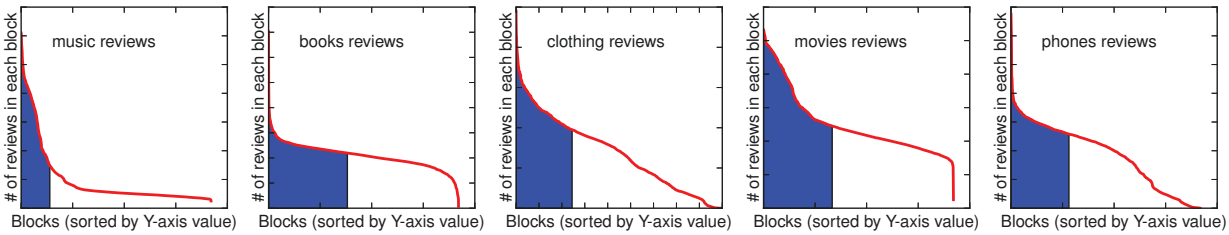


Fig. 1. The storage distribution of sub-datasets in the Amazon review dataset. Shaded area accounts for 50% of a sub-dataset.

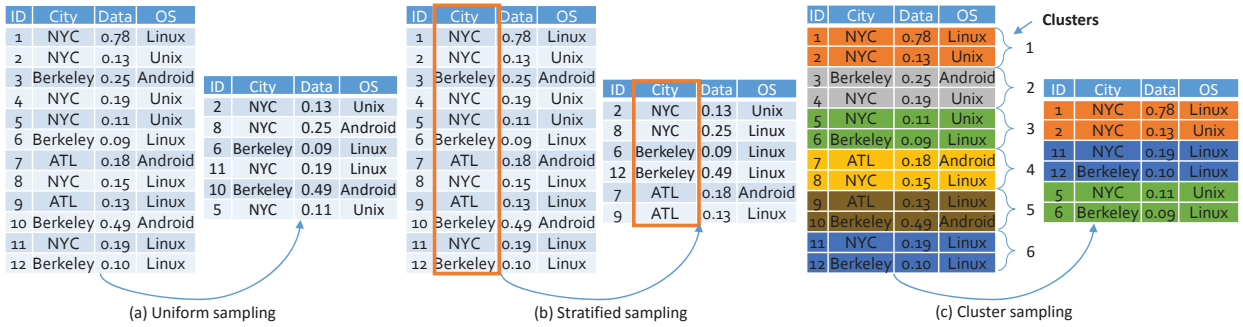


Fig. 2. Commonly used sampling methods

dataset storage distribution information from our efficient and flexible SegMap to estimate the inclusion probability of each segment. Section 3.2 introduces the estimation of inclusion probability and Section 3.3 introduces the creation, storage and retrieval of SegMap. According to the obtained inclusion probability, CLAP will then sample a list of segments for the requested sub-dataset. The sampled segments are further grouped to form input splits, which are finally fed to map tasks. The design of this sample generation procedure is detailed in Section 3.4. Section 3.6 introduces the implementation of the approximation Mapper and Reducer templates. In particular, we implement sampling theory into map and reduce tasks and the calculation of approximation answers and error bounds.

3.1 Applying cluster sampling with unequal probability

In cluster sampling, a cluster is the sampling unit. To apply cluster sampling in distributed file systems such as HDFS, we first define what is a cluster in HDFS. Files in HDFS are usually transformed into records before being fed to Hadoop jobs. In this case, the population in a HDFS file is defined as all the records in the HDFS file. In CLAP, we define cluster as a list of consecutively stored records, which is also referred to as ***segment***. The number of records in each segment is the same and can be an arbitrary integer. Section 3.7 gives a practical guide on how to set the optimal number of records in a segment. When we sample segments for a sub-dataset, the number of records belonging to the queried sub-dataset in each segment will be different due to the skewed distribution. If each segment is sampled with equal probability, an unpleasant outcome could be ending up many sampled segments with few records that belong to the queried sub-dataset, namely wasting a lot of I/O bandwidth. To improve sampling efficiency, we associate each segment with an inclusion probability proportional to its number of records that belong to the queried sub-dataset.

We formally define the inclusion probability of segment i as π_i. This sampling design is also known as the probability proportional to size (**pps**) method [20]. Based on this design, segments containing more records belonging to the queried sub-dataset will have a higher probability to be sampled. Suppose a sub-dataset is distributed over N segments and each segment i contains M_i records belonging to the queried sub-dataset, where M_i is referred to as the occurrences of a sub-dataset in segment i. We then calculate π_i as:

$$\pi_i = \frac{M_i}{\sum_{j=1}^{N} M_j} \tag{1}$$

Another design consideration is to make the variance of an estimator as small as possible. Take estimating the population total τ as an example, we denote the sub-total obtained from each segment i as τ_i. We sample n segments from N segments. The estimator of τ will be $\widehat{\tau} = \frac{1}{n}\sum_{i=1}^{n}(\tau_i/\pi_i)$. Ideally, we would use $\pi_i = \tau_i/\tau$, because for all possible samples, the estimation will be $\widehat{\tau} = \tau$ and the variance of $\widehat{\tau}$ will be 0. However, all the τ_i are unknown until sampled. Alternatively, we can estimate π_i based on some simple observations. For example, π_i can be estimated by using the number of relevant records in a segment, as τ_i is closely related to the number of relevant records in a segment. As discussed above, a better estimate of π_i will lead a more efficient and accurate approximation. This explains that blindly assigning an equal inclusion probability to all the segments will possibly incur large variance for the estimated results.

3.2 Segment inclusion probability estimation for an arbitrary sub-dataset

To make the above sampling design work, we need to record the occurrences of a sub-dataset in all of the segments. Since CLAP aims to support approximations on

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

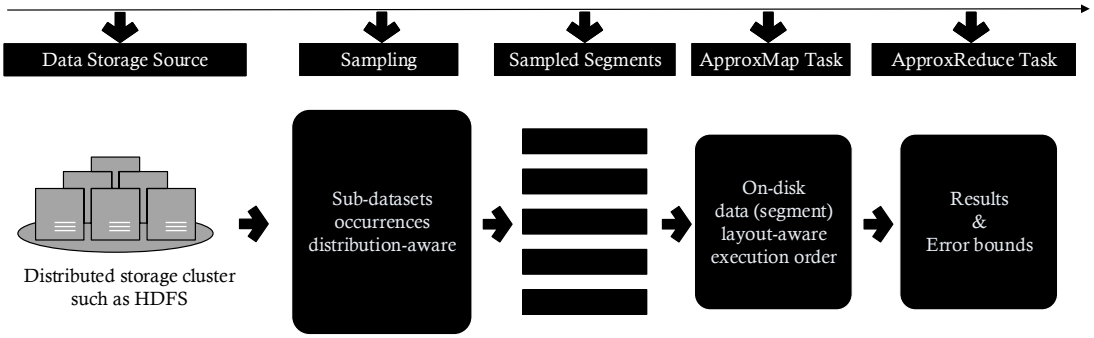


Fig. 3. CLAP architecture.

arbitrary sub-datasets, we need to quantify the number of all possible sub-datasets in a whole dataset. We formally define ϕ as a set of columns or fields in a data record and x as a tuple of values for a column set ϕ. For example, $x = (\text{“}New\ York\text{”}, \text{“}Linux\text{”})$ is a value for $\phi = \{City, OS\}$. Each unique value x represents a sub-dataset. $D(\phi)$ denotes the set of unique x-values over a given ϕ. We can conclude that for a dataset with f columns, there will be 2^f unique ϕ's, and the total number of sub-datasets is $K = \sum_{i=1}^{2^f} |D(\phi_i)|$. Because K is an exponential number, it incurs a prohibitive cost in order to record the entire distribution information. To resolve this problem, we develop a probabilistic distribution map, in which only the occurrences of a sub-dataset with $|\phi_i| = 1$ are recorded, while that of other sub-datasets are estimated using the conditional probability theory. The occurrences are stored in our data structure SegMap. The occurrences of a sub-dataset x across all file segments in SegMap is denoted as $M^x = \{M_1^x, M_2^x, ..., M_N^x\}$. The simplest case for $|\phi| > 1$ is that all columns in ϕ are mutually independent. Suppose there is a sub-dataset x with $x = (k_1, k_2, k_3, ..., k_l)$ for $|\phi_i| = l, 0 < l \leq f$. We can easily compute the probability that value k_j exists in segment i as $P_i(k_j) = M_i^{k_j}/S$, where $M_i^{k_j}$ is recorded in SegMap and S is the segment size. As a result, given the conditional probability under independent events, we can obtain the probability that x exists in segment i as:

$$P_i(x) = \prod_{j=1}^{l} P_i(k_j) \tag{2}$$

For sub-dataset x, segment i's inclusion probability π_i can be estimated as:

$$\pi_i = P_i(x) / \sum_{j=1}^{N} P_j(x) \tag{3}$$

Next, we deal with a more challenging case that columns in ϕ have dependencies. For sub-dataset $x = (k_1, k_2, k_3, ..., k_l)$, we divide its columns into two parts: $k_1 \sim k_g$ denotes columns having dependencies and $k_{g+1} \sim k_l$ represents independent columns. According to the chain rule of conditional probability, the probability that x will exist in segment i is:

$$P_i^{'}(x) = \prod_{j=1}^{g} P_i(k_j | \cap_{t=1}^{j-1} k_t) \times \prod_{j=g+1}^{l} P_i(k_j) \tag{4}$$

We cannot calculate $P_i^{'}(x)$, since we only know $P_i(k_j)$, for $j = 1 \sim l$. Any conditional probability $P_i(k_j | \cap_{t=1}^{j-1} k_t)$ is unknown. To compute $\hat{\pi}_i$, we actually do not need to compute each $P_i^{'}(x)$. Instead, if we can compute the ratios between any pair of $P_i^{'}(x)$ and $P_j^{'}(x)$, then $\hat{\pi}_i$ can be computed using the computed ratios. For example, we compute all the ratios r_i^x between $P_1^{'}(x)$ and all $P_i^{'}(x)$ with $i > 1$. Then, for sub-dataset x with column dependencies, segment i's inclusion probability π_i can be estimated as:

$$\begin{aligned} \pi_i &= \frac{P_i^{'}(x)}{P_1^{'}(x) + P_2^{'}(x) + ... + P_N^{'}(x)} \\ &= \frac{P_i^{'}(x)/P_1^{'}(x)}{1 + P_2^{'}(x)/P_1^{'}(x) + ... + P_N^{'}(x)/P_1^{'}(x)} \\ &= \frac{r_i^x}{1 + r_2^x + ... + r_N^x} \end{aligned} \tag{5}$$

We continue to introduce how to calculate r_i^x. For a single evidence k_1, we can obtain $P_1(k_1)$ and $P_i(k_1)$ from SegMap. Then the ratio $r_i^{k_1}$ between $P_1^{'}(x)$ and $P_i^{'}(x)$ based on evidence k_1 can be calculated as:

$$r_i^{k_1} = \frac{P_i(k_1) \times \prod_{j=2}^{g} P_i(k_j | \cap_{t=2}^{j-1} k_t)) \times \prod_{j=g+1}^{l} P_i(k_j)}{P_1(k_1) \times \prod_{j=2}^{g} P_1(k_j | \cap_{t=2}^{j-1} k_t)) \times \prod_{j=g+1}^{l} P_1(k_j)} \tag{6}$$

In Equation 6, the two conditional probabilities for segment i and 1: $\prod_{j=2}^{g} P_i(k_j | \cap_{t=2}^{j-1} k_t)$ and $\prod_{j=2}^{g} P_1(k_j | \cap_{t=2}^{j-1} k_t)$ are the dependencies for column $(k_1 \sim k_g)$. We assume that for the same set of columns, their dependencies are the same in any segment of a table. For example, in almost any credit card application record, if your occupation is “student”, then your home type has a high probability to be “rent”. Thus, $r_i^{k_1}$ based on evidence k_1 can be simplified as:

$$r_i^{k_1} = \frac{P_i(k_1) \times \prod_{j=g+1}^{l} P_i(k_j)}{P_1(k_1) \times \prod_{j=g+1}^{l} P_1(k_j)} \tag{7}$$

We limit the use of Equation (7) only when the columns in $k_1 \sim k_g$ have strong and consistent dependencies from the first row to the last row of a table. Otherwise, we place the column into the independent columns part $k_{g+1} \sim k_l$. For each dependent column k_j in $(k_1 \sim k_g)$, we can calculate a $r_i^{k_j}$ in a similar way. To aggregate these evidences, we use the geometric mean of the g ratios, as geometric mean is more stable when outliers are present [7], especially for data such as ratios. Due to the relatively small size of a segment, for example a few thousands, the ratios obtained

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

from different dependent column may vary a lot. Then, we calculate the ratio between $P_1^{'}(x)$ and $P_i^{'}(x)$ based on x as:

$$r_i^x = \sqrt[g]{\prod_{j=1}^{g} r_i^{k_j}} = \sqrt[g]{\prod_{j=1}^{g} \frac{P_i(k_j)}{P_1(k_j)}} \times \frac{\prod_{j=g+1}^{l} P_i(k_j)}{\prod_{j=g+1}^{l} P_1(k_j)} \quad (8)$$

Essentially, when dependent columns are present, we are trying to average the occurrence probability we get from each dependent column. This is an refined version of Equation 2 and 3, which simply multiplicate the occurrence probability from each column. From our experiment, correctly identifying dependent columns does produce more precise inclusion probabilities, but the gain is very small comparing to simply assuming independence. However, if columns are falsely identified as dependent columns, the averaging process will loose signals from the actual independent columns, which results in significant error on the inclusion probability estimation. We suggest users should always use Equation 3 if they are not confident on identifying the dependency between columns.

In summary, the number of sub-datasets to be recorded in our SegMap reduces from a factor of 2^f to f for both independent and dependent column scenarios, where f stands for the total number of columns.

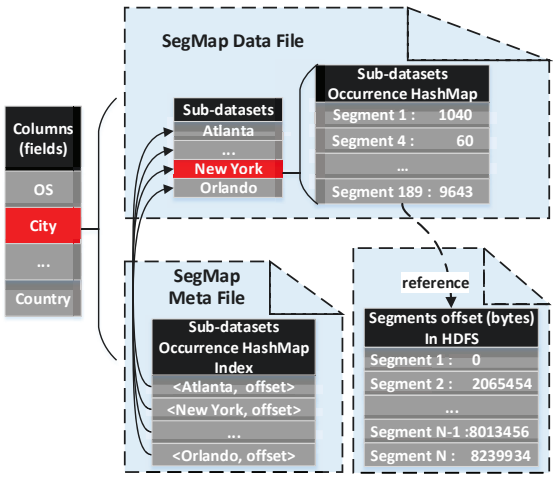


Fig. 4. SegMap structure.

3.3 Creation, update and lookup of SegMap

Next, we introduce the creation, update and lookup of SegMap. We use a common Hadoop job to create SegMap. The user first specifies all possible columns that will be filtered in the future. Then user also has to set a segment size S. In the Map phase, for every S records as a segment, we count the occurrences for each unique x-value in the specified columns and emit ($x + segmentID + column\ id$, occurrence) pairs. We partition the Map output by column id. That is, each reducer will be responsible for collecting the occurrence pairs for all unique x-values under a single column. Each Reducer's input will also be automatically grouped by x-values. If the whole file is divided into N segments, then each x-value group will have a maximum of N occurrence records in SegMap, since an x-value may not appear in all segments. Figure 4 gives an example of the structure of SegMap. Each reducer will generate a SegMap data file and a SegMap meta file for each column. The SegMap data file is binary and consists of multiple occurrence HashMap objects. Each occurrence HashMap object records the occurrences of a sub-dataset x. In order to efficiently locate these occurrence HashMap objects, we use an additional HashMap object in the SegMap meta file to record their offsets in the SegMap data file. The occurrence HashMap objects for frequently queried sub-datasets can also be cached in memory. Finally, each segment's offset in the original HDFS file is stored in the reference file, which will be used when forming input splits.

If a dataset is going to be analyzed only once, then CLAP will not be effective, since building SegMap requires a scan of the whole dataset. In an ideal case, SegMap should be built during the data ingest stage. For example, the building of SegMap can be implemented into Kafka [1]. One advantage of SegMap is that it can be **updated incrementally** when new data are appended, which only requires a scan of the new data.

We now estimate the storage efficiency of SegMap. In the SegMap data file, each HashMap entry is a (segmentID, occurrence) pair. In the SegMap meta file, each HashMap entry is a (x-value, offset) pair. Suppose each (segmentID, occurrence) entry requires r bytes and each (x-value, offset) entry requires k bytes. Assume all the HashMaps have an average load factor δ. The total number of recorded unique x-values is $d = \sum_{(|\phi_i|=1)} |D(\phi_i)|$, where each unique x-value represents a sub-dataset. The total maximum storage cost of SegMap for storing the distribution information of d sub-datasets can be calculated as:

$$Cost(SegMap) = \frac{d \times N \times r + d \times k + N \times r}{\delta} \quad (9)$$

For example, if we have 1TB of data with $16,384$ blocks of 64 MB, each block is assumed to further split into 8 segments. k and r are set to 16 and 32 bytes. Then for each unique x-value a maximum total of $16 \times 16,384 \times 8 + 32 \approx 2$ MB storage is needed. In practice, if a column has a large number of keys, then most of its keys exist in a few segments. We provide more storage overhead results in our conference paper [34].

At the job runtime, to locate the storage distribution information of a sub-dataset, the system needs to perform one sequential read of the whole SegMap meta file and a sequential read of the sub-dataset occurrence HashMap object in the SegMap data file.

3.3.1 *Further reducing storage overhead of SegMap*

If a sub-dataset is uniformly distributed over all the segments, it is unnecessary to record its occurrences for all segments. In order to further reduce the storage overhead of SegMap, we propose to only record an average occurrence for sub-dataset that is uniformly distributed. To test whether the storage distribution of a sub-dataset follows an uniform distribution, we employ the "chi-square" test [32]. This process is performed after the initial SegMap is created. With this test, the storage overhead of SegMap is greatly reduced.

3.4 Online input sampling for a single dataset

We implement the sampling stage in new *classes* of input parsing. For example, we implemented *CLAPTextInputFor-*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

mat, which is similar to Hadoop's *TextInputFormat*. Instead of using all the data blocks of a file to generate input splits, CLAPTextInputFormat will use a small list of file segments sampled from all the blocks according to a given sampling ratio or error. In CLAPTextInputFormat, it will first read all block information such as block offset, and blocks locations. Then, it will load the storage distribution information for all requested sub-datasets from SegMap. According to the sub-datasets storage distribution, each segment is assigned an inclusion probability. We adopt a random segment sampling procedure that models this unequal inclusion probability. The *cumulative-size* [3] method is employed and works as follows:

1) Generate N cumulative ranges as: $[0, \pi_1], [\pi_1, \pi_1 + \pi_2], [\pi_1 + \pi_2, \pi_1 + \pi_2 + \pi_3], ..., [\sum_{i=1}^{N-1} \pi_i, 1]$.
2) Draw a random number between 0 and 1. If this number falls into range i, then include segment i in the sample.
3) Repeat until the desired sample size is obtained.

After obtaining the sample list of segments, we grouped them to form input splits. Notice that all records in a segment are fed to the Mapper and filtering is still done at the Mapper. If segments are grouped arbitrarily, most of the generated splits may contain data that spans on multiple machines. This will ruin Hadoop's locality scheduling. To preserve Hadoop's data locality, we retrieve the locations of each sampled segment from HDFS block locations and implement a locality aware segment grouping procedure as shown in Figure 5. The basic idea is that all sampled segments that are located on the same node are randomly grouped to form splits. All the leftover segments on the same rack are then randomly grouped to form more splits. Finally, all remaining segments are arbitrarily grouped. This ensures that the data in most of the splits are either from the same node or from the same rack.

3.5 Online input sampling for multiple sub-datasets

One commonly used operator in SQL query is ***Group By***. Take the following query as an example.

```
SELECT SUM(reviews)
FROM Movies
Group By genres
```

This query will compute a result for each genre sub-dataset under the genres column. If we want to perform approximations for this query, we have to generate samples for each genre sub-dataset. The intuitive approach for this problem is simply repeating the sampling method on a single dataset introduced in Section 3.4 for each sub-dataset involved in the *Group By* query. However, this approach will quite probably suffer from I/O inefficiencies due to the different storage distribution of each sub-dataset. To better explain this problem, Figure 6 shows the storage distributions of five sub-datasets over the segments of the Amazon reviews dataset. We can see that different sub-dataset concentrates on different segments of the Amazon reviews dataset. Our sampling method on a single dataset will always include

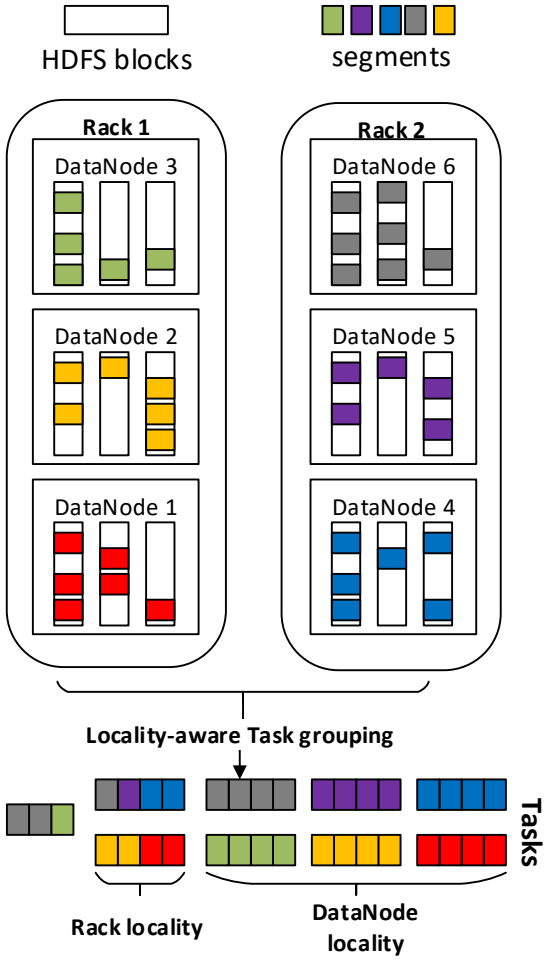


Fig. 5. Locality aware segments grouping

the segments where the interested sub-dataset concentrates on. As a result, the sampled segments for each sub-datasets involved in the *Group By* query may have little overlap. For example, for clothing sub-dataset, most of the segments will be sampled near the end of the Amazon reviews dataset, while for phones sub-dataset, most of the segments will be sampled around the middle of the Amazon reviews dataset. This may cause the majority of the Amazon reviews dataset to be loaded from disk, even though a very small sampling ratio is specified for the *Group By* query.

To improve the I/O efficiency, a second approach is to perform sampling for all the involved sub-datasets simultaneously. Specifically, we will compute a new inclusion probability for each segment by combining each sub-dataset's distribution in this segment. In the movie genres example, we can calculate the inclusion probability of a segment i for each genre sub-dataset according to methods introduced in Section 3.2. We denote all the genres G in the movies dataset as $\{g_1, g_2, ..., g_l\}$.Then the calculated inclusion probabilities of segment i for each genre can be denoted as $\pi_i(g_1), \pi_i(g_2), ..., \pi_i(g_l)$. The sum of these inclusion probabilities is denoted as $w_i = \sum_{j=1}^{l} \pi_i(g_j), 1 \leqq i \leqq N$. Then, we can compute the inclusion probability of segment i regarding of all the genres as:

$$\pi_i(G) = \frac{w_i}{\sum_{j=1}^{N} w_j} \tag{10}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

Fig. 6. The storage distribution of five sub-datasets in the Amazon reviews dataset. Different sub-dataset concentrates on different segments of the Amazon reviews dataset

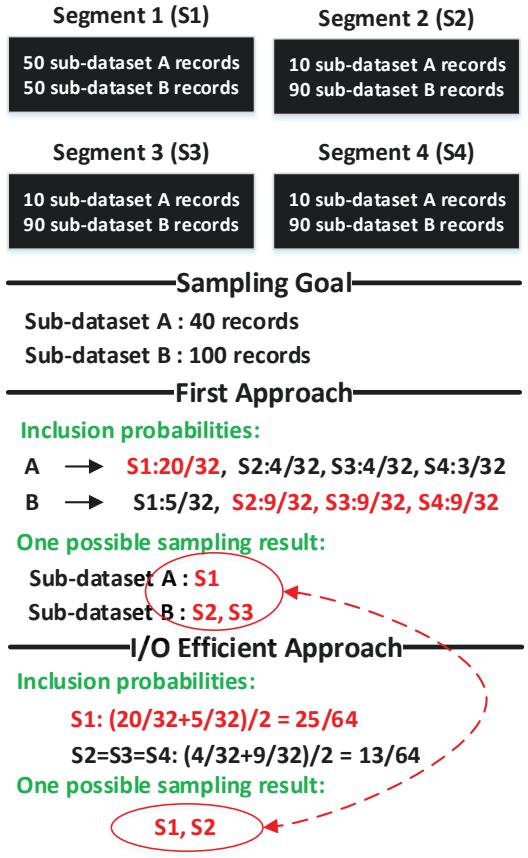


Fig. 7. An I/O efficient approach for sampling multiple sub-datasets. For the same sampling goal, the I/O efficient approach only needs to sample two segments from disk.

With this new inclusion probability, we can perform the online sampling for multiple sub-dataset as if we are performing sampling for a single sub-dataset. The only difference is that the *cumulative-size* sampling process stops only when all the involved sub-datasets' sampling ratios are satisfied. The design consideration of this approach is that the inclusion of a segment into the sample is based on its contribution to the goal of achieving the sampling ratios for all the sub-datasets rather than a single sub-dataset. Thus, any included segment is shared among the samples of each involved sub-dataset. As a result, a possible smaller number of segments is able to satisfy the sampling ratios of all the involved sub-datasets. Figure 7 gives an example of the I/O efficiency of this approach. As shown in the figure, for the same sampling goal, the I/O efficient approach only needs to sample two segments from disk instead of three segments with the first approach.

However, the second approach may incur a larger variance of the estimated result for each sub-dataset in the *Group By* query. As discussed in Section 3.1, the ideal inclusion probability π_i for a segment i should be proportional to τ_i. The estimation of inclusion probability for a single dataset in Section 3.2 follows this principle by relating π_i to the number of relevant records in segment i. The simple observation is that τ_i is closely related to the number of relevant records in a segment. However, the newly calculated inclusion probability π_i in the second approach is no longer a good indication of how many relevant records of a sub-dataset exist in segment i. To conclude, though the second approach improves the I/O efficiency, the accuracy of the estimated results is compromised.

Clearly, there is a trade-off between the two introduced methods. We formally refer the first accuracy-oriented approach as **LOWVAR** and the second I/O oriented approach as **LOWIO**. Benefiting from the design of CLAP, the sampling procedure can be carried out by only referring to our probabilistic SegMap without touching the physical datasets. Therefore, a possible practice is that when we perform sampling for multiple sub-datasets, both of the sampling methods are carried out. Each sampling method will give the actual size of data that needs to be read from disk. If the ratio of data size of **LOWVAR** to that of **LOWIO** does not exceed a threshold h, we will adopt **LOWVAR** for better accuracy, otherwise **LOWIO**.

3.6 Estimation in ApproxMap and ApproxReduce Task

3.6.1 *Closed-form based estimation*

CLAP adopts standard closed-form formulas from statistics [20] to compute error bounds for approximation applications that compute aggregations on various sub-datasets of the whole dataset. The set of supported aggregation functions includes SUM, COUNT, AVG, and PROPORTION.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

We use the approximation of SUM as an example, approximation for other aggregation functions is similar. Suppose that each element in segment i has an associated value v_{ij}. We want to compute the sum of these values across the population, $i.e,\ \tau = \sum_{i=1}^{N}\sum_{j=1}^{M_i} v_{ij}$. To approximate τ, we need to sample a list of n segments and they are random selected based on their inclusion probability π_i. The sum for each segment i can be obtained as $\tau_i = \sum_{j=1}^{M_i} v_{ij}$. One stage cluster sampling [20] with unequal probability then allows us to estimate the sum τ from this sample as:

$$\widehat{\tau} = \frac{1}{n}\sum_{i=1}^{n}(\tau_i/\pi_i) \pm \epsilon \tag{11}$$

where the error bound ϵ is defined as:

$$\begin{aligned} \epsilon &= t_{n-1,1-\alpha/2}\sqrt{\widehat{V}(\widehat{\tau})} \\ \widehat{V}(\widehat{\tau}) &= \frac{1}{n}\frac{1}{n-1}\sum_{i=1}^{n}(\frac{\tau_i}{\pi_i} - \widehat{\tau})^2 \end{aligned} \tag{12}$$

where $t_{n-1,1-\alpha/2}$ is the value of a t-distribution with $n-1$ degrees of freedom at a confidence interval of $1-\alpha$ and s_τ^2 is the variance of τ_i from each sampled segment. According to the definition of unbiased estimator in [20], $\widehat{\tau}$ is an unbiased estimator of τ, since $E[\widehat{\tau}] = \sum_{i=1}^{N}\pi_i \times \frac{\tau_i}{\pi_i} = \tau$. We can see that the accuracy of estimated π_i does not affect the unbiased property of our estimator. On the other hand, the reason why CLAP could produce a more accurate estimation and smaller variance is that the sum τ_i from each segment i is compensated by its inclusion probability π_i before calculating results and variance.

Our pre-defined ApproxMap and ApproxReduce templates implement the above estimation of aggregation result and error bounds. Specifically, the ApproxMap task collects necessary information such as segment inclusion probability and which segment each key/value pair comes from. The inclusion probabilities are passed to reducers using a special key, and segment id is tagged into the key of each key value pairs as *(key+segmentID)*. A customized partitioner is provided to extract the real key from the tagged key such that each key value pair is shuffled to reducers as usual. In the ApproxReduce task, all the key-value pairs for each key are automatically merge-sorted into n clusters by segment id. Each cluster is represented as *(segment i, list(v_{ij}))*. Together with the passed inclusion probability, we can estimate the final result and its error bounds with Equations (11) and (12). Figure 8 shows an example of using our templates to implement a SUM approximation job on a sub-dataset x and the command to submit this job.

```
class MyCLAPApp{
 class MyCLAPMapper extends CLAPMapper{

  //Input key value pairs are already sampled
  void map(key, value){
      if(value belongs to sub-dataset x)
          context.write(x, v_ij);
  }
 }
 class MyCLAPReducer extends CLAPReducer{

 //Equation (11)&(12) are applied autumatically
  void reducer(key, Iterator values){
      double sum;
      for value in values:
         sum+=value;
      context.write(key, sum);
  }
 }
 public static void main(){
  //job configurations
  //...
  setPartitionerClass(CLAPPartitioner);
  setInputFormatClass(CLAPTextInputFormat);
  run();
 }
 /*****************Job submission***********
 Hadoop jar CLAP.jar MyCLAPApp
 -r 0.2 -w column=x
 ----------------------------------------
 -r: sampling ratio
 -w: WHERE clause
 ****************************************/
```

Fig. 8. Example of developing a SUM approximation job on a sub-dataset x with CLAP.

3.6.2 Bootstrap based estimation

CLAP is also extended to support more complex approximations such as ratios, regression, and correlation coefficients using resampling methods such as bootstrapping for error estimation. Using bootstrap for error estimation has been explored in many works [18, 26]. CLAP's booststrap based estimation is based on the theories introduced in [27, 9, 5]. Work in [27] provides methods on how to bootstrap in cluster sampling. In cluster sampling, the n sampled segments are i.i.d. Therefore bootstrap is applicable at the segment level. For ease of understanding, we show how to estimate SUM using bootstrap. The estimator is the same as shown in Equation 11. The main difference is the way of calculating variance and confidence intervals. Closed-form based estimation calculates variance based on one single set of samples. The bootstrap based estimation attempts to obtain multiple set of samples from one seed set of samples via bootstrapping. Then the statistics obtained from all the bootstrapped samples are used to estimate the variance. For example, we will estimate a SUM from each bootstrapped set of sample, and then calculate the variance based on the distribution of these estimated SUMs. Specifically, we sample n segments as described in Section 3.4 and 3.5 as our seed sample set, and then obtain more samples set by repeating the procedure of re-sampling m segments with replacement and equal probability from the seed sample set. As shown in Section 3.6.1, the reducer already groups all the key-value pairs into n clusters represented as *(segment i, list(v_{ij}))*, which is our seed sample set. Thus the resampling procedure can be easily conducted at the reducer. The sum from each of the m resampled segment is denoted as: $\tau_1^*, \tau_2^*, ..., \tau_m^*$. Note that this resampling procedure does not involve extra I/O cost. To compute τ_i^* from segment i, the *list(v_{ij})* will not be re-iterated from disk or memory, since for the same segment i, τ_i^* is the same as τ_i which is obtained from the first iteration of the list(v_{ij}). From these m re-sampled segments we can compute one estimate of our target SUM, which we will denote as $\widehat{\tau}^*$. We repeat

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

this re-sampling procedure for B times and get B bootstrap estimates denoted as $\widehat{\tau}^{*1}, \widehat{\tau}^{*2}, ..., \widehat{\tau}^{*B}$. Then, we can get the variance as follows:

$$\widehat{V}(\widehat{\tau}^*) = \frac{1}{B-1}\sum_{i=1}^{B}(\widehat{\tau}^{*i} - \bar{\widehat{\tau}^*})^2 \quad (13)$$

where $\bar{\widehat{\tau}^*}$ is the mean of the B bootstrap estimates. Now, we describe how to compute $\widehat{\tau}^*$ from one set of re-sampled m segments. We first introduce using weights to compute estimation. By setting weight $w_i = \frac{1}{n\pi_i}$, we can write the estimation of SUM as $\widehat{\tau} = \sum_{i=1}^{n} w_i\tau_i$. Then the bootstrap estimation from the m re-sampled segments is $\widehat{\tau}^* = \sum_{i=1}^{m} w_i^*\tau_i^*$, where $w_i^* = (1-\lambda+\lambda r_i^*\frac{n}{m})w_i$, $\lambda = (\frac{m}{n-1})^{1/2}$, and r_i^* is the number of times that segment i was resampled. The question of optimal m is open. As suggested in [27, 9], the choice of $m = n-1$ is natural. Then the rescaled weight is simplified to $w_i^* = \frac{n}{n-1}r_i^*w_i$.

To calculate confidence interval, we employ the most popular percentile based method [8]. The idea is simple. To obtain the $(1-\alpha)$ confidence interval, the method takes the $\alpha/2$ and $1-\alpha/2$ percentile of the B bootstrap estimates. The issue of how many bootstrap samples (B) are required to provide an acceptable variance estimate arises. The precision of the variance estimator continues to increase as the number of bootstrap samples increase, while the resources needed to carry out these bootstrap samples obviously increase as well. In Booth's [5] and Efron's [8] work, the suggested choice of B is 800 and 1000, respectively. In this paper, we choose $B = 800$. To bootstrap for a more general approximation application θ, just replace the $\widehat{\tau}^*$ obtained from each bootstrap with $\widehat{\theta}^*$. All the remaining calculations are the same. For example, in our evaluation, we approximate the ratio of SUM between two sub-datasets i and j. For each of the resampled bootstrap sample, we can estimate the SUM for sub-dataset i as $\widehat{\tau}_i$ and sub-dataset j as $\widehat{\tau}_j$. Then the estimated ratio $\widehat{\theta}^*$ obtained from this bootstrap sample is calculated as $\widehat{\theta}^* = \frac{\widehat{\tau}_i}{\widehat{\tau}_j}$. Finally, repeat this procedure for B times to calculate the variance and confidence intervals as introduced before.

3.6.3 Sample size estimation

We also implement an option in CLAP to allow users to specify an error bound. Generally, with a larger sample size, the error of an approximation will be smaller. To achieve a desired approximation error, we have to find the minimum number of sample records needed. In simple random sampling (SRS), estimating the required sample size for a given error is quite easy. Taking estimating population mean as an example, its error bound is estimated by $\epsilon = t_{n-1,1-\alpha/2}\sqrt{\frac{s^2}{n_{srs}}}$ [20]. To estimate the required sample size n_{srs} for a given error, only population variance s^2 needs to be estimated. However, this estimated size may not be enough for cluster sampling. It mainly depends on the homogeneity of elements in each cluster. For example, suppose we have two bags of beans: one small bag of red beans and one big bag of black beans. Someone mixes them into one bag and we want to estimate the quantity for each kind of bean or their percentages in the new bag. We use a spoon to take some sample beans, if the beans are perfectly mixed, then just one spoon of beans should be enough. Otherwise more random spoons are needed to get a satisfactory result. This theory also applies to our segment sampling. The pilot job is used to estimate the segments' homogeneity. Formally, in sampling theory, this homogeneity leads to the sample design effect, which is defined as: $deff = \frac{Variance_{cluster}(\tau)}{Variance_{srs}(\tau)}$ [17]. It is the factor by which the variance of an estimator under the clustering sampling design is over or underestimated by the SRS design. Through $deff$ we can get the require sample size and number of segments under our sampling as:

TABLE 1
Example settings of optimal segment size (number of records). ρ is the homogeneity of segments.

Per record size \ ρ	0.01	0.5
10 byte	10,000	1,000
1 KB	1,000	100

$$n = \sum_{i=1}^{m} M_i = n_{srs} \times deff \quad (14)$$

Now, we describe how to estimate $deff$ in our segment sampling. Since the pilot job still uses the same segment sampling rather than SRS, so we can not directly compute a $variance_{srs}$. Alternatively, $deff$ can be estimated in the following way, $\widehat{deff} = \widehat{deff_\pi} \times \widehat{deff_\rho}$ [11], where $\widehat{deff_\pi}$ is the estimated design effect caused by un-equal inclusion probability and $\widehat{deff_\rho}$ is the estimated design effect caused by homogeneity ρ. In CLAP, $deff_\pi$ can be estimated as $\frac{\sum_{i=1}^{m}\pi_i^2}{(\sum_{i=1}^{m}\pi_i)^2}$. While $deff_\rho$ can be estimated using Kish's [17] formula $\widehat{deff_\rho} = 1 + (M_0 - 1)\widehat{\rho}$, where M_0 in CLAP is the weighted average number of records in all sampled segments. Homogeneity ρ can be estimated in many ways [10, 20, 28], and we pick the one that is commonly used in statistics software such as SPSS. This method estimates ρ by using information from inter-segment variance and intra-segment variance [28]. The inter and intra-segment variance can be obtained at the reduce phase, since the key value pairs to the Reducer are already grouped by segment id as discussed in section 3.6.

3.7 Deriving the optimal segment size

In this section, we give a practical guide on how to set an optimal segment size. Segment size is closely related to the variance of approximation answers and system costs. The costs in our system are divided into two parts: the I/O cost of reading sample data and the storage cost of storing sub-dataset storage distribution information. Generally, for a given sample size, the variance decreases with more segments and increases with a larger segment size. On the other hand, the cost increases with more segments and decreases with a larger segment size. For example, with more segments, Hadoop job must perform more disk seeks, and the storage cost of SegMap will also increase as indicated by Equation (9). We further divide the I/O cost of reading a segment into the seek cost and sequential read

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

cost. The cost of a sample design with m segments and a segment size of M_0 is formulated as:

$$C = mc_1 + mM_0c_2 \tag{15}$$

where C is the total cost, c_1 is the cost of one segment seek time in HDFS and the storage cost of storing one segment information in SegMap, and c_2 is the cost of reading one record in a segment. With a fixed variance $\widehat{V}(\widehat{\tau})$, we want to minimize the cost. To derive the optimal segment size from Equation (15), we will incorporate Kish's formula on cluster sampling [17]: $\widehat{V}(\widehat{\tau}) = \widehat{V}_{srs}(\widehat{\tau}) \times deff, deff = 1+(M_0-1)\rho$, where ρ denotes the homogeneity of records in a segment, $\widehat{V}_{srs}(\widehat{\tau})$ represents the variance using simple random sampling, M is the total number of records, s^2 is the variance of values in all the records and $deff$ is the design effect of cluster sampling.

$$\begin{aligned} \widehat{V}_{srs}(\widehat{\tau}) &= M^2 \times s^2/mM_0 \\ \widehat{V}(\widehat{\tau}) &= M^2s^2[1+(M_0-1)\rho]/mM_0 \\ m &= C/(c_1+M_0c_2) \\ C &= \frac{c_1M^2s^2(1+M_0c_2/c_1)[1+(M_0-1)\rho]}{M_0 \times \widehat{V}(\widehat{\tau})} \end{aligned} \tag{16}$$

By minimizing the above derived C, we get the optimal segment size as:

$$M_0 = \sqrt{\frac{c_1}{c_2}\frac{1-\rho}{\rho}} \tag{17}$$

Equation (17) suggests that if each record in a dataset is very large, then one will get a smaller segment size. Now, we have a closer look at c_1. The seek process in HDFS is complex. It first needs to contact namenode to find the datanodes containing the requested data, and then initiates a file stream followed by a local disk seek. In a local disk, we assume the seek time is about 10^4 times that of reading one byte. Here in HDFS, we assume $seek/read = 10^5$, and we also assume that the storage cost factor of storing one segment information in SegMap relative to the disk read of one record is about 100. For different estimated ρ and record size (byte), we can get the desirable segment sizes shown in Table 1. In practice, a user can set the cost ratio of c_1/c_2 according to their real settings, and ρ can be estimated by simply examining several segments as discussed in Section 3.6.3.

4 Evaluation

4.1 Experimental setup

TABLE 2
Datasets.

dataset	size(GB)	# of records	avg record size
Amazon review	116	79,088,507	1.5 KB
TPC-H	111	899,999,995	0.13 KB

Hardware. We evaluate CLAP on a cluster of 11 servers. Each server is equipped with two Dual-Core 2.33GHz Xeon processors, 4GB of memory, 1 Gigabit Ethernet and a 500GB SATA hard drive.

Datasets. We use an Amazon review dataset including product reviews spanning May 1996 to July 2014 [22], which can be obtained here http://jmcauley.ucsd.edu/data/amazon/ and the LINEITEM table from the TPC-H benchmark [2]. Table 2 gives their detailed information. Each Amazon review record contains columns such as price, rating, helpfulness, review content, category, *etc*. Each TPC-H record contains columns such as: price, quantity, discount, tax, ship mode, *etc*.

Approximation metrics explanation:
Actual error $= \frac{approximate\ answer - precise\ answer}{precise\ answer}$.
99% confidence interval: *for 99% of times, the approximate answers are within the interval.*

4.2 Accuracy validation of estimated inclusion probabilities

Since SegMap records accurate occurrences only for sub-datasets with $|\phi| = 1$, the calculated inclusion probability π_i for them are accurate. However, for sub-data-sets with $|\phi| > 1$, π_i is estimated based on the occurrences of sub-datasets with $|\phi| = 1$ using conditional probability. Figure 9 shows the estimated π_i for sub-datasets with $|\phi| = 2$, $|\phi| = 3$ and $|\phi| = 4$ on the two datasets and the accurate π_i. For each $|\phi|$, we pick an example sub-dataset and plot its estimated and precise distribution. Then the average error for each $|\phi|$ is shown on the right. The results show that the estimated π_i under the independence assumption have a very high accuracy. In Figure 9(b) and (c), we also plot the estimated π_i under the dependence assumption. For the TPC-H dataset, values in column "discount" and "tax" are independently generated. Therefore, estimating the π_i under the dependence assumption will incur large errors. For the Amazon review dataset, the overall rating and helpfulness of a review are lightly related. Therefore, estimating the π_i under the dependence assumption produces comparable accuracy as that of independence assumption. With larger $|\phi|$, the average error increases on both dataset. This is because with a larger $|\phi|$, the occurrence probability of a sub-dataset $P_i(x)$ in a segment i decreases. According to the law of large numbers, with a fix segment size, a very low $P_i(x)$ will not represent the real number of matching records very well.

To further validate the effectiveness of estimating inclusion probability under dependence, we generate a new synthetic table with 5 columns. We explicitly generate column 5 with dependency on column 4. For example, if the value on column 4 is k, the value on column 5 will be k' with a probability of 70%. The remaining columns are generated independently. We first plot the estimated distribution for $\phi = \{column\ 4, column\ 5\}$ in Figure 10(a). The results show that the estimated distribution with dependence matches very well with the precise distribution. The estimated π_i with independence is either too large or too small relative to the precise π_i. In Figure 10(b), each ϕ includes the two dependent columns. The reported average error with independence is always larger. In summary, we do not recommend estimating inclusion probabilities under the dependence assumption, unless users are certain that some columns have strong dependency.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

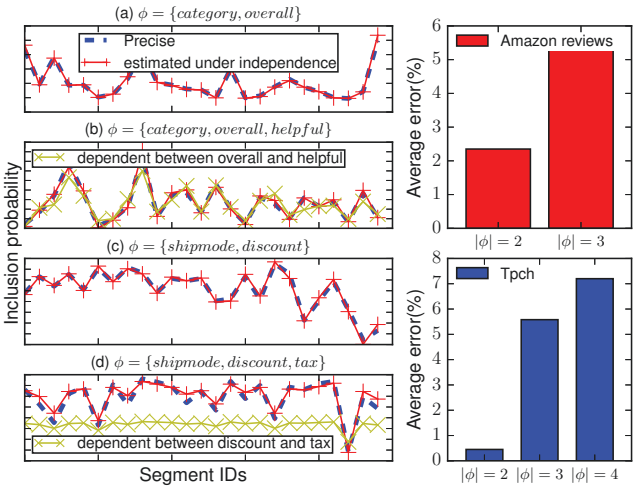


Fig. 9. Accuracy validation of estimated inclusion probability. Columns examined in the two datasets: (category, overall, helpful, time) and (shipmode, discount, tax, returnflag, linestatus).

4.3 Approximation accuracy and efficiency

4.3.1 Results for closed-form based estimation

We pick five sub-datasets from each dataset to evaluate CLAP's approximation accuracy. The average number of records in a sub-dataset from the Amazon review dataset is relatively smaller. Therefore, for each sub-dataset in the Amazon review dataset, we use a sampling ratio of 20%, and for the TPC-H dataset, we use a sampling ratio of 10%. Note that this ratio is the percentage of data sampled in each sub-dataset, while not the percentage of the whole dataset. The segment size is configured as 1,000 for the Amazon review dataset and 10,000 for TPC-H dataset. The approximation applications evaluated are AVG on "Music, Movies, Clothing, MAIL, SHIP" sub-datasets and SUM on "Books, Phones, AIR, RAIL, TRUCK" sub-datasets.

Figure 11(a)&(d) plot the approximation accuracy results on both datasets. The error bars show the 99% confidence intervals of the approximation results. All approximation results are normalized to the precise results indicated by the 100% guide line. The execution time and actual input data ratios of the whole dataset are plotted in Figure 11(b,c)&(e,f). For the Amazon review datasets, all approximation errors of the five sub-datasets are within 1%. The speedup over default Hadoop precise execution ranges from 5 to 8.5. These speedup can be explained by the actual low input data ratio as shown in Figure 11(c). For example, to sample 20% data of the music sub-dataset, CLAP only needs to read 6.4% of the whole dataset. The different speedups and input ratios are due to the different skewness of the storage distribution of a sub-dataset. Although the speedup on the music sub-dataset is the highest, its approximation confidence interval is larger than others. This is because with a smaller input ratio, the number of input segments is also smaller. As indicated in the variance formula, more segments will result in a smaller variance. The results on the TPC-H dataset are similar. However, the overall approximation error is much smaller than that on the Amazon dataset. The reason is that the size of one record in the Amazon review dataset is about 10 times of a record in the TPC-H dataset.

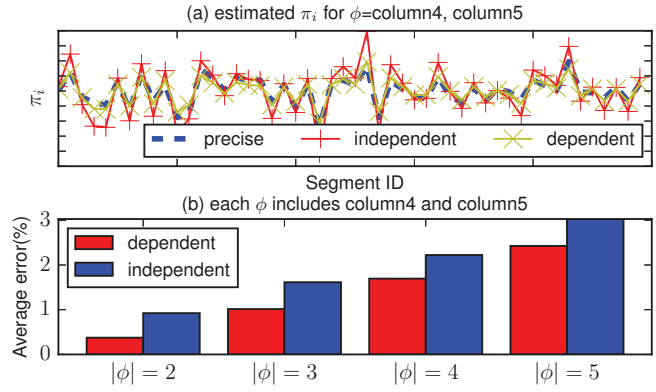


Fig. 10. Accuracy comparison of estimating inclusion probability with dependence and independence. Experiments are on a new synthetic table with 5 columns. Column 5 is generated with dependency on column 4. Other columns are generated independently.

Even with a smaller 10% sampling ratio, the total number of sampled records is still larger. In addition, the randomness of values in the synthetic TPC-H dataset is better than that of the Amazon review dataset. In the TPC-H dataset, the speedup for the MAIL sub-dataset is the highest. This is because the MAIL sub-dataset has the most skewed distribution while other sub-datasets have almost the same distribution as illustrated in Figure 12.

4.3.2 Results for bootstrap based estimation

Figure 13 shows CLAP's precision and execution time on a more complexed operation *ratio* using bootstrap based estimation. R1 is the set of pair wise ratios for sub-dataset in (MAIL, SHIP, AIR, RAIL, TRUCK). For example, the ratio of SUM between MAIL and AIR is $\frac{SUM(Item\ quantity\ with\ MAIL\ shipping)}{SUM(Item\ quantity\ with\ AIR\ shipping)}$. R2 is the set of pair wise ratios for sub-dataset in (Music, Books, Movies, Clothing, Phones). For example, the ratio between Book and Music is $\frac{SUM(Books\ reviews\ length)}{SUM(Music\ reviews\ length)}$ The sampling percentage for R1 is 10% and R2 is 20%. The average of precision and execution times are reported. The 99% confidence intervals for the two set of ratio estimations are about 4% and 3% respectively. The speedups are still significant, which is about 7 for R1 and is about 5.5 for R2. To further evaluate the performance of bootstrap based estimation, in Figure 14, we compare the precision and execution times for approximating SUM using both closed-form and bootstrap based estimation methods. Two sub-datasets are evaluated from both datasets. The results show that the precision of bootstrap based estimation is comparable with the closed-form based one. The execution time of bootstrap based estimation is only several seconds longer. This is caused by the extra B times of resampling procedures in the bootstrap estimation. However, this extra computation time is negligible in the overall execution time. The overall execution time is mainly determined by the amounts of accessed input data. Both of the two estimation methods use our weighted sampling method. Therefore, they can minimize the amounts of accessed input data.

4.4 Comparison with ApproxHadoop

In this sub-section, we compare CLAP's performance with the most recent online sample-based ApproxHa-doop. ApproxHadoop can only use HDFS block as sampling unit

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

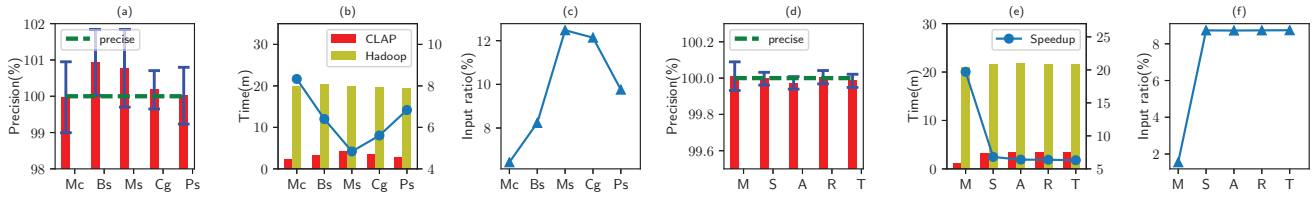


Fig. 11. Approximation accuracy and efficiency of CLAP on different sub-datasets. Results are normalized to precise result. The error bars are the 99% confidence intervals of approximation results. 20% and 10% records of each sub-dataset is sampled for Amazon and TPC-H datasets respectively. (Mc, Bs, Ms, Cg, Ps):(Music, Books, Movies, Clothing, Phones), (M, S, A, R, T):(MAIL, SHIP, AIR, RAIL, TRUCK).

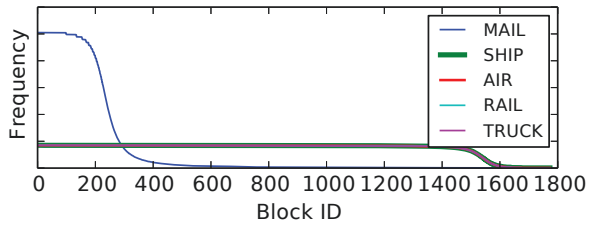


Fig. 12. Storage distributions of sub-datasets in TPC-H dataset

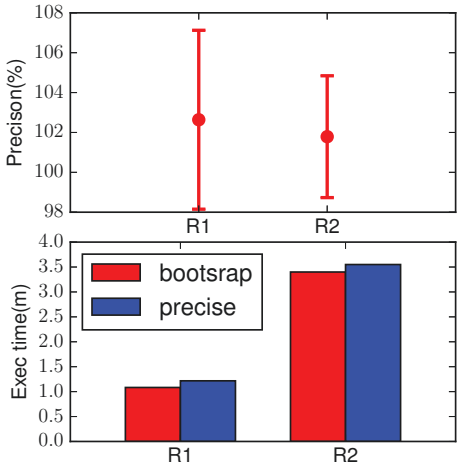


Fig. 13. Average approximation precision and execution time for two sets of ratio estimations with bootstrap. R1 is the set of pair wise ratios for sub-dataset in (MAIL, SHIP, AIR, RAIL, TRUCK). R2 is the set of pair wise ratios for sub-dataset in (Music, Books, Movies, Clothing, Phones).

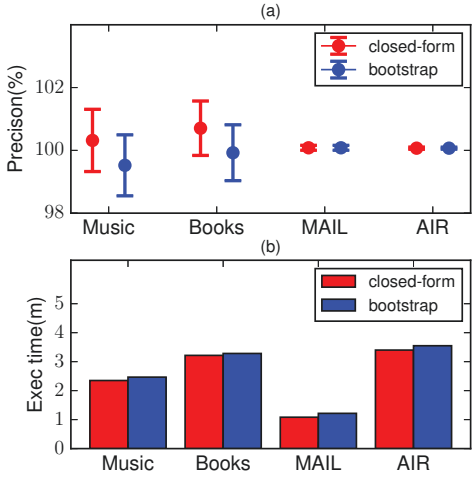


Fig. 14. Precision and execution time comparison between the closed-form and bootstrap based estimation. The approximated aggregation is SUM.

and it samples each block with equal probability regardless of which sub-dataset is queried. The approximation application evaluated is SUM and the queried sub-datasets are "Music" in Amazon review and "MAIL" in the TPC-H dataset. The segment size used in CLAP is 1,000 for the Amazon review dataset and 10,000 for the TPC-H dataset. The block size used in the experiment is 64 MB. For a fair comparison, we also configure CLAP to use HDFS block as the sampling unit.

Figure 15 reports the approximation accuracy and execution time comparison results. As shown in both Figure 15(a) and (e), the confidence intervals produced by ApproxHadoop are extremely wide, which are unacceptable. On the Amazon review dataset, the confidence intervals are about 3 times wider than CLAP with block unit and 16 times wider than CLAP with 1,000 unit. On the TPC-H dataset, the confidence intervals produced by ApproxHadoop are even worse, which is more than 40 times wider. This can be explained by ApproxHadoop's ignorance of the skewed storage distribution of the queried sub-datasets and its variance formula [13]: $\hat{Var}(\hat{\tau}) = N(N-n)\frac{s^2}{n}$, where n is the number of sampled blocks, N is the total number of blocks, and s^2 is the variance of the associated sum of each sampled block. Generally, if a block contains more number of records belonging to the queried sub-dataset, the sum computed from this block will also be larger. Therefore, the skewed distribution of the queried sub-dataset over all of the blocks makes the sum computed from each block has a very large variance. However, in CLAP, the sum computed from each block or segment is scaled by its inclusion probability, making the sum of each block or segment has a very low variance.

On the other hand, the actual errors of ApproxHadoop are also larger than that of CLAP. To explain this, we record the actual number of records sampled for a sampling ratio in both systems. In Figure 15(b) and (f), the sampling quantities are normalized to the precise quantity ($population \times ratio$). Negative values indicate that the number of records is less than the precise quantity, while positive values indicate the number of records is more than the precise quantity. The number of sampled records in ApproxHadoop is either larger or smaller. In both CLAP and ApproxHadoop, the computed sum from the samples is scaled by the sampling ratio to get the global sum. Clearly, if the number of sampled records is less than the precise quantity, the computed sum from the samples will be under-scaled, resulting a smaller global sum. Similarly, if the number of sampled records is greater than the precise quantity, the computed sum will be over-scaled, resulting a larger global sum.

Figure 15(c) and (g) show the execution time comparison results. The execution times of ApproxHadoop are 2 times

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

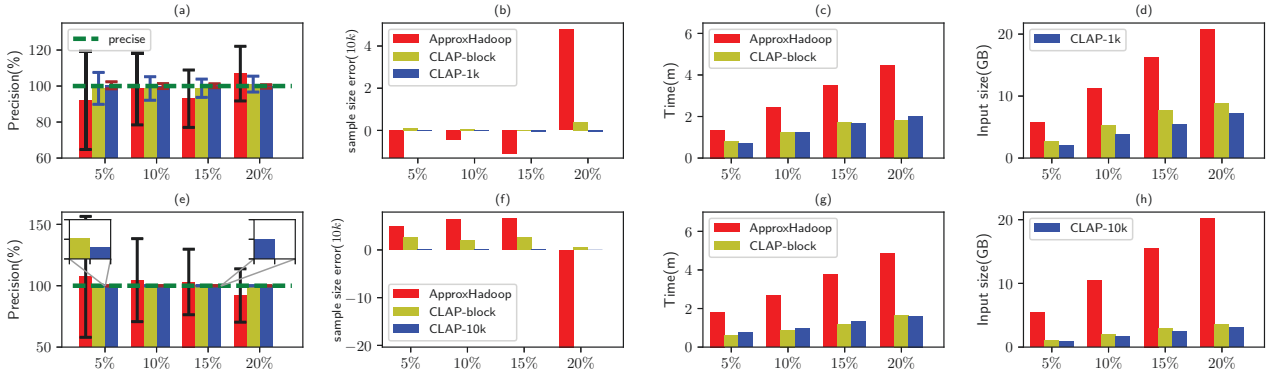


Fig. 15. Comparison with ApproxHadoop. ApproxHadoop can only use HDFS block as sampling unit while CLAP's sampling unit is configured as block and 1k for the Amazon review dataset, block and 10k for the TPC-H dataset. (a-d) show results on Amazon review dataset, (e-h) show results on TPC-H dataset.

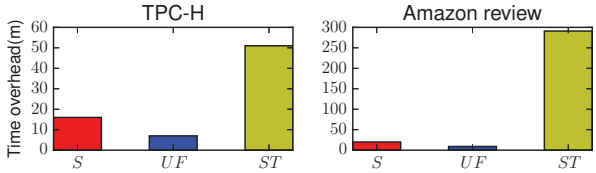


Fig. 16. Pre-processing time overhead comparison. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB.

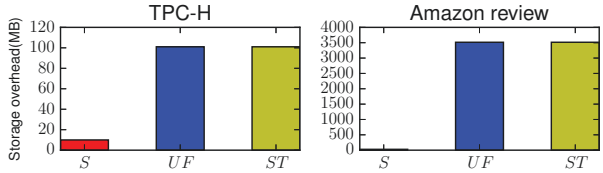


Fig. 17. Storage overhead comparison. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB.

and 3 times longer than those of CLAP on Amazon review and TPC-H datasets, respectively. This can be explained by the larger input sizes of ApproxHadoop shown in Figure 15(d) and (h), which indicates its inefficient sampling.

Finally, we conduct one more experiment on the Amazon review dataset to find out how much more data ApproxHadoop has to read to achieve the same confidence interval. We configure ApproxHadoop's sampling ratio to be 100%. This is an extreme case as reading the whole dataset will produce the precise result. However, we still assume that we are performing approximation and compute the confidence interval to make comparison with CLAP. The confidence interval it produced is about 1.66%. This is close to the produced interval of 1.36% when CLAP uses a sampling ratio of 10% with an input ratio of 3.2%. Therefore, ApproxHadoop needs to read $29\times$ more data to achieve the same error bounds as CLAP.

4.5 Comparison with BlinkDB

We conduct an end-to-end comparison with BlinkDB to identify the scenarios that Sapprox outperforms BlinkDB. Sapprox is not designed to replace BlinkDB but to be complementary to the systems like BlinkDB.

First, we compare the preprocessing time overhead. Blink-DB needs to create offline samples while Sapprox needs to build SegMap. The current BlinkDB implementation can create stratified samples for only one sub-dataset with one full scan of the whole dataset. It can also create uniform samples with one full scan of the whole dataset. Sapprox can build SegMap for all sub-datasets in the user specified columns using one full scan of the whole dataset.

Experimental settings: For the TPC-H dataset, Sapprox creates SegMap for all of the 7 sub-datasets under the "shipmode" column using a segment size of 10,000, while BlinkDB creates stratified samples for all of the 7 sub-dataset

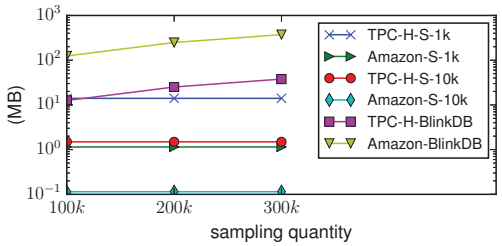


Fig. 18. Storage overhead comparison for one sub-dataset. S: Sapprox.

with a sample quantity cap of 100,000 for each sub-dataset. Blink-DB is also configured to create an uniform sample with the same storage budget ($7 \times 100,000$ samples). For the Amazon review dataset, Sapprox creates SegMap for all of the 33 sub-datasets under the "category" column using a segment size of 1,000, while BlinkDB creates stratified samples for all of the 33 sub-dataset with a sample quantity cap of 100,000 for each sub-dataset. BlinkDB also creates an uniform sample with the same storage budget ($33 \times 100,000$ samples). The reason why we choose 100,000 as the sampling quantity cap for a sub-dataset is that it is enough to produce an error under 1%.

Figure 16 shows the time overhead of building these offline samples and SegMap. The full scan time of Sapprox almost doubles that of BlinkDB. This is partially because that BlinkDB is implemented on top of Spark which utilize more memory space than Hadoop. Future implementation of Sapprox on Spark should be able to narrow the performance gap. On the other hand, the implementation SegMap job has the reducer phase, which incurs an extra shuffle phase relative to BlinkDB's sampling procedure. The size of the shuffle phase is the same as the size of the final SegMap. This extra shuffle is the main cause of the delay. However, creating stratified samples in BlinkDB induces multiple full scans of the whole dataset, which is determined by the num-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

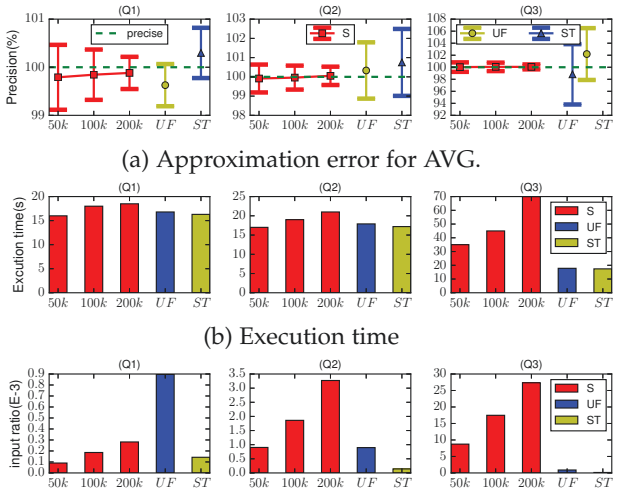


Fig. 19. Comparison results on TPC-H dataset. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB. (50k, 100k, 200k) on the x-axis are the corresponding sampling quantities of the increasing sampling ratios in Sapprox.

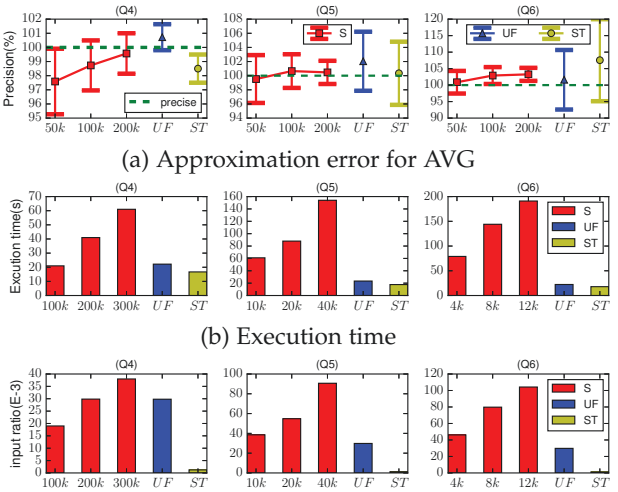


Fig. 20. Comparison results on Amazon review dataset. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB. (100k, 200k, 300k), (10k, 20k, 40k), (4k, 8k, 12k) on the x-axis are the corresponding sampling quantities of the increasing sampling ratios in Sapprox.

ber of sub-datasets to be sampled. This could potentially lead to a delay much longer than that of a single full scan in Sapprox,

Figure 17 shows the small storage overhead of Sapprox compared to BlinkDB with the above setting. On the TPC-H dataset, Sapprox's storage overhead is only about 0.01% of the whole dataset size while that of BlinkDB is about 0.09%. On the Amazon review dataset, Sapprox's storage overhead is only about 0.02% while that of BlinkDB is about 2.98%. BlinkDB consumes much more storage on the Amazon review dataset. This is because the size of each record in Amazon review dataset is about 10 times larger than that in the TPC-H dataset. The storage overhead of BlinkDB grows linearly with the record size and sampling ratio, while the storage overhead of Sapprox increases only with the total number of segments in a dataset. In order to understand these relationships, we compare the storage overhead of creating samples and SegMap for one sub-dataset in each of two datasets over different sampling quantities, as illustrated in Figure 18.

We continue to compare the approximation error and execution time of Sapprox and BlinkDB. The following six sets of queries with different queried columns in the WHERE clauses are evaluated on both systems. Notice that Sapprox only stores SegMap for sub-datasets in the "shipmode" and "category" columns, because the sub-datasets under other columns are almost uniformly distributed in the storage, which is examined by the Chi-square test introduced in Section 3.3.1.

```
--------------------TPC-H-------------------
Q1: shipmode=xx
Q2: shipmode=xx and discount=yy
Q3: shipmode=xx and discount=yy and tax=zz
---------------Amazon review---------------
Q4: category=xx
Q5: category=xx and rating=yy
Q6: category=xx and rating=yy and helpful=zz
-------------------------------------------
```

For each query set, we execute multiple queries and average the results. All the corresponding results are shown in Figure 19 and Figure 20. One major disadvantage of BlinkDB learned from experiments is that, given an offline sample, its lowest error and confidence interval are fixed. If the user desires a more accurate answer with a narrower confidence interval, the only option is generating a new offline sample with a larger size. Since the offline samples are stratified on the shipmode column in the TPC-H dataset and category column in the Amazon review dataset, for both Q1 and Q4, BlinkDB has exact matching stratified samples. However, for both Q2 and Q5, the offline stratified sample has a lower representativeness. Lastly, for Q3 and Q6, the representativeness of the offline stratified sample is the worst. No surprise, as shown in both Figure 19(a) and Figure 20(a), the approximation error and confidence interval increase dramatically from Q1 to Q3 and Q4 to Q6. For Q2 and Q3, if users of BlinkDB need more reliable answers, a new sample with a larger size is needed. However, generating a new sample requires a full scan of the whole dataset which incurs a comparable cost as getting a precise answer. Sapprox, on the other hand, can produce more accurate results by simply specifying a higher sampling ratio. Figure 19(a) and Figure 20(a) plot Sapprox's approximation results for Q1-Q6 with increasing sampling ratios. The x-axis labels are the corresponding sampling quantities of the increasing sampling ratios. The errors in Q2, Q3, Q4 and Q5 are much smaller than BlinkDB. As shown in Figure 19(b) and Figure 20(b), Sapprox does execute longer compared to BlinkDB as shown in Figure 11(b) and (e). The execution times of Sapprox and BlinkDB can be explained by the data input ratios shown in Figure 19(c) and Figure 20(c). For uniform sampling in BlinkDB, the inputs are the whole offline uniform samples, while for stratified sampling, the inputs are only one stratum of all the offline stratified samples. This explains why queries using stratified sampling has the smallest input size and shortest

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

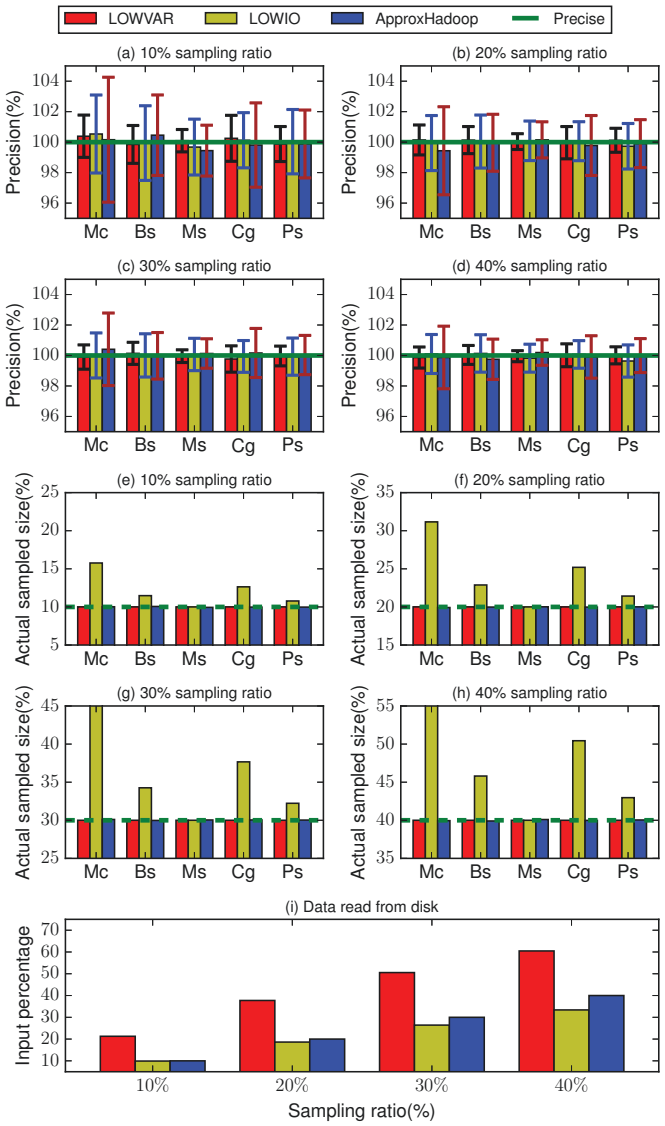


Fig. 21. Results of performing approximation simultaneously on 5 sub-datasets (Mc, Bs, Ms, Cg, Ps):(Music, Books, Movies, Clothing, Phones) in the Amazon review dataset.

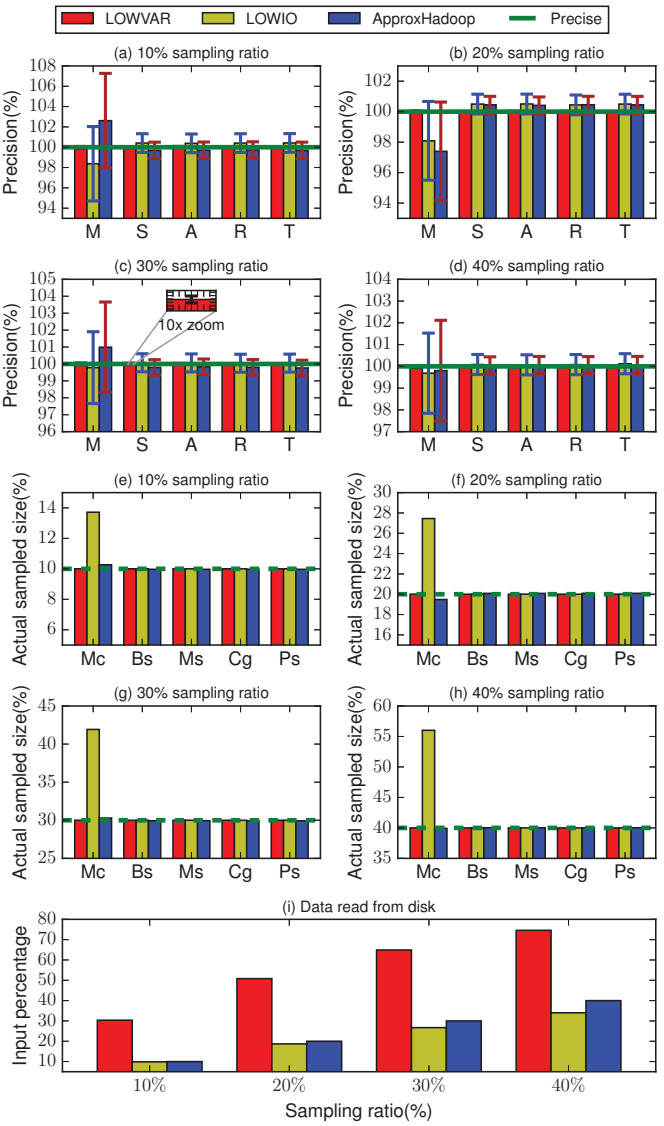


Fig. 22. Results of performing approximation simultaneously on 5 sub-datasets (M, S, A, R, T):(MAIL, SHIP, AIR, RAIL, TRUCK) in the TPC-H dataset.

execution time. For Sapprox, the input size grows with sampling ratio. Figure 19(c) shows that for the same sampling quantity, the input ratios of Sapprox increase from Q1 to Q3. This is because with more columns in the WHERE clause, the queried sub-dataset will have a smaller population ($population(Q3) > population(Q2) > population(Q1)$).

In summary, for queries that do not have good representative offline samples in systems like BlinkDB, Sapprox can deliver high accuracy results with extremely low storage overhead, at the cost of stretching the execution times a bit.

4.6 Results for performing approximation on multiple sub-datasets

For approximation on multiple sub-datasets, the two most important evaluation metrics are approximation error and the size of data read from disk. In the experiments, we will compare the performance of LOWIO, LOWVAR and ApproxHadoop on the two metrics. The three methods will employ the same segment size. We perform SUM approximation simultaneously on 5 sub-datasets (Music, Books, Movies, Clothing, Phones) in the Amazon review dataset and 5 sub-datasets (MAIL, SHIP, AIR, RAIL, TRUCK) in the TPC-H dataset, respectively.

Figure 21 (a-d) show the approximation errors and confidence intervals on the Amazon review dataset. LOWVAR has the smallest error and confidence interval as the inclusion probability π of each segment is computed in proportional to the sum τ associated with each segment. Additionally, CLAP's LOWIO has smaller error and confidence interval than that of ApproxHadoop. The reason is that though LOWIO relaxes the "proportional inclusion probability" property a little, it is still better than blindly assigning equal inclusion probability in ApproxHadoop. However, if the distributions of different sub-datasets are similar, then LOWIO and ApproxHadoop are expected to receive the same performance, which is shown in the error results for the 4 sub-datasets (SHIP, AIR, RAIL, TRUCK) in Figure 22 and their similar distribution in Figure 12.

The downside of LOWVAR in CLAP is that it needs to read much more data from disk than others. As shown

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

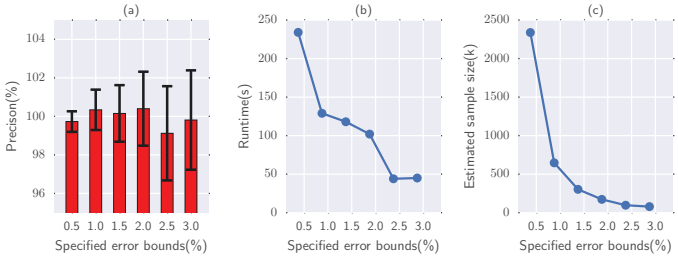


Fig. 23. Results for user specified error bound.

in Figure 21 (i) and Figure 22 (i), LOWVAR reads almost twice of data than LOWIO and ApproxHadoop. However, for an accuracy critical application, LOWVAR is still more efficient than others. According to sampling theory [20], the confidence interval is proportional to $\sim 1/\sqrt{n}$, where is n is the sample size. Therefore, according to the precision results, to achieve the same error bound, both LOWIO and ApproxHadoop need to read about $2\times$ more data than LOWVAR on the Amazon review dataset and $450\times$ more data than LOWVAR on the TPC-H dataset. On the other hand, LOWIO not only has smaller error bounds than ApproxHadoop but also reads less data from disk, which is about $(1\% \sim 7\%)$ less of the whole dataset.

From Figure 21 (e-h) and Figure 22 (e-h), we can see that LOWIO obtains much more samples for each sub-dataset than the other two methods while reading less data from disk. This lies in the fact that LOWIO pro-actively includes a segment into the sample if it has a larger contribution to the goal of achieving the sampling ratios for all queried sub-datasets. Therefore, for applications that treat sample size as the main consideration and error bounds as minor, LOWIO will be the best choice.

4.7 Results for user specified error bound

In this section, we evaluate CLAP's ability to achieve user specified error bounds. We use the music sub-dataset in the Amazon review dataset to do experiments. As shown in Figure 23(a), CLAP is able to satisfy all the specified error bounds. Figure 23(b) reports the corresponding runtime of each approximation job. For target errors 2.5% and 3.0%, the pilot job alone is enough to satisfy the target error bounds, so their runtime is the same, which is the execution time of the pilot job. For errors lower than 2.5%, the runtime is the sum of the execution time of the pilot job and real approximation job. It increases with larger estimated sample sizes. Figure 23 (c) shows the estimated sampled size by the pilot job. We can see that the relationship between user specified error bounds and estimated sample size aligns with the guide principle from sampling theory [20] that error bound is proportional to $\sim 1/\sqrt{n}$, where is n is the sample size.

5 Related Work

There has been many works [21, 23, 19] in traditional DBMS estimating the selectivity of a query against the whole dataset. Estimating the selectivity of relational join queries and the ones with composite predicates are especially challenging. Approaches fall into histogram-based and sampling based. The key component of CLAP is estimating the selectivity of a query against each segment rather than the whole dataset. CLAP can still potentially adopts existing methods. However, the selectivity estimation in CLAP is not about computing how many records satisfying the query as in existing work. CLAP is only interested in estimating the distribution of the selectivities across all the segments. Therefore, computing the number of satisfying records in each segment is not required, which allows CLAP to use a new probability based method to directly compute the ratios of selectivity between different segments.

Cluster sampling has been well explored in the traditional database literature. [14] explores the use of cluster sampling at the page level. Similar like ApproxHadoop, it does not address the sampling efficiency issues caused by arbitrary predicates in the query. In addition to BlinkDB and ApproxHadoop, another work that enables approximations in Hadoop is EARL [18]. Its pre-map sampling generates online uniform samples of the whole dataset by randomly reading input data at the line level. However, this will translate Hadoop's sequential read into a large number of random reads, which will degrade the performance of Hadoop. Similarly, in terms of sampling efficiency, it does not consider the skewness of sub-dataset distribution.

The most recent Quickr[16] targets at the single complex query that performs multiple passes over data. If data were sampled in one pass, all subsequent passes could be sped up. Hence Quickr focuses on what sampler to use and where to place the sampler in a query execution plan. However, their samplers need to read the whole dataset from disk for each new query. They do not take the I/O cost into consideration.

Some works do consider the skewed data distribution. Work in [31] attempts to reduce the sample size by taking into account the skewed distribution of attributes values. It is totally different from the storage distribution skewness of a sub-dataset. Another two works [25, 12] are in the area of online aggregation (OLA) [6] in Hadoop. As stated in both papers, the storage distribution skewness will break the randomness of samples. In order to keep strict randomness, the authors in [25] correlate the processing time of each block with the aggregation value. Its assumption is that if a block contains more relevant data, it will need more time to process. While in [12], it forces the outputs of mappers to be consumed by reducers in the same order as blocks are processed by mappers. However, both of them only correct the invalid randomness caused by the storage distribution skewness. They do not address the sampling inefficiency problem.

Lastly, most of these implementations need to change

the runtime semantics of Hadoop and therefore cannot be directly plugged into standard Hadoop clusters. CLAP requires no change of the current hadoop framework.

6 Conclusion

In this paper, we present CLAP to enable I/O efficient approximations on arbitrary sub-datasets in shared nothing distributed frameworks such as Hadoop. First, CLAP employs a probabilistic map called Seg-Map to capture the skewed storage distribution of sub-datasets. Second, we develop an online sampling method that is aware of this skewed distribution to efficiently sample data for sub-datasets in distributed file systems. We also quantify the optimal sampling unit size in distributed file systems. Third, we show how to use sampling theories to compute approximation results and error bounds in MapReduce-like systems. Finally, We have implemented CLAP into Hadoop as an example system and open sourced it on GitHub. Our comprehensive experimental results show that CLAP can significantly reduce application execution delay by up to one order of magnitude. Compared to existing systems, CLAP is more flexible than BlinkDB and more efficient than ApproxHadoop.

Acknowledgment

This project is supported in part by the US National Science Foundation Grant CCF-1337244, 1527249 and 1717388, and US Army/DURIP program W911NF-17-1-0208.

References

[1] Kafka. http://kafka.apache.org/.
[2] TPC-H benchmark. http://www.tpc.org/tpch/.
[3] F. Abdulla, M. Hossain, and M. Rahman. On the selection of samples in probability proportional to size sampling: Cumulative relative frequency method. *Mathematical Theory and Modeling*, 4(6):102–107, 2014.
[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.
[5] J. G. Booth and S. Sarkar. Monte carlo approximation of bootstrap variances. *The American Statistician*, 52(4):354–357, 1998.
[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1115–1118, New York, NY, USA, 2010. ACM.
[7] K. R. Das and A. R. Imon. Geometric median and its application in the identification of multiple outliers. *Journal of Applied Statistics*, 41(4):817–831, 2014.
[8] B. Efron. Nonparametric standard errors and confidence intervals. *canadian Journal of Statistics*, 9(2):139–158, 1981.
[9] B. Efron and B. Efron. *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM, 1982.
[10] S. Gabler, M. Ganninger, and P. Lahiri. A strictly positive estimator of intra-cluster correlation for the one-way random effects model. 2011.
[11] S. Gabler, S. Häder, and P. Lahiri. A model based justification of kish's formula for design effects for weighting and clustering. *Survey Methodology*, 25:105–106, 1999.
[12] Y. Gan, X. Meng, and Y. Shi. Processing online aggregation on skewed data in mapreduce. In *Proceedings of the Fifth International Workshop on Cloud Data Management*, CloudDB '13, pages 3–10, New York, NY, USA, 2013. ACM.
[13] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 383–397, New York, NY, USA, 2015. ACM.
[14] P. J. Haas and C. König. A bi-level bernoulli scheme for database sampling. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 275–286, New York, NY, USA, 2004. ACM.
[15] A. Java, X. Song, T. Finin, and B. Tseng. Why we twitter: Understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, WebKDD/SNA-KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.
[16] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, 2016. ACM.
[17] L. Kish. *Survey sampling*. John Wiley and Sons, 1965.
[18] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, June 2012.
[19] Y. Ling and W. Sun. A supplement to sampling-based methods for query size estimation in a database system. *ACM SIGMOD Record*, 21(4):12–15, 1992.
[20] S. Lohr. *Sampling: design and analysis*. Cengage Learning, 2009.
[21] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20(3):191–221, 1988.
[22] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 785–794, New York, NY, USA, 2015. ACM.
[23] J. K. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.
[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, Oct. 2014. USENIX Association.
[25] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11):1135–1145, 2011.
[26] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 587–598, New York, NY, USA, 2005. ACM.
[27] J. N. Rao and C. Wu. Resampling inference with complex survey data. *Journal of the american statistical association*, 83(401):231–241, 1988.
[28] P. E. Shrout and J. L. Fleiss. Intraclass correlations: uses in assessing rater reliability. *Psychological bulletin*, 86(2):420, 1979.
[29] J. Wang, J. Yin, J. Zhou, X. Zhang, and R. Wang. Datanet: A data distribution-aware method for sub-dataset analysis on distributed file systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 504–513. IEEE, 2016.
[30] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1071–1085, New York, NY, USA, 2016. ACM.
[31] Y. Yan, L. J. Chen, and Z. Zhang. Error-bounded sampling for analytics on big sparse data. *Proc. VLDB Endow.*, 7(13):1508–1519, Aug. 2014.
[32] F. Yates. Contingency tables involving small numbers and the χ 2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.
[33] J. Yin, Y. Liao, M. Baldi, L. Gao, and A. Nucci. A scalable distributed framework for efficient analytics on ordered datasets. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 131–138, Washington, DC, USA, 2013. IEEE Computer Society.
[34] X. Zhang, J. Wang, and J. Yin. Sapprox: enabling efficient and accurate approximations on sub-datasets with distribution-aware online sampling. *Proceedings of the VLDB Endowment*, 10(3):109–120, 2016.

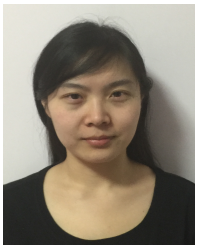
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2892765, IEEE Transactions on Parallel and Distributed Systems

Xuhong Zhang received his B.S. Degree in Software Engineering from Harbin Institute of Technology in 2011 and received his M.S. degree in Computer Science from Georgia State University in 2013. He is currently working toward his PhD degree in School of EECS at University of Central Florida, Orlando. His research interests include big data analytic and distributed storage file system.

Jun Wang is a full professor of Computer Science and Engineering, and director of the Computer Architecture and Storage Systems (CASS) Laboratory, the University of Central Florida, Orlando, FL, USA. He is the recipient of National Science Foundation Early Career Award 2009 and Department of Energy Early Career Principal Investigator Award 2005. He has authored more than 120 publications in premier journals such as the IEEE Transactions on Computers, the IEEE Transactions on Parallel and Distributed Systems, and leading HPC and systems conferences such as VLDB, HPDC, EuroSys, ICS, Middleware, FAST, IPDPS. He is a senior mem-ber of the IEEE.

Shouling Ji received the BS (Hons.) and MS degrees in computer science from Heilongjiang University, the PhD degree in computer science from Georgia State University, and the PhD degree in electrical and computer engineering from Georgia Institute of Technology. He is currently a ZJU 100-Young Professor with the College of Computer Science and Technology, Zhejiang University and a Research Faculty Member with the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include big data security and privacy, password security, and wireless networks. He is a member of IEEE and ACM and was the Membership Chair of the IEEE Student Branch at Georgia State University (2012-2013).

Jiangling Yin received his B.S. and M.S. degree in software engineering from the University of Macau in 2011. He received his Ph.D. degree in computer engineering from the Electrical Engineering and Computer Science Department, University of Central Florida in 2015. His research focuses on energy-efficiency computing and file/storage systems.



Rui Wang received her BS degree from Xian Jiaotong University City College, in 2011. She is pursuing a Master degree in computer engineering from Department of Electrical Engineering and Computer Science at University of Central Florida, since 2016. Her research interests include machine learning and approximate computing.

Xiaobo Zhou received the BS, MS, and PhD degrees in computer science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently, he is a professor and the chair of the Department of Computer Science, University of Colorado, Colorado Springs. His research lies broadly in computer network systems, specifically, cloud computing and datacenters, BigData parallel and distributed processing, autonomic and sustainable computing, and scalable Internet services and architectures. He received the NSF CAREER Award in 2009. He is a senior member of the IEEE.

Changjun Jiang received the PhD degree from the institute of Automation, Chinese Academy of Sciences, Beijing, China, in 1995. Currently he is a professor with the Key Laboratory of Embedded System and Service Computing, Tongji University, Shanghai. His current areas of research are concurrent theory, Petri net, and intelligent transportation systems. He is a member of the IEEE.