Sapprox: Enabling Efficient and Accurate Approximations on Sub-datasets with Distribution-aware Online Sampling

Xuhong Zhang, Jun Wang, Jiangling Yin
University of Central Florida
{xzhang, jwang, jyin}@eecs.ucf.edu

ABSTRACT

In this paper, we aim to enable both efficient and accurate 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 sampling efficiency and poor estimation accuracy on unevenly distributed sub-datasets.

To address the problem, we develop a distribution aware method called *Sapprox*. 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 facilitate online sampling. There are three thrusts in Sapprox. 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 sampling method for efficient online sub-dataset sampling. Third, we quantitatively derive the optimal sampling unit size in a distributed file system by associating it with approximation costs and accuracy. We have implemented Sapprox into Hadoop ecosystem as an example system and open sourced it on GitHub. Our comprehensive experimental results show that Sapprox can achieve a speedup by up to 20× over the precise execution.

1. INTRODUCTION

Despite the fact that today's computer clusters supply enormous data processing capacity, getting an ad-hoc query answer from a large scale dataset remains challenging. To attack the problem, recent years have seen a trend to promote approximate computing in big data analytic frameworks [12, 9, 4, 17, 14]. Approximate computing allows for faster execution on a much smaller sample of the original data by sacrificing a reasonable amount of accuracy. An approximation process often involves two basic phases: sample preparation and results estimation. How to prepare representative samples for the approximation jobs is essential to approximation systems. Take the following simple Hive [20] query as an example.

```
SELECT SUM(reviews)
FROM Movies
Where Type="Action"
```

The query is only on all of the "Action" movie records. In this paper, we define "the subset of data relevant to a set of attribute values" as *sub-dataset*. To prepare a good sample for this "Action" movie sub-dataset, we often employ *stratified sampling*, which first categorizes all the movie table records into several types of movie groups, and then randomly pick n records from the "Action" movie group. The most recent example is BlinkDB [4]. It creates stratified samples on the most frequently used "query column sets" (QCSs) in WHERE, GROUP BY, and HAVING clauses and caches them offline. For queries that match the sampled QCSs, it can compute results quite efficiently with high accuracy from the offline samples. Otherwise, it is possible that BlinkDB will generate answers with larger errors. To obtain a higher accuracy result, BlinkDB has to generate new offline samples for these queries. It may be noted that such operation is infeasible as it introduces a comparable cost as that of getting a precise answer. More importantly, it is prohibitive to generate offline samples for all sub-datasets. This lies in two facts: 1) the number of sub-datasets grows exponentially with an increasing number of columns in a table; 2) the storage space for caching these offline samples easily overwhelms current systems.

One possible solution is to enhance the BlinkDB-style systems by adding an online sampling function that can dynamically generate samples for arbitrary sub-datasets at runtime. The most recent online sampling based system is ApproxHadoop (ASPLOS 2015) [9], which is well recognized by the community. Its sampling and error estimation methods work fine with an assumption that sub-datasets are uniformly distributed in the whole dataset. Unfortunately, in many real-life cases, sub-datasets are actually spreaded unevenly over the partitions of a whole dataset, and sometimes in a very skewed fashion. A common phenomenon [11, 16, 21] is that, a small portion of the whole dataset contains most records of this sub-dataset, while other portions have few records belonging to this sub-dataset, as illustrated in Figure 1. Therefore, ApproxHadoop may sample a large

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 3
Copyright 2016 VLDB Endowment 2150-8097/16/11.

portion of the whole dataset, while obtain little sample data belonging to the queried sub-dataset. On the other hand, even if it collects enough samples of a sub-dataset, it may produce a result with a large variance [15]. In summary, the system could suffer from *inefficient sampling and large variance.*

Take estimating the total number of action movie records as an example. Suppose we know there are **200** action movie records out of 100,000 movie records. All the movie records are distributed in 250 blocks, and the distribution of the 200 action movie records over all the blocks follows a Zipf distribution, *e.g.* $\{137, 31, 11, 8, 4, 2, 1, 0, 1, ..., 1, 0\}$. A random procedure, which samples each block with equal probability, selects 25 out of the 250 blocks. Its simulation results show that there is about 80% probability that the number of action movies in the 25 sampled blocks is below **11**. As a result, for about 80% of the times, systems like ApproxHadoop will estimate the number of action movies below $11 \times \frac{\mathbf{250}}{\mathbf{25}} = \mathbf{110}$ and its variance as $[\mathbf{s}^2 \times (\frac{\mathbf{250}}{\mathbf{25}})^2]$, where s^2 is the variance of the number of action movie records in each of the sampled blocks. s^2 will also be large due to the skewed distribution.

Interestingly, we realize that one untapped method can be employing the sub-dataset distribution information to enforce both representative sampling and accurate estimation. More specifically, we developed a distribution-aware online sampling system called Sapprox. In Sapprox, 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. Sapprox samples units at a configurable segment level in contrast to the traditional HDFS block level. The effects of using a HDFS block as the default sampling unit is rarely explored. In Sapprox, 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 allows Sapprox to efficiently and effectively sample data for sub-datasets, and avoid accessing large amount of irrelevant data. Moreover, the different inclusion probabilities enable Sapprox to avoid over-representing or under-representing a segment unit when we compute the approximation result, and leads to a better accuracy. In a real world, Sapprox 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. Sapprox 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 Sapprox 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 Sapprox 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, Sapprox 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, Sapprox 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 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 sampling theories can be used to estimate sample size and compute error bounds for approximations in MapReduce-like systems.

- We implement Sapprox 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 cluster sampling. 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. A more efficient online sampling method is cluster sampling, which samples at cluster level. The cluster used in current systems [9, 17, 8] 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 2 shows the overal architecture of Sapprox. At the input stage of Hadoop, Sapprox will first retrieve the sub-dataset storage distribution information from our efficient

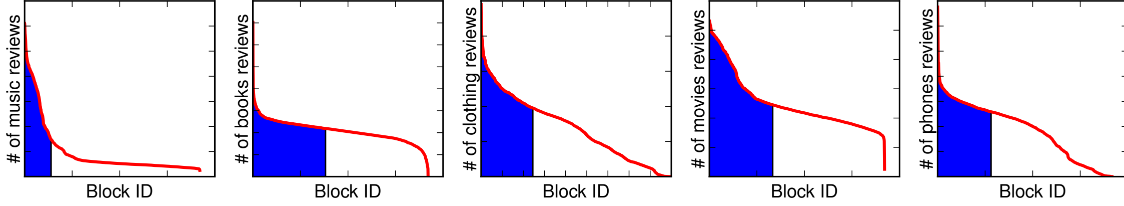


Figure 1: The storage distribution of sub-datasets in the Amazon review dataset. Shaded area accounts for 50% of a sub-dataset.

and flexible SegMap to estimate the inclusion probability of each segment. Section 3.2 introduces the estimation of inclusion probability and the creation, storage and retrieval of SegMap. According to the obtained inclusion probability, Sapprox 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 3.3. Section 3.4 introduces the implementation of the approximation Mapper and Reducer templates. In particular, we implement sampling theory into map and reduce tasks and calculate approximation answers and error bounds.

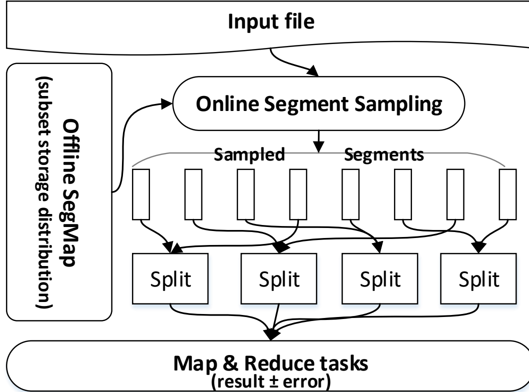


Figure 2: Sapprox architecture.

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 Sapprox, we define cluster as a list of consecutively stored records, referred to as ***segment***. The number of records in each segment is the same and can be an arbitrary integer. Section 3.5 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 [15]. 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. Taking 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, a simple observation is that τ_i is closely related to the number of relevant records in a segment.

3.2 Segment inclusion probability estimation

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

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

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:

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

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 combine these evidences, we use the geometric mean of the g ratios and calculate the ratio between $P_1^{'}(x)$ and $P_i^{'}(x)$ based on x as:

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

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.

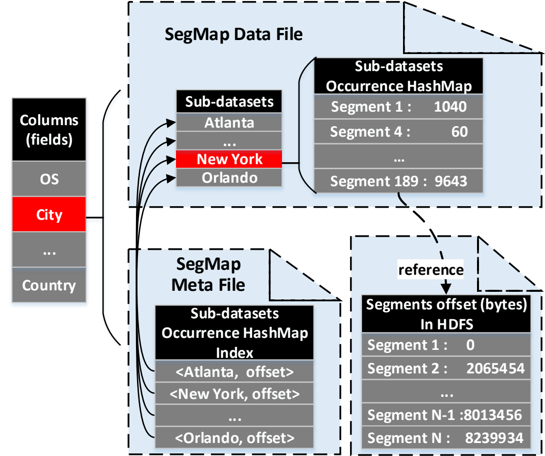


Figure 3: SegMap structure.

3.2.1 *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 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 3 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 Sapprox 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 HashMap objects 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 Figure 13.

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.

3.2.2 *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 [23]. This process is performed after the initial SegMap is created. With this test, the storage overhead of SegMap is greatly reduced.

3.3 Online input sampling

We implement the sampling stage in new *classes* of input parsing. For example, we implemented *SapproxTextInputFormat*, which is similar to Hadoop's *TextInputFormat*. Instead of using all the data blocks of a file to generate input splits, SapproxTextInputFormat will use a small list of file segments sampled from all the blocks according to a given sampling ratio or error. In SapproxTextInputFormat, 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. 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.

3.4 Approximation in Mapper and Reducer

Sapprox adopts standard closed form formulas from statistics [15] 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.

We use the approximation of SUM as an example, approximation for other aggregation functions are 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 [15] with unequal probability then allows us to estimate the sum τ from this sample as:

$$\widehat{\tau} = \frac{1}{n} \sum_{i=1}^{n} (\tau_i / \pi_i) \pm \epsilon \quad (10)$$

where the error bound ϵ is defined as:

$$\epsilon = t_{n-1,1-\alpha/2} \sqrt{\widehat{V}(\widehat{\tau})}, \;\; \widehat{V}(\widehat{\tau}) = \frac{1}{n} \frac{1}{n-1} \sum_{i=1}^{n} (\frac{\tau_i}{\pi_i} - \widehat{\tau})^2 \quad (11)$$

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. The reason why Sapprox 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 approximation Mapper and Reducer templates implement the above estimation of aggregation result and error bounds. Specifically, the Mapper 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 reducer, 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 (10) and (11). Figure 4 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 MySapproxApp{
 class MySapproxMapper extends SapproxMapper{
  void map(key, value){
      if(value belongs to sub-dataset x)
          context.write(x, v_ij);
  }
 }
 class MySapproxReducer extends SapproxReducer{
  void reducer(key, Iterator values){
      double sum;
      for value in values:
          sum+=value;
          context.write(key, sum);
  }
 }
 public static void main(){
  //job configurations
  //...
  setPartitionerClass(SapproxPartitioner);
  setInputFormatClass(SapproxTextInputFormat);
  run();
 }
/*****************Job submission*********************
Hadoop jar Sapprox.jar MySapproxApp -r 0.2 -w column=x
-r: sampling ratio
-w: WHERE clause
*****************************************************/
```

Figure 4: Example of developing a SUM approximation job on a sub-dataset x with Sapprox.

Sapprox 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 [14, 18]. Sapprox's booststrap based estimation is based on the theories introduced in [19, 7, 5]. For more details of the implementation and the corresponding evaluation results, please check our technical report [24].

We also implement an option in Sapprox to allow users to specify an error bound. More details can be found in our technical report [24].

3.5 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.

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

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

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

$$C = mc_1 + mM_0c_2 \tag{12}$$

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 (12), we will incorporate Kish's formula on cluster sampling [13]: $\widehat{V}(\widehat{\tau}) = \widehat{V}_{srs}(\widehat{\tau}) \times deff, deff = 1+(M_0-1)\rho$, 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.

$$\begin{aligned} \widehat{V}_{srs}(\widehat{\tau}) &= M^2 \times s^2/mM_0 \\ \widehat{V}(\widehat{\tau}) &= M^2s^2[1+(M_0-1)\rho]/mM_0 \\ m &= C/(c_1+M_0c_2) \\ C &= \frac{c_1M^2s^2(1+M_0c_2/c_1)[1+(M_0-1)\rho]}{M_0 \times \widehat{V}(\widehat{\tau})} \end{aligned} \tag{13}$$

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

$$M_0 = \sqrt{\frac{c_1}{c_2}\,\frac{1-\rho}{\rho}} \tag{14}$$

Equation (14) 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.

4. EVALUATION

4.1 Experimental setup

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

Table 2: Datasets.

Hardware. We evaluate Sapprox 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 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 $= \frac{approximation\ answer - precise\ answer}{precise\ answer}$.

99% confidence interval: *for 99% of times, the approximation 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-datasets with $|\phi| > 1$, π_i is estimated based on the occurrences of sub-datasets with $|\phi| = 1$ using conditional probability. Figure 5 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 5(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 6(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 6(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.

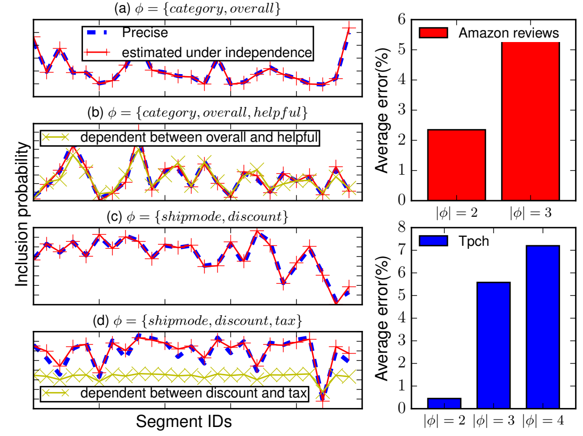


Figure 5: Accuracy validation of estimated inclusion probability. Columns examined in the two datasets: (category, overall, helpful, time) and (shipmode, discount, tax, returnflag, linestatus).

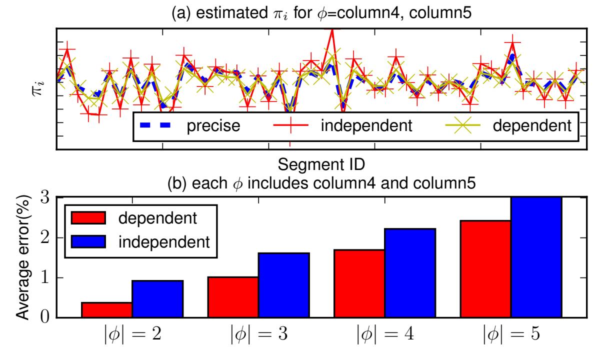


Figure 6: Accuracy comparison of estimating inclusion probability with dependence and independence. Experiments are on a new synthetic table with 5 columns. Column 5 is generated with dependency on column 4. Other columns are generated independently.

4.3 Approximation accuracy and efficiency

We pick five sub-datasets from each dataset to evaluate Sapprox'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 approxi-

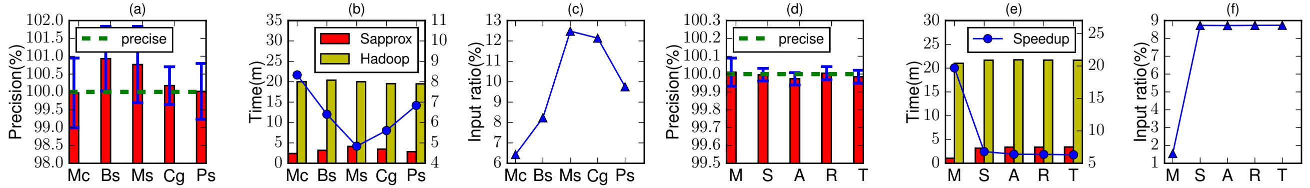


Figure 7: Approximation accuracy and efficiency of Sapprox on different sub-datasets. Results are normalized to precise result. The error bars are the 99% confidence intervals of approximation results. 20% and 10% records of each sub-dataset is sampled for Amazon and TPC-H datasets respectively. (Mc, Bs, Ms, Cg, Ps):(Music, Books, Movies, Clothing, Phones), (M, S, A, R, T):(MAIL, SHIP, AIR, RAIL, TRUCK).

mation applications evaluated are AVG on "Music, Movies, Clothing, MAIL, SHIP" sub-datasets and SUM on "Books, Phones, AIR, RAIL, TRUCK" sub-datasets.

Figure 7(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 7(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 7(c). For example, to sample 20% data of the music sub-dataset, Sapprox 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.

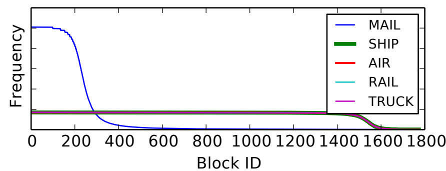


Figure 8: Storage distributions of sub-datasets in TPC-H dataset

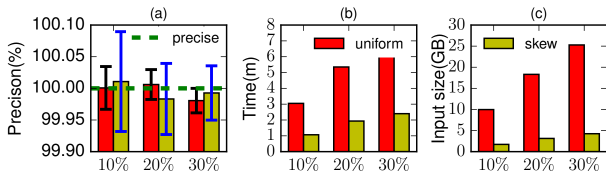


Figure 9: Effects of storage distribution skewness with different sampling ratios

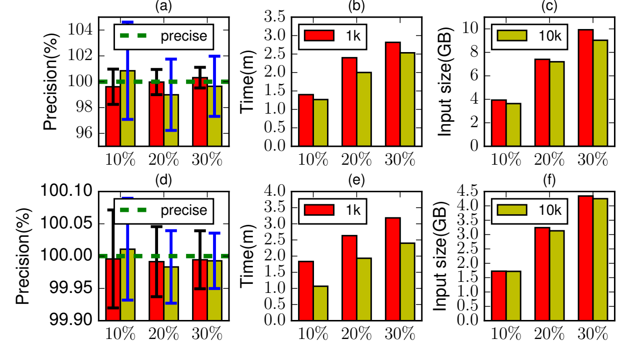


Figure 10: Effects of segment size with different sampling ratios. (a-c) show results on Amazon dataset, (d-f) show results on TPC-H dataset

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 8(a).

4.4 Effects of storage distribution skewness

We first study how the skewness of sub-dataset storage distribution affects the approximation error and runtime. The "MAIL" sub-dataset in TPC-H dataset is distributed with zipf distribution as shown in Figure 8, and for the same dataset, we also distribute it with uniform (even) distribution. Figure 9 shows the approximation results on the two distributions. From Figure 9(a), we can see that the average error and confidence interval are both smaller on the uniform distribution. This can be explained by the input sizes shown in Figure 9(c). For an uniform distribution, Sapprox has to read almost the same percentage of the whole dataset as the sampling ratio to meet the sampling ratio requirement. However, for a skewed distribution, Sapprox reads much less data of the whole dataset due to the storage concentration of the MAIL sub-dataset. A larger input size indicates a larger number of sampled segments. For the same sampling ratio, more sampled segments will improve the variety of samples, resulting in more accurate estimation. Accordingly, with greater input size, the execution time on the uniformly distributed sub-datasets is much longer.

Figure 10 shows the comparison results with two different segment sizes. The approximation error results on both

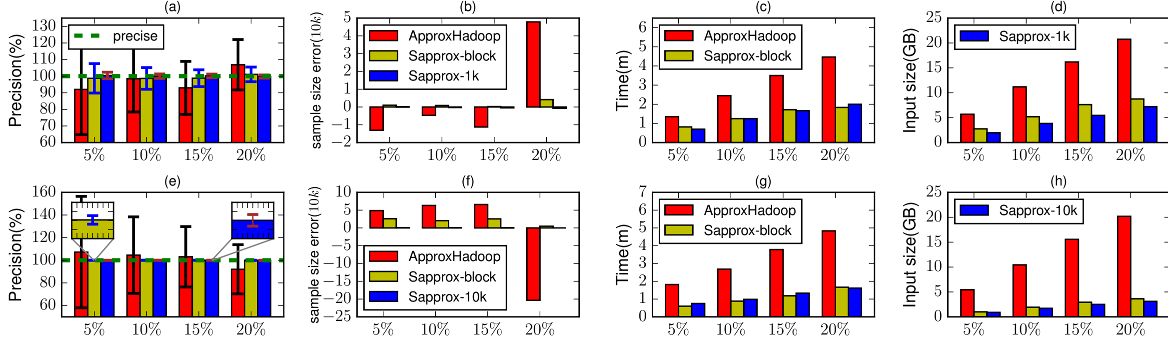


Figure 11: Comparison with ApproxHadoop. ApproxHadoop can only use HDFS block as sampling unit while Sapprox's sampling unit is configured as block and 1k for the Amazon review dataset, block and 10k for the TPC-H dataset. (a-d) show results on Amazon review dataset, (e-h) show results on TPC-H dataset.

datasets confirm that a smaller segment size will produce a more accurate estimation due to the larger variety of sampled segments. However, as shown in both Figure 10(b,e), the execution time with a smaller segment size will be longer. The reason is that for the same input size, there will be more random reads with a smaller segment, resulting in larger I/O overhead. One lesson learned from the results on TPC-H dataset is that, if the estimation error is already small enough, continuing to decease the segment size is not a wise choice. This is because the gain on estimation accuracy is very small relative to the sacrifice in execution delay.

4.5 Comparison with ApproxHadoop

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

Figure 11 reports the approximation accuracy and execution time comparison results. As shown in both Figure 11(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 Sapprox with block unit and 16 times wider than Sapprox 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 [9]: $\hat{Var}(\hat{\tau}) = N(N-n)\frac{s^2}{n}$, 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 Sapprox, 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 Sapprox. To explain this, we record the actual number of records sampled for a sampling ratio in both systems. In Figure 11(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 Sapprox 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 11(c) and (g) show the execution time comparison results. The execution times of ApproxHadoop are 2 times and 3 times longer than those of Sapprox on Amazon review and TPC-H datasets, respectively. This can be explained by the larger input sizes of ApproxHadoop shown in Figure 11(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 Sapprox. The confidence interval it produced is about 1.66%. This is close to the produced interval of 1.36% when Sapprox uses a sampling ratio of 10% with an input ratio of 3.2%. Therefore, ApproxHadoop needs to read 29× more data to achieve the same error bounds as Sapprox.

4.6 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.

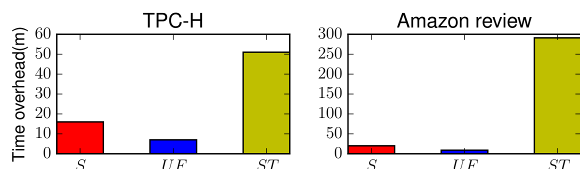


Figure 12: Pre-processing time overhead comparison. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB.

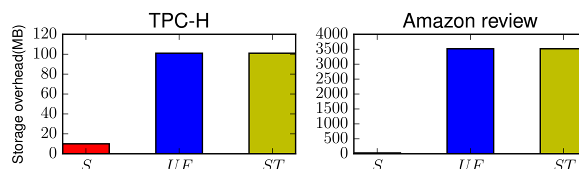


Figure 13: Storage overhead comparison. S: Sapprox, UF: uniform sampling in BlinkDB, ST: stratified sampling in BlinkDB.

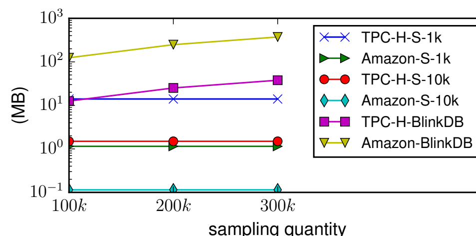


Figure 14: Storage overhead comparison for one sub-dataset. S: Sapprox.

First, we compare the preprocessing time overhead. BlinkDB 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. The command is shown below:

```
CREATE TABLE xxx_sample AS
SELECT * FROM xxx Where xxx.column=yyy
SAMPLEWITH ratio
```

BlinkDB 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:

```
Hadoop jar Sapprox.jar SegMap -c 1-2-3-4 -s 1000
-c column indexes to create SegMap
-s segment size
```

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 with a sample quantity cap of 100,000 for each sub-dataset. BlinkDB 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%.

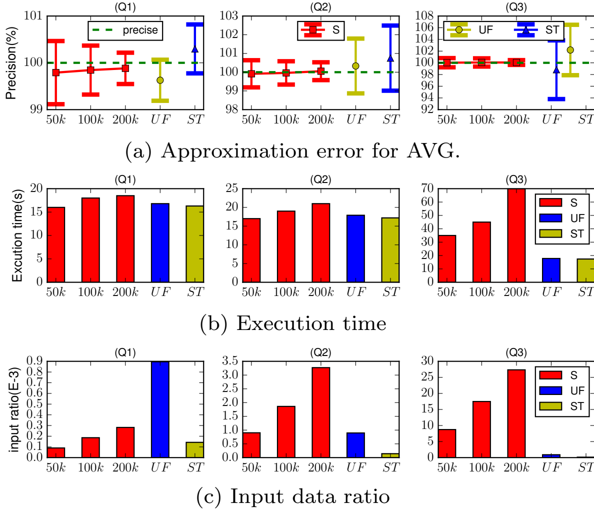
Figure 12 shows the time overhead of building these offline samples and SegMap. The full scan time of Sapprox almost doubles that of BlinkDB. This is partially because that BlinkDB is implemented on top of Spark which utilize more memory space than Hadoop. Future implementation of Sapprox on Spark should 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 number 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 13 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 14.

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 3.2.2.

```
--------------------TPC-H------------------------
Q1: WHERE shipmode=xx
Q2: WHERE shipmode=xx and discount=yy
Q3: WHERE shipmode=xx and discount=yy and tax=zz
----------------Amazon review--------------------
Q4: WHERE category=xx
Q5: WHERE category=xx and rating=yy
Q6: WHERE 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 15 and Figure 16. 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

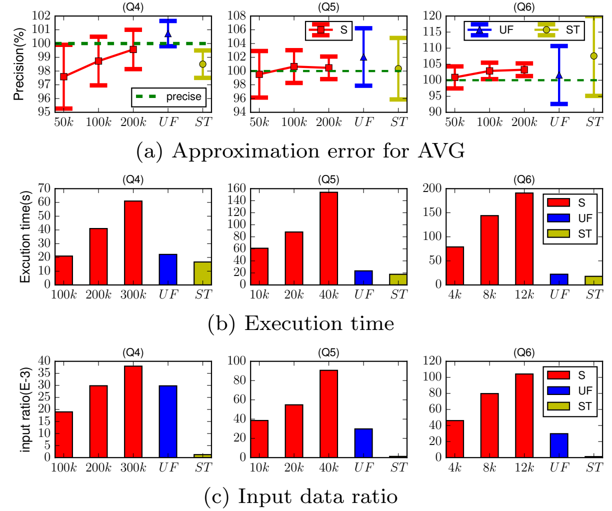


(a) Approximation error for AVG.

(b) Execution time

(c) Input data ratio

Figure 15: 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.



(a) Approximation error for AVG.

(b) Execution time

(c) Input data ratio

Figure 16: 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.

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 15(a) and Figure 16(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 15(a) and Figure 16(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 15(b) and Figure 16(b), Sapprox does execute longer compared to BlinkDB as shown in Figure 7(b) and (e). The execution times of Sapprox and BlinkDB can be explained by the data input ratios shown in Figure 15(c) and Figure 16(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 execution time. For Sapprox, the input size grows with sampling ratio. Figure 15(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.

5. RELATED WORK

Cluster sampling has been well explored in the traditional database literature. [10] 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 [14]. 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[12] 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 [22] 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 [17, 8] 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 [17] 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 [8], 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. Sapprox requires no change of the current hadoop framework.

6. CONCLUSION

In this paper, we present Sapprox to enable both efficient and accurate approximations on arbitrary sub-datasets in shared nothing frameworks such as Hadoop. First, Sapprox employs a probabilistic map called SegMap 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 Sapprox into Hadoop ecosystem as an example system and open sourced it on GitHub. Our comprehensive experimental results show that Sapprox can significantly reduce application execution delay by up to one order of magnitude. Compared to existing systems, Sapprox is more flexible than BlinkDB and more efficient than ApproxHadoop.

7. ACKNOWLEDGMENT

A special thanks goes to Shouling Ji from Zhejiang University and Xunchao Chen, Ruijun Wang, Dan Huang from University of Central Florida for their significant help. This work is supported in part by the US National Science Foundation Grant CCF-1337244, CCF-1527249, and National Science Foundation Early Career Award 0953946. This work is also supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE).

8. 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] B. Efron and B. Efron. *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM, 1982.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] L. Kish. *Survey sampling*. John Wiley and Sons, 1965.

[14] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, June 2012.

[15] S. Lohr. *Sampling: design and analysis*. Cengage Learning, 2009.

[16] 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.

[17] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11):1135–1145, 2011.

[18] 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.

[19] J. N. Rao and C. Wu. Resampling inference with complex survey data. *Journal of the american statistical association*, 83(401):231–241, 1988.

[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[21] 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.

[22] 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.

[23] 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.

[24] X. Zhang, J. Wang, J. Yin, R. Wang, X. Chen, and D. Huang. Sapprox technical report. `http://www.cass.eecs.ucf.edu/?page_id=114`, 2016.