

Approximation with Error Bounds in Spark

Guangyan Hu
Rutgers University
New Brunswick, NJ
gh279@cs.rutgers.edu

Sandro Rigo
University of Campinas
Campinas - SP, Brazil
srigo@unicamp.br

Desheng Zhang
Rutgers University
New Brunswick, NJ
d.z@rutgers.edu

Thu D. Nguyen
Rutgers University
New Brunswick, NJ
tdnguyen@cs.rutgers.edu

Abstract—Many decision-making queries are based on aggregating massive amounts of data, where sampling is an important approximation technique for reducing execution times. It is important to estimate error bounds when sampling to help users balance between accuracy and performance. However, error bound estimation is challenging because data processing pipelines often transform the input dataset in complex ways before computing the final aggregated values. In this paper, we introduce a sampling framework to support approximate computing with estimated error bounds in Spark. Our framework allows sampling to be performed at multiple arbitrary points within a sequence of transformations preceding an aggregation operation. The framework constructs a data provenance tree to maintain information about how transformations are clustering output data items to be aggregated. It then uses the tree and multi-stage sampling theories to compute the approximate aggregate values and corresponding error bounds. When information about output keys are available early, the framework can also use adaptive stratified reservoir sampling to avoid (or reduce) key losses in the final output and to achieve more consistent error bounds across popular and rare keys. Finally, the framework includes an algorithm to dynamically choose sampling rates to meet user-specified constraints on the CDF of error bounds in the outputs. We have implemented a prototype of our framework called *ApproxSpark* and used it to implement five approximate applications from different domains. Evaluation results show that *ApproxSpark* can (a) significantly reduce execution time if users can tolerate small amounts of uncertainties and, in many cases, loss of rare keys, and (b) automatically find sampling rates to meet user-specified constraints on error bounds. We also explore and discuss extensively tradeoffs between sampling rates, execution time, accuracy and key loss.

Index Terms—Spark, approximation, data provenance, multi-stage sampling, stratified sampling

I. INTRODUCTION

Data-driven discovery and decision support have become critical to the missions of many businesses, scientific and government enterprises. At the same time, the rate of data production and collection is outpacing technology scaling, implying that significant future investment, time, and energy will be needed for data processing [1], [2]. Approximate computing is a powerful tool to reduce these processing needs. Many data analytic applications such as data mining, log processing, and data visualization are amenable to approximation [3]. As a concrete example, suppose a company wants to know the age distribution of its customers for a particular product. In such an application, estimated counts derived from data samples may be sufficient, allowing tradeoffs between precision and processing time, energy consumption and/or cost.

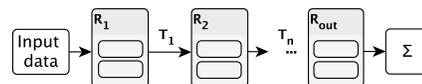


Fig. 1: A Spark computation having a chain of transformations ($\{T\}$), where each box in an RDD is a partition.

In this paper, we propose a framework for creating and running approximate Spark programs that use online sampling to efficiently aggregate massive amounts of data. The framework computes error bounds (i.e., confidence intervals) along with the approximate aggregate values. We focus on aggregation because many decision support tasks require aggregation queries: e.g., a study of a Microsoft SCOPE [4] data processing cluster reveals that 90% of 2,000 data mining jobs were aggregations [5]. Aggregation is also an important component in online analytical (OLAP) systems for summarizing data patterns in business intelligence [6], [7].

Spark is a popular data processing system that has been widely adopted in different domains [8]–[11]. Thus, embedding a general approximation framework in Spark will make approximation easily accessible to application developers in many different fields. In addition, while our work is specific to Spark, it should also be portable to other similar data processing systems.

Estimating error bounds is important, especially for decision support queries, because it allows users to intelligently balance precision and performance. However, Spark programs (and data processing pipelines in general) often include multiple complex transformations of the input data before the final aggregation [12], [13], making it challenging to compute error bounds. Consider a Spark computation comprising of a chain of transformations ending with a summation as shown in Figure 1. If we sample data items in the resilient distributed dataset (RDD) R_{out} immediately before the aggregation, then it is straightforward to use simple random sampling (SRS) theories to estimate the sums with error bounds [14]. However, this sampling is unlikely to reduce execution time by much since the additions saved are relatively inexpensive.

Alternatively, we can view each partition of R_{out} as a cluster and apply cluster sampling. We can then use two-stage cluster sampling theories for estimating sums and error bounds [14], although we would need to estimate populations in multi-key computations (see the discussion on multi-stage sampling in multi-key computations below). This can lead to much greater

execution time savings since we can avoid performing *all* of the transformations on the dropped partitions. Unfortunately, this locks the computation into a very coarse-grained sampling process that may not be tunable to achieve the desired tradeoff between precision and performance.

A natural solution is to sample earlier, e.g., sample when creating R_1 from the input data, where we can use a combination of dropping partitions and data items to achieve the right balance between precision and performance. As we discuss in Section III, a key insight behind our work is that it is possible to map such a sampling process to a multi-stage sampling process on R_{out} , and use the accompanying theories to compute the estimated aggregate values and error bounds.

As a concrete example, consider a program to count word occurrences in a text dataset, where a `map` parses each sentence and produces a list of `(word, 1)` pairs, and a subsequent `flatMap` breaks the lists to produce the final set of `(word, 1)` pairs, followed by summing the count of each unique word. In this computation, there are two levels of clustering, with each partition of the input being a cluster of sentences, and each sentence a cluster of words. Therefore, when sampling at the creation of R_1 by selecting a random subset of partitions and a random subset of sentences from each selected partition, the sampling errors need to be estimated using three-stage cluster sampling theories since the end populations are actually words rather than the sentences. The population size of each word also has to be estimated from its sample size given the sampling rate over the sentences, because if a sentence is not chosen for the sample, then it is unknown whether that dropped sentence would have produced counts for a particular word.

In Section III, we first explain how sampling at multiple arbitrary points within a sequence of transformations can be mapped to a multi-stage sampling process on the output RDD. We then propose an algorithm to build a *data provenance tree* to maintain information about this mapping. Finally, we propose another algorithm to extract information from the tree, estimate populations where needed, and compute the final approximate aggregate values and their error bounds. Critically, we show how to account for the imprecision introduced by population estimation. If the final keys are known early in the transformation sequence, we show how adaptive stratified reservoir sampling (ASRS) [15] can be integrated with multi-stage sampling to avoid losing rare keys, as well as balancing the sampling errors between popular and rare keys (Section IV).

We have implemented the proposed framework in a prototype system called ApproxSpark (Section V). Our framework supports a subset of common Spark transformations, including `map`, `flatMap`, `mapValues`, `sample` and `filter`, and aggregation operations `mean` and `sum`. When running an approximate computation, users have the flexibility to specify sampling rates or constraints on the CDF of relative error bounds for values associated with output keys—if the computation produces a single value or key-value pair, then the latter reduces to just the maximum allowable relative error bound.

When the user specifies constraints for the error bound CDF, ApproxSpark will run pilot executions of several partitions and use the results to select appropriate sampling rates.

We have used ApproxSpark to implement five approximate applications from different domains in text mining, graph analysis, and log analysis. We use the applications to evaluate ApproxSpark and explore the tradeoffs between performance and accuracy/precision. Among other findings, our results show that (i) ApproxSpark can significantly reduce execution time if users can tolerate small amounts of uncertainties and, in many cases, loss of rare keys; (ii) it is possible to automatically find sampling rates to meet user-specified constraints on the CDF of error bounds in the output; (iii) partition sampling can lead to greater reduction in execution time than data item sampling, but lead to more key loss and significantly larger error bounds, especially for the rarer keys; and (iv) ASRS with multi-stage sampling avoids or reduces key loss and leads to more consistent error bounds across keys.

In summary, our contributions include: (i) to our knowledge, our work is the first to apply multi-stage sampling theories to estimate aggregate values and error bounds when sampling within arbitrarily long sequences of transformations; (ii) we introduce algorithms for maintaining provenance information during the execution of the transformations and computing the approximate aggregate values and error bounds; (iii) we show how ASRS can be combined with multi-stage sampling for some applications to reduce key loss and equalize error bounds across popular and rare keys; (iv) we explore extensively the tradeoffs between sampling rates, execution time, key loss, and error bounds; and (v) we present an algorithm for automatically choosing sampling rates to meet user-specified constraints on the CDF of error bounds for output values.

II. BACKGROUND AND RELATED WORK

Spark. Spark has emerged as a popular distributed data processing engine. Spark introduces RDDs, which are fault-tolerant collections of data partitioned across server clusters that can be processed in parallel [16]. Spark has two types of operations: transformations and actions. A transformation is a lazy operation that produces an output RDD from an input RDD, where as an action computes non-RDD values from an input RDD, and triggers preceding transformations needed to produce the input RDD. Data items in RDDs can be key/value pairs, such that a Spark program may be computing a number of different aggregations in parallel. The word counting program in Section I is a good example. It is counting potentially many different unique words at once, computing an aggregation for each word.

Spark already contains random and stratified sampling transformations with several important limitations. First, there is no support for computing error bounds, especially across a sequence of multiple transformations. Second, stratified sampling can still lose some keys, because it adopts Bernoulli Sampling. Third, sampling is only implemented on existing RDDs, so that the entire input dataset has to be loaded before sampling can be applied.

Approximate query processing (AQP). A variety of approximation techniques have been employed by query processing systems to reduce execution time. These techniques include using random or stratified sampling to construct samples to provide bounded errors [5], [17]–[22] or online aggregation to sample data and produce a result within a time-bound [23], [24]. BlinkDB [21] maintains a set of offline-generated stratified samples by using an error-latency profile based on past queries. Sapprox [22] collects the occurrences of sub-datasets in offline preprocessing and uses it to drive online sampling. Many AQP systems use offline processing under the assumption that data will be used repeatedly. Online sampling is an efficient approximation method when the large dataset (e.g., logs) will be used only once or a few times [25].

Online sampling. ApproxHadoop [25] introduces approximation to the MapReduce paradigm [26]. It uses multi-stage sampling to trade off precision and performance similar to ApproxSpark (we discuss differences below). Users can specify sampling rates or a target maximum relative error. StreamApprox [27] approximates stream processing workloads based on Spark Streaming [28]. MaRSOS [19] is related to ApproxHadoop but proposes a stratified sampling algorithm to avoid losing keys and balance error bounds for popular and rare keys. Compared to MaRSOS, ApproxSpark’s implementation of stratified sampling using ASRS avoids the overheads of coordination between parallel tasks while still being able to balance error bounds.

Comparison with ApproxHadoop. While ApproxSpark and ApproxHadoop both use multi-stage sampling, there are important differences. First, ApproxSpark generalizes multi-stage sampling to handle sequences of transformations with arbitrary lengths, allowing sampling anywhere within the sequences, whereas ApproxHadoop is limited to using two- and three-stage sampling to handle a single map phase in MapReduce computations. Second, ApproxHadoop also relies on population estimation but does not account for the added uncertainties; ApproxSpark does. ApproxSpark implements ASRS to avoid losing keys and balance error bounds when output keys are known early in the computation. Finally, in this paper, we explore the rich space of tradeoffs between sampling rates, execution times, error bound distributions across all output keys, and loss of rare keys far beyond what was considered in the ApproxHadoop study [25].

III. MULTI-STAGE SAMPLING IN SPARK

Suppose we have a simple Spark program that reads a set of values into an RDD R_{in} and sums the values. We can reduce the execution time of this computation by (1) reading only a randomly selected subset of input partitions, (2) load a randomly selected subset of data items from each selected partition into R_{in} , and (3) compute an estimated sum $\hat{\tau}$ and its variance \hat{V} , which is needed for computing confidence intervals around $\hat{\tau}$, using two-stage cluster sampling theories as follows [14]:

$$\hat{\tau} = \frac{N}{n} \sum_{i=1}^n \left(\frac{M_i}{m_i} \sum_{j=1}^{m_i} v_{ij} \right) \quad (1)$$

$$\hat{V}(\hat{\tau}) = N(N-n) \frac{S_u^2}{n} + \frac{N}{n} \sum_{i=1}^n M_i (M_i - m_i) \frac{S_i^2}{m_i} \quad (2)$$

where N is the total number of partitions in the input data set, n is the number of selected partitions, M_i is the total number of values in partition i of the input data set, m_i is the number of values selected from partition i and loaded into R_{in} , v_{ij} is the j^{th} value from partition i in R_{in} , S_i^2 is the intra-cluster variance for partition i , and S_u^2 is the inter-cluster variance. Note that N and M_i ’s are attributes of the input data set, while n and m_i ’s are attributes of the sample. S_u^2 and S_i^2 are both computed using the sample.

Now consider a program where R_{in} is transformed by a sequence of transformation T_0, T_1, \dots, T_n to produce R_{out} , which is then summed. If each transformation T_i is a one-to-one mapping of an input value to a single output value (e.g., a Spark `map` operation), such that R_{in} , R_{out} and all intermediate RDDs contain the same number of data items, then it is possible to sample the input data when creating R_{in} in the same manner as above and still use the estimators given in Equations 1 and 2. Sampling the input data is exactly equivalent to sampling the R_{out} that would have been produced by processing the entire input dataset.

Spark, however, includes transformations that map input items to output items in more complex ways than one-to-one. As already mentioned, this complexity makes it much more challenging to compute error bounds when sampling early within a Spark computation. In the remainder of this section, we first show how generalized multi-stage sampling theories can be used when sampling at multiple different points within a Spark program. We then describe two algorithms necessary to track the multi-level clustering of data items in R_{out} as the input data is transformed, and to use the tracking information to estimate the aggregate values and error bounds. We discuss summation, but the discussion is equally applicable to average.

A. Multi-stage Sampling

Consider the Spark program and its execution as shown in Figure 2. The `flatMap` transformation can generate multiple output items for each input item, corresponding to a one-to-many mapping. An example is the generation of the two data items $c_2:e_1$ and $c_2:e_2$ in R_2 from the single data item c_2 from R_1 . In this case, when sampling, selecting an input data item to load into R_1 is equivalent to selecting a cluster of items from R_2 , and selecting a partition from the input data set is equivalent to selecting a cluster of clusters from the R_2 . This corresponds to a three-stage sampling process. In fact, general multi-stage sampling and population estimation can be used to handle Spark programs comprised of a subset of common transformations for both single- and multi-key computations.

Below, we generalize the two-stage sampling equations (Eq (1) and (2)) into recurrences for multi-stage sampling

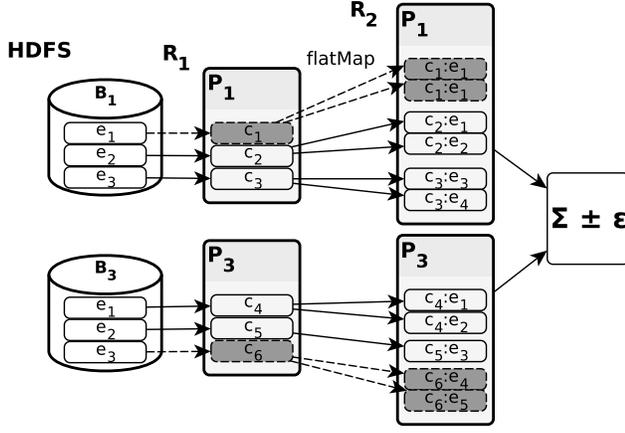


Fig. 2: HDFS blocks and input data items are sampled when read into RDD R_1 . Block B_2 not shown has been dropped. Gray boxes are dropped data items. In R_2 , $c_i : e_j$ means data item j is generated from the data item i in the input partition.

with estimated sum and variance. We use $I_k = i_0, i_1, \dots, i_k$ to denote the index of a specific cluster at level k . Note that in a multi-key computation, a sample is chosen for each key, so we will need to estimate the sum and variance for each key.

Sum estimation. We estimate the sum of a multi-stage sample with d sampling stages using the following recurrence:

$$\hat{\tau}_{I_k} = \begin{cases} \frac{N_{I_k}}{n_{I_k}} \sum_{j=1}^{n_{I_k}} \hat{\tau}_{I_k, j} & 0 \leq k < d, \\ v_{I_k} & k = d \end{cases} \quad (3)$$

where $\hat{\tau}_{I_k}$ is the estimated sum of cluster I_k (at level k), N_{I_k} is the total number of sub-clusters of cluster I_k , n_{I_k} is the number of sub-clusters chosen from cluster I_k , I_k, j is the index i_0, i_1, \dots, i_k, j such that $\hat{\tau}_{I_k, j}$ is the estimated sum of a sub-cluster of cluster I_k , and v_{I_k} is the value in the sample (at the last level $k = d$) with index I_k . The 0^{th} stage contains just one cluster comprising the entire population, so $\hat{\tau}_0$ is then the overall estimated sum.

Variance estimation. Similarly, we estimate the variance using the recurrence:

$$\hat{V}(\hat{\tau}_{I_k}) = \begin{cases} N_{I_k}(N_{I_k} - n_{I_k}) \frac{S_{u, I_k}^2}{n_{I_k}} + \frac{N_{I_k}}{n_{I_k}} \sum_{j=1}^{n_{I_k}} \hat{V}(\hat{\tau}_{I_k, j}) & 0 \leq k < d - 1, \\ M_{I_k}(M_{I_k} - m_{I_k}) \frac{S_{i, I_k}^2}{m_{I_k}} & k = d - 1 \end{cases} \quad (4)$$

where $\hat{V}(\hat{\tau}_{I_k})$ is the variance of $\hat{\tau}_{I_k}$, S_{u, I_k}^2 is the inter-cluster variance of the sub-clusters of cluster I_k , M_{I_k} is the total number of values in cluster I_k , m_{I_k} is the number of values from cluster I_k in the sample, and S_{i, I_k}^2 is the intra-cluster variance of cluster I_k . $V(\hat{\tau}_0)$ is then the overall estimated variance.

Confidence interval. Given the above estimated sum and variance, we can compute the confidence interval as: $\hat{\tau}_0 \pm \epsilon$,

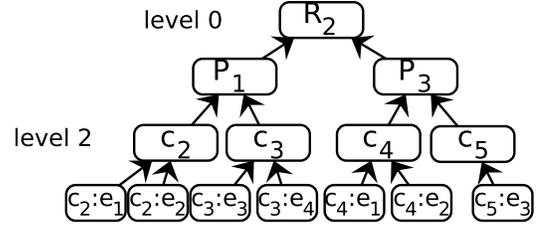


Fig. 3: An equivalent tree to the transformation chain shown in Figure 2, where R_2 is the final sample at level 0.

Transformation	Semantics
map (func)	Applies <i>func</i> to each data item (di).
flatMap (func)	Applies <i>func</i> to each di and flatten.
mapValues (func)	Applies <i>func</i> only to the value of each di.
filter (func)	Selects di's that satisfy a predicate <i>func</i> .
sample (r)	samples di's using sampling rate <i>r</i> .

TABLE I: Spark transformations that can be approximated by ApproxSpark and their semantics. A transformation generates a new RDD from a source RDD.

where $\epsilon = t_{n-1, 1-\alpha/2} \sqrt{\hat{V}(\hat{\tau}_0)}$, $t_{n-1, 1-\alpha/2}$ is the critical value under the Student's t distribution at the desired level of confidence α , and n is the degree of freedom (i.e., the number of chosen clusters at level 1) [14].

B. Data Provenance Tree

In this subsection, we propose to model the multi-stage sampling clusters resulted from a transformation chain as a *data provenance tree*, in order to compute Eq (3) and (4). The tree is in essence distributed since each partition builds and maintains the subtree that represents the multi-level clusters contained in it. Figure 2 can be seen as a tree shown in Figure 3, which maps the input to output items by each transformation. A node in the tree is used as a container for

Subroutine	Semantics
sampleInputPar (rate)	Samples input partitions with <i>rate</i> and returns selected partitions.
sampleInputDI (rate)	Samples input data items in the selected partitions with <i>rate</i> and returns selected data items.
createRoot ()	Creates root for the tree.
createNodes ({DI})	Creates new nodes using data items {DI}, where each node corresponds to one item, where an item is a data item in an RDD.
addLevel ({node})	Adds a new level to the tree using {node}, where each node's parent is the parent data item that has generated the data item that this node represents.
replaceLast ({node})	Replaces last level nodes with {node}. A new node (<i>e</i>) shares the parent of the node that has generated <i>e</i> in this transformation. Then the nodes that were originally in the last level are deleted.

TABLE II: Description of subroutines used in Algorithm 1.

necessary parameters such as variance of each cluster at every level (sampling stage) in order to recurrently compute Eq (3) and (4) eventually at the root. The tree mirrors the translation of sampling during the execution of an RDD transformation sequence, to an equivalent multi-stage sample being taken from the final output RDD. Therefore a level in the tree corresponds to a level of sampling clusters. Table I shows the subset of Spark transformations that our framework handles.

Overview. In the provenance tree, a *node* represents a sampling unit at a cluster level, which can be a partition or data item. We define two types of nodes, one is internal node, the other is leaf node. An internal node represents a sampling cluster, such as partition or data item that generates a cluster of data items, and leaf node represents a final output data item. A provenance tree will be incrementally built as each transformation executes in parallel. The computation of Eq (3) and (4) occurs after the tree has been built, which is accomplished by traversing the tree level by level from bottom to the top. The computation does not depend on the values of intermediate data items themselves, so an internal node stores its cluster members (children nodes), estimated sum/variance of the cluster it represents. Note that the estimated sum/variance of clusters for an internal node is not computed until the entire tree has been built, whereas a leaf node always keeps the value of the data item it represents.

Tree building. We introduce Algorithm 1 for building a *data provenance tree* mapped from an RDD transformation chain. Table II shows the semantics of subroutines in the algorithm that are not explicitly defined. We assume that input partitions and data items are sampled when input data is being loaded, so the tree’s total number of levels would be at least three: the final sample at the root (level 0), chosen input RDD partitions and input data items in the chosen partitions. The tree will have more levels if the transformation chain is mapped to more than two cluster levels, depending on each transformation’s semantics. Lines 1 to 9 contain the main algorithm, which takes as input a transformation chain $\{T\}$, partition sampling rate $pRate$ and input data item sampling rate $iRate$. Line 2 to 6 are executed sequentially, where it first creates a root node, then samples the input partitions with rate $pRate$ (line 3) and adds the sampled partitions ($\{P\}$) to the tree as a children to the root. We use $\{rate\}$ to keep track of each level’s sampling rate. Line 8 is the parallel execution of `buildSubtree` (line 10 to 28) for each partition, which builds a subtree rooted at each partition node (sans the partition node itself) as each transformation executes. Lines 11 to 13 add the sampled input data items in each chosen partition as a new level. Then the rest of the algorithm will update the tree based on each T_i ’s semantics, using $\{node\}_k$ created from data items generated from T_i . If T_i is `sample`, it replaces the nodes in the last tree level with $\{node\}_k$ generated by sampling the previous level, then updates $rate_k$ using `sample`’s sampling rate. If T_i is `flatMap` and there is sampling operation before it, a new level is added because sampling data items before applying `flatMap` is equivalent of dropping groups of data

Algorithm 1: Building data provenance tree

```

1 Algorithm DataProvenance ( $\{T\}$ ,  $pRate$ ,  $iRate$ )
2   createRoot (); // level 0
3    $\{P\} = \text{sampleInputPar}(pRate)$ ;
4    $\{node\}_P = \text{createNodes}(\{P\})$ ;
5    $rate_1 = pRate$ ;
6   addLevel( $\{node\}_P$ );
7   for  $P_i \in \{P\}$  do in parallel
8     | buildSubtree( $\{T\}$ ,  $iRate$ );
9   end
10 subroutine buildSubtree ( $\{T\}$ ,  $iRate$ )
11    $\{DI\} = \text{sampleInputDI}(iRate)$ ;
12    $\{node\}_{DI} = \text{createNodes}(\{DI\})$ ;
13   addLevel( $\{node\}_{DI}$ );
14    $rate_2 = iRate$ ;
15    $k = 3$ ; // tracks tree level
16    $rate_k = 1.0$ ; // initializing  $rate_3$ 
17   for  $T_i \in \{T\}$  do
18     |  $\{DI\}_k = \text{exec}(T_i)$ ;
19     |  $\{node\}_k = \text{createNodes}(\{DI\}_k)$ ;
20     | if  $T_i$  is sample then
21       | | replaceLast( $\{node\}_k$ );
22       | |  $rate_k *= \text{sample.rate}$ ;
23     | else if  $rate_k < 1.0$  and  $T_i$  is flatMap then
24       | | addLevel( $\{node\}_k$ );
25       | |  $rate_k = 1.0$ ,  $k++$ ;
26     | else if  $T_i$  is map or flatMap or mapValues
27       | | or filter then
28       | | replaceLast( $\{node\}_k$ );
29   end

```

items generated from this `flatMap`, thus adding a new level of clusters. In other cases, the last level’s nodes will be replaced by $\{node\}_k$ without adding a new level.

Multi-key computation. A transformation can produce multiple keys, and a transformation chain finally leads to multiple final output key spaces. However, only each final output key, instead of an intermediate key, defines an independent Spark computation. Since we are only interested in the estimator and error bounds of the final output RDD, the multi-level clustering in the final sample would only be determined by the leaf nodes in the same key space. Therefore in the provenance tree building process, the intermediate key spaces need not to be explicitly reflected in the internal nodes. The presence of multiple keys also introduces the need of population estimation which can be handled by the theory introduced earlier.

Limitations. We assume `sample` and `filter` will not eliminate all the data items from a particular partition, so that number of partitions stay the same after loading the input data. We does not consider `filter`’s effect over the sampling error, specifically we are not sure about its impact over the variance and how it is propagated through clusters to the final error bound. It is because `filter` deterministically eliminates

some data items based on its predicate, instead of randomly selecting data items where the sample sum and variance would follow a certain distribution. We leave exploring the impact of filter over error bounds as a future work.

C. Tree traversal-based statistics computation

Eq (3) and (4) can be computed by traversing the provenance tree built using Algorithm 1. The tree is traversed level by level starting from the leaf nodes, from which the estimated sum/variance of internal nodes at each level can be incrementally computed, and the desired final output is estimated sum/variance at the root. We introduce Algorithm 2 for this computation process. Lines 3 to 7 compute in parallel each partition’s statistics by calling the subroutine `ComputeNodeI`, which computes Eq (3) and (4) of a given node at level k . Line 8 computes the root’s estimated sum/variance for the final confidence interval output.

Algorithm 2: Confidence interval computation

```

1 Algorithm ComputeTree ( $tree$ )
2    $d = tree.numLevels - 1$ ;
3   for  $k \leftarrow d$  to 1 do in parallel
4     for  $node_i \in$  all nodes at level  $k$  do
5       ComputeNodeI ( $node_i, k, d$ );
6     end
7   end
8   ComputeNodeI ( $root, 0, d$ );
9   return  $CI(root.\hat{\tau}, root.\hat{V})$ ;
10 subroutine ComputeNodeI ( $node, k, d$ )
11    $\{c\} = node.children$ ;
12   if  $k$  is  $d$  then
13      $node.\hat{\tau} =$  data item’s value;
14   else if  $k$  is  $d - 1$  then
15      $m_{I_k} = \{c\}.size(), S_{u,I_k}^2 = Var(\{c.\hat{\tau}\})$ ;
16      $node.\hat{\tau} = Eq3(\{c.\hat{\tau}\}, m_{I_k}, M_{I_k})$ ;
17      $node.\hat{V} = Eq4(m_{I_k}, M_{I_k}, S_k^2)$ ;
18   else
19      $n_{I_k} = \{c\}.size(), S_{u,I_k}^2 = Var(\{c.\hat{\tau}\})$ ;
20      $node.\hat{\tau} = Eq3(\{c.\hat{\tau}\}, n_{I_k}, N_{I_k})$ ;
21      $node.\hat{V} = Eq4(n_{I_k}, N_{I_k}, S_{u,I_k}^2, \{c.\hat{V}\})$ ;
22   end

```

We illustrate the computation process by using the tree in Figure 3 representing three-stage cluster sampling as an example. We begin with the level where $k = 2$, we first compute the intra-variance of c_2 formed by $c_2 : e_1$ and $c_2 : e_2$. After computing other clusters (c_3, c_4 and c_5) in the same level, we decrement k to 1 and moves to second level nodes. Computing statistics (e.g., intra/inter-cluster variance) for P_1 depends on c_2 and c_3 ’s statistics (same for P_3), which has already been computed in the previous level. Finally, the variance at the root comprising P_1 and P_3 can be computed.

D. Per-key population estimation

A Spark transformation can generate multiple keys, thus sampling before a transformation is equivalent of sampling

a mixed-key population where sub-population size of each key is unknown. It is because sampling occurs before the transformation that actually generates a key. However, each sub-population size is needed because variance computation applies to each output key. We model estimated population size of a cluster at each sampling stage as a negative binomial distribution parameterized by sample size and sampling rate i.e. $\hat{N}_{I_k} \sim \mathcal{NB}(n_{I_k}, p)$, where \hat{N}_{I_k} is the population size, n_{I_k} is the sample size and p is the sampling rate applied for the sub-clusters. n_{I_k} at the current stage is equivalent to the estimated population size of the next stage (N_{I_k}), p , n_{I_k} and N_{I_k} corresponds to the success rate, number of successes and the number of trials in a binomial distribution. The unbiased estimator \hat{N}_{I_k} is $\frac{n_{I_k}}{p}$. The same logic also applies to the last sampling stage where the value of M_{I_k} needs to be estimated. The uncertainty coupled with estimating N_{I_k} and M_{I_k} , must be included in computing the variances in Eq (4) since their estimators affects the variance. We have detailed derivation on incorporating it into the variance computation in the Appendix of our technical report [29].

IV. STRATIFIED RESERVOIR SAMPLING

An inherent limitation of multi-stage sampling is that some rare output keys may either be lost or have large error bounds. We leverage an one-pass sampling algorithm *Adaptive Stratified Reservoir Sampling* (ASRS) [15] to address the rare key issues. ASRS combines stratified and reservoir sampling [14], [30], and uses power allocation [31] to divide the total sample size among different strata proportionally to each stratum’s running sampling error. ASRS dynamically increases the sampling rates of rare keys to compensate for their larger sampling errors and decreases sampling rates on popular keys [15].

ASRS with partition sampling. ASRS has a larger overhead compared to simple random sampling. In order to achieve balance among output key retaining, balanced error bound distributions and the overall execution time, we sample RDD partitions at the input and apply ASRS over the their elements, so that in the chosen RDD partitions, sampling errors among popular and rare keys are more even and rare keys are better retained. Partition sampling at the input will have significant execution time saving since much I/O time is reduced. We can estimate the result and the error bound using standard multi-stage sampling theory using Eq (3) and (4), because an ASRS sample is very close to a simple random sample [15].

Limitations. ASRS stratifies the sample by output keys, thus it cannot be applied unless the output keys are available. However, sampling right before aggregation will not save execution time since aggregation is relatively cheap. Therefore our solution is to apply ASRS over an intermediate RDD, which would make ASRS suitable for applications where an output key’s occurrence is proportional to an intermediate key.

V. APPROXSPARK IMPLEMENTATION

We have implemented our approximation mechanisms by either modifying/extending the original Spark framework. We

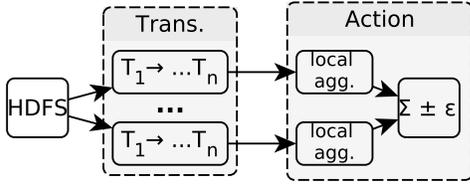


Fig. 4: Error bound computation process, divided across the transformation and action phases. The tree building happens in the transformation phase, and the error bound computation happens in the action phase.

extend Spark executor implementation to maintain our data provenance tree. We also extend Spark’s `StatCounter` class to store intra and inter-cluster variances, sample sizes, sampling rates, etc. ApproxSpark offers two methods for user to set the degree of approximation, either by specifying the sampling rates or error bound targets. In addition to setting specified sampling rates, user is also able to set target error bounds at different percentiles on the error bound CDF of all keys. For example, a user may specify that the 10th percentile of the error bound is at most 0.1, the 50th percentile at 0.3, the 90th percentile at most 0.6.

A. User-specified sampling rates

Multi-stage sampling. We modify the partition loading and computation mechanisms in Spark’s `HadoopRDD` class to support partition/input data item sampling when data is being loaded into an RDD. Subsequent RDDs’ data items can be sampled using the original `sample` function from Spark API. In order to forward information to the output RDD as in Table I, we extend the implementations of those transformations in RDD class to support the data provenance building algorithm shown in Algorithm 1. For example, `flatMap` not only tags a group of data items generated from the same data item i in the parent RDD with a cluster id c_i , it also implements the provenance tree building logic. ApproxSpark provides the user with a new RDD transformation `aggregateByKeyMultiStage`, for both intra and inter partition aggregations when multistage sampling is used. It is similar to RDD’s original `aggregateByKey` but has added error bound computation mechanisms.

Error bound estimation. The error computation process is shown in Figure 4. As introduced in Algorithm 1, the first two levels of the provenance tree are sequentially built by the Spark driver program. Then every subtree rooted at each partition node (sans the partition nodes) are built by each parallel task in the transformation phase, maintained by a coordinator in each Spark executor. In the action phase, an RDD partition is first locally aggregated by each Spark executor before sending them to reducers across the network for final aggregation. In the local aggregation phase, the subtree of each partition is traversed to compute each partition’s statistics. Then in the

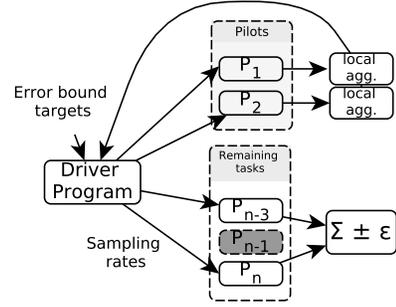


Fig. 5: User setting error bound targets design, gray shaded box is dropped partition(s).

final aggregation, the statistics of each partition are sent to the reducers for computing the inter-cluster variance among the RDD partitions and the final confidence interval. As transformations $T_1 \rightarrow T_n$ execute in parallel on every partition in the transformation phase, the subtree for every partition of the provenance tree is built; in the action phase, partitions are first locally aggregated to compute each partition’s statistics, then sent to the reducers across network for the final error bound computation.

Stratified reservoir sampling. We modify ASRS for Spark’s distributed environment by dividing the total reservoir size, taken as a user input, evenly among RDD partitions. Each partition is then sampled using ASRS independently without coordination among them. We implement ASRS as a transformation that produces another RDD, containing the resulting sample with balanced sampling errors among popular and rare keys. ASRS changes the sampling rate by changing the size of the portion of the reservoir allocated for a particular key. On the other hand, ASRS shrinks the size allocated to each existing key as it discovers more keys in the partition, where the initial reservoir size for the new key is set as the average of the sizes for existing keys. The benefit of implementing ASRS as a transformation is that the resulting RDD can be cached in memory for reuse. We provide `ASRSSample` for the user to sample an RDD using ASRS and `aggregateByKeyStratified` for the aggregation with error bound computation, both implemented as RDD transformations.

B. User-specified target error bounds

We propose a greedy algorithm to search for a sampling rate combination leading to a potentially tight error bound CDF constrained by the target errors, while aiming to significantly reduce execution time. Initially, partition and data item sampling rates are both initialized as 1.0. The algorithm includes two phases. 1) In the first phase, a wave of pilot tasks are executed and the partially aggregated results from these tasks are sent back to the driver program, where the number of data items M and inter/intra cluster variances for each key are computed. It uses a Spark’s job submission mode that returns the partially aggregated partitions to the driver instead of sending them for shuffling. 2) In the second phase, the algorithm uses statistics gathered in the first phase to

predict error bounds: it first lowers partition sampling rate for potentially maximum execution time reduction until it would violate any error bound target, then it searches for an appropriate input data item sampling rate, before the predicted error CDF would violate any of the user-specified targets. When predicting errors for keys that are not encountered in the first phase, the algorithm just uses the average of the statistics for the keys obtained in the first phase. When the predicted error distribution meets all the error targets with the lowest possible sampling rates, the algorithm proceeds to the second phase and uses them for the remaining Spark tasks. Figure 5 shows the architecture of user setting error bounds.

The algorithm exploits a property that partition sampling may incur more sampling error [14], but reduces more execution time compared with data item sampling. Our algorithm follows a principle in online aggregation - *minimum time to accuracy* [23], i.e. minimizing the time to achieve a useful estimated value. However, online aggregation typically outputs a running confidence interval for a single estimator as data is being aggregated in a random order, whereas ApproxSpark applies multi-stage sampling over the data and outputs the error bounds for multiple keys at the end of execution.

Limitations. The algorithm assumes the keys are distributed evenly and the pilot partitions are representative of the entire dataset. However, when keys are not distributed evenly, the pilot wave is not able to accurately estimate the parameters. The error bound computation in our implementation has only considered two-stage sampling, while in theory, user can insert multiple sampling operations along the chain and achieve the target error bounds. We leave this more complicated case as future work.

VI. EVALUATION

We evaluate ApproxSpark using five real world applications from different application domains (see Table III). We begin by briefly describing the applications. We then use them to extensively explore the tradeoff space between sampling and precision. Finally, we explore ApproxSpark’s ability to find appropriate sampling rates for user specified target error bound constraints.

Experimental environment. All experiments are run on a cluster of four servers. Each server is equipped with a 2.5GHZ Intel Xeon CPU with 12 cores, 256GB of RAM, and a SATA hard disk. The cluster is interconnected with 1Gbps Ethernet. All servers run Linux 3.10.0. ApproxSpark is implemented on top of Spark version 1.6.1 and is configured with 16 executors, each of which runs up to 6 tasks, so that each server has 4 executors, running up to 24 tasks.

A. Applications

Word Co-occurrence (Co-occur). Co-occurrence is a common text mining application that computes the frequencies of pairs of words [32]. In this study, the application counts co-occurrences of topic tags in the MEDLINE database [33], containing more than 20M citation records of publications

Application	Domain	Input Dataset	Size (GB)
Co-occur	Text Mining	MEDLINE database	7.5
Speed	Smart City	GPS trace	36.0
Twitter	NLP	Tweets2011 (TREC)	2.2
PageRank	Graph Analysis	Wikipedia snapshot	53.0
Clickstream	Log Analysis	Wikipedia clickstream	6.5

TABLE III: List of applications, the domains they come from, and the input datasets used in our evaluation.

in life sciences. Each citation record contains a set of topic tags, listing the major topics relevant to the publication. The application first reads the input data into an RDD, and then performs a `map` to extract the list of major topic tags from each citation record. It then performs a `flatMap` to generate key-value pairs ((co-occurring tag pair), 1). Finally, it sums and outputs the count of each co-occurring tag pairs.

Vehicular Average Speed Analysis (Speed). This application analyzes the average speed of vehicles moving in a geographical area each hour at three different granularities: around a point-of-interest (POI) (e.g., a restaurant), on a road segment, and within a region. An analysis of vehicular traces is useful for monitoring urban traffic, predicting passenger demand, recommending taxi routes, etc. [34]. We analyze a taxi GPS dataset containing status records collected every 30 seconds from 14,000 taxis operating in Shenzhen, China, over one week [35]. Each record contains information about a taxi, including a timestamp and the taxi’s GPS location and speed. The dataset has ~ 291 M records that covers an area of ~ 790 square miles divided into 491 regions, containing ~ 569 k POIs and ~ 198 k road segments. Each POI is assigned to a road segment and each road segment belongs to a region. The application reads the input data into an RDD, and then performs three transformations using metadata and three actions. The three transformations are three `map` operations that: (1) transform each GPS entry into a ((POI, hour), speed) key-value pair; (2) transform each ((POI, hour), speed) pair into a ((road segment, hour), speed) key-value pair; and, (3) transform each ((road segment, hour), speed) pair into a ((region, hour), speed) pair. The three actions use the three intermediate RDDs to compute the average speed per hour at each POI, each road segment, and each region, respectively.

Twitter Hashtags Sentiment Analysis (Twitter). Sentiment analysis computes quantitatively whether a piece of text is positive, negative or neutral using natural language processing (NLP) techniques [36]. In this study, the application computes the average sentiment for each unique hashtag in the Tweets2011 Twitter dataset from TREC 2011 [37], using the Stanford CoreNLP library [38]. This dataset contains ~ 16 M tweets sampled over 17 days in early 2011. The application first reads the input data into an RDD, and then performs a `map` to compute a score in the interval $[0, 5]$ with 0 being *very negative*, 3 being *neutral*, and 5 being *very positive* for each tweet. It then performs a `flatMap` to extract all hashtags from each tweet and associates each with the sentiment score for the tweet. Finally, it computes and outputs the average

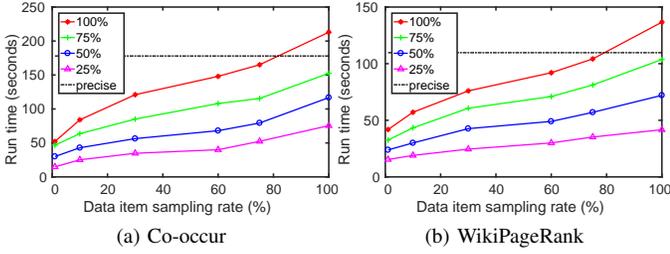


Fig. 6: Execution times under different sampling rates. Each line corresponds to a partition sampling rate. The x-axis shows the sampling rate for input data items. The dashed line gives the run time of precise executions.

sentiment for each hashtag.

WikiPageRank (PageRank). This application counts the number of articles that link to each article in a set, emulating one of the main processing components of PageRank [39]. We use the Wikipedia data snapshot from 2016 with $\sim 5M$ articles [40]. The application first loads the data into an RDD, then applies a `map` to parse the XML, generating a list of outbound links for each article. It next performs a `flatMap` to generate pairs of (destination article, 1). Finally, it sums and outputs the count for each destination article.

WikiClickstream (Clickstream). Clickstream analysis can be used to generate a weighted network of linked articles showing the probability of users navigating from one article to another. We use a Wikipedia clickstream dataset from 2016 [41] containing $\sim 149M$ tuples of (source, destination, count), where count is the number of times that a user has visited the destination page from the source page. The application computes the total count for each unique (source, destination) pair. Specifically, it reads the input data into an RDD, performs a `map` to generate a key-value pair for each entry, and then sums and outputs the total count for each unique (source, destination) pair.

B. Results for multi-stage sampling

We explore the performance and accuracy of multi-stage sampling using four of the above applications: Co-occur, Twitter, WikiPageRank, and WikiClickstream. In most experiments, we sample the input data as it is read into the first RDD because this will lead to the highest speedups. However, we also explore sampling from RDDs later in the applications' transformation chains to explore the trade-off between performance and accuracy of such scenarios.

Execution times. Figure 6 plots the execution times for two of the applications, Co-occur, WikiPageRank at different partition and data item sampling rates. Consistent with previous results from [25], we observe that (a) multi-stage sampling significantly reduces execution times, and (b) partition sampling can lead to larger execution time savings than data item sampling. The latter is because dropping a partition eliminates overheads such as I/O time for reading the blocks, the creation of an RDD partition in memory, etc., whereas data item sampling

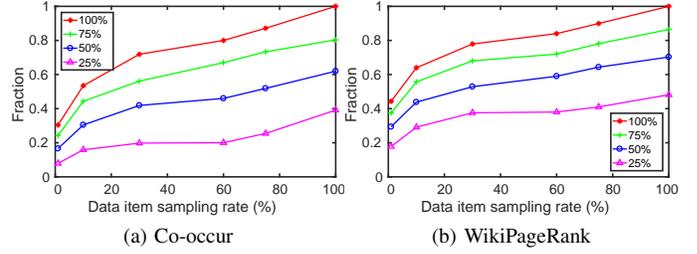


Fig. 7: Fraction of unique keys (normalized against number of keys produced under precise execution) outputted under different sampling rates. Each line represents a particular partition sampling rate.

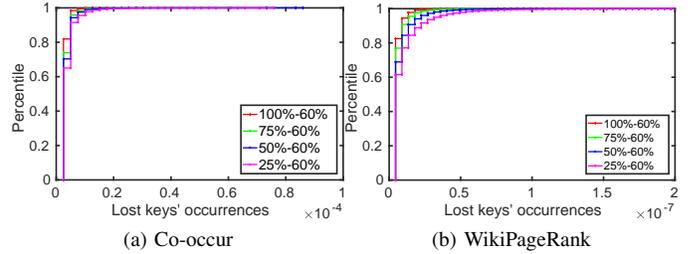


Fig. 8: CDFs of occurrences of the lost keys, normalized against the total number of data items across all keys at a data item sampling rate of 60%. Each line corresponds to a specific partition sampling rate.

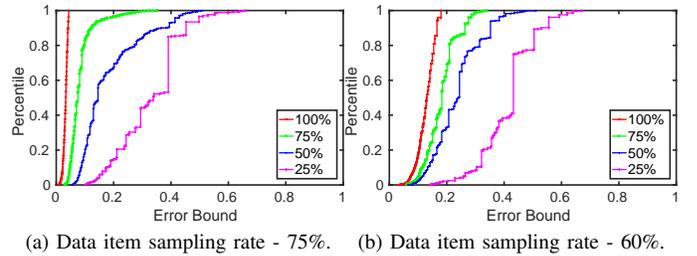


Fig. 9: Each graph plots CDFs of errors with 95% confidence error at a fixed input data item sampling rate for Co-occur application. Each line in a graph plots the error CDF at a particular partition sampling rate.

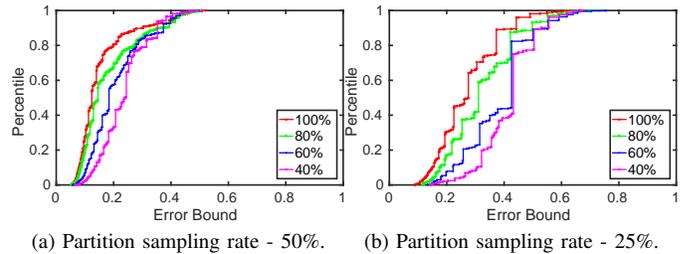


Fig. 10: CDFs of errors at a fixed partition sampling rate for Co-occur application. Each line in a graph plots the error CDF at a particular input data item sampling rate.

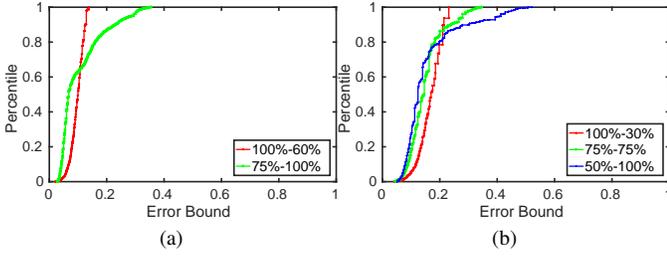


Fig. 11: Error distribution trade off under different partition and data item sampling rates combination from the Co-occur application. The legends indicate partition and data item sampling rates respectively.

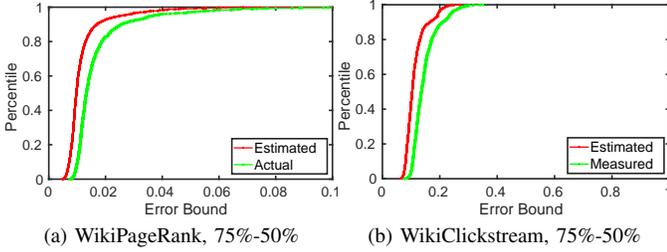


Fig. 12: Estimated and actual relative error comparison.

still requires some processing for each partition. The sampling framework imposes some overheads; i.e., execution time for the (100% (partition sampling), 100% (data item sampling)) case is somewhat greater than that of the precise version, ran on unmodified Spark.

Fraction of keys in output. As already mentioned, multi-stage sampling can result in loss of keys in the output for jobs that produce more than one key. Figure 7 plots the fractions of keys present in the output at different sampling rates for the same three applications, normalized against the total number of keys produced in the precise executions. Figure 8 shows the occurrence frequencies of lost keys in the input RDD of the final aggregation action in a precise execution, normalized against the total number of data items in the RDD. We observe that significant fractions of keys can be lost, especially at higher partitioning sampling rates. For example, sampling rates of (75%, 60%) for Co-occur reduce execution time by 40% at the expense of losing 25% of the keys produced by the precise execution. However, Figure 8 shows that only *rare* keys are lost. For example, for the same (75%, 60%) sampling rates in Co-occur, the most frequently appearing key that was lost accounted for only a very small fraction 0.85×10^{-4} of the total number of data items in the input RDD of the final aggregation action, while 90% of the lost keys each accounted for less than or equal to 0.08×10^{-4} of the total number of data items in the RDD. The lost keys are even more rare in the WikiPageRank and WikiClickstream applications, where the occurrences of each lost key accounting for 10^{-7} of the total number of data items.

Effect of sampling rates on error bounds. Figure 9 plots

the CDFs of the estimated error bounds computed as $\frac{\epsilon}{v_{approx}}$, which are the ratios of sampling error to estimated value, for all keys with 95% confidence for Co-occur. Each graph in the figure plots CDFs for several different partition sampling rates while keeping data item sampling rate fixed. We observe that, as pointed out in [19], multi-stage sampling without considering keys in the final output over-samples popular keys and under-samples rare keys, leading to uneven relative error bounds. This can lead to large relative error bounds in the tails of the relative error bounds CDFs.

We observe that even relatively high partition sampling rates (e.g., 75% - green curve in Figure 9(a)) can significantly impact error bounds for more rare keys (pushing the CDF curve for $>60\%$ to the right) while not affecting the frequently appearing keys much (the CDF curve does not change much for $<60\%$). Interestingly, a 75% partition sampling rate affects error bounds less or comparable to a 75% data item sampling rate (red curve in Figure 9(b)) for up to 60% of the keys, but the tail is significantly worse for partition sampling. We believe this is caused by the clustering of data items with the same keys within partitions. As either or both sampling rates decrease, the entire error bound CDF shifts to the right (larger error bounds). However, the observation that partition sampling affects the tail of error bounds CDF much more strongly than data item sampling remains consistent throughout.

Figure 10 shows the error bound CDFs when the partition sampling rates are fixed with varying data item sampling rates, the tails of the error bound CDFs are similar under the same partition sampling rates. This points to a fundamental trade-off: partition sampling can reduce execution time over data item sampling, but trades off higher error bounds for the rarer keys to do so. Figure 11 shows that the 95% relative error CDFs can exhibit trade-offs with different combinations of partition and data item sampling rates. In each subgraph, the sampling rates are chosen so that they have similar execution time as in Figure 6(a). We can see that their error CDFs intersect, with the error CDFs from lower partition sampling rates having worse tails. It shows that different partition and data item sampling rates combinations can achieve similar execution time, but different error bound distributions. For example in Figure 11(a), (100%-60%) has better smaller errors after the 62th percentile, but performs worse on frequent keys that have smaller errors. It is because (100%-75%) processes more data than (100%-60%), so the frequent keys result in smaller errors but the rarer keys have worse error due to partition dropping.

Comparison with relative error. Figure 12 plots the distributions of estimated error bounds, versus the relative error against ground truth - $|1 - \frac{\hat{v}}{v}|$. We can see that ApproxSpark’s error estimation is constantly lower than the actual relative error. We can also see that the estimation is more accurate at lower percentiles and less so at higher percentiles. It is because the popular keys usually provides more statistical information to the error estimation process than rare keys.

Sources of uncertainty. As previously explained, uncertain-

Source	Sampling Rates	
	(100%, 30%)	(75%, 75%)
Partition sampling	0%	78%
Data item sampling	88%	12%
Pop. estimate partitions	0%	5%
Pop. estimate data items	12%	5%

TABLE IV: Breakdown of uncertainty on average across all keys for the four sources of errors in multi-stage sampling for Co-occur.

Sampling Rates	Execution Time (s)	Error 100 th	Bound 90 th	Percentile 50 th	% Keys Present
100%-60%	149.2	0.17	0.12	0.10	80.0
75%-100%	150.8	0.37	0.22	0.06	80.3
100%-30%	120.9	0.22	0.20	0.17	71.8
75%-75%	121.4	0.32	0.28	0.13	72.3
50%-100%	119.7	0.51	0.31	0.11	61.5

TABLE V: Comparison of run times, error bounds at 100th, 90th, 50th percentiles, and fraction of unique keys for Co-occurrence.

ties (leading to estimated error bounds) can arise from the sampling as well as population estimations. Table IV shows the percentages of the error bounds, averaged across all keys in the output, attributable to each of four sources for Co-occurrence. We observe that the inter-cluster variance from partition sampling accounts for by far the largest portion of the estimated error bounds, which is consistent with [14]. The intra-cluster variance from data item sampling accounts for the next largest portion, while population estimations for number of partitions, the number of groups of co-occurred words, within each partition for each key, account for only small portions of the error bounds.

Summary. Putting together the observations made above, we conclude that multi-stage sampling works well to significantly reduce execution time while introducing small to modest relative errors, as long as the loss of rare keys are acceptable. Further, data item sampling would typically be preferable to partition sampling because it gives more consistent error bounds across keys. To more clearly support this conclusion, Table V presents data for two sets of sampling rates for Co-occur, $\{(100\%, 60\%), (75\%, 100\%)\}$ and $\{(100\%, 30\%), (75\%, 75\%), (50\%, 100\%)\}$, where members

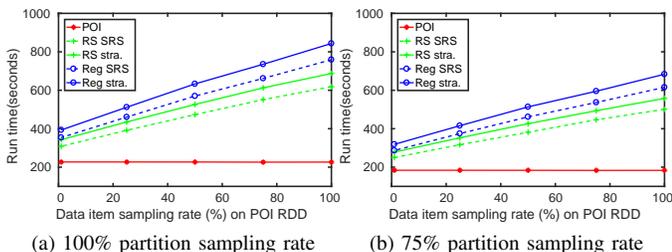


Fig. 13: Run times for the aggregations at POI, road segment and region RDD. The dashed curves represent run times under SRS, solid curves represent stratified sampling (stra).

within each set have similar execution times. As the partition sampling rate increases, the tail of the error bounds CDF worsen significantly. The trend is less clear for lost keys; however, high partition sampling rates (e.g., 50%) can clearly lead to significantly increased number of lost keys. Looking at Figure 6(a), this implies that partition sampling rates of 50% and 75% are not as useful since similar performance is achievable with (100%, $x\%$) sampling rates. On the other hand, execution time can be reduced using a partition sampling rate of 25% if one is willing to tolerate the accompanying key loss and increased error bounds.

C. Results for stratified sampling using ASRS

In the Speed application, we explored both stratified sampling using ASRS with power allocation technique, and simple random sampling (SRS) on the data items in POI RDD. In addition, partition sampling is also applied when reading the input data. When stratified sampling is performed over the data items in the POI RDD, it also creates stratification effect for both road segment and region RDDs since each POI maps to a road segment, which in turn maps to a region. We use power allocation techniques to balance the sampling errors at each strata in the POI RDD when stratified sampling is applied.

Execution times. Figure 13 shows the execution times for aggregating at multiple RDD along the chain. We see that aggregating the street and region RDD both have run time reduction as the sampling rate on the POI RDD lowers, whether stratified sampling or SRS is performed. However, we do see that stratified sampling using ASRS has much higher overhead than the SRS since it needs to perform stratification and power allocation over the keys.

Confidence interval. Figure 14 plots the estimated values with error bars of the average speed at the region level. The precise result is plotted in the dashed black curves, estimated value in green curve and 95 % confidence intervals as red error bars. Usually higher average speed is observed in regions that are away from city centers and at hours that are early in the morning or late night, as a result these points come with lower taxi densities, i.e., fewer samples which also tend to cluster over a few partitions. These points would have larger error bars without balancing the sample sizes among popular and rare keys, as shown in the Figure 14(a) and (c), whereas stratified reservoir sampling coupled with power allocation technique increases the sampling rates of these rare keys, resulting in smaller error bars for them. However, we do observe that the popular keys have shorter error bars under SRS compared to stratified, it is because the popular keys have a much larger representation in the sample than the rare keys.

Fraction of keys shown in the output. In Figure 15, we see that the stratified sampling constantly loses less keys than SRS at same sampling rate. In Figure 15(b), we see that when partition sampling rate is set, stratified sampling can preserve the number of output keys without being affected by the sampling rate over the data item shown in (a). This shows

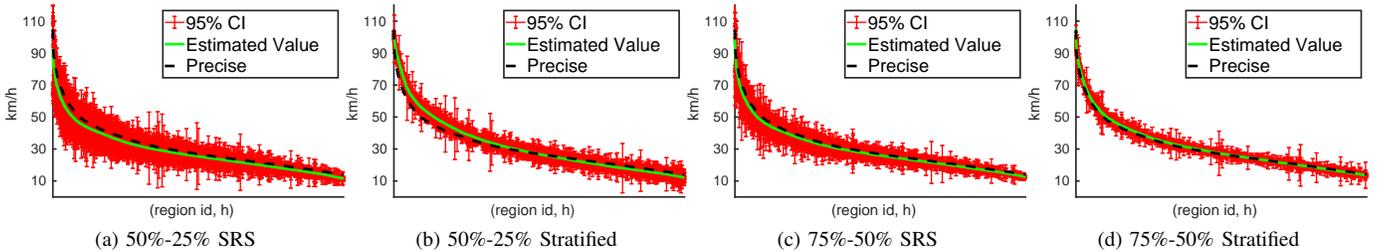


Fig. 14: Region average speed at each hour at different partition and data item sampling rates combinations. Comparisons of 95 % Confidence Interval width when sample random or stratified sampling is applied at the POI RDD, coupled with partition sampling at the input.

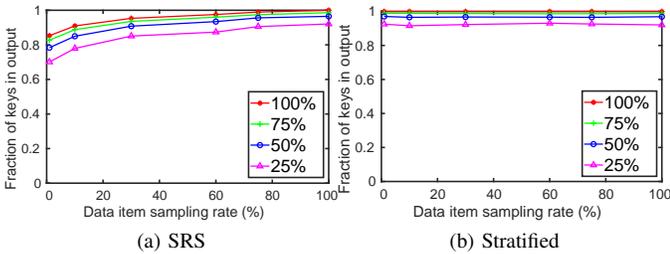


Fig. 15: Number of output keys (normalized) occurred in the output, SRS or stratified sampling performed at the road segment RDD. Each line represents a partition sampling rate at the initial RDD.

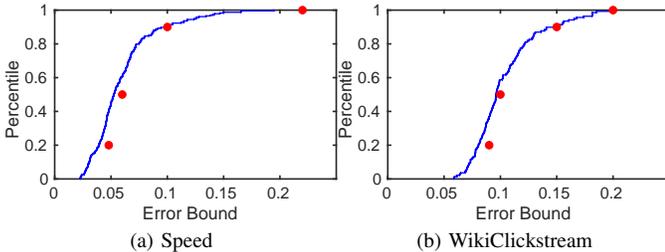


Fig. 16: User-specified error bounds targets shown on the error CDF achieved by ApproxSpark.

that stratified reservoir sampling is much better at preserving output keys in the result.

Summary. ASRS can not only achieves a balanced error distribution among popular and rare keys, it also loses much fewer output keys, consistent throughout varying sampling rates over the data items. However, ASRS has a higher sampling overhead than SRS which is reflected in the execution times.

D. Results for user-specified error targets

We now demonstrate ApproxSpark’s capability in allowing users to set error bounds target at different percentiles over the relative error distribution. The red dots in Figure 16 show the target error bounds at 20th, 50th, 90th, 100th percentiles; the blue curves show the resulting CDF achieved by ApproxSpark, setting both partition and data item sampling step sizes to be 0.1%. We can see that the CDFs are bounded by the targets set by the user. Figure 16(a) is the error distribution for average Taxi speed with 50% partition, and 80% data item sampling

rates at the POI RDD. Figure 16(b) is the error distribution for WikiClickstream aggregation result with 40% partition and 60% data item sampling rates. We randomly select 10% of the RDD partitions to be executed in the pilot wave. This approximation mode incurs more overhead compared with setting the sampling rates that satisfy the user-specified error targets. We have observed that the pilot wave causes about 20% and 25% extra execution time respectively in the two applications compared with setting the sampling rates directly.

VII. CONCLUSION

In this paper, we motivate, design, and implement a system called ApproxSpark, which features a set of approximation mechanisms leveraging sampling theories to adapt to Spark’s computing paradigm. Our proposed multi-stage sampling theories together with a data provenance tree allows for a general approximate computing framework under Spark’s parallel computing model with multi-step RDD transformations. We utilize a set of metrics to rigorously evaluate ApproxSpark including run times, the error bound distribution for all keys, and the number of missed keys using applications from different domains. We have also gained important insights on an interesting yet complicated trade-off space in terms of the error bounds, run times and number of keys, when choosing different sampling schemes, such as specific sampling rates, partition vs data item sampling, whether to use stratified sampling, different combinations of partition/data item sampling rates. We also have found that input data that contains less rare keys for the application is more amenable to approximation. In a real-world setting, a user would choose the most appropriate sampling setup catering to her approximation goal. Based on our experience and results, we conclude that our framework and system can make efficient and customized approximation to big data practitioners using Spark.

APPENDIX

Estimated sum for all clusters is:

$$\hat{\tau} = \frac{\hat{N}}{n} \sum_{i \in S} v_i = \hat{N} \bar{\tau} \quad (5)$$

Sample mean among the cluster totals is:

$$\bar{\tau} = \frac{1}{n} \sum_{i \in S} v_i \quad (6)$$

Estimated total number of clusters N is:

$$\hat{N} = \frac{n}{p_1} \quad (7)$$

Since $\hat{N} \sim \mathcal{NB}(n, p_1)$, the variance of \hat{N} is:

$$Var(\hat{N}) = \frac{n(1-p_1)}{p_1^2} \quad (8)$$

If we treat it as simple random sampling, the variance of mean of cluster total is:

$$Var(\bar{\tau}) = (1-p_1) \frac{s_t^2}{n} \quad (9)$$

Variance of cluster totals $Var(\hat{\tau})_{srs} =$

$$\begin{aligned} & Var(\hat{N}\bar{\tau}) \\ &= \hat{N}^2 Var(\bar{\tau}) + \bar{\tau}^2 Var(\hat{N}) + Var(\hat{N})Var(\bar{\tau}) \\ &= \left(\frac{n}{p_1}\right)^2 (1-p_1) \frac{s_t^2}{n} + \frac{n(1-p_1)}{p_1^2} (\bar{\tau}^2 + (1-p_1) \frac{s_t^2}{n}) \end{aligned} \quad (10)$$

thus:

$$Var_{inter} = \left(1 - \frac{1}{p_1}\right) Var_{srs}(\hat{\tau}) \quad (11)$$

Estimated sum of cluster i is:

$$\hat{\tau}_i = \hat{M}_i \bar{\tau}_i \quad (12)$$

where sample mean $\bar{\tau}_i$ in cluster i is:

$$\bar{\tau}_i = \frac{1}{m_i} \sum_{j \in S_i} v_{ij} \quad (13)$$

where m_i is the number of sampled items in cluster i , M_i is the population total in cluster i and p_2 is the data item sampling rate.

Since $\hat{M}_i \sim \mathcal{NB}(m_i, p_2)$, estimated \hat{M}_i is:

$$\hat{M}_i = \frac{m_i}{p_2} \quad (14)$$

with variance:

$$Var(\hat{M}_i) = \frac{(m_i)(1-p_2)}{p_2^2} \quad (15)$$

The variance of sample mean in cluster i is:

$$Var(\bar{\tau}_i) = (1-p_2) \frac{s_i^2}{m_i} \quad (16)$$

The variance of estimated sum in cluster i is:

$$\begin{aligned} Var(\hat{\tau}_i) &= \hat{M}_i^2 Var(\bar{\tau}_i) + \bar{\tau}_i^2 Var(\hat{M}_i) + Var(\hat{M}_i)Var(\bar{\tau}_i) \\ &= \left(\frac{m_i}{p_2}\right)^2 (1-p_2) \frac{s_i^2}{m_i} \\ &+ \frac{(m_i)(1-p_2)}{p_2^2} \left(\bar{\tau}_i + \frac{(m_i)(1-p_2)}{p_2}\right) \end{aligned} \quad (17)$$

Intra-cluster variance is:

$$Var_{intra} = \frac{1}{p_1} \sum_{i \in S} V(\hat{\tau}_i) \quad (18)$$

The total variance is:

$$\begin{aligned} Var(\hat{\tau}) &= Var_{inter} + Var_{intra} \\ &= Var(\hat{\tau})_{srs} + \frac{1}{p_1} \sum_{i \in S} Var(\hat{\tau}_i) \end{aligned} \quad (19)$$

REFERENCES

- [1] F. T. Chong, M. J. R. Heck, P. Ranganathan, A. A. M. Saleh, and H. M. G. Wassel, "Data Center Energy Efficiency: Improving Energy Efficiency in Data Centers Beyond Technology Scaling," *IEEE Design & Test*, vol. 31, no. 1, 2014.
- [2] W. Dai, L. Qiu, A. Wu, and M. Qiu, "Cloud infrastructure resource allocation for big data applications," *IEEE Transactions on Big Data*, vol. 4, no. 3, pp. 313–324, 2018.
- [3] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2893356>
- [4] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.
- [5] Y. Yan, L. J. Chen, and Z. Zhang, "Error-bounded sampling for analytics on big sparse data," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1508–1519, 2014.
- [6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatarao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.
- [7] X. Xie, K. Zou, X. Hao, T. B. Pedersen, P. Jin, and W. Yang, "Olap over probabilistic data cubes ii: Parallel materialization and extended aggregates," *IEEE Transactions on Knowledge and Data Engineering*, 2019.
- [8] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 2015, pp. 2323–2324.
- [9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi et al., "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.
- [10] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015, p. 70.
- [11] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, vol. 30, no. 18, pp. 2652–2653, 2014.
- [12] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of big data on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, 2015.
- [13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: <http://dx.doi.org/10.14778/2824032.2824076>
- [14] S. Lohr, *Sampling: Design and Analysis*. Cengage Learning, 2009.
- [15] M. Al-Kateb and B. S. Lee, "Adaptive stratified reservoir sampling over heterogeneous data streams," *Information Systems*, vol. 39, pp. 199–216, 2014.
- [16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. [Online]. Available: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>
- [17] S. Chaudhuri, G. Das, and V. Narasayya, "Optimized Stratified Sampling for Approximate Query Processing," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 2, 2007.

- [18] M. Al-Kateb and B. S. Lee, "Stratified reservoir sampling over heterogeneous data streams," in *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*. Springer Berlin Heidelberg, 2010, pp. 621–639.
- [19] M. Thottethodi, T. Vijaykumar, M. Kulkarni *et al.*, "Stratified online sampling for sound approximation in mapreduce," 2015.
- [20] J. Peng, D. Zhang, J. Wang, and J. Pei, "Aqp++: connecting approximate query processing with aggregate precomputation for interactive analytics," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1477–1492.
- [21] 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 Eurosys Conference*, 2013.
- [22] X. Zhang, J. Wang, and J. Yin, "Sapprox: Enabling efficient and accurate approximations on sub-datasets with distribution-aware online sampling," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 109–120, Nov. 2016. [Online]. Available: <https://doi.org/10.14778/3021924.3021928>
- [23] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online Aggregation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.
- [24] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica, "Hold 'em or fold 'em?: Aggregation queries under performance variations," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 7:1–7:14. [Online]. Available: <http://doi.acm.org/10.1145/2901318.2901351>
- [25] 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*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 383–397. [Online]. Available: <http://doi.acm.org/10.1145/2694344.2694351>
- [26] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251254.1251264>
- [27] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe, "Streamapprox: Approximate computing for stream analytics," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17. New York, NY, USA: ACM, 2017, pp. 185–197. [Online]. Available: <http://doi.acm.org/10.1145/3135974.3135989>
- [28] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.
- [29] G. Hu, D. Zhang, S. Rigo, and T. D. Nguyen, "Approximation with error bounds in spark," *arXiv preprint arXiv:1812.01823*, 2018.
- [30] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.
- [31] M. D. Bankier, "Power allocations: determining sample sizes for sub-national areas," *The American Statistician*, vol. 42, no. 3, pp. 174–177, 1988.
- [32] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*. Springer, 2006, pp. 25–71.
- [33] "MEDLINE Data," 2017, https://www.nlm.nih.gov/databases/download/pubmed_medline.html/.
- [34] D. Zhang, T. He, F. Zhang, M. Lu, Y. Liu, H. Lee, and S. H. Son, "Carpooling service for large-scale taxicab networks," *ACM Trans. Sen. Netw.*, vol. 12, no. 3, pp. 18:1–18:35, Aug. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2897517>
- [35] D. Zhang, J. Huang, Y. Li, F. Zhang, C. Xu, and T. He, "Exploring human mobility with multi-source data at extremely large metropolitan scales," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14. New York, NY, USA: ACM, 2014, pp. 201–212. [Online]. Available: <http://doi.acm.org/10.1145/2639108.2639116>
- [36] B. Liu, "Sentiment analysis and opinion mining," *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167, 2012.
- [37] (2011) Tweets 2011. <http://trec.nist.gov/data/tweets/>.
- [38] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60. [Online]. Available: <http://www.aclweb.org/anthology/P/P14/P14-5010>
- [39] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Tech. Rep., 1999.
- [40] "Wikipedia database," http://en.wikipedia.org/wiki/Wikipedia_database., 2016.
- [41] (2016) Wikipedia clickstream. https://meta.wikimedia.org/wiki/Research:Wikipedia_clickstream/.