DISTRIBUTED FRAMEWORKS FOR APPROXIMATE DATA ANALYTICS

by

GUANGYAN HU

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Thu D. Nguyen

and approved by

New Brunswick, New Jersey

October, 2020

ABSTRACT OF THE DISSERTATION

Distributed Frameworks for Approximate Data Analytics

By Guangyan Hu

Dissertation Director:

Thu D. Nguyen

Data-driven discovery has become critical to the mission of many enterprises and scientific research. At the same time, the rate of data production and collection is outpacing technology scaling, suggesting that significant future investment, time, and energy will be needed for data processing. Straightforwardly increasing hardware resources can address the extra processing needs by either adding more CPU cores/memory (scale-up) or more worker nodes (scale-out). However, it will introduce higher computing cost that may not be feasible when budget is limited. One powerful tool to address the above challenge is approximate computing, which trades off computational time and resources with computational accuracy by reducing the amount of data needed to be processed. Fortunately, many data analytic applications such as data mining, log processing, video/image processing are amenable to approximation.

In this thesis, we describe the design and implementation of approximation frameworks to accelerate distributed data analytics. We present the frameworks targeting a variety of tasks and datasets, including log aggregation, text analytics and video querying and aggregation:

1. Our first work targets approximating aggregation jobs with error estimation. Aggregation is central to many decision support queries. Aggregation is also an

important component in OLAP (Online Analytical Processing) systems, and is frequently used for summarizing data patterns in business intelligence. Aggregation jobs often involve multiple transformation steps in a data processing pipeline. We design and implement a sampling-based approximation framework called ApproxSpark, that can rigorously derive estimators with error bounds for approximate aggregation.

2. Our second work targets approximate text analytic tasks. We propose and evaluate a framework called EmApprox that uses sampling-based approximation to speed up the processing of a wide range of queries over large text datasets. EmApprox builds an index for a dataset by learning a natural language processing model, producing vectors representing words and subcollections of documents. Our approximation index can significantly improve approximate quality while processing a small amount of the data. It will apply to each sampling unit with a sampling rate proportional to its similarity to the query. We have implemented a prototype of EmApprox as a Python library, and used it to approximate aggregation, information retrieval, and recommendation tasks.

3. Finally, we target approximate video analytics. Video data embed rich and high-quality information. Yet video analytics is particularly compute intensive as it often involves invoking a deep convolutional neural network (CNN) for object detection. We design and implement a approximate video analytics framework called VidApprox for accelerating video queries that involve object detection. VidApprox first leverages cheaåp CNNs to learn vector representations of video segments, and further processes the vectors as a persistent index structure. At query processing time, the index lookup will serve as auxiliary information for only retrieving a subset of more similar video segments. It make downstream processing such as object detection or aggregation more efficient by only performing expensive operations such as CNN inference on the relevant video data.

We show that approximation is a promising technique for reducing processing time for large datasets. However, approximation poses multifaceted challenges when applied to data processing tasks across different domains. In particular, approximation when applied can present a complicated trade-off space that involves processing time reduction, quality of computation results and preprocessing complexity. Our works not only demonstrates that it is possible to balance computational accuracy with processing time reduction, but also that a machine-learned compact representation of the data generated can function as index structure for improving approximation quality across different domains and data sets.

Acknowledgements

First of all, I would like to thank my advisor Prof. Thu D. Nguyen for providing me an opportunity to pursue doctoral study in the Department of Computer Science at Rutgers University. He gave me freedom to pursue research topics of my interests and motivated me to take on more challenges and think deeper in research. I would also like to thank him for funding my Ph.D. study, which is crucial for me to finish this thesis.

Next, I would like to thank my collaborators Prof. Desheng Zhang and Prof. Yongfeng Zhang from Rutgers University, Prof. Sandro Rigo from University of Campinas, Brazil. The collaboration with them was both instrumental for my works and a wonderful learning experience for me.

Last but not least, I would like to thank my parents for their constant support and encouragement through the ups and downs of my Ph.D. journey.

Dedication

This dissertation is dedicated to mom and dad.

Table of Contents

Chapter 1
Introduction

The past decade has witnessed exponential growth of digital data. The volume of digital data is projected to reach 149 zettabytes by 2024 [1]. Data growth is also outpacing technology scaling [2, 3]. In the mean time, data-driven services have become crucial for many industries and business. Nowadays, the challenges introduced by data growth data are characterized as three *V*s [4] - *Volume* refers to the massive sizes of data, *Velocity* describes the fast growth data and lastly, *Variety* points to the diversity of data sources (e.g. sensors, clickstreams), as well as its structure such as structured data (e.g., relational data), semi-structured data (e.g., JSON), and unstructured data (e.g, multimedia data, text) [5]. In order to mine rich and valuable information, data analytic frameworks are expected to scale and adapt to the vast amounts of data, a variety of data types as well as data processing within reasonable response times.

1.1 Approximate Computing

Employing more computational resources such as adding more CPU cores (scale-up) or more worker nodes (scale-out), can cope with high volumes of data. However, organizations or projects that are under budget limits may not be able to absorb the additional costs. For example, to process 10x amounts of the original data, one would certainly need at least 10x the original amount of computational power.

A powerful tool to address the challenge is approximate computing, which trades off execution time with computational accuracy by reducing the amount of data needed to be processed. Fortunately, many classes of applications are amenable to approximation, including log analytics, sentiment analysis, Monte Carlo computations, and image/audio/video processing [5, 6]. Since approximate computing paradigm performs analytics over a subset instead of the entire datasets, it requires less processing time and resources, and therefore allows to achieve more efficient resource utilization with less processing time. As a concrete example, e-commerce websites often want to know the popularity of individual products for various purposes such as recommendation, which can be computed from aggregating the purchase logs of users. The accumulation of logs can grow quickly. However, relative popularity instead of exact counts may be sufficient in this scenario, and a website often wants to have the statistics computed within a certain latency. Thus, approximate computing is promising as it can lead to a significant reduction in processing time and resources.

The idea behind approximate computing in the context of distributed data processing is to execute analytical queries on representative synopses of the original dataset for reduced processing time. Approximate computing summarizes the massive data in such a way that the synopsis effectively captures the original dataset's features and attributes [5]. Various techniques for building the synopses has been proposed including sampling [7], sketches [8] and online aggregation [9]. Among these, sampling has been widely used since it can provide probabilistic error bounds estimators for approximate aggregation [5,10]. With sampling-based approximation, user will has the flexibility of adjusting sampling rates to strike a balance between latency and resource consumption; user may also be able to apply non-uniform sampling rates among data items based on a priori knowledge of the data for enhanced approximation quality under resource constraints [10,11].

1.2 Thesis Contributions

Approximate data analytic frameworks can address the data processing challenges characterized by the three Vs. The objective of our work is to design and implement distributed sampling-based approximation techniques for accelerating a variety of data analytic tasks. We target log data aggregation, text analytics and video analytics. User can specify a sampling rate for the input dataset, which can translate to the processing budget of a server cluster. For aggregation jobs, user may also submit target error bounds where our system can autonomously search for appropriate sampling rates. We also propose to use machine learning model trained from the input dataset, that learn the distribution of the dataset which is used as approximate index to select the most relevant sub-dataset. The approximate index allows much more efficient processing for a sample of the same size that is constructed using uniform sampling. We leverage sampling-based approximation throughout our frameworks with different emphasis: the first work designs algorithms for computing an error bound for aggregation by expanding existing sampling theories, the second and third works propose techniques that result in improved approximation quality by processing less data. In this thesis, we present these three following approximation frameworks:

1.2.1 ApproxSpark

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 precision 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.

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. To highlight ApproxSpark's performance, with a combined sampling rate of 22%, ApproxSpark can achieve a speedup of 5x with a median error bound of 1% across all output keys.

1.2.2 EmApprox

Enterprises are increasingly seeking to extract insights for decision making from text data sets. At the same time, data is being generated at an unprecedented rate, so that text data sets can get very large. Processing such large text data sets using sophisticated algorithms is computationally expensive.

Traditional approximate query processing (AQP) systems have targeted answering aggregation queries over relational datasets with an estimator and an error bound [6, 11–14]. However, 95% of big data is unstructured [4, 15], where a bounded estimator may not be the desired output. Examples include document retrieval, where ranking accuracy is the quality metric, and visual analytics, where user perception is the key concern [16]. A ubiquitous form of unstructured data is text (e.g., Web pages, emails, news archives), which often contains important insights that can be useful toward decision making [4, 15]. However, text datasets can be very large so that processing text analytical queries can be expensive, e.g., the Google book Ngrams dataset contains 2.2 TB of text data [17], and CommonCrawl corpus petabytes of web pages [18].

The above challenge is exacerbated when it is desirable to run different types of queries against a data set, making it expensive to build multiple indices to speedup query processing. For example, given a data set comprising user reviews on products, an enterprise may want to count positive vs. negative reviews, use the reviews to make recommendations, or retrieve reviews relating to a particular product [19]. Currently, a different index is required for quickly answering each of these query types.

We propose and evaluate a framework called EmApprox that uses sampling-based approximation to speed up the processing of a wide range of queries over large text data sets. The key insight is that different types of queries can be approximated by processing subsets of data that are most similar to the queries. EmApprox builds an index for a data set by learning a natural language processing model, producing a set of highly compressed vectors representing words and subcollections of documents. When processing a query comprising one or more words, a vector representing the query is computed from the vectors representing the words. EmApprox then samples the data set, with the probability of selecting each subcollection of documents being proportional to its *similarity* to the query as computed using their corresponding vectors.

We have implemented a prototype of EmApprox as a library, and used it to approximate three types of queries: aggregation, information retrieval, and recommendation. Experimental results show that EmApprox can achieve significant speedups if users can tolerate small inaccuracies. For example, when sampling at 10%, EmApprox speeds up a set of queries counting phrase occurrences by almost 10x while achieving estimated relative errors of less than 22% for 90% of the queries.

1.2.3 VidApprox

Volumes of video data have been growing at an unprecedented rate: over 300 hours of Youtube videos are uploaded every minute [20], surveillance cameras are ubiquitous in every major city. Video data contain rich and high-quality information. An an increasing number of industries/business, ranging from smart city initiative to marketing, have turned to extracting insights from video [21]. Video query often involves examining the contents of video frames, which would involve object detection: e.g., one may be interested in identifying frames that have at least an object of class X, or estimating average number of cars per frame etc. Advancement of convolutional neural networks (CNN) has led to accurate objection detection and image classification [22]. However, deep CNN inference is very expensive to apply at scale [23]. For example, using an state-of-the-art object detector such as YOLOv2 [24] to identify frames with a given object class (e.g., truck) on a month-long traffic video can take roughly 190 hours on a high-end GPU (such as NVIDIA P100) [23].

Large-scale video workloads such as autonomous vehicle development, urban planning, etc [21, 25, 26], are often run on distributed systems. The typical workflow starts with capturing, followed by storage then retrieval and finally the consumption of video frames [21]. More often than not, the consumer operator will only need to process frames that are relevant by supplying a predicate. While processing a large video dataset, much inefficiency comes from processing frames that are irrelevant especially when downstream processing is costly (e.g., involving CNN inference). For example, suppose an application is to perform object detection and OCR on the license plates of cars that appear in any frame. A naive implementation would read the frames sequentially without knowing ahead of time whether a frame has any car object present or not. If a frame does not have a car, then performing object detection and OCR on this frame would be wasteful as both operations are relatively costly. Therefore, the end-to-end latency will be greatly reduced if the application can directly retrieve frames of interest, i.e. frames that have cars.

It is common for large-scale video applications to subsample frames to keep cost manageable. Then a natural question is how to generate a sample that results in the most efficient utilization of limited resources. That is, relevant data to a query should have higher probabilities of being chosen. For example, if the likelihood of some regions in the video containing red car is high, then the search for frames with red cars should be biased toward these regions. The other challenge is then to efficiently retrieve the frames for further distributed processing. Suppose the dataset is stored across many servers, and the selected frames only concentrate in a small number of servers, then I/O may bottleneck the processing. Therefore, it is desirable to distribute the I/O load among different servers.

Large video data often contains regions that are irrelevant. Processing video frames that are of interest will result in efficient utilization of limited computing resources. Video analytics pipeline starts with video data capturing and storage, followed by retrieval and finally consumption. We design and implement a approximate video analytics framework called VidApprox for accelerating video queries that involve object detection. VidApprox provides integrated support for video queries through indexing and placement of frames through approximation in a distributed system. VidApprox first compresses/encodes frames into segment. It then leverages cheap object detection CNNs to learn vector representations of video segments, and further processes the vectors as index structure. The primary objective of indexing is early pruning of irrelevant segments. It uses cheap CNNs to learn feature vectors in offline indexing, that is used to index and cluster segments that are similar in the same video. At query time, a subset of most similar clusters of segments to the query are retrieved facilitated by the vector-based index. The size of the subset is given by user input. Our evaluation shows that VidApprox can achieve a significant speedup with a small accuracy loss in both video aggregation and retrieval tasks. VidApprox can also significantly outperform uniform random sampling over video segments in terms of approximation quality. For example, at 10% segment sampling rate, VidApprox can achieve a speedup of almost 10^3x compared against processing all frames, with 8% error bound for aggregation tasks that count the number of the queried object.

Our works target different approximation scenarios, since approximation can present different challenges when applied to different types of datasets and domains. ApproxSpark is more effective when we have large log datasets that need to be aggregated once or a few times, since the preprocessing cost may outweigh the time savings gained by approximation. EmApprox and VidApprox are more effective when users may pose repeated queries to the same dataset, so that preprocessing cost of the data can be amortized across runs. They both demonstrate that an index structure learned from the dataset as compact representation can guide the approximation of text and video datasets. We also believe our approach of "learning an index" from the data is not limited to text and video data, the general technique should be applicable other data with inherent distribution to extract such as genomics, time series, etc.

1.3 Organization

The remainder of the thesis is organized as follows:

In Chapter 2, we present the design and implementation of ApproxSpark.

In Chapter 3, we present the design and implementation of EmApprox.

In Chapter 4, we present the design and implementation of VidApprox.

Finally in Chapter 5, we conclude our works and lay out an outlook for future work.

Chapter 2
Approximation with Error Bounds in Spark

In this chapter, 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 [27] data processing cluster reveals that 90% of 2,000 data mining jobs were aggregations [28]. It is also an important component in online analytical processing systems that are often used for summarizing data patterns in business intelligence [29,30].

Spark is a popular data processing system that has been widely adopted in different domains [31–34]. 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 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

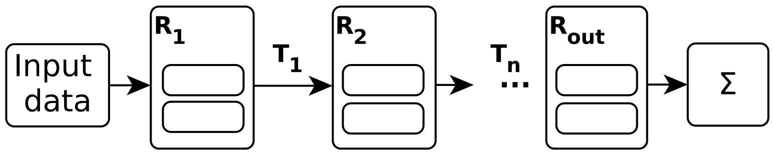


Figure 2.1: A Spark computation comprising a chain of transformations ($\{T\}$) followed by an aggregation.

transformations of the input data before the final aggregation [35, 36], making it challenging to compute error bounds. Consider a Spark computation comprising a chain of transformations ending with a summation as shown in Figure 2.1. 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 [10]. 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 [10], 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 use a combination of partition and data item sampling to achieve the right balance between precision and performance. As we discuss in Section 2.2, 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 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 2.2, 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 a second 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) [37] 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 2.3).

We have implemented the proposed framework in a prototype system called ApproxSpark (Section 2.4). 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, including text mining, graph analysis, and log analysis. We use the applications to evaluate ApproxSpark and explore the tradeoffs between performance and 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.

2.1 Background and Related Work

Spark. Spark introduces RDDs, which are fault-tolerant collections of data partitioned across server clusters that can be processed in parallel [38]. 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, so that a Spark program may be running multiple computations in parallel.

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 [11, 13, 28, 39–42] or online aggregation to sample data and produce a result within a time-bound [9, 43]. BlinkDB [42] maintains a set of offline-generated stratified samples by using an error-latency profile based on past queries. Sapprox [11] collects the occurrences of sub-datasets in offline preprocessing and uses them 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 [6].

Online sampling. ApproxHadoop [6] introduces approximation to the MapReduce paradigm [44]. 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 [45] approximates stream processing workloads based on Spark Streaming [46]. MaRSOS [41] 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 chapter, 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 [6].

2.2 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) loading a randomly selected subset of data items from each selected partition into R_{in}, and (3) computing 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 [10]:

$$\hat{\tau} = \frac{N}{n}\sum_{i=1}^{n} v_i \tag{2.1}$$

$$\hat{\tau} = \frac{N}{n}\sum_{i=1}^{n}(\frac{M_i}{m_i}\sum_{j=1}^{m_i} v_{ij}) \tag{2.2}$$

$$\hat{V}(\hat{\tau}) = N(N-n)\frac{S_u^2}{n} + \frac{N}{n}\sum_{i=1}^{n} M_i(M_i - m_i)\frac{S_i^2}{m_i} \tag{2.3}$$

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. Unfortunately, sampling R_o only reduces the execution time of the summation. For many Spark programs, this will lead to minimal savings because the sequence of transformations dominates overall execution time. Thus, we are motivated to sample earlier in the computation. In fact, if we can sample when the input is being read into R_i, we will reduce both the I/O time and the execution time of the entire sequence of transformations and the summation.

Now consider a program where R_{in} is transformed by a sequence of transformation $T_0, T_1, ..., T_n$ 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 `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 2.2 and 2.3. 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.

2.2.1 Multi-stage sampling

Consider the Spark program and its execution as shown in Figure 2.2. 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_3 and c_2:e_4 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 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.

The `filter` and `sample` transformations, on the other hand, can produce zero

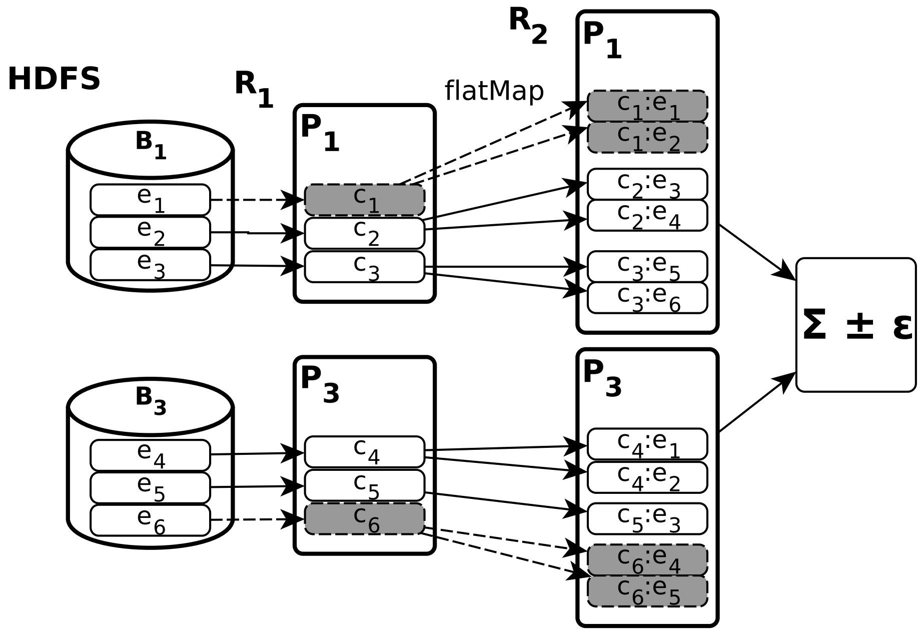


Figure 2.2: HDFS blocks and input data items are sampled when read into RDD R_1. Block B_2 not shown has been dropped. Gray boxes are dropped data items. In R_2, $c_i : e_j$ means data item j is generated from the data item i in the input partition.

output items for each input item, corresponding to an one-to-zero mapping. In this case, if we apply a `map` or `flatMap` after sampling or filtering an RDD, we cannot deduce the impact of clusters or items not chosen from the input RDD had `filter` and `sample` not applied on the output RDD; i.e., some cluster/items not chosen from the input RDD would have produced items in the output RDD, while others would not have. This introduces the need to estimate the population of the output RDD from the sample size to apply the cluster sampling theories for estimating the sampling error.

Below, we generalize the two-stage sampling equations (Eq (2.2) and (2.3)) into recurrences for multi-stage sampling with estimated sum and variance. We use $I_k = i_0, i_1, ..., i_k$ 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.

Consider an application that computes the distribution of page lengths within a Web site. The input is a set of all Web pages, with the first transformation producing a key/value pair $(length, 1)$ for each page, while the action is to count the number of pages for each distinct length. In this case, the input dataset really contains a mix of populations (pages of different lengths) that are being counted. The first transformation and the data shuffle that occurs between the transformation and action effectively sort the input into a set of different populations, each population corresponding to the pages of a particular length. Each population is then counted by the summation on each different length. This implies that the number of items in the input does not correspond to the size of any of the subpopulations. We will need to maintain sufficient information to estimate the size of each subpopulation, and compute its impact on *$V(\hat{\tau})$*. In this case, we will still need to estimate the subpopulations of each key.

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

$$\hat{\tau}_{I_k} = \begin{cases} \frac{N_{I_k}}{n_{I_k}} \sum_{j=1}^{n_{I_k}} \hat{\tau}_{I_k,j} & 0 \le k < d \ , \\ v_{I_k} & k = d \end{cases} \tag{2.4}$$

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 $i_0, i_1, ..., i_k, j$ 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:

$$\hat{V}(\hat{\tau}_{I_k}) = \begin{cases} N_{I_k}(N_{I_k} - n_{I_k})\frac{S^2_{u,I_k}}{n_{I_k}} + \frac{N_{I_k}}{n_{I_k}} \sum_{j=1}^{n_{I_k}} \hat{V}(\hat{\tau}_{I_k,j}) & 0 \le k < d-1 \ , \\ M_{I_k}(M_{I_k} - m_{I_k})\frac{S^2_{i,I_k}}{m_{I_k}} & k = d-1 \end{cases} \tag{2.5}$$

where $\hat{V}(\hat{\tau}_{I_k})$ is the variance of $\hat{\tau}_{I_k}$, S^2_{u,I_k} 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^2_{i,I_k} 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 compute the confidence interval as: $\hat{\tau}_0 \pm \epsilon$, 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) [10].

2.2.2 Data provenance tree

We now show how a data provenance tree can be constructed to capture the multi-level (multi-stage) clustering of items in the output RDD resulting from sampling while executing a transformation chain. We show how a provenance tree can be used to compute Equations (2.4) and (2.5) in the next subsection. Note that a tree is constructed in parallel, with sub-trees being constructed on servers processing RDD partitions. Figure 2.3 shows the data provenance tree corresponding to the sampling and execution shown in Figure 2.2.

Overview. A provenance tree starts with a root node that represents the top level clustering of all data items in the output RDD. Then, a level is added to the tree whenever a transformation leads to a new clustering level; e.g., the execution of a

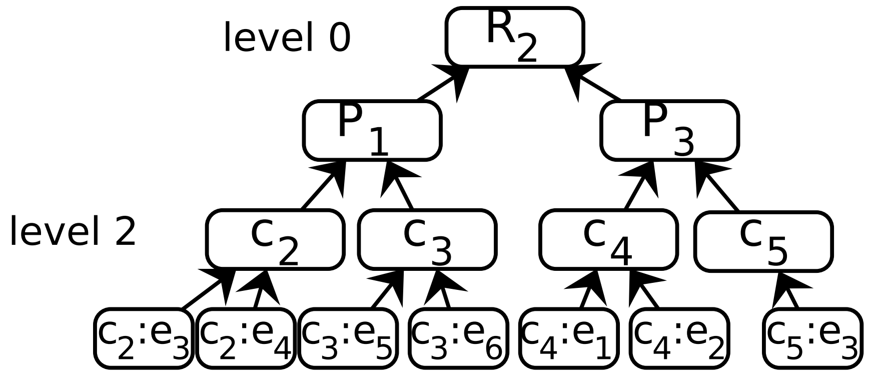


Figure 2.3: The provenance tree built for the sampling and execution shown in Figure 2.2.

Transformation	**Functionality**
`map(f)`	Applies f to each input data item, producing one output data item.
`flatMap(f)`	Applies f to each input data item, producing possibly multiple output data items.
`mapValues(f)`	Applies f to the value of each input key/value pair, producing one output key/value pair.
`filter(p)`	Outputs each input data item that satisfies the predicate p.
`sample(r)`	Samples from the input RDD using sampling rate r and outputs the sample.

Table 2.1: Spark transformations that are supported in ApproxSpark for sampling-based approximation with error bounds.

flatMap may add a level to the tree since it corresponds to a one-to-many mapping of input to output data items, whereas the execution of a map will not.

Nodes at each level of the tree represent sampling units at the corresponding clustering level. Internal nodes correspond to clusters, while leaves correspond to data items in the output RDD. Each leaf node contains the corresponding value (or key/value pair) in the output RDD. Each internal node stores intermediate values in support of computing the recurrences in Equations 2.4) and (2.5 (which are not computed until after the entire tree has been built). An edge relates each sub-cluster/data item to its parent cluster. Table 2.1 shows the subset of Spark transformations that ApproxSpark currently handles.

Tree building. We introduce Algorithm 1 for building data provenance trees. Table 2.2 describes the subroutines that are not explicitly defined. We assume that input partitions and data items are sampled when input data is being loaded, so a provenance tree will always have at least three levels: the root (level 0), the first level of clustering defined by sampling of the RDD partitions, and the selected data items from the chosen partitions.

Lines 1 to 9 comprise the main routine, taking as input a transformation chain $\{T\}$, partition sampling rate $pRate$, and input data item sampling rate $iRate$. Lines 2 to 6 are executed sequentially, creating the root node, then sampling the input partitions with rate $pRate$ and adding the sampled partitions ($\{P\}$) to the tree as children of the root. We use variable $rate_k$ to keep track of the sampling rate at every tree level. Line 8 is the parallel execution of the buildSubtree routine for each partition in $\{P\}$, which builds a sub-tree rooted at each node in level 1 as each transformation executes.

Lines 11 to 13 add the sampled input data items in each partition to the tree. In order to do that, a new node is created for each selected data item in $\{DI\}$ and those nodes are inserted in a new level. The rest of the algorithm updates the tree based on the semantics of each transformation T_i in T. If T_i is sample, it replaces the nodes in the last tree level with a set of nodes $\{node\}$ generated by sampling the previous level, then updates $rate_k$ by multiplying it with the sampling rate. If T_i is flatMap and there is a sampling operation before it, a new level is added because sampling data items before applying `flatMap` is equivalent to selecting groups of data items generated from this `flatMap`, thus adding a new level of clustering. In other cases, the last level's nodes will be replaced by the new set of nodes $\{node\}$ without adding a new level. Note that replacing nodes will also maintain the appropriate parent/child relationships.

Multi-key computation. A transformation can produce multiple keys, and a transformation chain can lead to multiple aggregations over different keys in the output RDD. However, only each final output key, rather than intermediate keys, 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 with the same key. 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 the number of partitions stay the same after loading the input data. We do not consider `filter`'s effect over the sampling error; specifically we do not account for 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.

2.2.3 Tree traversal-based statistics computation

Equations (2.4) and (2.5) can be computed by traversing the provenance tree built using Algorithm 1. The tree is traversed level by level starting from the leaves, and the estimated sum/variance for each cluster represented by an internal node are computed incrementally using information stored in the node's children. We introduce Algorithm 2 for this computation. Lines 3 to 7 compute in parallel each partition's statistics by

Algorithm 1: Building data provenance tree

```
1  Algorithm DataProvenance($\{T\}$, $pRate$, $iRate$)
2      createRoot();                                    // root is at level 0
3      $\{P\}$ = sampleInputPar($pRate$);
4      $\{node\}_P$ = 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     $\{DI\}$ = sampleInputDI($iRate$);
12     $\{node\}$ = createNodes($\{DI\}$);
13     addLevel($\{node\}$);
14     $rate_2 = iRate$;
15     $k = 3$, $rate_3 = 1.0$;
16     for $T_i \in \{T\}$ do
           /* data items generated by $T_i$                                  */
17         $\{DI\}$ = exec($T_i$);
           /* each tree nodes corresponds to a data item in $\{DI\}$         */
18         $\{node\}$ = createNodes($\{DI\}$);
19         if $T_i$ is sample then
20             replaceLast($\{node\}$);
21             $rate_k$ *= sample.$rate$;
           /* if $T_i$ is flatMap and there is sampling operation before
              it, then a new level needs to be added                         */
22         else if $rate_k < 1.0$ and $T_i$ is flatMap then
23             addLevel($\{node\}$);
24             $rate_k = 1.0$, $k$ ++;
25         else if $T_i$ is map or flatMap or mapValues or filter then
26             replaceLast($\{node\}$);
27     end
```

Subroutine	**Semantics**
`sampleInputPar(rate)`	Samples input partitions with *rate* and returns selected partitions.
`sampleInputDI(rate)`	Samples input data items in the selected partitions with *rate* 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, and 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 (e) shares the parent of the node that has generated e in this transformation. Then the nodes that were originally in the last level are deleted.

Table 2.2: Description of subroutines used in Algorithm 1.

calling the subroutine `ComputeNodeI`, which computes Equations (2.4) and (2.5) of a given node at level k. Line 8 computes the root's estimated sum/variance for the final confidence interval output. We illustrate the computation process by using the tree in Figure 2.3 representing three-stage cluster sampling as an example. We begin at the level where $k = 2$, where we first compute the intra-variance of c_2 formed by $c_2 : e_3$ and $c_2 : e_4$ using the subroutine `ComputeNodeI` shown in Algorithm 2. After computing other clusters (c_3, c_4 and c_5) in the same level, we decrement k to 1 and move to second level nodes. Computing statistics (e.g., estimated sum, 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 sum/variance at the root comprising P_1 and P_3 can be computed using `ComputeNodeI`.

2.2.4 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. Since we adopt Bernoulli sampling to at each cluster level, 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 (2.5) since their estimators affects the variance. We have detailed derivation on incorporating it into the variance computation in the Appendix of our technical report [47].

Logically, an RDD could be viewed as the *zeroth* level cluster containing partitions,

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, $d$);
       /* turns estimator and variance into confidence interval     */
9      return ConfidenceInterval($root.\hat{\tau}, root.\hat{V}$);
10 subroutine ComputeNodeI($node$, $k$, $d$)
11     $\{c\} = node.children$;
       /* computes Eq 3, 4 for each tree node                        */
12     if $k$ is $d$ then
13         $node.\hat{\tau}$ = data item's value;
14     else if $k$ is $d - 1$ then
15         $m_{I_k} = \{c\}.size()$, $S^2_{i,I_k} = Var(\{c.\hat{\tau}\})$;
16         $node.\hat{\tau}$ = Eq2.4($\{c.\hat{\tau}\}, m_{I_k}, M_{I_k}$);
17         $node.\hat{V}$ = Eq2.5($m_{I_k}, M_{I_k}, S^2_k$);
18     else
19         $n_{I_k} = \{c\}.size()$, $S^2_{u,I_k} = Var(\{c.\hat{\tau}\})$;
20         $node.\hat{\tau}$ = Eq2.4($\{c.\hat{\tau}\}, n_{I_k}, N_{I_k}$);
21         $node.\hat{V}$ = Eq2.5($n_{I_k}, N_{I_k}, S^2_{u,I_k}, \{c.\hat{V}\}$);
22     end
```

Algorithm 3: Computing equation 2.5

```
input  : sampling rates, data item clustering info, sampling stages $d$
output: Overall estimated variance
1  $k = d$ ;
2  Using equation 2.5:
3  while $k \geq 0$ do
4      if $k = d$ then
5          Compute each $\hat{V}(\hat{\tau}_{I_k})$ from the sample under the case when $k = d$, using
             the estimated value for $N_{I_k}$;
6      else
7          Compute each $\hat{V}(\hat{\tau}_{I_k})$ from the sample under the case when $0 \leq k < d$
             using the estimated value for $M_{I_k}$;
8      end
9      $k- = 1$
10 end
```

and the sub-clusters within each cluster are its "data items" containing sub-clusters, with only the final output data items in R_{out} being monolithic. It is convenient to have this nested view because both sampling the output data items and clusters can be modeled as sampling the immediate higher level clusters' data items. For example, partition sampling can be viewed as sampling the data items of the *zeroth* cluster, the entire RDD. Estimated statistics such as sums, variance, cluster population sizes can all be computed backward from the output data items in R_{out} to the second cluster level, until the *zeroth* cluster level in a recurrence fashion.

The strategy we identify cluster sampling stages is through the appearances of `flatMap`, which generates a group of data items from one data item. Thus it establishes one-to-many relationship between an data item in the RDD on which `flatMap` is applied and the resulting group of data items in the following RDD. We treat implicitly the creation process of the first RDD in a transformation as the first `flatMap`. For example, in Figure 2.2, the raw input is seen as a single data item and the HDFS blocks are modeled as the result of an application of `flatMap` on the singular input which generates a number of blocks. On the other hand, `map` only generates one data item from one data item, which forms one-to-one relationship between them. Thus `map` does not create new cluster sampling stages, which means that RDDs created by consecutive `map`s would be collapsed as the same sampling stage.

The goal of mapping data items and RDD partitions to multi-stage sampling is to form a confidence interval for the estimated result, which requires estimating the sampling error for the estimator using information on how data items/partitions form multiple levels of clusters.

2.3 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 a one-pass sampling algorithm *Adaptive Stratified Reservoir Sampling* (ASRS) [37] to address the rare key issues. ASRS combines stratified and reservoir sampling [10,48], and uses power allocation [49] 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 [37].

Stratified sampling partitions a heterogeneous population into disjoint homogeneous subgroups (strata), from which a random sample is taken from each. On the other hand, reservoir sampling [48] is a sampling algorithm for selecting a uniform random sample from an input stream by storing randomly selected items in the reservoir. What makes reservoir sampling attractive is that it samples online and only makes one pass over the data.

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 may have significant execution time saving since it reduces I/O time. We can estimate the result and the error bound using standard multi-stage sampling theory using Eq (2.4) and (2.5), because an ASRS sample is very close to a simple random sample [37].

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.

2.4 ApproxSpark implementation

We have implemented our approximation framework by modifying and extending the original Spark system. We extend the 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 the user to set the degree of approximation, either by specifying the sampling rates or error bound targets. In addition to setting specified sampling rates, the user is also able to set target error bounds at different percentiles on the error bound CDF of all keys. For example, the user may specify that the 10^{th} percentile of the error bound is at most 0.1, the 50^{th} percentile at 0.3, the 90^{th} percentile at most 0.6.

2.4.1 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. We extend the implementations of the transformations shown in Table 2.1 in the 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 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 2.4. 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 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 final aggregation, the statistics of each partition are sent to the reduces for computing the inter-cluster variance among the RDD partitions and the final confidence interval. As transformations $T_1 \rightarrow T_n$ execute

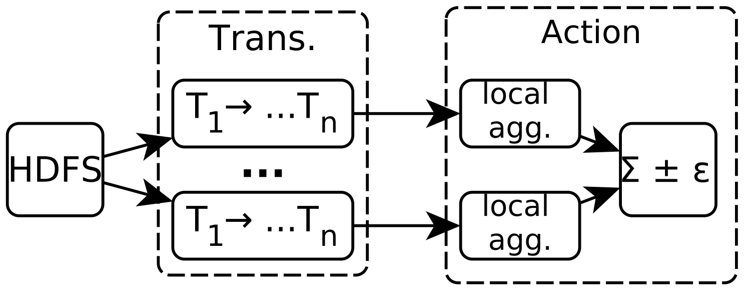


Figure 2.4: Error bound computation process, divided across the transformation and action phases. The tree building happens in the transformation phase, and the error bound computation happens in the action phase.

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.

```
val sc = new SparkContext(new SparkConf())
```

```
.setAppName("WordCount"))
// p1: partition sampling rate
// p2: input data item sampling rate
val tokenized = sc.textFileWithSampling(args(0), p1, p2)
.flatMap(_.split(" ,;."))
// count the occurrence of each word
val wordCounts = tokenized.map((_, 1))
.countByKeyApproxMultiStage()
// containing statistics of each word
wordCounts.saveAsTextFile()
```

Listing 2.1: ApproxSpark approximate word count

2.4.2 User-specified target error bounds

Algorithm 4: Two-stage sampling rates search

```
input : $t_{perc1}$,...$t_{percn}$, $pCluster = 1.0$, $pItem = 1.0$
output: $pCluster$ and $pItem$
1  ————————————-Phase I————————————-
2  Execute pilot tasks and return partially aggregated partitions to the driver;
3  Estimate $M$, $S_i^2$ and $S_u^2$
4  while None of the predicted error exceeds the target error do
5      Use $pCluster$ to compute error bounds in $perc1$, $perc2 ... percn$ ;
6      $pCluster$ -= $bstep$;
7  end
8  $pCluster$ += $bstep$;
9  fix $pCluster$, find $pItem$ in the same way as $pCluster$;
10 ————————————-Phase II————————————-
11 Continue the remaining Spark tasks setting sampling rates to be $pCluster$ and
     $pItem$;
```

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 2.5 shows the architecture of user setting error bounds.

The algorithm exploits a property that partition sampling may incur more sampling error [10], but reduces more execution time compared with data item sampling. Our algorithm follows a principle in online aggregation - *minimum time to accuracy* [9], 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.

Discussions. Our proposed searching algorithm only considers partition and data item sampling rate for the input RDD. We think that sampling the input RDD is sufficient to meet the error bound targets without too much added complexity. It is because sampling operations placed far from the input will have diminishing effect over the error bound as well as execution time. In fact, expanding the algorithm to search for

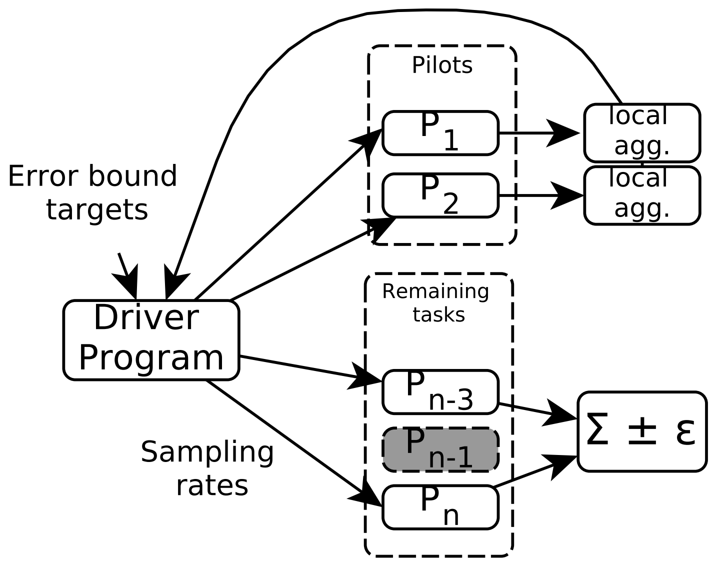


Figure 2.5: Design of the algorithm for automatically selecting sampling rates based on user specified target error bounds. Gray shaded box is dropped partition(s).

more than two sampling rates is straightforward, as we can add more while loops to find sampling rates for following RDDs down the transformation chain.

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.

2.5 Evaluation

We evaluate ApproxSpark using five real world applications from different application domains (see Table 2.3). 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

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

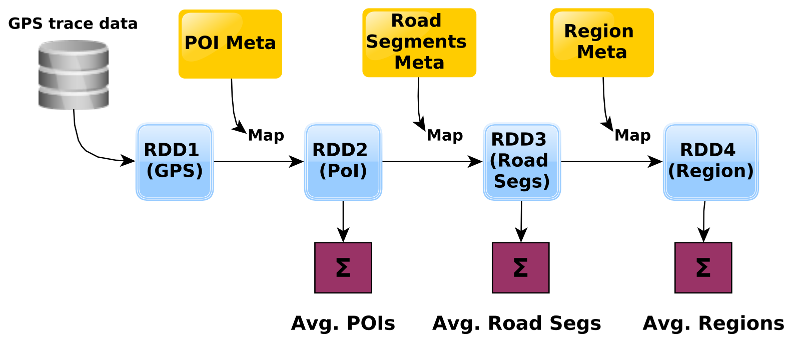


Figure 2.6: Multi-step transformation for Speed

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.

2.5.1 Applications

Word Co-occurrence (Co-occur). Co-occurrence is a common text mining application that computes the frequencies of pairs of words [50]. In this study, the application counts co-occurrences of topic tags in the MEDLINE database [51], containing more than 20M citation records of publications 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. [52]. We analyze a taxi GPS dataset containing status records collected every 30 seconds from 14,000 taxis operating in Shenzhen, China, over one week [53]. Each record contains information about a taxi, including a timestamp and the taxi's GPS location and speed. The dataset has ∼291M records that covers an area of ∼790 square miles divided into 491 regions, containing ∼569k POIs and ∼198k 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.

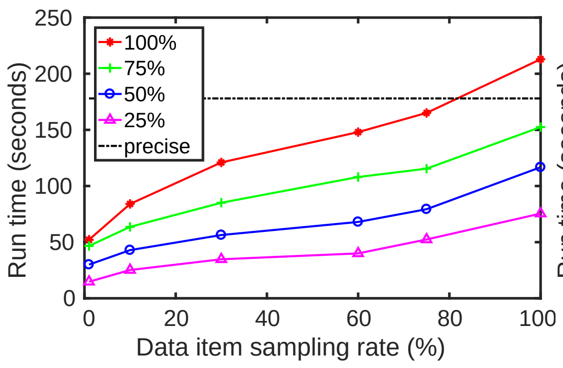
Twitter Hashtags Sentiment Analysis (Twitter). Sentiment analysis computes quantitatively whether a piece of text is positive, negative or neutral using natural language processing (NLP) techniques [54]. In this study, the application computes the average sentiment for each unique hashtag in the Tweets2011 Twitter dataset from TREC 2011 [55], using the Stanford CoreNLP library [56]. This dataset contains ∼16M 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 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 [57]. We use the Wikipedia data snapshot from 2016 with ∼5M articles [58]. 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.

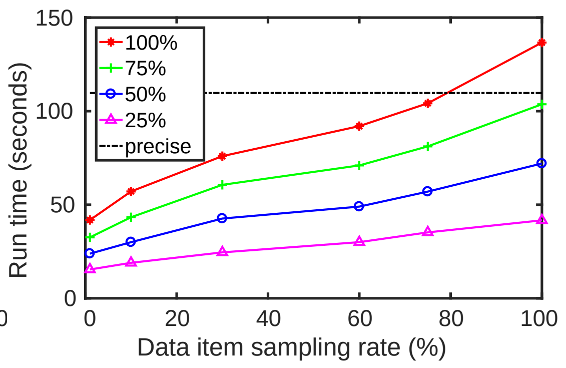
WikiClickstream (Clickstream). Clickstream analysis can be used to generate a weighted network of linked articles showing the probability of users navigating from one article to another. We use a Wikipedia clickstream dataset from 2016 [59] containing ∼149M tuples of (source, destination, count), where 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.

2.5.2 Results for multi-stage sampling

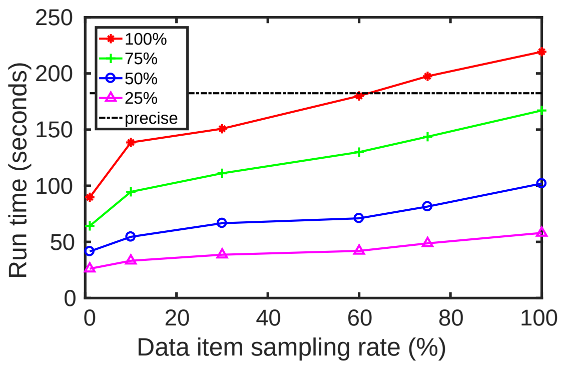
We explore the performance and accuracy of multi-stage sampling using four of the above applications: Co-occur, Twitter, WikiPageRank, and WikiClickstream. In most experiments, we sample the input data as it is read into the first RDD because this will lead to the highest speedups. However, we also explore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



(a)



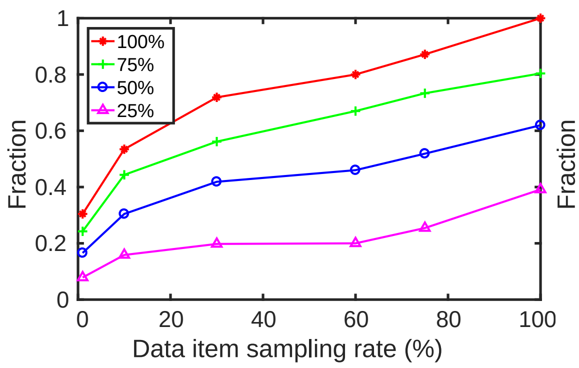
(b)



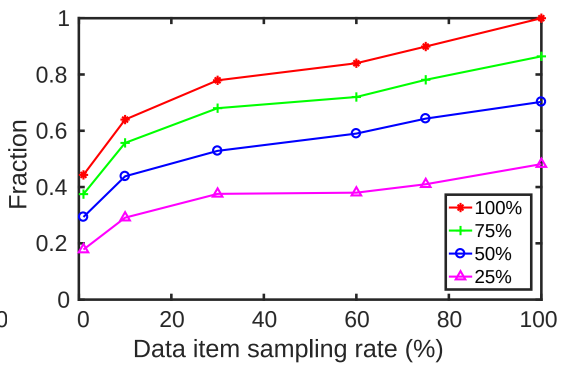
(c)

Figure 2.7: 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.

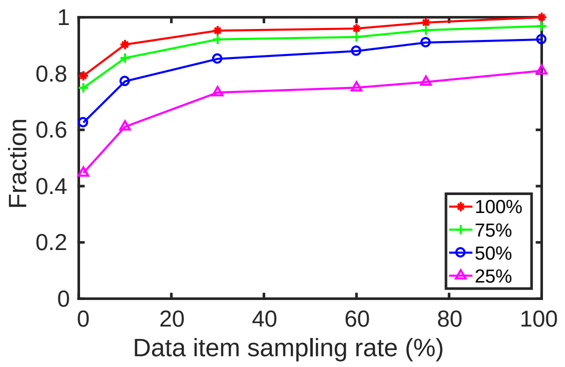
Execution times. Figure 2.7 plots the execution times for two of the applications, Co-occur, WikiPageRank at different partition and data item sampling rates. Consistent with previous results from [6], we observe that (a) multi-stage sampling significantly reduces execution times, and (b) partition sampling 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



(a) Co-occur

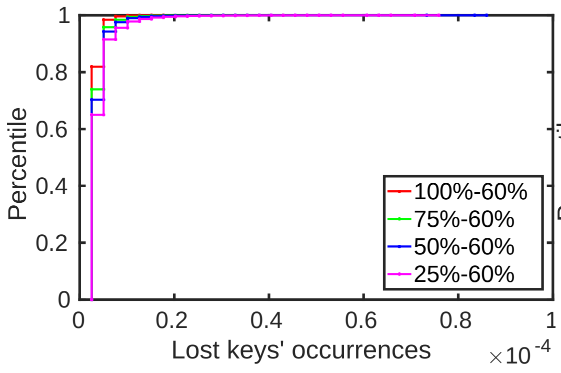


(b) WikiPageRank

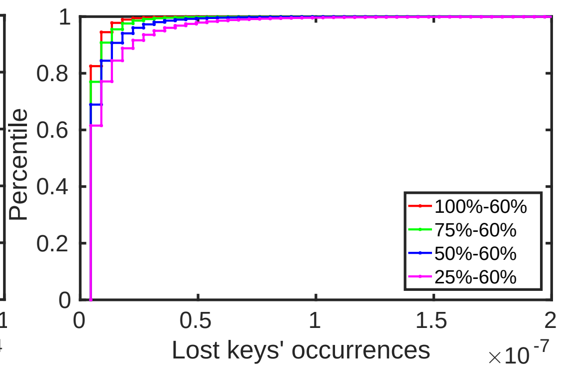


(c) WikiClickstream

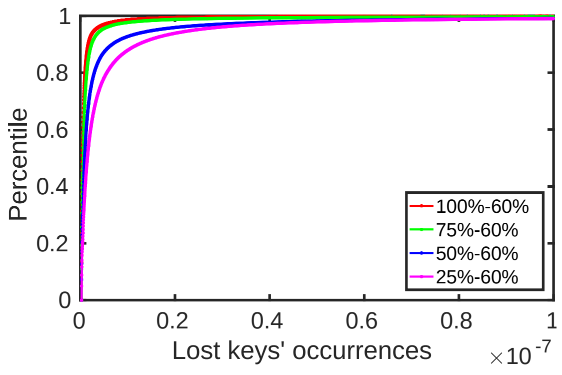
Figure 2.8: Fraction of unique keys (normalized against number of keys produced under precise execution) outputed under different sampling rates. Each line represents a particular partition sampling rate.



(a) Co-occur

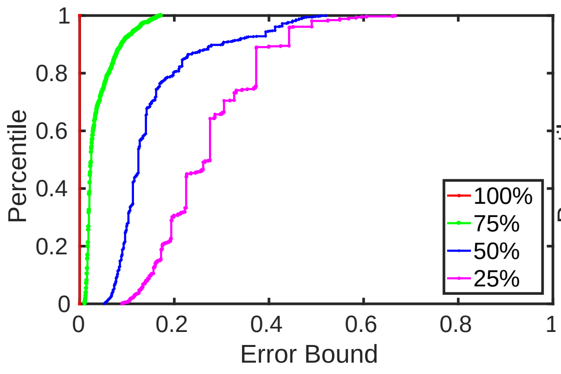


(b) WikiPageRank

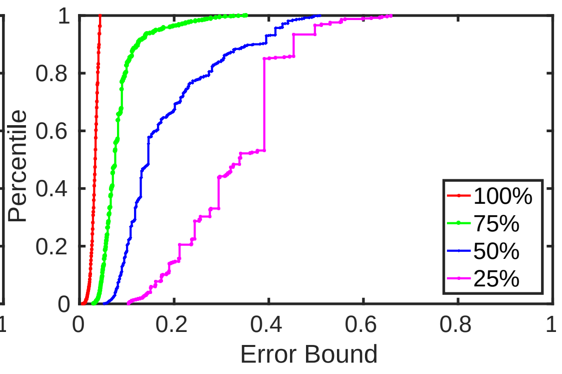


(c) WikiClickstream

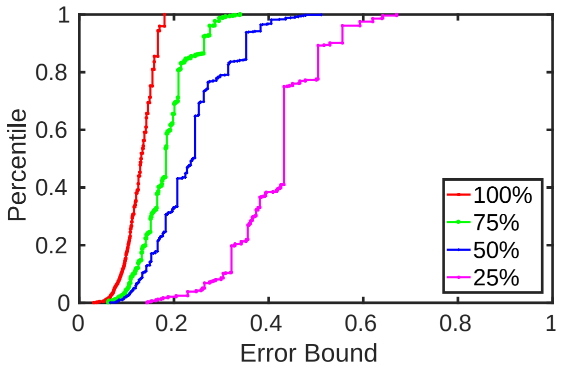
Figure 2.9: CDFs of occurrences of the lost keys, normalized against the total number of data items across all keys at a data item sampling rate of 60%. Each line corresponds to a specific partition sampling rate.



(a) Data item sampling rate - 100%.

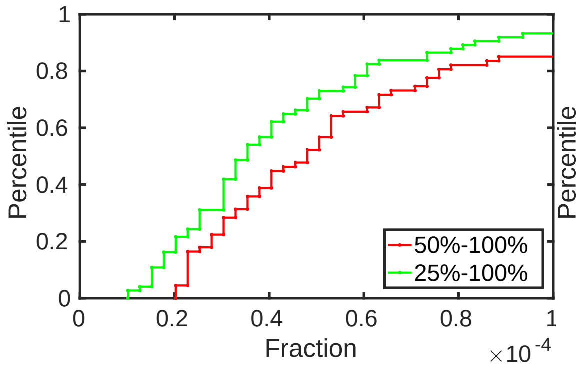


(b) Data item sampling rate - 75%.

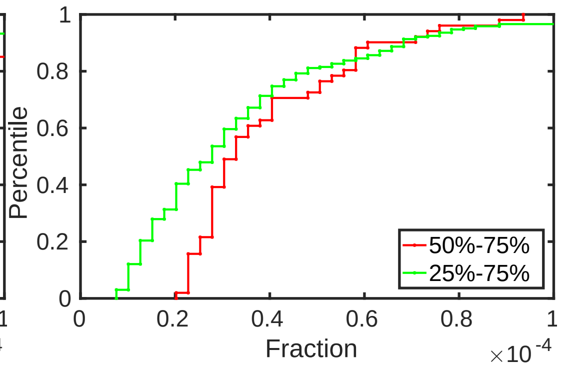


(c) Data item sampling rate - 60%.

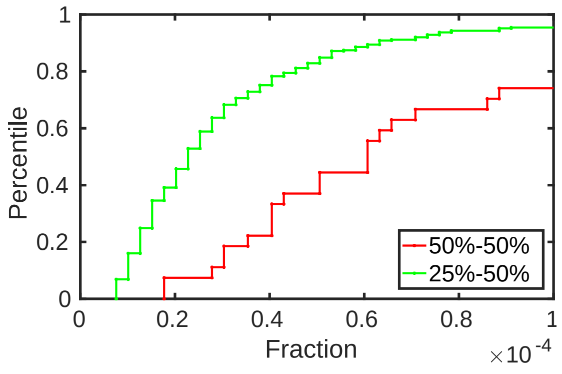
Figure 2.10: Each graph plots CDFs of errors with 95% confidence at a fixed input data item sampling rate Co-occur. Each line in a graph plots the error CDF at a particular partition sampling rate.



(a)

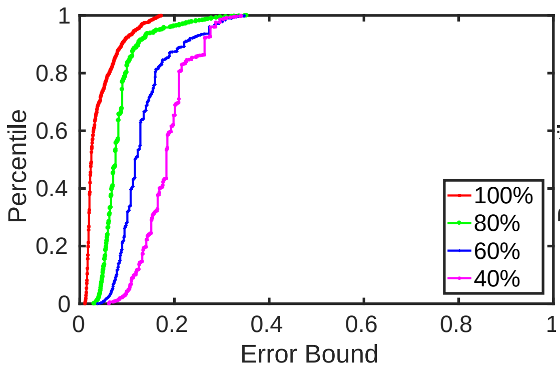


(b)

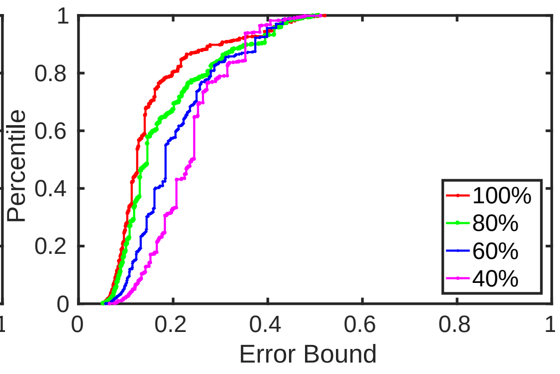


(c)

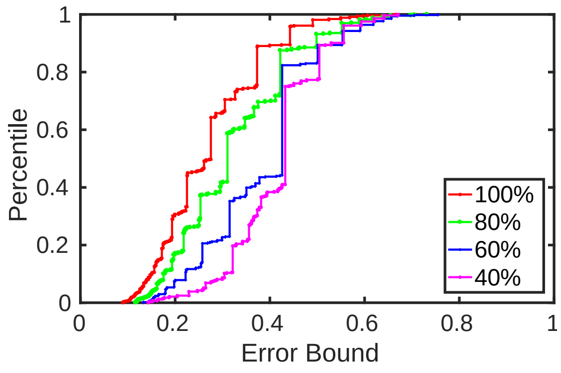
Figure 2.11: CDFs of the occurrences of keys with error bound of over 40%, normalized against the total number of data items across all keys, in the Co-occur application. The legends indicate partition and data item sampling rates respectively.



(a) Partition sampling rate - 75%.

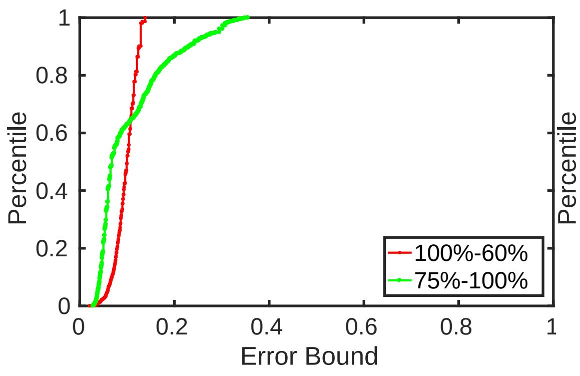


(b) Partition sampling rate - 50%.

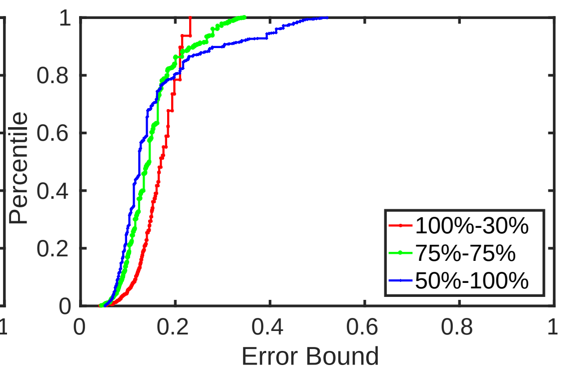


(c) Partition sampling rate - 25%.

Figure 2.12: 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.



(a)



(b)

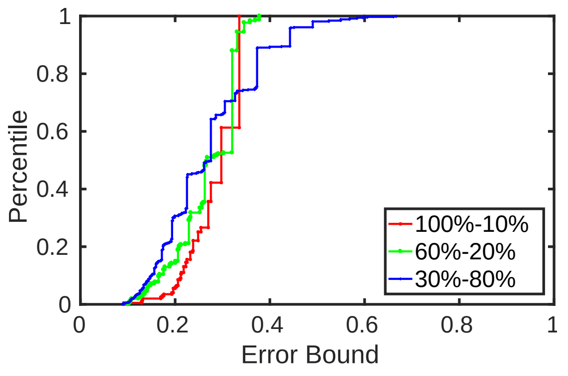
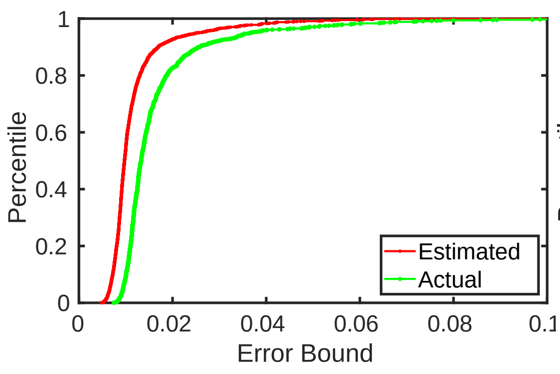


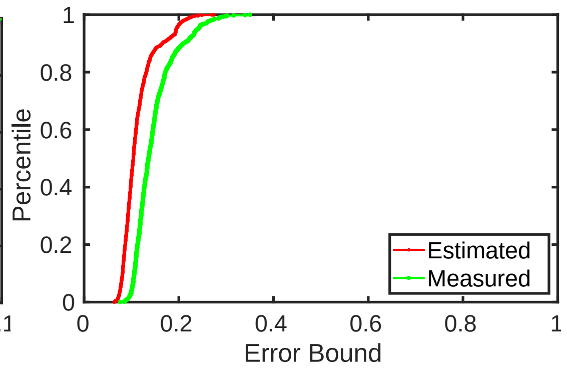
Figure 2.13: Error distribution trade off under different partition and data item sampling rates combination from the Co-occur application. The legends indicate partition and data item sampling rates respectively.

in memory, etc., whereas data item sampling 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 2.8 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 2.9 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



(a) WikiPageRank, 75%-50%



(b) WikiClickstream, 75%-50%

Figure 2.14: Estimated and actual relative error comparison.

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 2.9 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 2.10 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 [41], 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 2.10(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 2.10(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 2.12 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 2.13 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 2.7(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 2.13(a), (100%-60%) has better smaller errors after the 62^{th} 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 2.14 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

	Sampling Rates	
Source	**(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 2.4: Breakdown of uncertainty on average across all keys for the four sources of errors in multi-stage sampling for Co-occur.

information to the error estimation process than rare keys.

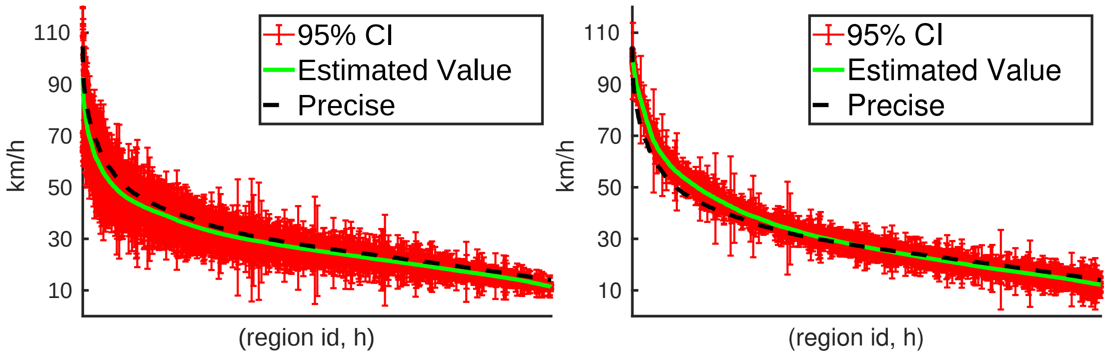
Effect of sampling data items. Figure 2.12 shows that the CDFs follow the same trend as varying partition sampling rates. However we observe that the tails tend to converge under the same partition sampling rates if we vary the input data item sampling rates. We see that varying the data item sampling rate shifts the body (lower percentiles) more rather than the tail portions of the CDF curves. The reason is that large errors come from rare keys that tend to concentrate in only a few RDD partitions in the resulting multi-stage sample. Since the partition sampling rates are the same, percentage of keys that have large errors tend to be similar across different data item sampling rates.

Effect of where to sample. Sampling at different parts of the RDD transformation chain can produce different error bound distributions. Figure 2.17 shows the error bound distributions when data item sampling occurs at different points in the transformation chain. Since data item sampling before the `flatMap` corresponds to dropping small clusters generated by the `flatMap`, we can see that under the same partition and data item sampling rates, sampling after the `flatMap` results in an error distribution that is smaller than sampling before the `flatMap`.

Sensitivity to underlying data distribution. In the dataset for our Twitter application, the tweets are ordered chronologically which implies that the keys (Hashtag)

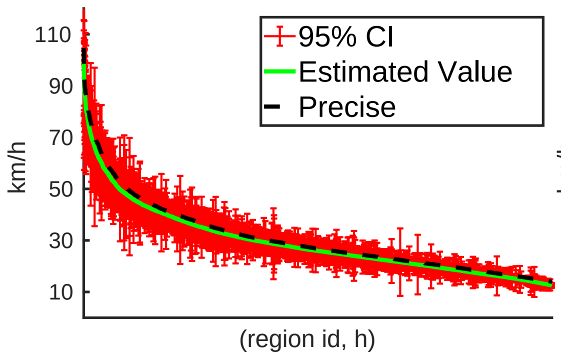
Sampling Rates	**Execution Time (s)**	**Error Bound Percentile 100^{th}**	**90^{th}**	**50^{th}**	**% Keys Present**
100%-60%	149.2	0.17	0.12	0.10	80.0
75%-100%	150.8	0.37	0.22	0.06	80.3
100%-30%	120.9	0.22	0.20	0.17	71.8
75%-75%	121.4	0.32	0.28	0.13	72.3
50%-100%	119.7	0.51	0.31	0.11	61.5

Table 2.5: Comparison of run times, error bounds at 100^{th}, 90^{th}, 50^{th} percentiles, and fraction of unique keys for Co-occurrence.

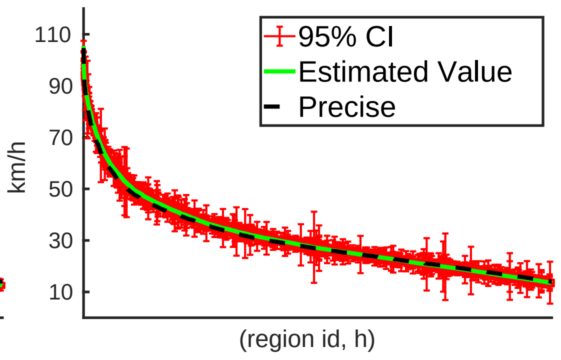


(a) 50%-25% SRS (b) 50%-25% Stratified

Figure 2.15: 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.

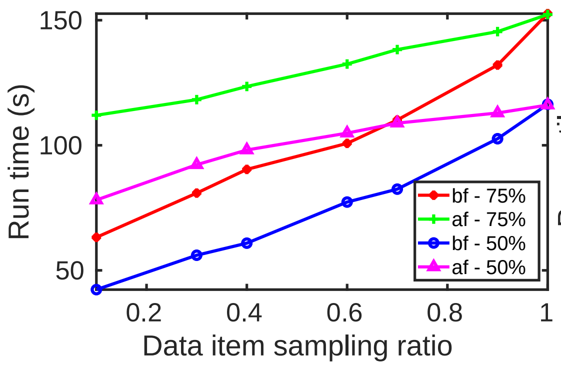


(a) 75%-50% SRS

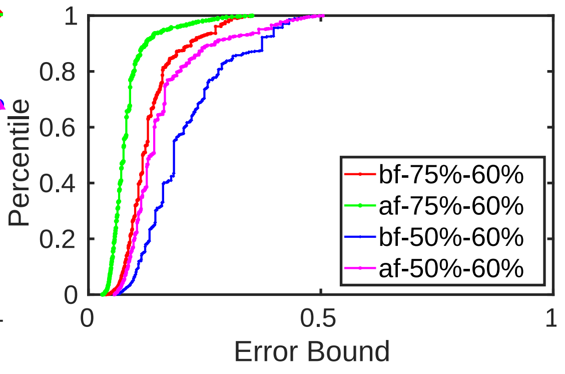


(b) 75%-50% Stratified

Figure 2.16: 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.



(a) Run time



(b) Error bound CDF

Figure 2.17: Run time comparison between data item sampling at the input data (before `flatMap`) and after `flatMap`, coupled with different input partition sampling ratios

tend to cluster. We explored shuffling the data items and see what effects it has over the result.

Sampling error. Sampling error is smaller in the shuffled data case. It because when data items is shuffled, there is less inter-cluster variance and less uncertainty in population size estimation. Figure 2.18 shows the comparison of sampling error CDFs between the shuffled and unshuffled data when applying a 50% partition sampling rate.

Fraction of keys shown in the output. Effect of sampling partitions misses fewer keys when the input data is shuffled, mitigating the clustering effect of the keys. So if temporal info is not important to the application, shuffling the data offline as preprocessing will improve the result in terms of sampling error and number of missed keys, assuming data will be reused.

Sources of uncertainty. As previously explained, uncertainties (leading to estimated error bounds) can arise from the sampling as well as population estimations. Table 2.4 shows the percentages of the error bounds, averaged across all keys in the output, attributable to each of four sources for Co-occurence. 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 [10]. 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 2.5 presents data for two sets of sampling rates for Co-occur, $\{(100\%, 60\%), (75\%, 100\%)\}$ and $\{(100\%, 30\%), (75\%, 75\%), (50\%, 100\%)\}$, where members 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

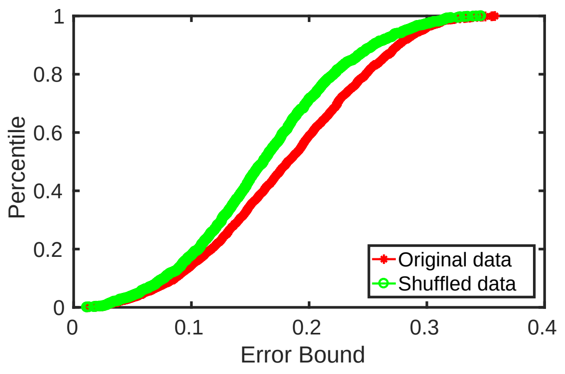


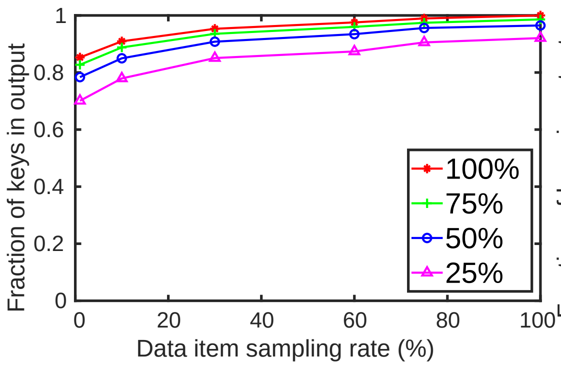
Figure 2.18: Comparison of error bound in twitter application between the shuffled data and unshuffled data when applying 50% partition sampling rate.

partition sampling rates (e.g., 50%) can clearly lead to significantly increased number of lost keys. Looking at Figure 2.7(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.

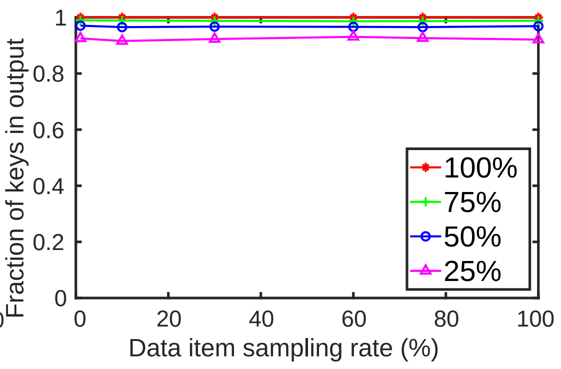
2.5.3 Results for stratified sampling using ASRS

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. We have observed 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



(a) SRS



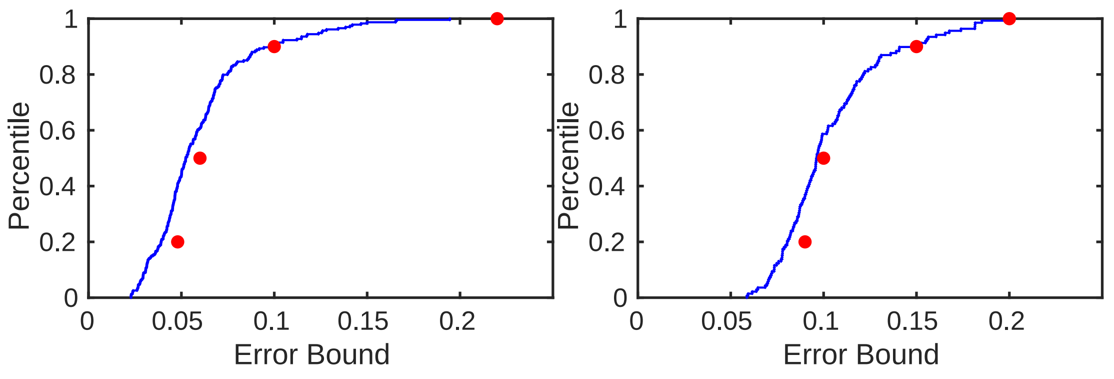
(b) Stratified

Figure 2.19: 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.

has much higher overhead than the SRS since it needs to perform stratification and power allocation steps over the key space.

Confidence interval. Figure 2.15 and 2.16 plot 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 2.15(a) and 2.16(a), 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 2.19, we see that the stratified sampling constantly loses less keys than SRS at same sampling rate. In Figure 2.19(b),



(a) Speed (b) WikiClickstream

Figure 2.20: User-specified error bounds targets shown on the error CDF achieved by ApproxSpark.

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 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.

2.5.4 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 2.20 show the target error bounds at 20^{th}, 50^{th}, 90^{th}, 100^{th} 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 2.20(a) is the error distribution for average Taxi speed with 50% partition, and 80% data item sampling rates at the POI RDD. Figure 2.20(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.

2.6 Conclusion

In this chapter, we have proposed a sampling-based framework that supports approximate data processing with estimated error bounds in Spark. The framework includes algorithms that track clustering of data items to be aggregated as the input data is sampled and transformed and use multi-stage sampling theories to estimate output aggregate values and corresponding error bounds. We have implemented our framework in a prototype system called ApproxSpark. We used ApproxSpark to implement five approximate applications from different application domains. Experiments with these applications show that ApproxSpark can effectively allow users to tradeoff precision for significantly reduced execution time, although in many cases, it must be acceptable to lose some output keys. We also used the applications to explore and discuss extensively tradeoffs between sampling rates, execution time, precision and key loss. Based on our experience and results, we conclude that frameworks such as the one we propose here can make efficient and controlled approximation more easily accessible to Spark programmers, as well as data processing systems similar to Spark.

Acknowledgement

We thank the Rutgers Discovery Informatics Institute (RDI^2) for partially supporting this research, as well as providing the computing platform for our experimentation and evaluation.

2.7 Appdendix: Cluster Sampling Variance with Population Estimation

Estimated sum for all clusters is:

$$\hat{\tau} = \frac{\hat{N}}{n} \sum_{i \in S} v_i = \hat{N}\bar{\tau} \tag{2.6}$$

Sample mean among the cluster totals is:

$$\bar{\tau} = \frac{1}{n} \sum_{i \in S} v_i \tag{2.7}$$

Estimated total number of clusters N is:

$$\hat{N} = \frac{n}{p_1} \tag{2.8}$$

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

$$Var(\hat{N}) = \frac{n(1-p_1)}{p_1^2} \tag{2.9}$$

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

$$Var(\bar{\tau}) = (1-p_1)\frac{s_t^2}{n} \tag{2.10}$$

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

$$\begin{aligned} & Var(\hat{N}\bar{\tau}) \\ &= \hat{N}^2 Var(\bar{\tau}) + \bar{\tau}^2 Var(\hat{N}) + Var(\hat{N})Var(\bar{\tau}) \\ &= (\frac{n}{p_1})^2(1-p_1)\frac{s_t^2}{n} + \frac{n(1-p_1)}{p_1^2}(\bar{\tau}^2 + (1-p_1)\frac{s_t^2}{n}) \end{aligned} \tag{2.11}$$

thus:

$$Var_{inter} = (1 - \frac{1}{p_1})Var_{srs}(\hat{\tau}) \tag{2.12}$$

Estimated sum of cluster i is:

$$\hat{\tau}_i = \hat{M}_i \bar{\tau}_i \tag{2.13}$$

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

$$\bar{\tau}_i = \frac{1}{m_i} \sum_{j \in S_i} v_{ij} \tag{2.14}$$

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:

$$\hat{M_i} = \frac{m_i}{p_2} \tag{2.15}$$

with variance:

$$Var(\hat{M_i}) = \frac{(m_i)(1-p_1)}{p_2^2} \tag{2.16}$$

The variance of sample mean in cluster i is:

$$Var(\bar{\tau_i}) = (1-p_2)\frac{s_i^2}{m_i} \tag{2.17}$$

The variance of estimated sum in cluster i is:

$$\begin{aligned} Var(\hat{\tau_i}) &= \hat{M_i}^2 \hat{Var}(\bar{\tau_i}) + \bar{\tau_i}^2 Var(M_i) + Var(\hat{M_i})Var(\bar{\tau_i}) \\ &= (\frac{m_i}{p_2})^2(1-p_2)\frac{s_i^2}{m_i} \\ &+ \frac{(m_i)(1-p_1)}{p_1^2}(\bar{\tau_i} + \frac{(m_i)(1-p_1)}{p_1^2}) \end{aligned} \tag{2.18}$$

Intra-cluster variance is:

$$Var_{intra} = \frac{1}{p_1}\sum_{i \in S} V(\hat{\tau_i}) \tag{2.19}$$

The total variance is:

$$\begin{aligned} Var(\hat{\tau}) &= Var_{inter} + Var_{intra} \\ &= Var(\hat{\tau})_{srs} + \frac{1}{p_1}\sum_{i \in S} Var(\hat{\tau_i}) \end{aligned} \tag{2.20}$$

2.8 Appdendix: Optimal Allocation for Stratified Reservoirs

Under the power allocation technique with power q setting to 0, we compute the reservoir size $|r_i|$ for key i as [37]:

$$|r_i| = |r| \times \frac{\sigma_i/(\frac{\sum_{j=1}^{|R_i|} y_{ij}}{|R_i|})}{\sum_{k=1}^{n} \sigma_k/(\frac{\sum_{j=1}^{|R_k|} y_{kj}}{|R_k|})} \tag{2.21}$$

where $|R_i|$ is the population size of key i and y_{ij} is the j^{th} item value of key i and σ_i is the standard deviation of key i. Then at any point in time t during the sampling process, reservoir size $|r_i(t)|$ is determined by this formula [37]:

$$|r_i(t)| = |r| \times \frac{\sigma_i(t)/(\frac{\sum_{j=1}^{|S_i(t)|} y_{ij}}{|S_i(t)|})}{\sum_{k=1}^{n} \sigma_k(t)/(\frac{\sum_{j=1}^{|S_k(t)|} y_{kj}}{|S_k(t)|})} \tag{2.22}$$

where $|r_i(t)|$ denotes the size of a sub-sample allocated for key S_i at time point t, $\sigma_i(t)$ denotes the running standard deviation of key i up to t,and $|S_i(t)|$ denotes the number of tuples processed up to t from key i.

Chapter 3

Similarity Driven Approximation for Text Analytics

In this chapter, we propose a framework called EmApprox to speed up a wide range of queries over large text data sets. The key idea behind EmApprox is to build a general index that guides the processing of a query toward a subset of the data that is most *similar* to the query. For example, consider a query that seeks to count the number of occurrences of a given phrase. EmApprox would select a sample of the data set, preferentially choosing items most similar to the query phrase, count the occurrences in the sample, and use the count to estimate the number of occurrences in the entire data set. Clearly, the result is *approximate* so that users of EmApprox would need to tolerate some imprecision in the estimated results. EmApprox allows users to trade off precision and performance by adjusting the sampling rate.

Our approach is related to the many approximate query processing (AQP) systems that answer aggregation queries over relational data sets by processing estimators with error bounds using samples of the data [6, 11–13]. In essence, one can think of EmApprox as extending AQP to text analytics. EmApprox supports the estimation of errors bounds when possible; e.g., for aggregation queries. However, EmApprox can also be used in scenarios where it is not possible to estimate error bounds such as information retrieval, making it widely applicable to many different text analytic queries/applications.

As we know, index structures can make data access/search more efficient. For example in a database, B-tree based index can speed up range queries, Bloom filters are used for quickly checking record existence and hash maps makes single-key lookups $O(1)$ operations [60]. It is also pointed out in [60] that traditional index structures can be viewed as models, and using ML model to learn the distribution of keys can result

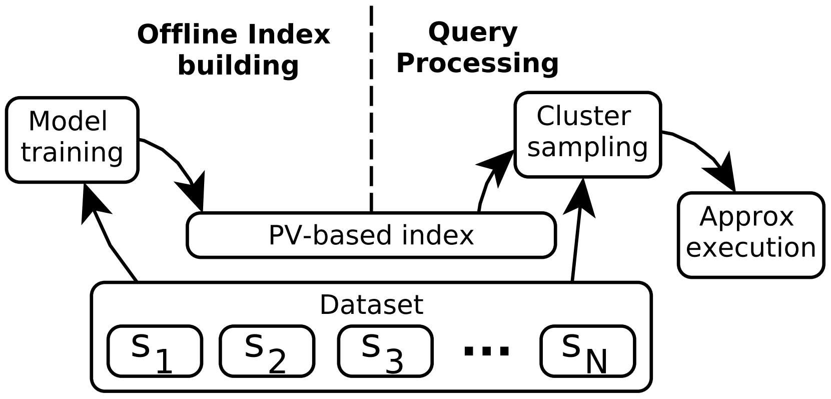


Figure 3.1: Overview. S_n are subcollections of documents.

in a general-purpose index which can complement existing index structures.

System overview. Figure 3.1 gives an overview of EmApprox. As mentioned above, EmApprox executes a query on a sample of the data set to reduce query processing time. Straightforward use of random sampling can lead to large errors, however, when sampling from a skewed distribution [11]. To mitigate this issue, EmApprox builds an index *offline*, then consults the index at query processing time to guide sampling toward subsets of data that are most similar to the query.

Specifically, EmApprox uses a natural language processing (NLP) model [61] to learn vector representations for unique words and documents. The resulting vectors can be composed and used to compute a similarity metric. Then, assuming that the data set is partitioned into a number of subcollections as shown in Figure 3.1, EmApprox computes a vector for each subcollection from the vectors of the documents contained in it.[1] The final index contains vectors for unique words together with vectors for the subcollections.

At query processing time, EmApprox computes a vector for the query using vectors of the words in the query. It then computes a sampling probability for each subcollection that is proportional to the similarity between the subcollection and the query using their vector representations. Finally, it selects a sample of subcollections using

[1]Data sets may be partitioned into subcollections for a variety of reasons, including storage in a distributed file system such as HDFS [62].

unequal probability cluster sampling. It is also known as *probability proportional to size* (pps) [10] [2] over the subcollections, where sampling probabilities are proportional to subcollections' similarity to the query as computed using the subcollections' and query's vectors.

EmApprox uses locality-sensitive hashing (LSH) to hash each real-valued vector to a bit vector [63] to reduce the storage overhead of the index. LSH works well because it preserves the distance between the original vectors. Computing similarity using LSH bit vectors is also extremely cheap; it is simply the Hamming distance of two bit vectors that can be computed efficiently using `XOR`. This optimization has greatly increased the scalability and efficiency of EmApprox's index.

Queries. We have implemented a prototype of EmApprox and used it to support approximate processing for three different types of queries: (1) aggregation queries that count occurrences within a text data set, (2) retrieval queries, both Boolean and ranked, that retrieve relevant documents, and (3) recommendation queries that predict users' ratings for products. For aggregation queries, we show how to compute estimated error bounds along with the approximate results. We also show that the training objective of PV-DBOW [61], the specific NLP model that we use, is directly related to minimizing the variance of the estimated results when using similarity driven sampling. For the retrieval queries, we use EmApprox in a similar fashion to *distributed information retrieval* (DIR), where a query is only processed against subcollections that are expected to be most relevant to the query [11]. Finally, for the recommendation queries, we use the user-centric collaborative filtering (CF) algorithm [64] to predict a target user's ratings using the average of other users' ratings weighted by similarities between their product reviews.

Evaluation. We generate a large number of queries for each query type, and execute them on three different data sets. We adopt equal probability cluster sampling [10] over subcollections as the baseline for our evaluation. We show that EmApprox can achieve

[2] Size is referring to the number of relevant data items in a cluster.

significant improvements on different domain-specific metrics (e.g., error bounds, precision@k, etc.) compared to the baseline with very little extra overhead during query processing. For example, to match the error bounds in aggregation queries achieved by EmApprox, the baseline would have to process ∼4x the amount of data. We also show that EmApprox can achieve significant speedups if users can tolerate modest amounts of imprecision. For example, when sampling at 10%, EmApprox speeds up a set of queries counting phrase occurrences by almost 10x while achieving estimated relative errors of less than 22% for 90% of the queries.

EmApprox is extremely efficient for processing queries that estimate results such as aggregation and recommendation queries. In contrast, like all sampling-based approaches, EmApprox is less effective for speeding up queries similar to information retrieval queries. This is because these queries are seeking specific data items in the data set, and it is impossible to estimate missed data items based on the sample.

Contributions. In summary, our contributions include: (i) to our knowledge, our work is the first to leverage an NLP model to build a general-purpose index to guide the approximate execution of text analytic queries; (ii) we show that the training objective of PV-DBOW is directly correlated with minimizing the variance of counting queries; (iii) we propose similarity driven sampling that can significantly increase accuracy compared to random sampling for three distinct types of approximate queries in three different application domains; (iv) we show that hashing real-valued vectors into light-weight LSH bit vectors significantly improves storage and computation efficiency without compromising precision.

3.1 Background and Related Work

3.1.1 Approximate query processing

Traditional AQP systems target aggregation queries over relational data sets. BlinkDB [12] is an AQP system that selects offline stratified samples based on historical information about queries for query processing. ApproxHadoop [6] and ApproxSpark [65] are online sampling-based frameworks that supports approximating aggregation with error bounds. Sapprox [11] has offline and online components. It collects the occurrences of column values offline, and uses the information to facilitate online cluster sampling. EmApprox is related to the above AQP systems in that it uses sampling and similar statistical theories to estimate error bounds for certain classes of queries. EmApprox is different in its target of many different types of queries on unstructured text data sets, and its use of an NLP model to estimate similarity.

ApproxSpark [65] generalizes ApproxHadoop by adapting online cluster sampling from the MapReduce computation model to multi-step RDD transformations. It also implements a distributed version of stratified sampling [66] to reduce sampling errors for rare keys, but its performance gain is limited because of a larger overhead from online stratified sampling.

3.1.2 Cluster sampling

Large data sets are typically partitioned such that a partition can naturally correspond to a cluster for cluster sampling. When this is true, cluster sampling is especially efficient since it avoids the need to access clusters that are not selected for a sample, and so is used in many approximate computing systems, e.g., [6, 11, 12, 65].

Suppose we need to estimate the frequency τ of a phrase Z occurring in a text data set partitioned into subcollections. If we take a cluster sample from the data set using subcollections as the sampling clusters, we can use the estimator $\hat{\tau} = \frac{1}{n} \sum_{s \in S} \frac{\tau_s}{\phi_s} \pm \epsilon$ [10], where S is the chosen sample, n is the number of subcollections in S, τ_s is the frequency of Z in subcollection s, ϕ_s is the sampling probability for s, and ϵ is the estimated error bound. $\epsilon \propto \sqrt{\hat{V}(\hat{\tau})}$, with

$$\hat{V}(\hat{\tau}) = \frac{\sum_{s \in S} (\frac{\tau_s}{\phi_s} - \hat{\tau})^2}{n(n-1)} \tag{3.1}$$

We observe that as the ϕ_s's approach $\frac{\tau_s}{\hat{\tau}}$, $\hat{V}(\hat{\tau})$ and hence ϵ will approach 0. The goal of *probability proportional to size* (pps) sampling [10] is to set each ϕ_s close to $\frac{\tau_s}{\hat{\tau}}$ by leveraging auxiliary information of each sampling unit, so that we can reduce the error bound for our estimator $\hat{\tau}$. Below, our approach is to set ϕ_s close to $\frac{\tau_s}{\tau}$, which should be close to $\frac{\tau_s}{\hat{\tau}}$ if $\hat{\tau}$ is a good estimator.

3.1.3 Paragraph Vectors

Recent advances in NLP have shown that semantically meaningful representations of words and documents can be efficiently learned by neural embedding models [61, 67]. Word2vec uses an unsupervised neural network to learn vector representations (embeddings) for words [67]. It seeks to produce vectors that are close in a vector space for words having similar contexts, which refers to the words surrounding a word in a pre-determined window. For example, synonyms like “smart” and “intelligent,” or closely related words such as “copper” and “iron,” are likely to be surrounded by similar words, so that Word2vec will produce spatially close vector representations for them. Similarity between two words can thus be scored based on the dot product distance between their corresponding vectors. The learned vectors also exhibit additive compositionality, enabling semantic reasoning through simple arithmetic such as element-wise addition over different vectors. For example, $vec(\text{“}king\text{”}) - vec(\text{“}man\text{”}) \approx vec(\text{“}queen\text{”}) - vec(\text{“}woman\text{”})$.

Paragraph Vector (PV) [61] is similar to Word2vec, which jointly learns vector representations for words and variable-length text ranging from sentences to entire documents in a method. Distributed Bag of Words PV (PV-DBOW) is a version of PV that has been shown to be effective in information retrieval due to its direct relationship to word distributions in text data sets [68]. By setting PV-DBOW’s window size to be each of the document’s length, the generative probability of word w in a document d is modeled by a softmax function:

$$P_{PV}(w|d) = \frac{exp(\vec{w} \cdot \vec{d})}{\sum_{w' \in V} exp(\vec{w'} \cdot \vec{d})} \tag{3.2}$$

where $\vec{w}$ and $\vec{d}$ are vector representations for w and d, and V is the vocabulary (i.e., the set of unique words in the data set). PV-DBOW learns the word and document vectors using standard maximum likelihood estimation (MLE), by maximizing the likelihood of observing the training text data set under the distribution defined by Eq (3.2). As a result, the training process will output word and document vectors that satisfy Eq (3.2) which formulates the theoretical foundation of our approximation index.

To reduce the expense of computing Eq (3.2) during training, a technique called negative sampling has been proposed [67] that randomly samples a subset of words in that document according to a noise distribution to approximate Eq (3.2). The training process is equivalent to implicitly factorizing a shifted matrix of point-wise information (*PMI*) between words and documents: [69]:

$$\vec{w} \cdot \vec{d} = PMI(w, d) - log(k) \tag{3.3}$$

$$PMI(x, y) = log\frac{p(x, y)}{p(x)p(y)} = log\frac{p(x|y)}{p(x)} \tag{3.4}$$

where $PMI(w, d)$ is the point-wise information between word w and document d, and k is a constant representing the number of negative samples for each positive instance in the training process. PMI can be estimated empirically by observing the frequencies of words in documents in the data set as $log\frac{\#(w,d)}{|d|} \cdot \frac{|D|}{\#(w,D)}$, where $\#(w, d)$ is the frequency of w in document d, $|d|$ is the length of (number of words in) d, D is the data set, $\#(w, D)$ is the total number of occurrences of w in D, $|D|$ is the total number of words in D.

Given PMI's definition, Eq (3.3) reveals that the exponential of the distance between a document and word vector is proportional to the probability of document *predicting* this word $p(w|d)$, which indicates that if a word is chosen randomly from d, then what is the probability that it would be w:

$$exp(\vec{w} \cdot \vec{d}) = \frac{p(w|d)}{p(w)k} \propto p(w|d) \tag{3.5}$$

We can see from Eq (3.5) that the inner-product distance between the word vector and each document vector is proportional to the percentage of the total occurrence of w contributed by each document d.

Negative sampling randomly samples words according to a predefined noise distribution and uses these words to approximate eq (3.2). The global training objective of PV-DBOW using negative sampling is:

$$l = \sum_{(w,d)\in D} \#(w, d)\ (log\ \sigma(\vec{w} \cdot \vec{d})\ + k\ \cdot E_{w_N \sim P_n}[log\ \sigma(-\vec{w_N} \cdot \vec{d})]) \tag{3.6}$$

where $\#(w, d)$ denotes the frequency of an observed word-document pair, D is the corpus, k is the number of negative samples, σ is the sigmoid function and $E_{w_N \sim P_v}[log\ \sigma(-\vec{w_N} \cdot \vec{d})]$ is the expected value of $log\ \sigma(-\vec{w_N} \cdot \vec{d})$ given a noise distribution P_n for w_N [61].

3.1.4 Locality-Sensitive Hashing

Hashing methods have been studied extensively for searching for similar data samples in a high-dimensional data set to solve the approximate nearest neighbor problem. LSH is among the most popular choices for indexing a data set using hashing [63]. The basic idea behind LSH is to transform each item in a high dimensional space into a bit vector with b bits, using b binary-valued hash functions h_0, ..., h_b. In order for the bit vector to preserve the original vectors' similarity, each hash function h must satisfy the property:

$$Pr[h(\vec{x}) = h(\vec{y})] \propto sim(\vec{x}, \vec{y})$$

where $\vec{x}$ and $\vec{y}$ are two vectors in the data set; sim is a similarity measure, such as Jaccard, Euclidean or cosine. $Pr[h(\vec{x}) = h(\vec{y})]$ is computed as one minus the ratio of the Hamming distance between two bit vectors over the total number of bits in them. Similarity between two items is preserved in the mapping, that is, if two items' LSH bit vectors are close in Hamming distance then the probability that they are close to each other in the original metric space is also high. This property allows items' LSH bit vectors to efficiently index a data set for similarity search [63].

3.2 Similarity-driven Sampling

In this section, we discuss cluster sampling with probabilities proportional to similarities of a query to subcollections for aggregation queries. Cluster sampling has been adopted for approximating aggregation queries (section 3.1.2) that seek to compute a sum/mean over the data set, such as counting the occurrences of a phrase or number of documents related to a given topic. We show that the similarities, as auxiliary information for sampling, can be computed online using the offline trained PV-DBOW vectors.

3.2.1 Query vector

We assume a query q contains l words $\{w_i\}$. Under the bag-of-words assumption, the probability of q in a document d is the joint probability of its words w_i:

$$p(q|d) = \prod_{i \in l} p(w_i|d) \tag{3.7}$$

We define $\vec{q}$ as element-wise arithmetic sum of its individual words' vectors: $\vec{q} = \sum_{i=1}^{l} \vec{w_i}$, then by combining Eq (3.5) and Eq (3.7) we can derive that $p(q|d)$ is proportional to $exp(\vec{q} \cdot \vec{d})$ from PV-DBOW's training objective:

$$p(q|d) = \prod_{i \in l} p(w_i|d) \propto exp(\vec{q} \cdot \vec{d}) \tag{3.8}$$

which is the exactly the same form as Eq (3.5) when a query only comprises a single word. Therefore by computing $\vec{q}$ this way, we can conveniently derive the probability of document predicting this query, under the assumption that the words are independent.

3.2.2 Sampling probability estimation

We define a document's *similarity* to a query as $p(q|d)$, the probability of d predicting q. $exp(\vec{q} \cdot \vec{q})$ in Eq (3.8) can be used to compute $p(q|d)$ at sampling time using $\vec{q}$ and $\vec{d}$, both contained in the offline trained PV-DBOW model. Suppose we use documents as cluster sampling units to estimate the quantity of q throughout the data set, then we can use $exp(\vec{q} \cdot \vec{d})$ as each document's auxiliary information for setting sampling probabilities proportional to its similarity to q:

$$\phi_d(q) = \frac{p(q|d)}{\sum_{d' \in D} p(q|d')} = \frac{exp(\vec{q} \cdot \vec{d})}{\sum_{d' \in D} exp(\vec{q} \cdot \vec{d'})} \tag{3.9}$$

As large data set is often partitioned into subcollections, it more efficient if we use subcollection as sampling clusters [6,11]. Similar to sampling documents with probabilities proportional to similarity, we propose to use a subcollection's vector representation to compute its similarity to the query and set sampling probabilities proportionally. Intuitively, we propose to define a subcollection' vector representation using the element-wise arithmetic mean of the vectors of the subcollection's documents: $\vec{s} = \frac{1}{n} \sum_{d \in s} \vec{d}$, where n is the number of documents in s, and d is a document in s.

We now demonstrate that choosing arithmetic mean of document vectors as subcollection's vector representation is reasonable. Given $\vec{s}$ and Eq (3.8), we can derive the exponential of the dot product between a subcollection and query's vectors as the geometric mean of each $p(q|d)$ in s:

$$exp(\vec{q} \cdot \vec{s}) = \sqrt[n]{\prod_{d \in s} exp(\vec{q} \cdot \vec{d})} \propto \sqrt[n]{\prod_{d \in s} p(q|d)} \tag{3.10}$$

Let $p(q|s)$ denote $\sqrt[n]{\prod_{d \in s} p(q|d)}$, then we can rewrite Eq (3.10) in a similar form as Eq (3.8):

$$p(q|s) \propto exp(\vec{q} \cdot \vec{s}) \tag{3.11}$$

where we use $p(q|s)$ to express the *similarity* of a subcollection to the query. Similar to computing the similarity of q to a document, we can use Eq (3.11)'s left hand side to compute the similarity of q to a subcollection. Following the same idea as using documents as sampling units, we define a probability distribution $\phi_s(q)$ for each subcollection s with respect to q in the same form as Eq (3.9):

$$\phi_s(q) = \frac{p(q|s)}{\sum_{s' \in D} p(q|s')} = \frac{exp(\vec{q} \cdot \vec{s})}{\sum_{s' \in D} exp(\vec{q} \cdot \vec{s'})} \tag{3.12}$$

where $exp(\vec{q} \cdot \vec{s})$ can be computed at sampling time.

A cluster sample over the subcollections with probabilities according to Eq (3.12) will reduce the variance (Eq (4.1)) in estimating the occurrence of $\{w_i\}$. It is because each subcollection is sampled with probability proportional to the probability each subcollection predicts the query. Variants of aggregations include estimating number of documents that contain $\{w_i\}$ or number of documents similar to $\{w_i\}$ in semantics. Both distributions defined in Eq (3.9) and (3.12) normalize the probability of a phrase appearing in a document or subcollection. Interestingly, Eq (3.9) and (3.12) have the same form as a softmax classifier over query words $\{w_i\}$ which predicts its probabilities conditioned on a document or subcollection.

3.3 Vector-based index structure

We slightly modify the gradient descent-based training process of PV-DBOW: at each update step, we normalize the original vectors to be unit length so that the dot product of the trained vectors is equivalent to the cosine similarities between two vectors rather than dot product. The resulting approximation index includes vectors for every word, document and subcollection. The index can occupy significant storage space for a large data set, for which we propose to map the real-valued vectors to LSH bit vectors to reduce the required storage. And, the cost of computing the similarities between bit vectors is also much more efficient than dot product between real-valued vectors.

Locality-sensitive hashing. LSH is among the most popular choices for indexing a data set using hashing [63]. The basic idea behind LSH is to transform each item in a high dimensional space into a bit vector with b bits, using b binary-valued hash functions h_0, ..., h_b. In order for the bit vector to preserve the original vectors' similarity, each hash function h must satisfy the property: $Pr[h(\vec{x}) = h(\vec{y})] \propto sim(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are two vectors in the data set; sim is a similarity measure, such as Jaccard, Euclidean or cosine. $Pr[h(\vec{x}) = h(\vec{y})]$ can be computed using the Hamming distance between $\vec{x}$ and $\vec{y}$s' corresponding bit vectors. Computing the Hamming distance of the LSH bit vectors using `XOR` is much more efficient than dot product of two real-valued vectors.

The hash function for preserving cosine distance depends on the dot product between a random plane $\vec{r}$ and an item vector $\vec{x}$, where $h_r(\vec{x})$ evaluates 1 if $\vec{r} \cdot \vec{x} \geq 0$, and 0 otherwise, where $\vec{r}$ usually has a standard multi-dimensional Gaussian distribution $\mathcal{N}(\mathbf{0}, I)$, and a new $\vec{r}$ is generated each time the hash function is applied [63]. In order to generate the LSH signature for a real value vector, we first choose a dimension l for the bit vector, then apply hash function $h_r(\vec{x})$ l times to generate each bit, each choosing a random $\vec{r}$. Specifically, we can approximate $exp(\vec{w} \cdot \vec{d})$ using $exp(cos(\frac{m}{l}\pi))$, where m is the Hamming distance between $\vec{w}$ and $\vec{d}$'s corresponding LSH bit vectors.

3.4 Retrieval-based Queries

In this section, we describe approximating DIR and recommendation queries using similarity-driven sampling extrapolated from aggregation. The goal of IR is to identify "similar" documents to a query, and that many recommendation techniques also require identifying "similar" users to a target user. We characterize our targeted queries as following: 1) the query can be represented by words; 2) similarity of query to a subset of data is proportional to how likely the subset contains relevant data.

Symbol	Description
B_i	i_{th} HDFS subcollection
d	d_{th} doc
n_i	number of docs in B_i
l_d	number of words in d
ϕ_i	average topic distribution in B_i
$\hat{q}_i$	estimated amount of queried data in B_i
π_i	B_i's sampling probability $= \frac{\hat{q}_i}{\sum_i \hat{q}_i}$
P_i	any probability w.r.t. B_i

3.4.1 Distributed information retrieval

In IR, a score is usually computed for each document indicating how relevant it is to the user's query (usually a sequence of words), to retrieve the relevant documents. The score can be computed based on metrics such as *tf-idf* of the query words or the generative probability of the query given a document using a language model, known as query likelihood [70]. IR can benefit from representations of words and documents in vectors that encode semantic information [68], where scoring of the documents can be computed based on the similarity between a query and documents. A retrieval model proposed in [70] uses latent Dirichlet allocation (LDA) for query likelihood. LDA is a generative language model that abstracts a document as mixture of topics, represented by a stochastic vector $\theta_d \in \Delta^K$, where K is the number of topics. Inspired by [70], a Paragraph Vector-based retrieval model was proposed in [68] that has $P_{PV}(w|d)$ defined in eq (3.2) in its retrieval model.

$$P(w|d) = (1-\lambda)P_{QL}(w|d) + \lambda P_{PV}(w|d) \tag{3.13}$$

where $P(w|d)$ is the probability of w given d, $P_{QL}(w|d)$ is defined as $P_{PV}(w|d)$ is defined in eq (3.2), λ is a free parameter between 0 and 1.

Information retrieval from disjoint subcollections of documents is known as distributed information retrieval (DIR), where many irrelevant subcollections are ignored for improved retrieval efficiency [71]. EmApprox can facilitate subcollection selection under the vector space retrieval paradigm for DIR [71]. We target Boolean and ranked retrieval models for DIR in the following discussion. However, our goal is not to compete with existing DIR solutions, but to demonstrate that our proposed index can complement previous AQP systems and extend approximation tasks to DIR with efficiently.

Boolean retrieval. A Boolean query is a Boolean expression of words (e.g., $w_0 \vee (w_1 \wedge w_2)$), where a term w_i only evaluates to true when contained in a document [71]. The retrieval result is a set of documents that satisfy the Boolean expression.

To answer a Boolean query q_b, we first compute its similarity to each subcollection - $p(q_b|s)$, which can be computed using the same order as evaluating the Boolean query, i.e. $\wedge$ takes precedence over $\vee$. We compute $p(w_i|s)$ of each term w_i for a subcollection s using Eq (3.11), in order to compute the query's overall similarity. Since $w_i \wedge w_j$ implies that w_i and w_j both have to exist for the entire expression to be true, whereas satisfying $w_i \vee w_j$ requires either w_i or w_j to exist, the generative probability of $w_i \wedge w_j$ is equivalent of $p(w_i|s) \cdot p(w_j|s)$; similarly, the generative probability of $w_i \vee w_j$ is therefore $p(w_i|s) + p(w_j|s)$. For example, suppose we have a Boolean query $q_b = w_0 \vee (w_1 \wedge w_2)$, then $p(q_b|s)$ can be computed as $p(w_0|s) + (p(w_1|s) \cdot p(w_2|s))$, where each $p(w_i|s)$ is computed using Eq (3.11). Finally, we use the overall query *similarity* to the subcollections $p(q_b|s)$ to compute sampling probability of each subcollection to sample a subset of subcollections, where only documents in the chosen subcollections are evaluated against the Boolean query.

Ranked retrieval. Instead of returning a set of documents that precisely match a Boolean expression, ranked retrieval returns a list of documents ranked by their relevancy to the query. Each document is assigned a score using a function, such as the *tf-idf* of the query terms. Similar to Boolean retrieval, our proposed framework first samples a subset of subcollections with probabilities proportional to the query's *similarity* to each subcollection - we just use Eq (3.12) to compute its *similarity* to a subcollection. We then apply a user-specified scoring function to documents from the chosen subcollections, such as BM25 [72].

3.4.2 Recommendation

User-centric collaborative filtering (CF) is one of the most successful recommendation techniques [64]. It identifies a subset of similar users as *neighbors* to a target user u, then uses the similarity score to predict a rating for item i by user u. It computes the prediction by taking an average of u's neighbors' rating weighted by their similarity scores.

Review text can embed rich information and has been leveraged to model users' behavior. It's predicting performance has been shown to be more effective than numerical data [19]. A user's vector representation can be learned under PV-DBOW model by defining a document as all the reviews a user has written [19]. Consequently, a user's vector $\vec{u}$ encodes u's preference. When the number of total users in the data set is large, the process of identifying neighbors from them to a target user u would be expensive. Our vector-based index structure makes it more amenable for large data sets by selecting only more similar subcollections of users. Suppose the review data set is sorted by users, then similarity of a user u to a collection of reviews can be computed using Eq (3.11) to sample most similar collections of users. Then predicted ratings for the new user can be computed using any model/metric with the neighbors in the chosen subcollections.

As a concrete example, rating of an item i by user u's predicted value can directly leverage u's review text vector's similarity to a user v who has also rated item i, computed as $\sum_{v \in U'} sim(u,v) r(v,i)$, where U' is the set of all users who have rated item i in the chosen subcollections, $r(v,i)$ is user v's rating for item i, $sim(u,v)$ can be characterized by their similarity in the review texts u and v have written, defined as:

$$\sum_{v \in U'} \frac{exp(\vec{u} \cdot \vec{v})}{\sum_{v' \in U'} exp(\vec{u} \cdot \vec{v'})} \tag{3.14}$$

Suppose the data set is the Amazon reviews, the purchase history of a user is the set of distinct products he has written reviews for, the "features" of a product is all the reviews it has received. Then respectively, if we treat both each user and each product as documents, embedding vectors of both the user and products can be learned using doc2vec. Suppose the trained user and product vectors are stored in plain text files, they could also be stored in HDFS and loaded into memory as RDD as well since they could be multiple GBs in size. The original review data set is also stored in HDFS. As before, we compute the vectors of each HDFS subcollection, representing groups of products and sample a percentage of the subcollections based on the similarities of the subcollection vectors and the user we'd like to recommend products to. Note that the RDD that stores the model should be partitioned in the same way as the data set RDD. After the subcollections are sampled, we first compute the list of top N products from the "sampled" model and retrieve relevant metadata (reviews, ratings, etc that accompanies selected products) from the selected HDFS subcollections storing the original data set.

Usually each user is represented by all the reviews she has written, where a user's vector representation can be learned under PV-DBOW model by defining a document as all the reviews a user has written. A user's vector $\vec{u}$ would encode the information of all review text she has written which potentially represents u's preference. We believe that $\vec{u}$ can also be obtained by averaging the word vectors in her reviews if this user is not already present in the data set.

3.4.3 Discussion

We hypothesize that using documents as sampling units is better for information retrieval-oriented tasks where subcollection-based cluster sampling may be too coarse. It is because IR is very sensitive to the allocation of documents in the data sets. If semantically documents are not clustered together, dropping subcollections may result in missing highly similar documents to the query.

However, using subcollection as sampling units is preferred since computing closed-form estimator to the query is less prone to losing particular documents. Sampling subcollections is more efficient because it eliminates the need of I/O for the dropped subcollections.

3.4.4 Discussion on Document Allocation

The document allocation policy can affect DIR's performance, where the storage of documents needs to be skewed for DIR to be effective. This is because the query expects to retrieve as many relevant documents to the query as possible from only a subset of the data set. The document allocation policy can also affect recommendation's performance, since identifying similar users is similar to retrieving relevant documents. On the other hand, the accuracy of aggregation estimators is not dependent on document allocation, since the local sum of each chosen subcollection is multiplied by the inverse of its own sampling probability as a scaling factor to compute an overall estimator, i.e., subcollections with a large local sum will just have a small scaling factor.

We propose to allocate documents based on their vectors' pair-wise cosine distance through clustering the documents in the original data set using *spherical K-means*, that uses the cosine as distance metric. The clustering process takes as input the collection of document vectors, and produces an allocation where semantically similar documents are clustered.

When documents d_1, d_2, ..., d_n are semantically similar, it indicates that the probabilities $p(w|d_1)$, $p(w|d_2)$, ..., $p(w|d_n)$ are also similar for a query word w. The result of clustering is a more skewed distribution of the documents, therefore documents that have the same probability of predicting a query word tend to be allocated together. As Eq (3.10) shows that

$$p(w|s) = \sqrt[n]{\prod_{d' \in s} p(w|d')} \tag{3.15}$$

which is the geometric mean of each document's probability of predicting w in that subcollection s – so if $p(w|d')$'s are similar in each s, then their geometric mean $p(w|s)$ will approach a local maximum equivalent to the arithmetic mean of all the $p(w|d')$, according to the AM-GM inequality [73]. It therefore suggests that this allocation policy would produce a skewed sampling probability distribution $\phi_s(w)$, which is desired for a retrieval query.

3.5 Limitations

EmApprox can approximate a range of text analytical queries, we nevertheless highlight a couple of limitations.

Model drift. We assume the text data set is historical and stable. If new documents are added to the existing data set, PV-DBOW is able to infer the vectors for an unseen document using the words in the new document [61]. However, the originally trained PV-DBOW model may drift due to document updates. Therefore PV-DBOW model should be retrained to capture the true word/frequent distributions in the data set, which requires the offline index to be rebuilt.

Unseen words in the query. Currently we assume the query does not include words outside the vocabulary of the data set, so that any word vector in the query can be directly obtained.

3.6 Implementation

We have implemented a prototype of EmApprox as a Python library comprising two parts: one for building offline indexers and one for building approximate query processing applications (queries for short). Users write indexers and queries as Spark programs in Python using the PySpark [74] and EmApprox libraries. Figure 3.2 gives an overview of a system built using EmApprox, where documents are stored in blocks of an HDFS filesystem, with blocks considered subcollections of documents and used as sampling units. Note that a single indexer can be used to index many different data sets of the same type, and many different queries can be executed against each index/data set.

We leave the task of writing indexers to the user because it allows the flexibility for indexing many different types of data (e.g., different data layouts and definitions of documents). It is quite simple to write indexers given the EmApprox library. Specifically, users need to write code to parse a given data set to extract the documents (much of this code can come from standardized libraries) and identify documents in each subcollection (HDFS block). All other functionalities are implemented in the EmApprox

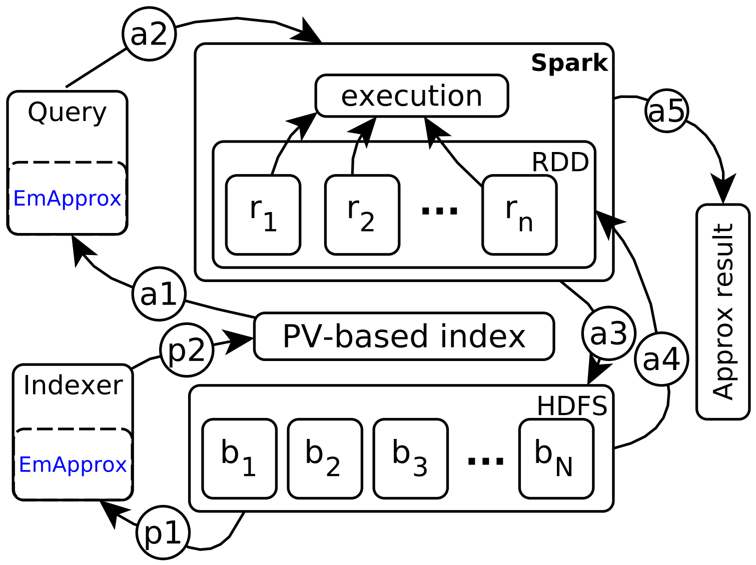


Figure 3.2: System architecture. $\{b_N\}$ are HDFS blocks, $\{r_n\}$ are Spark RDD partitions based on selected blocks.

library, and simply requires the user program to call several functions. Similarly, the main difference between an approximate query built using EmApprox and a precise query is the invocation of several EmApprox functions.

An offline indexer uses EmApprox to learn vector representations for words and documents, cluster documents (when desired) using K-means as discussed in Section 3.4.4, compute vectors for blocks (subcollections), compute corresponding LSH bit vectors, and prepare the index. This process is shown as steps $p1$ and $p2$ in Figure 3.2. (We do not show the clustering for simplicity.) We use Gensim [75] as the default library for PV-DBOW model training, but we can also use alternative implementations that can run on distributed frameworks such as Tensorflow [76] to reduce the training time. We use Gensim in our prototype because it is a widely adopted PV-DBOW implementation.

The execution of an approximate query is shown as steps $a1$ through $a5$ in Figure 3.2. In step $a1$, the query uses EmApprox to read the index into an in-memory hash table, compute sampling probabilities for all HDFS blocks, and choose a sample using *pps* sampling. Step $a2$ launches a Spark job. Steps $a3-4$ are part of the Spark job and use

Query	Domain	Description	Metrics
phrase occurrence	aggregation	estimates frequency for target phrase.	error bound
Boolean retrieval	DIR	retrieves a (sub)set of documents that precisely match a Boolean query.	recall
ranked retrieval	DIR	retrieves top-k documents ranked by a given scoring function over a set of query terms.	precision
user-centric CF	recommendation	predicts ratings on unbought products and outputs top-k recommendations for a user.	MSE, precision

Table 3.1: Approximation queries and metrics summary

EmApprox and PySpark to read the sample from the data set into an RDD. Step $a5$ is the execution of the rest of the Spark job. We provide two simple reduce functions that compute the estimated sum and average, along with the confidence interval.

3.7 Evaluation

3.7.1 Setup

Data. We use three data sets: a snapshot of Wikipedia [58], a news corpus from Common Crawl (CCNews) [77], and a set of Amazon user reviews [19]. Table 3.2 summarizes the data sets, their use in different queries, the training time of PV-DBOW using Gensim, and the size of the resulting indices after compression using LSH.

Experimental platform. Experiments are run on a cluster of 6 servers interconnected with 1Gbps Ethernet. Each server is equipped with a 2.5GHz Intel Xeon CPU with 12 cores, 256GB of RAM, and a 100GB SSD. Data sets are stored in an HDFS file system

Dataset	Description	Size	Document	T-Time	Idx-size
Wikipedia	∼5 million articles in XML.	62GB	each Wikipedia article	4.3h	125MB
CCNews	∼22 million news articles crawled in 2016 in JSON.	65GB	each news article	6.2h	280MB
Amazon reviews	∼142 million reviews in JSON.	55GB	reviews from a user	3.8h	87.5MB

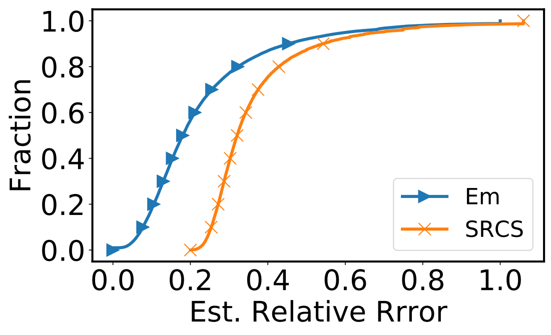
Table 3.2: Dataset descriptions, notion of a document (Document) for PV-DBOW, training time (T-Time) and index sizes.

hosted on the servers' SSDs, configured with a replication factor of 2 and block size of 32MB. Applications are written in Python 3 and run on Spark 2.0.2.

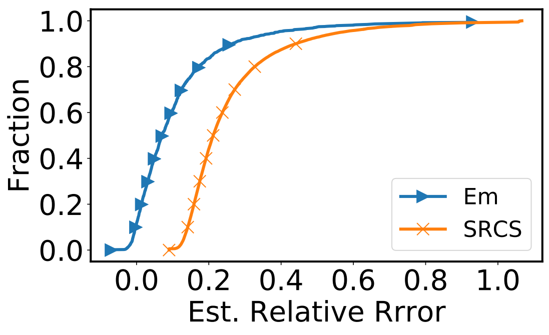
Index construction. We train PV-DBOW to produce word and document vectors with 100 dimensions. We use LSH vectors of 100 bits, which compresses the PV-DBOW learned vectors by a factor of 64. We explore the sensitivity of EmApprox to these parameters in Section 3.7.3.

Baselines. We analyze and compare EmApprox's performance against simple random clustered sampling (SRCS) [10] of HDFS blocks and precise execution. We compare execution times (speedups) and query-specific metrics. We run the precise executions as "pure" Spark programs on an unmodified Spark system. We run SRCS using the EmApprox prototype, replacing pps sampling with simple random sampling.

Aggregation queries. We run aggregation queries that estimate the numbers of occurrences and the corresponding relative errors of target phrases in the Wikipedia data set. We create 200 queries by randomly selecting phrases from the data set. The lengths of the phrases follow a normal distribution with a mean of 2 words and a standard deviation of 1. The approximate answer to each query executed with a given sampling rate



(a) 5% Sampling Rate



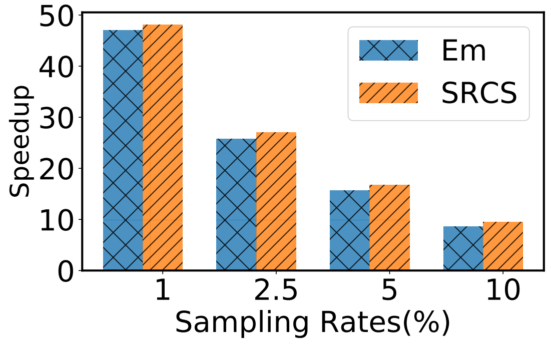
(b) 10% Sampling Rate

Figure 3.3: CDFs of estimated relative error for phrase occurrences under different block sampling rates.

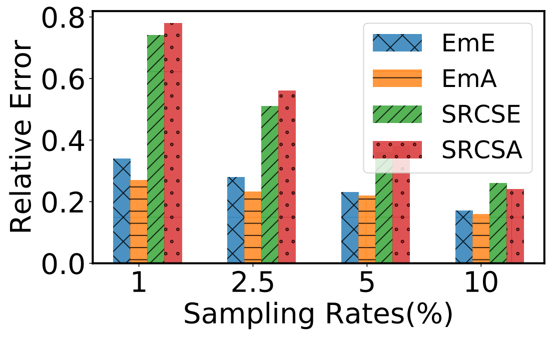
includes the estimated count ($\hat{\tau}$) with an error bound ($\hat{\tau} \pm \epsilon$). We report the *estimated relative error* at 95% confidence level as the ratio of ϵ over $\hat{\tau}$. We also compare the estimated relative error with the actual relative error, computed as $\frac{|\hat{\tau}-\tau|}{\tau}$, where τ is the precise answer.

DIR queries. We cluster documents within the Wikipedia and CCNews data sets as explained in Section 3.4.4, setting the number of centroids equal to the number of HDFS blocks in the file holding the data set. We generate 200 sets of randomly chosen words, 100 from the Wikipedia data set and 100 from CCNews. Set sizes follow a normal distribution with an average size of 3 and a standard deviation of 1. We randomly insert Boolean operators (`and` and `or`) to form Boolean queries, and use the sets of words directly as queries in ranked retrieval. We use the BM25 ranking function [72] in ranked retrieval. We choose BM25 from a plethora of ranking functions, including functions that use Paragraph Vector [68], because it is widely adopted by search platforms such as Solr [78] and IR libraries such as Apache Lucene [79]. In Boolean retrieval, we report *recall*, defined as ratio of the number of documents retrieved by the approximate query processing to the ground truth. In ranked retrieval, we report *precision-at-k* (P@k), defined as the percentage of the top k documents retrieved by the approximate query processing that is in the top k retrieved by the precise query execution.

Recommendation queries. We approximate user-centric CF, which takes as input

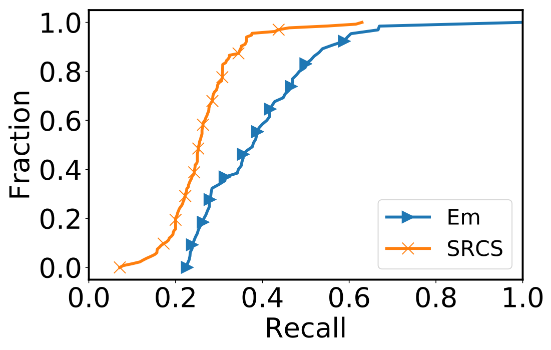


(a) Avg. Speedup

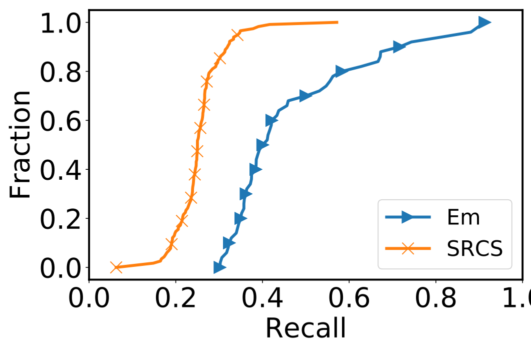


(b) Actual and estimated rel. errors

Figure 3.4: Average speedups and relative errors for phrase occurrence query. (b) shows the comparison of estimated (E) and actual average relative error (A) using EM and SRCS (e.g. EmE means estimated relative error using EmApprox, SRCSA means actual relative error using SRCS).

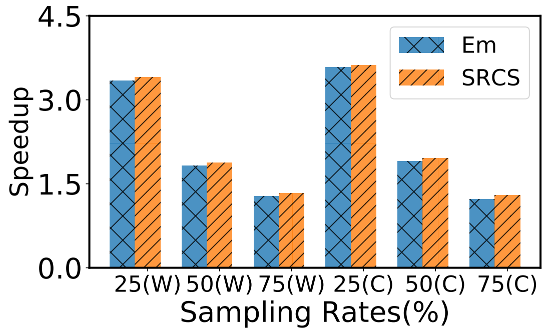


(a) Wikipedia, 25% sampling

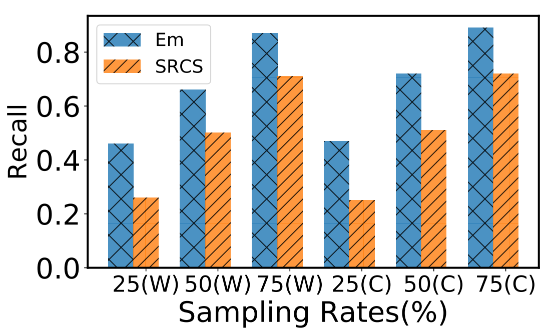


(b) CCNews, 5% sampling

Figure 3.5: CDFs of recall for Boolean retrieval queries over Wikipedia and CCNews data sets at 25% sampling rate.



(a) Avg. Speedup - Boolean



(b) Avg. Recall - Boolean

Figure 3.6: (a) and (b) show speedup and recalls averaged across the test queries under 25%, 50% and 75% sampling rates for Boolean retrieval ((W) and (C) represent Wikipedia and CCNews data sets respectively).

a target user with past reviews/purchase history then outputs predicted ratings for unpurchased items. It also generates a top-k recommended item list sorted by their predicted ratings. We rearrange the reviews in the original data set to group all reviews written by a unique user together. Each group of reviews written by a unique user is then considered a single document. We randomly select 100 users and remove 20% of each selected user's ratings from the data set to be used as test data. We then cluster the remaining documents as we did for the DIR queries and construct the index. Finally, we construct 100 queries for the selected users, where each query outputs the predicted ratings as computed by the CF algorithm for the users/reviews in the test data. The rating scale is 1-5 in the Amazon data set. We report mean squared error (MSE) and P@k to measure prediction performance. MSE is computed for the predicted vs. actual ratings as a measure of accuracy for the predictions. P@k is the percentage of the items predicted top-k list that were purchased by the target user.

3.7.2 Results

Aggregation. Figure 3.3 plots the CDFs of estimated relative errors when running the 200 queries under EmApprox and SRCS at 5% and 10% sampling rates. We observe that: (1) EmApprox consistently achieves smaller estimated relative errors than SRCS at the same sampling rate; (2) the "tails" of the CDFs are "shorter," meaning that there are fewer query answers with large estimated relative errors; and, (3) EmApprox achieves smaller maximum estimated relative errors. Under SRCS, the estimated relative errors can be large at very low sampling rates. For example, the estimated relative errors are 80% and 95% for the 50^{th} (median) and 90^{th} percentile, respectively, at 1% sampling rate. Under EmApprox, they are reduced to 25% and 45%. Errors become much smaller with increasing sampling rates.

Figures 3.4(a) and (b) show average speedups compared to precise execution and average relative errors (both estimated and actual), respectively. We observe that speedups are slightly smaller for EmApprox compared to SRCS. This is because EmApprox does incur a small amount of extra overhead to compute the sampling probabilities for blocks. On the other hand, EmApprox achieves much smaller relative errors than SRCS. Specifically, SRCS has to process roughly 4x the amount of data processed by EmApprox to achieve similar relative errors.

In summary, EmApprox significantly outperforms SRCS. If the user can tolerate the estimated relative error profile for EmApprox at 10% sampling rate (i.e., 30% and 40% at 50^{th} and 90^{th} percentiles, respectively), then EmApprox achieves an average speedup of ~10x for the 200 aggregation queries. Further, the user can trade off between accuracy and performance by adjusting the sampling rate.

DIR. Figure 3.5 plots the CDFs of recall for the Boolean queries under EmApprox and SRCS out of 100 queries over Wikipedia and CCNews data sets. Similar to aggregation, we observe that EmApprox significantly outperforms SRCS. Figures 3.6(a) and (b) show Boolean retrieval's speedups over precise execution and the average achieved recall rates, respectively. We observe that the much higher sampling rates required to achieve higher recall rates constrain achievable speedups. We have our ranked retrieval results in our technical report [80].

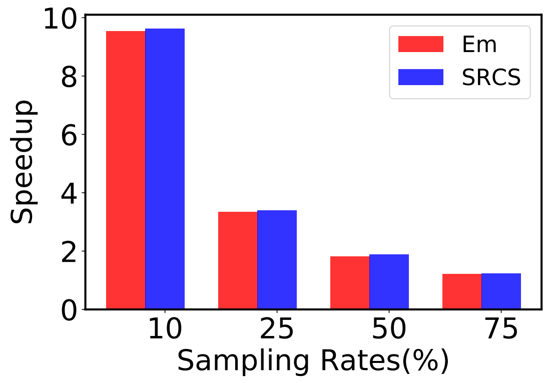
In summary, EmApprox significantly outperforms SRCS. However, the nature of the problem, which is to find specific items in a large data set, reduces the effectiveness of sampling, even when sampling is directed by some knowledge of content. Thus,

	Speedup (x)		**MSE**			**P@10 (%)**		
Sampling Rates(%)	Em	SRCS	Em	SRCS	Impr (%)	Em	SRCS	Impr (%)
10	48.04	48.12	1.29	1.42	9.15	0.25	0.23	8.69
25	26.76	27.01	1.16	1.26	7.93	0.28	0.26	7.69
50	16.70	16.75	1.08	1.12	3.50	0.29	0.28	3.57
75	9.63	9.52	1.04	1.06	2.80	0.31	0.30	3.33
100	9.63	9.52	1.015	1.015	0	0.32	0.32	0

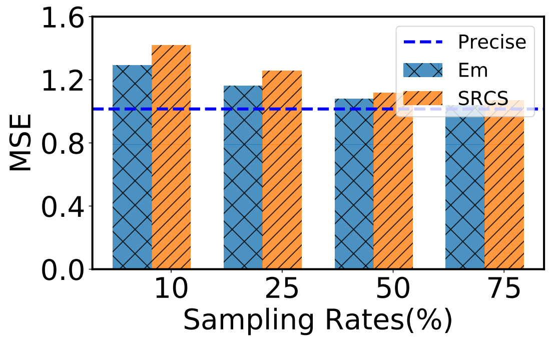
Table 3.3: Average speedups over precise execution, and recall for *Top-10 recommendation* under EmApprox and SRCS. Impr column contains the improvement in recall under EmApprox compared with SRCS.

EmApprox can only achieve modest speedups while achieving relatively high recall rates and precision levels. For example, EmApprox achieves an average speedup of 1.3x at a sampling rate of 75%. Achieved average recall and P@10 are 0.89 and 0.78, respectively.

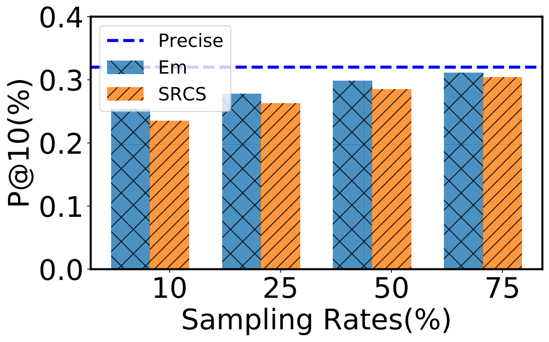
Recommendation. Figure 3.7 shows average MSE and P@10 under different sampling rates for EmApprox and SRCS. Similar to results for the other two query types, EmApprox outperforms SRCS. The differences between the two approaches are less pronounced, however. For example, EmApprox outperforms SRCS by 8% and 7.4% for average MSE and average P@10, respectively, at a sampling rate of 25%. This is likely due to the fact that user-centric CF itself does not achieve high accuracy—the precise execution achieves an average MSE of 1.015 and average P@10 of 0.32%—so that selecting customers most similar to the target customer does not have a large impact compared to random selection. EmApprox incurs minimal overheads at query processing time, however, and so its increased accuracy is still desirable, especially when the number of customers is large. EmApprox speeds up the query processing time by almost 9x while degrading P@10 by 18.7% at 10% sampling rate. Speedup is over 3x with 12.5% degradation of P@10 at 25% sampling.



(a) Speedup

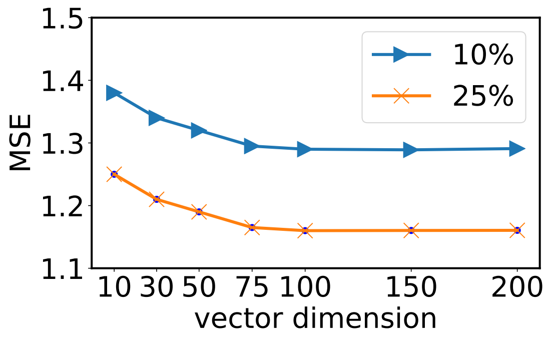


(b) Avg. MSE

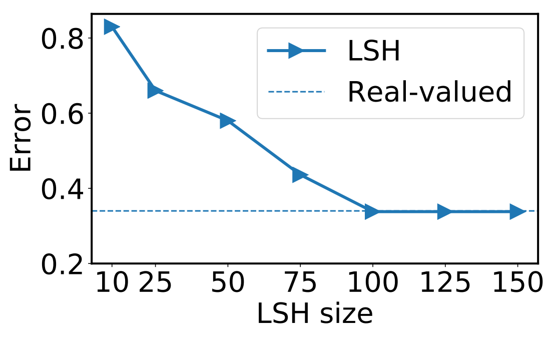


(c) Avg. P@10 (%)

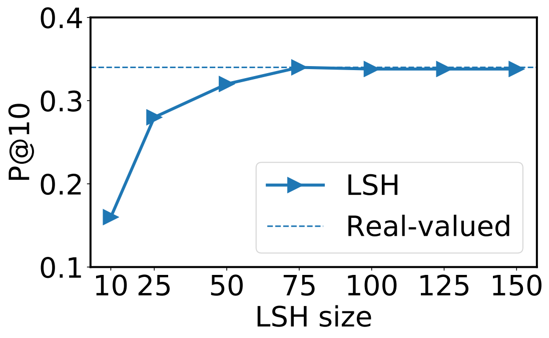
Figure 3.7: Recommendation results showing prediction MSE and p@10 across the customers under various sampling rates.



(a) Recommendation - MSE

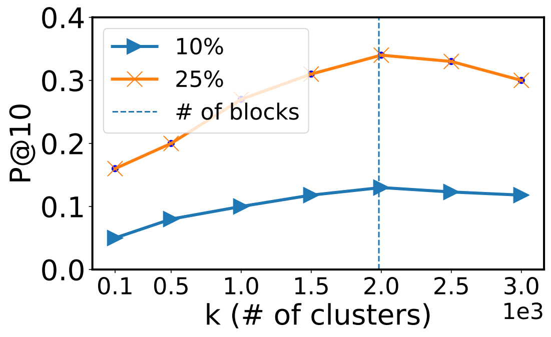


(b) Aggegation - Error

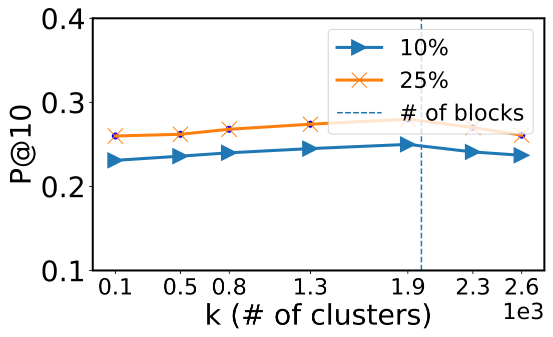


(c) Ranked retrieval - P@10

Figure 3.8: (a) shows the impact of vector dimension over MSE in the recommendation results under 10% and 25% sampling rates. (b) shows the impact of LSH bits to the aggregation results using the Wikipedia dataset (d) shows the impact of LSH bits to P@10 using the Wikipedia dataset



(a) Ranked retrieval



(b) Recommendation

Figure 3.9: Impact of k in the K-means clustering over P@10 for ranked retrieval (Wikipedia) and CF, under 5% and 25% sampling rates.

3.7.3 Sensitivity analysis

We briefly explore the impact of PV-DBOW and LSH bit vectors' dimensions (λ_1 and λ_2), as well as number of clusters (k) in K-means in offline preprocessing. Our technical report contains a more extensive discussion [80].

Vector dimensions. Experiments show that EmApprox's performance improves gradually with increasing λ_1 and λ_2, and then stabilizes when they are sufficiently large. For example, MSE for recommendation improves roughly linearly until λ_1 reaches 75, and then completely flattens out when λ_1 grows beyond 100. These observations led us to choose $\lambda_1 = 100$ and $\lambda_2 = 100$ in our evaluation.

Number of Clusters for K-means. EmApprox's performance for DIR and recommendation queries is sensitive to k (the number of clusters) in K-means. Experiments show that the performance gradually improves as k increases and plateaus as it approaches the number of HDFS blocks in a data set. Beyond that, performance may actually decrease because data within each block (sampling unit) is no longer clustered.

3.7.4 Summary

EmApprox can speed up aggregation and recommendation queries by up to 10x if the user can tolerate modest imprecision. In addition, EmApprox can gracefully trade off imprecision with performance by adjusting sampling rates. For example, under 10% sampling, EmApprox achieves an estimated relative error of 18.2% for aggregation, and a degradation on average of 18.7% in P@10 for recommendation queries.

3.8 Conclusions

We present an approximation framework for a wide range of analytical queries over large text data sets, using a light-weight index based on an NLP model (PV-DBOW). We formally show that the training objective of PV-DBOW maximizes the generative probability of a query given a collection of documents. Our experiment shows that our light-weight index can reduce the execution time by almost an order of magnitude while degrading gracefully in approximation quality with decreasing sampling rates. EmApprox is particularly useful for exploratory text analytics.

Chapter 4

Video Querying with Approximate Indexing

In this chapter, we propose an approximate video analytics framework called VidApprox for accelerating video queries that involve object detection. A video processing workflow would consist of video capturing, followed by storage, retrieval and finally consumption [21]. VidApprox aims to facilitate the efficient retrieval of relevant video data to reduce the processing needs of downstream operators (e.g. a deep CNN for batch object detection jobs). To reduce the storage cost, VidApprox encodes raw frames into segments, that is the smallest unit for decoding [81]. Each segment has a key frame representative of the entire segment [82]. The key frame can represent a visual summary and meaningful information about a part of the video. It can be useful in many applications such as searching, information retrieval, and indexing [82]. VidApprox mainly employs three techniques to facilitate the retrieval process, where the first two are performed offline and the third is online:

1. **Vector-based segment indexing.** A key frame's vector representation can be obtained through performing inference over object recognition CNNs with the frame as the input. Since the key frame is representative of an entire segment, we just use its vector representation to represent the segment. An image and a segment's similarity can be calculated by taking the cosine distance of their corresponding vectors. A real-valued vector can be further compressed using its mapping to locality sensitive hashing, which approximately preserves cosine/euclidean distance between a pair of original vectors [83].

2. **Similarity-aware segment placement.** We cluster similar segments based on their cosine distance between their vector representations. The intuition is that if a group of segments are not similar to a reference image, then they can

be pruned altogether. The clustering can make the segments distribution more skewed that can make retrieving relevant segments more efficient. As the segments are clustered, we use the centroid's vector to represent the cluster. We assume the video data set is stored in partitioned distributed storage. Within the same partition, we place the clusters such that the minimum distance among their centroids are maximized. The probability that all similar clusters end up on the same partitions is significantly lowered to distribute workload across the server nodes.

3. **Relevant segments retrieval.** User will need to provide a reference image that contains the target object category for the query. We then employ probability proportional to similarity sampling, where a cluster's similarity to will determine the sampling probability of each segment. As a result, only the most similar subset of segment clusters can be retrieved for further processing. A most similar subset is more efficient for both aggregation and retrieval oriented queries. Although we focus on single video data throughout our work, we can easily extend our technique to the case with a video library - dataset that contain multiple video streams.

Several recent works have sought to reduce video query latency. Noscope [23] aims to reduce the latency of batch binary classification tasks. It places a cascade of specialized shallower CNNs before the ground truth CNN for inference, and returns the result once predictions from one of the cheaper CNNs have sufficient confidence level. Focus [84] can reduce more query latency by building approximate object to frame index at ingest time by paying a preprocessing cost upfront. Both of the aforementioned works aim at query that identifies frames with objects of class X, and assume the frames are already present in memory. In practice, it is much more efficient to process encoded video segments rather than raw frames. For example, many video types such as surveillance video feeds do not contain scene changes, therefore encoded segments will significantly save storage space.

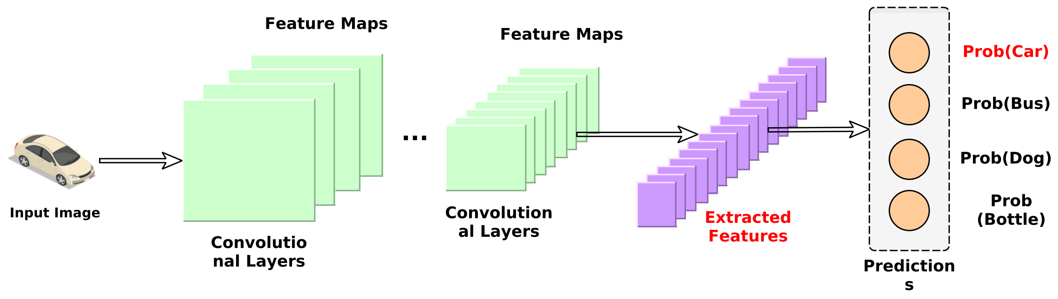


Figure 4.1: Example image classification CNN

We use our proposed segment indexing and clustering scheme to process two categories of video querying tasks: (1) aggregation queries that compute statistic such as average number of certain object per frame/segment, (2) retrieval queries that retrieve segments matching a predicate (e.g., containing a certain object class). For aggregation queries that estimate quantity of an object type, we leverage CNN inference for object detection in the retrieved segments and then cluster sampling theory to compute an estimator with probabilistic error bounds. We then show that indexing segments can result in a significant speedup with a small loss in accuracy. For retrieval queries, we show that our framework can facilitate retrieving a large number relevant segments by processing a fraction of the original video data. The retrieval scenario can make the downstream video consumer significantly more efficient by avoiding decoding and processing irrelevant segments.

In summary, our contributions include: (i) To our knowledge, our work is the first that leverages CNN-learned vectors to index similar video segments for approximate video processing. (ii) We show that our segment indexing and clustering scheme results in reduced error bounds for video aggregation tasks, and retrieval of a large number of relevant segments by processing only a fraction of the data.

4.1 Related work and Background

4.1.1 Approximate computing

Traditional AQP systems have mainly targeted aggregation queries over relational data sets. *AQP++* [13] is a recent database system that uses sampling-based AQP and precomputed aggregates to achieve interactive response time for aggregation queries. *BlinkDB* [12] is an AQP system that selects offline-generated samples to answer queries. It uses query column sets (QCSs) for representing the sets of columns appearing in past workloads, and stratified samples are created for each QCS. It assumes QCSs are stable over time, which does not perform well for queries outside the QCS coverage in the offline samples. *ApproxHadoop* [6] and *ApproxSpark* [65] are online cluster sampling based frameworks that supports approximating aggregation with error bounds. However, their result estimation is prone to large error bounds over skewed data. *Sapprox* [11] has an offline and an online component. It collects the occurrences of sub-data sets offline, and the information to facilitate online cluster sampling.

4.1.2 Visual data management

Visual data management has aimed to organize and query visual data, early systems include Chabot [85] and QBIC [86]. These systems were followed by a range of visual database for storing, querying and managing video data [87]. Many of these systems and languages use classic computer vision techniques such as low-level image features and rely on manual annotations (e.g., text) over the images for semantic queries. However, recent advances in deep learning based computer vision models allows learning the semantic information directly from the datasets.

4.1.3 Convolutional neural networks

Convolution Neural Networks (CNNs) have become effective for many computer vision tasks such as object detection and classification [88–90]. CNNs consist of different types of layers including convolutional layers, pooling layers and fully-connected layers, where convolutional layer combine nearby pixels via convolution operators; pooling layers reduce the dimensionality of the subsequent layers; rectified linear unit (ReLU) layers perform a non-linear transformation), and a fully connected layer outputs the actual prediction. For example in classification task, output from the final layer of the CNN is the probabilities of all object classes, and the class with the highest probability will be the predicted class for the object or input. image [24].

CNN can be considered as a feature extractor. Output of the second to last layer of a CNN is the representative features of the input image [88], where the layers before are considered transformations applied to the input image. The feature is represented as a real-valued vector, which has been shown that images with similar feature vectors (i.e., cosine distance) are visually similar [91, 92]. Thus, the distance between feature vectors is a common metric to measure similarity of images in many applications, such as face recognition [93] and image retrieval [94, 95].

Since inference using deep CNNs is computationally expensive, two main techniques have been developed to reduce the cost of inference. First, compression is a set of techniques that can dramatically reduce the cost of inference at the expense of accuracy. Such techniques include removing some expensive convolutional layers, reducing input image resolution. For example, ResNet18, which is a ResNet152 variant with only 18 layers, is 8x cheaper. Likewise, Tiny YOLO [24], a shallower variant of the YOLO object detector, is 5x cheaper than YOLOv2. However, the tradeoff is that compressed CNNs are usually less accurate than the original CNNs. The second technique is CNN specialization [24], where the CNNs are trained on a subset of a dataset specific to a particular context (such as a video stream). Specialization simplifies the task of a CNN because specialized CNNs only need to consider a particular context. For example, differentiating object classes in any possible video is much more difficult than doing so in a traffic video, which is likely to contain far fewer object classes (e.g., cars, bicycles, pedestrians). As a result, specialized CNNs can be more accurate and smaller at the expense of generality.

4.1.4 Video analytics

A video query is typically executed through a pipeline of filters that may include deep CNNs. However due to the high inference cost of CNNs, several works have been proposed to reduce video query's latency [23, 84, 96, 97]. NoScope [23] places the filters in the pipeline ordered by complexity, where the cheapest models sits in the front and most expensive model (e.g. a full object recognition NN) in the end. In this configuration, frames are executed in a short-circuit evaluation fashion, where a frame is passed onto the next filter only if the result from the current filter does not meet the accuracy requirement. The query latency is still quite high because most of the heavy lifting is performed at query time. Additionally, user has to train and maintain several specialized CNNs that are cheaper to execute at query-time. Focus [84] proposes to preprocess the frames during ingest time in order to reduce query latency. At ingest time, it uses a specialized CNN to construct an approximate index for query time. The index in Focus is approximate in order to keep a balance between ingest and query cost. These two works target retrieval queries only. Besides the retrieval queries, Blazeit [96] also handles queries for aggregate statistics, but it suffers similar issues with NoScope since the NN specialization technique does not reduce significant query latency. It also does not consider distributed processing of the video data, and the placement of video frames/segment for efficient retrieval.

4.1.5 Cluster sampling

Large datasets are typically partitioned such that a partition can naturally correspond to a cluster for cluster sampling. In this scenario, cluster sampling is especially efficient since it avoids the need to access clusters that are not selected for a sample, and so is used in many approximate computing systems, e.g., [6, 11, 12, 65].

Suppose we need to estimate the frequency τ of data item Z occurring in a dataset partitioned into subcollections. If we take a cluster sample from the dataset using subcollections as the sampling clusters, we can use the estimator $\hat{\tau} = \frac{1}{n}\sum_{s \in S} \frac{\tau_s}{\phi_s} \pm \epsilon$ [10], where S is the chosen sample, n is the number of subcollections in S, τ_s is the frequency

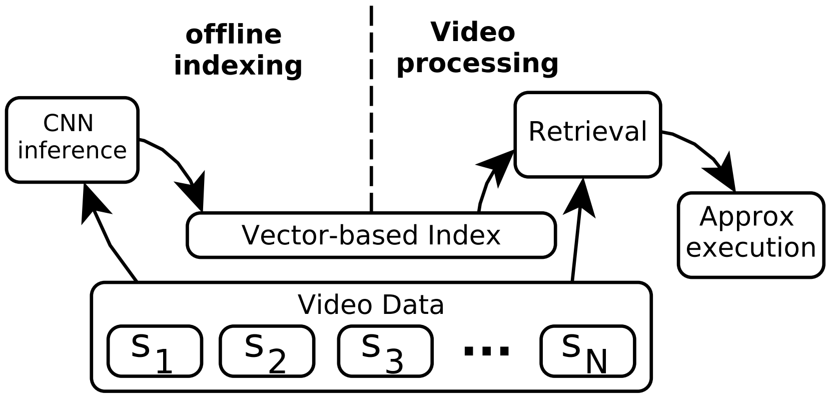


Figure 4.2: Overview: S_n are clusters of segments

of Z in subcollection s, ϕ_s is the sampling probability for s, and ϵ is the estimated error bound. $\epsilon \propto \sqrt{\hat{V}(\hat{\tau})}$, with

$$\hat{V}(\hat{\tau}) = \frac{\sum_{s \in S}(\frac{\tau_s}{\phi_s} - \hat{\tau})^2}{n(n-1)} \tag{4.1}$$

We observe that as the ϕ_s's approach $\frac{\tau_s}{\hat{\tau}}$, $\hat{V}(\hat{\tau})$ and hence ϵ will approach 0. The goal of *probability proportional to size* (pps) sampling [10] is to set each ϕ_s close to $\frac{\tau_s}{\hat{\tau}}$ by leveraging auxiliary information of each sampling unit, so that we can reduce the error bound for our estimator $\hat{\tau}$.

4.2 VidApprox Design

4.2.1 Overview

In this section, we present the design of VidApprox. Our proposed framework consists of offline indexing and online approximation components. The offline component is responsible for placement and indexing of video data. Online component handles the retrieval of relevant parts of the video for downstream processing operators. Figure 4.2 shows the overview of VidApprox. The offline index is produced mainly through applying inference over the object detection and image classification CNNs. The index will project the query and the video data into the same vector space, so that their similarity can be computed. The online sampling of a subset of the video data is facilitated by consulting the index that is persistent for retrieving the most relevant parts of the video to the query.

4.2.2 Segment vector representation

Under the bag-of-words model in computer vision [98], a frame can be viewed as a bag of objects. We propose to have include the information of the objects encoded in its vector representation, ignoring the background as we target video processing tasks that involve object detection. Similar to representing a text document using the average of word vectors [80], we use the average of the object vectors in the image as its vector representation.

Storing and moving a large number of raw video frames is costly, making encoded video segments an attractive option. An encoded video is a sequence of segments [81]), in which a segment contains near-duplicate frames. A key frame is representative of the entire segment, and is used for many applications such as browsing, searching, information retrieval, and indexing [81]. We encode the video data using a video encoder e.g., ffmpeg [99], and use segment as the smallest unit of storage in our system, as a segment is also the smallest data unit that can be decoded independently.

An object detection CNN can output the bounding boxes of the objects in an image. For each detected object, we use another image classification CNN to output its feature as a real-valued vector. As mentioned in the background section, the feature vector is the output of the second-to-last layer of an image classification CNN. Previous works suggest that if two images/objects have similar vectors, then they are also visually similar [84]. The distance between two vectors can be easily quantified using a distance measure such as cosine.

Since a key frame is representative of a segment, we use key frame's vector representation as the segment's vector. In order to keep the inference cost low for preprocessing, we choose to use a lightweight objection CNN (e.g., TinyYolo [24]) to detect the object boundaries then use another relatively shallow image classification CNN (e.g. ResNet50) to capture each individual object's features. Algorithm 5 shows the procedure for generating the vector representation for each segment. First the object CNN will detect the objects for every segment's key frame, and output their bounding boxes, where n is the number of objects in the key frame. Each of these objects will be fed through ResNet for feature extraction, the output of which is used as its vector representation. The final vector representation for the segment will be the arithmetic mean of these the objects' vectors. This will automatically take into account the weighting of the objects. Suppose a query is to estimate the number of cars across the video, then segments with a higher similarity to car, i.e. its vector has a relatively larger weight for the car object, its inclusion probability should be higher according to pps cluster sampling theory [10].

Algorithm 5: Segment vector generation

```
1 Function SegmentVectorGen({$S$})
2     for $S_i \in \{S\}$ do
3         Object detection($I_i$)$\rightarrow \{obj\}_n$;
4         Image classification$\{obj\}_n \rightarrow \{\vec{obj}\}_n$;
5         $\frac{1}{n}\sum_{i=1}^{n}\{\vec{obj}\} \rightarrow \vec{S_i}$;
6     end
```

In summary, a segment's vector representation contributes to the indexing, placement and retrieval of segments. The most similar segments to the query reference image can then be identified through cosine distance between the vectors of the reference of a segment. In order to sample the most similar segments, the sampling probability of each segment can be set proportional to its similarity to the reference image.

4.2.3 Locality sensitive hashing

We propose to compress the vectors using locality sensitive hashing. LSH bit vectors can preserve different distance metrics (e.g., cosine) between their real-valued vector counterparts. Computing the Hamming distance of the LSH bit vectors using `XOR` is also much more efficient than dot product. The value of the hash function for preserving cosine distance depends on the dot product between a random plane $\vec{r}$ and an item vector $\vec{x}$, where $h_r(\vec{x})$ evaluates 1 if $\vec{r} \cdot \vec{x} \geq 0$, and 0 otherwise, where $\vec{r}$ usually has a standard multi-dimensional Gaussian distribution $\mathcal{N}(\mathbf{0}, I)$, and a new $\vec{r}$ is generated each time the hash function is applied [63]. In order to generate the LSH signature for a real value vector, we first choose a dimension l for the bit vector, then apply hash function $h_r(\vec{x})$ l times to generate each bit, each choosing a random $\vec{r}$. Specifically, we can approximate $exp(\vec{w} \cdot \vec{d})$ using $exp(cos(\frac{m}{l}\pi))$, where m is the Hamming distance between $\vec{w}$ and $\vec{d}$'s corresponding LSH bit vectors.

4.2.4 Similarity-aware segment placement

As we know, a segment contains near-duplicate frames. On the other hand, there will be similar segments throughout a video dataset as well. Intuitively, if a “group” of segments are not similar to a reference image, then this group can be pruned altogether to avoid processing redundant segments. On the other hand, if a group of segments are similar to the reference image, then this group should be retrieved at once for improved retrieval efficiency. The method we form groups is by cluster the segments using K-means, using the cosine distance between segment vectors as distance metric. After clustering, we just use a cluster's centroid segment's vector as the cluster's vector representation. The vector representation for a cluster is obtained similarly as vector for a frame, which is the element-wise arithmetic mean of the segment vectors that exist in the same cluster.

We assume a large video dataset is stored in a distributed storage, and the number of segment clusters is greater than the number of partitions. If we cluster the segments too aggressively, i.e., there are two few centroids, the segments within a cluster may not be similar enough, such that the centroid segment is no longer representative. Within the same partition, we randomly place clusters into the partitions to lower the probability that similar clusters ending up in the same partition. The benefit is that for a group of similar clusters, I/O for loading and decoding the segments can be evenly distributed across the storage nodes rather than concentrated only on a few nodes, which may slow down processing.

4.2.5 Vector-based index structure

We store the index structure for segment clusters in database tables. Every row in the index table represents a cluster of video segments, including its cluster id, the ids of segments that in this cluster, vector representation of the cluster. We also use a another table to store the segment's vectors. We compress the vectors using distance-preserving LSH bit vectors, as the LSH compression would make storing segment vectors much more space-efficient. As part of the index structure, we also store the original object recognition/detection CNNs themselves to produce the query image's vector through neural network inference.

4.3 Approximate segment querying

We present two types of queries to demonstrate our system's efficiency. The first is aggregation and the second is retrieval. Common to both of the query types, the aim to identify relevant segments/frames without having to search through the entire data. We obtain the vector representation for the reference image. We apply cluster sampling to sample the clusters with probability proportional to the similarity to the reference image. Each cluster has a sampling probability using:

$$\phi_d(q) = \frac{\vec{q} \cdot \vec{s}}{\sum_{s' \in D} \vec{q} \cdot \vec{s'}} \tag{4.2}$$

In the querying process, the segment vector's cosine similarity to the query image's vector will be computed which is used as auxiliary information for unequal-probability cluster sampling. After a subset of segment clusters have been retrieved, only these segments will be fed through the objection detection CNN to further identify the segments of interest. We refer to the sampling method as similarity-driven sampling [80].

Aggregation. Aggregation query outputs statistics of the video, where user may want to estimate total the number of frames/segments that contain object X, or average number of object X per frame, or number of similar frames to the reference image. The output is a closed-form estimator plus or minus a central limit theorem based error bound [10]. The user also needs to supply a reference image containing the object that is of interest. The error bound and estimator can be estimated using Eq (4.1) from the cluster sampling theory [10]. For example, if the query is to estimate the average number of cars per segment. By processing the chosen clusters, we are able to estimate the average number of cars in the clusters of segments that are not chosen by leveraging the similarity as sampling auxiliary information.

Retrieval. If the frames of interest are sparsely distributed throughout the entire dataset, then significant processing time will be spent on identifying and retrieving those relevant frames.user wants to retrieve all segments/frames that contain object X. The retrieved frames can optional be sent to downstream operators for further consumption or simply written to stable storage. Instead of scrubbing the entire video data, we should bias the search to the most promising regions or segments of the video data. For example, an OCR application that needs to perform license plate recognition on all red cars. The application first needs to retrieve frames that contain at least a red car in the first place. A variant of retrieval query is to find a given number of segments containing object X. The use case is that user may want to search for certain events for manual inspection. As segments are processed in the order of their similarities, the search can stop once user has enough segments retrieved. This would significantly reduce the segments that need to be searched compared with randomly processing segments.

4.4 Implementation

We have implemented a prototype of our framework VidApprox as a Python library. It comprises two components: one for building offline video indexer and one for building the online approximate querying of the video segments. Figure 4.3 shows the key components of VidApprox. We provide library functions for user to write offline indexer and customized query processor.

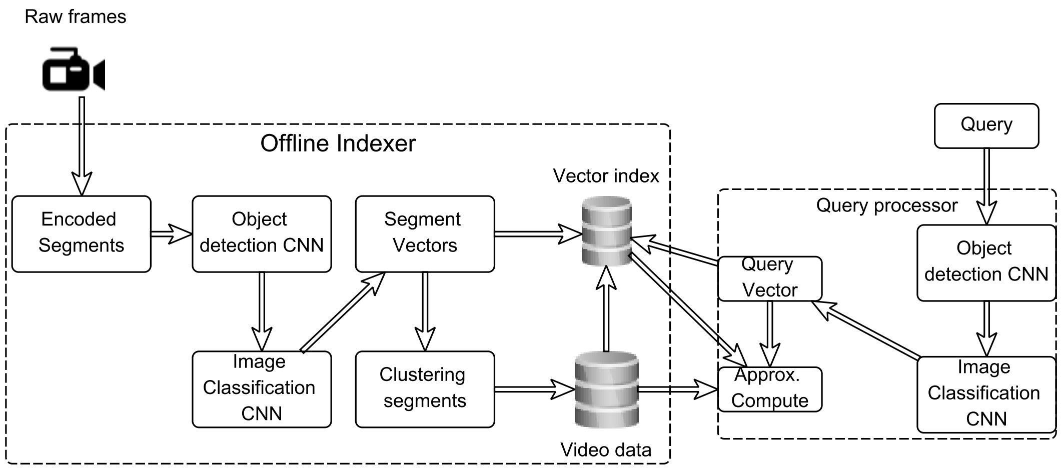


Figure 4.3: 2

4.4.1 Offline indexer

As previously mentioned, the offline part is mainly responsible for indexing and clustering/placing video segments. We adopt Yolo [24] and ResNet [100] respectively for object recognition and image classification. In the indexing process, we adopt the shallow versions of the CNNs respectively TinyYolo and ResNet50 in order to save preprocessing costs. Given a video dataset with raw frames, the index generation part will first encode the video with segments using ffmpeg [81]. Then inference will be performed over the CNNs to generate the vector representations for the segments following algorithm (5). We use Tensorflow [76] for the inference process. After the vectors have been generated, the segments will be clustered based on the cosine distance. Then the clusters will be randomly placed on HDFS, where each partition contains a number of clusters. We also save the standard Yolo and ResNet models for querying time.

We use Cassandra [101] as the storage for the index structure, for which we use two tables. The first table (CLUSTER) stores information for the segment clusters; the second table (SEGMENT) contains mapping between each segment and its vector representation. In the CLUSTER table, the row key is the cluster id, column keys are its vector representation, the HDFS partition id, and list of segment ids that are included in the cluster. In the SEGMENT table, the row key is the segment id, and the

ClusterId	PartitionId	Vector	Segments
0	12	010100...	{0,4,7..}
...	...	...	...

Table 4.1: Example index structure of CLUSTER table

column key is its vector representation. Note that we convert the real-valued vectors into LSH bit vector so we can just use blob data type for the vectors. We show example index tables below.

4.4.2 Query processor

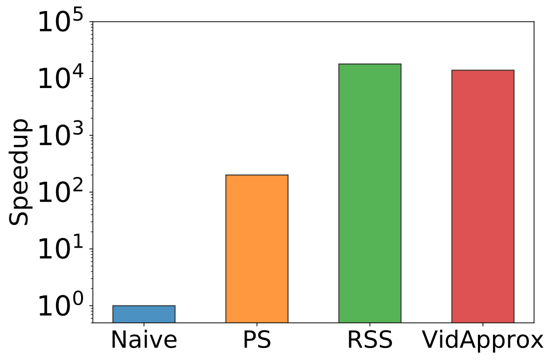
The reference image contained in the query will first be converted into a vector that encodes the objects in it. The query vector will be compared against each cluster vector where each cluster will be assigned a similarity score for sampling. The chosen clusters will have corresponding partitions. Then these partitions will be further processed. The key frames in each segment in the chosen clusters will be fed through YOLO and its inference result will determine whether to this segment should be returned. After the segments have been identified, they are either aggregated or retrieved depending on the specific user query.

4.5 Evaluation

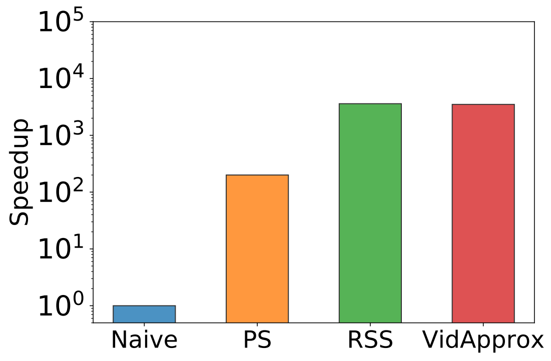
We evaluate VidApprox on a variety of aggregation and retrieval queries on a number of video streams, such as traffic feeds and news archives. We divide our queries into two categories: aggregation and retrieval, in which aggregation outputs statistics about the video and retrieval retrieves segments based on a user-specified predicate.

4.5.1 Methodology

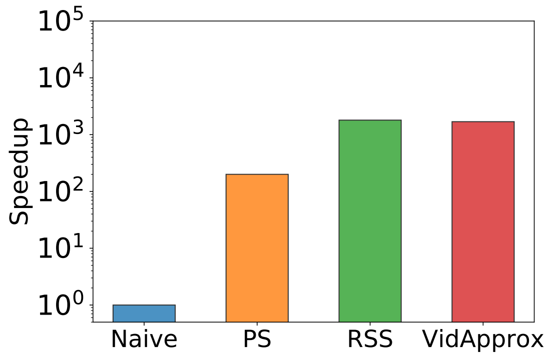
Software. We use ffmpeg [99] to extract the segments and keyframes the videos. VidApprox relies on Tensorflow trained CNNs for inference both at offline indexing and



(a) 1% Sampling rate

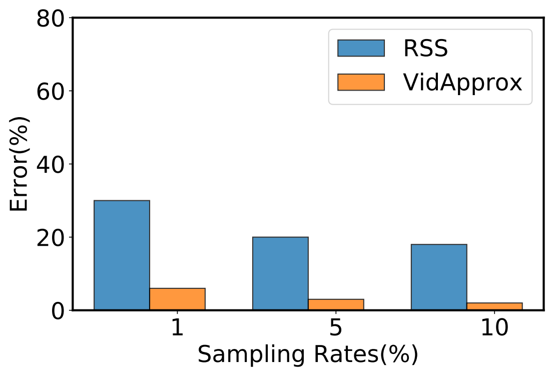


(b) 5% Sampling rate

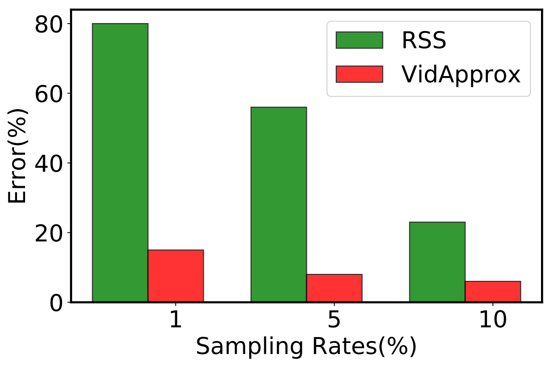


(c) 10% Sampling Rate

Figure 4.4: Comparison of speedups across variants of aggregation query under different sampling rates for RSS and VidApprox. The dataset is street. The speedups are relative to the naive execution. The Y axis is in log scale.



(a)



(b)

Figure 4.5: Comparison of relative error across variants of the aggregation query under different sampling rates for RSS and VidApprox. The dataset is street. (a) shows popular object class and (b) shows a relatively rare object class.

Name	Description	Length	FPS
street	Traffic camera from an intersection in Auburn	10 hr	30
aqua	An aquarium video	12 hr	30
news	News clip	12 hr	30

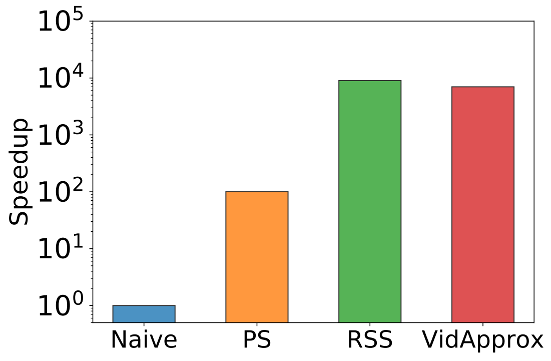
Table 4.2: Datasets

online querying stages. The vector-based segment index is stored in Cassandra DB [101].

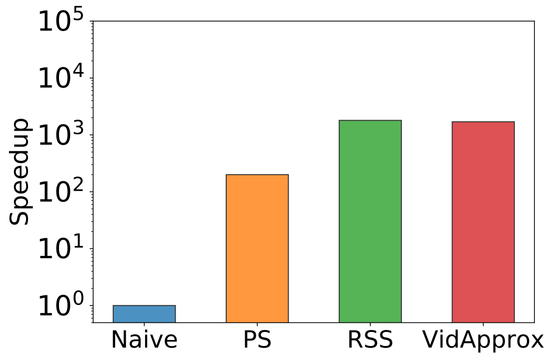
Hardware platform. Experiments are run on a cluster of 2 servers interconnected with 1Gbps Ethernet. Each server is equipped with a 2.5GHz Intel Xeon CPU with 8 cores, 64GB of RAM, and a 100GB hard drive. Data sets are stored in an HDFS file system across the same server cluster.

Data sets. We evaluated video streams that span across traffic camera feeds to news clips scraped from Youtube. The description is shown in table 4.2.

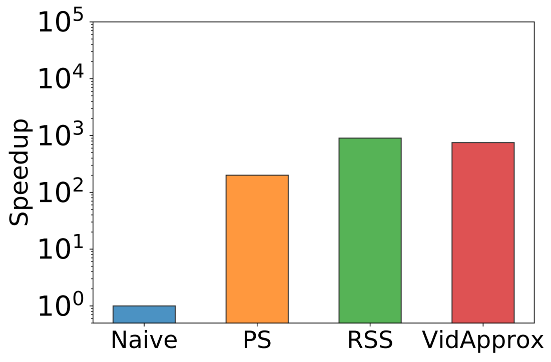
Baselines. We run the following variants to demonstrate VidApprox's effectiveness and how it can trade-off accuracy with execution time. The first three are compared



(a) 1% Sampling rate



(b) 5% Sampling rate



(c) 10% Sampling rate

Figure 4.6: Comparison of speedups across variants of retrieval query under different sampling rates for RSS and VidApprox. The dataset is the aqua. The speedups are relative to the naive execution. The Y axis is in log scale.

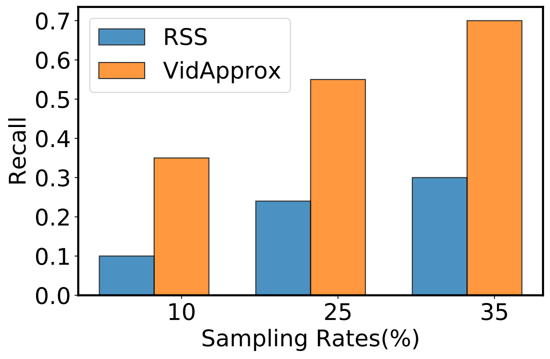


Figure 4.7: Comparison of recall across variants of retrieval query under different sampling rates for RSS and VidApprox. The dataset is aqua.

against as baselines.

- Naive: performs object detection on every raw frame.
- PS: Precise execution on encoded segments. Performs object detection on all segments with no sampling.
- RSS: Simple random sampling on segments. Performs random sampling over segments, then object detection on the chosen segments.
- VidApprox: performs similarity-driven cluster sampling over segments, then object detection on the segments from the chosen clusters.

4.5.2 Aggregation

We evaluate VidApprox on the following approximate aggregation queries across multiple datasets with specified target object type contained in the data: 1) Average number of object X per segment; 2) Number of segments that contain object X. Optionally user may supply beginning and ending time stamps to filter the segments, such as from time t_1 to time t_2. For example, for a traffic video feed, user may be interested in the average number of cars per frame during a certain time period, in order to see the traffic condition during that time. If a time filter is specified, we first narrow down the segments we would like to search by filtering the timestamp attached to each segment. The metrics we report is error bound around the estimator, and end-to-end speedups. User needs to supply a sampling rate when running these queries. Our results below are from the query on the traffic cam video to count the average number of cars per frames.

Speedup. Figure 4.4 shows the speedups across variant setups of the aggregation query under different sampling rates. We observe that PS, which is object detection on the segment level (key frames) can already outperform the naive execution significantly. With sampling, both RSS and VidApprox can outperform PS by speedups scaling almost linearly with the sampling rates. For example, under a 1% sampling rate, RSS and VidApprox can achieve a speedup of almost 10^4x compared with Naive. They can also outperform PS by almost 80x. However, VidApprox does have to pay a preprocessing cost for building the vector-based index and clustering. The actual VidApprox speedup is less than RSS as looking up the index tables involves a small overhead.

Estimation error. We report the relative error, computed as $\frac{|\hat{\tau}-\tau|}{\tau}$, where τ is the ground truth answer. Figure 4.5(a) shows the relative error across the variants of the query under different sampling rate for a relatively popular object class car. The Naive and PS execution are the ground truth thus not shown in the graph. We can see that VidApprox constantly outperforms RSS under different sampling rates. For example, under 1% sampling rate, VidApprox can outperform RSS by almost 6x. It is because under VidApprox, the sampling probability of the clusters are adjusted proportional to their similarity to the target object. RSS and VidApprox can achieve similar speedups, but VidApprox has to pay a preprocessing cost and storage of the index tables. It is similar to what we have observed in EmApprox [80]. We have also observed in Figure 4.5(b) that VidApprox can achieve a smaller error when the aggregation target object is relatively rare. For example, estimating average number of cars has smaller error bound than estimating average number of pedestrians, since car is more popular in street video data. Furthermore, we have also found VidApprox helps reduce the error for aggregation for more rare objects also shown in Figure 4.5.

4.5.3 Retrieval

For retrieval queries, we run the following retrieval query: find segments with at least one object X. The metric we report is recall, which is defined as ratio of the number of segments retrieved by the approximate query processing to the ground truth. Similarly user may specify a time range as the additional filter as with the aggregation query. Our results below are from the query.

Speedup. Figure 4.6 shows the speedups across variant setups of the retrieval query under different sampling rates. We observe that PS, which is object detection on the segment level (key frames) can already outperform the naive execution significantly. With sampling, both RSS and VidApprox can outperform PS by speedups scaling almost linearly with the sampling rates. This is inline with our aggregation speedup results.

Recall. Figure 4.7 shows the recall across the variants of retrieval query under different sampling rates. We can see that VidApprox constantly outperforms RSS under different sampling rates. For example, under 10% sampling rate, VidApprox can outperform RSS by almost 3x. However, compared with aggregation queries, retrieval queries do require a higher sampling rate since it is more sensitive to missing data items.

4.5.4 Summary

VidApprox can speed up aggregation and retrieval queries significantly if the user can tolerate modest imprecision and afford preprocessing. In addition, VidApprox can gracefully trade off imprecision with performance by adjusting sampling rates. The result is more pronounced when user queries for relatively rare object types in the video.

4.6 Conclusion

We present VidApprox, a system that provides integrated support for distributed video analytics by indexing the video segments, and facilitates the efficient storage and retrieval of relevant video segments. Our experiment shows that our vector-based index for video segments can reduce the execution time significantly while retrieving a large number of relevant segments, as well as degrading gracefully in approximation quality with decreasing sampling rates. This is particularly useful in the retrieval phase of a video processing pipeline, for reducing the processing needs of potentially expensive downstream operators.

Chapter 5

Conclusion

Approximate computing has emerged as a powerful tool for reducing the processing needs to cope with growing volumes of data. This thesis has shown approximate computing can be effective toward efficiently processing a variety of tasks with different types of data. We mainly employ sampling-based approximation methods, built infrastructure mechanisms to allow the application of existing statistical theories, and/or utilizing machine learning models learned offline from the data as index structure for enhanced online approximation quality. We show that our approaches can gracefully tradeoff large execution reduction with a tolerable loss in computation accuracy.

As data sets grow exponentially, batch processing jobs are becoming more heterogeneous and time consuming than ever. Though approximation has been shown as an effective method for reducing processing time, it can present different challenges when applied to different types of datasets and application domains. As we know, today's large data sets are characterized by the the three V's (*Volume*, *Variety*, *Velocity*). However, past approximation research as been focusing on relational data, and today's popular distributed data processing frameworks such as Spark still lack approximation mechanisms that can adapt to these different scenarios. Our proposed approximation structures, such as provenance tree, vector-based index; as well as sampling strategies such as similarity-driven sampling can be incorporated into a distributed data processing framework to bridge the gap between today's characteristics of big data and current approximation techniques. As conclusion, we present a summary of our works and lay out an outlook for future work.

5.1 Summary

- **ApproxSpark** - a sampling framework to support approximate computing with estimated error bounds in Spark. It 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. It also includes an algorithm to dynamically choose sampling rates to meet user-specified constraints on the CDF of error bounds in the outputs.

- **EmApprox** - a sampling-based approximation framework to speed up the processing of a wide range of queries over large text datasets. It builds an index for a data set by learning a natural language processing model, producing a set of highly compressed vectors representing words and subcollections of documents. When processing a query comprising one or more words, a vector representing the query is computed from the vectors representing the words. It then samples the data set, with the probability of selecting each subcollection of documents being proportional to its *similarity* to the query derived from the vectors.

- **VidApprox** - a system that provides integrated support for video queries through indexing and placement of encoded frames through approximation in a distributed system. It consists of two components, offline indexing and online approximation. The primary objective of indexing is early pruning of irrelevant frames, where a user's target and groups of frames' similarities can be pre-computed. It uses a CNN to learn feature vectors in offline indexing, that is used to index and cluster segments. The query component is approximate in nature. At query time, a subset of most similar clusters of segments to the query are sampled by consulting the vector-based index.

5.2 Future Work

In our work, the user has to provide sampling ratios over the input dataset, that will be determined by the available computational budget. However, the relationship between sampling rate and processing need is not so clear, so that the user needs to rely on prior experience to set an appropriate sampling rate. Future versions of our approximation frameworks can allow user to submit more intuitive representation of processing budget. It can be in the form of latency guarantee, desired energy consumption, etc. This will require the system to construct profiles of processing needs for the supported tasks from prior runs, that will be available at online query processing time.

We also believe our proposed model-based index structure can be extended to accelerating more big data processing applications involving other datasets. For example, it can be used to efficiently detect the sentiment for a large text dataset; it can also potentially be used toward selecting relevant users/products for more complex recommendation methods, such as model-based algorithms involving multiple data sources. We believe plan our methodology – learning an index directly from the data set to facilitate approximate computation – is promising toward other applications, such as query against audio, time series, bioinformatics data, etc. As future research, it will be interesting to see how effective our approximation methodology may be extended to these areas, and help answer user-specified queries with reduced latency with tolerable loss in accuracy.

References

[1] (2020) Volume of data. https://www.statista.com/statistics/871513/worldwide-data-created/.

[2] 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.

[3] 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.

[4] I. Lee, "Big data: Dimensions, evolution, impacts, and challenges," *Business Horizons*, vol. 60, no. 3, pp. 293–303, 2017.

[5] D. Quoc, "Approximate data analytics systems," Ph.D. dissertation, PhD thesis, Technische Universität Dresden (TU Dresden), 2017.

[6] 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

[7] M. N. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes." in *VLDB*, 2001, pp. 343–352.

[8] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

[9] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 1997, pp. 171–182.

[10] S. Lohr, *Sampling: Design and Analysis.* Cengage Learning, 2009.

[11] 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

[12] 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*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465355

[13] 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.

[14] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 481–492.

[15] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International journal of information management*, vol. 35, no. 2, pp. 137–144, 2015.

[16] 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

[17] "Google Book Ngrams," 2019, http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

[18] "Common crawl," https://registry.opendata.aws/commoncrawl/, 2018.

[19] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proceedings of the 7th ACM Conference on Recommender Systems*, ser. RecSys '13. New York, NY, USA: ACM, 2013, pp. 165–172. [Online]. Available: http://doi.acm.org/10.1145/2507157.2507163

[20] "YouTube Stats," 2019, https://merchdope.com/youtube-stats/.

[21] T. Xu, L. M. Botelho, and F. X. Lin, "Vstore: A data store for analytics on large videos," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, p. 16.

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[23] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1586–1597, 2017.

[24] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[25] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for iot applications in smart cities," *Journal of Big Data*, vol. 6, no. 1, p. 52, 2019.

[26] H. Kavalionak, C. Gennaro, G. Amato, C. Vairo, C. Perciante, C. Meghini, and F. Falchi, "Distributed video surveillance using smart cameras," *Journal of Grid Computing*, vol. 17, no. 1, pp. 59–77, 2019.

[27] 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.

[28] 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.

[29] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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

[37] M. Al-Kateb and B. S. Lee, "Adaptive stratified reservoir sampling over heterogeneous data streams," *Information Systems*, vol. 39, pp. 199–216, 2014.

[38] 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

[39] 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.

[40] 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.

[41] M. Thottethodi, T. Vijaykumar, M. Kulkarni *et al.*, "Stratified online sampling for sound approximation in mapreduce," 2015.

[42] 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.

[43] 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

[44] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting 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

[45] 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

[46] 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.

[47] G. Hu, D. Zhang, S. Rigo, and T. D. Nguyen, "Approximation with error bounds in spark," *arXiv preprint arXiv:1812.01823*, 2018.

[48] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[49] M. D. Bankier, "Power allocations: determining sample sizes for subnational areas," *The American Statistician*, vol. 42, no. 3, pp. 174–177, 1988.

[50] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data.* Springer, 2006, pp. 25–71.

[51] "MEDLINE Data," 2017, https://www.nlm.nih.gov/databases/download/pubmed_medline.html/.

[52] 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

[53] 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

[54] B. Liu, "Sentiment analysis and opinion mining," *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167, 2012.

[55] (2011) Tweets 2011. http://trec.nist.gov/data/tweets/.

[56] 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

[57] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Tech. Rep., 1999.

[58] "Wikipedia database," http://en.wikipedia.org/wiki/Wikipedia_database., 2018.

[59] (2016) Wikipedia clickstream. https://meta.wikimedia.org/wiki/Research:Wikipedia_clickstream/.

[60] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 489–504.

[61] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

[62] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on.* Ieee, 2010, pp. 1–10.

[63] P. Indyk and R. Motwani, “Approximate nearest neighbors: Towards removing the curse of dimensionality,” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’98. New York, NY, USA: ACM, 1998, pp. 604–613. [Online]. Available: http://doi.acm.org/10.1145/276698.276876

[64] F. Ricci, L. Rokach, and B. Shapira, “Recommender systems: introduction and challenges,” in *Recommender systems handbook*. Springer, 2015, pp. 1–34.

[65] G. Hu, S. Rigo, D. Zhang, and T. Nguyen, “Approximation with error bounds in spark,” in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Oct 2019, pp. 61–73.

[66] M. Al-Kateb and B. S. Lee, “Adaptive stratified reservoir sampling over heterogeneous data streams,” *Inf. Syst.*, vol. 39, pp. 199–216, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.is.2012.03.005

[67] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[68] Q. Ai, L. Yang, J. Guo, and W. B. Croft, “Analysis of the paragraph vector model for information retrieval,” in *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, ser. ICTIR ’16. New York, NY, USA: ACM, 2016, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/2970398.2970409

[69] O. Levy and Y. Goldberg, “Neural word embedding as implicit matrix factorization,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’14. Cambridge, MA, USA: MIT Press, 2014, pp. 2177–2185. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969033.2969070

[70] X. Wei and W. B. Croft, “Lda-based document models for ad-hoc retrieval,” in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 2006, pp. 178–185.

[71] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2015, vol. 283.

[72] S. Robertson, H. Zaragoza *et al.*, “The probabilistic relevance framework: Bm25 and beyond,” *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[73] M. D. Hirschhorn, “The am-gm inequality,” *The Mathematical Intelligencer*, vol. 29, no. 4, pp. 7–7, 2007.

[74] “PySpark,” 2017, https://spark.apache.org/docs/latest/api/python/index.html.

[75] “Gensim,” 2018, https://radimrehurek.com/gensim/.

[76] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[77] "Common crawl news dataset," http://commoncrawl.org/2016/10/news-dataset-available., 2016.

[78] "Apache Solr," 2019, http://lucene.apache.org/solr/.

[79] "http://lucene.apache.org/," 2019, http://lucene.apache.org/.

[80] G. Hu, Y. Zhang, S. Rigo, and T. D. Nguyen, "Similarity driven approximation for text analytics," *arXiv preprint arXiv:1910.07144*, 2019.

[81] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 363–376.

[82] U. Gawande, K. Hajari, and Y. Golhar, "Deep learning approach to key frame detection in human action videos," in *Recent Trends in Computational Intelligence*. IntechOpen, 2020.

[83] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *2009 IEEE 12th international conference on computer vision*. IEEE, 2009, pp. 2130–2137.

[84] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu, "Focus: Querying large video datasets with low latency and low cost," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 269–286.

[85] V. E. Ogle and M. Stonebraker, "Chabot: Retrieval from a relational database of images," *Computer*, vol. 28, no. 9, pp. 40–48, 1995.

[86] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic *et al.*, "Query by image and video content: The qbic system," *computer*, vol. 28, no. 9, pp. 23–32, 1995.

[87] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini, "Visual query systems for databases: A survey," *Journal of Visual Languages & Computing*, vol. 8, no. 2, pp. 215–260, 1997.

[88] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[89] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[90] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[91] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky, "Neural codes for image retrieval," in *European conference on computer vision.* Springer, 2014, pp. 584–599.

[92] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[93] G. B. Huang, M. Mattar, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database forstudying face recognition in unconstrained environments," 2008.

[94] A. Babenko and V. Lempitsky, "Aggregating deep convolutional features for image retrieval," *arXiv preprint arXiv:1510.07493*, 2015.

[95] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[96] D. Kang, P. Bailis, and M. Zaharia, "Blazeit: Fast exploratory video queries using neural networks," *arXiv preprint arXiv:1805.01046*, 2018.

[97] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* ACM, 2018, pp. 253–266.

[98] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.

[99] F. Team, "Ffmpeg," *URL http://FFmpeg. org*, 2013.

[100] Z. Wu, C. Shen, and A. Van Den Hengel, "Wider or deeper: Revisiting the resnet model for visual recognition," *Pattern Recognition*, vol. 90, pp. 119–133, 2019.

[101] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.