Approximate Data Analytics Systems

by

Do Le Quoc

A dissertation submitted for the degree of
Doktoringenieur (Dr.-Ing.)

in

Technische Universität Dresden

Faculty of Computer Science
Institute of Systems Architecture
Chair of Systems Engineering

Supervisors:
Prof. Dr. Christof Fetzer
Prof. Dr. Pramod Bhatotia

Submitted November 5, 2017

Abstract

Today, most modern online services make use of big data analytics systems to extract useful information from the raw digital data. The data normally arrives as a continuous data stream at a high speed and in huge volumes. The cost of handling this massive data can be significant. Providing interactive latency in processing the data is often impractical due to the fact that the data is growing exponentially and even faster than Moore's law predictions. To overcome this problem, approximate computing has recently emerged as a promising solution. Approximate computing is based on the observation that many modern applications are amenable to an approximate, rather than the exact output. Unlike traditional computing, approximate computing tolerates lower accuracy to achieve lower latency by computing over a partial subset instead of the entire input data. Unfortunately, the advancements in approximate computing are primarily geared towards batch analytics and cannot provide low-latency guarantees in the context of stream processing, where new data continuously arrives as an unbounded stream. In this thesis, we design and implement approximate computing techniques for processing and interacting with high-speed and large-scale stream data to achieve low latency and efficient utilization of resources. To achieve these goals, we have designed and built the following approximate data analytics systems:

- StreamApprox—a data stream analytics system for approximate computing. This system supports approximate computing for low-latency stream analytics in a transparent way and has an ability to adapt to rapid fluctuations of input data streams. In this system, we designed an online adaptive stratified reservoir sampling algorithm to produce approximate output with bounded error.
- IncApprox—a data analytics system for incremental approximate computing. This system adopts approximate and incremental computing in stream processing to achieve high-throughput and low-latency with efficient resource utilization. In this system, we designed an online stratified sampling algorithm that uses self-adjusting computation to produce an incrementally updated approximate output with bounded error.
- PrivApprox—a data stream analytics system for privacy-preserving and approximate computing. This system supports high utility and low-latency data analytics and preserves user's privacy at the same time. The system is based on the combination of privacy-preserving data analytics and approximate computing.
- ApproxJoin—an approximate distributed joins system. This system improves the performance of joins — critical but expensive operations in big data systems. In this system, we employed a sketching technique (Bloom filter) to avoid shuffling non-joinable data items through the network as well as proposed a novel sampling mechanism that executes during the join to obtain an unbiased representative sample of the join output.

Our evaluation based on micro-benchmarks and real world case studies shows that these systems can achieve significant performance speedup compared to state-of-the-art systems by tolerating negligible accuracy loss of the analytics output. In addition, our systems allow users to systematically make a trade-off between accuracy and throughput/latency and require no/minor modifications to the existing applications.

This dissertation is lovingly dedicated to my mother.

Acknowledgements

First of all, I would like to thank my advisor Prof. Christof Fetzer for providing an opportunity to join Systems Engineering chair at TU Dresden to pursue doctoral study. He gave me freedom to choose my research topics and motivated me to take challenges in doing research. Of course, the road was not always smooth and had its ups and downs. Fortunately, it works out at the end. Second, I would like to thank my co-advisor Prof. Pramod Bhatotia for taking me out of my comfort zone and guiding me to go through difficulties of my PhD life. He spent a tremendous amount of time helping me improving research skills. I have learned a lot from him. Third, I would like to thank Prof. Thorsten Strufe for providing me the deep knowledge in privacy research, supporting me at the final stage of my PhD, and being my "Fachreferent". In addition, I would like to thank Prof. Jan S. Rellermeyer for serving as an external reviewer of my thesis.

Next, I would like to thank my collaborators at Bell Labs (Ruichuan Chen, Dr. Istemi Ekin Akkus, and Dr. Volker Hilt), at TU Dresden (Martin Beck and Dhanya R Krishnan), and at the Ohio State University (Prof. Spyros Blanas). Without their support and technical contribution, I would not be able to publish papers in top-tier conferences. The collaboration with them was amazing.

Further, I would like to thank the members of the Systems Engineering Group at TU Dresden (Dmitry, Sergey, Franz, Rasha, Oleksii, and Bohdan). They commented on and discussed all my early paper drafts and encouraged me when I felt stuck. It was great having them around during my PhD.

Next, I would especially like to thank my amazing mom for her love, support, and constant encouragement. My mom has sacrificed her life to raise me and my sister. She once told me that studying is the only way out of poverty. Thus, she was willing to sell everything in our house for her children to go to school to have a better life.

Finally, I would especially like to thank Trang for cooking Vietnamese food for me and being with me during the time I was in Germany for my PhD.

Publications

The content of this thesis is based on the following publications.

- **StreamApprox: Approximate Computing for Stream Analytics**. Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. In *proceedings of the annual ACM/IFIP/USENIX Middleware conference (Middleware)*, 2017.
- **IncApprox: A Data Analytics System for Incremental Approximate Computing**. Dhanya R Krishnan , Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. In *proceedings of the 25th International World Wide Web Conference (WWW)*, 2016.
- **PrivApprox: Privacy-Preserving Stream Analytics**. Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. In *proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- **ApproxJoin: Approximate Distributed Joins**. Do Le Quoc, Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. *Submitted to the International Conference on Very Large Data Bases (VLDB)*, 2018.

Contents

List of Figures

List of Tables

1 Introduction

Over the last ten years, the growth of digital data has accelerated exponentially. In 2017, Twitter users generated $456,000$ tweets, Instagram users uploaded $46,740$ photos, and Google conducted $3,607,080$ searches every minute[1]. The volume of digital data is projected to reach 40 zettabytes[2] or more by 2020 [126, 185]. In modern online services, which are mostly data-driven services, the need for real-time data analytics will continue to increase with high rate since it is crucial for their business. To exploit and mine rich and valuable information, data analytics systems are required to scale and adapt in order to handle vast amounts of data, high acquisition velocities, and a great variety of data sources and types. These challenges are associated with a hot topic in industry and academia called "big data". These challenges also are described with "3Vs" [138], namely, *Volume* which reflects the massive data volumes, *Velocity* which implies the rapid growth of digital data, and *Variety* which refers to the diversity of data from various sources (e.g., sensors, web clickstreams, and social media) and also the types of data such as structured data (e.g., relational data), semi-structured data (e.g., XML and JSON), and unstructured data (e.g., video, audio, and plain text).

1.1 Brief History of Big Data Systems

Traditional technologies for analyzing data using SQL-based RDBMS and data warehousing became impractical for handling such vast amounts of data [30, 53]. Thus, cluster computing with a shared-nothing architecture naturally has become a design choice for building big data analytics systems. In these systems, data is partitioned and distributed on clusters of commodity machines, and computations are performed on the same machines with data locality-aware scheduling, fault tolerance, and scalability. Figure 1.1 shows briefly the history of big data systems development. Google pioneered this trend by introducing MapReduce [75] programming model, followed shortly by its open-source implementation, Apache Hadoop [222]. MapReduce is a simple and general programming model for parallel data analytics. One of fundamental principles of it is to move computations to the data for processing rather than moving data to avoid network bottlenecks. In addition, it allows users to focus on computation on the data and the details of distributed execution and fault tolerance are transparently handled by the framework itself.

MapReduce works well for several workloads such as Web log analysis and text processing. However, it has been shown to be ineffective for other types of workloads: (*i*) iterative processing [89, 229] such as machine learning [153] and graph processing [153, 154], (*ii*) data stream processing [227].

For iterative programs, the classical MapReduce framework issues a MapReduce job for each

[1] https://www.domo.com/learn/data-never-sleeps-5

[2] A zettabyte corresponds to one thousand exabytes, that is 10^{21} bytes.

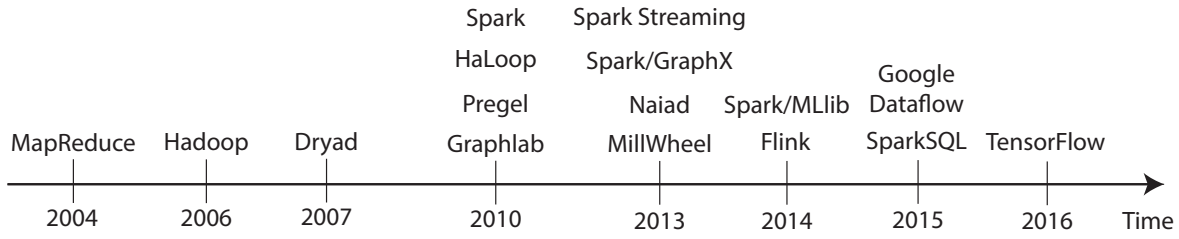

Figure 1.1 – Timeline of the big data analytics systems development.

iteration and handles the intermediate output of each iteration via a driver program. This incurs significant latency since, although the major data remains unchanged from one iteration to the next, the data needs to be loaded and processed repeatedly at each iteration, thus wasting not only computing resources but also network bandwidth. As a result, several variants of MapReduce have been developed to overcome this limitation. For example, HaLoop [48] caches the unchanged data in the first iteration and reuses them in the next iterations whereas iMapReduce [231] pipelines the intermediate data between tasks and between jobs to support iterative computations.

In addition, many complex computations such as graph processing and machine learning are not naturally suitable for only two phases of map and reduce. Dryad [127] overcomes this problem by using not only the two map-reduce phases, but arbitrary Directed Acyclic Graphs (DAGs) to describe computations. In a DAG, vertices represent operations and directed edges represent the communication between the operations. The operations are user-defined and the communication can be established through a variety of channels including files, shared memory FIFO, and TCP pipes. Inspired by the DAG-based dataflow concept, Apache Spark [21, 22, 229], introduces a flexible data-sharing abstraction called Resilient Distributed Datasets (RDDs) allowing to implement in-memory operations. With this abstraction, Spark can support multiple computing workloads including graph processing, machine learning, and SQL in a unified computation engine. Recently, Tensorflow [2] has been designed specifically for deep learning workloads.

Much of digital data is produced and generated as a continuous data stream and in huge volumes (e.g., video streaming, data streams from IoT sensors, and Twitter's stream data). Real-time analytics of this stream data is hard. In principle, the fundamental design of MapReduce requires the map phase to be completed *before* the reduce phase starts. All intermediate data is persisted to hard disks before sending to reducers. Thus, it introduces significant latency and makes this computing model unfit for stream processing. As a result, a wide range of dataflow-based stream processing systems has been proposed both in academia and industry such as Google MillWheel [210], Naiad [168], Spark Streaming [23, 227], and Google Dataflow [9], Apache Flink [17], IBM Streams, Microsoft StreamInsight, and Amazon Kinesis.

These distributed stream processing systems can be classified into two prominent categories: (*i*) batched stream processing model, and (*ii*) pipelined stream processing model. In batched stream processing systems such as Spark Streaming [23, 227] and FlumeJava [54], an input data stream is considered as a sequence of micro-batches and each micro-batch is processed by a distributed data-parallel job. This model introduces high latency since it requires forming micro-batches. However, it allows the fault tolerant mechanism in these

systems to use partial results for recovery. In addition, the recovery process can be performed in a parallel and distributed manner. On the other hand, in pipelined stream systems such as Apache Flink and Naiad, each arriving data item is forwarded to the next operator in DAG as soon as it is ready to be processed without forming the whole batch. Thus, this processing model achieves lower latency. However, the recovery from failures is more expensive since it requires the state of operators to be recovered from the last checkpoint.

Recently, the Databricks team introduced a new computing model for processing high-speed data streams in Apache Spark, namely *Structured Streaming*. Instead of treating the input data stream as a sequence of micro-batches as Spark Streaming, Structured Streaming treats the data stream as an append-only input table [202]. A new data item of the stream is considered as a row appended to the table. This design enables real-time stream processing with high throughput and low latency. In addition, it allows users to analyze the input stream by incrementally executing SQL queries on the table.

1.2 The Need for Approximation

Analyzing data to extract valuable information quickly is critical in terms of business of most online services. These data-driven services are typically user-interactive applications (e.g., search engines, recommendation systems, and social networks) which require strict latency constraints. With the rapid growth of data, meeting strict latency requirements in the process becomes really difficult.

Typically, the performance (throughput and latency) constraints can be met by employing more computing resources. As mentioned above, almost all data analytics systems in modern online services are based on the data-parallel programming model [17, 21, 75], thus they are able to achieve almost linear scalability by increasing computing resources, i.e., adding more CPU cores (scale-up) and worker nodes (scale-out). However, this comes at a significantly higher computing cost, which is not always acceptable in practice because of resource budget limits. For example, to speed up the data analytics $10\times$ by horizontal scaling, we need to pay $10\times$ more for the computing resources we added. This economic problem needs to be carefully considered in modern online services. In addition, in large-scale services such as social media, video streaming, and stock market analysis, the growth of data over time will quickly overwhelm available resources. Briefly, data analytics systems of modern online services have two desirable, but contradictory design requirements [168, 227]: (*i*) achieving high-throughput/low-latency and (*ii*) efficient resource utilization.

To address this dilemma, we need to reduce the amount of data that needs to be processed. A promising solution for this approach is *approximate computing* which is based on the observation that many modern online services do not always require exact data analytics results (e.g., recommendation services, sentiment analysis, and stock market trend prediction). For these services, the trends extracted from the data are more important than exact statistical results. In addition, in some cases, it is impossible to get exact results since the input data often contains noise (e.g, the data from IoT devices may contain inherent measurement errors). Thus, for these services, we can apply *approximate computing* to strike a balance between throughput/latency and resource constraints. Since approximate computing paradigm performs analytics over a subset of data items instead of the entire datasets, it requires less time and resources, and thus allows to achieve desirable latency and efficient resource utilization.

1.3 The Promise of Approximate Computation

Approximate computing has recently emerged as a promising solution to reduce the computing resources usage, processing time, and even energy consumption of data analytics systems [7, 8, 103]. As mentioned above, the computing paradigm is based on the observation that for many modern online services it is acceptable to approximate rather than produce exact output results. Such services include search engines (Google, Microsoft Bing, Yandex, and Yahoo search), recommendation systems (Youtube, Facebook, Amazon, and Netflix), speech recognition (Apple Siri and Google voice search), and computer vision (online games). The "correctness" or quality of output of these services is defined as providing good enough or sufficient quality of results for users satisfaction. In addition, several applications have to accept approximate results since their input data contains noise. Thus, even if a data analytics system processes the whole input data, it cannot produce an exact result.

As a result, for these services and applications, it is possible to make a trade-off between accuracy, high-throughput/low-latency and efficient resource utilization by computing over a partial subset instead of the entire large-scale input data. This trade-off domain can be depicted as a "throughput/latency – resource utilization – accuracy" triangle (see Figure 1.2). The domain can be further extended with energy efficiency or even privacy-preserving analytics objectives.

The idea behind approximate computing is to build synopses of the original massive data to quickly answer analytics queries by executing them over the synopses instead of the original data. In general, approximate computing techniques provide mechanisms to summarize the massive data in such a way that this summary (synopsis) of the data effectively captures all features and attributes that we are interested in. Ideally, the synopsis first should be small compared to the original massive data, such as it can be kept and processed in our system. Thus, it allows providing approximate analytical results with a significantly lower latency. Second, it should be able to incrementally update with new arriving data. Finally, it should be able to be processed to get the insights.

Over the last decade, approximate computing has been applied to various levels of the computing stack for both hardware and software [91, 224, 225]. In detail, applications of approximate computing have been explored in various domains such as programming languages [29, 194], hardware design [193], query processing [7, 59, 200], and distributed systems [70, 121, 171]. Various approximation techniques have been proposed including sampling [11, 100], sketches [73], and online aggregation [121, 171] to build the synopses. These techniques make different trade-offs with respect to the output quality, supported query interface, and workload. However, the early work in approximate computing mainly targeted the centralized database architecture, and it was unclear whether these techniques could be extended in the context of big data analytics.

Among these approximation techniques, sampling has been widely applied for approximate computing since this approach has the ability to provide probabilistic error bounds for approximate results; it also fits well with a wide range of applications including graph processing [197] and machine learning [49, 139]. Recently, sampling-based approaches have been successfully adopted for distributed data analytics [7, 8, 103, 204]. Unfortunately, these "big data" systems are mainly targeted towards batch processing, where the input data remains unchanged during the course of sampling. Therefore, these systems cannot be used for stream analytics, which requires real-time continuous processing of input data streams. One of challenges

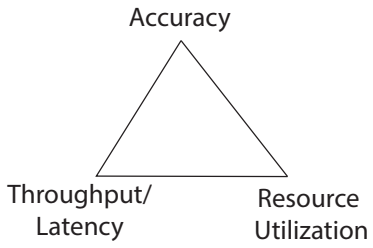

Figure 1.2 – Throughput/latency, resource utilization, and accuracy triangle.

in building approximate data anatlytics systems is to solve high-speed large-volume data streams in distributed environments and producing approximate results for various queries with resonable latencies.

1.4 Thesis Research: Approximate Data Analytics Systems

Thesis Statement. The aim of our work is to design and implement approximation techniques to build scalable resilient systems for processing and interacting with high-speed large-volume datasets with low latency and efficient utilization of resources. In other words, in this thesis, we design systems to process big data with high-throughput low latency and efficient resource utilization using approximate computing.

Scope. There exist many approximation techniques including sampling, sketches, wavelets, histograms, and on-line aggregation. The main purpose of these techniques is to efficiently summarize the input data using synopses and thereafter process the summary data instead of the original input data. In this thesis, we focus on sampling-based approaches which can be used to approximately answer a wide range of statistical queries on data streams. Sampling-based approaches build a representative subset of the input data obtained using stochastic mechanisms, then process this subset to provide an approximate output result.

Goals. In this thesis, we design and build approximate data analytics systems with three goals in mind:

- **Transparency** Existing applications are able to get the benefit from our systems without any modifications or with a minor change in their code base. To ensure that, we build our systems on top of state-of-the-art big data analytics platforms such as Apache Spark and Apache Flink without changing or with slightly changing the existing interfaces of these systems.
- **Practicality** We offer a simple and convenient interface for approximate computation which allows users systematically make a trade-off between the output quality, throughput/latency or computing resources for data analytics.
- **Efficiency** We aim to handle very large-scale and high-speed data in an efficient and cost-effective manner. To achieve this goal, we design and propose sampling based techniques for approximate computing.

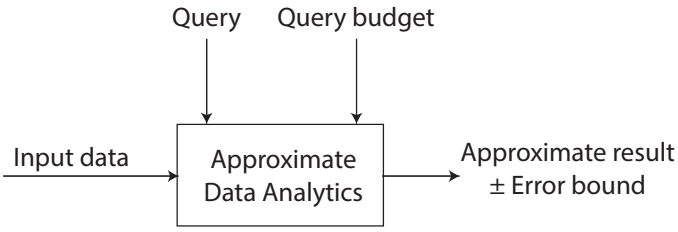

Figure 1.3 – Systems Overview.

1.5 Thesis Contributions

In this thesis, we describe the design and implementation of big data analytics systems that employ approximate computing. Figure 1.3 depicts the high-level architecture of our approximate computing data analytics systems for processing large-scale input datasets. The input data can be in form of batches (data-in-rest) or streams (data-in-motion). We offer the convenience of querying on this input data by supporting a user interface that consists of a query and a query budget for executing the query. A user submits the query as well as specifies a query budget to the system. The query budget can either be in the form of latency/throughput guarantees, desired processing, computing resources available or energy consumption for query processing. Our systems make sure that the computation done over the data remains within the specified budget and the query result is emitted along with a confidence interval or error bounds.

In this thesis, we present the following approximate data analytics systems.

1.5.1 StreamApprox— A Data Analytics System for Approximate Computing

We start this thesis with building a stream processing system called StreamApprox. The aim of this work is to build a system that supports approximate computing for low-latency stream analytics in a transparent manner. Furthermore, the system has the ability to adapt to rapid fluctuations in input data streams. To realize this, we design an online adaptive stratified reservoir sampling algorithm to produce approximate output with bounded error that requires no modifications to the existing applications. Importantly, our proposed algorithm is generic and can be applied to two prominent types of stream processing systems: *(i)* batched stream processing such as Apache Spark Streaming, and *(ii)* pipelined stream processing such as Apache Flink. To showcase the effectiveness of our algorithm, we implemented StreamApprox as a fully functional prototype based on Apache Spark Streaming and Apache Flink. We evaluated StreamApprox using a set of micro-benchmarks and real-world case studies. Our evaluation shows that StreamApprox has the ability to handle high throughput input stream data with fluctuated arriving rates and at the same time provide high accuracy in data analytics.

1.5.2 IncApprox— A Data Analytics System for Approximate and Incremental Computing

After building StreamApprox, we reviewed the design to see whether we can improve the performance of StreamApprox further. We recognized that it is possible to improve the per-

formance by using an additional computing paradigm called incremental computing besides approximate computing. Incremental computing relies on the memoization of intermediate results from previous computations and reuses them in subsequent computations. Since computing over a subset of the input requires less time and resources, this computing paradigm also achieves low-latency and efficient resource utilization as approximate computing. We combine the two computing paradigms by designing an online stratified sampling algorithm that uses self-adjusting computation to produce an incrementally updated approximate output with bounded error. We implemented our algorithm in a system called IncApprox based on Apache Spark Streaming. Our evaluation shows that the two computing paradigms, incremental and approximate computing, are complementary and the combination allows our system to leverage the benefits of both.

1.5.3 PrivApprox— A Data Analytics System for Approximate and Privacy-Preserving Computing

After designing StreamApprox and IncApprox, the systems that allow users to systematically make a trade-off between throughput/latency and accuracy in stream analytics, we wondered whether it is possible to expand the trade-off domain to more than just throughput/latency and accuracy? This question motivated us to build the third system called PrivApprox by expanding the trade-off domain with a new and interesting dimension — privacy. Our objective in this work is to develop a system that supports high-utility and low-latency data analytics while preserving users privacy at the same time. To achieve this goal, we combined privacy-preserving data analytics and approximate computing to build PrivApprox. Privacy-preserving analytics adds explicit noise to the final query output to protect users privacy. In particular, we make use of randomized response mechanisms to preserve the privacy of users and sampling techniques to achieve low-latency in data analytics. Interestingly, by applying these sampling techniques for approximate computing, our system actually strengthens users privacy.

1.5.4 ApproxJoin— A Data Analytics System for Approximate Distributed Joins

In PrivApprox, the system needs to perform a join operation between two input data streams to decrypt the user input stream data. Unfortunately, this join operation is a bottleneck of the system. This motivated us to build the forth system called ApproxJoin to improve the performance of join operations in data analytics using approximate computing. To realize this idea, we designed a novel join sampling algorithm that combines two approximate techniques (*i*) Bloom filters - a sketch technique and (*ii*) stratified sampling. Bloom filters remove unnecessary data items which do not participate in the join operation (non-join items), thus reducing the shuffled data size during the join operation. Meanwhile, stratified sampling allows users to take samples over the join operation to make a trade-off between accuracy and latency. The combination of the two techniques allows users to obtain unbiased random samples over joins. We implemented ApproxJoin in Apache Spark and evaluated it with micro-benchmarks and real-world case-studies. Our evaluation shows that the system not only improves the performance of join operation regarding latency but also significantly reduces total shuffled data size through the network during the join processing.

1.6 Organization

The remainder of the thesis is organized as follows.

In Chapter 2, we present the background about approximate computing techniques and big data analytics frameworks used in this thesis.

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

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

In Chapter 5, we present the design and implementation of PrivApprox.

In Chapter 6, we present the design and implementation of ApproxJoin.

Finally, in Chapter 7, we give the conclusion of our works.

2 Background

In this chapter, we first introduce approximate computing, and then briefly review approximation techniques for data analytics systems including sampling techniques and sketches. Next, we present big data analytics platforms used to implement our systems in this thesis.

2.1 Approximate computing

Approximate computing is a computing paradigm making a trade-off between accuracy and efficiency. Approximate computing is based on the observation that approximate results are sufficient for many applications such as machine learning, social media, and search engines. For example, in machine learning, it is impossible to compute exact models for classification and prediction problems. In video streaming applications, dropping few frames will not affect the quality of output since human perception is limited. In search engines, there is no exact answer for a certain search query, but a list of approximate answers.

Approximate computing has been applied across the whole computing stack including hardware [193], compilers [192], programming languages [29, 194] (refer to [191] for a detailed survey) and data analytics systems [7, 103, 137, 204] to trade accuracy of computations for higher throughput, lower latency, and efficient resource utilization or efficient energy consumption. For example, in K-means clustering computation, sacrificing only 5% classification accuracy loss can save up to $50\times$ energy [164].

Our work mainly builds on the advancements in the databases community. In this section, we survey the approximation techniques in this context. The databases community has proposed various approximation techniques based on sampling [11, 100], online aggregation [121], and sketches [73]. However, these early works in approximate computing were mainly designed for the centralized database architecture. In the context of big data analytics, it was still not clear whether these techniques could be applied or not.

In the last five years, sampling-based approaches have been successfully adopted for distributed data analytics [7, 103, 204]. For example, BlinkDB [7] proposed an approximate distributed query processing engine that uses stratified sampling [11] to support ad-hoc queries with error and response time constraints. First, BlinkDB creates and maintains a collection of multi-dimensional stratified samples using off-line stratified sampling. Second, based on accuracy or response time requirements of users input query, it selects a relevant sample to execute the query. Later, the systems SnappyData [186] and SparkSQL [27] adopted the approximation techniques from BlinkDB to support approximate queries. ApproxHadoop [103] uses multi-stage sampling [152] for approximate MapReduce job execution. These systems show that it is possible to make a trade-off between the output accuracy and performance gains (also efficient resource utilization) by employing sampling-based approaches to compute over a subset of data items. However, these "big data" systems target

batch processing and cannot cater required low-latency guarantees for stream analytics.

Like BlinkDB, Quickr [204] also supports complex ad-hoc queries in big-data clusters. Quickr deploys distributed sampling operators to reduce execution costs of parallelized queries. In particular, Quickr first injects sampling operators into the query plan; thereafter, it searches for an optimal query plan among sampled query plans to execute input queries. However, Quickr is also designed for static databases, and it does not account for stream analytics.

As mentioned before (see §1.3), the key idea behind approximate computing is to execute computations over synopses, which captures the vital characteristics of the original big data, instead of the original data to approximate results with a significantly lower latency. To build the approximate synopses (i.e., compressed representations or summaries) of the big data, there are two major approximate techniques that can be employed: (*i*) sampling and (*ii*) sketches (for other approximate techniques, a detailed survey is provided in [73, 164]).

2.1.1 Sampling

Sampling techniques have become the norm in approximate data analytics. They provide an efficient and cost-effective way to reduce the size of big data meanwhile maintaining vital properties of the original data.

The main objective of sampling mechanisms is to make use of stochastic techniques to take samples representative of the original data. These samples can be used to quickly provide approximate answers for a wide range of queries. The sampling mechanisms allow us to make trade-offs between accuracy and throughput/latency by tuning the sample size of the samples. In fact, online aggregation mechanisms make use of sampling mechanisms with the observation that the accuracy of approximate results can be incrementally improved simply by repeatedly adding more samples.

Random sampling mechanisms are perhaps the most widely used for approximate data analytics including approximate query processing (AQP) [7, 8, 103, 186, 204], graph processing [197], and machine learning [49, 139]. An abundance of methods of building and maintaining samples of big data has been proposed along with multiple mechanisms to provide an estimator for a given query to calculate error-bounds for approximate results. For aggregation queries such as sum, average, and count, unbiased estimators based on Central Limit Theorem [205] can straightforwardly provide error-bounds for approximate results.

For approximate data analytics, there are several key sampling schemes including Bernoulli sampling, simple random sampling, stratified sampling and reservoir sampling.

Bernoulli sampling. Bernoulli sampling is an equal-probability, without-replacement sampling scheme where all data items in the original data have an equal chance of being included in the sample. Since each data item in the original data is considered independently for the sample, the sample size of the sample is not fixed. Implementing Bernoulli sampling is quite straightforward [114]. To perform Bernoulli sampling with sampling fraction of q, we can implement a mechanism as follows. For each data item, we generate a random number in the range [0, 1] (flipping a biased coin), and if the random number is less than q, we add the data item into the sample, otherwise, we just skip it. We can further improve this algorithm by simulating the gaps between subsequent selected items. This is based on the observation that the number of data items that are ignored between successive selections has a geometric

distribution. By simulating the gaps we can save CPU cycles for generating the random number for non-selected data items [114]. The Bernoulli sampling mechanism can be applied easily to take a sample of an input data stream within one pass in distributed environments. However, the problem of Bernoulli is that the sample size is random and not fixed. Thus, it is difficult to estimate the processing latency over the samples. *Simple random sampling* is a sampling sheme overcoming this limitation of Bernoulli sampling.

Simple random sampling. Simple random sampling is a sampling scheme that allows us to take k distinct data items (k is the sample size) from n data items of the original data in such a way that every possible combination of k items is equally likely to be the sample selected (see the definition in [161, 205]). In simple random sampling, each data item has an equal chance to be included in the selected sample. It is considered as the most common sampling scheme for taking a sample of the original data.

In this sampling scheme, simple random sampling *with replacement* allows each data item to appear multiple times in the sample, meanwhile simple random sampling *without replacement* allows each data item appear at most one time in the sample. Given the same sample size, simple random sampling without replacement can capture more vital properties of the original data. This is due to the fact that in simple random sampling with replacement, although the sample may contain duplicated data items, the sample size is still the same. In the case the sample size is very small compared to the population size of the original data, simple random sampling with and without replacement are almost the same since the probability of selecting a given data item into the sample more than once is negligible.

Although simple random sampling is widely used for approximate computing, it may potentially compromise the statistical quality of the sample in the case where the original data contains multiple homogeneous groups (e.g., an input data stream may contain several substreams) with different distributions. This is due to the fact that simple random sampling may overlook some groups having only a few but important data items. Simple random sampling does not ensure that each group in the original data is considered fairly to contribute its data items for the sample. Stratified sampling [11, 205] was proposed to overcome this limitation of simple random sampling.

Stratified sampling. Stratified sampling is a sampling scheme in which the original data is divided into a homogeneous disjoint set of groups (strata); from each group (stratum) a random sample is drawn and these are combined together to build the sample of the original data [205]. Stratified sampling ensures that data items from each group are considered fairly to be present in the sample and no group will be overlooked.

Compared to simple random sampling, stratified sampling provides higher statistical precision and reduces sampling error of the sample. It also means that it requires a smaller sample size to achieve the same accuracy as simple random sampling, thus further improving performance and utilizing less computing resources. Therefore, stratified sampling is considered as a building block for our approximate data analytics systems in this thesis.

There are two types of stratified sampling: (*i*) proportionate stratified sampling and (*ii*) disproportionate stratified sampling. In proportionate stratified sampling, the same sampling fraction is applied for each stratum, i.e., the sample size of each stratum is proportionate to the size of the stratum, whereas in disproportionate stratified sampling, different sampling fractions are applied for different strata. In this thesis, we only consider the proportionate stratified sampling as a building block to design our systems.

Note that, in the case we apply simple random sampling for each stratum, the stratified sampling requires to know the size of the stratum in advance. Hence, it may become impractical in sampling data streams since the population size of a data stream is typically unknown in advance, and thus, sampling fraction cannot be defined beforehand. In addition, the data stream may contain a large number of data items which could not be stored in the system, then the stratified sampling also does not work. *Reservoir sampling* was proposed to overcome these issues.

Reservoir sampling. Reservoir sampling is a sampling scheme that allows us to take and maintain a sample of size k from an input data stream. Reservoir sampling receives data items from the input data stream and maintains a sample in a buffer called *reservoir*. Specifically, to take a sample with size k from the input data stream, Reservoir sampling first fills up the reservoir with the first k data items of the input stream, thereafter it selects a new arriving data item with probability $\frac{k}{i}$ where i is the index of the item in the input stream, then randomly replaces one existing item in the reservoir by the selected item [114]. Reservoir sampling does not require to know the size of the input data stream in advance. In addition, this sampling scheme makes sure that each data item in the input stream is included in the reservoir with equal probability.

Although reservoir sampling is resource-efficient, similar to simple random sampling, it may potentially mutilate the statistical quality of the sampled data items in the reservoir, especially when the input data stream combines multiple sub-streams with different distributions. The data items received from an infrequent substream could easily get overlooked in reservoir sampling. In addition, the scheme is difficult to implement to work in distributed environments since it is required to maintain a global view of the probability $\frac{k}{i}$.

There have been several variants of reservoir sampling scheme proposed. Brown et al. [47] extend the scheme to sample multiple streams in parallel. Al-Kateb et al. [11] combine stratified sampling and reservoir sampling to take samples of a heterogeneous input data stream. However, these mechanisms are not able to neither ensure the statistitical quality of samples nor be implemented in distributed environments.

2.1.2 Sketches

Sketches are techniques to build synopses of big data including batch and stream data. Typically, this technique is suitable to handle high-speed and large-volume data streams [73]. A sketch is a compressed synopsis or a compacted data structure that captures all interesting features and attributes of an input stream. One of the advantages of sketch techniques is that sketches are much smaller compared to the input data stream, thus executing queries on sketches instead of the original data significantly improves the latencies and achieves more efficiency in resource utilization. In general, sketches are built specifically to serve a certain type of query in mind [73], e.g., membership query [41], finding Heavy Hitters in data streams [72] and finding Top-k data items in data streams [156]. The similarity between sketches is that they make use of underlying hashing schemes to summarize the original data [125]. Next, we review several sketches which are widely adopted as technical building blocks for approximate data analytics systems.

Bloom filter. Among sketches, Bloom filter [41] is probably the most well-known proba-

bilistic sketch for supporting membership queries. A Bloom filter is a space-efficient data structure designed to query whether an element is present in a dataset rapidly and memory-efficiently [41]. The Bloom filter data structure is basically an m-bit vector. Each cell in the vector is represented by a bit. Initially, all cells are set to 0. To add a data item to the Bloom filter, we simply hash it h times using h hash functions to obtain bit indices and set the corresponding bits to 1. To check for membership of a data item, we also hash it h times using the same hash functions, and if all bits of its h hash indices are 1, then the data item is in the dataset. For Bloom filters, false positives are possible but false negatives are not. There is a trade-off between the size of a bit vector m and the probability of a false positive. A larger m has less false positives but consumes more memory, whenever a smaller m requires less memory at the risk of more false positives. The false positive rate can be computed as follows [41]:

$$p \approx (1 - e^{\frac{hn}{m}})^h \tag{2.1}$$

In this thesis, we also consider Bloom filter as a technical building block to design our approximate data analytics systems since it provides three main advantages: (*i*) querying the membership of data items is efficient (O(h) complexity); (*ii*) the size of a Bloom filter is linearly correlated with the size of the input data, but requires a significantly smaller storage compared to the original data; (*iii*) building a Bloom filter requires only a single pass over the input data.

Count-Min Sketch. Count-Min sketch is a probabilistic data structure that is used to estimate the frequencies of data items in an input data stream. It extends the design of Bloom filter with a bit array structure into a table counter (a two-dimensional array) in order to maintain approximate frequency counts of input data stream items [71]. Count-Min sketch uses a set of hash functions to map input data items into the table. The table is organized as a sequence of rows, each data item is mapped into the first row using the first hash function, into the second row using the second hash function, and so on. Each row maintains a counter that is incremented on every occurrence of the data item. This counter could also potentially be incremented by other items mapped to the same location as collisions are expected to happen. To estimate the frequencies of a data item, Count-Min sketch simply takes the smallest counter among rows of the item since the counters contain the desired count plus noise due to collisions [71, 73].

HyperLogLog. HypperLogLog is a probabilistic data structure that is used to estimate the number of distinct data items in a dataset or an input data stream. HyperLogLog uses a hash function to map each data item in the original data to a hash value and assumes that the hash function produces uniform hash values. If the binary representation of hash values has l maximum number of leading zeros at the beginning (00..0 1* - l zeros), then the number of distinct data items in the original data is estimated to 2^l [92, 122]. This algorithm is based on the observation that the number of distinct data items (cardinality) of a list of *uniformly distributed random numbers* can be estimated by measuring the maximum number of leading zeros in the bit-string representation of each number in that list. If the maximum number of zeros obtained among bit-strings is l, the number of distinct items in the list can be estimated as 2^l [92, 122]. Refer to [122] for deep discussion on how to use HyperLogLog in practice.

In general, sketches can be used to build synopses or summaries of data streams to capture interesting features for answering queries. One of the limitations of sketching techniques

compared to sampling techniques is that each sketch is designed to support a specific type of query. For example, Bloom Filter is designed for membership queries, Count-Min sketch for frequency queries, HyperLogLog for count distinct queries, and MinHash sketch [67] for measuring the similarity between two sets. Meanwhile, sampling techniques can support more general queries. Thus, in this thesis, we focus on using sampling techniques to design approximate analytics systems. However, we also try to combine sampling techniques and sketching techniques to support a wide-range of queries and provide better approximations for those queries.

2.2 Big data analytics frameworks

In this section, we provide a brief description of big data frameworks which we used to implement our approximate data analytics systems including Apache Spark [21] and Apache Flink [17]. These two frameworks have been widely used in both academia and industry for big data analytics.

2.2.1 Apache Spark

Apache Spark [21] is an open-source large-scale data processing framework. Comparing to Hadoop MapReduce, Spark is much faster in processing large-scale datasets since instead of using the disk-based cluster computing, Spark applies the in-memory computing concept where intermediate data is cached in memory to reduce latency [133].

Spark introduces the core abstraction – Resilient Distributed Datasets (RDDs) [229] for distributed data-parallel computing. An RDD is an immutable and fault-tolerant collection of elements (objects) that is distributed or partitioned across a set of nodes in a cluster [21]. To achieve fault tolerance, Spark provides the *lineage* mechanism in which each RDD maintains *meta-data* how it was created from other RDDs. The *meta-data* is used to recover the RDD in the case of failure [133, 229].

RDDs support two types of operations: *t*ransformations and *a*ctions. Transformations return a new RDD, such as *map()* and *filter()*, and actions are operations performed on RDDs that return the output to the driver program or store it in a persistent storage system (e.g *reduce()*, *count()*, *collect()*, *saveAsTextFile()*, etc). Transformations are performed lazily, i.e., they are computed whenever an action operation is invoked [133].

Spark platform contains multiple integrated components including MLlib [22] for machine learning, GraphX [105] for graph-based analytics, Spark SQL [27] for working with structured data, and Spark Streaming [23] for processing data streams. Spark Streaming extends the RDD abstraction by introducing the DStreams APIs [227], which is a sequence of RDDs arrived during a time window. Spark Streaming splits the input data stream into micro-batches, and for each micro-batch, a distributed data-parallel job (Spark job) is launched to process the micro-batch.

Spark supports RDD based sampling functions, which in turn can be used to support approximate computing [229]. In particular, Spark's sampling functions can be classified into the following two categories: 1) Simple Random Sampling (SRS) using `sample()`, and 2) Stratified Sampling (STS) using `sampleByKey()` and `sampleByKeyExact()`. Spark implements

these sampling functions in a "batch" fashion, where all data items are first accumulated in a batch, and then the actual sampling is carried out.

2.2.2 Apache Flink

Apache Flink [17] is an open-source framework for both distributed batch and stream data analytics. Apache Flink and Spark Streaming both are DAG-based distributed stream processing engines. At a high level, both frameworks provide similar dataflow operators (e.g., map, flatmap, reduce, and filter). However, Spark Streaming is a batched stream processing engine, whereas Flink is a pipelined stream processing engine. In particular, Apache Flink adopts a pipelined architecture in which whenever an operator in the DAG dataflow emits an item, this item is *immediately* forwarded to the next operator without waiting for a whole data batch. This mechanism allows Apache Flink to support truely stream processing. In addition, Flink considers batches as a special case of streaming, whereas Spark considers streaming as a special case of batches.

Similar to Spark, Apache Flink also provides a fault-tolerance scheme to consistently recover the state of data stream processing jobs in the case of failures. This scheme takes consistent snapshots of the input data stream and state of operators. In the case of failures, the snapshots are used for recovery. Apache Flink supports both stream and batch processing by offering two main abstractions: *(i) DataStream* for stream processing and *(ii) DataSet* for batch processing.

In addition, Flink also contains multiple integrated components including Flink Gelly [93] for graph processing, FlinkML [95] for machine learning, and Flink SQL [94] for supporting SQL queries for both stream and batch data analytics.

3 StreamApprox: Approximate Stream Analytics

In this chapter, we present the design, implementation, and evaluation of StreamApprox, an approximate computing system for stream analytics. StreamApprox supports high-throughput and low-latency stream analytics by adopting approximate computing in a transparent manner. In StreamApprox, we designed an online stratified reservoir sampling algorithm to produce approximate output with rigorous error bounds.

This chapter is organized as follows. We first motivate the design of StreamApprox in Section §3.1. We next briefly highlight the contributions of StreamApprox in Section §3.2. Thereafter, we present the overview of StreamApprox in Section §3.3. Next, we describe the detailed design of StreamApprox in Section §3.4. Then, we present the implementation of StreamApprox based on the design in Section §3.5. Next, we present an experimental evaluation of StreamApprox using micro-benchmarks in Section §3.6 and two real world case-studies in Section §3.7. Thereafter, we present the limitations in Section §3.8. Finally, the related work and conclusion are discussed in Section §3.9 and Section §3.10, respectively.

The content of this chapter is based on our conference paper [184] and technical report [181]. This work is based on a joint collaboration with Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe.

3.1 Motivation

Stream analytics systems are extensively used in the context of modern online services to transform continuously arriving raw data streams into useful insights [17, 168, 227]. These systems target low-latency execution environments with strict service-level agreements (SLAs) for processing the input data streams.

In the current deployments, the low-latency requirement is usually achieved by employing more computing resources and parallelizing the application logic over the distributed infrastructure. Since most stream processing systems adopt a data-parallel programming model [75], almost linear scalability can be achieved with increased computing resources.

However, this scalability comes at the cost of ineffective utilization of computing resources and reduced throughput of the system. Moreover, in some cases, processing the entire input data stream would require more than the available computing resources to meet the desired latency/throughput guarantees.

To strike a balance between the two desirable, but contradictory design requirements — low latency and efficient utilization of computing resources — there is a surge of *approximate computing* paradigm that explores a novel design point to resolve this tension. In particular, approximate computing is based on the observation that many data analytics jobs are amenable to an approximate rather than the exact output [80, 170]. For such workflows, it is possible to trade the output accuracy by computing over a subset instead of the entire data stream. Since

computing over a subset of input requires less time and computing resources, approximate computing can achieve desirable latency and computing resource utilization.

To design an approximate computing system for stream analytics, we need to address the following three important design challenges: Firstly, we need an online sampling algorithm that can perform "on-the-fly" sampling on the input data stream. Secondly, since the input data stream usually consists of sub-streams carrying data items with disparate population distributions, we need the online sampling algorithm to have a "stratification" support to ensure that all sub-streams (strata) are considered fairly, i.e., the final sample has a representative sub-sample from each distinct sub-stream (stratum). Finally, we need an error-estimation mechanism to interpret the output (in)accuracy using an error bound or confidence interval.

Unfortunately, the advancements in approximate computing are primarily geared towards batch analytics [7, 103, 204], where the input data remains unchanged during the course of sampling (see §3.9 for details). In particular, these systems rely on pre-computing a set of samples on the static database, and take an appropriate sample for the query execution based on the user's requirements (aka query execution budget). Therefore, the state-of-the-art systems cannot be deployed in the context of stream processing, where the new data continuously arrives as an unbounded stream.

As an alternative, we could in principle *repurpose* the available sampling mechanisms in Apache Spark (primarily available for machine learning in the MLib library [22]) to build an approximate computing system for stream analytics. In fact, as a starting point, we designed and implemented an approximate computing system for stream processing in Apache Spark based on the available sampling mechanisms.

Unfortunately, as we will show later, Spark's stratified sampling algorithm suffers from three key limitations for approximate computing, which we address in our work (see §3.5 for details). First, Spark's stratified sampling algorithm operates in a "batch" fashion, i.e., all data items are first collected in a batch as Resilient Distributed Datasets (RDDs) [229], and thereafter, the actual sampling is carried out on the RDDs. Second, it does not handle the case where the arrival rate of sub-streams changes over time because it requires a pre-defined sampling fraction for each stratum. Lastly, the stratified sampling algorithm implemented in Spark requires synchronization among workers for the expensive join operation, which imposes a significant latency overhead.

3.2 Contribution

In this chapter, we present the design and implementation of StreamApprox, an approximate computing system for stream analytics. In this system, we designed an *online stratified reservoir sampling algorithm* for stream analytics. Unlike existing Spark-based systems, we perform the sampling process "on-the-fly" to reduce the latency as well as the overheads associated with the process of forming RDDs. Importantly, our algorithm *generalizes to two prominent types of stream processing models*: (1) batched stream processing employed by Apache Spark Streaming [23], and (2) pipelined stream processing employed by Apache Flink [17].

More specifically, our sampling algorithm makes use of two techniques: reservoir sampling and stratified sampling. We perform reservoir sampling for each sub-stream by creating a

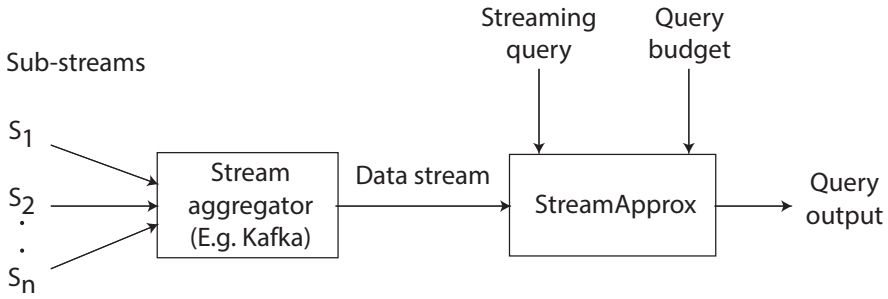

Figure 3.1 – System overview.

fixed-size reservoir per stratum. Thereafter, we assign weights to all strata respecting their respective arrival rates to preserve the statistical quality of the original data stream. The proposed sampling algorithm naturally adapts to varying arrival rates of sub-streams, and requires no synchronization among workers (see §3.4).

We implemented StreamApprox based on Apache Spark Streaming [23] and Apache Flink [17], and evaluate its effectiveness via various micro-benchmarks. Furthermore, we also report our experiences on applying StreamApprox to two real-world case studies. Our evaluation shows that Spark- and Flink-based StreamApprox achieves a significant speedup of $1.15\times$ to $3\times$ over the native Spark Streaming and Flink executions, with varying sampling fraction of 80% to 10%, respectively.

In addition, for a fair comparison, we have also implemented an approximate computing system leveraging the sampling modules already available in Apache Spark's MLib library (in addition to the native execution comparison). Our evaluation shows that, for the same accuracy level, the throughput of Spark-based StreamApprox is roughly $1.1\times$—$2.4\times$ higher than the Spark-based approximate computing system for stream analytics.

To summarize, we make the following main contributions.

- We propose the online adaptive stratified reservoir sampling (OASRS) algorithm that preserves the statistical quality of the input data stream, and is resistant to the fluctuation in the arrival rates of strata. Our proposed algorithm is generic and can be applied to the two prominent stream processing models: batched and pipelined stream processing models.
- We extend our algorithm for distributed execution. The OASRS algorithm can be parallelized naturally without requiring any form of synchronization among distributed workers.
- We provide a confidence metric on the output accuracy using an error bound or confidence interval. This gives a measure of accuracy trade-off on the result due to the approximation.
- Finally, we have implemented the proposed algorithm and mechanisms based on Apache Spark Streaming and Apache Flink. We have extensively evaluated the system using a series of micro-benchmarks and real-world case studies.

3.3 Overview

This section gives an overview of StreamApprox, its computational model, and the design assumptions. Lastly, we conclude this section with a brief background on the technical building blocks.

3.3.1 System Overview

StreamApprox is designed for real-time stream analytics. Figure 3.1 presents the high-level architecture of StreamApprox. The input data stream usually consists of data items arriving from diverse sources. The data items from each source form a *sub-stream*. We make use of a stream aggregator (e.g., Apache Kafka [130]) to combine the incoming data items from disjoint sub-streams. StreamApprox then takes this combined stream as the input for data analytics.

We facilitate data analytics on the input stream by providing an interface for users to specify the streaming query and its corresponding query budget. The query budget can be in the form of expected latency/throughput guarantees, available computing resources, or the accuracy level of query results.

StreamApprox ensures that the input stream is processed within the specified query budget. To achieve this goal, we make use of approximate computing by processing only a subset of data items from the input stream, and produce an approximate output with rigorous error bounds. In particular, StreamApprox designs a parallelizable online sampling technique to select and process a subset of data items, where the sample size can be determined based on the query budget.

3.3.2 Computational Model

The state-of-the-art distributed stream processing systems can be classified in two prominent categories: (*i*) batched stream processing model, and (*ii*) pipelined stream processing model. These systems offer three main advantages: (a) efficient fault tolerance, (b) "exactly-once" semantics, and (c) unified programming model for both batch and stream analytics. *Our proposed algorithm for approximate computing is generalizable to both stream processing models, and preserves their advantages.*

Batched stream processing model. In this computational model, an input data stream is divided into small batches using a predefined batch interval, and each such batch is processed via a distributed data-parallel job. Apache Spark Streaming [23] adopted this model to process input data streams. While this model is widely used for many applications, it cannot adapt to the cases where low-latency is critical since this model waits for the batching to complete before processing the batch. Sampling the input data stream in a continuous "on-the-fly" fashion can be challenging to address in this computational model. However, StreamApprox overcomes this challenge by performing sampling operations before the batches are formed.

Pipelined stream processing model. In contrast to the batched stream processing model, the pipelined model streams each data item to the next operator as soon as the item is ready to be processed without forming the whole batch. Thus, this model achieves low latency. Apache Flink [17] implements the pipelined stream processing model to support a truly native stream

processing engine. StreamApprox can adopt this computational model easily by sampling the input data stream in an online manner.

Note that both stream processing models support the sliding window computation [38]. The processing window slides over the input stream, whereby the newly incoming data items are added to the window and the old data items are removed from the window. The number of data items within a sliding window may vary in accordance to the arrival rate of data items.

3.3.3 Design Assumptions

StreamApprox is based on the following assumptions. We discuss the possible means to reduce these assumptions in §3.8.

1. We assume there exists a virtual cost function which translates a given query budget (such as the expected latency or throughput guarantees, or the required accuracy level of query results) into the appropriate sample size.
2. We assume that the input stream is stratified based on the source of data items, i.e., the data items from each sub-stream follow the same distribution and are mutually independent. Here, a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.
3. We assume a time-based window length. Based on the arrival rate, the number of data items within a window may vary accordingly. Note that this assumption is consistent with the sliding window APIs in the aforementioned stream processing systems.

3.3.4 Background: Technical Building Blocks

We next describe the two main technical building blocks of StreamApprox: (a) reservoir sampling, and (b) stratified sampling.

Reservoir sampling. Suppose we have a stream of data items, and want to randomly select a sample of N items from the stream. If we know the total number of items in the stream, then the solution is straightforward by applying the simple random sampling [152]. However, if a stream consists of an unknown number of items or the stream contains a large number of items which could not fit into the storage, then the simple random sampling does not work and a sampling technique called *reservoir sampling* can be used [12, 213].

Reservoir sampling receives data items from a stream, and maintains a sample in a buffer called *reservoir*. Specifically, the technique populates the reservoir with the first N items received from the stream. After the first N items, every time we receive the i-th item ($i > N$), we replace each of the N existing items in the reservoir with the probability of $1/i$, respectively. In other words, we accept the i-th item with the probability of N/i, and then randomly replace one existing item in the reservoir. In doing so, we do not need to know the total number of items in the stream, and reservoir sampling ensures that each item in the stream has equal probability of being selected for the reservoir. Reservoir sampling is resource-friendly, and its pseudo-code can be found in Algorithm 1.

Stratified sampling. Although reservoir sampling is widely used in stream processing, it could potentially mutilate the statistical quality of the sampled data in the case where the

Algorithm 1: Reservoir algorithm

Input: $N \leftarrow$ *sample size*

```
1  begin
2      reservoir ← ∅; // Set of items sampled from the input stream
3      foreach arriving item x_i do
4          if |reservoir| < N then
5              // Fill up the reservoir
6              reservoir.append(x_i);
7          else
8              p ← N/i;
9              // Flip a coin comes heads with probability p
10             head ← flipCoin(p);
11             if head then
12                 // Get a random index in the reservoir
13                 j ← getRandomIndex(0, |reservoir| − 1);
14                 // Replace old item in reservoir by x_i
15                 reservoir[j] ← x_i
```

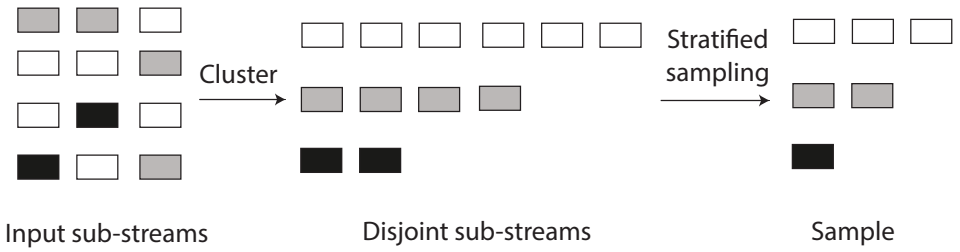

Figure 3.2 – Stratified sampling with the sampling fraction of 50%.

input data stream contains multiple sub-streams with different distributions. This is because reservoir sampling may overlook some sub-streams consisting of only a few data items. In particular, reservoir sampling does not guarantee that each sub-stream is considered fairly to have its data items selected for the sample. *Stratified sampling* [11, 152] was proposed to cope with this problem. Stratified sampling first clusters the input data stream into disjoint sub-streams, and then performs the sampling (e.g., simple random sampling) over each sub-stream independently, as illustrated in Figure 3.2. Stratified sampling guarantees that data items from every sub-stream can be fairly selected and no sub-stream will be overlooked. Stratified sampling, however, works only in the scenario where it knows the statistics of all sub-streams in advance (e.g., the length of each sub-stream).

3.4 Design

In this section, we first present the StreamApprox's workflow (§3.4.1). Then, we detail its sampling mechanism (§3.4.2), and its error estimation mechanism (§3.4.3).

Algorithm 2: : StreamApprox's algorithm overview

Input: streaming *query* and query *budget*

```
1  begin
2      // Computation in sliding window model (§3.3.2)
3      foreach time interval do
4          // Cost function gives the sample size based on the budget (§3.8)
5          sampleSize ← costFunction(budget);
6          forall arriving items in the time interval do
7              // Perform OASRS Sampling (§3.4.2)
8              // W denotes the weights of the sample
9              sample, W ← OASRS(items, sampleSize);
10         // Run query as a data-parallel job to process the sample
11         output ← runJob(query, sample, W);
12         // Estimate the error bounds of query result/output (§3.4.3)
13         output ± error ← estimateError(output);
```

3.4.1 System Workflow

Algorithm 2 presents the workflow of StreamApprox. The algorithm takes the user-specified streaming *query* and the query *budget* as the input. The algorithm executes the query on the input data stream as a sliding window computation (see §3.3.2).

For each time interval, we first derive the sample size (*sampleSize*) using a cost function based on the given query budget (see §3.8). As described in §3.3.3, we currently assume that there exists a cost function which translates a given query budget (such as the expected latency or throughput guarantees, or the required accuracy level of query results) into the appropriate sample size. We discuss the possible means to implement such a cost function in §3.8.

We next propose a sampling algorithm (detailed in §3.4.2) to select the appropriate $sample$ in an online fashion. Our sampling algorithm further ensures that data items from all sub-streams are fairly selected for the sample, and no single sub-stream is overlooked.

Thereafter, we execute a data-parallel job to process the user-defined *query* on the selected sample. As the last step, we run an error estimation mechanism (as described in §3.4.3) to compute the error bounds for the approximate query result in the form of $output \pm error$ bound.

The whole process repeats for each time interval as the computation window slides [39]. Note that, the query budget can change across time intervals to adapt to user's requirements for the query budget.

3.4.2 Online Adaptive Stratified Reservoir Sampling

To realize the real-time stream analytics, we propose a novel sampling technique called Online Adaptive Stratified Reservoir Sampling (OASRS). It achieves both stratified and reservoir samplings without their drawbacks. Specifically, OASRS does not overlook any sub-streams regardless of their popularity, does not need to know the statistics of sub-streams before the

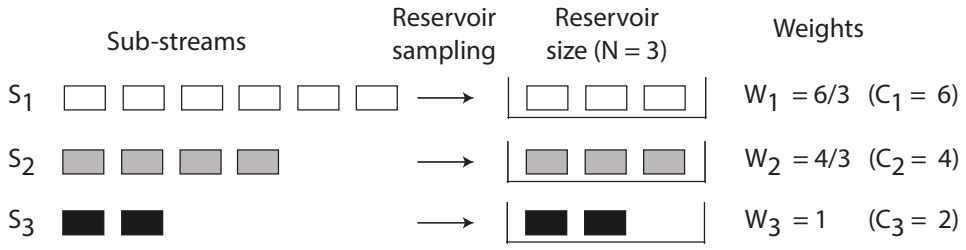

Figure 3.3 – OASRS with the reservoirs of size three.

sampling process, and runs efficiently in real time in a distributed manner.

The high-level idea of OASRS is simple, as described in Algorithm 3. We stratify the input stream into sub-streams according to their sources. We assume data items from each sub-stream follow the same distribution and are mutually independent. (Here a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.) We then sample each sub-stream independently, and perform the reservoir sampling for each sub-stream individually. To do so, every time we encounter a new sub-stream S_i, we determine its sample size N_i according to an adaptive cost function considering the specified query budget (see §3.8). For each sub-stream S_i, we perform the traditional reservoir sampling to select items at random from this sub-stream, and ensure that the total number of selected items from S_i does not exceed its sample size N_i. In addition, we maintain a counter C_i to measure the number of items received from S_i within the concerned time interval (see Figure 3.3).

Applying reservoir sampling to each sub-stream S_i ensures that we can randomly select at most N_i items from each sub-stream. The selected items from different sub-streams, however, should *not* be treated equally. In particular, for a sub-stream S_i, if $C_i > N_i$ (i.e., the sub-stream S_i has more than N_i items in total during the concerned time interval), then we randomly select N_i items from this sub-stream and each selected item represents C_i/N_i original items on average; otherwise, if $C_i \leq N_i$, we select all the received C_i items so that each selected item only represents itself. As a result, in order to statistically recreate the original items from the selected items, we assign a specific weight W_i to the items selected from each sub-stream S_i:

$$W_i = \begin{cases} C_i/N_i & \text{if } C_i > N_i \\ 1 & \text{if } C_i \leq N_i \end{cases} \tag{3.1}$$

We support *approximate linear queries* which return an approximate weighted sum of all items received from all sub-streams. One example of linear queries is to compute the sum of all received items. Suppose there are in total X sub-streams $\{S_i\}_{i=1}^{X}$, and from each sub-stream S_i we randomly select at most N_i items. Specifically, we select Y_i items $\{I_{i,j}\}_{j=1}^{Y_i}$ from each sub-stream S_i, where $Y_i \leq N_i$. In addition, each sub-stream associates with a weight W_i generated according to expression 3.1. Then, the approximate sum SUM_i of all items received from each sub-stream S_i can be estimated as:

$$SUM_i = (\sum_{j=1}^{Y_i} I_{i,j}) \times W_i \tag{3.2}$$

As a result, the approximate total sum of all items received from all sub-streams is:

$$SUM = \sum_{i=1}^{X} SUM_i \tag{3.3}$$

A simple extension also enables us to compute the approximate mean value of all received items:

$$MEAN = \frac{SUM}{\sum_{i=1}^{X} C_i} \tag{3.4}$$

Here, C_i denotes a counter measuring the number of items received from each sub-stream S_i. Using a similar technique, our OASRS sampling algorithm supports any types of approximate linear queries. This type of queries covers a range of common aggregation queries including, for instance, sum, average, count, histogram, etc. Though linear queries are simple, they can be extended to support a large range of statistical learning algorithms [42, 43]. It is also worth mentioning that, OASRS not only works for a concerned time interval (e.g., a sliding time window), but also works across the entire life cycle of the input data stream.

To summarize, our proposed sampling algorithm combines the benefits of stratified and reservoir samplings via performing the reservoir sampling for each sub-stream (i.e., stratum) individually. In addition, our algorithm is an online algorithm since it can perform the "on-the-fly" sampling on the input stream without knowing all data items in a window from the beginning [13].

Distributed execution. OASRS can run in a distributed fashion naturally as it does not require synchronization. One straightforward approach is to make each sub-stream S_i be handled by a set of w worker nodes. Each worker node samples an equal portion of items from this sub-stream and generates a local reservoir of size no larger than N_i/w. In addition, each worker node maintains a local counter to measure the number of its received items within a concerned time interval for weight calculation. The rest of the design remains the same.

3.4.3 Error Estimation

We described how we apply OASRS to randomly sample the input data stream to generate the approximate results for linear queries. We now describe a method to estimate the accuracy of our approximate results via rigorous error bounds.

Similar to §3.4.2, suppose the input data stream contains X sub-streams $\{S_i\}_{i=1}^{X}$. We compute the approximate sum of all items received from all sub-streams by randomly sampling only Y_i items from each sub-stream S_i. As each sub-stream is sampled independently, the variance of the approximate sum is:

$$Var(SUM) = \sum_{i=1}^{X} Var(SUM_i) \tag{3.5}$$

Further, as items are randomly selected for a sample within each sub-stream, according to the random sampling theory [205], the variance of the approximate sum can be estimated as:

$$\widehat{Var}(SUM) = \sum_{i=1}^{X} \left(C_i \times (C_i - Y_i) \times \frac{s_i^2}{Y_i} \right) \tag{3.6}$$

Algorithm 3: : Online adaptive stratified reservoir sampling

```
1  OASRS(items, sampleSize)
2  begin
3      sample ← ∅; // Set of items sampled within the time interval
4      S ← ∅; // Set of sub-streams seen so far within the time interval
5      W ← ∅; // Set of weights of sub-streams within the time interval
6      Update(S); // Update the set of sub-streams
7      // Determine the sample size for each sub-stream
8      N ← getSampleSize(sampleSize, S);
9      forall S_i in S do
10         C_i ← 0; // Initial counter to measure #items in each sub-stream
11         forall arriving items in each time interval do
12             Update(C_i); // Update the counter
13             sample_i ← RS(items, N_i); // Reservoir sampling
14             sample.add(sample_i); // Update the global sample
15             // Compute the weight of sample_i according to Equation 3.1
16             if C_i > N_i then
17                 W_i ← C_i / N_i;
18             else
19                 W_i ← 1;
20             W.add(W_i); // Update the set of weights
21     return sample, W
```

Here, C_i denotes the total number of items from the sub-stream S_i, and s_i denotes the standard deviation of the sub-stream S_i's sampled items:

$$s_i^2 = \frac{1}{Y_i - 1} \times \sum_{j=1}^{Y_i} (I_{i,j} - \bar{I}_i)^2, \text{where } \bar{I}_i = \frac{1}{Y_i} \times \sum_{j=1}^{Y_i} I_{i,j} \tag{3.7}$$

Next, we show how we can also estimate the variance of the approximate mean value of all items received from all the X sub-streams. According to Equation 3.4, this approximate mean value can be computed as:

$$\begin{aligned} MEAN &= \frac{SUM}{\sum_{i=1}^{X} C_i} = \frac{\sum_{i=1}^{X} (C_i \times MEAN_i)}{\sum_{i=1}^{X} C_i} \\ &= \sum_{i=1}^{X} (\omega_i \times MEAN_i) \end{aligned} \tag{3.8}$$

Here, $\omega_i = \frac{C_i}{\sum_{i=1}^{X} C_i}$. Then, as each sub-stream is sampled independently, according to the random sampling theory [205], the variance of the approximate mean value can be estimated

as:

$$\begin{aligned}\widehat{Var}(MEAN) &= \sum_{i=1}^{X} Var(\omega_i \times MEAN_i) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times Var(MEAN_i)\right) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times \frac{s_i^2}{Y_i} \times \frac{C_i - Y_i}{C_i}\right)\end{aligned} \tag{3.9}$$

Above, we have shown how to estimate the variances of the approximate sum and the approximate mean of the input data stream. Similarly, by applying the random sampling theory, we can easily estimate the variance of the approximate results of any linear queries.

Error bound. According to the "68-95-99.7" rule [1], our approximate result falls within one, two, and three standard deviations away from the true result with probabilities of 68%, 95%, and 99.7%, respectively, where the standard deviation is the square root of the variance as computed above. This error estimation is critical because it gives us a quantitative understanding of the accuracy of our sampling technique.

3.5 Implementation

To showcase the effectiveness of our algorithm, we provide two implementations of StreamApprox based on two types of stream processing systems (§3.3.2): (*i*) Apache Spark Streaming [23] — a batched stream processing system, and (*ii*) Apache Flink [17] — a pipelined stream processing system.

Furthermore, we also built an improved baseline (in addition to the native execution) for Apache Spark, which provides sampling mechanisms for its machine learning library MLib [22]. In particular, we *repurposed* the existing sampling modules available in Apache Spark (primarily used for machine learning) to build an approximate computing system for stream analytics. To have a fair comparison, we evaluate our Spark-based StreamApprox with two baselines: the Spark native execution and the improved Spark sampling based approximate computing system. Meanwhile, Apache Flink does not support sampling operations for stream analytics, therefore we compare our Flink-based StreamApprox with the Flink native execution.

In this section, we first present the necessary background on Apache Spark Streaming (and its existing sampling mechanisms) and Apache Flink (§3.5.1). Thereafter, we provide the implementation details of our prototypes (§3.5.2).

3.5.1 Background

Apache Spark Streaming and Apache Flink both are DAG-based distributed data processing engines. At a high level, both frameworks provide similar dataflow operators (e.g., map, flatmap, reduce, and filter). However, as described in §3.3.2, at the core, Spark Streaming is a batched stream processing engine, whereas Flink is a pipelined stream processing engine.

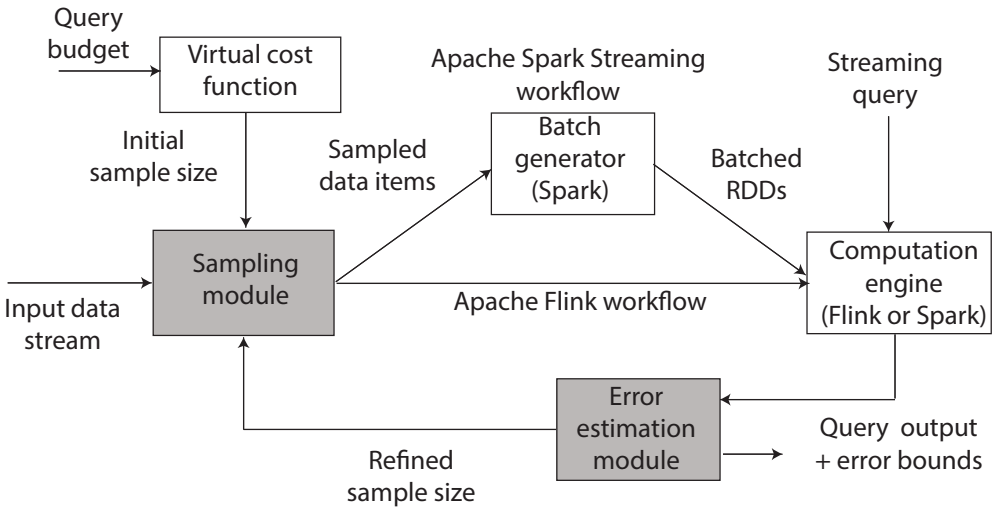

Figure 3.4 – Architecture of StreamApprox prototypes (shaded boxes depict the implemented modules). We have implemented our system based on Apache Spark Streaming and Apache Flink.

Apache Spark Streaming

Apache Spark Streaming splits the input data stream into micro-batches, and for each micro-batch a distributed data-parallel job (Spark job) is launched to process the micro-batch. To sample the input data stream, Spark Streaming makes use of RDD-based sampling functions supported by Apache Spark [229] to take a sample from each micro-batch. These functions can be classified into the following two categories: 1) Simple Random Sampling (SRS) using `sample`, and 2) Stratified Sampling (STS) using `sampleByKey` and `sampleByKeyExact`.

Simple random sampling (SRS) is implemented using a random sort mechanism [161] which selects a sample of size k from the input data items in two steps. In the first step, Spark assigns a random number in the range of $[0, 1]$ to each input data item to produce a key-value pair. Thereafter, in the next step, Spark sorts all key-value pairs based on their assigned random numbers, and selects k data items with the smallest assigned random numbers. Since sorting "Big Data" is expensive, the second step quickly becomes a bottleneck in this sampling algorithm. To mitigate this bottleneck, Spark reduces the number of items before sorting by setting two thresholds, p and q, for the assigned random numbers. In particular, Spark discards the data items with the assigned random numbers larger than q, and directly selects data items with the assigned numbers smaller than p. For stratified sampling (STS), Spark first clusters the input data items based on a given criterion (e.g., data sources) to create strata using `groupBy(strata)`. Thereafter, it applies the aforementioned SRS to data items in each stratum.

Apache Flink

In contrast to batched stream processing, Apache Flink adopts a pipelined architecture: whenever an operator in the DAG dataflow emits an item, this item is *immediately* forwarded to the next operator without waiting for a whole data batch. This mechanism makes Apache

Flink a true stream processing engine. In addition, Flink considers batches as a special case of streaming. Unfortunately, the vanilla Flink does not provide any operations to take a sample of the input data stream. In this work, we provide Flink with an operator to sample input data streams by implementing our proposed sampling algorithm (see §3.4.2).

3.5.2 StreamApprox Implementation Details

We next describe the implementation of StreamApprox. Figure 3.4 illustrates the architecture of our prototypes, where the shaded boxes depict the implemented modules. We showcase workflows for Apache Spark Streaming and Apache Flink in the same figure.

Spark-based StreamApprox

In the Spark-based implementation, the input data items are sampled "on-the-fly" using our sampling module *before* items are transformed into RDDs. The sampling parameters are determined based on the query budget using a virtual cost function. In particular, a user can specify the query budget in the form of desired latency or throughput, available computational resources, or acceptable accuracy loss. As noted in the design assumptions (§3.3.3), we have not implemented the virtual cost function since it is beyond the scope of this chapter (see §3.8 for possible ways to implement such a cost function). Based on the query budget, the virtual cost function determines a sample size, which is then fed to the sampling module.

Thereafter, the sampled input stream is transformed into RDDs, where the data items are split into batches at a pre-defined regular batch interval. Next, the batches are processed as usual using the Spark engine to produce the query output. Since the computed output is an approximate query result, we make use of our error estimation module to give rigorous error bounds. For cases where the error bound is larger than the specified target, an adaptive feedback mechanism is activated to increase the sample size in the sampling module. This way, we achieve higher accuracy in the subsequent epochs.

I: Sampling module. The sampling module implements the algorithm described in §3.4.2 to select samples from the input data stream in an online adaptive fashion. We modified the Apache Kafka connector of Spark to support our sampling algorithm. In particular, we created a new class `ApproxKafkaRDD` to handle the input data items from Kafka, which takes required samples to define an RDD for the data items before calling the `compute` function.

II: Error estimation module. The error estimation module computes the error bounds of the approximate query result. The module also activates a feedback mechanism to re-tune the sample size in the sampling module to achieve the specified accuracy target. We made use of the Apache Common Math library [158] to implement the error estimation mechanism as described in §3.4.3.

Flink-based StreamApprox

Compared to the Spark-based implementation, implementing a Flink-based StreamApprox is straightforward since Flink supports online stream processing natively.

I: Sampling module. We created a sampling operator by implementing the algorithm described in §3.4.2. This operator samples input data items on-the-fly and in an adaptive manner. The sampling parameters are identified based on the query budget as in Spark-based StreamApprox.

II: Error estimation module. We reused the error estimation module implemented in the Spark-based StreamApprox.

3.6 Evaluation

In this section, we present the evaluation results of our implementation. In the next section, we report our experiences on deploying StreamApprox for real-world case studies (§3.7).

3.6.1 Experimental Setup

Synthetic data stream. To understand the effectiveness of our proposed OASRS sampling algorithm, we evaluated StreamApprox using a synthetic input data stream with Gaussian distribution and Poisson distribution. For the Gaussian distribution, unless specified otherwise, we used three input sub-streams A, B, and C with their data items following Gaussian distributions of parameters ($\mu = 10,\ \sigma = 5$), ($\mu = 1000,\ \sigma = 50$), and ($\mu = 10000,\ \sigma = 500$), respectively. For the Poisson distribution, unless specified otherwise, we used three input sub-streams A, B, and C with their data items following Poisson distributions of parameters ($\lambda = 10$), ($\lambda = 1000$), and ($\lambda = 100000000$), respectively.

Methodology for comparison with Apache Spark. For a fair comparison with the sampling algorithms available in Apache Spark, we also built an Apache Spark-based approximate computing system for stream analytics (as described in §3.5). In particular, we used two sampling algorithms available in Spark, namely, Simple Random Sampling (SRS) via `sample`, and Stratified Sampling (STS) via `sampleByKey` and `sampleByKeyExact`. We applied these sampling operators to each small batch (i.e., RDD) in the input data stream to generate samples. Note that, the Apache Flink does not support sampling natively.

Evaluation questions. Our evaluation analyzes the performance of StreamApprox, and compares it with the Spark-based approximate computing system across the following dimensions: (a) varying sample sizes in §3.6.2, (b) varying batch intervals in §3.6.3, (c) varying arrival rates for sub-streams in §3.6.4, (d) varying window sizes in §3.6.5, (e) scalability in §3.6.6, and (f) skew in the input data stream in §3.6.7.

3.6.2 Varying Sample Sizes

Throughput. We first measure the throughput of StreamApprox w.r.t. the Spark- and Flink-based systems with varying sample sizes (sampling fractions). To measure the throughput of the evaluated systems, we increase the arrival rate of the input stream until these systems are saturated.

Figure 3.5 (a) first shows the throughput comparison of StreamApprox and the two sampling algorithms in Spark. Spark-based stratified sampling (STS) scales poorly because of

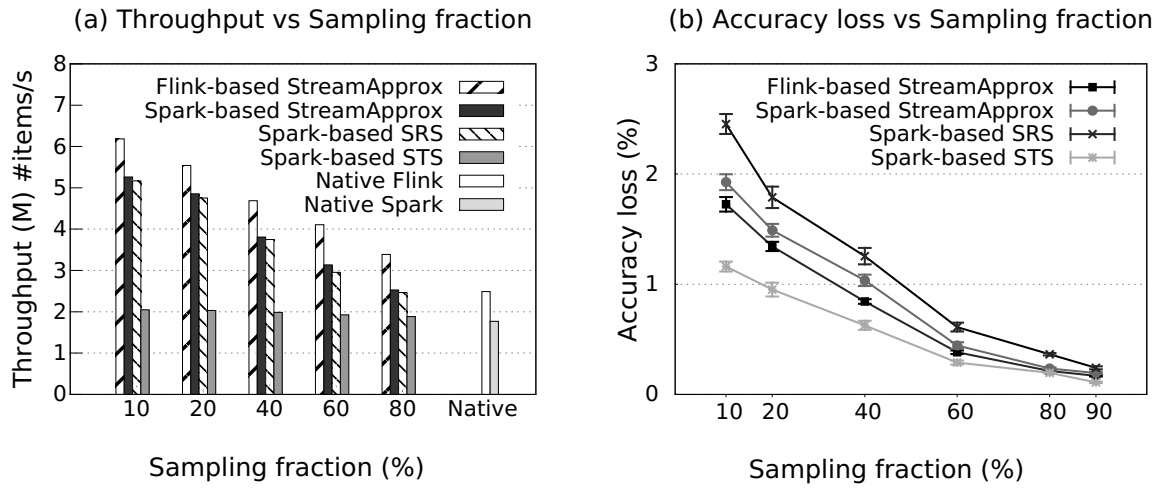

Figure 3.5 – Comparison between StreamApprox, Spark-based SRS, Spark-based STS, as well as native Spark and Flink systems. (a) Peak throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions.

its synchronization among Spark workers and the expensive sorting during its sampling process (as detailed in §3.5.1). Spark-based StreamApprox achieves a throughput of $1.68\times$ and $2.60\times$ higher than Spark-based STS with sampling fractions of 60% and 10%, respectively. In addition, the Spark-based simple random sampling (SRS) scales better than STS and has a similar throughput as in StreamApprox, but SRS looses the capability of considering each sub-stream fairly.

Meanwhile, Flink-based StreamApprox achieves a throughput of $2.13\times$ and $3\times$ higher than Spark-based STS with sampling fractions of 60% and 10%, respectively. This is mainly due to the fact that Flink is a truly pipelined stream processing engine. Moreover, Flink-based StreamApprox achieves a throughput of $1.3\times$ compared to Spark-based StreamApprox and Spark-based SRS with the sampling fraction of 60%.

We also compare StreamApprox with native Spark and Flink systems, i.e., without any sampling. With the sampling fraction of 60%, the throughput of Spark-based StreamApprox is $1.8\times$ higher than the native Spark execution, whereas the throughput of Flink-based StreamApprox is $1.65\times$ higher than the native Flink execution.

Accuracy. Next, we compare the accuracy of our proposed OASRS sampling with that of the two sampling mechanisms with the varying sampling fractions. Figure 3.5 (b) first shows that StreamApprox systems and Spark-based STS achieve a higher accuracy than Spark-based SRS. For instance, with the sampling fraction of 60%, Flink-based StreamApprox, Spark-based StreamApprox, and Spark-based STS achieve the accuracy loss of 0.38%, 0.44%, and 0.29%, respectively, which are higher than Spark-based SRS which only achieves the accuracy loss of 0.61%. This higher accuracy is due to the fact that both StreamApprox and Spark-based STS integrate stratified sampling which ensures that data items from each sub-stream are selected fairly. In addition, Spark-based STS achieves even higher accuracy than StreamApprox, but recall that Spark-based STS needs to maintain a sample size of each sub-stream proportional to the size of the sub-stream (see §3.5.1). This leads to a much lower throughput than StreamApprox which only maintains a sample of a fixed size for each sub-stream.

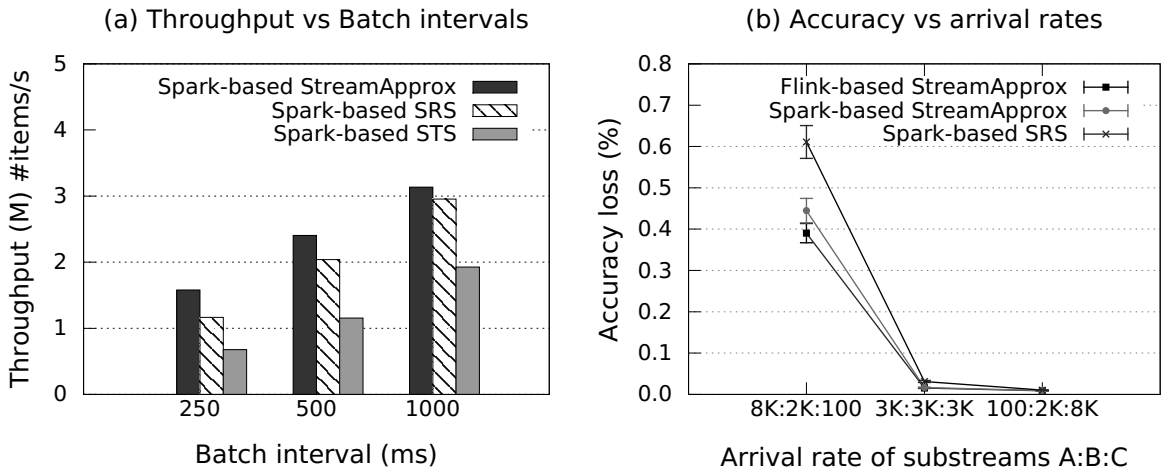

Figure 3.6 – Comparison between StreamApprox, Spark-based SRS, Spark-based STS. (a) Peak throughput with varying batch intervals. (b) Accuracy loss with varying arrival rates. The sliding window size is 10 seconds, and each sliding step is 5 seconds.

3.6.3 Varying Batch Intervals

Spark-based systems adopt the batched stream processing model. Next, we evaluate the impact of varying batch intervals on the performance of Spark-based StreamApprox, Spark-based SRS, and Spark-based STS system. We keep the sampling fraction as 60% and measure the throughput of each system with different batch intervals.

Figure 3.6 (a) shows that, as the batch interval becomes smaller, the throughput ratio between Spark-based systems gets bigger. For instance, with the 1000ms batch interval, the throughput of Spark-based StreamApprox is $1.07\times$ and $1.63\times$ higher than the throughput of Spark-based SRS and STS, respectively; with the 250ms batch interval, the throughput of StreamApprox is $1.36\times$ and $2.33\times$ higher than the throughput of Spark-based SRS and STS, respectively. This is because Spark-based StreamApprox samples the data items without synchronization before forming RDDs and significantly reduces costs in scheduling and processing the RDDs, especially when the batch interval is small (i.e., low-latency real-time analytics).

3.6.4 Varying Arrival Rates for Sub-Streams

In the following experiment, we evaluate the impact of varying rates of sub-streams. We used an input data stream with Gaussian distributions as described in §3.6.1. We maintain the sampling fraction of 60% and measure the accuracy loss of the four Spark- and Flink-based systems with different settings of arrival rates.

Figure 3.6 (b) shows the accuracy loss of these four systems. The accuracy loss decreases proportionally to the increase of the arrival rate of the sub-stream C which carries the most significant data items compared to other sub-streams. When the arrival rate of the sub-stream C is set to 100 items/second, Spark-based SRS system achieves the worst accuracy since it may overlook sub-stream C which contributes only a few data items but has significant

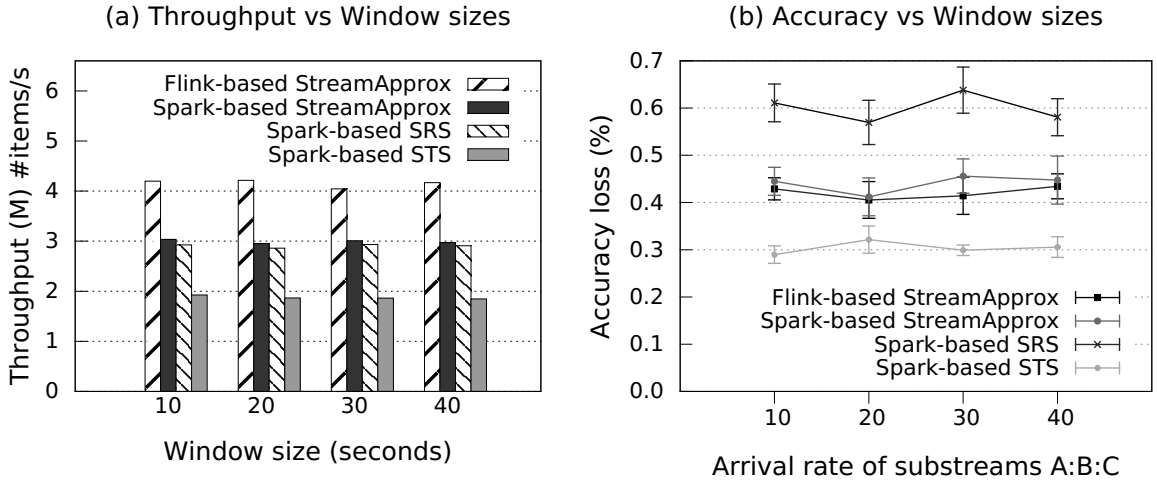

Figure 3.7 – Comparison between StreamApprox, Spark-based SRS, and Spark-based STS. (a) Peak throughput with varying window sizes. (b) Accuracy loss with varying window sizes. The sampling fraction is set to 60%.

values. On the other hand, when the arrival rate of sub-stream C is set to 8000 items/second, the four systems achieve almost the same accuracy. This is mainly because all four systems do not overlook sub-stream C which contains items with the most significant values.

3.6.5 Varying Window Sizes

Next, we evaluate the impact of varying window sizes on the throughput and accuracy of the four systems. We used the same input as described in §3.6.4 with its three sub-streams' arrival rates being 8000, 2000, and 100 items per second. Figure 3.7 (a) and Figure 3.7 (b) show that the window sizes of the computation do not affect the throughput and accuracy of these systems significantly. This is because the sampling operations are performed at every batch interval in the Spark-based systems and at every slide window interval in the Flink-based StreamApprox.

3.6.6 Scalability

To evaluate the scalability of StreamApprox, we keep the sampling fraction as 40% and measure the throughput of StreamApprox and the Spark-based systems with different numbers of CPU cores (scale-up) and different numbers of nodes (scale-out).

Figure 3.8 (a) shows unsurprisingly that StreamApprox and Spark-based SRS scale better than Spark-based STS. For instance, with one node (8 cores), the throughput of Spark-based StreamApprox and Spark-based SRS is roughly 1.8× higher than that of Spark-based STS. With three nodes, Spark-based StreamApprox and Spark-based SRS achieve a speedup of 2.3× over Spark-based STS. In addition, Flink-based StreamApprox achieves a throughput even 1.9× and 1.4× higher compared to Spark-based StreamApprox with one node and three nodes, respectively.

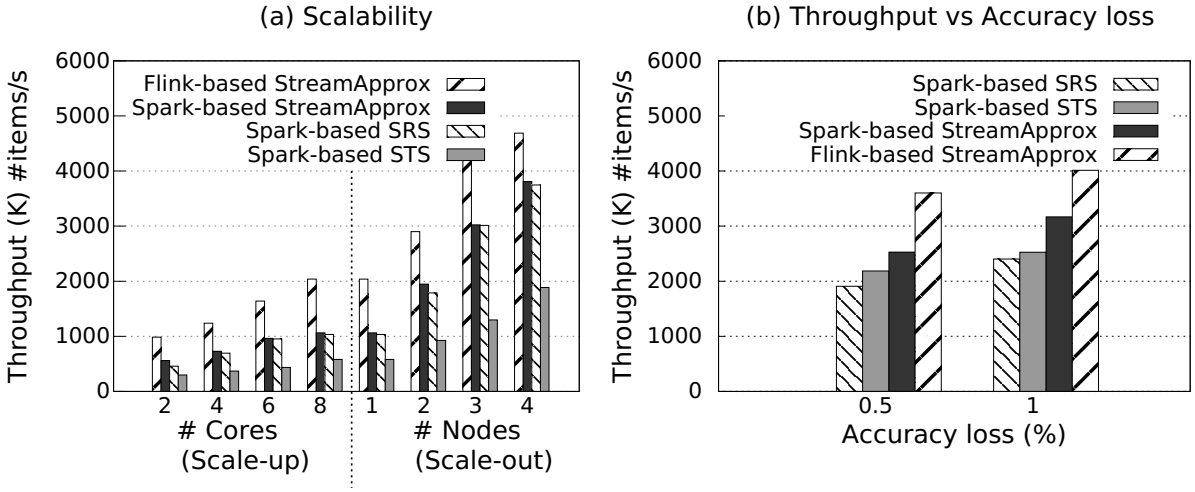

Figure 3.8 – Comparison between StreamApprox, Spark-based SRS, and Spark-based STS. (a) Peak throughput with different numbers of CPU cores and nodes. (b) Peak throughput with accuracy loss.

3.6.7 Skew in the Data Stream

Lastly, we study the effect of the non-uniformity in sub-stream sizes. In this experiment, we construct an input data stream where one of its sub-streams dominates the other sub-streams. In particular, we evaluated the skew in the input data stream using the following two data distributions: (*i*) Gaussian distribution and (*ii*) Poisson distribution.

I: Gaussian distribution. First, we generated an input data stream consisting of three sub-streams A, B, and C with the Gaussian distribution of parameters ($\mu = 100,\ \sigma = 10$), ($\mu = 1000,\ \sigma = 100$), and ($\mu = 10000,\ \sigma = 1000$), respectively. The sub-stream A comprises 80% of the data items in the entire data stream, whereas the sub-streams B and C comprise only 19% and 1% of data items, respectively. We set the sliding window size to $w = 10$ seconds, and each sliding step to $\delta = 5$ seconds.

We keep the accuracy loss across all four systems the same and then measure their respective throughputs. Figure 3.8 (b) shows that, with the same accuracy loss of 1%, the throughput of Spark-based STS is $1.05\times$ higher than Spark-based SRS, whereas Spark-based StreamApprox achieves a throughput $1.25\times$ higher than Spark-based STS. In addition, Flink-based StreamApprox achieves the highest throughput which is $1.68\times$, $1.6\times$, and $1.26\times$ higher than Spark-based SRS, Spark-based STS, and Spark-based StreamApprox, respectively.

II: Poisson distribution. In the next experiment, we generated an input data stream with the Poisson distribution as described in §3.6.1. The sub-stream A accounts for 80% of the entire data stream items, while the sub-stream B accounts for 19.99% and the sub-stream C comprises only 0.01% of the data stream items, respectively. Figure 3.9 shows that StreamApprox systems and Spark-based STS outperform Spark-based SRS in terms of accuracy. The reason for this is StreamApprox systems and Spark-based STS do not overlook sub-stream C which has items with significant values. Furthermore, this result strongly demonstrates the superiority of the proposed sampling algorithm OASRS over simple random sampling in processing long-tail data which is very common in practice.

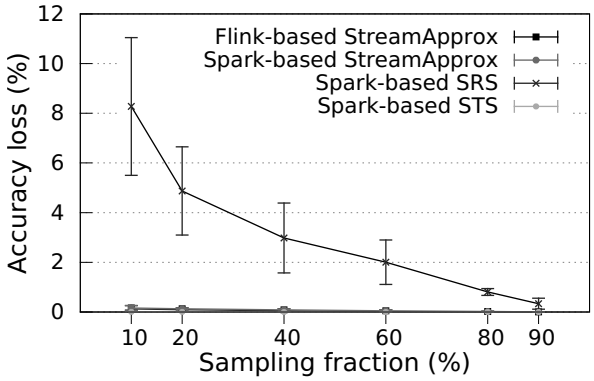

Figure 3.9 – The accuracy loss comparison between StreamApprox, Spark-based SRS, and Spark-based STS with varying sampling fractions.

3.7 Case Studies

In this section, we report our experiences and results with the following two real-world case studies: (a) network traffic analytics (§3.7.2) and (b) New York taxi ride analytics (§3.7.3).

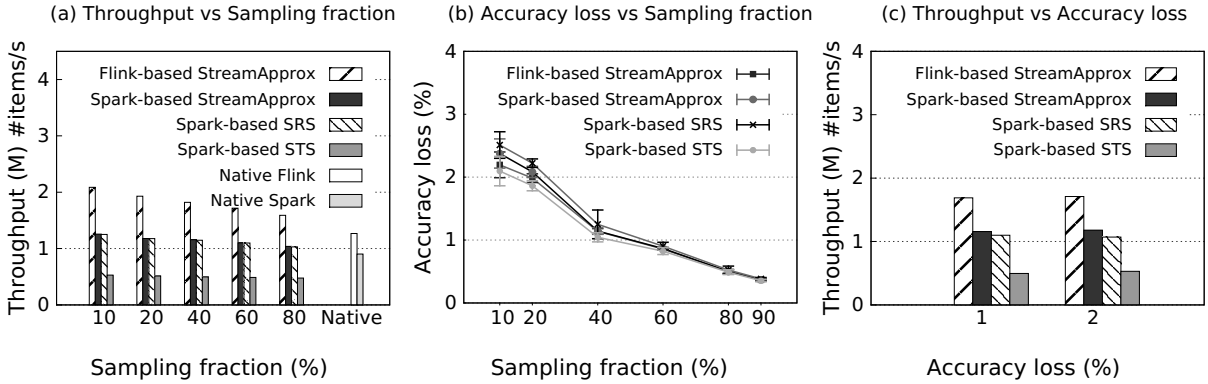

Figure 3.10 – The case study of network traffic analytics with a comparison between StreamApprox, Spark-based SRS, Spark-based STS, as well as the native Spark and Flink systems. (a) Peak throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions. (c) Peak throughput with different accuracy losses.

3.7.1 Experimental Setup

Cluster setup. We performed experiments using a cluster of 17 nodes. Each node in the cluster has 2 Intel Xeon E5405 CPUs (quad core), 8GB of RAM, and a SATA-2 hard disk, running Ubuntu 14.04.5 LTS. We deployed our StreamApprox prototype on 5 nodes (1 driver node and 4 worker nodes), the traffic replay tool on 5 nodes, the Apache Kafka-based stream aggregator on 4 nodes, and the Apache Zookeeper [25] on the remaining 3 nodes.

Measurements. We evaluated StreamApprox using the following key metrics: (a) throughput: measured as the number of data items processed per second; (b) latency: measured as the total

time required for processing the respective dataset; and lastly, (c) accuracy loss: measured as $|approx - exact|/exact$ where $approx$ and $exact$ denote the results from StreamApprox and a native system without sampling, respectively.

Methodology. We built a tool to efficiently replay the case-study dataset as the input data stream. In particular, for the throughput measurement, we tuned the replay tool to first feed 2000 messages/second and continued to increase the throughput until the system was saturated. Here, each message contained 200 data items.

For comparison, we report results from StreamApprox, Spark-based SRS, Spark-based STS systems, as well as the native Spark and Flink systems. For all experiments, we report measurements based on the average over 10 runs. Lastly, the sliding window size was set to 10 seconds, and each sliding step was set to 5 seconds.

3.7.2 Network Traffic Analytics

In the first case study, we deployed StreamApprox for a real-time network traffic monitoring application to measure the TCP, UDP, and ICMP network traffic over time.

Dataset. We used the publicly-available 670GB network traces from CAIDA [51]. These were recorded on the high-speed Internet backbone links in Chicago in 2015. We converted the raw network traces into the NetFlow format [66], and then removed unused fields (such as source and destination ports, duration, etc.) in each NetFlow record to build a dataset for our experiments.

Query. We deployed the evaluated systems to measure the total sizes of TCP, UDP, and ICMP network traffic in each sliding window.

Results. Figure 3.10 (a) presents the throughput comparison between StreamApprox, Spark-based SRS, Spark-based STS systems, as well as the native Spark and Flink systems. The result shows that Spark-based StreamApprox achieves more than $2\times$ throughput than Spark-based STS, and achieves a similar throughput compared with Spark-based SRS (which looses the capability of considering each sub-stream fairly). In addition, due to Flink's pipelined stream processing model, Flink-based StreamApprox achieves a throughput even $1.6\times$ higher than Spark-based StreamApprox and Spark-based SRS. We also compare StreamApprox with the native Spark and Flink systems. With the sampling fraction of 60%, the throughput of Spark-based StreamApprox is $1.3\times$ higher than the native Spark execution, whereas the throughput of Flink-based StreamApprox is $1.35\times$ higher than the native Flink execution. Surprisingly, the throughput of the native Spark execution is even higher than the throughput of Spark-based STS. The reason for this is that Spark-based STS requires the expensive extra steps (see §3.5.1).

Figure 3.10 (b) shows the accuracy loss with different sampling fractions. As the sampling fraction increases, the accuracy loss of StreamApprox, Spark-based SRS, and Spark-based STS decreases (i.e., accuracy improves), but not linearly. StreamApprox systems produce more accurate results than Spark-based SRS but less accurate results than Spark-based STS. Note however that, although both StreamApprox systems and Spark-based STS integrate stratified sampling to ensure that every sub-stream is considered fairly, StreamApprox systems are much more resource-friendly than Spark-based STS. This is because Spark-based STS requires synchronization among workers for the expensive join operation to take samples from the input data stream, whereas StreamApprox performs the sampling operation with a

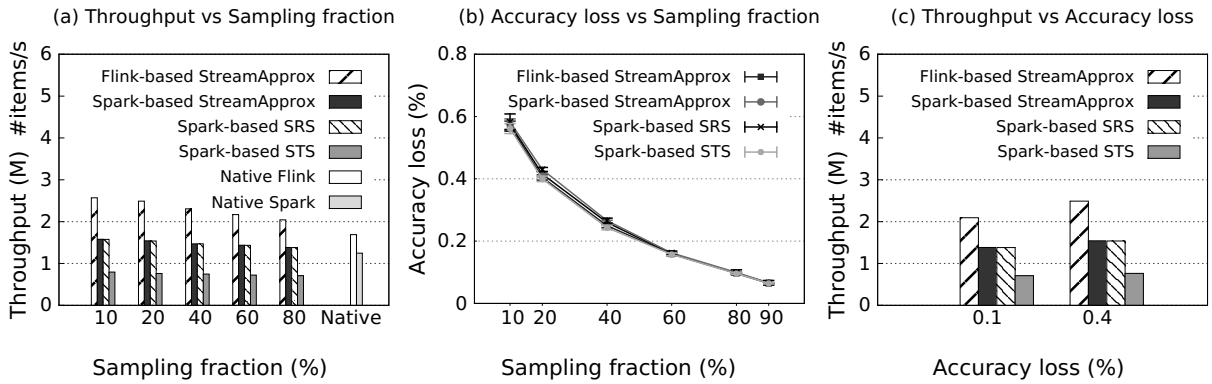

Figure 3.11 – The case study of New York taxi ride analytics with a comparison between StreamApprox, Spark-based SRS, Spark-based STS, as well as the native Spark and Flink systems. (a) Peak throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions. (c) Peak throughput with different accuracy losses.

configurable sample size for sub-streams requiring no synchronization between workers.

In addition, to show the benefit of StreamApprox, we fixed the same accuracy loss for all four systems and then compared their respective throughputs. Figure 3.10 (c) shows that, with the accuracy loss of 1%, the throughput of Spark-based StreamApprox is $2.36\times$ higher than Spark-based STS, and $1.05\times$ higher than Spark-based SRS. Flink-based StreamApprox achieves a throughput even $1.46\times$ higher than Spark-based StreamApprox.

Finally, to make a comparison in terms of latency between these systems, we implemented our proposed sampling algorithm OASRS in Spark-core, and then measured the latency in processing the network traffic dataset. Figure 3.12 indicates that the latency of Spark-based StreamApprox is $1.39\times$ and $1.69\times$ lower than Spark-based SRS and Spark-based STS in processing the network traffic dataset.

3.7.3 New York Taxi Ride Analytics

In the second case study, we evaluated StreamApprox with a taxi ride dataset to measure the average distance of trips starting from different boroughs in New York City.

Dataset. We used the NYC *Taxi Ride* dataset from the DEBS 2015 Grand Challenge [129]. The dataset consists of the itinerary information of all rides across $10,000$ taxies in New York City in 2013. In addition, we mapped the start coordinates of each trip in the dataset into one of the six boroughs in New York.

Query. We deployed StreamApprox, Spark-based SRS, Spark-based STS systems, as well as the native Spark and Flink systems to measure the average distance of the trips starting from various boroughs in each sliding window.

Results. Figure 3.11 (a) shows that Spark-based StreamApprox achieves a similar throughput compared with Spark-based SRS (which, however, does not consider each sub-stream fairly), and a roughly $2\times$ higher throughput than Spark-based STS. In addition, due to Flink's pipelined streaming model, Flink-based StreamApprox achieves a $1.5\times$ higher throughput compared to

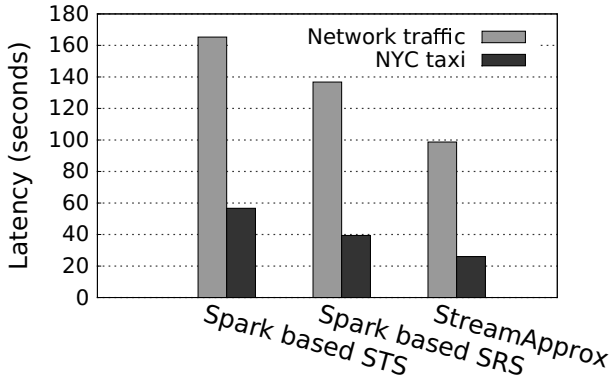

Figure 3.12 – The latency comparison between StreamApprox, Spark-based SRS, and Spark-based STS with the real-world datasets. The sampling fraction is set to 60%.

Spark-based StreamApprox and Spark-based SRS. We again compared StreamApprox with the native Spark and Flink systems. With the sampling fraction of 60%, the throughput of Spark-based StreamApprox is $1.2\times$ higher than the throughput of the native Spark execution, whereas the throughput of Flink-based StreamApprox is $1.28\times$ higher than the throughput of the native Flink execution. Similar to the result in the first case study, the throughput of the native Spark execution is higher than throughput of Spark-based STS.

Figure 3.11 (b) depicts the accuracy loss of these systems with different sampling fractions. The results show that they all achieve a very similar accuracy in this case study. In addition, we also fixed the same accuracy loss of 1% for all four systems to measure their respective throughputs. Figure 3.11 (c) shows that Flink-based StreamApprox achieves the best throughput which is $1.6\times$ higher than Spark-based StreamApprox and Spark-based SRS, and $3\times$ higher than Spark-based STS. Figure 3.12 further indicates that Spark-based StreamApprox provides the $1.52\times$ and $2.18\times$ lower latency than Spark-based SRS and Spark-based STS in processing the NYC taxi ride dataset.

3.8 Discussion

The design of StreamApprox is based on the assumptions mentioned in §3.3.3. Reducing these assumptions is beyond the scope of this chapter. Nevertheless, in this section, we discuss some approaches that could be used to meet our assumptions.

I: Virtual cost function. We currently assume that there exists a virtual cost function to translate a user-specified query budget into the sample size. The query budget could be specified, for instance, as either available computing resources, desired accuracy or desired latency requirement.

For instance, with an accuracy budget, we can define the sample size for each sub-stream based on a desired width of the confidence interval using Equation 3.9 and the "68-95-99.7" rule. With a desired latency budget, users can specify it by defining the window time interval or the slide interval for their computations over the input data stream. It becomes a bit more challenging to specify a budget for resource utilization. Nevertheless, we discuss some existing techniques that could be used to implement such a cost function to achieve the

desired resource target. In particular, we refer to the two existing techniques: (a) Virtual data center [16], and (b) resource prediction model [218–220] for the latency requirement.

Pulsar [16] proposes an abstraction of a virtual data center (VDC) to provide performance guarantees to tenants in the cloud. In particular, Pulsar makes use of a virtual cost function to translate the cost of a request processing into the required computational resources using a multi-resource token algorithm. We could adapt the cost function for our purpose as follows: we consider a data item in the input stream as a request and the "amount of resources" required to process it as the cost in tokens. Also, the given resource budget is converted in the form of tokens, using the pre-advertised cost model per resource. This allows us to compute the number of items, i.e., the sample size, that can be processed within the given resource budget.

For any given latency requirement, we could employ a resource prediction model [218–220]. In particular, we could build the prediction model by analyzing the diurnal patterns in resource usage [57] to predict the future resource requirement for the given latency budget. This resource requirement can then be mapped to the desired sample size based on the same approach as described above.

II: Stratified sampling. In our design in §3.4, we currently assume that the input stream is already stratified based on the source of events, i.e., the data items within each stratum follow the same distribution. This assumption is practical in many cases. For example, consider an IoT use-case which analyzes data streams from sensors to measure the temperature of a city. The data stream from each individual sensor will follow the same distribution since it measures the temperature at the same location in the city. Therefore, a straightforward way to stratify the input data streams is to consider each sensor's data stream as a stratum (sub-stream). In more complex cases when we cannot classify strata based on the sources, we need a preprocessing step to stratify the input data stream. This stratification problem is orthogonal to our work, nevertheless for completeness, we discuss two proposals for the stratification of evolving data streams, namely bootstrap [87] and semi-supervised learning [157].

Bootstrap [87] is a well-studied non-parametric sampling technique in statistics for the estimation of distribution for a given population. In particular, the bootstrap technique randomly selects "bootstrap samples" with replacement to estimate the unknown parameters of a population, for instance, by averaging the bootstrap samples. We can employ a bootstrap-based estimator for the stratification of incoming sub-streams. Alternatively, we could also make use of a semi-supervised algorithm [157] to stratify a data stream. The advantage of this algorithm is that it can work with both labeled and unlabeled data streams to train a classification model.

3.9 Related Work

Given the advantages of making a trade-off between accuracy and efficiency, approximate computing is applied to various domains: graphics, machine learning, scientific simulations, etc. In this context, approximation mechanisms have been proposed at various levels of the system stack, from hardware to applications — including languages, tools, processors, accelerators, memory, and compilers (refer to [191] for a detailed survey). Our work mainly builds on the advancements in the databases community. In this section, we survey the

approximation techniques in this context.

Over the last two decades, the databases community has proposed various approximation techniques based on sampling [11, 100], online aggregation [121], and sketches [73]. These techniques make different trade-offs with respect to the output quality, supported query interface, and workload. However, the early work in approximate computing mainly targeted towards the centralized database architecture.

Recently, sampling-based approaches have been successfully adopted for distributed data analytics [7, 103, 137, 204]. In particular, BlinkDB [7] proposes an approximate distributed query processing engine that uses stratified sampling [11] to support ad-hoc queries with error and response time constraints. ApproxHadoop [103] uses multi-stage sampling [152] for approximate MapReduce job execution. Both BlinkDB and ApproxHadoop show that it is possible to make a trade-off between the output accuracy and the performance gains (also the efficient resource utilization) by employing sampling-based approaches to compute over a subset of data items. However, these "big data" systems target batch processing and cannot provide required low-latency guarantees for stream analytics.

Like BlinkDB, Quickr [204] also supports complex ad-hoc queries in big-data clusters. Quickr deploys distributed sampling operators to reduce execution costs of parallelized queries. In particular, Quickr first injects sampling operators into the query plan; thereafter, it searches for an optimal query plan among sampled query plans to execute input queries. However, Quickr is also designed for static databases, and it does not account for stream analytics. IncApprox [137] is a data analytics system that combines two computing paradigms together, namely, approximate and incremental computations [34–37] for stream analytics. The system is based on an online "biased sampling" algorithm that uses self-adjusting computation [33] to produce incrementally updated approximate output. Lastly, PrivApprox [183] supports privacy-preserving data analytics using a combination of randomized response and approximate computation.

By contrast, in StreamApprox, we designed an "online" sampling algorithm solely for approximate computing, while avoiding the limitations of existing sampling algorithms.

3.10 Conclusion

In this chapter, we presented StreamApprox, a stream analytics system for approximate computing. StreamApprox allows users to make a systematic trade-off between the output accuracy and the computation efficiency. To achieve this goal, we designed an online stratified reservoir sampling algorithm which ensures the statistical quality of the sample from the input data stream. Our proposed sampling algorithm is generalizable to two prominent types of stream processing models: batched and pipelined stream processing models.

To showcase the effectiveness of our proposed algorithm, we built StreamApprox based on Apache Spark Streaming and Apache Flink. We evaluated the effectiveness of our system using a series of micro-benchmarks and real-world case studies. Our evaluation shows that, with varying sampling fractions of 80% to 10%, Spark- and Flink-based StreamApprox achieves a significantly higher throughput of $1.15\times$—$3\times$ compared to the native Spark Streaming and Flink executions, respectively. Furthermore, StreamApprox achieves a speedup of $1.1\times$—$2.4\times$ compared to a Spark-based sampling system for approximate computing, while maintaining

the same level of accuracy for the query output. As mentioned at the beginning, the content of this chapter is based on our conference publication [184] and technical report [181]. Finally, the source code of StreamApprox is publicly available: https://streamapprox.github.io/.

4 IncApprox: Approximate and Incremental Stream Analytics

After building StreamApprox, we analyzed and reviewed its design to improve the performance of it further. We observed that in slide window computation, there is an overlap between two consecutive slide windows which contains data items common to both slide windows. For these overlapping data items, we do not need to recompute the computation over them, thus we can incrementally compute the next slide window based on the result of the preceding slide window. As a result, this observation creates an opportunity for us to improve the performance of StreamApprox using incremental computing paradigm.

In this chapter, we show that the two computing paradigms, incremental and approximate computing, are complementary. Both computing paradigms rely on computing over a subset of data items instead of the entire dataset to achieve low latency and efficient cluster utilization. Therefore, we can combine these paradigms together in order to leverage the benefits of both. To concretize this idea, we designed an online stratified sampling algorithm that uses self-adjusting computation to produce an incrementally updated approximate output with bounded error. We implemented our algorithm in a data analytics system called IncApprox based on Apache Spark Streaming.

This chapter is organized as follows. We first motivate the design of IncApprox in Section §4.1. We next briefly highlight the contributions of IncApprox in Section §4.2. Then, we present an overview of the system in Section §4.3. We next present the design in Section §4.4, implementation in Section §4.5, and evaluation of IncApprox in Section §4.6, and two real world case-studies in Section §4.7. Finally, discussion, related work, and conclusions are presented in Section §4.8, Section §4.9, and Section §4.10, respectively.

The content of this chapter is based on our conference paper [137]. This work is based on a joint collaboration with Dhanya R Krishnan, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues.

4.1 Motivation

The motivation of this work is similar to StreamApprox which is to strike a balance between two desirable, but contradictory design targets in stream processing systems: (*i*) high throughput/low latency and (*ii*) efficient utilization of computing resources. However, to achieve this goal, in IncApprox, we apply not only *approximate computing* paradigm but also *incremental computing* since the both computing paradigms prefer to compute over a subset of data instead of the entire input data. Computing only over a subset of the input data requires much less time and computing resources, and thus, these computing paradigms can achieve high performance and efficient resource utilization in processing continuous high-speed

input data stream.

Incremental computing. Incremental computation is based on the observation that many data analytics jobs operate incrementally by repeatedly invoking the same application logic or algorithm over an input data that differs slightly from that of the previous invocation [112, 119]. For example, in stream processing, the same computations repeatably perform over data items in each slide window. In such a workflow, small, localized changes to the input often require only small updates to the output, creating an opportunity to update the output incrementally instead of recomputing everything from scratch [3]. Since the work done is often proportional to the change size rather than the total input size, incremental computation can achieve significant performance gains (low latency) and efficient utilization of computing resources [35, 176].

The most common way for incremental computation is to rely on programmers to design and implement an application-specific incremental update mechanism (or a *dynamic algorithm*) for updating the output as the input changes [46, 65, 76, 77, 90, 110]. While dynamic algorithms can be asymptotically more efficient than re-computing everything from scratch, research in the algorithms community shows that these algorithms can be difficult to design, implement and maintain even for simple problems. Furthermore, these algorithms are studied mostly in the context of the uniprocessor computing model, making them ill-suited for parallel and distributed settings which is commonly used for large-scale data analytics.

Recent advancements in self-adjusting computation [3–5, 143] overcome the limitations of dynamic algorithms. Self-adjusting computation transparently benefits existing applications, without requiring the design and implementation of dynamic algorithms. At a high level, self-adjusting computation enables incremental updates by creating a dynamic dependence graph of the underlying computation, which records control and data dependencies between the sub-computations. Given a set of input changes, self-adjusting computation performs change propagation, where it reuses the *memoized* intermediate results for all sub-computations that are unaffected by the input changes, and re-computes only those parts of the computation that are transitively affected by the input change. As a result, self-adjusting computation computes only on a subset ("*delta*") of the computation instead of re-computing everything from scratch.

Approximate computing. Approximate computation is based on the observation that many data analytics jobs are amenable to an approximate, rather than the exact output [80, 162, 170, 199]. For instance, for click-stream analysis, it is sufficient to know the relative popularity of webpages rather than the exact access counts for each webpage. For such an approximate workflow, it is possible to trade accuracy by computing over a partial subset instead of the entire input data to achieve low latency and efficient utilization of resources.

Over the last two decades, researchers in the database community proposed many techniques for approximate computing including sampling [11, 100], sketches [73], and online aggregation [121]. These techniques make different trade-offs with respect to the output quality, supported query interface, and workload. However, the early work in approximate computing was mainly targeted towards the centralized database architecture, and it was unclear whether these techniques could be extended in the context of big data analytics.

Recently, sampling based approaches have been successfully adopted for distributed data analytics [7, 103]. These systems show that it is possible to have a trade-off between the output accuracy and performance gains (also efficient resource utilization) by employing

sampling-based approaches for computing over a *subset* of data items. However, these "big data" systems target batch processing workflow and cannot provide low-latency guarantees for stream analytics.

The marriage. We make the observation that the two computing paradigms, incremental and approximate computing, are complementary. Both computing paradigms rely on computing over a subset of data items instead of the entire dataset to achieve low latency and efficient resource utilization.

4.2 Contribution

In this chapter, we propose to combine incremental and approximate computing paradigms together in order to leverage the benefits of both. Furthermore, we achieve incremental updates without requiring the design and implementation of application-specific dynamic algorithms, and support approximate computing for stream analytics.

The high-level idea is to design a sampling algorithm that biases the sample selection to the memoized data items from previous runs. We realize this idea by designing an online sampling algorithm that selects a representative subset of data items from the input data stream. Thereafter, we bias the sample to include data items for which we already have memoized results from previous runs, while preserving the proportional allocation of data items of different (strata) distributions. Next, we run the user-specified streaming query on this biased sample by making use of self-adjusting computation and provide the user an incrementally updated approximate output with error bounds.

We implemented our algorithm in a system called IncApprox based on Apache Spark Streaming [23], and evaluated its effectiveness by applying IncApprox to various micro-benchmarks. Furthermore, we report our experience on applying IncApprox on two real-world case-studies: (*i*) real-time network monitoring, and (*ii*) data analytics on a Twitter stream. Our evaluation using real-world case-studies shows that IncApprox achieves a speedup of $\sim 2\times$ over the native Spark Streaming execution, and $\sim 1.4\times$ over the individual speedups of both incremental and approximate computing.

To summarize, we makes the following contributions.

- We present the first system that transparently combines the incremental and approximate computing paradigms for stream analytics.
- We present an adaptive execution mechanism using an interface of a query budget that allows users to make a trade-off between the output accuracy and latency guarantees or available computing resources.
- Finally, we provide a confidence metric on the output accuracy using an error bound or confidence interval. This gives a measure of accuracy trade-off on the result due to the approximation.

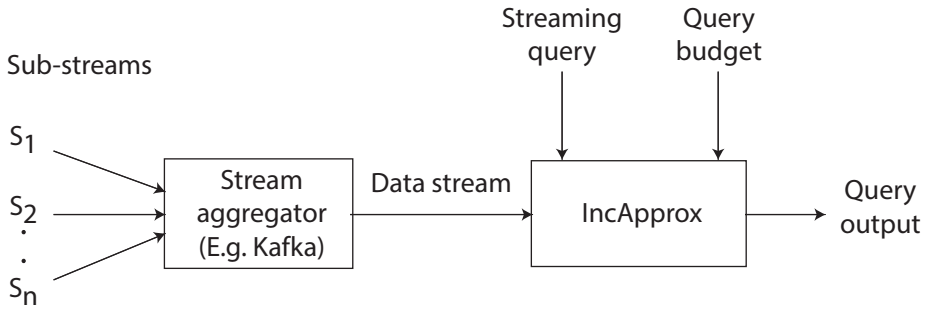

Figure 4.1 – System overview.

4.3 Overview

4.3.1 System Overview

IncApprox is designed for real-time data analytics on online data streams. Figure 4.1 depicts the high-level design of IncApprox. The online data stream consists of data items from diverse sources of events or sub-streams. We use a stream aggregator (such as Apache Kafka [130], Apache Flume [18], Amazon Kinesis [14], etc.) that integrates data from these sub-streams, and thereafter, the system reads this integrated data stream as the input. We facilitate user querying on this data stream by providing a user interface that consists of a streaming query and a query budget. The user submits the streaming query to the system as well as specifies a query budget. The query budget can either be in the form of latency guarantees/SLAs for data processing, desired result accuracy, or computing resources available for query processing. Our system makes sure that the computation done over the data remains within the specified budget. To achieve this, the system makes use of a mixture of incremental and approximating computing for real-time processing over the input data stream, and emits the query result along with the confidence interval or error bounds.

4.3.2 Design Goals

The goals of the IncApprox system are to:

- *Provide application transparency*: We aim to support unmodified applications for stream processing, i.e., the programmers do not have to design and implement application-specific dynamic algorithms or sampling techniques.
- *Guarantee query budget*: We aim to provide an adaptive execution interface, where the users of the system can specify their query budget in terms of tolerable latency/SLAs, desired result accuracy, or the available cluster resources, and our system guarantees the processing within the budget.
- *Improve efficiency*: We aim to achieve high efficiency with a mix of incremental and approximate computing.
- *Guarantee a confidence level*: We aim to provide a confidence level for the approximate output, i.e., the accuracy of the output will remain within an error range.

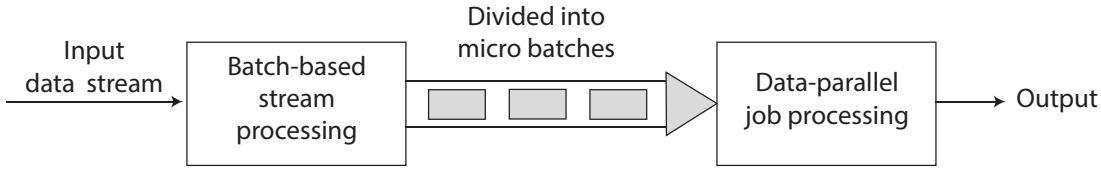

Figure 4.2 – Batch-based stream processing.

4.3.3 System Model

Before we explain the design of IncApprox, we present the system model assumed in this work.

Programming model. Our system supports a *batched streaming processing* programming model. In batched stream processing (see Figure 4.2), the online data stream is divided into small batches or sets of records; and for each batch a distributed data-parallel job is launched to produce the output.

As opposed to the trigger-based programming model (see [227] for details), the batched streaming model provides three main advantages: (*i*) it provides simple fault tolerance based on re-computation of tasks, and efficient handling of stragglers using speculative execution; (*ii*) it provides consistent "exact-once" semantics for records processing instead of weaker semantics such as "at least once" or "at most once"; and finally, (*iii*) it provides a unified data-parallel programming model that could be utilized for batch as well as stream processing workflows. Given these advantages, the batched streaming model is widely adopted by many stream processing frameworks including Spark Streaming [23], Flink [17], Slider [38], TimeStream [180], Trident [208], MapReduce Online [70], Comet [119], Kineograph [64], and NOVA [50].

Computation model. Our computation model for stream processing is *sliding window computations*. In this model (see Figure 4.3), the computation window slides over the input data stream, where new arriving input data items are added to the window and the old data items are dropped from the window as they become less relevant to the analysis.

In sliding window computations, there is a substantial overlap of data items between the two successive computation windows, especially, when the size of the window is large relative to the slide interval. This overlap of unchanged data-items provides an opportunity to update the output incrementally.

Assumptions. Our system makes the following assumptions. We discuss these assumptions and the different possible methods to enforce them in §4.8.

1. We assume that the input stream is stratified based on the source of event, i.e., the data items within each stratum follow the same distribution, and are mutually independent. Here a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.
2. We assume the existence of a virtual function that takes the user specified budget as the input and outputs the sample size for each window based on the budget.
3. We assume that the memoized results for incremental computation are stored in the way that is fault-tolerant.

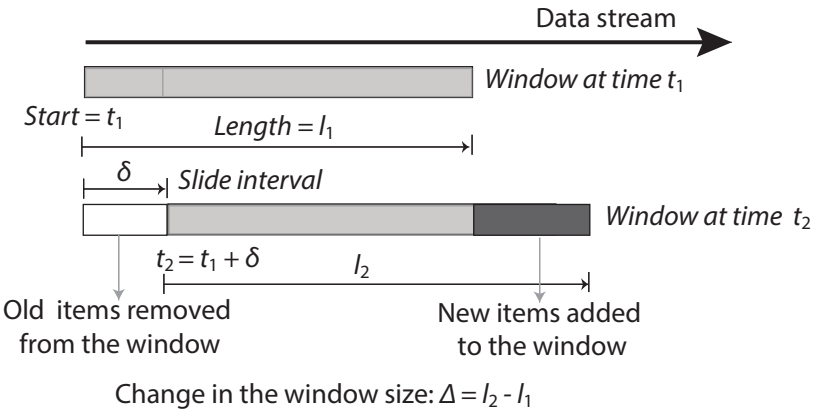

Figure 4.3 – Sliding window computation over data stream.

Lastly, we assume a time-based window length, and based on the arrival rate, the number of data items within a window may vary accordingly. Note that this assumption is consistent with the sliding window APIs in the aforementioned systems.

4.3.4 Building Blocks

Our system leverages several computational and statistical techniques to achieve the goals discussed in §4.3.2. Next, we briefly describe these techniques and the motivation behind our design choices.

Stratified sampling. In a streaming environment, since the window size might be very large, for a realistic rate of execution, we perform approximation using samples taken within the window. But the data stream might consist of data from disparate events. As such, we must make sure that every sub-stream is considered fairly to have a representative sample from each sub-stream. For this we use stratified sampling [11]. Stratified sampling ensures that data from every stratum is selected and none of the minorities are excluded. For statistical precision, we use proportional allocation of each sub-stream to the sample [12]. It ensures that the sample size of each sub-stream is in proportion to the size of sub-stream in the whole window.

Self-adjusting computation. For incremental sliding window computations, we use self-adjusting computation [3–5, 143] to re-use the intermediate results of sub-computations across successive runs of jobs. In this technique we maintain a dependence graph between sub-computations of a job, and reuse memoized results for sub-computations that are unaffected by the changed input in the computation window.

Error estimation. For defining a confidence level on the accuracy of the approximated output, we use error estimation [68]. This specifies a confidence interval or error bound for the output, i.e., we emit the output in the following form : output $\pm$ error margin. A confidence level along with the margin of error tells how accurate is the approximate output.

4.4 Design

In this section, we present the detailed design of IncApprox.

4.4.1 Algorithm Overview

Algorithm 4 presents an overview of our approach. The algorithm computes a user-specified streaming *query* as a sliding window computation over the input data stream. The user also specifies a query *budget* for executing the query, which is used to derive the sample size ($sampleSize$) for the window using a cost function (see §4.3.3 and §4.8). The cost function ensures that processing remains within the query budget.

For each window (see Figure 4.3), we first adjust the computation window to the current start time t by removing all old data items from the $window$ ($timestamp < t$). Similarly, we also drop all old data items from the list of memoized items ($memo$), and the respective memoized results of all sub-computations that are dependent on those old data items.

Next, we read the new incoming data items in the $window$. Thereafter, we perform proportional stratified sampling (detailed in §4.4.2) on the $window$ to select a sample of size provided by the cost function. The stratified sampling algorithm ensures that samples from all strata are proportional, and no stratum is neglected.

Next, we bias the stratified sample to include items from the memoized sample, in order to enable the reuse of memoized results from previous sub-computations. The biased sampling algorithm (detailed in §4.4.3) biases samples *specific to each stratum*, to ensure reuse, and at the same time, retain proportional allocation.

Thereafter, on this biased sample, we run the user specified *query* as a data-parallel job *incrementally*, i.e., we reuse the memoized results for all data items that are unchanged in the window, and update the output based on the changed (or new) data items. After the job finishes, we memoize all the items in the sample and their respective sub-computation results for reuse for the subsequent windows. The details are covered in §4.4.4.

The job provides an estimated output which is bound to a range of error due to approximation. We perform error estimation (as described in §4.4.5) to estimate this error bound and define a confidence interval for the result as: $output \pm error\ bound$.

The entire process repeats for the next window, with updated windowing parameters and the sample size. (Note that the query budget can be updated across windows during the course of stream processing to adapt to the user's requirements.)

4.4.2 Stratified Reservoir Sampling

Stratified sampling clusters the input stream into homogenous disjoint sets of strata (here $homogenous$ means the items within a stratum have same distribution) and selects a random sample from each stratum. Meanwhile, reservoir sampling selects a uniform random sample of *fixed size* without replacement, from an input stream of unknown size. We perform a combined *stratified reservoir sampling*, adopted from the approach in [11], along with proportional allocation, i.e., we sample the streaming data within a sliding window by stratifying the stream, and applying reservoir sampling within each stratum proportionally. By combining these two

Algorithm 4: Basic algorithm

Input: streaming *query* and query *budget*
Windowing parameters (see Figure 4.3):
$t \leftarrow$ start time; $\delta \leftarrow$ slide interval;

```
1  begin
2      window ← ∅; // List of items in the window
3      memo ← ∅; // List of items memoized from the window
4      sample ← ∅; // Set of items sampled from the window
5      biasedSample ← ∅; // Set of items in biased sample
6      foreach window in the incoming data stream do
7          // Remove all old items from window and memo
8          forall elements in the window and memo do
9              if element.timestamp < t then
10                 window.remove(element);
11                 memo.remove(element);
12         // Add new items to the window
13         window ← window.insert(new items);
14         // Cost function gives the sample size based on the budget
15         sampleSize ← costFunction(budget);
16         // Do stratified sampling of window (§4.4.2)
17         sample ← stratifiedSampling(window, sampleSize);
18         // Bias the stratified sample to include memoized items (§4.4.3)
19         biasedSample ← biasSample(sample, memo);
20         // Run query as an incremental data parallel job for the window (§4.4.4)
21         output ← runJobIncrementally(query, biasedSample);
22         // Memoize all items & respective sub-computations for sample (§4.4.4)
23         memo ← memoize(biasedSample);
24         // Estimate error for the output (§4.4.5)
25         output ± error ← estimateError(output);
26         // Update the start time for the next window
27         t ← t + δ;
```

techniques, statistical quality of the sample is maintained—as sample from *every stratum* is selected *proportionally*, and a random sample of *fixed size*—given by cost function is selected from the window.

The stratified reservoir sampling algorithm (described in Algorithm 5) uses a fixed size reservoir with size equal to the sample size. It allocates the space in the reservoir proportionally to the samples from each stratum, based on number of items seen so far in the corresponding stratum. As we move forward through the window for sampling, the arrival rate of items in each stratum may change, hence the proportional allocation must be updated. Therefore, periodically, the algorithm re-allocates the space in the reservoir to ensure proportional allocation. Thereafter, based on this re-allocation, we adapt the algorithm to use an adaptive reservoir sampling (ARS) [12] for those strata whose sub-reservoir sizes are changed, and conventional reservoir sampling (CRS) [11] for those strata whose sub-reservoir sizes are unchanged. (Let reservoir consists of a group of sub-reservoirs, each for storing sample from each stratum). ARS ensures that we periodically adjust the proportional allocation (based on the arrival rate), and CRS ensures randomness in sampling technique. Once the

Algorithm 5: Stratified reservoir sampling algorithm

```
Input: T ← Interval for re-calculation of sub-reservoir size
1  stratifiedSampling(window, sampleSize)
2  begin
3      S ← ∅ // Ordered set of all strata seen so far in window
4      forall item belonging to stratum S_i in window do
5          S.add(S_i); // Add new stratum seen to S
6          i ← Index of stratum S_i ;
7          // Fill reservoir until sampleSize is reached
8          if (∑_{h=1}^{|S|} |sample[h]|) < sampleSize then
9              sample[i].add(item); // Add item to its sub-reservoir
10         else
11             if T interval is passed then
12                 forall S_i in S do
13                     i ← Index of stratum S_i ;
14                     // Compute new sub-reservoir size using Equation 4.1
15                     newSize[i] ← sample[i].computeSize();
16                     if newSize[i] ≠ |sample[i]| then
17                         c ← newSize[i] − |sample[i]|;
18                         // Do Adaptive Reservoir Sampling
19                         sample[i] ← ARS(c, sample[i], S_i);
20                     else
21                         // Do Conventional Reservoir Sampling
22                         sample[i] ← CRS(item, sample[i], S_i);
23                     // Skip items in window, if seen by ARS or CRS
24                     skipItemsSeen(); // Details omitted
25             else
26                 // Until T, do Conventional Reservoir Sampling
27                 sample[i] ← CRS(item, sample[i], S_i);
```

sub-reservoir's proportional allocation is handled using ARS, the sampling technique switches back to CRS, until the next re-allocation interval.

Algorithm 5 works as follows: For each $item$ seen in a window, if the stratum of the item is newly seen, then we add it to the set of strata seen so far. Initially, we fill the reservoir of sample until it is full. Here the reservoir is a store for our stratified sample $'sample'$, and can be considered as a group of sub-reservoirs of different strata such that: $|sample| = \sum_{i=0}^{|S|-1} |sample[i]|$ where S is the ordered set of all strata seen so far in the window, and $sample[i]$ is the sub-reservoir of the sample from the i^{th} stratum. We fill the reservoir by adding each $item$ to its corresponding sub-reservoir, based on the stratum to which the $item$ belongs.

Once the reservoir is full, then until a pre-decided periodical time interval T to re-allocate sub-reservoir sizes, we proceed with a conventional reservoir sampling (CRS). In CRS technique, for each of the further items seen in each stratum S_i, we decide with a probability $\frac{|sample[i]|}{|S_i|}$ whether to accept or reject the item, i.e., all items in a stratum have equal probability

Algorithm 6: Subroutines for the stratified sampling algorithm

```
1  Let incomingItems[ ] represent incoming items seen when moving forward through window
2  ARS(c, sample[i], S_i)
3  begin
4      if c > 0 then
5          // Add c items to sample[i] from incoming items belonging to S_i
6          ∀j ∈ {0, ..., c − 1} : sample[i].add(incomingItems[S_i].get(j));
7      else
8          // Evict random c items from sample[i]
9          ∀j ∈ {0, ..., c − 1} :
10         // random(a, b)gives a random number between [a, b]
11         sample[i].remove(random(0, |sample[i]| −1));
12 CRS(item, sample[i], S_i)
13 begin
14     p ← |sample[i]| / |S_i|; // Probability of replacement
15     // Replace a random item from sample[i] with item, using probability p
16     sample[i].replace(sample[i][random(0, |sample[i]| −1)],
17     item, p);
```

of inclusion [11]. If the item is accepted, then we replace a randomly selected item in the corresponding sub-reservoir with the accepted item.

After T interval of time, we re-allocate the sub-reservoir sizes of each stratum, to ensure proportional allocation. This T interval determines how frequently proportional allocation is verified. Thus, T is selected based on frequency of change in the arrival rate in each stratum (since change in arrival rate changes proportional allocation), by counting the number of items of each stratum per time unit at the stream aggregator. First, after interval T, we compute the size of sub-reservoir to be allocated to each i^{th} stratum at current time t'. It is computed proportional to the total number of items seen so far in the corresponding stratum within the window, using the equation:

$$|sample[i](t')| = sampleSize * \frac{|S_i|}{k} \tag{4.1}$$

where $sampleSize$ is the total size allocated to reservoir, $|S_i|$ is the number of items seen so far in the stratum S_i and k is the total number of items seen so far in the window.

Thereafter, if the re-allocated sub-reservoir size $|sample[i](t')|$ at current point of time t' is different from the previously adjusted sub-reservoir size (i.e., if there is any change in sub-reservoir size), we proceed with ARS—to adapt according to this change in size (described in Algorithm 6) as follows: When sub-reservoir size of S_i has increased by c, then from the incoming stream, we insert c items that belong to stratum S_i, to the corresponding sub-reservoir $sample[i]$. If the sub-reservoir size has decreased by c, we evict c number of items from the sub-reservoir. This ensures that proportional allocation is retained.

If the re-allocated sub-reservoir size of a stratum is unchanged, we proceed with CRS for the stratum as explained before.

We perform stratified reservoir sampling until the window terminates and the resulting

Algorithm 7: Biased sampling algorithm

```
1  biasSample(sample, memo)
2  begin
3      S ← sample.getAllStrata(); // Set of all strata in sample
4      foreach i^th stratum S_i in S do
5          x ← memo[i].size(); // no. of items memoized from S_i
6          y ← sample[i].size(); // no. of items in sample from S_i
7          biasedSample[i] ← ∅; // List of items in biased sample from S_i
8          if x ⩾ y then
9              // Add y items from memo[i] to biasedSample[i] to enable re-use
10             ∀j ∈ {0, ..., y − 1} : biasedSample[i].add(memo[i].get(j));
11         else
12             // First add x items from memo[i] to biasedSample[i]
13             ∀j ∈ {0, ..., x − 1} : biasedSample[i].add(memo[i].get(j));
14             // Fill the remaining (y − x) items from the stratified sample
15             int j = 0;
16             while (biasedSample[i].size() < y) do
17                 biasedSample[i].add(sample[i].get(j));
18                 j + +;
```

stratified sample consists of samples from each stratum, proportional to the size of corresponding stratum seen in the whole window.

4.4.3 Biased Sampling

Biased sampling. Biased sampling enables result reuse by including memoized data items in the sample, but at the same time, ensures that the proportional allocation of samples from each stratum is retained. In sliding window computations where the window slides by small intervals, there is only a small change in the input based on insertion and deletion from the window (see Figure 4.3). Hence, we memoize and reuse the results of sub-computations whose input is unchanged. However, if we reuse *all* memoized results from the previous window, the proportional allocation is lost, since proportions in different windows may vary due to difference in the arrival rate of sub-streams. Therefore, we select the *number of memoized items for result reuse*, based on the number of items in the sample from each stratum.

Algorithm 7 describes our biased sampling algorithm. In this algorithm, we bias the sample from each stratum separately. Note that here, "memoized items" and "sample size" are specific to each stratum. The algorithm works as follows: If the number of memoized items x is greater than or equal to the sample size y, then we create a biased sample with only y items from the memoized list, and neglect the extra memoized items. If the number of memoized items is less than sample size, then we give priority to memoized items and create a biased sample with all memoized items first, and later we add more items to this biased sample from the stratified sample until the size of biased sample becomes equal to the size of stratified sample. This ensures proportional allocation. However, some of the memoized items in $memo$ might be already in the stratified sample, and this might cause duplicates in the biased sample. Therefore, in practice, we use a data structure such as a HashSet for storing $biasedSample$ to

remove duplicates automatically. Finally, we get a biased sample which includes all essential memoized items as well as stratified samples based on the arrival rate, thus ensuring both reuse and proportional allocation.

Precision and accuracy in biased sampling. An estimated result is *precise* if similar results are obtained with repeated sampling, and it is *accurate* if estimated result is closer to the true result (a precise result doesn't necessarily be accurate always) [152]. Our stratified sample is precise than a random sample since it considers every stratum, and uses proportional allocation. Accuracy of a stratified sample is more if (*i*) different strata have major differences and (*ii*) within each stratum, there is homogeneity [152]. Based on our assumptions in §4.3.3, our stratified sample is accurate since different stratum have different distribution, and items within each stratum follow same distribution (homogenous).

We bias the sample from each stratum separately, thus preserving the statistics of stratified sampling, i.e., after the bias, the biased sample still consists of items from each stratum in the same proportional allocation obtained from stratified sampling. Further, even though the items selected within a stratum are biased to include memoized items which belong to the same stratum, since the items follow same distribution, there is little difference between items within a stratum. Thus our bias sampling technique is as precise and accurate as how the stratified sample is, provided the assumptions hold.

4.4.4 Run Job Incrementally

Next, we run the user-specified streaming query as an *incremental* data-parallel job on the biased sample (derived in §4.4.3). For that, we make use of self-adjusting computation [3, 4].

In self-adjusting computation, the computation is divided into sub-computations, and a dynamic dependence graph is constructed to record dependencies between these sub-computations. Formally, a Dynamic Dependence Graph $DDG = (V, E)$ consists of nodes (V) representing sub-computations and edges (E) representing data and control dependencies between the sub-computations. Thereafter, a change propagation algorithm is used to update the output by propagating the input changes through the dependence graph. The change propagation algorithm identifies a set of sub-computations that directly depend on the changed data and re-executes those sub-computations. This in turn leads to re-computation of other data-dependent sub-computations. Change propagation terminates when all transitively dependent sub-computations are re-computed. For all the unaffected sub-computations, the algorithm reuses memoized results from previous runs without re-computation. Lastly, results for all re-computed (or newly computed) sub-computations are memoized for the next incremental run.

Next, we illustrate the application of self-adjusting computation to a data-parallel job based on the MapReduce model [75]. (Note that our implementation is based on Spark Streaming [23], which is a generic extended version of MapReduce.) Figure 4.4 shows the dependence graph built based on the data-flow graph of the MapReduce model. The data-flow graph is represented by a DAG, where *map* and *reduce* tasks represent nodes (or sub-computations) in the dependence graph, and the directed edges represent the dependencies between these tasks. For an incremental run, we launch *map* tasks for the newly added data items in the sample (M_5 and M_6), and reuse the memoized results for the *map* tasks from previous runs (M_1 to M_4). The output of the newly computed *map* tasks invalidates the

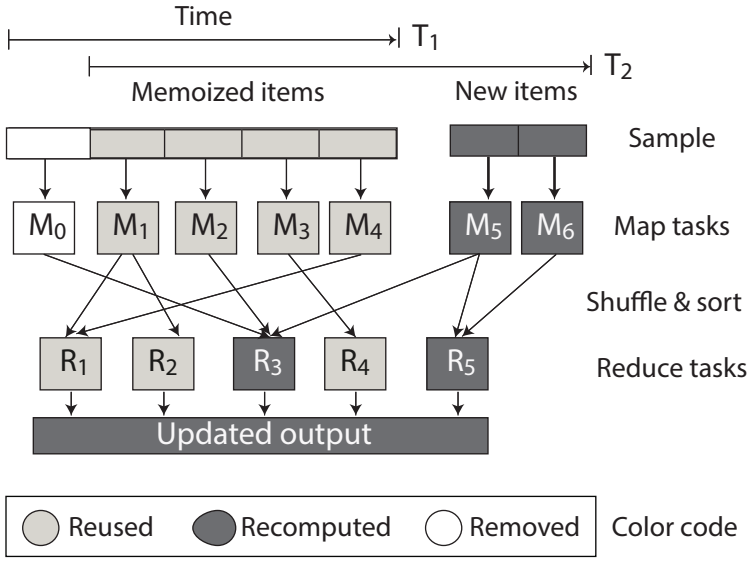

Figure 4.4 – Run data-parallel job incrementally.

dependent *reduce* tasks (R_3 and R_5). However, all *reduce* tasks that are unaffected by the changed input can simply reuse their memoized results without re-computation (R_1, R_2, and R_4). Lastly, we memoize the results for all *freshly* executed tasks for the next incremental run. Note that the items removed from the window also act as the input change (e.g., M_0), and sub-computations dependent on the removed items are also re-computed (e.g., R_3).

4.4.5 Estimation of Error Bounds

In order to provide a confidence interval for the approximate output, we estimate the error bounds due to approximation.

Approximation for aggregate functions. Aggregate functions require results based on all the data items or groups of data items in the population. But since we compute only over a small sample from the population, we get an *estimated* result based on the weightage of the sample.

Consider an input stream S, within a window, consisting of n disjoint strata S_1, S_2 ..., S_n, i.e., $S = \sum_{i=1}^{n} S_i$. Suppose the i^{th} stratum S_i has B_i items and each item j has an associated value v_{ij}. Consider an example to take sum of these values, across the whole window, represented as $\sum_{i=1}^{n}(\sum_{j=1}^{B_i} v_{ij})$. To find an approximate sum, we first select a sample from the window based on stratified and biased sampling as described in §4.4, i.e., from each i^{th} stratum S_i in the window, we sample b_i items. Then we estimate the sum from this sample as: $\hat{\tau} = \sum_{i=1}^{n}(\frac{B_i}{b_i}\sum_{j=1}^{b_i} v_{ij}) \pm \epsilon$ where the error bound ϵ is defined as:

$$\epsilon = t_{f,1-\frac{\alpha}{2}}\sqrt{\widehat{Var}(\hat{\tau})} \tag{4.2}$$

Here, $t_{f,1-\frac{\alpha}{2}}$ is the value of the t-distribution (i.e., *t-score*) with f degrees of freedom and $\alpha = 1 -$ confidence level. The degree of freedom f is expressed as:

$$f = \sum_{i=1}^{n} b_i - n \tag{4.3}$$

The estimated variance for sum, $\widehat{Var}(\hat{\tau})$ is represented as:

$$\widehat{Var}(\hat{\tau}) = \sum_{i=1}^{n} B_i * (B_i - b_i)\frac{s_i^2}{b_i} \tag{4.4}$$

where s_i^2 is the population variance in the i^{th} stratum. Since the bias sampling is such that the statistics of stratified sampling is preserved, we use the statistical theories [205] for stratified sampling to compute the error bound.

Currently, we support error estimation only for aggregate queries. For supporting queries that compute extreme values, such as minimum and maximum, we can make use of extreme value theory [68, 150] to compute the error bounds.

Error bound estimation. For error bound estimation, we first identify the sample statistic used to estimate a population parameter, e.g., *sum*, and we select a desired confidence level, e.g., 95%. In order to compute the margin of error ϵ using t-score as given in Equation 4.2, the sampling distribution must be nearly normal. The Central Limit Theorem (CLT) states that when the size of sample is sufficiently large ($>= 30$), then the sampling distribution of a statistic approximates to *normal distribution*, regardless of the underlying distribution of values in the data [205]. Hence, we compute t-score using a t-distribution calculator [158], with the given degree of freedom f (see Equation 4.3), and cumulative probability as $1 - \alpha/2$ where $\alpha = 1 -$ confidence level [152]. Thereafter, we estimate the variance using the corresponding equation for the sample statistic considered (for *sum*, the Equation is 4.4). Finally, we use this t-score and estimated variance of the sample statistic and compute the margin of error using Equation 4.2.

4.5 Implementation

We implemented IncApprox based on the Apache Spark Streaming framework [23]. Figure 4.5 presents the high-level architecture of our prototype, where the shaded boxes depict the implemented modules. In this section, we first give a brief necessary background on Spark streaming, and next, we present the design details of the implemented modules.

Background. Spark Streaming is a scalable and fault-tolerant distributed stream processing framework. It offers batched stream processing APIs (as described in §4.3.3), where a streaming computation is treated as a series of batch computations on small time intervals. For each interval, the received input data stream is first stored on a cluster's memory and a distributed file system such as HDFS [19] or Tachyon [147]. Thereafter, the input data is processed using Apache Spark [229], a distributed data-parallel job processing framework similar to MapReduce [75] or Dryad [127].

Spark Streaming is built on top of Apache Spark, which uses Resilient Distributed Datasets (RDDs) [229] for distributed data-parallel computing. An RDD is an immutable and fault-tolerant collection of elements (objects) that is distributed or partitioned across a set of nodes in a cluster. Spark Streaming extends the RDD abstraction by introducing the DStreams APIs [227], which is a sequence of RDDs arrived during a time window.

IncApprox implementation. Our implementation builds on the Spark Streaming APIs to implement the approximate and incremental computing mechanisms. At a high-level (see

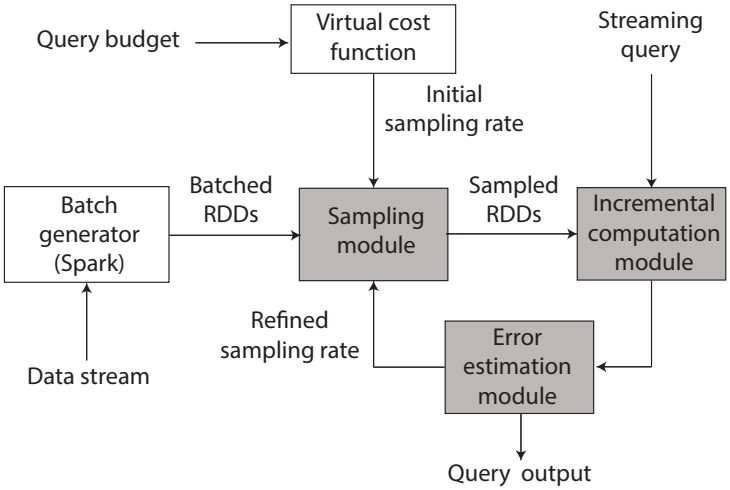

Figure 4.5 – Architecture of IncApprox prototype (shaded boxes depict the implemented modules).

Figure 4.5), the input data stream is split into batches based on a pre-defined interval (e.g., one second). Each batch is defined as a sequence of RDDs. Next, the RDDs in each batch are sampled by the sampling module, with an initial sampling rate computed from the query budget using the virtual cost function. The sampled RDDs are inputs for the incremental computation module. In this module, the sampled RDDs are processed incrementally to provide the query result to the user. Finally, the error is estimated by the error estimation module. If the value of the error is higher than the error bound target, a feedback mechanism is activated to tune the sampling rate in the sampling module to provide higher accuracy in the subsequent query results. We next explain the details for the implemented modules.

I: Sampling module. The sampling module implements the approximation mechanism as described in §4.4. For that, we adapt sampling methods available in Spark, namely *sample*(), to implement our sampling algorithm.

II: Incremental computation module. The incremental computation module implements the self-adjusting computation mechanism as described in §4.4.4. To implement this component, we reuse the caching mechanism available in Spark to memoize the intermediate results for the tasks. For the reduction operations, we adapt a windowing operation in Spark Streaming, namely *reduceByKeyAndWindow*() to incrementally update the output. Finally, the dependence graph is maintained at Spark's job controller.

III: Error estimation module. Finally, the error estimation module calculates the error bounds for the output and sends feedback to the sampling module to tune the sample size in order to satisfy the accuracy constraint. We implement the algorithm described in §4.4.5 using the *Apache Common Math* library [158].

In general, our modifications in Spark Streaming are fairly straightforward, and could easily be adapted to other batched streaming processing frameworks (described in §4.3.3). More importantly, we support unmodified applications since we did not modify the application programming interface.

4.6 Evaluation

In this section, we first present a micro-benchmarks based evaluation, and next, we report our experience on deploying IncApprox for the real-world case-studies (§4.7).

For analyzing the effectiveness of memoization in improving the result reuse rate, we evaluate IncApprox using a simulated data stream. In particular, our evaluation analyzes the impact of varying four different parameters, namely, sample size, slide interval, window size, and arrival rate for sub-streams.

We generated a synthetic data stream with three different sub-streams. Each sub-stream is generated with an independent Poisson distribution and different mean arrival rates. For the first three experiments, i.e., to analyze the impact of sample size, slide interval, and window size on memoization, we generated three sub-streams with a mean arrival rate of $3:4:5$ data items per unit time respectively. To analyze the impact of the fluctuating arrival rate of events, we generated two sub-streams with fluctuating arrival rates, and kept the third sub-stream with a constant arrival rate, for a comparative analysis.

4.6.1 Varying Sample Sizes

We first study the effect of varying sample sizes on memoization by applying our algorithm (described in §4.4) to the synthetic data stream. For the experiment, we keep the other parameters—window size and slide interval—fixed. We measure the average number of memoized items from each sub-stream S_1, S_2, S_3 with different arrival rates $3:4:5$ respectively, by varying the total sample size.

Figure 4.6 (a) shows our measurements with a fixed window size of $10,000$ items, 4% slide interval (i.e., 400) and varying sample sizes (on x-axis): 10%, 20%, 40%, 60% and 80% of the window size. We observe that the average number of data items memoized is directly proportional to the sample size and the arrival rate. When the sample size increases, the average number of data items memoized increases constantly because more items from the previous window is available for memoization. We also observe a higher memoization rate for sub-streams with higher arrival rates, the reason being a proportional allocation of sub-sample sizes.

4.6.2 Varying Slide Intervals

Next, we evaluate the impact of varying slide intervals on memoization with constant window and sample sizes. We measure the average number of items memoized from each sub-stream with different slide intervals.

Figure 4.6 (b) shows our measurements with a fixed window size of $10,000$ and sample size of 10% window size (i.e., 1000), but varying slide intervals (on x-axis): 1%, 2%, 4%, 8%, and 16% of the window size. We observe that when the slide interval is 1%, our algorithm memoizes an average of 99.5% of total samples, which greatly improves the reuse rate, and thus, leads to higher efficiency. As evident from the plot, when the slide interval increases, the percentage of memoized items decreases, because larger slides allow fewer samples to reuse from the previous window. We also repeated the experiments with different window sizes, but observed very similar results. Thus, the results illustrate that smaller slides (which

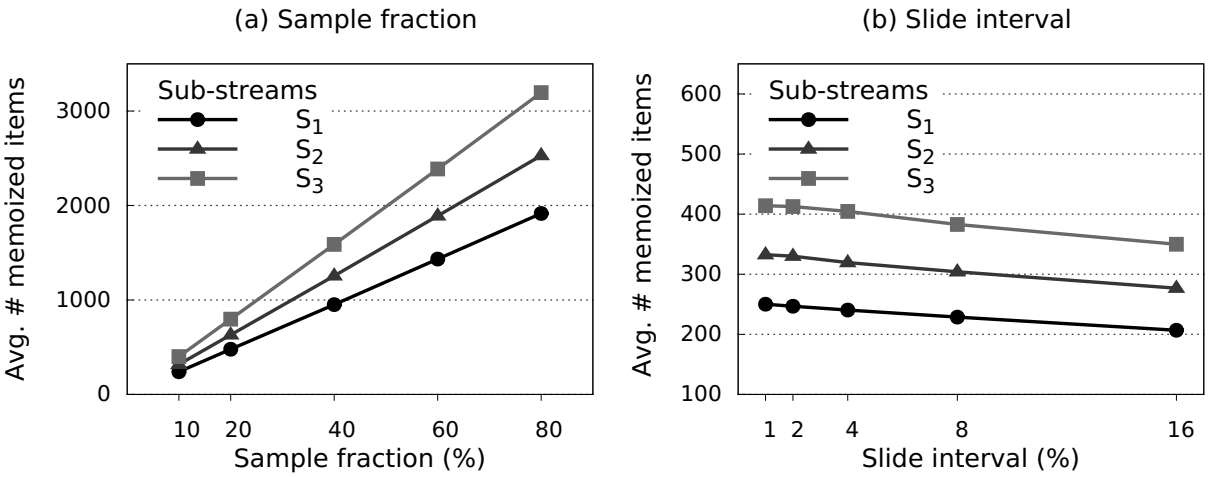

Figure 4.6 – The effect of various sample fractions and slide intervals on the memoization.

is the usual case for an incremental workflow) allow higher memoization and thus higher result reuse.

4.6.3 Varying Window Sizes

Next, we evaluate the impact of varying window sizes on memoization. We measure the number of items in a sample and the number of items memoized from the previous window, and analyze the reuse rate based on this measurement. We begin our experiment with a window of $10,000$ items, and then increase/decrease the window size, e.g., we first increase the window size by 200, then decrease it by 100, etc.

Figure 4.7 (a) shows our measurements, with a fixed 2% slide interval and 10% sample size for each corresponding window size. The x-axis represents Δ, i.e., the change in window size between two adjacent windows (see Figure 4.3). The figure illustrates that whenever the window size decreases (*i.e.*, Δ *is negative*), memoized samples are more than the samples needed in the current window. For example, when Δ is -100, sample size is 1010 and we have 1017 memoized items from the previous window i.e., decreasing window size can allow a 100% re-use rate, provided the slide interval is considerably low (here 2%). The figure also depicts that when window size increases (*i.e*, Δ *is positive*), the sample size is higher than the number of memoized items from the previous window, and the larger the increase in the window size, the larger is the difference between samples needed and memoized. This implies a lesser result reuse rate.

4.6.4 Varying Arrival Rates for Sub-Streams

Lastly, we evaluate the effect of fluctuating arrival rate of sub-streams. As mentioned earlier, we generated two sub-streams, each with fluctuating arrival rates, and a third sub-stream with a constant arrival rate for the analysis. We measure the percentage of items memoized from each sub-stream.

Figure 4.7 (b) depicts the memoization based on fluctuating arrival rates, for a fixed window

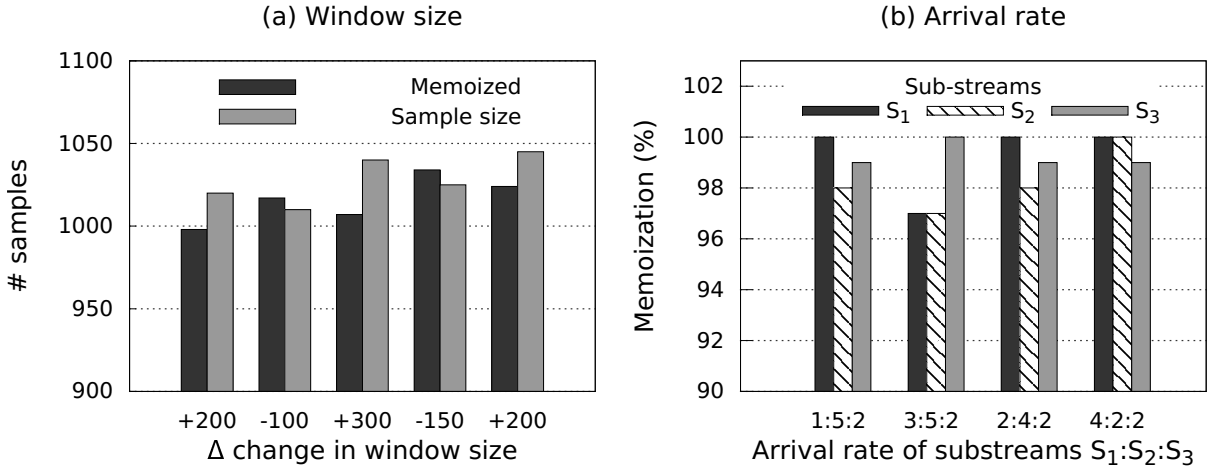

Figure 4.7 – The effect of various window sizes and arrival rates on the memoization.

of $10,000$ items and sample size of 10%. The x-axis shows the arrival rate for the three substreams S_1, S_2, and S_3. The figure illustrates that memoization is inversely proportional to the arrival rate. For example, for sub-stream S_1, when the arrival rate increases from 1 to 3, the percentage of memoization decreases, because the sample size gets higher due to proportional allocation, but memoized items available are lesser. When S_1's arrival rate decreases from 3 to 2, we observe that the memoization increases since we have more items memoized from the previous window. Sub-stream S_2 also depicts similar behaviour. However we notice that even though arrival rate is constant for the third sub-stream, its memoization rate differs relative to the other two sub-streams since we use a proportional allocation of sample sizes. The figure illustrates that in spite of the fluctuations in arrival rates, IncApprox has a memoization rate greater than 97%.

4.7 Case Studies

Next, we present our experience on deploying IncApprox for the following two real-world case-studies: (*i*) network traffic monitoring, and (*ii*) data analytics on Twitter stream.

4.7.1 Experimental Setup

Cluster setup. For the evaluation, we used a cluster of 24 nodes connected via Gigabit Ethernet (1000BaseT full duplex). Each node has 2 Intel Xeon E5405 CPUs (quad core), 8GB of RAM, and a SATA-2 hard disk, running Debian Linux 5.0 with kernel $2.6.26$. We deployed IncApprox on 20 nodes and Apache Kafka [130] stream aggregator on the remaining 4 nodes (the setup is similar to Figure 4.1).

Measurements. We evaluated two key metrics: throughput and accuracy loss. The throughput is defined as the number of processed records per second, and the accuracy loss is defined as $(approx - exact)/exact$ where $approx$ and $exact$ are the results obtained from approximate and native executions respectively. We report the average over 20 runs for all measurements.

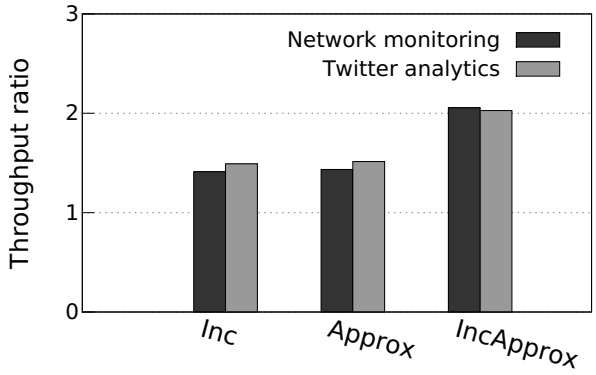

Figure 4.8 – The ratio of peak throughput of Inc, Approx, and IncApprox to the peak throughput of native Spark Streaming with the sampling fraction is set to 60% and with end-to-end latency $350ms$.

Finally, to assess the individual performance benefits of incremental (Inc) and approximate (Approx) computing paradigms, we switched on Inc (incremental computation) and Approx (sampling + error estimation) modules separately. For Inc, the window size is set to 10 seconds, and the window slide interval is set to 2 seconds. For Approx, the sampling fraction is set to 60%.

4.7.2 Network Traffic Analytics

Network traffic monitoring plays an important role in network management [52, 144]. In this case study, we evaluated the performance of IncApprox in a real-time network traffic monitoring application that measures the number of TCP, UDP, and ICMP packets over time.

Dataset. We used network traffic traces from CAIDA [51]. In particular, we used the network traces captured on the high-speed Internet backbone links in Chicago (labeled as A) containing around 670 GB of network traffic in year 2015.

Methodology. From the CAIDA traces, we created a NetFlow [66] dataset for our experiments. We developed a tool that allows us to tune the throughput of the NetFlow stream, i.e., the number of messages sent per second and the number of NetFlow records per message. The experiment was conducted for 30 minutes. The throughput of the stream is tuned to measure the system throughput. The stream starts with 1000 messages/second and continues to increase throughput until the system is exhausted. Each message from the stream contains 200 NetFlow records.

Results.

Figure 4.8 shows the throughput comparison between Approx, Inc, IncApprox, and native Spark Streaming. The individual throughput for approximate computing (Approx) and incremental computing (Inc) is $1.41\times$ and $1.43\times$ higher than native Spark Streaming execution, respectively. However, IncApprox performs significantly better with the combined benefits of both paradigms, an improvement of $2.1\times$ over the native execution.

Figure 4.9 (a) indicates that the throughput decreases quickly with the increasing sampling fraction. With the sampling fraction of 5%, IncApprox achieves a $2.6\times$ higher throughput compared to the native execution, whereas the throughput of Approx is $2.14\times$ higher than

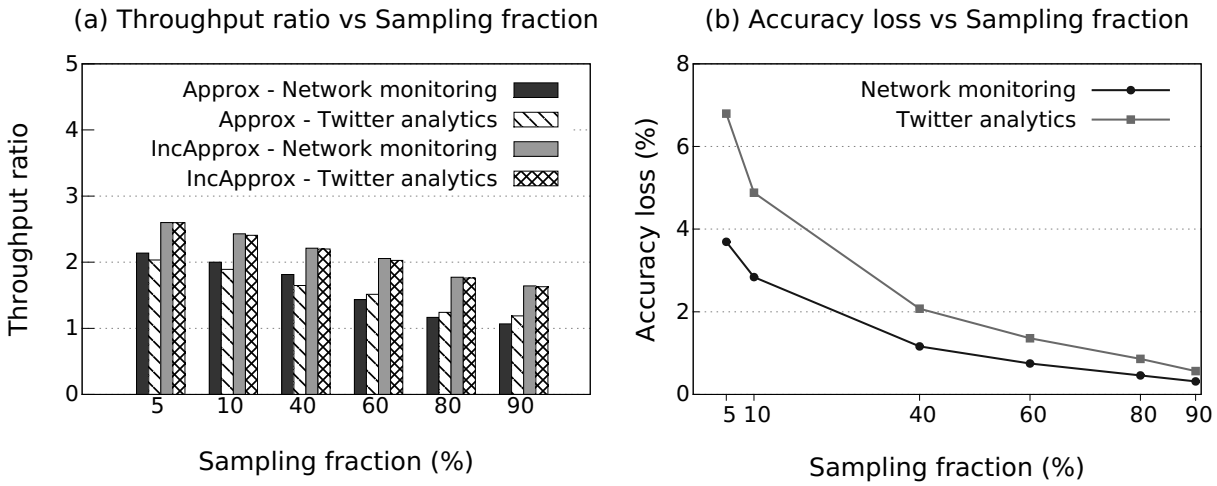

Figure 4.9 – The effect of varying sampling fractions on: (*a*) The ratio of the peak throughput of Approx and IncApprox to the peak throughput of native Spark Streaming, and (*b*) The accuracy loss of Approx and IncApprox with end-to-end latency $350ms$.

the throughput of the native execution. At the sampling fraction of 90%, the throughput is $1.58\times$ and $1.9\times$ less than the throughput with 5% sampling fraction for IncApprox and Approx, respectively.

Figure 4.9 (b) shows the accuracy loss during the approximate computation under different sampling rates. As the sample size increases, the accuracy loss gets smaller (in other words, accuracy improves), though not in a linear fashion.

4.7.3 Twitter Analytics

Analyzing online social networks is an active research area [163]. In the second case-study, we evaluated IncApprox on a real-time Twitter data stream to compute trending topics.

Dataset. For this case study, we developed a crawler using the Twitter API [209] to collect publicly available tweets during three days, from September 17 to September 19, 2015.

Methodology. Since the Twitter API rate limits the number of returned tweets per request, we first dumped the crawled tweets dataset to a CSV file, and developed a tool to replay the tweets as a Twitter stream. This tool allows to control the throughput of the tweet stream. In our experiments, the throughput of the tweet stream is started with 1000 messages/second and continuously increased until the system is exhausted.

Results.

Figure 4.8 represents the throughput of each approach while processing the tweet stream. Approx and Inc achieve $1.49\times$ and $1.51\times$ higher throughput than native Spark Streaming. IncApprox is $2\times$ better than native in terms of throughput.

Figure 4.9 (a) indicates that sampling fractions directly affect the throughput of IncApprox and Approx. With the sampling fraction of 5%, the throughput of IncApprox is $2.6\times$ higher than the throughput of the native execution, whereas this value of Approx is $2.03\times$. At the sampling fraction of 90%, the throughput is $1.6\times$ and $1.7\times$ less than the throughput with 5%

sampling fraction for IncApprox and Approx, respectively.

Figure 4.9 (b) shows the accuracy loss with different sampling rates. The accuracy loss in case of Twitter analytics has a similar but slightly higher curve as for network monitoring.

4.8 Discussion

The design of IncApprox is based on three assumptions (see §4.3.3). Solving these assumptions is beyond the scope of this chapter. However, in this section, we discuss some of the approaches that could be used to meet our assumptions.

I: Stratification of sub-streams. Currently we assume that sub-streams are stratified, i.e., the data items of individual sub-streams have the same distribution. However, it may not be the case. As also discussed in Chapter §3, we next present two alternative approaches, namely bootstrap [87, 88, 175] and a semi-supervised learning algorithm [157] to classify evolving data streams.

Bootstrap [87, 88, 175] is a non parametric re-sampling technique used to estimate parameters of a population. It works by randomly selecting a large number of bootstrap samples with replacement and with the same size as in the original sample. Unknown parameters of a population can be estimated by averaging these bootstrap samples. We could create such a bootstrap-based classifier from the initial reservoir of data, and the classifier could be used to classify sub-streams. Alternatively, we could employ a semi-supervised algorithm [157] to stratify a data stream. This algorithm manipulates both unlabeled and labeled data items to train a classification model.

II: Virtual cost function. Secondly, we assume that there exists a virtual function that computes the sample-size based on the user-specified query budget. The query budget could be specified as either available computing resources or latency requirements. As also discussed in the previous chapter, we suggest two existing approaches—Pulsar [16] and resource prediction model [99, 155]—to design such a virtual function for given computing resources and latency requirements, respectively.

Pulsar [16] is a system that allocates resources based on tenants' demand, using a multi-resource token bucket. It provides a workload independent guarantee using a pre-advertised cost model, i.e., for each appliance and network, it advertises a virtual cost function that maps a request to its cost in tokens. We could adopt a similar cost model as follows: An "item", i.e., *a data block* to be processed, could be considered as a request and "amount of resources" needed to process it could be the cost in tokens. Since the resource budget gives total resources (*here tokens*) to be used, we could find the number of items, i.e., the sample size, that can be processed using these resources, ruling out faults and stragglers.

To find the sample-size for a given latency budget, we could use a resource prediction model based on performance metrics and QoS parameters in SLAs. Such a model could analyze the diurnal patterns of resource usage [57], e.g., off-line predictions based on pre-recorded resource usage log or predictions based on statistical machine learning [99, 155], to predict the future resource requirements based on workload and latency. Once we get the resource requirement for a latency budget using this model, we could find the sample-size needed by using the above suggested method similar to Pulsar.

III: Fault tolerance. Our current algorithm does not take into account the failure of nodes in the cluster where memoized results are stored. We discuss three different approaches that could be adopted for fault tolerance if memoized results are unavailable: *(i)* we could continue processing the window without using any memoized items, albeit with lower efficiency; *(ii)* we could use a similar approach for fault tolerance as provided in Spark [229], where the lineage of memoized RDDs is used to recompute only the lost RDD partitions; *(iii)* we could make use of underlying distributed fault tolerant file-systems (HDFS [19]) to *asynchronously* replicate the memoized results.

4.9 Related Work

IncApprox builds on two computing paradigms, namely, incremental and approximate computing. In this section, we survey the techniques proposed in these two paradigms.

Incremental computation. Since modifying the output of computation incrementally is asymptotically more efficient than re-computing everything from scratch, incremental computation is an active area of research for "big data" analytics. Earlier big data systems for incremental computation proposed an alternative programming model where the programmer is asked to implement an efficient incremental-update mechanism. Examples of non-transparent systems include Google's Percolator [173], and Yahoo's CBP [151]. A downside of these early proposals is that they depart from the existing programming model, and also require implementation of dynamic algorithms on per-application basis, which could be difficult to design and implement.

To overcome the limitations of the aforementioned systems, researchers proposed transparent approaches for incremental computation. Examples of transparent systems include Incoop [35, 37], Comet [119], DryadInc [176], Slider [38] and NOVA [50]. These systems leverage the underlying data-parallel programming model such as MapReduce [75] or Dryad [127] for supporting incremental computation. Our work builds on transparent big data systems for incremental computation. In particular, we leverage the advancements in self-adjusting computation [3] to improve the efficiency of incremental computation. In contrast to the existing approaches, our approach extends incremental computation with the idea of approximation, thus further improving the performance and throughput for applications.

Approximate computation. Approximation techniques such as sampling [11, 100], sketches [73], and online aggregation [121] have been well-studied over the decades in the context of traditional (centralized) database systems. Recently proposed systems such as ApproxHadoop [103] and BlinkDB [7, 8] showed that it is possible to achieve the benefits of approximate computing also in the context of distributed big data analytics.

ApproxHadoop [103] uses multi-stage sampling [152] for approximate MapReduce [75] job execution. BlinkDB [7, 8] proposed an approximate distributed query processing engine that uses stratified sampling [11] to support ad-hoc queries with error and response time constraints. Our system builds on the advancements in approximate computing for big data analytics. However, our system is different from the existing approximate computing systems in two crucial aspects. First, unlike the existing systems, ApproxHadoop and BlinkDB, that are designed for batch processing—we target stream processing. Second, we extend approximate computing with incremental computation.

4.10 Conclusion

In this chapter, we presented the marriage of incremental and approximate computations. Our approach transparently benefits unmodified applications, i.e., programmers do not have to design and implement application-specific dynamic algorithms or sampling techniques. We build on the observation that both computing paradigms rely on computing over a subset of data items instead of computing over the entire dataset. We marry these two paradigms by designing a sampling algorithm that biases the sample selection to the memoized data items from previous runs. We implemented our algorithm in a data analytics system called IncApprox based on Apache Spark Streaming. Our evaluation shows that IncApprox achieves improved benefits of low-latency execution and efficient utilization of resources. As mentioned at the beginning of this chapter, the content of this chapter is based on our conference publication [137].

5 PrivApprox: Approximate and Privacy-Preserving Stream Analytics

We have so far designed and implemented StreamApprox and IncApprox that strike a balance between the two desirable classical goals in stream processing systems: (*i*) achieving high-throughput/low-latency and (*ii*) efficient utilization of computing resources. These systems allow users to systematically make a trade-off between throughput/latency and accuracy. However, we wanted to explore new trade-off domains in addition to throughput/latency and accuracy.

This motivated us to conduct the third work in this thesis — PrivApprox to expand the trade-off domain with a new *privacy* dimension. This work is based on the observation that nowadays, online advertisement is a major economic force for modern online services, where users' private data is continuously collected for real-time data analytics. In the current advertisement eco-system, the goals of users and data analysts are at odds: users seek stronger privacy, while analysts strive for high-utility data analytics in near real time. In this chapter, we target to design a pragmatic privacy-preserving data analytics system PrivApprox that resolves this tension.

This chapter is organized as follows. We first motivate the design of PrivApprox in Section §5.1. We next briefly highlight the contributions of PrivApprox in Section §5.2. Thereafter, we present the overview of PrivApprox in Section §5.3. Next, we describe the detailed design of PrivApprox in Section §5.4. Next, we present the implementation of PrivApprox based on the design in Section §5.5. Then, we present an experimental evaluation of PrivApprox using micro-benchmarks in Section §5.6 and two real world case-studies in Section §5.7. Thereafter, we discuss the limitations of our design in Section §5.8. Finally, the related work and conclusions are presented in Section §5.9 and Section §5.10, respectively. Note that we present the privacy analysis and proofs of PrivApprox system in Appendix §A.1.

The content of this chapter is based on our conference paper [183] and our technical report [182]. This work is based on a joint collaboration with Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe.

5.1 Motivation

Many online services continuously collect users' private data for real-time analytics. Much of this data arrives as a data stream and in huge volumes, requiring real-time stream processing based on distributed systems [17, 20, 23, 24].

In the current ecosystem of data analytics, the analysts usually have direct access to the users' private data, and must be trusted not to abuse it. However, this trust has been violated in the past [69, 120, 178, 196].

A pragmatic eco-system has two desirable, but contradictory design requirements: (*i*) stronger privacy guarantees for the users; and (*ii*) high-utility stream analytics in real-time. Users seek stronger privacy, while analysts strive for high-utility analytics in real time.

To meet these two design requirements, there is a surge of novel computing paradigms that address these concerns, albeit *separately*. Two such paradigms are *privacy-preserving analytics* to protect user privacy and *approximate computation* for real-time analytics.

Privacy-preserving analytics. Recent privacy-preserving analytics systems favor a distributed architecture to avoid central trust (see §5.9 for details), where users' private data is stored locally on their respective client devices. Data analysts use a publish-subscribe mechanism to run aggregate queries over the distributed private dataset of a large number of clients. Thereafter, such systems add noise to the aggregate output to provide useful privacy guarantees, such as differential privacy [82]. Unfortunately, these state-of-the-art systems normally deal with single-shot batch queries, and therefore, these systems cannot be used for real-time stream analytics.

Approximate computation. As also mention in previous chapters, approximate computation is based on the observation that many data analytics jobs are amenable to an approximate, rather than the exact output (see §5.9 for details). For such an approximate workflow, it is possible to trade accuracy by computing over a partial subset (usually selected via a sampling mechanism) instead of the entire input dataset. Thereby, data analytics systems based on approximate computation can achieve low latency and efficient utilization of resources. However, the existing systems for approximate computation assume a centralized dataset, where the desired sampling mechanism can be employed. Thus, existing systems are not compatible with the distributed privacy-preserving analytics systems.

The marriage. We make the observation that the two computing paradigms, privacy-preserving analytics and approximate computation, are complementary. Both paradigms strive for an approximate instead of the exact output, but they differ in their *means* and *goals* for approximation. Privacy-preserving analytics adds explicit *noise* to the aggregate query result to protect users' privacy. Whereas, approximate computation relies on a representative *sampling* of the entire dataset to compute over only a subset of data items to enable low-latency/efficient analytics.

5.2 Contribution

In this chapter, we marry privacy-preserving analytics and approximate computing paradigms together in order to leverage the benefits of both. The high-level idea is to achieve privacy (via approximation) by directly computing over a subset of sampled data items (instead of computing over the entire dataset) and then adding an explicit noise for privacy-preservation.

To realize this marriage, we designed an approximation mechanism that also achieves privacy-preserving goals for stream analytics. Our design (see Figure 5.1) targets a distributed setting, similar as aforementioned, where users' private data is stored locally on their respective personal devices, and an analyst issues a streaming query for analytics over the distributed private dataset of users. The analyst's streaming query is executed on the users' data periodically (a configurable epoch) and the query results are transmitted to a centralized aggregator via a set of proxies. The analyst interfaces with the aggregator to get the aggregate

query output periodically.

We employ two core techniques to achieve our goal. Firstly, we employ *sampling* [166] directly at user's site for approximate computation, where each user randomly decides whether to participate in answering the query in the current epoch. Since we employ sampling at the data source, instead of sampling at a centralized infrastructure, we are able to squeeze out the desired data size (by controlling the sampling parameter) from the very "first stage" in the analytics pipeline, which is essential in low-latency environments.

Secondly, if the user participates in the query answering process, we employ a *randomized response* [96] mechanism to add noise to the query output at user's site, again locally at the source of the data in a decentralized fashion. In particular, each user locally randomizes its truthful answer to the query to achieve local differential privacy guarantees (§5.4.2). Since we employ noise addition at the source of data, instead of adding the explicit noise to the aggregate output at a trusted aggregator or proxies, we enable a truly "synchronization-free" distributed architecture, which requires *no coordination* among proxies and the aggregator for the mandated noise addition.

The last, but not the least, silver bullet of our design: it turns out that the combination of the two aforementioned techniques (i.e., sampling and randomized response) led us to achieve zero-knowledge privacy [101], a privacy bound tighter than the state-of-the-art differential privacy [82]. (We prove our claim in §A.1.)

To summarize, we present the design and implementation of a practical system for privacy-preserving stream analytics in real time. In particular, our system is a novel combination of the sampling and randomized response techniques, as well as a scalable "synchronization-free" routing scheme employing a light-weight XOR encryption scheme [62]. The resulting system ensures zero-knowledge privacy, anonymization, and unlinkability for users (§5.3.2). Altogether, we make the following contributions:

- We present a marriage of sampling and randomized response to achieve improved performance and stronger privacy guarantees.
- We present an adaptive query execution interface for analysts to systematically make a trade-off between the output accuracy, and the query execution budget.
- We present a confidence metric on the output accuracy using a confidence interval to interpret the approximation due to sampling and randomization.

To empirically evaluate our approach, we implemented our design as a fully-functional prototype in a system called PrivApprox[1] based on Apache Flink [17] and Apache Kafka [130]. In addition to stream analytics, we further extended our system to support privacy-preserving "historical" batch analytics over users' private datasets. The evaluation based on micro-benchmarks and real-world case-studies shows that this marriage is, in fact, made in heaven!

5.3 Overview

In this section, we present an overview of our system called PrivApprox.

[1] The source code of PrivApprox along with the experimental evaluation setup is publicly available: https://PrivApprox.github.io.

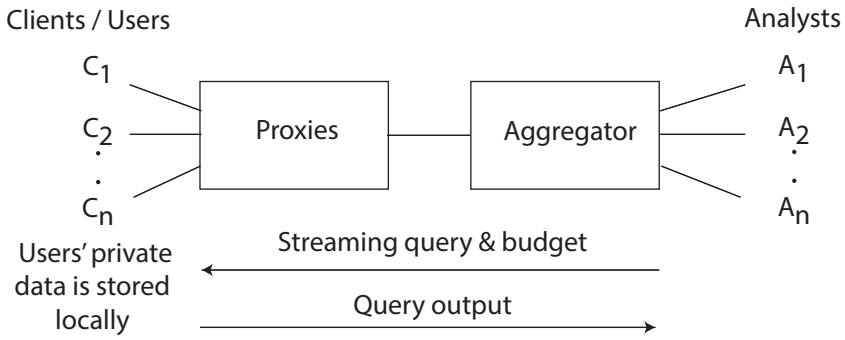

Figure 5.1 – System overview.

5.3.1 System Architecture

PrivApprox is designed for privacy-preserving stream analytics on distributed users' private dataset. Figure 5.1 depicts the high-level architecture of PrivApprox. Our system consists of four main components: clients, proxies, aggregator, and analysts.

Clients locally store users' private data on their respective personal devices, and subscribe to queries from the system. *Analysts* publish streaming queries to the system, and also specify a query execution budget. The query execution budget can either be in the form of latency guarantees/SLAs, output quality/accuracy, or the available computing resources for query processing. Our system ensures that the computation remains within the specified budget.

At a high-level, the system works as follows: a query published by an analyst is distributed to clients via the aggregator and proxies. Clients answer the analyst's query locally over the users' private data using a privacy-preserving mechanism. Client answers are transmitted to the aggregator via anonymizing *proxies*. The *aggregator* aggregates received answers from the clients to provide privacy-preserving stream analytics to the analyst.

5.3.2 System Model

Before we explain the design of PrivApprox, we present the system model assumed in this work.

Query Model

PrivApprox supports the SQL query language for *analysts* to formulate streaming queries. While queries can be complex, the results of a query are expressed as counts within histogram buckets, i.e., each bucket represents a small range of query's answer values. Specifically, each query answer is represented in the form of binary buckets, where each bucket stores a possible answer value '1' or '0' depending on whether or not the answer falls into the value range represented by that bucket. For example, an analyst can learn the driving speed distribution across all vehicles in San Francisco by formulating an SQL query "`SELECT speed FROM vehicle WHERE location=`San Francisco'". The analyst can then define 12 answer buckets on speed: '0', '1∼10', '11∼20', $\cdots$, '81∼90', '91∼100', and '> 100'. If a vehicle is moving at 15 mph in San Francisco, it answers '1' for the third bucket and '0' for all others.

Our query model supports not only numeric queries as described above, but also non-numeric queries. For non-numeric queries, each bucket is specified by a matching rule or a regular expression. Note that, at first glance, our query model may appear simple, it however supports a range of queries such as histogram queries and frequency queries. In addition, it has been shown to be effective for a wide-range of analytics algorithms [42, 43].

Computation Model

PrivApprox adopts a *batched stream* programming model [17, 23] in which the online data stream is split into small batches; and each small batch is processed by launching a distributed data-parallel job. The batched streaming model is adopted widely compared to trigger-based systems [20, 24] for the following advantages: exact-once semantics, efficient fault-tolerance, and a common data-parallel programming model for both stream and batch analytics.

In particular, PrivApprox employs *sliding window computations* over batched stream processing [38, 39]. For sliding windows, the computation window slides over the input data stream, where the new incoming data items are added, and the old data items are dropped from the window as they become less relevant. Note that these systems [17, 23] expose a time-based window length, and based on the arrival rate, the number of data items within a window may vary accordingly.

Threat Model

Analysts are potentially malicious. They may try to violate the PrivApprox's privacy model, i.e., de-anonymize clients, build profiles through the linkage of requests and answers, or de-randomize (remove added noise from) the answers.

Clients are potentially malicious. They could generate false or invalid responses to distort the query result for the analyst. However, we do not defend against the Sybil attack [81], which is beyond the scope of this work [214].

Proxies are also potentially malicious. They may transmit messages between clients and the aggregator in contravention of the system protocols. PrivApprox includes at least two proxies, and there are at least two proxies which do not collude with each other.

The *aggregator* is assumed to be Honest-but-Curious (HbC): the aggregator faithfully conforms to the system protocol, but may try to exploit the information about clients. The aggregator does not collude with any proxy, nor the analyst.

Finally, we assume that all end-to-end communications use authenticated and confidential connections (are protected by long-lived TLS connections), and no system component could monitor all network traffic.

Privacy Properties

Our privacy properties include: (*i*) zero-knowledge privacy, (*ii*) anonymity, and (*iii*) unlinkability.

All aggregate query results in the system are independently produced under *zero-knowledge privacy* guarantees. The chosen privacy metric *zero-knowledge privacy* [101] builds upon

differential privacy [82] and provides a tighter bound on privacy guarantees compared to differential privacy. Informally, zero-knowledge privacy states that essentially everything that an adversary can learn from the output of an zero-knowledge private mechanism could also be learned using aggregate information. *Anonymity* means that no system components can associate query answers or query requests with a specific client. Finally, *unlinkability* means that no system component can join any pair of query requests or answers to the same client, even to the same anonymous client.

For the formal definitions, analysis, and proofs—refer §A.1.

Assumptions

We make the following assumptions.

1. We assume that the input stream is stratified based on the source of event, i.e., the data items within each stratum follow the same distribution, and are mutually independent. Here a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.
2. We assume the existence of a virtual function that takes the query budget as the input and outputs the sample size for each window based on the budget.
3. We assume that the aggregator faithfully follows the system protocol. We could use trusted computing such as remote attestation [195] based on Trusted Platform Modules (TPMs) to relax the HbC assumption.

We discuss different possible means to meet the first two assumptions in §5.8.

5.4 Design

PrivApprox consists of two main phases (see Figure 5.1): *submitting queries* and *answering queries*. In the first phase, an analyst submits a query (along with the execution budget) to clients via the aggregator and proxies. In the second phase, the query is answered by the clients in the reverse direction.

5.4.1 Submitting Queries

To perform statistical analysis over users' private data streams, an analyst creates a query using the query model described in §5.3.2. In particular, each query consists of the following fields, and is signed by the analyst for non-repudiation:

$$Query := \langle Q_{ID}, SQL, A[n], f, w, \delta \rangle \tag{5.1}$$

- Q_{ID} denotes a unique identifier of the query. This can be generated by concatenating the identifier of the analyst with a serial number unique to the analyst.
- SQL denotes the actual SQL query, which is passed on to clients and executed on their respective personal data.

- $A[n]$ denotes the format of a client's answer to the query. The answer is an n-bit vector where each bit associates with a possible answer value in the form of a "0" or "1" per index (or answer value range).
- f denotes the answer frequency, i.e., how often the query needs to be executed at clients.
- w denotes the window length for sliding window computations [38]. For example, an analyst may only want to aggregate query results for the last ten minutes, which means the window length is ten minutes.
- δ denotes the sliding interval for sliding window computations. For example, an analyst may want to update the query results every one minute, and so the sliding interval is set to one minute.

After forming the query, the analyst sends the query, along with the query execution budget, to the aggregator. Once receiving the pair of the query and query budget from the analyst, the aggregator first converts the query budget into system parameters for sampling (s) and randomization (p, q). We explain these system parameters in the next section §5.4.2. Hereafter, the aggregator forwards the query and the converted system parameters to clients via proxies.

5.4.2 Answering Queries

After receiving the query and system parameters, we next explain how the query is answered by clients and processed by the system to produce the result for the analyst. The query answering process involves several steps including (*i*) sampling at clients for low-latency approximation; (*ii*) randomizing answers for privacy preservation; (*iii*) transmitting answers for anonymization and unlinkability; and finally, (*iv*) aggregating answers with error estimation to give a confidence level on the approximate output. We next explain the entire workflow using these four steps. (The algorithms are detailed in §5.4.3.)

Step I: Sampling at Clients

We make use of approximate computation to achieve low-latency execution by computing over a subset of data items instead of the entire input dataset. Specifically, our work builds on sampling-based techniques [7, 103, 137, 204] in the context of "Big Data" analytics. Since we aim to keep the private data stored at individual clients, PrivApprox applies an input data sampling mechanism locally at the clients. In particular, we use *Simple Random Sampling* (SRS) [166].

Simple Random Sampling (SRS). SRS is considered as a fair way of selecting a sample from a given population since each individual in the population has the same chance of being included in the sample. We make use of SRS at the clients to select clients that will participate in the query answering process. In particular, the aggregator passes the *sampling parameter* (s) on to clients as the probability of participating in the query answering process. Thereafter, each client flips a coin with the probability based on the sampling parameter (s), and decides whether to participate in answering a query. Suppose that we have a population of U clients, and each client i has an answer a_i. We want to calculate the sum of these answers across the population, i.e., $\sum_{i=1}^{U} a_i$. To compute an approximate sum, we apply the SRS at clients to get a

sample of U' clients. The estimated sum is then calculated as follows:

$$\hat{\tau} = \frac{U}{U'} \sum_{i=1}^{U'} a_i \pm error \tag{5.2}$$

Where the error bound $error$ is defined as:

$$error = t\sqrt{\widehat{Var}(\hat{\tau})} \tag{5.3}$$

Here, t is a value of the t-distribution with $U' - 1$ degrees of freedom at the $1 - \alpha/2$ level of significance, and the estimated variance $\widehat{Var}(\hat{\tau})$ of the sum is:

$$\widehat{Var}(\hat{\tau}) = \frac{U^2}{U'}\sigma^2(\frac{U - U'}{U}) \tag{5.4}$$

Where σ^2 is the sample variance of sum.

Note that we currently assume that all clients produce the input stream with data items following the same distribution, i.e., all clients' data streams belong to the same stratum. We further extend it for stratified sampling in §5.4.4.

Step II: Answering Queries at Clients

Clients that participate in the query answering process make use of the *randomized response* technique [96] to preserve answer privacy, with *no* synchronization among clients.

Randomized response. Randomized response protects user's privacy by allowing individuals to answer sensitive queries without providing truthful answers all the time, yet it allows analysts to collect statistical results. Randomized response works as follows: suppose an analyst sends a query to individuals to obtain the statistical result about a sensitive property. To answer the query, a client locally randomizes its answer to the query [96]. Specifically, the client flips a coin, if it comes up heads, then the client responds its truthful answer; otherwise, the client flips a second coin and responds "Yes" if it comes up heads or "No" if it comes up tails. The privacy is preserved via the ability to refuse responding truthful answers.

Suppose that the probabilities of the first coin and the second coin coming up heads are p and q, respectively. The analyst receives N randomized answers from individuals, among which R_y answers are "Yes". Then, the number of original truthful "Yes" answers before the randomization process can be estimated as:

$$E_y = \frac{R_y - (1 - p) \times q \times N}{p} \tag{5.5}$$

Suppose A_y and E_y are the actual and the estimated numbers of the original truthful "Yes" answers, respectively. The accuracy loss η is then defined as:

$$\eta = \left| \frac{A_y - E_y}{A_y} \right| \tag{5.6}$$

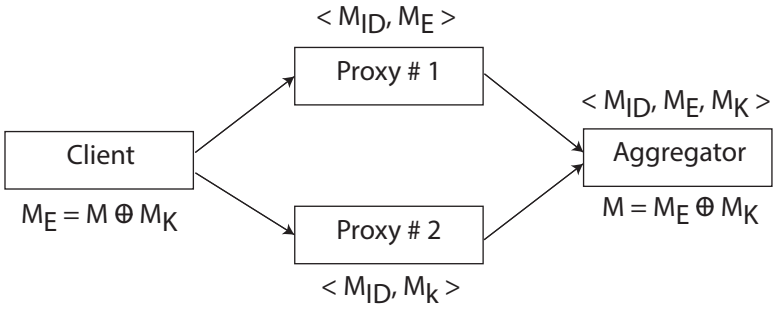

Figure 5.2 – XOR-based encryption with two proxies.

It has been proven in [83] that, the randomized response mechanism achieves ϵ-differential privacy [82], where:

$$\varepsilon = \ln\left(\frac{\Pr[Response = Yes|Truth = Yes]}{\Pr[Response = Yes|Truth = No]}\right) \tag{5.7}$$

More specifically, the randomized response mechanism achieves ϵ-differential privacy, where:

$$\varepsilon = \ln\left(\frac{p + (1-p) \times q}{(1-p) \times q}\right) \tag{5.8}$$

The reason is: if a truthful answer is "Yes", then with the probability of '$p + (1-p) \times q$', the randomized answer will still remain "Yes". Otherwise, if a truthful answer is "No", then with the probability of '$(1-p) \times q$', the randomized answer will become "Yes".

It is worth mentioning that, combining randomized response with the sampling technique used in Step I, we achieve not only differential privacy but also zero-knowledge privacy [101] which is a privacy bound tighter than differential privacy. We prove our claim in §A.1.

Step III: Transmitting Answers via Proxies

After producing randomized responses, clients transmit them to the aggregator via the proxies. To achieve anonymity and unlinkability of the clients against the aggregator and analysts, we utilize the XOR-based encryption together with source rewriting, which has been used for anonymous communications [62, 63, 79, 188]. Under the assumptions that:

- at least two proxies are not colluding
- the proxies don't collude with the aggregator, nor the analyst
- the aggregator and analyst have only a local view of the network

neither the aggregator, nor the analyst will learn any (pseudo-)identifier to deanonymize or link different answers to the same client. This property is achieved by source rewriting, which is a typical building block for anonymization schemes [79, 188]. At the same time the content of the answers is hidden from the proxies using the XOR-based encryption.

XOR-based encryption. At a high-level, the XOR-based encryption employs extremely efficient bit-wise XOR operations as its cryptographic primitive compared to expensive public-key cryptography (see Figure 5.2). This allows us to support resource-constrained clients, e.g., smartphones and sensors. The underlying idea of this encryption is simple: if Alice wants to

send a message M of length l to Bob, then Alice and Bob share a secret M_K (in the form of a random bit-string of length l). To transmit the message M privately, Alice sends an encrypted message '$M_E = M \oplus M_K$' to Bob, where '$\oplus$' denotes the bit-wise XOR operation. To decrypt the message, Bob again uses the bit-wise XOR operation: $M = M_E \oplus M_K$.

Specifically, we apply the XOR-based encryption to transmit clients' randomized answers as follows. At first, each randomized answer is concatenated with its associated query identifier Q_{ID} to build a message M:

$$M = Q_{ID}, RandomizedAnswer \tag{5.9}$$

Thereafter, the client generates $(n-1)$ random l-bit key strings M_{K_i} with $2 \leq i \leq n$ using a cryptographic pseudo-random number generator (PRNG) seeded with a cryptographically strong random number. The XOR of all $(n-1)$ key strings together forms the secret M_K.

$$M_K = \bigoplus_{i=2}^{n} M_{K_i} \tag{5.10}$$

Next, the client performs an XOR operation with M and M_K to produce an encrypted message M_E.

$$M_E = M \oplus M_K \tag{5.11}$$

As a result, the message M is split into n messages $\langle M_E, M_{K_2}, \cdots, M_{K_n} \rangle$. Afterwards, a unique message identifier M_{ID} is generated, and sent along with the split messages to the n proxies via anonymous channels enabled by source rewriting [79, 188].

$$\begin{aligned} \text{Client} &\longrightarrow \text{Proxy1} : \langle M_{ID}, M_E \rangle \\ \text{Client} &\longrightarrow \text{Proxy}i : \langle M_{ID}, M_{K_i} \rangle \end{aligned} \tag{5.12}$$

Upon receiving the messages (either $\langle M_{ID}, M_E \rangle$ or $\langle M_{ID}, M_{K_i} \rangle$) from clients, the n proxies transmit these messages to the aggregator.

The message identifier M_{ID} ensures that M_E and all associated M_{K_i} will be joined later to decrypt the original message M at the aggregator. Note that, $\langle M_{ID}, M_E \rangle$ and all $\langle M_{ID}, M_{K_i} \rangle$ are computationally indistinguishable, which hides from the proxies if the received data contains the encrypted answer or is just a pseudo-random bit string.

Step IV: Generating Result at the Aggregator

At the aggregator, all data streams ($\langle M_{ID}, M_E \rangle$ and $\langle M_{ID}, M_{K_i} \rangle$) are received, and can be joined together to obtain a unified data stream. Specifically, the associated M_E and M_{K_i} are paired by using the message identifier M_{ID}. To decrypt the original randomized message M from the client, the XOR operation is performed over M_E and M_K: $M = M_E \oplus M_K$ with M_K being the XOR of all M_{K_i}: $M_K = \bigoplus_{i=2}^{n} M_{K_i}$. As the aggregator cannot identify which of the received messages is M_E, it just XORs all the n received messages to decrypt M.

The joined answer stream is processed to produce the query results as a sliding window. For each window, the aggregator first adapts the computation window to the current start time t by removing all old data items, with $timestamp < t$, from the window. Next, the aggregator

adds the newly incoming data items into the window. Then, the answers in the window are decoded and aggregated to produce the query results for the analyst. Each query result is an estimated result which is bound to a range of error due to the approximation. The aggregator estimates this error bound using Equation 5.3 and produces a confidence interval for the result as: $queryResult \pm errorBound$. The entire process is repeated for every window.

Note that an adversarial client might answer a query many times in an attempt to distort the query result. However, we can handle this problem, for example, by applying the *triple splitting* technique [62].

Error bound estimation. We provide an error bound estimation for the aggregate query results. The accuracy loss in PrivApprox is caused by two processes: *(i)* sampling and *(ii)* randomized response. Since the accuracy loss of these two processes is statistically independent (see §5.6), we estimate the accuracy loss of each process separately. Furthermore, Equation 5.2 indicates that the error induced by sampling can be described as an additive component of the estimated sum. The error induced by randomized response is contained in the a_i values in Equation 5.2. Therefore, independent of the error induced by randomized response, the error coming from sampling is simply being added upon. Following this, we sum up both independently estimated errors to provide the total error bound of the query results.

To estimate the accuracy loss of the randomized response process, we make use of an experimental method. We run several micro-benchmarks at the beginning of the query answering process without performing the sampling process, to estimate the accuracy loss caused by randomized response. We measure the accuracy loss using Equation 5.6.

On the other hand, to estimate the accuracy loss of the sampling process, we apply the statistical theory of the sampling techniques. In particular, we first identify a desired confidence level, e.g., 95%. Then, we compute the margin of error using Equation 5.3. Note that, to use this equation the sampling distribution must be nearly normal. According to the Central Limit Theorem (CLT), when the sample size U' is large enough (e.g., ≥ 30), the sampling distribution of a statistic becomes close to the normal distribution, regardless of the underlying distribution of values in the dataset [205].

5.4.3 Algorithms

In this section, we describe the algorithmic details of PrivApprox's system protocol. We present two algorithms: *(i)* the workflow at a client carrying out sampling and randomization; and *(ii)* the workflow at the aggregator.

#I: Workflow at a client. Algorithm 8 summarizes how a client processes a query. Each client maintains its personal data in a local database. Upon receiving a query, the client first flips a sampling coin to decide whether to answer the query or not. If the coin comes up heads, then the client executes the query on its local database to create a truthful answer to the query. The truthful answer is in the form of bit buckets with a "1" or "0" per bucket, depending on whether or not the "Yes" answer falls within that bucket. The answer may have more than one bucket containing a "1" depending on the query. Next, the client randomizes the answer using the randomized response mechanism. In particular, the client flips the first randomization coin, if it comes up heads, the client responds its truthful answer. If it comes up tails, then the client flips the second randomization coin and reports the result of this coin flipping. The

Algorithm 8: Answering a query at clients

```
1  Input: Query and query budget
2  s, p, q ← costFunction(budget); // s is the sampling parameter
3  // p and q are the randomizing parameters
4  A[n] ← Query.A[n]; // Answer bit-vector
5  execute−At−Client() //Execute the method every f seconds
6  begin
7      flipResult0 ← coinFlip(s);// Flip the sampling coin
8      if flipResult0 ==“Heads” then
9          client.participate ← “True” ;
10         truthfulAnswer ← localDataProcess(Query.SQL);
11         flipResult1 ← coinFlip(p); // First randomizing coin
12         if flipResult1 ==“Heads” then
13             A[n] ← truthfulAnswer; // Process the local data
14         else
15             flipResult2 ← coinFlip(q); // Second coin
16             if flipResult2 ==“Heads” then
17                 // for all “Yes” in the bit-vector
18                 ∀i ∈ {1, ..., n}: if(A[i] == 1) A[i] ← 1;
19             else
20                 // for all “No” in the bit-vector
21                 ∀i ∈ {1, ..., n}: if(A[i] == 1) A[i] ← 0;
22         sendAnswer(A[n]); // Send the answer to the aggregator
```

randomized answer is still in the binary string format after the randomization process.

#II: Workflow at the aggregator. The aggregator receives clients' data streams from the proxies, and joins them to obtain a combined data stream. Thereafter, the aggregator processes the joined stream to produce the output for the analyst. Algorithm 9 describes the overall process at the aggregator. The algorithm computes the query results as a sliding window computation over the incoming answer stream. For each window, the aggregator first adapts the computation window to the current start time t by removing all old data items, i.e., with $timestamp < t$, from the computation window. Next, the aggregator adds the new incoming data items in the window and decrypts the answers in the data stream. Thereafter, the input data items for a window are aggregated to produce the query output for the analyst. We also estimate the error in the output due to approximation and randomization. The aggregator estimates this error bound and defines a confidence interval for the result as: $queryResult \pm errorbound$. The entire process is repeated for the next window, with the updated windowing parameters and query budget (for the adaptive execution).

5.4.4 Practical Considerations

Next, we present three design enhancements to improve the practicality of PrivApprox.

Algorithm 9: Generating query result at the aggregator

```
1  Input: w ← query.time_window; δ ← query.slide_interval;
2  t ← start time of window;
3  execute−At−Aggregator() //Execute the method every δ seconds
4  begin
5      window ← ∅; // List of items in the window
6      foreach (window w in the incoming stream ) do
7          forall elements in w do
8              if element.timestamp < t then
9                  w.remove(element); // Remove all old items
10         w ← w.insert(new items); // Add new items
11         queryResult ← ∅; // query result
12         forall answer in the sample do
13             queryAnswer ← decryptAnswer(answer);
14             // Get query results associate with analyst IDs
15             queryResult ← aggregateAnswer(queryAnswer);
16         queryResult ± error ← estimateError(queryResult);
17         t ← t + δ; // Update the start time for the next window
```

Stratified Sampling

As described in §5.4.2, we employ Simple Random Sampling (SRS) at clients for approximate computation. The assumption behind using SRS is that all clients produce data streams following the same distribution, i.e., all clients' data streams belong to the same *stratum*. However, in a distributed environment, it may happen that different clients produce data streams with disparate distributions.

Accommodating such cases requires that all strata are considered fairly to have a representative sample from each stratum. To achieve this we use the stratified sampling technique [7, 137]. Stratified sampling ensures that data from every stratum is proportionally selected (based on the arrival rate) and none of the minorities are excluded.

To perform stratified sampling, instead of just one sampling parameter s, we use a set of sampling parameters $S = \{s_i\}$ where $i \in \{1, \cdots, n\}$ (n is the number of disparate distribution sub-streams in the input stream). All clients within a given stratum i flip a sampling coin with the probability s_i to decide on the participation in the answering process. The value s_i is determined based on the proportional arrival rate of the sub-stream (or stratum). The rest of the answering process remains unchanged (as in §5.4.2).

Accordingly, we adapt the error estimation for stratified sampling to provide a confidence interval for the query result. Suppose the clients C come from n sources (disjoint strata) C_1, $C_2 \cdots, C_n$, i.e., $C = \sum_{i=1}^{n} C_i$, and the i^{th} stratum C_i has B_i clients and each such client j has an associated answer a_{ij} in binary format.

To compute an approximate sum of the "Yes" answers, we first select a sample from all clients C based on the stratified sampling, i.e., we sample b_i items from each i^{th} stratum C_i. Then we estimate the sum from this sample as: $\hat{\tau} = \sum_{i=1}^{n} (\frac{B_i}{b_i} \sum_{j=1}^{b_i} a_{ij}) \pm \epsilon$ where the error

bound ϵ is defined as:

$$\epsilon = t_{f,1-\frac{\alpha}{2}} \sqrt{\widehat{Var}(\hat{\tau})} \tag{5.13}$$

Here, $t_{f,1-\frac{\alpha}{2}}$ is the value of the t-distribution (i.e., *t-score*) with f degrees of freedom and $\alpha = 1 -$ confidence level. The degree of freedom f is calculated as:

$$f = \sum_{i=1}^{n} b_i - n \tag{5.14}$$

The estimated variance for the sum, $\widehat{Var}(\hat{\tau})$, can be expressed as:

$$\widehat{Var}(\hat{\tau}) = \sum_{i=1}^{n} B_i * (B_i - b_i) \frac{r_i^2}{b_i} \tag{5.15}$$

Here, r_i^2 is the population variance in the i^{th} stratum. Similar to the SRS described in §5.4.2, we use the statistical theories [205] for stratified sampling to calculate the error bound.

Historical Analytics

In addition to providing real-time data analytics, we further extended PrivApprox to support historical analytics. The historical analytics workflow is essential for the data warehousing setting, where analysts wish to analyze user behaviors over a longer time period. To facilitate historical analytics, we support the "batch analytics" over the users' data at the aggregator. The analyst can analyze users' responses stored in a fault-tolerance distributed storage (HDFS) at the aggregator to get the aggregate query result over the desired time period.

We further extend the adaptive execution interface for historical analytics, where the analyst can specify query execution budget, for example, to suit dynamic pricing in spot markets in the cloud deployment. Based on the query budget, we perform an additional round of sampling at the aggregator to ensure that batch analytics computation remains within the query budget. We omit the sampling details at the aggregator due to space constraints.

Query Inversion

In the current setting, some queries may result in very few "Yes" truthful answers in users' responses. For such cases, PrivApprox can only achieve lower utility as the fraction of truthful "Yes" answers gets far from the second randomization parameter q (see experimental results in §5.6.4). For instance, if q is set to a high value (e.g., $q = 0.9$), having few "Yes" answers in the user responses will affect the overall utility of the query result.

To address this issue, we propose a *query inversion* mechanism. If the fraction of truthful "Yes" answers is too small or too large compared to the q value, then the analysts can invert the query to calculate the truthful "No" answers instead of the truthful "Yes" answers. In this way, the fraction of truthful "No" answers gets closer to q, resulting in a higher utility of the query result.

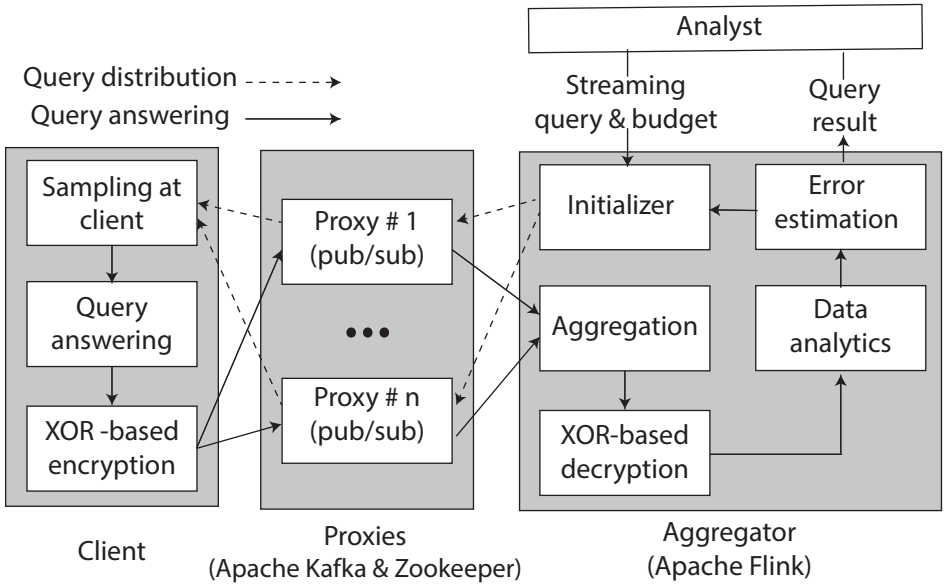

Figure 5.3 – Architecture of PrivApprox prototype.

5.5 Implementation

We implemented PrivApprox as an end-to-end stream analytics system. Figure 5.3 presents the architecture of our prototype. Our system implementation consists of three main components: (*i*) clients, (*ii*) proxies, and (*iii*) the aggregator.

First, the query and the execution budget specified by the analyst are processed by the *i*nitializer module to decide on the sampling parameter (s) and the randomization parameters (p and q). These parameters along with the query are then sent to the clients.

Clients. We implemented Java-based clients for mobile devices as well as for personal computers. A client makes use of the sampling parameter (based on the *s*ampling module) to decide whether to participate in the query answering process (§5.4.2). If the client decides to participate then the *q*uery answer module is used to execute the input query on the local user's private data stored in SQLite [203]. The client makes use of the randomized response to execute the query (§5.4.2). Finally, the randomized answer is encrypted using the XOR-based encryption module; thereafter, the encrypted message and the key messages are sent to the aggregator via proxies (§5.4.2).

Proxies. We implemented proxies based on Apache Kafka (which internally uses Apache Zookeeper [25] for fault tolerance). In Kafka, a *topic* is used to define a stream of data items. A stream *producer* can publish data items to a topic, and these data items are stored in Kafka servers called *brokers*. Thereafter, a *consumer* can subscribe to the topic and consume the data items by pulling them from the brokers. In particular, we make use of Kafka APIs to create two main topics: *key* and *answer* for transmitting the key message stream and the encrypted answer stream in the XOR-based encryption protocol, respectively (§5.4.2).

Aggregator. We implemented the aggregator using Apache Flink for real-time stream analytics and also for historical batch analytics. At the aggregator, we first make use of the join method (using the *a*ggregation module) to combine the two data streams: (*i*) encrypted answer stream

Table 5.1 – Utility and privacy of query results with different randomization parameters p and q.

p	q	Accuracy loss (η)	Privacy Level (ϵ)
0.3	0.3	0.0278	1.7047
	0.6	0.0262	1.3862
	0.9	0.0268	1.2527
0.6	0.3	0.0141	2.5649
	0.6	0.0128	2.0476
	0.9	0.0136	1.7917
0.9	0.3	0.0098	4.1820
	0.6	0.0079	3.5263
	0.9	0.0102	3.1570

and (*ii*) key stream. Thereafter, the combined message stream is decoded (using the XOR-based decryption module) to reproduce the randomized query answers. These answers are then forwarded to the analytics module. The *a*nalytics module processes the answers to provide the query result to the analyst. Moreover, the *e*rror estimation module is used to estimate the error (§5.4.2), which we implemented using the Apache Common Math library. If the error exceeds the error bound target, a feedback mechanism is activated to re-tune the sampling and randomization parameters to provide higher utility in the subsequent epochs.

For the historical analytics, we asynchronously store the (randomized responses) data in HDFS [45] at the aggregator (as a separate pipeline, which is not shown in Figure 5.3 for simplicity). To support historical analytics on the stored data at the aggregator, we also implemented a sampling method *sample()* in Flink to support our sampling mechanism (§5.4.4).

5.6 Evaluation

In this section, we evaluate PrivApprox using a series of micro-benchmarks. For all microbenchmark measurements, we report the average over 100 runs.

5.6.1 Varying Sampling and Randomization Parameters

We first measure the effect of randomization parameters on the utility and the privacy guarantee of the query results. In particular, the utility is measured by the query results' accuracy loss (Equation 5.6), and privacy is measured by the level of achieved zero-knowledge privacy (Equation A.4). For the experiment, we generated $10,000$ original answers randomly, 60% of which are "Yes" answers. The sampling parameter s is set to 0.6.

Table 5.1 shows that different settings of the two randomization parameters, p and q, do affect the utility and the privacy guarantee of the query results. The higher p means the higher probability that a client responds with its truthful answer. As expected, this leads to higher utility (i.e., smaller accuracy loss η) but weaker privacy guarantee (i.e., higher privacy level ϵ). In addition, Table 5.1 also shows that the closer we set the probability q to the fraction of truthful "Yes" answers (i.e., 60% in this microbenchmark), the higher utility the query result provides. Nevertheless, to meet the utility and privacy requirements in various scenarios, we should carefully choose the appropriate p and q. In practice, the selection of the ϵ value

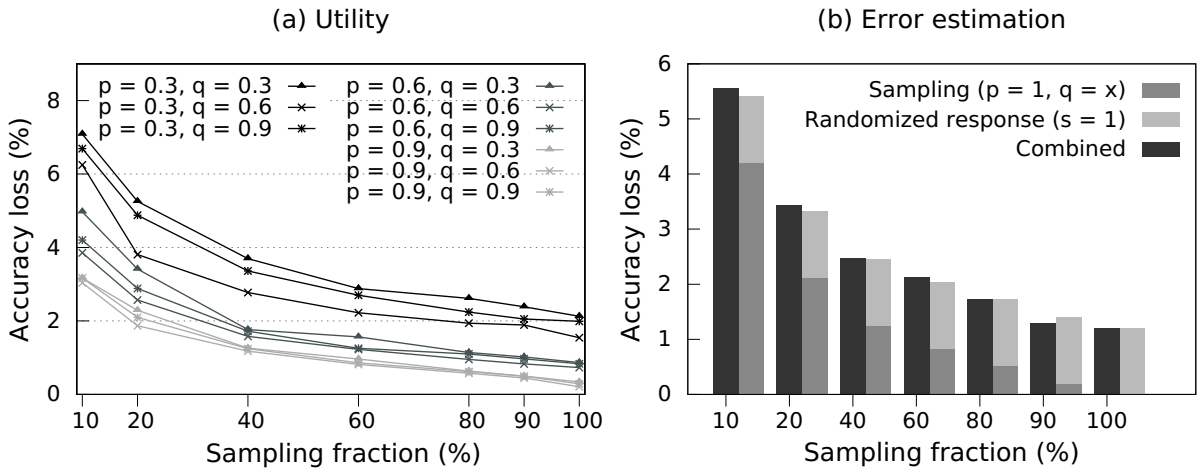

Figure 5.4 – (a) Accuracy loss with varying sampling and randomization parameters. (b) Error estimation during the randomized response process and sampling process, combined and individually.

depends on real-world applications [140].

We also measured the effect of sampling parameter on the accuracy loss. Figure 5.4 (a) shows that the accuracy loss decreases with the increase of sampling fraction, regardless of the settings of randomization parameters p and q. The benefits reach diminishing returns after the sampling fraction of 80%. The system operator can set the sampling fraction using resource prediction model [218–220] for any given SLA.

5.6.2 Error Estimation

To analyze the accuracy loss, we first measured the accuracy loss caused by sampling and randomized response *separately*. For comparison, we also computed the total accuracy loss after running the two processes in succession as in PrivApprox. In this experiment, we set the number of original answers to $10,000$ with 60% of which being "Yes" answers. We measure the accuracy loss of the randomized response process by setting the sampling parameter to 100% ($s = 1$) and the randomization parameters p and q to 0.3 and 0.6, respectively. Meanwhile, we measure the accuracy loss of the sampling process without the randomized response process by setting p to 1.

Figure 5.4 (b) represents that the accuracy loss during the two experiments is statistically independent to each other. In addition, the accuracy loss of the two processes can effectively be added together to calculate the total accuracy loss.

5.6.3 Varying Number of Clients

We next analyzed how the number of participating clients affects the utility of the results. In this experiment, we fix the sampling and randomization parameters s, p and q to 0.9, 0.9 and 0.6, respectively, and set the fraction of truthful "Yes" answers to 60%.

Figure 5.5 (a) shows that the utility of query results improves with the increase of the

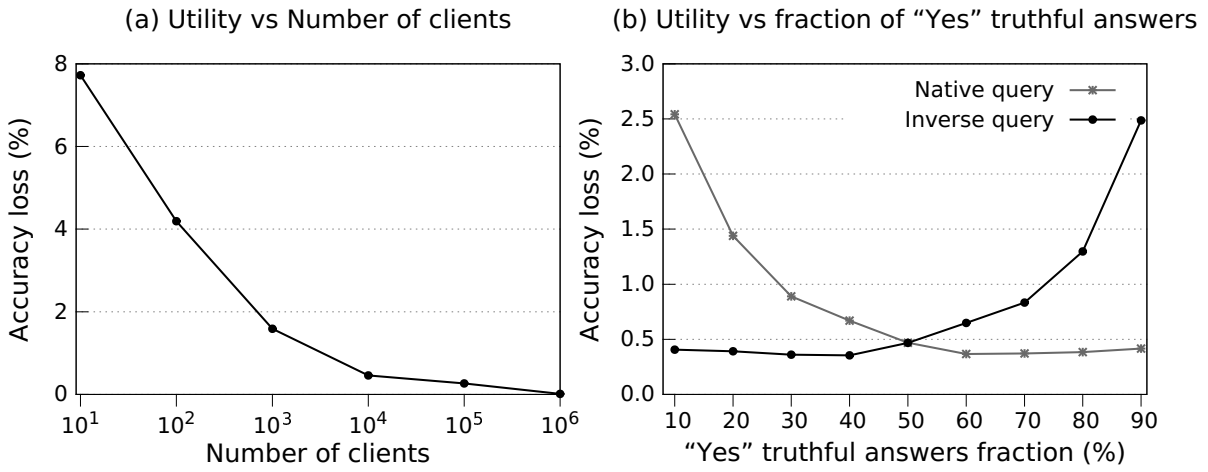

Figure 5.5 – (a) Accuracy loss with varying # of clients. (b) Accuracy loss for the native and inverse query results with different fractions of truthful "Yes" answers.

Table 5.2 – Comparison of crypto overheads (# operations/sec). The public-key crypto schemes use a 1024-bit key.

	Encryption						Decryption					
	Phone		Laptop		Server		Phone		Laptop		Server	
RSA [10]	937	16×	2,770	341×	4,909	275×	126	25890×	698	23666×	859	26401×
Goldwasser [63]	2,106	7×	17,064	55×	22,902	59×	127	25686×	6,329	2610×	7,068	3209×
Paillier [187]	116	129×	489	1930×	579	2335×	72	45308×	250	66076×	309	73392×
PrivApprox	15,026		943,902		1,351,937		3,262,186		16,519,076		22,678,285	

number of participating clients, and few clients (e.g., < 100) may lead to low-utility query results.

Note that increasing the number of participating clients leads to higher network overheads. However, we can tune the number of clients using the sampling parameter s and thus decrease the network overhead (see §5.7.3).

5.6.4 Varying Fraction of Truthful Answers

We also measured the utility of both the native and the inverse query results with different fractions of truthful "Yes" answers. For the experiment, we still keep the sampling and randomization parameters s, p and q to 0.9, 0.9 and 0.6, respectively, and set the total number of answers to $10,000$.

Figure 5.5 (b) shows that PrivApprox achieves higher utility as the fraction of truthful "Yes" answers gets closer to 60% (i.e., the q value). In addition, when the fraction of truthful "Yes" answers y is too small compared to the q value (e.g., $y = 0.1$), the accuracy loss is quite high at 2.54%. However, by using the query inversion mechanism (§5.4.4), we can significantly reduce the accuracy loss to 0.4%.

Table 5.3 – Throughput (# operations/sec) at clients

No. of operations/sec	Phone	Laptop	Server
SQLite read	1,162	19,646	23,418
Randomized response	168,938	418,668	1,809,662
XOR encryption	15,026	943,902	1,351,937
Total	1,116	17,236	22,026

5.6.5 Varying Answer's Bit-vector Sizes

We measured the throughput at proxies with various bit-vector sizes of client answers (i.e., $A[n]$ in §5.4.1). We conducted this experiment with a 3-node cluster (see §5.7.1 for the experimental setup). Figure 5.6 (a) shows that the throughput, as expected, is inversely proportional to the answer's bit-vector sizes.

5.6.6 Effect of stratified sampling

To illustrate the use of stratified sampling, we generated a synthetic data stream with three different stream sources S_1, S_2, S_3. Each stream source is created with an independent Poisson distribution. In addition, the three stream sources have an arrival rate of $3 : 4 : 5$ data items per time unit, respectively. The computation window size is fixed to $10,000$ data items.

Figure 5.6 (b) shows the average number of selected items of each stream source with varying sample fractions using the stratified sampling mechanism. As expected, the average number of sampled data items from each stream source is proportional to its arrival rate and the sample fractions.

5.6.7 Computational Overhead of Crypto Operations

We compared the computational overhead of crypto operations used in PrivApprox and prior systems. In particular, these crypto operations are XOR in PrivApprox, RSA in [10], Goldwasser-Micali in [63], and Paillier in [187]. For the experiment, we measured the number of crypto operations that can be executed on: (*i*) Android Galaxy mini III smartphone running Android 4.1.2 with a 1.5 GHz CPU; (*ii*) MacBook Air laptop with a 2.2 GHz Intel Core i7 CPU running OS X Yosemite 10.10.2; and (*iii*) Linux server running Linux 3.15.0 equipped with a 2.2 GHz CPU with 32 cores.

Table 5.2 shows that the XOR operation is extremely efficient compared with the other crypto mechanisms. This highlights the importance of XOR encryption in our design.

5.6.8 Throughput at Clients

We measured the throughput at clients. In particular, we measured the number of operations per second that can be executed at clients for the query answering process. In this experiment, we used the same set of devices as in the previous experiment.

Table 5.3 presents the throughput at clients. To further investigate the overheads, we measured the individual throughput of three sub-processes in the query answering process:

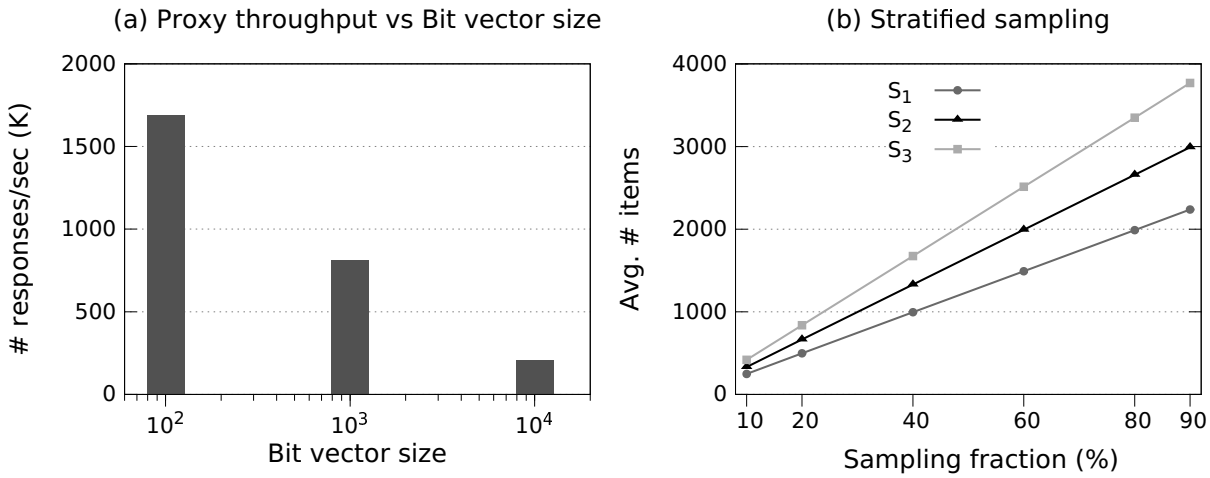

Figure 5.6 – (a) Throughput of proxies with different bit-vector sizes for the query answer. (b) Average number of sampled data items after stratified sampling with different sampling fractions.

(*i*) database read, (*ii*) randomized response, and (*iii*) XOR encryption. The result indicates that the performance bottleneck in the answering process is actually the database read operation.

5.6.9 Comparison with Related Work

First, we compared PrivApprox with SplitX [62], a high-performance privacy-preserving analytics system. We compare the latency incurred at proxies in both PrivApprox and SplitX. SplitX is geared towards batch analytics, but can be adapted to enable privacy-preserving data analytics over data streams. Since PrivApprox and SplitX share the same architecture, we compare the latency incurred at proxies in both systems.

Figure 5.7 (a) shows that, with different numbers of clients, the latency incurred at proxies in PrivApprox is always nearly one order of magnitude lower than that in SplitX. The reason is simple: unlike PrivApprox, SplitX requires synchronization among its proxies to process query answers in a privacy-preserving fashion. This synchronization creates a significant delay in processing query answers, making SplitX unsuitable for dealing with large-scale stream analytics. More specifically, in SplitX, the processing at proxies consists of a few sub-processes including adding noise to answers, answer transmission, answer intersection, and answer shuffling; whereas, in PrivApprox, the processing at proxies contains only the answer transmission. Figure 5.7 (a) also shows that with 10^6 clients, the latency at SplitX is 40.27 sec, whereas PrivApprox achieves a latency of just 6.21 sec, resulting in a $6.48\times$ speedup compared with SplitX.

Next, we compared PrivApprox with a recent privacy-preserving analytics system called RAPPOR [211]. Similar to PrivApprox, RAPPOR applies a randomized response mechanism to achieve differential privacy. However, RAPPOR is not designed for stream analytics, and therefore, we compared PrivApprox with RAPPOR for privacy only. To make an "apple-to-apple" comparison between PrivApprox and RAPPOR in terms of privacy, we make a mapping between the system parameters of the two systems. We set the sampling parameter $s = 1$,

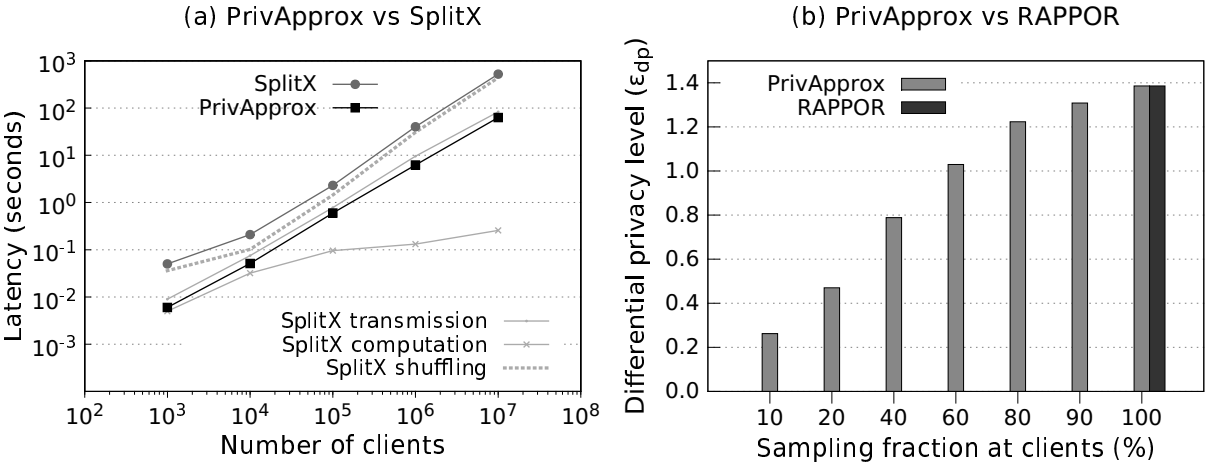

Figure 5.7 – (a) Latency comparison b/w SplitX (batch analytics) and PrivApprox (stream analytics). (b) Differential privacy level comparison b/w RAPPOR and PrivApprox.

and the randomized parameters $p = 1 - f$, $q = 0.5$ in PrivApprox, where f is the parameter used in the randomized response process of RAPPOR [211]. In addition, we set the number of hash functions used in RAPPOR to 1 ($h = 1$) for a fair comparison. In doing so, the two systems have the same randomized response process. However, since PrivApprox makes use of the sampling mechanism before performing the randomized response process, PrivApprox achieves stronger privacy. Figure 5.7 (b) shows the differential privacy level of RAPPOR and PrivApprox with different sampling fractions s.

It is worth mentioning that, by applying the sampling mechanism, PrivApprox achieves stronger privacy (i.e., zero-knowledge privacy) for clients. The comparison between differential privacy and zero-knowledge privacy is presented in §A.1.

Recently, several privacy-preserving stream analytics systems have been proposed [111, 215, 217]. These systems make use of the Laplace mechanism [82, 84] to achieve differential privacy. In particular, they add Laplace noise to the truthful answers *at the aggregator* to protect the users' privacy. However, their approach relies on strong trust assumptions of the aggregator as well as the connection between clients and the aggregator. On the contrary, PrivApprox applies randomized response mechanism to process users' private data locally at clients under the control of users. Combined with the sampling mechanism, PrivApprox achieves stronger privacy guarantees (with a tighter bound for ϵ_{dp}-differential privacy and ϵ_{zk}-zero-knowledge privacy).

5.7 Case Studies

We next present our experience using PrivApprox in the following two case studies: (*i*) New York City (NYC) taxi ride, and (*ii*) household electricity consumption.

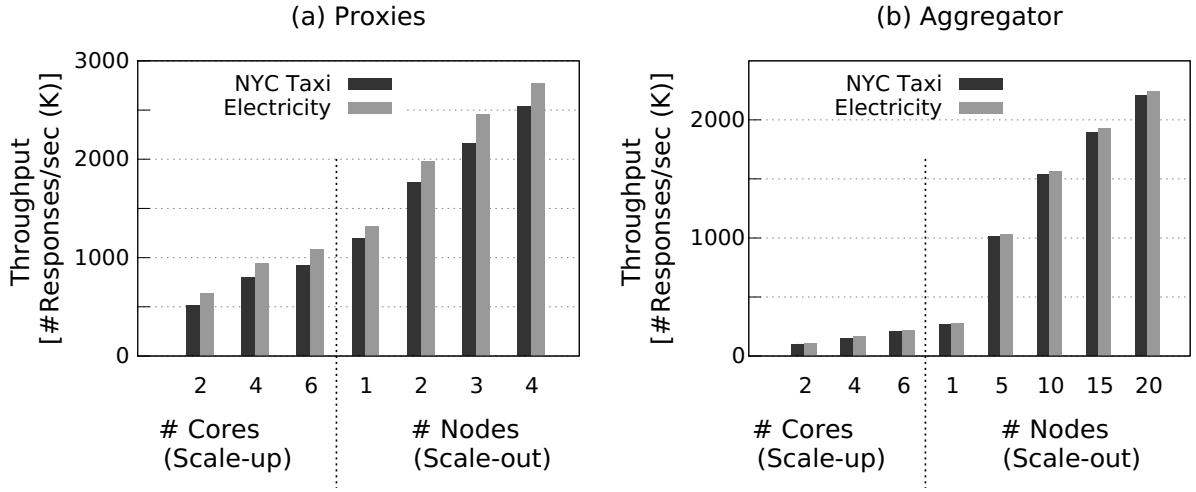

Figure 5.8 – Throughput at proxies and the aggregator with different numbers of CPU cores and nodes.

5.7.1 Experimental Setup

Cluster setup. We used a cluster of 44 nodes connected via a Gigabit Ethernet. Each node contains 2 Intel Xeon quad-core CPUs and 8 GB of RAM running Debian 5.0 with Linux kernel 2.6.26. We deployed two proxies with Apache Kafka, each of which consists of 4 Kafka broker nodes and 3 Zookeeper nodes. We used 20 nodes to deploy Apache Flink as the aggregator. In addition, we employed the remaining 10 nodes to replay the datasets to generate data streams for evaluating our PrivApprox system.

Datasets. For the first case study, we used the NYC *Taxi Ride* dataset from the DEBS 2015 Grand Challenge [129]. The dataset consists of the itinerary information of all rides across 10,000 taxies in New York City in 2013. For the second case study, we used the *Household Electricity Consumption* dataset [190]. This dataset contains electricity usage (kWh) measured every 30 minutes for one year by smart meters.

Queries. For the NYC taxi ride case-study, we created a query: "*What is the distance distribution of taxi trips in New York?*". We defined the query answer with 11 buckets as follows: [0, 1) mile, [1, 2) miles, [2, 3) miles, [3, 4) miles, [4, 5) miles, [5, 6) miles, [6, 7) miles, [7, 8) miles, [8, 9) miles, [9, 10) miles, and [10, $+\infty$) miles.

For the second case-study, we defined a query to analyze the electricity usage distribution of households over the past 30 minutes. The query answer format is as follows: [0, 0.5] kWh, (0.5, 1] kWh, (1, 1.5] kWh, (1.5, 2] kWh, (2, 2.5] kWh, and (2.5, 3] kWh.

Evaluation metrics. We evaluated our system using four key metrics: throughput, latency, utility, and privacy level. *Throughput* is defined as the number of data items processed per second, and *latency* is defined as the total amount of time required to process a certain dataset. *Utility* is the accuracy loss defined as $|\frac{estimate-exact}{exact}|$, where $estimate$ and $exact$ are the query results produced by applying PrivApprox and the native computation, respectively. Finally, *privacy level* (ϵ_{zk}) is calculated using Equation A.4. For all measurements, we report the average over 10 runs.

5.7.2 Scalability

We measured the scalability of the two main system components: proxies and the aggregator. We first measured the throughput of proxies with various numbers of CPU cores (scale-up) and different numbers of nodes (scale-out). This experiment was conducted on a cluster of 4 nodes. Figure 5.8 (a) shows that, as expected, the throughput at proxies scales quite well with the number of CPU cores and nodes. In the NYC Taxi case-study, with 2 cores, the throughput of each proxy is $512,348$ answers/sec, and with 8 cores (1 node) the throughput is $1,192,903$ answers/sec; whereas, with a cluster of 4 nodes each with 8 cores, the throughput of each proxy reaches $2,539,715$ answers/sec. In the household electricity case-study, the proxies achieve relatively higher throughput because the message size is smaller than in the NYC Taxi case-study.

We next measured the throughput at the aggregator. Figure 5.8 (b) depicts that the aggregator also scales quite well when the number of nodes for aggregator increases. The throughput of the aggregator, however, is much lower than the throughput of proxies due to the relatively expensive join operation and the analytical computation at the aggregator. We notice that the throughput of the aggregator in the household electricity case study does not significantly improve in comparison to the first case study. This is because the difference in the size of messages between the two case studies does not affect much the performance of the join operation and the analytical computation.

5.7.3 Network Bandwidth and Latency

Next, we conducted the experiment to measure the network bandwidth usage. By leveraging the sampling mechanism at clients, our system reduces network traffic significantly. Figure 5.9 (a) shows the total network traffic transferred from clients to proxies with different sampling fractions. In the first case study, with the sampling fraction of 60%, PrivApprox can reduce the network traffic by $1.62\times$; whereas in the second case study, the reduction is $1.58\times$.

Besides the benefit of saving network bandwidth, PrivApprox achieves also lower latency in processing query answers by leveraging approximate computation. To evaluate this advantage, we measured the effect of sampling fractions on the latency of processing query answers. Figure 5.9 (b) depicts the latency with different sampling fractions at clients. For the first case-study, with the sampling fraction of 60%, the latency is $1.68\times$ lower than the execution without sampling; whereas, in the second case-study this value is $1.66\times$ lower than the execution without sampling.

5.7.4 Utility and Privacy

Figure 5.10 (a)(b)(c) show the utility, the privacy level, and the trade-off between them, respectively, with different sampling and randomization parameters. The randomization parameters p and q vary in the range of (0, 1), and the sampling parameter s is calculated using Equation A.4. Here, we show results only for NYC Taxi dataset. As the sampling parameter s and the first randomization parameter p increase, the utility of query results improves (i.e., accuracy loss gets smaller) whereas the privacy guarantee gets weaker (i.e., privacy level gets higher). Since the New York taxi dataset is diverse, the accuracy loss and the privacy level change in a

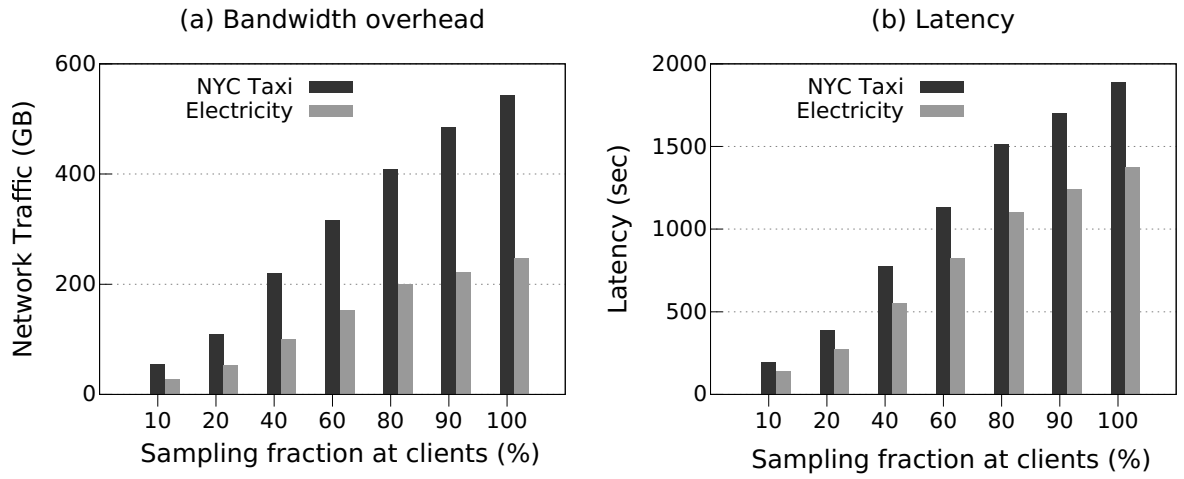

Figure 5.9 – Total network traffic and latency at proxies with different sampling fractions at clients.

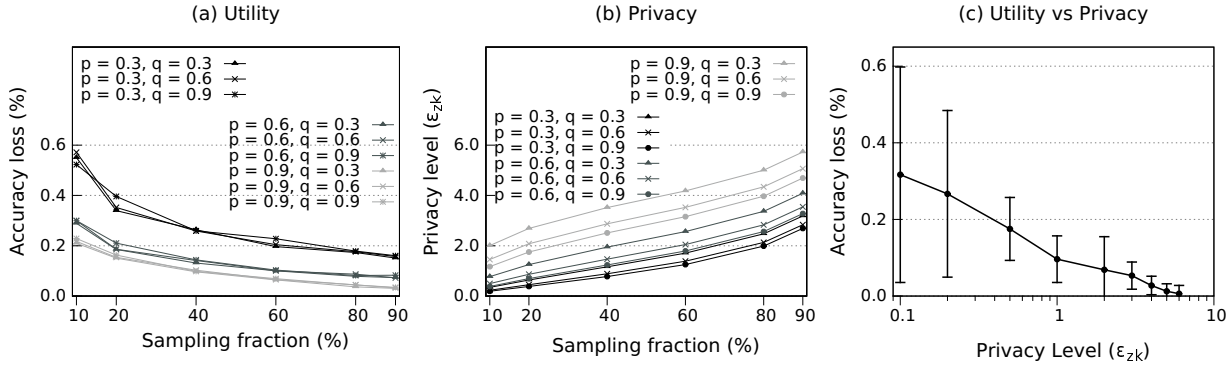

Figure 5.10 – Results from the NYC taxi case-study with varying sampling and randomization parameters: (a) Utility, (b) Privacy level, (c) Comparison between utility and privacy.

non-linear fashion with different sampling fractions and randomization parameters. Interestingly, the accuracy loss does not always decrease as the second randomization parameter q increases. The accuracy loss gets smaller when $q = 0.3$. This is due to the fact that the fraction of truthful "Yes" answers in the dataset is 33.57% (close to $q = 0.3$).

5.7.5 Historical Analytics

To analyze the performance of PrivApprox for historical analytics, we executed the queries on the datasets stored at the aggregator. Figure 5.11 (a) (b) present the latency and throughput, respectively, of processing historical datasets with different sampling fractions. We can achieve a speedup of $1.86\times$ over native execution in historical analytics by setting the sampling fraction to 60%.

We also measured the accuracy loss when the approximate computation was applied (for the NYC Taxi case-study). Figure 5.11 (c) shows the accuracy loss in processing historical data with different sampling fractions. With the sampling fraction of 60%, the accuracy loss is only

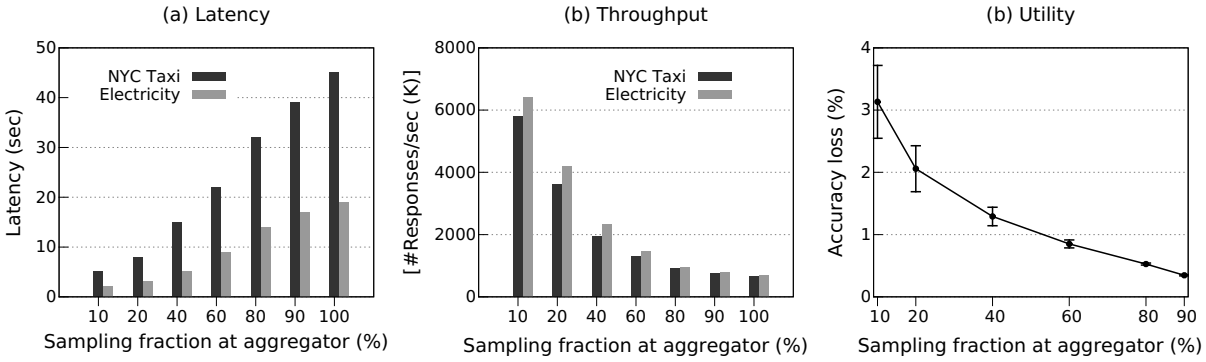

Figure 5.11 – Historical analytics results with varying sampling fractions: (a) Latency, (b) Throughput, and (c) Utility.

less than 1%.

5.8 Discussion

In this section, we discuss some approaches that could be used to meet our assumptions listed in §5.3.2.

Stratified sampling. In our design in §5.4, we currently assume that the input stream is already stratified based on the source of event, i.e., the data items within each stratum follow the same distribution. However, it may not be the case in real-world applications. As also discussed in previous chapters, we next describe two proposals for the stratification of evolving data streams, namely bootstrap [87, 88, 175] and semi-supervised learning [157].

Bootstrap [87, 88, 175] is a well-studied non-parametric sampling technique in statistics for the estimation of distribution for a given population. In particular, the bootstrap method randomly selects "bootstrap samples" with replacement to estimate the unknown parameters of a population; for instance, by averaging the bootstrap samples. We can employ a bootstrap-based estimator for the stratification of incoming sub-streams. Alternatively, we could also make use of a semi-supervised algorithm [157] to stratify a data stream. The advantage of the algorithm is that it can work with both the labeled and unlabeled data stream to train a classification model.

Virtual cost function. Currently, in our implementation described in §5.5, for a given user-specified query budget about privacy ϵ_{zk}, the sampling and randomizing parameters can be computed using the reversed function of Equation A.4. However, for the query budget involving available computing resources or latency requirements (SLAs)—we currently assume that there exists a virtual function that determines the sampling parameter based on the query budget. As also discussed in previous chapters, we recommend two existing approaches—Pulsar [16] and resource prediction model [99, 155]—to design and implement such a virtual function for the given computing resources and latency requirements, respectively.

Pulsar [16] is a "virtual datacenter (VDC)" system that allows users to allocate resources based on tenants' demand. The system proposes a multi-resource token bucket algorithm that uses a pre-advertised cost model for supporting workload independent guarantees. We could apply a similar cost model based on Pulsar as follows: A data item to be processed could

be considered as a request, and "amount of resources" needed to process these items could be the cost in tokens. Since the resource budget gives total resources (*here tokens*) to be used, we could find the number of clients, i.e., the sampling fraction at clients, that can be processed using these resources.

To find the sampling parameter for a given latency budget, we could use a resource prediction model [218–220]. The resource prediction model could build by analyzing the diurnal patterns of resource usage [57], and predicts the resource requirement to meet SLAs leveraging statistical machine learning [99, 155]. Once we have the resource requirement in place to meet a given SLA—we can find the appropriate sampling parameter by using the above suggested method similar to Pulsar.

5.9 Related Work

Privacy-preserving analytics. Since the notion of differential privacy [82, 84], a plethora of systems have been proposed to provide differential privacy with centralized trusted databases supporting linear queries [145], graph queries [134], histogram queries [118], Airavat-MapReduce [189], SQL-type PINQ queries [159, 160, 179] and even general programs, such as GUPT [165] and Fuzz [116]. In practice, however, such central trust can be abused, leaked, or subpoenaed [69, 120, 178, 196].

To overcome the limitations of the centralized database schemes, recently a flurry of systems have been proposed with a focus on achieving users' privacy (mostly, differential privacy) in a distributed setting where the private data is kept locally. Examples include Privad [108], PDDP [63], DJoin [169], SplitX [62], πBox [141], KISS [212], Koi [109], xBook [201], Popcorn [113], and many other systems [10, 86, 117]. However, these systems are designed to deal with the "one-shot" batch queries only, whereby the data is assumed to be static during the query execution.

To overcome the limitations of the aforementioned systems, several differentially private stream analytics systems have been proposed recently [55, 56, 85, 97, 124, 187, 198]. Unfortunately, these systems still contain several technical shortcomings that limit their practicality. One of the first systems [85] updates the query result only if the user's private data changes significantly, and does not support stream analytics over an unlimited time period. Subsequent systems [55, 124] remove the limit on the time period, but introduce extra system overheads. Some systems [187, 198] leverage expensive secret sharing cryptographic operations to produce noisy aggregate query results. These protocols, however, cannot work at large scale under churn; moreover, in these systems, even a single malicious user can substantially distort the aggregate results without detection. Recently, some other privacy-preserving distributed stream monitoring systems have been proposed [56, 97]. However, they all require some form of synchronization, and are tailored for heavy-hitter monitoring only. Streaming data publishing systems like [215] use a stream-privacy metric at the cost of relying on a trusted party to add noise. In contrast, PrivApprox does not require a trusted proxy or aggregator to add noise. Furthermore, PrivApprox provides stronger privacy properties (zero-knowledge privacy).

Sampling and randomized response. Sampling and randomized response, also known as *input perturbation* techniques, are being studied in the context of privacy-preserving analytics,

albeit they are explored separately. For instance, the relationship between *sampling* and privacy is being investigated to provide k-anonymity [58], differential privacy [165], and crowd-blending privacy [102]. In contrast, we show that sampling combined with randomized response achieves the zero-knowledge privacy, a privacy bound strictly stronger that the state-of-the-art differential privacy. Furthermore, PrivApprox achieves these guarantees for stream processing with a distributed private dataset.

Randomized response [96, 216] is a surveying technique in statistics, since 1960s, for collecting sensitive information via input perturbation. Recently, Google, in a system called RAPPOR [211], made use of randomized response for privacy-preserving analytics for the Chrome browser. RAPPOR provides differential privacy (ϵ_{dp}) for clients while enabling analysts to collect various types of statistics. Like RAPPOR, PrivApprox utilizes randomized response. However, RAPPOR is designed for heavy-hitter collection, and does not deal with the situation where clients' answers to the same query are changing over time. Therefore, RAPPOR does not fit well with the stream analytics. Furthermore, since we combine randomized response with sampling, PrivApprox (ϵ_{zk}) provides a privacy bound tighter than RAPPOR (ϵ_{dp}).

Secure multi-party computation. In theory, secure multi-party computation (SMC) [104, 223] could be used for privacy-preserving analytics. It is, however, expensive for real-world deployment, especially for stream analytics, even though there have been several proposals reducing SMC's computational overhead [107, 128, 148, 149, 174, 221]. Furthermore, SMC guarantees input-privacy during computation, but is orthogonal to output-privacy as provided by differential privacy.

Approximate computing. Approximation techniques such as sampling [11, 59, 100], sketches [73], and online aggregation [121] have been well-studied over the decades in the databases community. Recently, sampling-based systems (such as ApproxHadoop [103], BlinkDB [7, 8], IncApprox [137], Quickr [204], StreamApprox [184]) and online aggregation-based systems (such as MapReduce Online [70, 171], G-OLA [230]) have also been shown effective for "Big Data" analytics.

We build on the advancements of sampling-based techniques. However, we differ in two crucial aspects. First, we perform sampling in a distributed way as opposed to sampling in a centralized dataset. Second, we extend sampling with randomized response for privacy-preserving analytics.

5.10 Conclusion

In this chapter, we presented PrivApprox, a privacy-preserving stream analytics system. Our approach builds on the observation that both computing paradigms — privacy-preserving data analytics and approximate computation — strive for approximation, and can be combined together to leverage the benefits of both. Our evaluation shows that PrivApprox not only improves the performance to support real-time stream analytics, but also achieves provably stronger privacy (i.e., zero-knowledge privacy) guarantees than the state-of-the-art differential privacy.

As mentioned before, the content of this chapter is based on our conference publication [183] and our technical report [182]. PrivApprox's source code is publicly available: https://PrivApprox.github.io.

6 ApproxJoin: Approximate Distributed Joins

In PrivApprox, to achieve the anonymity property, the system needs to split the original messages into two pieces and send to the aggregator to perform a join operation between two input data streams to decrypt the user input stream data. Unfortunately, this join operation is a bottleneck of the system. This motivated us to conduct the fourth work in this thesis to improve the performance of join operations. We start our work by building a system for joins between input static datasets since it naturally can be extended for a batch based stream processing system.

In this chapter, we present the design, implementation, and evaluation of ApproxJoin, an approximate distributed join system. ApproxJoin allows us to mitigate the overhead of joins in big data analytics by adopting approximate computing techniques. We implemented ApproxJoin combining a sketching technique (Bloom filter) and a sampling technique (stratified sampling) to reduce the shuffled data size as well as latency during the join operations.

This chapter is organized as follows. We first present the motivation behind ApproxJoin in Section §6.1. We next describe the contributions of ApproxJoin in Section §6.2. After that, we give the overview of ApproxJoin in Section §6.3. Next, we describe in details the design of ApproxJoin in Section §6.4. Thereafter, we present the implementation of ApproxJoin based on the design in Section §6.5. We next describe an experimental evaluation of ApproxJoin using micro-benchmarks in Section §6.6 and two real world case studies in Section §6.7. Then, we present the discussion and the related work in Section §6.8 and Section §6.9, respectively. Finally, we conclude our work in this chapter in Section §6.10.

This content of this chapter is based on our under-submission paper. This work is based on a joint collaboration with Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe.

6.1 Motivation

Over the last ten years, the growth of publicly available digital data has accelerated exponentially. To store this large amounts of data, analytical data repositories called "*data lakes*" have been used instead of the DBMS. While the publicly raw data is huge and valuable, extracting useful knowledge from the lake of data is really a challenge since it requires data analytics systems to scale and adapt to handle the vast amount of data with great variety of data source and types. Therefore, to extract valuable insights from the raw data, modern online services make use of big data systems based on parallel and distributed frameworks such as Hadoop [19], Apache Spark [21], and Apache Flink [17].

Since in data lakes, the raw data comes from many sources, to pluck the valuable information for the sea of data, the big data systems need to combine the data from many sources and perform analytics on the merged data. Thus, *join* operations become a critical building block

in these systems. While joins are important, they are expensive, especially with regard to communication costs. For example, commercial database systems could take hours or even days to process complex join queries [146]. Therefore, improving the performance of join operations can drastically reduce the overall latency of the big data systems.

One way to achieve this is to simply throw more computing resources, which comes at significant computing cost. A new and better way is to leverage *approximate computing*. As also mentioned in previous chapters, approximate computing is based on the observation that for many real-world applications it is sufficient to produce approximate rather than exact results [80, 170]. For such applications, it is possible to compute over a partial subset instead of the entire massive of data, achieving lower latency and efficient resource utilization.

Over the last decade, approximate computing has been applied in various domains such as programming languages [29, 194], hardware design [193], query processing [7, 59, 200], and distributed systems [70, 121, 171]. Various approximation techniques have been proposed to make trade-offs between required resources and output quality, including sampling [11, 100], sketches [73], and online aggregation [121, 171].

Sampling techniques have been widely applied for distributed data analytics [7, 8, 146, 204]. Moreover, the same techniques can be adopted to reduce the overhead of join operations [146, 204]. However, previous work requires intricate knowledge of the joined tables [6, 7, 204] or requires to perform off-line sampling [7] and take a relevant sample for each query. Wander Join [146] provides a mechanism to perform online-sampling over join using random walks. However, it still needs to handle non-join data items from input datasets (null vertices in the join data graph). Furthermore, Wander Join supports only single node settings, but not distributed system environments.

To understand the benefits and challenges of approximate join, consider joining two datasets R and S that share a join attribute A. The inner-join of R and S is the subset of the cross-product of these datasets containing pairs (r, s) such that $r \in R, s \in S$, and $r.A = s.A$. The complexity of a simple hash join is $O(|R| + |S|)$, and $O(|R| \times |S|)$ for a simple nested loop join. Therefore, sampling R and S before the join operation would have significant performance benefits.

However, obtaining approximate join results by sampling is an inherently difficult problem from the correctness perspective. We illustrate this with the classical example: joining two datasets $R = (a, v_0), (b, v_1), (b, v_2), \ldots, (b, v_n)$ and $S = (b, v'_0), (a, v'_1), (a, v'_2), \ldots, (a, v'_n)$ [61]. If a naive random sampling mechanism is applied to sample R and S before join, the result is likely to be an empty set. Thus, implementing *correct and precondition-free sampling* on joins is a challenging task.

6.2 Contribution

In this chapter, we present the design and implementation of ApproxJoin, an approximate distributed join system that improves the performance of join operations in big data analytics using approximate computing.

In this work, we tackle the challenges of distributed join operations (see § 6.1) by designing a novel join sampling algorithm that combines a *Bloom filter* sketch technique with *stratified sampling*. Bloom filter allows us to remove unnecessary data items which do not participate

in join operations. Therefore, these data items are not shuffled around as in native join operations, thus reducing communication overhead. Meanwhile, stratified sampling allows us to take a uniform sample over input datasets, such that operations after the join process only this resulting sample instead of the entire join output. Our proposed join mechanism can be used for both *two-way* joins and *multi-way joins*.[1]

We implemented our algorithm in a system called ApproxJoin based on Apache Spark [21, 229] and evaluate its effectiveness on micro-benchmarks, TPC-H queries, and a real-world use case. Our evaluation shows that ApproxJoin achieves a speedup of $(6-9\times)$ over Spark-based systems with the same sampling fraction. ApproxJoin leverages Bloom filtering to reduce the shuffled data volume during the join operation by $5-82\times$ compared to Spark-based systems. Without any sampling, our microbenchmark evaluation shows that ApproxJoin achieves a speedup of $2-10\times$ over the native Spark RDD join [229] and $1.06-3\times$ over a Spark repartition join. In addition, our evaluation with TPC-H benchmark shows that ApproxJoin is $(1.2-1.77\times)$ faster than the state-the-art related system — SnappyData [186].

To summarize, we make the following contributions.

- We propose a novel mechanism to perform sampling over joins that preserves the randomness (i.e., statistical quality) of sampling.
- We provide a confidence metric on the output accuracy using an error bound or confidence interval.
- Finally, we implement the proposed algorithms and mechanisms based on Apache Spark and extensively evaluate the system using a series of micro-benchmarks and real-world datasets.

6.3 Overview

In this section, we first describe how join operations are implemented in state-of-the-art big data systems. We then highlight the shortcomings of distributed join processing approaches. Afterwards, we present an overview of ApproxJoin that addresses these shortcomings.

6.3.1 Distributed Joins in Big Data Systems

Many big data systems, such as Apache Spark [21], Apache Flink [17] and Hadoop [19], partition the input datasets into smaller data chunks stored on different nodes of a deployment cluster. Although this partitioning helps with storing and processing large datasets in a fault-tolerant and parallel fashion, it also creates a challenge for join operations involving multiple datasets: the join operation requires the systems to move data items from one node to another, so that the cross-product operation can be performed on each node. To perform this data movement, the systems typically adopt two join strategies: (*i*) repartition join and (*ii*) broadcast join.

Repartition Join. This mechanism is the default join execution approach in big data systems [17, 21, 133]. The basic idea is to ensure that data items having the same join keys from all input datasets are collected at the same node in the cluster. To achieve this goal, the input

[1] A *two-way* join is the equi-join between two datasets ($R \bowtie S$), whereas a multi-way join is the composition of several two-way joins ($R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, where $R_i (i = 1, ..., n)$ is an input dataset).

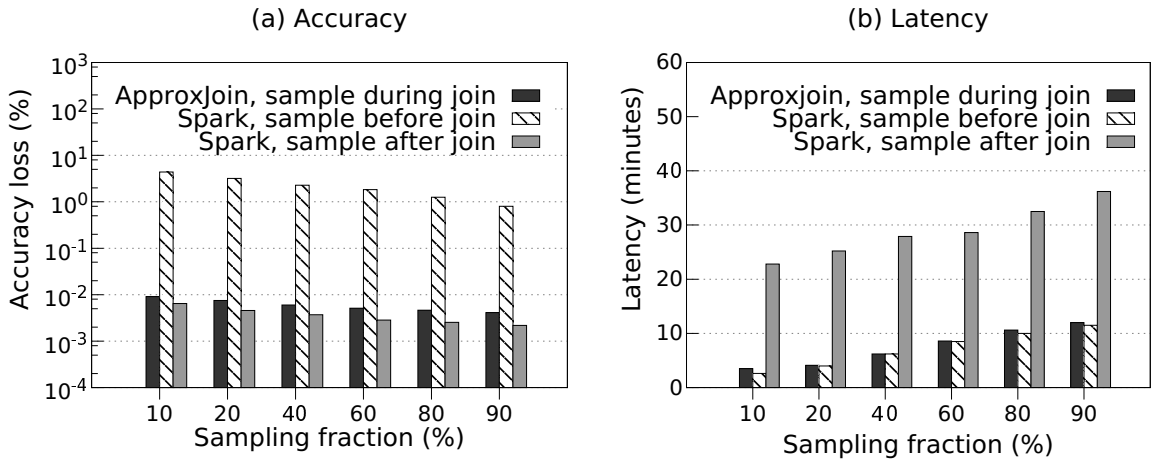

Figure 6.1 – The comparison between sampling over joins mechanisms with different sampling fractions. **(a)** Latency; **(b)** Accuracy loss.

datasets are repartitioned using the same partitioner (e.g., the same hash function), so that a given join key is collected by a single node. This node then performs the cross-product operation between the data items.

The advantage of this mechanism is that it does not require sophisticated steps to perform the join operation: The repartitioning operation can be modeled as a simple MapReduce job, whose shuffle phase naturally transfers the data items to the desired nodes. The disadvantage, however, is that this transfer can involve a significant amount of data and the transferred data items can include items that are not going to participate in the join. When the overlap fraction of the data items in different datasets is low, this approach leads to significant overhead in data transfer, which can also affect latency (see Section 6.6). In this chapter, the *overlap fraction* is defined as the total number of data items participating in the join operation divided by the total number of data items of all inputs.

Broadcast Join. When one input dataset is much smaller than the others, the systems can broadcast the smaller dataset to all nodes storing the other datasets. Afterwards, the join operation is performed at the map phase of the processing [40]. Broadcast join is very efficient for joins between a single, large dataset and one or more small datasets, because it avoids moving the data items of the large dataset through the network. However, this situation may not always be possible in real world traces (see §6.7).

6.3.2 Problem Statement

Join operations are often considered critical in big data systems. Unfortunately, they are expensive operations even with the state-of-the-art big data systems, such as Apache Spark and Apache Flink. This high cost is not only related to the computation but also to the communication required to perform the join operation. In practice, these big data systems can take hours or even days to process a join operation between dataset that is gigabytes or terabytes in size.

A naïve solution to improve the performance of joins is to employ more computing resources,

which comes at a significant monetary cost. An alternative approach is to utilize approximate computing, where only a subset of input data is used to obtain results. Approximate computing trades off accuracy for low latency and efficient resource utilization, by providing users with approximate results with associated error bounds. In the case of joins of big datasets, this trade-off is based on the observation that the exact results of a distributed join operation may not be necessary: the computation of exact results incurs additional delay for user response, but does not provide any additional value.

One widely used technique for approximate computing is sampling. Sampling over joins, however, is not a straightforward task and becomes challenging, especially in a distributed environment. One approach – *pre-join sampling* – performs sampling over the input datasets before the join and uses the samples to perform the join operation. Although this approach is simple and quite fast, it produces poor quality output, as explained in § 6.1 (see Figure 6.1 (a)). Another approach – *post-join sampling* – preserves the statistical properties of the join output by performing sampling after the join operation is finished. Unfortunately, this sampling requires the entire join operation to complete, and thus, introduces significantly high latency (see Figure 6.1 (b)).

A recent join approach, Wander Join [146], aims to model the join relationship among the data items as a graph, where a vertex represents a data item and an edge is drawn between two vertices if they can join with each other. Wander Join then performs *random walks*, so that each path would correspond to a sampled join output. However, Wander Join is not designed for distributed settings.

We make the following observations: Distributed analytics systems deal with a huge amount of data. The latency and accuracy of join operations in these systems become sensitive to the amount of data transferred during the join; therefore, it is important to ensure that only necessary data items are transferred over the network. Furthermore, the size of the input datasets also means that the cross product phase in the join operation can have a detrimental effect on the latency for obtaining the join results.

The case for approximation in distributed joins. Filtering data items not participating in the join can reduce the amount of data transfer significantly; thus, reducing the latency of the join operation. However, this reduction depends on the number of such data items: if the 'overlapping fraction' (i.e., data items that are going to participate in the join) is large, these items will still have to be transferred over the network. More importantly, the bottleneck of the join operation will shift to the cross product of these items (see §6.6).

6.3.3 ApproxJoin Overview

ApproxJoin is designed to mitigate the overhead of join operations in big data analytics systems. Figure 6.2 shows the high-level architecture of ApproxJoin. The input of ApproxJoin consists of several datasets to be joined. We facilitate joins on the input datasets by providing a simple user interface. The user submits the join query and its corresponding query execution budget. The query budget can be in the form of expected latency guarantees, or the desired accuracy level. Our system ensures that the input data is processed within the specified query budget. To achieve this, ApproxJoin applies the approximate computing paradigm by processing only a partial input data items from the datasets to produce an approximate output with error bounds. At a high level, ApproxJoin makes use of a combination of sketching and sampling to

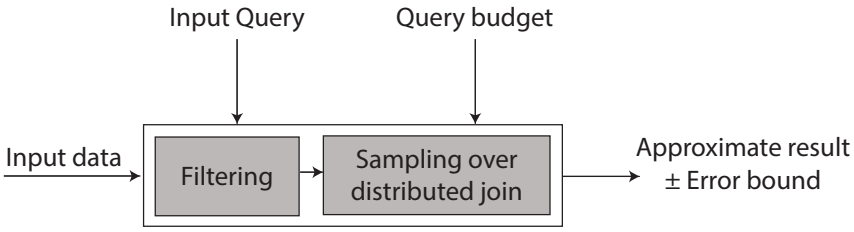

Figure 6.2 – ApproxJoin overview.

select a subset of input datasets based on the user specified query budget.

Design goals. We had the following goals when we designed and implemented ApproxJoin:

- *Transparency*: Provide a simple programming interface to users that is similar to the join operation of state-of-the-art systems. This goal implies that there will be negligible (or no) modifications to existing programs.
- *Query budget guarantees*: Ensure that the join operation is performed within the query budget supplied by the user in the form of desired latency or desired error bound. This goal implies that the system should accurately estimate the latency and error bounds of the approximation in the join operation.
- *Efficiency*: Handle very large input datasets in an efficient and cost-effective manner. This goal implies that the system reduces the usage of resources (e.g., network, CPU) as much as possible.

Query interface. ApproxJoin supports joins with algebraic aggregation functions, such as SUM, AVG, COUNT, and STDEV. In addition, a query execution budget is provided to specify either the latency requirement or desired error bound. More specifically, consider the case where a user wants to perform an aggregation query after an equal-join on attribute A for n input datasets $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, where $R_i (i = 1, ..., n)$ represents an input dataset. The user sends the query q and supplies a query budget q_{budget} to ApproxJoin. The query budget can be in the form of desired latency $d_{desired}$ or desired error bound $err_{desired}$. For instance, if the user wants to achieve a desired latency (e.g., $d_{desired} = 120$ seconds), or a desired error bound (e.g., $err_{desired} = 0.01$ with confidence level of 95%), he/she defines the query as follows:

SELECT SUM($R_1.V$ **+** $R_2.V$ **+ ... +** $R_n.V$**)**
FROM R_1**,** R_2**, ...,** R_n
WHERE $R_1.A$ **=** $R_2.A$ **= ... =** $R_n.A$
WITHIN 120 **SECONDS**
OR
ERROR 0.01 **CONFIDENCE** 95%

ApproxJoin executes the query and returns the most accurate query result within the desired latency which is 120 seconds, or returns the query result within the desired error bound ± 0.01 at a 95% confidence level.

Design overview. The basic idea of ApproxJoin is to address the shortcomings of the existing join operations in big data systems by reducing the number of data items that need to

be processed. Our first intuition is that we can reduce the latency and computation of a distributed join by removing redundant transfer of data items that are not going to participate in the join. Our second intuition is that the exact results of the join operation may be desired, but not necessarily critical, so that an approximate result with well-defined error bounds can also suffice for the user.

Figure 6.2 shows an overview of our approach. There are two stages in ApproxJoin for the execution of the user's query:

Stage #1: Filtering redundant items. In the first stage, ApproxJoin determines the data items that are going to participate in the join operation and filters the non-participating items. This filtering reduces the data transfer that needs to be performed over the network for the join operation. It also ensures that the join operation will not include 'null' results in the output that will require special handling, as in WanderJoin [146]. ApproxJoin employs a well-known data structure, *Bloom filter* [41]. Our filtering algorithm executes in parallel at each node that stores partitions of the input and handles multiple input tables at the same time.

Stage #2: Approximation in distributed joins. In the second stage, ApproxJoin uses a sampling mechanism that is executed *during* the join process: we sample the input datasets while the cross product is being computed. This mechanism overcomes the limitations of the previous approaches and enables us to achieve low latency as well as preserve the quality of the output as highlighted in Figure 6.1. Our sampling mechanism is executed during the join operation and preserves the statistical properties of the output.

In addition, we combine our mechanism with *stratified sampling* [11], where tuples with distinct join keys are sampled independently with simple random sampling. As a result, data items with different join keys are fairly selected to represent the sample, and no join key will be overlooked. The final sample will contain all join keys—even the ones with few data items—so that the statistical properties of the sample are preserved.

More specifically, ApproxJoin executes the following steps for approximation in distributed joins:

Step #2.1: Determine sampling parameters. ApproxJoin employs a *cost function* to compute an optimal sample size according to the corresponding budget of the query. This computation ensures that the query is executed within the desired latency and error bound parameters of the user.

Step #2.2: Sample and execute query. Using this sampling rate parameter, ApproxJoin samples during the join and then executes the aggregation query q using the obtained sample.

Step #2.3: Estimate error. After executing the query, ApproxJoin provides an approximate result together with a corresponding error bound in the form of $result \pm error\_bound$ to the user.

Note that our sampling parameter estimation provides an adaptive interface for selecting the sampling rate based on the user-defined accuracy and latency requirements. ApproxJoin adapts by activating a feedback mechanism to refine the sampling rate after learning the data distribution of the input datasets (see §6.5).

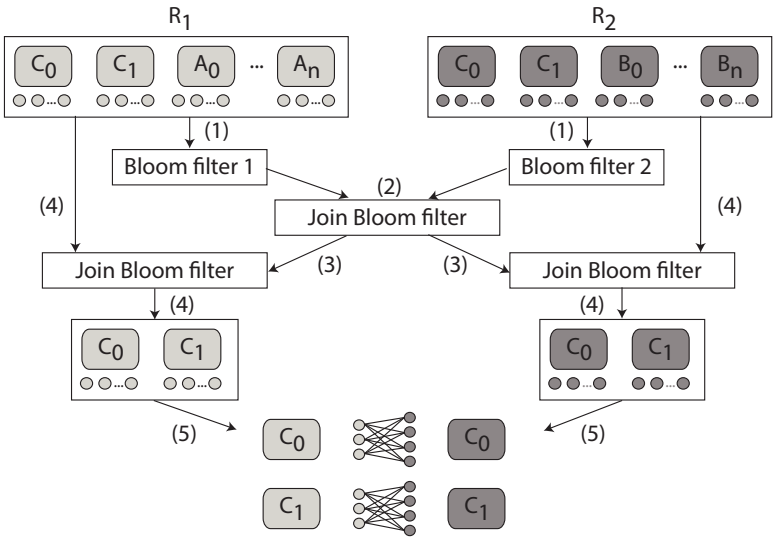

Figure 6.3 – Bloom filter building.

6.4 Design

In this section, we explain the details of ApproxJoin's operation. We first describe how we filter redundant data items for multiple datasets to support multiway joins (§6.4.1). Then, we describe how we perform approximation in distributed joins using three main steps: (1) how we determine the sampling parameter to satisfy the user-specified query budget (§6.4.2), (2) how our novel sampling mechanism executes during the join operation (§6.4.3), and finally (3) how we estimate the error for the approximation (§6.4.4).

6.4.1 Filtering Redundant Items

In a distributed setting, join operations can be expensive due to the communication cost of the data items. This cost can be especially high in multi-way joins, where several datasets are involved in the join operation. One reason of this high cost is that data items not participating in the join are shuffled through the network during the join operation.

To reduce this communication cost, we need to distinguish such redundant items and avoid transferring them over the network. In ApproxJoin, we use Bloom filters for this purpose. The basic idea is to utilize Bloom filters as a compressed set of all items present at each node and combine them to find the intersection of the datasets used in the join. This intersection will represent the set of data items that are going to participate in the join.

A Bloom filter is a data structure designed to query the presence of an element in a dataset in a rapid and memory-efficient way [41]. There are three advantages why we choose Bloom filters for our purpose. First, querying the membership of an element is efficient: it has $O(h)$ complexity, where h denotes a constant number of hash functions. Second, the size of the filter is linearly correlated with the size of the input, but it is significantly smaller compared to the original input size. Finally, constructing a Bloom filter is fast and requires a single pass over the input.

Bloom filters have been exploited to improve distributed joins in the past [31, 142, 206, 207].

Algorithm 10: Filtering using multi-way Bloom filter

Input: n: number of input datasets
$|BF|$: size of the Bloom filter
fp: false positive rate of the Bloom filter
R: input datasets

```
1  // Build a Bloom filter for the join input datasets R
2  buildJoinFilter(R, |BF|, fp)
3  begin
4      // Build a Bloom filter for each input R_i
5      // Executed in parallel at worker nodes
6      ∀i ∈ {1...n}: BF_i ← buildInputFilter(R_i, |BF|, fp);
7      // Merge input filters BF_i for the overlap between inputs
8      // Executed sequentially at master node
9      BF ← ∩_{i=1}^{n} BF_i;
10     return BF
11 // Build a Bloom filter for input R_i
12 // Executed in parallel at worker nodes
13 buildInputFilter(R_i, |BF|, fp)
14 begin
15     |p_i| := number of partitions of input dataset R_i
16     p_i := {p_{i,j}}, where j = 1, ..., |p_i|
17     //MAP PHASE
18     //Initialize a filter for each partition
19     forall j in {1...|p_i|} do
20         p-BF_{i,j} ← BloomFilter(|BF|, fp);
21         ∀r_j ∈ p_{i,j}: p-BF_{i,j}.add(r_j.key);
22     //REDUCE PHASE
23     // Merge partition filters to the dataset filter
24     BF_i ← ∪_{j=1}^{|p_i|} p-BF_{i,j};
25     return BF_i
```

However, these proposals support only two-way joins. Although one can cover joins with multiple input datasets by chaining two-way joins, this approach would add to the latency of the join results. ApproxJoin handles multiple datasets at the same time and supports multi-way joins without introducing additional latency.

For simplicity, we first explain how our algorithm uses a Bloom filter to find the intersection of two input datasets. Afterwards, we explain how our algorithm finds the intersection of multiple datasets at the same time.

I: Two-way Bloom filter. For the two-way filtering, consider the join operation of two datasets $R_1 \bowtie R_2$ (see Figure 6.3). First, we construct a Bloom filter for each input (step 1 in Figure 6.3), which we refer to as *dataset filter*. We perform AND among the dataset filters (step 2). The resulting Bloom filter represents the intersection of both datasets and is referred to as *join filter*.

Afterwards, we broadcast the join filter to all nodes (step 3). Each node checks the membership of the data items in its respective input dataset in the join filter. If a data item is not present, it is discarded. In Figure 6.3, all data items with keys $C0$ and $C1$ are preserved.

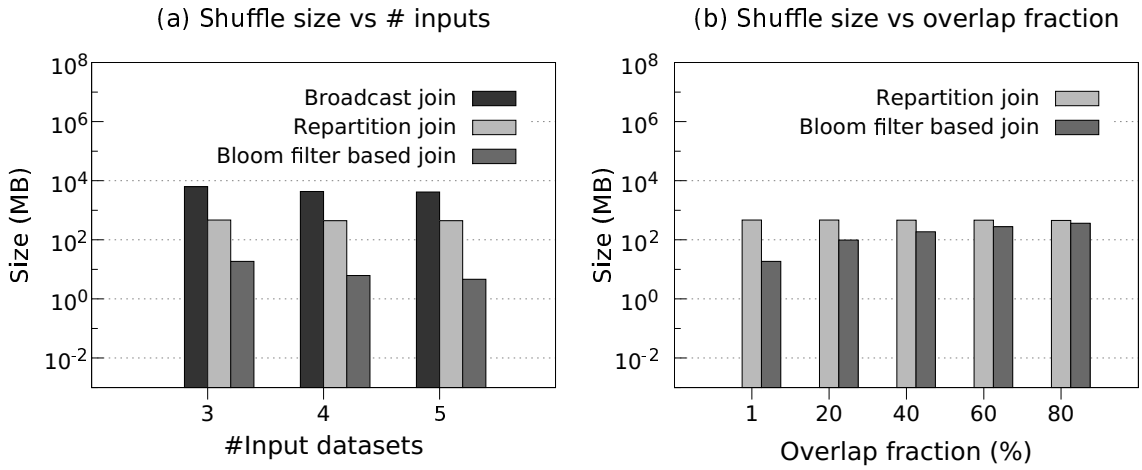

Figure 6.4 – **(a)** The shuffled data size comparison between join mechanisms. The overlap fraction between input datasets is set to 1%; **(b)** The shuffled data size of repartition join and Bloom filter based join with different overlap fractions. The number of input datasets is 3 (Simulation with 100 nodes cluster).

II: Multi-way Bloom filter. We generalize the two-way Bloom filter, so that it applies to n input datasets. Consider the case where we want to perform a join operation between multiple input datasets R_i, where $i = 1, ..., n$: $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$.

Algorithm 10 presents the two main steps to construct the multi-way join filter. In the first step, we create a Bloom filter BF_i for each input R_i, where $i = 1, ..., n$ (lines 4-6), which is executed in parallel at all worker nodes that have the input datasets. In the second step, we combine the n dataset filters into the join filter by simply applying the logical AND operation between the dataset filters (lines 7-9). This operation adds virtually no additional overhead to build the join filter, because the logical AND operation with Bloom filters is fast, even though the number of dataset filters being combined is n instead of two.

Note that an input dataset may consist of several partitions hosted on different nodes. To build the dataset filter for these partitioned inputs, we perform a simple MapReduce job that can be executed in distributed fashion: We first build the *partition filters* $\text{p-BF}_{i,j}$, where $j = 1, ..., |p_i|$ and $|p_i|$ is the number of partitions for input dataset R_i during the Map phase, which is executed at the nodes that are hosting the partitions of each input (lines 15-21). Then, we combine the partition filters to obtain the dataset filter BF_i in the Reduce phase by merging the partition filters via the logical OR operation into the corresponding dataset filter BF_i (lines 22-24). This process is executed for each input dataset and in parallel (see buildInputFilter()).

Is Filtering Sufficient?

After constructing the join filter and broadcasting it to the nodes, one straightforward approach would be to complete the join operation by performing the cross product with the data items present in the intersection. Figure 6.4 (a) shows the advantage of performing such a join operation with multiple input datasets based on a simulation (see §6.4.5). With the broadcast

join and repartition join mechanisms, the transferred data size gradually increases with the increasing number of input datasets. However, with the Bloom filter based join approach, the transferred data size significantly reduces even when the number of datasets in the join operation increases.

Although this filtering seems to significantly reduced transferred data among nodes, this reduction may not always be possible. Figure 6.4 (b) shows that even with a modest overlap fraction between three input datasets (i.e., 40%), the amount of transferred data becomes comparable with the repartition join mechanism. Furthermore, the cross product operation will involve a significant amount of data items, potentially becoming the bottleneck.

In ApproxJoin, we first filter redundant data items as described in this section. Afterwards, we check whether the overlap fraction between the datasets is small enough, such that we can meet the latency requirements of the user. If so, we perform the cross product of the data items participating in the join. In other words, we do not need approximation in this case (i.e., we compute the exact join result). If the overlap fraction is large, we continue with our approximation technique, which we describe next.

6.4.2 Approximation: Cost Function

ApproxJoin supports the query budget interface for users to define a desired latency ($d_{desired}$) or a desired error-bound ($err_{desired}$) as described in §6.3. ApproxJoin ensures the join operation executed within the specified query budget by tuning the sampling parameter accordingly. In this section, we describe how ApproxJoin converts the join requirements given by a user (i.e., $d_{desired}, err_{desired}$) into an optimal sampling parameter. To meet the budget from the user, ApproxJoin makes use of two types of cost functions to determine the sample size: (*i*) latency cost function, (*ii*) error-bound cost function.

I: Latency cost function. In ApproxJoin, we consider the latency for the join operation being dominated by two factors: 1) the time to filter and transfer participating join data items, d_{dt}, and 2) the time to compute the cross product, d_{cp}. To execute the join operation within the delay requirements of the user, we have to estimate each contributing factor.

The latency for filtering and transferring the join data items, d_{dt}, is measured during the filtering stage (described in §6.4.1). We then compute the remaining allowed time to perform the join operation:

$$d_{rem} = d_{desired} - d_{dt} \tag{6.1}$$

To satisfy the latency requirements, the following must hold:

$$d_{cp} \leq d_{rem} \tag{6.2}$$

In order to estimate the latency of the cross product phase, we need to estimate how many cross products we have to perform. Imagine that the output of the filtering stage consists of data items with m distinct keys $C_1, C_2 \cdots, C_m$. To fairly select data items, we perform sampling for each join key independently (explained in §6.4.3). In other words, we will perform *stratified sampling*, such that each key C_i corresponds to a stratum and has B_i data items. Let b_i represent the sample size for C_i. The total number of cross products is given by:

$$CP_{total} = \sum_{1}^{m} b_i \tag{6.3}$$

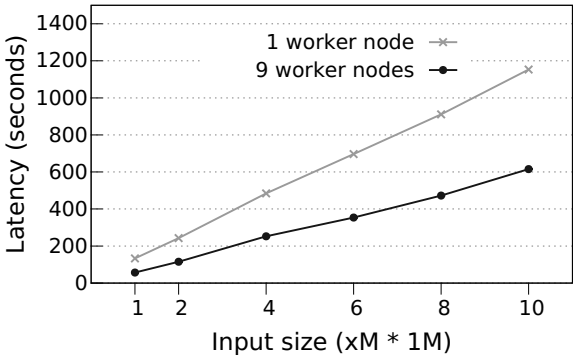

Figure 6.5 – The latency of cross-product operation with varying input sizes.

The latency for the cross product phase would be then:

$$d_{cp} = \beta_{compute} * CP_{total} \tag{6.4}$$

where $\beta_{compute}$ denotes the scale factor that depends on the computation capacity of the cluster (e.g., #cores, total memory).

We determine $\beta_{compute}$ empirically via a microbenchmark by profiling the compute cluster as an offline stage. In particular, we measure the latency to perform cross products with varying input sizes. Figure 6.5 shows that the latency is linearly correlated with the input size, which is consistent with plenty of I/O bound queries in parallel distributed settings [7, 15, 228]. Based on this observation, we estimate the latency of the cross product phase as follows:

$$d_{cp} = \beta_{compute} * CP_{total} + \varepsilon \tag{6.5}$$

where ε is a noise parameter.

Given a desired latency $d_{desired}$, the sampling fraction $s = \frac{CP_{total}}{\sum_{1}^{m} B_i}$ can be computed as:

$$s = (\frac{d_{rem} - \varepsilon}{\beta_{compute}}) * \frac{1}{\sum_{1}^{m} B_i} = (\frac{d_{desired} - d_{dt} - \varepsilon}{\beta_{compute}}) * \frac{1}{\sum_{1}^{m} B_i} \tag{6.6}$$

Then, the sample size b_i of stratum C_i can be then selected as follows:

$$b_i \leq s * B_i \tag{6.7}$$

According to this estimation, ApproxJoin checks whether the query can be executed within the latency requirement of the user. If not, the user is informed accordingly.

II: Error bound cost function. If the user specified a requirement for the error bound, we have to execute our sampling mechanism, such that we satisfy this requirement. Our sampling mechanism utilizes simple random sampling for each stratum (see §6.4.3). As a result, the error err_i can be computed as follows [152]:

$$err_i = z_{\frac{\alpha}{2}} * \frac{\sigma_i}{\sqrt{b_i}} \tag{6.8}$$

where b_i represents the sample size of C_i and σ_i represents the standard deviation.

Unfortunately, the standard deviation σ_i of stratum C_i cannot be determined without knowing the data distribution. To overcome this problem, we design a feedback mechanism to refine the sample size (the implementation details are in §6.5): For the first execution of a query, the standard deviation of σ_i of stratum C_i is computed and stored. For all subsequent executions of the query, we utilize these stored values to calculate the optimal sample size using Equation 6.10. Alternatively, one can estimate the standard deviation using a bootstrap method [7, 152]. Using this method, however, would require performing offline profiling of the data.

With the knowledge of σ_i and solving for b_i gives:

$$b_i = (z_{\frac{\alpha}{2}} * \frac{\sigma_i}{err_i})^2 \tag{6.9}$$

With 95% confidence level, we have $z_{\frac{\alpha}{2}} = 1.96$; thus, $b_i = 3.84 * (\frac{\sigma_i}{err_i})^2$. err_i should be less or equal to $err_{desired}$, so we have:

$$b_i \geq 3.84 * (\frac{\sigma_i}{err_{desired}})^2 \tag{6.10}$$

Equation 6.10 allows us to calculate the optimal sample size given a desired error bound $err_{desired}$.

III: Combining latency and error bound. From Equations 6.7 and 6.10, we have a trade-off function between the latency and the error bound with confidence level of 95%:

$$d_{desired} \approx 3.84 * (\frac{\sigma_i}{err_{desired}})^2 * \frac{\beta_{compute}}{B_i} * (\sum_{1}^{m} B_i) + d_{dt} + \varepsilon \tag{6.11}$$

6.4.3 Approximation: Sampling and Execution

In this section, we describe our sampling mechanism that executes during the cross product phase of the join operation. Executing approximation during the cross product enables ApproxJoin to have highly accurate results compared to pre-join sampling. To preserve the statistical properties of the exact join output, we combine our technique with *stratified sampling*. Stratified sampling ensures that no join key is overlooked: for each join key, we perform simple random sampling over data items independently. This method fairly selects data items from different join keys. The filtering stage (§6.4.1) guarantees that this selection is executed only from the data items participating in the join.

For simplicity, we first describe how we perform stratified sampling during the cross product on a single node. We then describe how the sampling can be performed on multiple nodes in parallel.

I: Single node stratified sampling. Consider an inner join example of $J = R_1 \bowtie R_2$ with a pair keys and values, $((k_1, v_1), (k_2, v_2))$, where $(k_1, v_1) \in R_1$ and $(k_2, v_2) \in R_2$. This join operation produces an item $(k_1, (v_1, v_2) \in J$ if and only if $(k_1 = k_2)$.

Consider that R_1 contains (C_0, v_1), (C_0, v_2), and (C_0, v_3), and that R_2 contains (C_0, v_4), (C_0, v_5), (C_0, v_6), and (C_0, v_7). The join operation based on key C_0 can be modeled as a complete bipartite graph (shown in Figure 6.6). To execute stratified sampling over the join, we

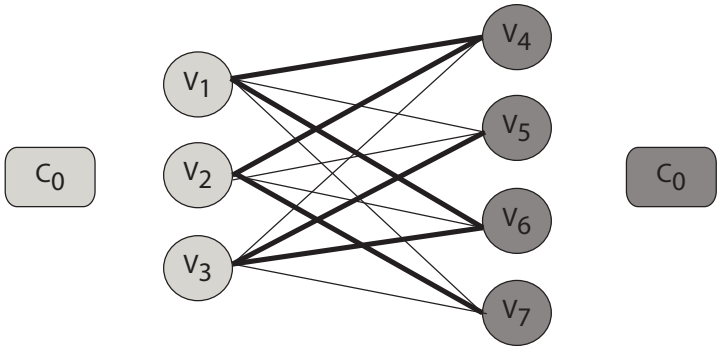

Figure 6.6 – The cross-product operation graph of join data items having overlap key C_0.

perform random sampling on data items having the same join key (i.e., key C_0). As a result, this process is equal to performing *edge sampling* on the complete bipartite graph.

Sampling edges from the complete bipartite graph would require building the graph, which would correspond to computing the full cross product. To avoid this cost, we propose a mechanism to randomly select edges from the graph without building the complete graph. The function *sampleAndExecute*() in Algorithm 11 describes the algorithm to sample edges from the complete bipartite graph. To include an edge in the sample, we randomly select one endpoint vertex from each side and then yield the edge connecting these vertices (lines 19-23). To obtain a sample of size b_i, we repeat this selection b_i times (lines 17-18 and 24). This process is repeated for each join key C_i (lines 15-24).

II: Distributed stratified sampling. The sampling mechanism can naturally be adapted to execute in a distributed setting. Algorithm 11 describes how this adaptation can be achieved. In the distributed setting, the data items participating in the join are distributed to worker nodes based on the join keys using a partitioner (e.g., hash-based partitioner). A master node facilitates this distribution and directs each worker to start sampling (lines 4-5). Each worker then performs the function *sampleAndExecute*() in parallel to sample the join output and execute the query (lines 12-26).

III: Query execution. After the sampling, each node executes the input query on the sample to produce a partial query result, $result_i$, and return it to the master node (lines 25-26). The master node collects these partial results and merges them to produce a query result (lines 6-8). The master node also performs the error bound estimation (lines 9-10), which we describe in the following subsection (§6.4.4) . Afterwards, the approximate query result and its error bounds are returned to the user (line 11).

6.4.4 Approximation: Error Estimation

As the final step, ApproxJoin computes an error-bound for the approximate result. The approximate result is then provided to the user as $approxresult \pm error\_bound$.

Our sampling algorithm (i.e., the *sampleAndExecute*() function in Algorithm 11) described in the previous section can produce an output with duplicate edges. For such cases, we use the Central Limit Theorem to estimate the error bounds for the output. This error estimation is possible because the sampling mechanism works as a random sampling with replacement.

Algorithm 11: : Stratified sampling over join

Input:
b_i: sample size of join key C_i
N_{1i} & N_{2i}: set of vertices (items) in two sides of complete bipartite graph of join key C_i
m: number of join keys
C: set of all join keys (i.e., $\{\forall i \in \{1, ..., m\} : C_i\}$)

```
1  // Executed sequentially at master node
2  sampleDuringJoin()
3  begin
4      foreach worker_i in workerList do
5          result_i ← worker_i.sampleAndExecute();// Direct workers to sample and execute the query
6      result ← ∅; // Initialize empty query result
7      foreach C_i in C do
8          result ← merge(result_i);// Merge query results from workers
9      // Estimate error for the result
10     result ± error_bound ← errorEstimation(result);
11     return result ± error_bound;
12 // Executed in parallel at worker nodes
13 sampleAndExecute()
14 begin
15     foreach C_i in C do
16         sample_i ← ∅; // Sample of join key C_i
17         count_i ← 0;// Initialize a count to keep track # selected items
18         while count_i < b_i do
19             // Select two random vertices
20             v ← random(N_1i);
21             v' ← random(N_2i);
22             // Add an edge between the selected vertices and update the sample
23             sample_i.add(< v, v' >);
24             count_i ← count_i + 1; // Update counting
25         result_i ← query(sample_i); // Execute query over sample
26         return result_i;
```

We can also remove the duplicate edges during the sampling process by using a hash table, and repeat the algorithm steps until we reach the desired number of data items in the sample. This approach might worsen the randomness of the sampling mechanism and could introduce bias into the sample data. In this case, we use the Horvitz-Thompson [123] estimator to remove this bias. We next explain the details of these two error estimation mechanisms.

I: Error estimation using the Central Limit Theorem. Suppose we want to compute the approximate sum of data items after the join operation. The output of the join operation contains data items with m different keys $C_1, C_2 \cdots, C_m$, each key (stratum) C_i has B_i data items and each such data item j has an associated value $v_{i,j}$. To compute the approximate sum of the join output, we sample b_i items from each join key C_i according to the parameter we computed (described in §6.4.2). Afterwards, we estimate the sum from this sample as $\hat{\tau} = \sum_{i=1}^{m}(\frac{B_i}{b_i}\sum_{j=1}^{b_i} v_{ij}) \pm \epsilon$, where the error bound ϵ is defined as:

$$\epsilon = t_{f,1-\frac{\alpha}{2}}\sqrt{\widehat{Var}(\hat{\tau})} \tag{6.12}$$

Here, $t_{f,1-\frac{\alpha}{2}}$ is the value of the t-distribution (i.e., *t-score*) with f degrees of freedom and $\alpha = 1 - confidencelevel$. The degree of freedom f is calculated as:

$$f = \sum_{i=1}^{m} b_i - m \tag{6.13}$$

The estimated variance for the sum, $\widehat{Var}(\hat{\tau})$, can be expressed as:

$$\widehat{Var}(\hat{\tau}) = \sum_{i=1}^{m} B_i * (B_i - b_i) \frac{r_i^2}{b_i} \tag{6.14}$$

Here, r_i^2 is the population variance in the i^{th} stratum. We use the statistical theories for stratified sampling [205] to compute the error bound.

II: Error estimation using the Horvitz-Thompson estimator. Consider the second case, where we remove the duplicate edges and resample the endpoint nodes until another edge is yielded. The bias introduced by this process can be estimated using the Horvitz-Thomson estimator. Horvitz-Thompson is an unbiased estimator for the population sum and mean, regardless whether sampling is with or without replacement.

Let π_i is a positive number representing the probability that data item having key C_i is included in the sample under a given sampling scheme. Let y_i is the sample sum of items having key C_i: $y_i = \sum_{j=1}^{b_i} v_{ij}$. The Horvitz-Thompson estimation of the total is computed as [205]:

$$\hat{\tau_\pi} = \sum_{i=1}^{m} (\frac{y_i}{\pi_i}) \pm \epsilon_{ht} \tag{6.15}$$

where the error bound ϵ_{ht} is given by:

$$\epsilon_{ht} = t_{\frac{\alpha}{2}} \sqrt{\widehat{Var}(\hat{\tau_\pi})} \tag{6.16}$$

where t has $n-1$ degrees freedom. The estimated variance of the Horvitz-Thompson estimation is computed as:

$$\widehat{Var}(\hat{\tau_\pi}) = \sum_{i=1}^{m} (\frac{1-\pi_i}{\pi_i^2}) * y_i^2 + \sum_{i=1}^{m} \sum_{j \neq i} (\frac{\pi_{ij} - \pi_i \pi_j}{\pi_i \pi_j}) * \frac{y_i y_j}{\pi_{ij}} \tag{6.17}$$

where π_{ij} is the probability that both data items having key C_i and C_j are included.

Note that the Horvitz-Thompson estimator does not depend on how many times a data item may be selected: each distinct item of the sample is used only once [205].

6.4.5 Analysis of ApproxJoin

In this section, we first analyze the communication complexity of ApproxJoin. Thereafter, we provide computational complexity analysis for the proposed stratified sampling over joins in the sampling stage of ApproxJoin.

I. Communication Complexity

For the communication complexity, we analyze the performance gain of ApproxJoin in terms of shuffled data size during the filter stage with various setting of Bloom filters using a model-based analysis. We compare the gains of ApproxJoin with the broadcast and repartition join mechanisms. Based on our analysis, we also show how to select input parameters for Bloom filter to achieve an optimal trade-off between reducing the shuffled data volume and the desired false positive value in the Bloom filters.

Suppose we want to execute a multi-way join operation on attribute A for n input datasets $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, where $R_i (i = 1, ..., n)$ is an input dataset. For simplicity, we assume that $|R_1| < |R_2| < ... < |R_n|$. The number of nodes in our experimental cluster is k and $k > 1$.

Broadcast join. In broadcast join, we broadcast all smaller datasets to all nodes that contain the largest dataset. The total shuffled data volume is bounded by:

$$S_{bc} = \langle |R_1| + |R_2| + ... + |R_{n-1}| \rangle * (k - 1) \tag{6.18}$$

When we add one more node to the cluster, the relative increase in the shuffled data volume in broadcast join will be:

$$\frac{\theta_F}{\theta_k} = |R_1| + |R_2| + ... + |R_{n-1}| \tag{6.19}$$

When we add one more dataset R_{n+1} to the join operation, the relative increase in the shuffled data volume will be:

$$\frac{\theta_F}{\theta_n} = |R_n| * (k - 1) \tag{6.20}$$

Repartition join. In repartition join, we shuffle data items of datasets across the cluster to make sure that each node in the cluster will keep at least a chunk/partition of each dataset. Therefore, the shuffled data volume in repartition join is computed as follows:

$$S_{re} = \langle |R_1| + |R_2| + ... + |R_{n-1}| + |R_n| \rangle * \frac{k - 1}{k} \tag{6.21}$$

When we add one more node to the cluster, the relative increase in the shuffled data volume in repartition join will be:

$$\frac{\theta_F}{\theta_k} = \langle |R_1| + |R_2| + ... + |R_{n-1}| + |R_n| \rangle * \frac{1}{k * (k + 1)} \tag{6.22}$$

When we add one more dataset R_{n+1} to the join operation, the relative increase in the shuffled data volume will be:

$$\frac{\theta_F}{\theta_n} = |R_{n+1}| * \frac{k - 1}{k} \tag{6.23}$$

ApproxJoin. Algorithm 10 describes our proposed filtering using a Bloom filter for multi-way joins. The algorithm builds a Bloom filter BF_i for each dataset R_i using the function *buildInputFilter*. In the Map phase, the function builds Bloom filters for all partitions of the input dataset R_i. In the Reduce phase, all the partitioned Bloom filters are merged to build the Bloom filter BF_i for the input dataset R_i. Since we fix the size for all Bloom filters, the volume of the shuffled data for building Bloom filters for all inputs is computed as $|BF| * (k-1) * n$, where

$|BF|$ is the size of each Bloom filter. Thereafter, the Bloom filters of all inputs are combined to build a join Bloom filter with size $|BF|$ for all join input using the function *buildJoinFilter*.

Next, the algorithm broadcasts the join Bloom filter to all nodes to filter out all data items that do not participate in the join operation. The shuffled data size of the broadcast step is calculated as $|BF| * (k-1)$. The volume of shuffled data for the filtering step is computed as $\langle |r_1| + |r_2| + ... + |r_n| \rangle * \frac{k-1}{k}$; where $|r_i|$ is the size of data items participating in the join operation of input R_i .

In summary, the total volume of shuffled data in the proposed filtering mechanism is calculated as follows:

$$\begin{aligned} S_{bf} = \quad & |BF| * (k-1) * (n+1) + \\ & + \langle |r_1| + |r_2| + ... + |r_n| \rangle * \frac{k-1}{k} \end{aligned} \tag{6.24}$$

When we add one more node to the cluster, the relative increase in the shuffled data volume in ApproxJoin will be:

$$\begin{aligned} \frac{\theta_F}{\theta_k} = \quad & |BF| * (n+1) + \\ & + \langle |r_1| + |r_2| + ... + |r_{n-1}| + |r_n| \rangle * \frac{1}{k * (k+1)} \end{aligned} \tag{6.25}$$

When we add one more dataset R_{n+1} to the join operation, the relative increase of the shuffled data volume is computed as:

$$\frac{\theta_F}{\theta_n} = |BF| * (k-1) + |r_{n+1}| * \frac{k-1}{k} \tag{6.26}$$

Note that for Bloom filters, false positives are possible, but false negatives are not. There is a trade-off between the size of a bit vector $|BF|$ and the probability of a false positive. A larger $|BF|$ has fewer false positives but consumes more memory, whereas a smaller $|BF|$ requires less memory at the risk of more false positives. The false positive rate can be computed as [41]: $p \approx (1 - e^{\frac{N*h}{|BF|}})^h$; where h is the number of hash functions and N is the number of data items inserted to the Bloom filter. For a given $|BF|$ and N, the value of h that minimizes the false positive probability [41, 172] is $h = \frac{|BF|}{N} * \ln 2$. Therefore, we have: $p \approx (1 - e^{\ln 2})^{\frac{|BF|}{N} * \ln 2}$ which can be simplified to: $\ln p = -\frac{|BF|}{N} * (\ln 2)^2$. Thus $|BF|$ can be computed as follows:

$$|BF| = -\frac{N * \ln p}{(\ln 2)^2} \tag{6.27}$$

In our design, we select $N = |R_n|$; where $|R_n|$ is the size of the largest input dataset.

We use a simulation-driven approach, based on the aforementioned model, to analyze the trade-off between reducing the shuffled data volume and the desired false positive value in the Bloom filters. We conduct an experiment by using the simulation. We create three input datasets R_1, R_2, and R_3; where $|R_1| = 10000$, $|R_2| = 1000000$, $|R_3| = 10000000$. We set the overlap fraction to 1%; and the number of keys in R_1, R_2, and R_3 to 1000, 100000, and 1000000, respectively. The value of each data item in the datasets follows Poisson distribution with lambda parameter 10. Finally, we set the number of nodes in the cluster $k = 100$. We run the

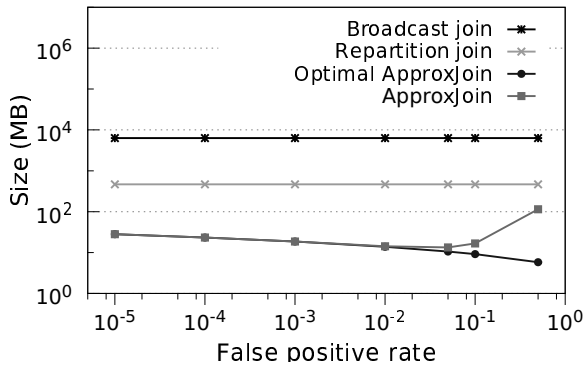

Figure 6.7 – The data shuffle size comparison between broadcast join, repartition join, optimal ApproxJoin, and ApproxJoin with different desired false positive values. Optimal ApproxJoin is the case that there is no false positive happen during the join operation in ApproxJoin.

simulation with the input parameters and analyze the shuffled data volume with different false positive values. Figure 6.7 shows the shuffled data volume of broadcast join, repartition join, optimal ApproxJoin, and ApproxJoin. Optimal ApproxJoin is the case when there are no false positives during the join operation of ApproxJoin. When the false positive rate is set to less than or equal to 0.01, ApproxJoin reaches the optimal case.

This simulation allows us to quickly set the desired false positive parameter for ApproxJoin with varying input datasets.

II. Computational Complexity

Since ApproxJoin significantly reduces the communication overhead for distributed join operations, it becomes important to ensure that the bottleneck is not shifted to another part of the system, potentially hindering improved performance. Thus, another important aspect of the performance analysis is the computational complexity, which theoretically represents the amount of time required to execute the proposed algorithm. Here, we provide the computational complexity analysis of our sampling mechanism (§6.4.3) in comparison with the broadcast and repartition join mechanisms.

Consider that we want to perform a join operation for n inputs $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, where $R_i (i = 1, ..., n)$ is the input dataset. The inputs contain m join keys $C_j (j = 1, ..., m)$. Let $|r_{ij}|$ be the number of data items participating in the join operation from input R_i with join key C_j.

In repartition join or broadcast join, we need to perform the full cross product operation over these data items. As a result, the computational complexity for each join key C_j is $O(\prod_{i=1}^{n} |r_{ij}|)$.

On the other hand, in ApproxJoin, we perform sampling over the cross product operation. As a result, for each join key C_j, the sampling mechanism performs b_j random selections on each side of the bipartite graph (§6.4.3), where b_j represents the sample size of join key C_j. b_j is computed as $s * \prod_{i=1}^{n} |r_{ij}|$, where s is the sampling fraction. Thus, the computational

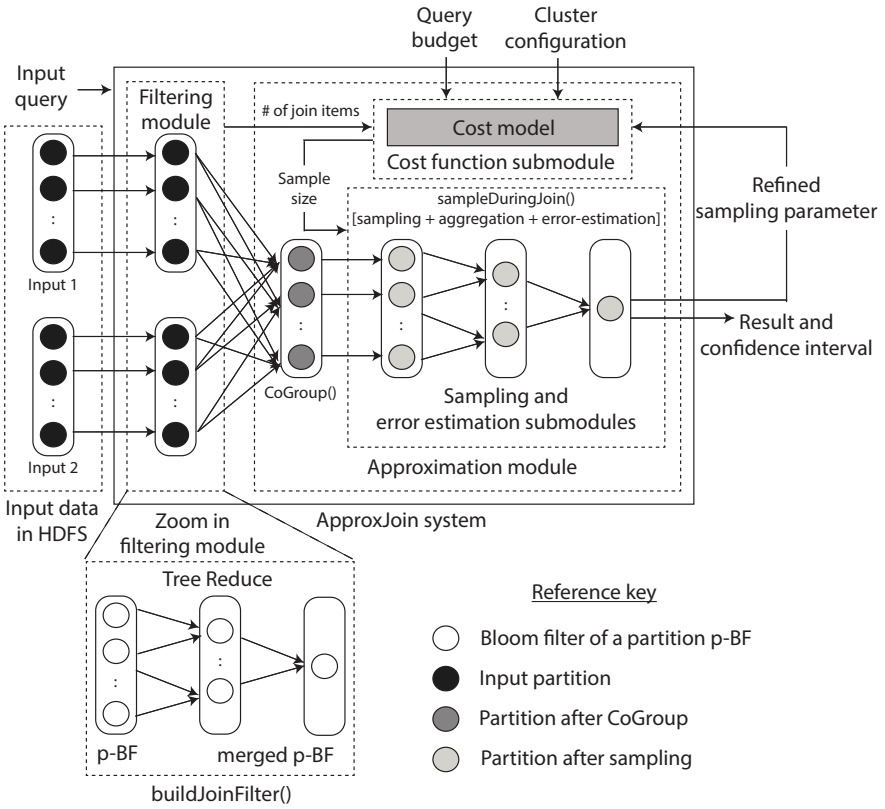

Figure 6.8 – System implementation: the figure shows distributed dataflow graph execution (on y-axis) for different stages (on x-axis) in ApproxJoin.

complexity of the proposed sampling mechanism is O(b_i). Rewriting b_j as $s * \prod_{i=1}^{n} |r_{ij}|$, the computational complexity for each join key C_j becomes O($s * \prod_{i=1}^{n} |r_{ij}|$)).

6.5 Implementation

In this section, we describe the implementation details of ApproxJoin. At the high level, ApproxJoin composed of two main modules: (*i*) filtering and (*ii*) approximation. The filtering module constructs the join filter to determine the data items participating in the join. These data items are fed to the approximation module to perform the join query within the query budget specified by the user.

We implemented our design by modifying Apache Spark [21]. Spark uses Resilient Distributed Datasets (RDDs) [229] for scalable and fault-tolerant distributed data-parallel computing. An RDD is an immutable collection of objects distributed across a set of machines. To support existing programs, we provide a simple programming interface that is also based on the RDDs. In other words, all operations in ApproxJoin, including filtering and approximation, are transparent to the user. To this end, we have implemented a PairRDD for *approxjoin*() function to perform the join query within the query budget over inputs in the form of RDDs. Figure 6.8 shows in detail the directed acyclic graph (DAG) execution of ApproxJoin.

I: Filtering module. The join Bloom filter module implements the filtering stage described in §6.4.1 to eliminate the non-participating data items. A straightforward way to implement *buildJoinFilter()* in Algorithm 10 is to build Bloom filters for all partitions (p-BFs) of each input and merge them in the driver of Spark in the Reduce phase. However, in this approach, the driver quickly becomes a bottleneck when there are multiple data partitions located on many workers in the cluster. To solve this problem, we leverage the *treeReduce* scheme [133]. In this model, we combine the Bloom filters in a hierarchical fashion, where the reducers are arranged in a tree with the root performing the final merge (Figure 6.8). If the number of workers increases (i.e., ApproxJoin deployed in a bigger cluster), more layers are added to the tree to ensure that the load on the driver remains unchanged. After building the join filter, ApproxJoin broadcasts it to determine participating join items in all inputs and feed them to the approximation module.

The approximation module consists of three submodules including the cost function, sampling and error estimation. The cost function submodule implements the mechanism in §6.4.2 to determine the sampling parameter according to the requirements in the query budget. The sampling submodule performs the proposed sampling mechanism (described in §6.4.3) and executes the join query over the filtered data items with the sampling parameter. The error estimation submodule computes the error-bound (i.e., confidence interval) for the query result from the sampling module (described in §6.4.4). This error estimation submodule also performs fine-tuning of the sample size used by the sampling submodule to meet the accuracy requirement in subsequent runs.

II: Approximation: Cost function submodule. The cost function submodule converts the query budget requirements provided by the user into the sampling parameter used in the sampling submodule. We implemented a simple cost function by building a model to convert the desired latency into the sampling parameter. To build the model, we perform offline profiling of the compute cluster. This model empirically establishes the relationship between the input size and the latency of cross product phase by computing the $\beta_{compute}$ parameter from the micro-benchmarks. Afterwards, we utilize Equation 6.7 to compute the sample sizes.

III: Approximation: Sampling submodule. After receiving the intersection of the inputs from the filtering module and the sampling parameter from the cost function submodule, the sampling submodule performs the sampling during the join as described in §6.4.3. We implemented the proposed sampling mechanism in this submodule by creating a new Spark PairRDD function *sampleDuringJoin()* that executes stratified sampling during the join.

The original *join()* function in Spark uses two operations: 1) *cogroup()* shuffles the data in the cluster, and 2) *cross-product* performs the final phase in join. In our *approxjoin()* function, we replace the second operation with *sampleDuringJoin()* that implements our mechanism described in §6.4.3 and Algorithm 11. Note that the data shuffled by the *cogroup()* function is the output of the filtering stage. As a result, the amount of shuffled data can be significantly reduced if the overlap fraction between datasets is small. Another thing to note is that *sampleDuringJoin()* also performs the query execution as described in Algorithm 11.

IV: Approximation: Error estimation submodule. After the query execution is performed in *sampleDuringJoin()*, the error estimation submodule implements the function *errorEstimation()* to compute the error bounds of the query result. The submodule also activates a feedback mechanism to re-tune the sample sizes in the sampling submodule to achieve the specified accuracy target as described in §6.4.2. We use the Apache Common Math

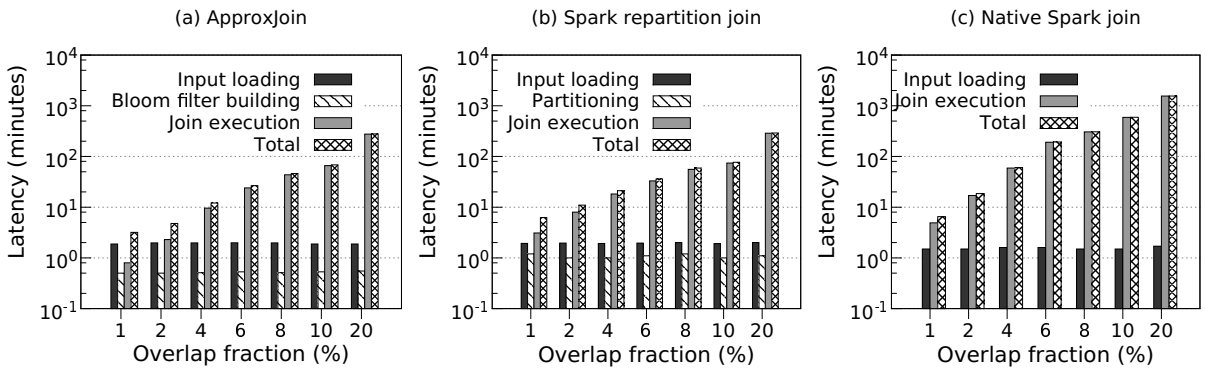

Figure 6.9 – The latency breakdown of: **(a)** ApproxJoin without sampling; **(b)** Spark repartition join; **(c)** Native Spark join.

library [158] to implement the error estimation mechanism described in §6.4.4.

6.6 Evaluation

In this section, we present the evaluation results of ApproxJoin based on micro-benchmarks and the TPC-H benchmark. In the next section, we will report evaluation based on real-world case studies.

6.6.1 Experimental Setup

Cluster setup. Our cluster consists of 10 nodes, each equipped with two Intel Xeon E5405 quad-core CPUs, 8GB memory and a SATA-2 hard disk, running Ubuntu 14.04.1.

Synthetic datasets. We analyze the performance of ApproxJoin using synthetic datasets following Poisson distributions with λ in the range of $[10, 10000]$. The number of distinct join keys is set to be proportional to the number of workers.

Metrics. We evaluate ApproxJoin using three metrics: latency, shuffled data size, and accuracy loss. Specifically, the latency is defined as the total time consumed to process the join operation; the shuffled data size is defined as the total size of the data shuffled across nodes during the join operation; the accuracy loss is defined as $(approx - exact)/exact$, where $approx$ and $exact$ denote the results from the executions with and without sampling, respectively.

6.6.2 Benefits of Filtering

The join operation in ApproxJoin consists of two main stages: (*i*) filtering stage for reducing shuffled data size, and (*ii*) sampling stage for approximate computing. In this section, we activate only the filtering stage (without the sampling stage) in ApproxJoin, and evaluate the benefits of the filtering stage.

Two-way joins. First, we report the evaluation results with two-way joins. Figure 6.9(a)(b)(c) show the latency breakdowns of ApproxJoin, Spark repartition join, and native Spark join,

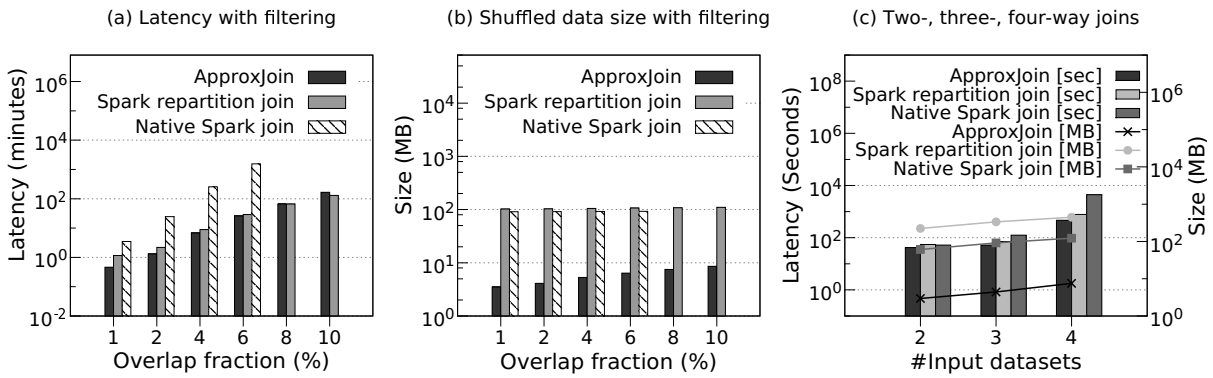

Figure 6.10 – Comparison between ApproxJoin and Spark join systems: **(a)** Latency with varying overlap fractions; **(b)** Shuffled data size with varying overlap fractions; **(c)** Latency and shuffled data size with varying # input datasets.

respectively. Unsurprisingly, the results show that building Bloom filters in ApproxJoin is quite efficient (only around 42 seconds) compared with the cross-product-based join execution (around 43× longer than building Bloom filters, for example, when the overlap fraction is 6%). The results also show that the cross-product-based join execution is fairly expensive across all three systems.

When the overlap fraction is less than 4%, ApproxJoin achieves 2× and 3× shorter latencies than Spark repartition join and native Spark join, respectively. However, with the increase of the overlap fraction, there is an increasingly large amount of data that has to be shuffled and the expensive cross-product operation cannot be eliminated in the filtering stage; therefore, the benefit of the filtering stage in ApproxJoin gets smaller. For example, when the overlap fraction is 10%, ApproxJoin speeds up only 1.06× and 8.2× compared with Spark repartition join and Spark native join, respectively. When the overlap fraction increases to 20%, ApproxJoin's latency does not improve (or may even perform worse) compared with the Spark repartition join. At this point, we need to activate the sampling stage of ApproxJoin to reduce the latency of the join operation, which we will evaluate in §6.6.3.

Multi-way joins. Next, we present the evaluation results with multi-way joins. Specifically, we first conduct the experiment with three-way joins whereby we create three synthetic datasets with the same aforementioned Poisson distribution. We measure the latency and the shuffled data size during the join operations in ApproxJoin, Spark repartition join and native Spark join, with varying overlap fractions. Figure 6.10(a) shows that, with the overlap fraction of 1%, ApproxJoin is 2.6× and 8× faster than Spark repartition join and native Spark join, respectively. However, with the overlap fraction larger than 8%, ApproxJoin does not achieve much latency gain (or may even perform worse) compared with Spark repartition join. This is because, similar to the two-way joins, the increase of the overlap fraction prohibitively leads to a larger amount of data that needs to be shuffled and cross-producted. Note also that, we do not have the evaluation results for native Spark join with the overlap fractions of 8% and 10%, simply because that system runs out of memory. In addition, Figure 6.10(b) shows that ApproxJoin significantly reduces the shuffled data size. For example, with the overlap fraction of 6%, ApproxJoin reduces the shuffled data size by 16.68× and 14.5× compared with Spark repartition join and native Spark join, respectively.

Next, we conduct experiments with two-way, three-way and four-way joins. In two-way joins, we use two synthetic datasets A and B that have an overlap fraction of 1%; in three-way joins, the three synthetic datasets A, B, and C have an overlap fraction of 0.33%, and the overlap fraction between any two of them is also 0.33%; in four-way joins, the four synthetic datasets have an overlap fraction of 0.25%, and the overlap fraction between any two of these datasets is also 0.25%.

Figure 6.10(c) presents the latency and the shuffled data size during the join operation with different numbers of input datasets. With two-way joins, ApproxJoin speeds up by $2.2\times$ and $6.1\times$, and reduces the shuffled data size by $45\times$ and $12\times$, compared with Spark repartition join and native Spark join, respectively. In addition, with three-way and four-way joins, ApproxJoin achieves even larger performance gain. This is because, with the increase of the number of input datasets, the number of non-join data items also increases; therefore, ApproxJoin gains more benefits from the filtering stage.

Scalability. Finally, we keep the overlap fraction of 1% and evaluate the scalability of ApproxJoin with different numbers of compute nodes. Figure 6.11(a) shows that ApproxJoin achieves a lower latency than Spark repartition join and native Spark join. With two nodes, ApproxJoin achieves a speedup of $1.8\times$ and $10\times$ over Spark repartition join and native Spark join, respectively. Meanwhile, with 8 nodes, ApproxJoin achieves a speedup of $1.7\times$ and $6\times$ over Spark repartition join and native Spark join.

6.6.3 Benefits of Sampling

As shown in previous experiments, ApproxJoin does not gain much latency benefit from the filtering stage when the overlap fraction is large. To reduce the latency of the join operation in this case, we activate the second stage of ApproxJoin, i.e., the sampling stage.

For a fair comparison, we re-purpose Spark's built-in sampling algorithm (i.e., stratified sampling via sampleByKey) to build a "sampling over join" mechanism for the Spark repartition join system. Specifically, we perform the stratified sampling over the join results after the join operation has finished in the Spark repartition join system. We then evaluate the performance of ApproxJoin, and compare it with this *extended* Spark repartition join system.

Latency. We measure the latency of ApproxJoin and the extended Spark repartition join with varying sampling fractions. Figure 6.11(b) shows that the Spark repartition join system scales poorly with a significantly higher latency as it could perform stratified sampling only after finishing the join operation. Even if we were to enable the Spark repartition join system to perform stratified sampling over the input datasets and then perform the join operation over these samples, this would come with a significant accuracy loss.

Accuracy. Next, we measure the accuracy of ApproxJoin and the extended Spark repartition join. Figure 6.11(c) shows that the accuracy losses in both systems decrease with the increase of sampling fractions, although ApproxJoin's accuracy is slightly worse than the Spark repartition join system. Note however that, as shown in Figure 6.11(b), ApproxJoin achieves an order of magnitude speedup compared with the Spark repartition join system since ApproxJoin performs sampling during the join operation.

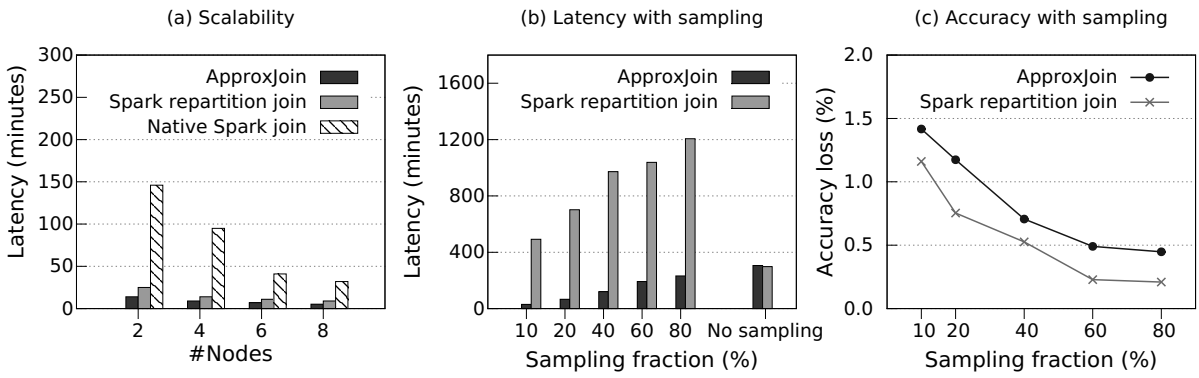

Figure 6.11 – Comparison between ApproxJoin and Spark join systems: **(a)** Scalability; **(b)** Latency; **(c)** Accuracy loss with varying sampling fraction. Note that Spark repartition join performs sampling after the join operation in this experiment.

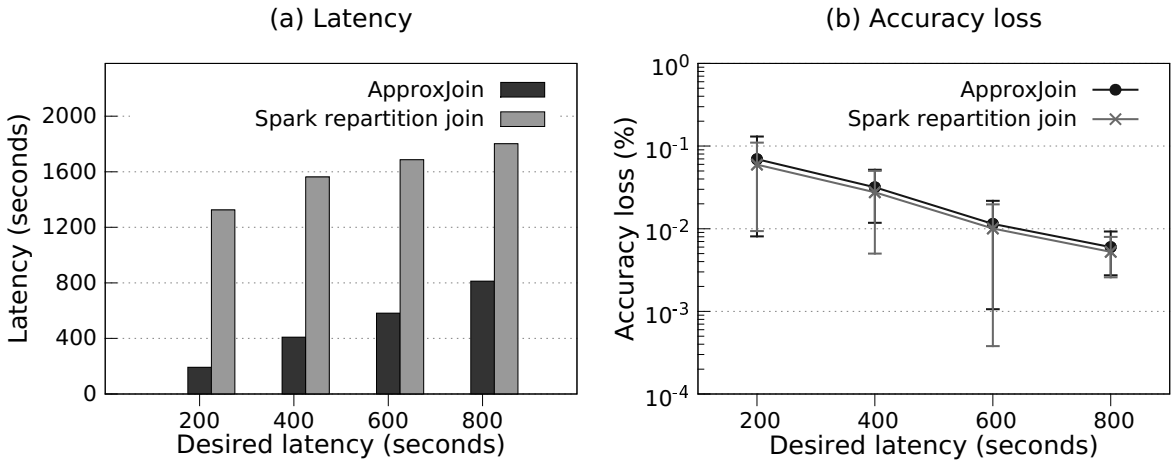

Figure 6.12 – Comparison between ApproxJoin and Spark repartition join with sampling after the join operation: **(a)** Latency; **(b)** Accuracy loss with varying latency budgets.

6.6.4 Effectiveness of Cost Function

ApproxJoin provides users with a query budget interface, and uses a cost function to convert the query budget into a sample size (see §6.4.2). In this experiment, a user sends ApproxJoin a join query along with a latency budget (i.e., the desired latency the user wants to achieve). ApproxJoin uses the cost function, whose parameter is set according to the micro-benchmarks ($\beta = 4.16 * 10^{-9}$ in our cluster), to convert the desired latency to the sample size. We measure the latency of ApproxJoin and the extended Spark repartition join in performing the join operations with the identified sample size. Figure 6.12(a) shows that ApproxJoin can rely on the cost function to achieve the desired latency quite well (with the maximum error being less than 12 seconds). Note also that, the Spark repartition join incurs a much higher latency than ApproxJoin since it performs the sampling after the join operation has finished. In addition, Figure 6.12(b) shows that ApproxJoin can achieve a very similar accuracy to the Spark

repartition join system.

6.6.5 Comparison with SnappyData using TPC-H

In this section, we evaluate ApproxJoin using TPC-H benchmark. TPC-H benchmark consists of 22 queries, and has been widely used to evaluate various database systems. We compare ApproxJoin with the state-of-the-art related system — SnappyData [186].

SnappyData is a hybrid distributed data analytics framework which supports a unified programming model for transactions, OLAP and data stream analytics. It integrates GemFine, an in-memory transactional store, with Apache Spark. SnappyData inherits approximate computing techniques from BlinkDB [7] (off-line sampling techniques) and the data synopses to provide interactive analytics. SnappyData does not support sampling over joins.

In particular, we compare ApproxJoin with SnappyData using the TPC-H queries $Q3$, $Q4$ and $Q10$ which contain join operations. To make a fair comparison, we only keep the join operations and remove other operations in these queries. We run the benchmark with a scale factor of $10\times$, i.e., 10GB datasets.

First, we use the TPC-H benchmark to analyze the performance of ApproxJoin with the filtering stage but without the sampling stage. Figure 6.13(a) shows the end-to-end latencies of ApproxJoin and SnappyData in processing the three queries. ApproxJoin is $1.34\times$ faster than SnappyData in processing $Q4$ which contains only one join operation. In addition, for the query $Q3$ which consists of two join operations, ApproxJoin achieves a $1.3\times$ speedup than SnappyData. Meanwhile, ApproxJoin speeds up by $1.2\times$ compared with SnappyData for the query $Q10$.

Next, we evaluate ApproxJoin with both filtering and sampling stages activated. In this experiment, we perform a query to answer the question "*what is the total amount of money the customers had before ordering*?". To process this query, we need to join the two tables $CUSTOMER$ and $ORDERS$ in the TPC-H benchmark, and then sum up the two fields $o\_totlaprice$ and $c\_acctbal$.

Since SnappyData does not support sampling over the join operation, in this experiment it first executes the join operation between the two tables $CUSTOMER$ and $ORDERS$, then performs the sampling over the join output, and finally calculates the sum of the two fields $o\_totalprice$ and $c\_acctbal$. Figure 6.13(b) presents the latencies of ApproxJoin and SnappyData in processing the aforementioned query with varying sampling fractions. SnappyData has a significantly higher latency than ApproxJoin, simply because it performs sampling only after the join operation finishes. For example, with a sampling fraction of 60%, SnappyData achieves a $1.77\times$ higher latency than ApproxJoin, even though it is faster when both systems do not perform sampling (i.e., sampling fraction is 100%). Note however that, sampling is inherently needed when one handles joins with large-scale inputs that require a significant number of cross-product operations. Figure 6.13(c) shows the accuracy losses of ApproxJoin and SnappyData. ApproxJoin achieves an accuracy level similar to SnappyData. For example, with a sampling fraction of 60%, ApproxJoin achieves an accuracy loss of 0.021%, while SnappyData achieves an accuracy loss of 0.016%.

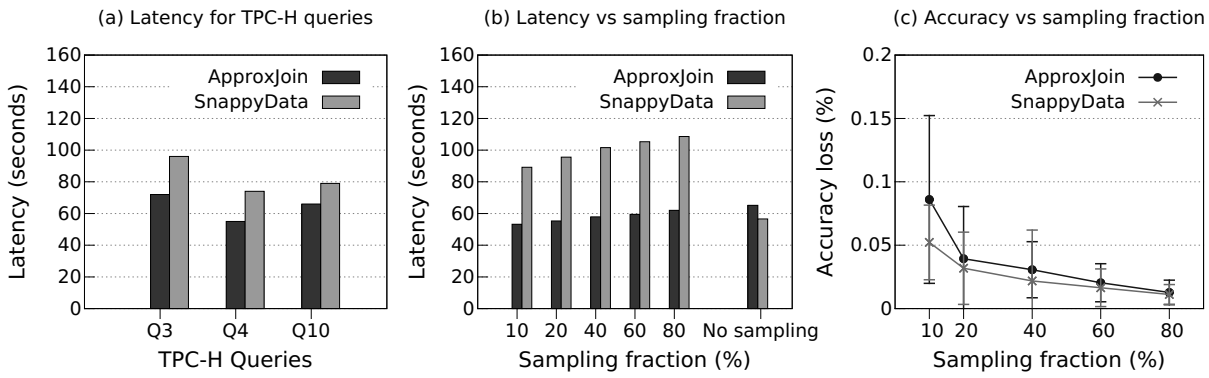

Figure 6.13 – Comparison between ApproxJoin and SnappyData: **(a)** Latency with with TPC-H queries; **(b)** Latency; **(c)** Accuracy loss with varying sampling fractions.

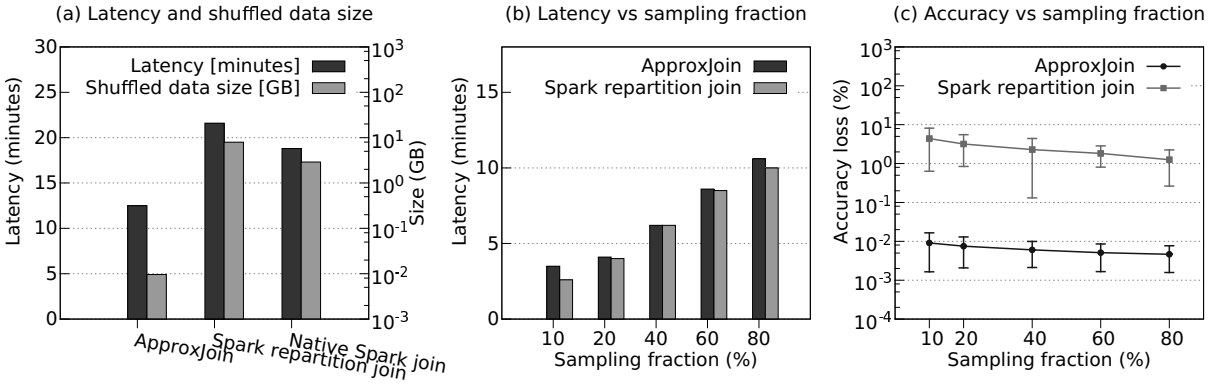

Figure 6.14 – Comparison between ApproxJoin and Spark join systems in network traffic case study: **(a)** Latency and shuffle data; **(b)** Latency with various sampling fractions; **(c)** Accuracy loss with various sampling fraction.

6.7 Case Studies

We evaluate ApproxJoin based on case studies with two real-world datasets: (a) network traffic dataset and (b) Netflix Prize dataset.

6.7.1 Network Traffic Analytics

Dataset. We use the CAIDA network traces [51] which were collected on the Internet backbone links in Chicago in 2015. In total, this dataset contains $115,472,322$ TCP flows, $67,098,852$ UDP flows, and $2,801,002$ ICMP flows. Here, a flow denotes a two-tuple network flow that has the same source and destination IP addresses.

Query. We use ApproxJoin to process the query: *What is the total size of the flows that appeared in all TCP, UDP and ICMP traffic?* To answer this query, we need to perform a join operation across TCP, UDP and ICMP flows.

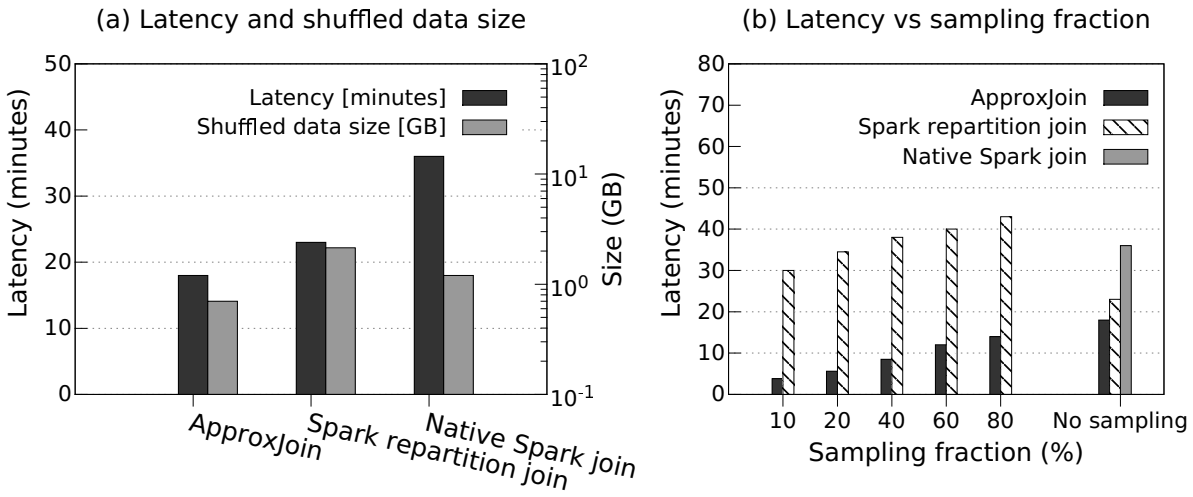

Figure 6.15 – Comparison between ApproxJoin and Spark join systems in Netflix case study: **(a)** Latency and shuffle data; **(b)** Latency with various sampling fractions.

Results. Figure 6.14(a) first shows the latency comparison between ApproxJoin (with filtering but without sampling), Spark repartition join, and native Spark join. ApproxJoin achieves a latency $1.72\times$ and $1.57\times$ lower than Spark repartition join and native Spark join, respectively. Interestingly, native Spark join achieves a lower latency than Spark repartition join. This is because the dataset is distributed quite uniformly across worker nodes in terms of the join-participating flow items, i.e., there is little data skew. Figure 6.14(a) also shows that ApproxJoin significantly reduces the shuffled data size by a factor of $300\times$ compared with the two Spark join systems.

Next, different from the experiments in §6.6, we extend Spark repartition join by enabling it to sample the dataset before the actual join operation. This leads to the lowest latency it could achieve. Figure 6.14(b) shows that ApproxJoin achieves a similar latency even to this extended Spark repartition join. In addition, Figure 6.14(c) shows the accuracy loss comparison between ApproxJoin and Spark repartition join with different sampling fractions. As the sampling fraction increases, the accuracy losses of ApproxJoin and Spark repartition join decrease, but not linearly. ApproxJoin produces around $42\times$ more accurate query results than the Spark repartition join system with the same sampling fraction.

6.7.2 Netflix Prize Analytics

Dataset. We also evaluate ApproxJoin based on the Netflix Prize dataset which includes around $100M$ ratings of $17,770$ movies by $480,189$ users. Specifically, this dataset contains $17,770$ files, one per movie, in the $training\_set$ folder. The first line of each such file contains $MovieID$, and each subsequent line in the file corresponds to a rating from a user and the date, in the form of $\langle UserID, Rating, Date\rangle$. There is another file $qualifying.txt$ which contains lines indicating $MovieID$, $UserIDs$ and the rating $Dates$.

Query. We perform the join operation between the dataset in $training\_set$ and the dataset in $qualifying.txt$ to evaluate ApproxJoin in terms of latency. Note that, we cannot find a

meaningful aggregation query for this dataset; therefore, we focus on only the latency but not the accuracy of the join operation.

Results. Figure 6.15(a) shows the latency and the shuffled data size of ApproxJoin (with filtering but without sampling), Spark repartition join, and native Spark join. ApproxJoin is $1.27\times$ and $2\times$ faster than Spark repartition join and native Spark join, respectively. The result in Figure 6.15(a) also shows that ApproxJoin reduces the shuffled data size by $3\times$ and $1.7\times$ compared with Spark repartition join and native Spark join, respectively. In addition, Figure 6.15(b) presents the latency comparison between these systems with different sampling fractions. For example, with the sampling fraction of 10%, ApproxJoin is $6\times$ and $9\times$ faster than Spark repartition join and native Spark join, respectively. Even without sampling (i.e., sampling fraction is 100%), ApproxJoin is still $1.3\times$ and $2\times$ faster than Spark repartition join and native Spark join, respectively.

6.8 Discussion

In our design, to set optimal values for the Bloom filter parameters, it requires to know in advance, the size of the input datasets. However, in reality, it is not always possible to know in advance the size of input datasets. This might lead to excessive calibration of the size of Bloom filters with orders of magnitude differences compared to the optimal setting. To overcome this problem, in this section, we discuss three alternative design choices for Bloom filters that we considered in ApproxJoin to filter the redundant items (step 1). To evaluate different variants of Bloom filters in terms of size and computation cost, we used a simulation with one input dataset containing $100K$ data items and built the corresponding Bloom filters. Figure 6.16 shows the size of each Bloom filter used.

Invertible Bloom filter. In addition to the membership check, an *Invertible Bloom Filter* (IBF)) [106] also allow to *g*et the list of all items present in the filter. As a result, the participating join items can be obtained by using the subtraction operation of the IBF. However, the IBF comes at a higher cost for computation and storage of the filter: Each cell in an IBF is not a single bit as a regular Bloom filter, but a data structure with a count maintaining the number of collisions and an invertible value of keys. Moreover, just as regular Bloom filters have false positives, there is a probability that a *g*et operation returns a "not found" result, although the data might still be in the filter, but due to collisions it cannot be found. This probability is the same as the false positive rate for the corresponding Bloom filter. Thus, the filtering step may have *false negatives* (due to the "not found" result), negatively affecting the join result. Note that such a false negative is not possible with the regular Bloom filters.

Counting Bloom filter. One can also use a *Counting Bloom Filter* (CBF) [44] for the filtering stage. CBFs also provide the *r*emove/subtraction operation, similar to IBFs, but not the *g*et operation. Unlike an IBF, each cell in a CBF is only a count that tracks the number of collisions. As a result, CBFs can be considerably smaller than IBFs (see Figure 6.16). However, the size CBFs is still significant larger than regular Bloom filter (see Figure 6.16).

Scalable Bloom filter. In our design, we need to know the size of the input datasets for configuring optimal values for the Bloom filters. In practice, however, this information may not always be available in advance. As a result, non-optimal values for Bloom filters may be chosen. To address this problem, we could employ *Scalable Bloom filters* (SBFs) [172], where

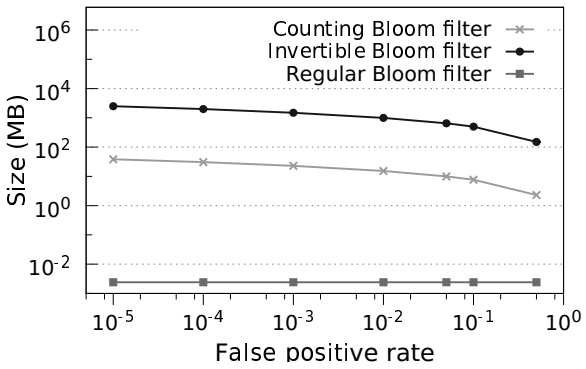

Figure 6.16 – Comparison of size of different Bloom filters with varying false positive rate.

the input dataset can be represented without knowing the number of data items to be put in the filter. This mechanism adapts to the growth of the input size by using a series of regular Bloom filters of increasing sizes and tighter error probabilities.

To build a global SBF (as our join filter), we need to merge local SBFs from all worker nodes in the cluster. Unfortunately, the current design and implementation of SBFs do not support the union operation to perform this merging. We show how to implement this merge operation by creating a pull request[2] to the SBF repository. Our implementation takes advantage of the fact that SBFs contain a set of regular Bloom filters. As a result, we perform the union operation between two SBFs by executing the union of regular Bloom filters under the hood.

6.9 Related Work

Over the last decade, approximate computing has been applied in various domains such as programming languages [29, 194], hardware design [193], and distributed systems [70, 121, 171]. Our techniques mainly target the databases research community [6, 7, 131, 132, 177, 204, 226]. In particular, various approximation techniques have been proposed to make trade-offs between required resources and output quality, including sampling [11, 100], sketches [73], and online aggregation [121, 171]. Chaudhari et al. provide a sampling over join mechanism by taking a sample of an input and considering all statistical characteristics and indices of other inputs [61]. AQUA [6] system makes use of simple random sampling to take a sample of joins of inputs that have primary key-foreign key relations. BlinkDB [7] proposes an approximate distributed query processing engine that uses stratified sampling [11] to support ad-hoc queries with error and response time constraints. SnappyData [186] and SparkSQL [27] adopt the approximation techniques from BlinkDB to support approximate queries. Quickr [204] deploys distributed sampling operators to reduce execution costs of parallel, ad-hoc queries that may contain multiple join operations. In particular, Quickr first injects sampling operators into the query plan and searches for an optimal query plan among sampled query plans to execute input queries.

Unfortunately, all of these systems require a priori knowledge of the inputs. For example, AQUA [6] requires join inputs to have primary key-foreign key relations. For another example,

[2]https://github.com/josephfox/pythonbloomfilter/pull/11

the sampling over join mechanism [61] needs the statistical characteristics and indices of inputs. Finally, BlinkDB [7] utilizes most frequently used column sets to perform off-line stratified sampling over them. Afterwards, the samples are cached, such that queries can be served by selecting the relevant samples and executing the queries over them. While useful in many applications, BlinkDB and these other systems cannot process queries over new inputs, where queries or inputs are typically not known in advance.

Ripple Join [115] implements online aggregation for joins. Ripple Join repeatedly takes a sample from each input. For every item selected, it is joined with all items selected in other inputs so far. Recently, Wander Join [146] improves over Ripple Join by performing random walks over the join data graph of a multi-way join. However, their approach crucially depends on the availability of indices, which are not readily available in "big data" systems like Apache Spark. In addition, the current Wander Join implementation is single-threaded, and parallelizing the walk plan optimization procedure is non-trivial. In this work, we proposed a simple but efficient sampling mechanism over joins which works not only on a single node but also in a distributed setting.

Recently, an approximate query processing (AQP) formulation [98] has been proposed to provide low-error approximate results without any preprocessing or a priori knowledge of inputs. The formulation based on probability theory allows to reuse results of past queries to improve the performance of future query processing. However, the current version of AQP formulation does not support joins.

The keynote speakers at SIGMOD 2017 [60, 136, 167] highlighted the challenges and opportunities in approximate query processing. In a follow up succinct blog post [26], Chaudhuri explains the reasons why AQP research does not have a significant impact in services and production. One of the reasons they indicate is that the current applicable techniques that take samples over joins by uniformly sampling the join inputs have both poor performance and accuracy. In this work, we tried to tackle this problem by proposing a stratified sampling mechanism to take samples during the join to achieve low latency and accuracy at the same time.

6.10 Conclusion

In this chapter, we strive to address the challenge associated in performing approximate joins for distributed data analytics systems. We achieve this by performing sampling during the join operation to achieve low latency as well as high accuracy. In particular, we employed a sketching technique (Bloom filters) to reduce the size of the shuffled data during the joins, and also proposed a stratified sampling mechanism that executes during the join in a distributed setting. We implemented our techniques in a system called ApproxJoin using Apache Spark and evaluated its effectiveness using a series of micro-benchmarks and real-world datasets.

Our evaluation shows that ApproxJoin significantly reduces query response times as well as data shuffled through the network during the join operation without losing accuracy of the query results due to sampling compared with the state-of-the-art systems. More specifically, ApproxJoin achieves a speedup of $6-9\times$ and reduces $5-82\times$ the shuffled data volume compared to Spark based systems with the same sampling fraction.

7 Conclusion and Future Work

Since the growth of digital data has been faster than the growth of computational power, *approximate computing* is emerging as an essential technique for big data analytics with interactive response time. This thesis shows how approximate computing can be applied in big data analytics systems to improve performance as well as achieve efficient resource utilization. Our approach makes use of sampling techniques to enable transparent, practical and efficient approximate computation in big data systems. Interestingly, this thesis contains cases that, by applying sampling techniques for approximate computing, not only achieve higher performance in data analytics but also strengthen users privacy. We conclude this thesis with a summary of our results, a few of lessons that we learned, and an outlook on future work.

7.1 Summary of Results

This thesis presents the design and implementation of the following four systems for approximate big data analytics:

- StreamApprox—a system for approximate stream analytics (Chapter 3). This system applies approximate computing for low-latency stream analytics in a transparent way. In addition, it is able to adapt to rapid fluctuations of input data streams. One of the key insights of this system is an online adaptive stratified reservoir sampling algorithm to produce approximate output with bounded error. This algorithm has the ability to perform "on-the-fly" sampling on the input data stream and can be parallelized naturally without requiring any form of synchronization among distributed workers.
- IncApprox—a data analytics system for incremental approximate computing (Chapter 4). This system extends StreamApprox by not only adopting approximate computing but also incremental computing in stream processing to achieve high-throughput and low-latency with efficient resource utilization. The key insight of this system is an online stratified sampling algorithm that uses self-adjusting computation to produce an incrementally updated approximate output with bounded error.
- PrivApprox—a privacy-preserving stream analytics system (Chapter 5). This system supports high-utility and low-latency data analytics and preserves user's privacy at the same time. The key idea of this system is the combination of (*i*) a randomized response mechanism to preserve the privacy of users and (*ii*) sampling techniques to achieve low-latency in data analytics. Interestingly, we proved that by applying the sampling techniques for approximate computing, our system actually strengthens users privacy.
- ApproxJoin—an approximate distributed joins system (Chapter 6). This system improves the performance of joins — critical but expensive operations in big data systems. The key insight of this system is a sketching technique (Bloom filter) to avoid shuffling non-

participating join data items over the network, and a sampling mechanism that executes during the join to obtain an unbiased representative sample of the join output.

Our experiences using micro-benchmarks and real-world case-studies show that our proposed approaches can significantly improve the performance of the-state-of-the-art big data analytics systems with efficient resource utilization without sacrificing the accuracy of analytics.

Since the digital data continues to grow exponentially and even faster than the Moore's law, we hope that our proposed techniques for approximate computing will be useful for big data analytics community to improve the performance of their systems and at the same time achieve the efficiency in resource usage.

Lessons Learned. The most important lesson learned from this thesis is that using native available sampling techniques may counteract the performance of big data analytics as highlighted in Chapter 3. The reason is that performing sampling over large-scale distributed datasets is a challenge and it also introduces significant overhead. However, if the sampling process can be performed in a proper manner, it significantly improves the performance of not only batch but also stream data analytics in terms of throughput and latency (see Chapter 3 and Chapter 4).

The second lesson learned is that computing accurate error bounds for approximate answers is critical in approximate data analytics systems since sampling techniques introduce error, and users always want to know how far the approximate results diverge from the exact results. The sampling techniques which are based on the statistics theory work fine in computing the error bounds for linear queries such as SUM/AVG, COUNT, and STDEV with certain assumptions on the input datasets. However, applying these techniques in a naive manner does not work for complex queries containing join operations as described in Chapter 6. The reason is that performing joins over dataset samples would not preserve statistical properties of the join output. Therefore, sampling over joins requires careful analysis of error bounds.

Finally, the surprising lesson learned from this thesis is that applying the sampling techniques can strengthen the privacy of users of modern online services. Chapter 5 shows that the combination of the sampling and randomized response techniques led us to achieve zero-knowledge privacy [101], a privacy bound tighter than the state-of-the-art differential privacy [82].

7.2 Future Work

Currently, we mainly use sampling techniques to build our approximate data analytics systems. These techniques support a wide range of query types not required to be fully known in advance. They allow executing queries over samples to get approximate results rather than over the original data. However, they perform poorly for queries asked for rare data items in the original data which are unlikely to be added to any sample, e.g., queries for abnormal data items for anomaly detection, distinct counts queries, and frequencies queries. Therefore, as a future work, we want to extend our systems to support various query types and achieve even better performance by combining sampling techniques and sketching techniques. We want to combine stratified sampling techniques with several sketches such as Count-Min sketch, HyperLogLog and MinHash Sketch. We applied this approach in the work of ApproxJoin by

combining stratified sampling and Bloom filter to improve the performance of joins. However, we also want to apply this approach in StreamApprox, IncApprox, and PrivApprox. Moreover, ApproxJoin is designed to improve the performance of joins between static datasets. As a future work, we will extend further our proposed mechanism in ApproxJoin to improve the performance of joins between not only static input datasets but also data streams.

In addition, in the architecture of PrivApprox, we currently assume that the aggregator is deployed on a trusted infrastructure. However, most likely, this will not be the case. Ideally, we would like to deploy the aggregator on an untrusted platform in the public cloud. To achieve this goal, as a future work, we want to design a trusted aggregator for privacy-preserving data analytics using a trusted computing platform such as Intel SGX [74].

Recently, a new computing model namely *Structured Streaming* has been proposed in Apache Spark. This computing model considers an input data stream as an append-only table [202] in which a new arriving data item is treated as a row appended to the table. With this computing model, Spark Structured Streaming is $3\times$ faster than Apache Flink [32]. Thus, we also want to implement our proposed sampling algorithm in StreamApprox and IncApprox into the Spark Structured Streaming to verify whether or not our approach can further improve the performance of the stream processing framework.

Next, we would like to close this thesis by raising several questions for future research.

Approximate computing for advanced analytics. Currently, our systems support linear queries. While they are useful for many statistics applications, users may also want to perform advanced analytics such as machine learning or graph processing, it is not clear whether the proposed approaches can be applied for these application domains. To answer this question, we need to investigate the workflow of these applications to determine which parts of the workflow can enable approximate computing. In addition, we need to define a new accuracy metric for these applications which is definitely a challenge.

Approximate computing for IoT applications. In the context of Internet of Things (IoT), a large number of IoT devices continue to generate a massive amount of data. To perform real-time analytics over this massive data, *edge computing* architecture has been proposed to move computations to the edges of networks which are closer to data sources. Unfortunately, most computing nodes at edges have limited capacity (e.g., computing, storage, and energy resources) to support real-time analytics over the data from IoT devices. Thus, to overcome the challenge, we may apply approximate computing for edge analytics. A natural research question is: can we apply sampling techniques to achieve efficiency for edge analytics as well as reduce the communication cost (i.e., transferring only selected data items) from edges to the cloud?

Approximate computing for efficiency and security. Nowadays, cloud computing has become a popular platform for delivering modern online services. Since these services store and maintain the sensitive data of their customers on clouds, security becomes a major concern. In this context, trusted execution environments [74] allow us to make the cloud-based services more resilient against security attacks. However, to enable the security property for data analytics, this approach may introduce new sources of overhead for data analytics systems [28]. Thus, a research question arises naturally: can approximate techniques help us to remove this overhead? In other words, does combining approximate computing and trusted execution based mechanisms allow modern online services to achieve not only efficiency but also security at the same time? In addition, can we measure security the same way we

measure privacy? These questions are compelling but also challenging for future research.

A Appendix

A.1 Privacy Analysis and Proofs

In this section, we present the privacy analysis and proofs of PrivApprox system (see Chapter 5). Note that these mathematical proofs were done by Martin Beck. PrivApprox achieves three privacy properties (*i*) zero-knowledge privacy, (*ii*) anonymity, and (*iii*) unlinkability as introduced in §5.3.2.

Property # I: Zero-knowledge privacy. We show that the system designed in Section 5.4 achieves ϵ_{zk}-zero-knowledge privacy and prove a tighter bound for ϵ_{dp}-differential privacy, than what generally follows from zero-knowledge privacy [101]. The basic idea is that all data from the clients is already differentially private due to the use of randomized response. Furthermore, the combination with pre-sampling at the clients makes it zero-knowledge private as well. Following the privacy definitions, any computation upon the results of differentially, as well as, zero-knowledge private algorithms is guaranteed to be private again.

In the following paragraphs we show that:

- Independent and identically distributed (IID) sampling decomposes easily and is self-commutative. See Lemma 1.
- Sampling and randomized response mechanisms commute. See Lemma 2.
- Pre-sampling and post-sampling can be traded arbitrarily around a randomized response mechanism. See Corollary 1.
- A ϵ_{zk}-zero-knowledge privacy bound for our system. See Theorem 1
- A ϵ_{dp}-differential privacy bound for our system. See Theorem 2
- Our differential privacy bound is tighter than the general differential privacy bound derived from a zero-knowledge private algorithm. See Proposition 1.

Intuitively, differential privacy limits the information that can be learned about any individual i by the difference occurring from either including i's sensitive data in a differentially private computation or not. Zero-knowledge privacy on the other hand also gives the adversary access to aggregate information about the remaining individuals denoted as D_{-i}. Essentially everything that can be learned about individual i can also be learned by having access to some aggregate information upon D_{-i}.

Let $San()$ be a sanitizing algorithm, which takes a database D of sensitive attributes a_i of individuals $i \in \mathrm{P}$ from a population P as input and outputs a differentially private or zero-knowledge private result $San(D)$. For brevity, we write $San_A(D)$ for the output of the adversary A with arbitrary external input z and access to $San(D)$. Similarly, we omit the explicit usage of the external information z as input to the simulator S, as well as the total size of the database. See [102] Definition 1 and 2 for the extended notation. Let $O \subseteq Range(San_A)$

be any set of possible outputs. ϵ_{dp}-differential privacy can be defined as

$$\Pr[San_A(D) \in O] \leq e^{\epsilon_{dp}} \cdot \Pr[San_A(D_{-i}) \in O] \tag{A.1}$$

while ϵ_{zk}-zero-knowledge privacy is defined as

$$\Pr[San_A(D) \in O] \leq e^{\epsilon_{zk}} \cdot \Pr[S(T(D_{-i}), |D|) \in O]. \tag{A.2}$$

Before proving the desired properties, we need to introduce some notation. Let $D = \{a_i\}$ be a database of sensitive attributes of individuals $i \in \mathrm{P}$. For ease of presentation and without loss of generality we restrict the individual's sensitive attribute to a boolean value $a_i \in \{0, 1\}$ and $\mathrm{D} = a_{i'}$ for all $i' \in \mathrm{P}$. Furthermore, let $\mathsf{D}(\mathrm{D}) = \{U : U \subseteq \mathrm{D}\}$ be the super-set of all possible databases and $Sam(D, u)\colon \mathsf{D}(\mathrm{D}) \times (0, 1) \to \mathsf{D}(\mathrm{D})$ be a randomized algorithm that i.i.d. samples rows or individuals with their sensitive attributes from database D with probability s without replacement. Let $San(D, p, q)\colon \mathsf{D}(\mathrm{D}) \times (0, 1) \times (0, 1) \to \mathsf{D}(\mathrm{D})$ be a two-coin randomized response algorithm that decides for any individual i' in database D with probability p if it should be part of the output. If it is not included in the output, the result of tossing a biased coin (coming up heads with probability q) is added to the output.

Lemma 1. (Decompose and commute sampling) *Let $s = uv$ with $s, u, v \in (0, 1)$ being sampling probabilities for a sampling function $Sam()$. It follows that $Sam()$ can be composed and decomposed easily and is self-commutative.*

$$\begin{aligned} Sam(D, s) &\approx Sam(Sam(D, u), v) \\ &\approx Sam(Sam(D, v), u). \end{aligned}$$

Proof. Let Sam_u, Sam_v be sampling algorithms that sample rows i.i.d. from a database with probability u and v respectively. By applying $Sam_u(D)$, any row in D has probability u of being sampled. The probability for any row in D being sampled by Sam_v is equivalently v. Using function composition the probability for any row in D being sampled by $Sam_s = (Sam_u \circ Sam_v)(D)$ is

$$s = uv. \tag{A.3}$$

From multiplication being commutative ($u \cdot v = v \cdot u$) follows that Sam_u and Sam_v commute, that is $Sam_u \circ Sam_v = Sam_v \circ Sam_u$. This is true for deterministic functions and can easily be extended to randomized functions described as random variables, as random variables are commutative under addition and multiplication. For ease of presentation and without loss of generality we keep the notion of functions instead of random variables. Let $Sam_s(D) = Sam(D, s)$ be a sampling function that samples rows i.i.d. from a given database D with probability s. Decomposing sampling function $Sam_s()$ with probability s into two functions with probabilities u and v follow from (A.3). It also follows that two sampling functions with probabilities u, v can be composed into a single sampling function with sampling probability s. □

Lemma 2. (Commutativity of sampling and randomized response) *Given a sampling algorithm $Sam()$ and a randomized response algorithm $San()$, the result of the pre-sampling algorithm $F_{pre}(D, s, p, q) = San(Sam(D, s), p, q)$ is statistically indistinguishable from the result of the post-sampling algorithm $F_{post}(D, s, p, q) = Sam(San(D, p, q), s)$. It follows that sampling and randomized response commute under function composition: $Sam \circ San = San \circ Sam$.*

Proof. For any individual i having $a_i \in D$ we have to consider eight different possible cases. In case the sampling algorithm $Sam()$ decides to not sample i, it obviously doesn't matter if it gets removed before the randomized response algorithm is run of afterwards. We thus condition on $Sam()$ to include i in the output.

1. Let us first consider the case that $San()$ outputs the real value for individual i. As $Sam()$ is fixed to output i independent of its value, there is no difference between F_{pre} and F_{post}.
2. In case $San()$ outputs a randomized answer $Sam()$ again is not influenced by the outcome of any of the coin tosses and passes i along to the output. This is of course also independent of the actual randomized result.

This concludes the proof that sampling and randomized response are independent regarding their order of execution and thus commute. □

Corollary 1. (Arbitrary sampling around randomized response) *Let $s = uv$ for $s, u, v \in (0, 1)$ be sampling probabilities for a sampling function $Sam()$ and $San()$ be a two-coin randomized response mechanism with probabilities (p, q). Sampling can be arbitrarily traded between pre-sampling and post-sampling around the randomized response mechanism San.*

$$\begin{aligned} San(Sam(D, s), p, q) &\approx Sam(San(Sam(D, u), p, q), v) \\ &\approx Sam(San(D, p, q), s). \end{aligned}$$

Proof. This follows directly from applying Lemma 1 and Lemma 2. □

We will now give a bound on ϵ_{zk} for the privacy of our system under the zero-knowledge privacy setting, as well as derive a tighter bound for (ϵ_{dp})-differential privacy, than the bound that generally follows from zero-knowledge privacy.

Theorem 1. (ϵ_{zk}-zero-knowledge privacy) *Let A be an algorithm that applies sampling with probability s, together with a two-coin randomized response algorithm using probabilities (p, q). A is ϵ_{zk}-zero-knowledge private with*

$$\epsilon_{zk} = ln\left(s\frac{2-s}{1-s}\left(\frac{p+(1-p)\,q}{(1-p)\,q}\right)+(1-s)\right). \tag{A.4}$$

The system design is described in Section 5.4.

Proof. From [102], Theorem 1 follows that a (k, ϵ_{rr})-crowd-blending private mechanism combined with a pre-sampling using probability s achieves ϵ-zero-knowledge privacy with

$$\epsilon_{zk} = ln\left(s \cdot \left(\frac{2-s}{1-s}e^{\epsilon_{rr}}\right)+(1-s)\right).$$

We omit the description for the additive error δ, which can be derived equivalently from [102] Theorem 1. Following Proposition 1 from [102] every ϵ_{rr}-differentially private mechanism is also k, ϵ_{rr}-crowd-blending private, thus randomized response being an ϵ_{rr}-differentially

private mechanism, also satisfies (k, ϵ_{rr})-crowd-blending privacy with $k = 1$. Combining both results with (5.8) $\epsilon_{rr} = ln\left(\frac{p+(1-p)q}{(1-p)q}\right)$ gives an

$$\epsilon_{zk} = ln\left(s \cdot \left(\frac{2-s}{1-s}\left(\frac{p+(1-p)q}{(1-p)q}\right)\right) + (1-s)\right)$$

zero-knowledge private mechanism for randomized response combined with pre-sampling. Using Corollary 1 we can replace pre-sampling with a combination of pre- and post-sampling (with probabilities u, v respectively and $s = u \cdot v$) while keeping ϵ_{zk} fixed. We thus have

$$\epsilon_{zk} = ln\left(uv\frac{2-uv}{1-uv}\left(\frac{p+(1-p)\,q}{(1-p)\,q}\right) + (1-uv)\right).$$

□

If we do not aim at achieving zero-knowledge privacy, we can fall back to differential privacy using the result from [101], Proposition 3, which states that any ϵ-zero-knowledge private algorithm is also 2ϵ-differentially private. Using the results from sampling secrecy [135], which achieve a privacy boost by applying pre-sampling before using a differentially private algorithm, we derive a tighter bound for differential privacy, than what follows generally from zero-knowledge privacy.

Theorem 2. (ϵ_{dp}-differential privacy) *Let A be an algorithm that applies sampling with probability s, followed by a two-coin randomized response algorithm using probabilities (p, q). A is ϵ_{dp}-differentially private with*

$$\epsilon_{dp} = ln\left(1 + s\left(\frac{p+(1-p)\,q}{(1-p)\,q} - 1\right)\right). \tag{A.5}$$

Proof. We use the result from [78], Proof of Lemma 3, which bounds an ϵ_{rr}-differential private algorithm combined with pre-sampling using probability s by $\epsilon_{dp} = ln(1 + s(exp(\epsilon_{rr}) - 1))$. Let $\epsilon_{rr} = ln\left(\frac{p+(1-p)q}{(1-p)q}\right)$ be the bound derived for randomized response, we get

$$\epsilon_{dp} = ln\left(1 + s\left(\frac{p+(1-p)\,q}{(1-p)\,q} - 1\right)\right).$$

Applying Corollary 1 we derive an ϵ_{dp} bound for the combination of pre-sampling, randomized response and post-sampling of:

$$\epsilon_{dp} = ln\left(1 + (uv)\left(\frac{p+(1-p)\,q}{(1-p)\,q} - 1\right)\right).$$

□

Proposition 1. (Tighter ϵ_{dp}-differential privacy bound) *The bound ϵ_{dp} for differential privacy of a sampled randomized response system derived in Theorem 2 is tighter than ϵ_{zk}-differential privacy, which is again tighter than the general $2\epsilon_{zk}$-differential privacy bound that follows from ϵ_{zk}-zero-knowledge privacy [101].*

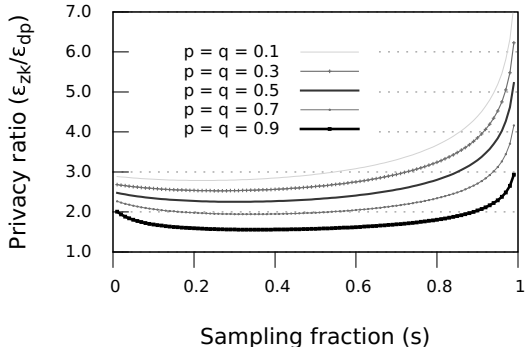

Figure A.1 – Ratio of $\frac{\epsilon_{zk}}{\epsilon_{dp}}$ depending on the sampling parameter s for different values p and q.

We directly proof Proposition 1 by comparing ϵ_{dp} from Theorem 2 with ϵ_{zk} from Theorem 1. As we want to prove a bound that is tighter than ϵ, we drop the factor of 2. This is possible because a ϵ-differentially private algorithm is also 2ϵ-differentially private. If we succeed in proving a bound tighter than ϵ, then 2ϵ-differential privacy is trivially fulfilled.

Proof. Proposition 3 from [101] states that every ϵ-zero-knowledge private algorithm is also 2ϵ-differentially private. Using Theorem 1 we get a ϵ_{zk}-differentially private system with $2\epsilon_{zk} = 2ln\left(s\frac{2-s}{1-s}\left(\frac{p+(1-p)q}{(1-p)q}\right)+(1-s)\right)$. Theorem 2 proves a bound of $\epsilon_{dp} = ln\left(1+s\left(\frac{p+(1-p)q}{(1-p)q}-1\right)\right)$. Let $e^{\epsilon_{rr}} = \frac{p+(1-p)q}{(1-p)q}$. Putting together Theorem 2, Theorem 1, Proposition 1 and Proposition 3 [101] we have:

$$
\begin{aligned}
ln\left(1+s\left(\frac{p+(1-p)\,q}{(1-p)\,q}-1\right)\right) &< ln\left(s\frac{2-s}{1-s}\left(\frac{p+(1-p)\,q}{(1-p)\,q}\right)+(1-s)\right)\\
s\left(\frac{p+(1-p)\,q}{(1-p)\,q}-1\right) &< \frac{2-s}{1-s}s\left(\frac{p+(1-p)\,q}{(1-p)\,q}-1\right)\\
s\left(e^{\epsilon_{rr}}-1\right) &< \frac{2-s}{1-s}s\left(e^{\epsilon_{rr}}-1\right)\\
1 &< \frac{2-s}{1-s}
\end{aligned}
$$

As $s \in (0,1)$ is the sampling parameter with a minimal right side for $s_{min} = \text{argmin}_{s\in(0,1)}\left(\frac{2-s}{1-s}\right) = 0$ the above inequality becomes $1 < 2$, which holds and concludes the proof. □

Relation of differential privacy and zero-knowledge privacy. Zero-knowledge privacy and differential privacy describe the advantage ϵ of an adversary in learning information about individual i by using an output from an algorithm $San()$ running over database D containing sensitive information $a_i \in D$ of individual i compared to using a result of a second — possibly different — algorithm $San'()$ running over D_{-i}. Zero-knowledge privacy is a strictly stronger privacy metric through the additional access to aggregate information of the remaining database D_{-i} compared to differential privacy [101]. By intuition, as differential privacy is a special case of zero-knowledge privacy and the adversary aims at maximizing its advantage, the advantage of an adversary in the zero-knowledge model is at least as high and possibly higher than the advantage of an adversary in the differential privacy model: $\epsilon_{zk} \geq \epsilon_{dp}$. Figure A.1 draws the ratio $\frac{\epsilon_{zk}}{\epsilon_{dp}}$ between the zero-knowledge privacy level ϵ_{zk} and the differential privacy level ϵ_{dp} given identical parameters p, q and s. Put differently, as the adversary is allowed to do

more in the zero-knowledge model, the privacy level is lower, which is reflected by a higher ϵ_{zk} value compared to the differential privacy level ϵ_{dp} — given identical system parameters.

Property # II: Anonymity. We make the following assumptions to achieve the remaining two privacy properties:

(A1) At least two out of the n proxies are not colluding.

(A2) The aggregator does not collude with any of the proxies.

(A3) The aggregator and analysts cannot — at the same time — observe the communication around the proxies.

(A4) The adversary, seen as an algorithm, lies within the polynomial time complexity class.

To provide anonymity, we require that no system component (proxy, aggregator, analyst) can relate a query request or answer to any of the clients. To show the fulfillment of that requirement we take the view of all three parties.

a) A *proxy* can of course link the received data stream to a client, as it is directly connected. However, as the data stream is encrypted, it would need to have the plaintext query request or response for the received data stream. To get the plaintext the proxy would either need to break symmetric cryptography, which breaks assumption (A4), collude with *all* other proxies for decryption, which breaks assumption (A1) or collude with the aggregator to learn the plaintext, which breaks assumption (A2).

b) Anonymity against the *aggregator* is achieved by source-rewriting, which is a standard anonymization technique typically used by proxies and also builds the basis for anonymization schemes [79, 188]. To break anonymity the aggregator must be a global, passive attacker, which means that he is able to simultaneously listen to incoming and outgoing traffic of any proxy. This would violate assumption (A3). The other possibility to bridge the proxies is by colluding with any of them — breaking assumption (A2).

c) The *analyst* knows the query request, but doesn't get to learn the single query answers. He needs to collude with the aggregator, to see single responses. Thus the problem reduces to breaking anonymity from the view of the aggregator. Collusion with the aggregator and any proxy would break assumption (A2). Collusion with up to $n - 1$ proxies reduces to breaking anonymity from the proxy view.

Property # III: Unlinkability. Unlinkability is provided by the source-rewriting scheme as in anonymity. Breaking unlinkability on any *proxy* is similar to breaking anonymity, as the proxy would need to get the plaintext query. The *aggregator* only gets query results, but no source information, as this is hidden by the anonymization scheme. The query results sent by the clients also do not contain linkable information, just identically structured answers without quasi-identifiers. The view of the *analyst* doesn't receive responses, so it must collude with either a proxy or the aggregator, effectively reducing to the same problem as described above.

Bibliography

[1] 68-95-99.7 *Rule*. https://en.wikipedia.org/wiki/68-95-99.7_rule. Accessed: Nov, 2017.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2016.

[3] Umut A. Acar. "Self-Adjusting Computation". PhD thesis. Carnegie Mellon University, 2005.

[4] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. "An Experimental Analysis of Self-Adjusting Computation". In: ACM *Transactions on Programming Languages and Systems* (TOPLAS). 2009.

[5] Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. "Dynamic Well-Spaced Point Sets". In: *Proceedings of the 26th Annual Symposium on Computational Geometry* (SoCG). 2010.

[6] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. "The Aqua Approximate Query Answering System". In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (SIGMOD). 1999.

[7] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data". In: *Proceedings of the* ACM *European Conference on Computer Systems* (EuroSys). 2013.

[8] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. "Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems". In: *Proceedings of the* ACM *SIGMOD International Conference on Management of Data* (SIGMOD). 2014.

[9] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2015.

[10] Istemi Ekin Akkus, Ruichuan Chen, Michaela Hardt, Paul Francis, and Johannes Gehrke. "Non-tracking web analytics". In: *Proceedings of the ACM Conference on Computer and Communications Security* (CCS). 2012.

[11] Mohammed Al-Kateb and Byung Suk Lee. "Stratified Reservoir Sampling over Heterogeneous Data Streams". In: *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management* (SSDBM). 2010.

[12] Mohammed Al-Kateb, Byung Suk Lee, and X Sean Wang. "Adaptive-size reservoir sampling over data streams". In: *Proceedings of the 19th International Conference on Scientific and Statistical Database Management* (SSBDM). 2007.

[13] Susanne Albers. "Online algorithms: a survey". In: *Mathematical Programming*. 2003.

[14] *Amazon Kinesis Streams*. https://aws.amazon.com/kinesis/. Accessed: Nov, 2017.

[15] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. "Reining in the Outliers in Map-reduce Clusters Using Mantri". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2010.

[16] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. "End-to-end Performance Isolation Through Virtual Datacenters". In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2014.

[17] *Apache Flink*. https://flink.apache.org/. Accessed: Nov, 2017.

[18] *Apache Flume*. https://flume.apache.org/. Accessed: Nov, 2017.

[19] *Apache Hadoop*. http://hadoop.apache.org/. Accessed: Nov, 2017.

[20] *Apache S4*. http://incubator.apache.org/s4. Accessed: Nov, 2017.

[21] *Apache Spark*. https://spark.apache.org. Accessed: Nov, 2017.

[22] *Apache Spark MLib*. http://spark.apache.org/mllib/. Accessed: Nov, 2017.

[23] *Apache Spark Streaming*. http://spark.apache.org/streaming. Accessed: Nov, 2017.

[24] *Apache Storm*. http://storm-project.net/. Accessed: Nov, 2017.

[25] *Apache ZooKeeper*. https://zookeeper.apache.org/. Accessed: Nov, 2017.

[26] *Approximate query processing where do we go from here?* http://wp.sigmod.org/?p=2183. Accessed: Nov, 2017.

[27] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the International Conference on Management of Data* (SIGMOD). 2015.

[28] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2016.

[29] Woongki Baek and Trishul M. Chilimbi. "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2010.

[30] K. Bakshi. "Considerations for big data: Architecture and approach". In: *2012 IEEE Aerospace Conference*. 2012.

[31] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. "Memory-efficient Hash Joins". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 2014.

[32] *Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems*. https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html. Accessed: Nov, 2017.

[33] Pramod Bhatotia. "Incremental Parallel and Distributed Systems". PhD thesis. Max Planck Institute for Software Systems (MPI-SWS), 2015.

[34] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. "Shredder: GPU-Accelerated Incremental Storage and Computation". In: *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. 2012.

[35] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. "Incoop: MapReduce for Incremental Computations". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2011.

[36] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Bjoern Brandenburg, and Rodrigo Rodrigues. "iThreads: A Threading Library for Parallel Incremental Computation". In: *proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[37] Pramod Bhatotia, Alexander Wieder, Istemi Ekin Akkus, Rodrigo Rodrigues, and Umut A. Acar. "Large-scale incremental data processing with change propagation". In: *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*. 2011.

[38] Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. "Slider: Incremental Sliding Window Analytics". In: *Proceedings of the 15th International Middleware Conference (Middleware)*. 2014.

[39] Pramod Bhatotia, Marcel Dischinger, Rodrigo Rodrigues, and Umut A. Acar. *Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis*. Tech. rep. MPI-SWS-2012-004. MPI-SWS, 2012.

[40] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. "A Comparison of Join Algorithms for Log Processing in MaPreduce". In: *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*. 2010.

[41] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM*. 1970.

[42] Avrim Blum, Katrina Ligett, and Aaron Roth. "A learning theory approach to non-interactive database privacy". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing (STOC)*. 2008.

[43] Avrim Blum, Cynthia Dwork, Frank McSherry, and Kobbi Nissim. "Practical privacy: the SuLQ framework". In: *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*. 2005.

[44] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. "An Improved Construction for Counting Bloom Filters". In: *Proceedings of the 14th Conference on Annual European Symposium (ESA)*. 2006.

[45] Dhruba Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website*. 2007.

[46] Gerth Stolting Brodal and Riko Jacob. "Dynamic Planar Convex Hull". In: *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2002.

[47] Paul G Brown and Peter J Haas. "Techniques for warehousing of sample data". In: *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. 2006.

[48] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. "HaLoop: Efficient Iterative Data Processing on Large Clusters". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 2010.

[49] Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. "Sample Size Selection in Optimization Methods for Machine Learning". In: *Math. Program.* 2012.

[50] C. Olston et al. "Nova: Continuous Pig/Hadoop Workflows". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2011.

[51] CAIDA. *The CAIDA UCSD Anonymized Internet Traces 2015 (equinix-chicago-dirA)*. http://www.caida.org/data/passive/passive_2015_dataset.xml. 2015.

[52] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok. "A Survey on Internet Traffic Identification". In: *Communications Surveys Tutorials, IEEE*. 2009.

[53] Rick Cattell. "Scalable SQL and NoSQL Data Stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2011.

[54] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: Easy, Efficient Data-parallel Pipelines". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2010.

[55] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. "Private and Continual Release of Statistics". In: *ACM Trans. Inf. Syst. Secur.* 2011.

[56] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. "Differentially Private Continual Monitoring of Heavy Hitters from Distributed Streams". In: *Proceedings of the 12th International Conference on Privacy Enhancing Technologies (PETS)*. 2012.

[57] Reiss Charles, Tumanov Alexey, Ganger Gregory, H. Katz Randy, and Kozuch Michael. *Towards understanding heterogeneous clouds at scale: Google trace analysis*. Technical Report. 2012.

[58] Kamalika Chaudhuri and Nina Mishra. "When Random Sampling Preserves Privacy". In: *Proceedings of the 26th Annual International Conference on Advances in Cryptology (CRYPTO)*. 2006.

[59] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. "Optimized Stratified Sampling for Approximate Query Processing". In: *ACM Trans. Database Syst.* 2007.

[60] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. "Approximate Query Processing: No Silver Bullet". In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2017.

[61] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. "On Random Sampling over Joins". In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1999.

[62] Ruichuan Chen, Istemi Ekin Akkus, and Paul Francis. "SplitX: High-performance Private Analytics". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2013.

[63] Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johanes Gehrke. "Towards Statistical Queries over Distributed Private User Data". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[64] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. "Kineograph: taking the pulse of a fast-changing and connected world". In: *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 2012.

[65] Y.-J. Chiang and R. Tamassia. "Dynamic Algorithms in Computational Geometry". In: *Proceedings of the IEEE*. 1992.

[66] Benoit Claise. *Cisco systems NetFlow services export version 9*. https://tools.ietf.org/html/rfc3954. 2004.

[67] Edith Cohen. "Size-estimation framework with applications to transitive closure and reachability". In: *Journal of Computer and System Sciences*. 1997.

[68] Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer, 2001.

[69] *ComScore Reaches $14 Million Settlement in Electronic Privacy Class Action*. http://www.alstonprivacy.com/comscore-reaches-14-million-settlement-in-electronic-privacy-class-action. Accessed: Nov, 2017. url: http://www.alstonprivacy.com/comscore-reaches-14-million-settlement-in-electronic-privacy-class-action.

[70] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. "MapReduce Online". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2010.

[71] Graham Cormode. “Data Sketching”. In: *Commun. ACM*. 2017.

[72] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. “Finding Hierarchical Heavy Hitters in Data Streams”. In: *Proceedings of the 29th International Conference on Very Large Data Bases* (VLDB). 2003.

[73] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches”. In: *Found. Trends databases*. 2012.

[74] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptology ePrint Archive*. 2016.

[75] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2004.

[76] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. “Handbook on Data Structures and Applications”. In: Chapman & Hall/CRC, 2004.

[77] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. “Handbook on Data Structures and Applications, Chapter 36: Dynamic Graphs”. In: Chapman & Hall/CRC, 2005.

[78] *Differential privacy and the secrecy of the sample*. Sept. 2009. url: https://adamdsmith.wordpress.com/2009/09/02/sample-secrecy/.

[79] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. DTIC Document, 2004.

[80] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. “On Sequential Monte Carlo Sampling Methods for Bayesian Filtering”. In: *Statistics and Computing*. 2000.

[81] John R. Douceur. “The Sybil Attack”. In: *Proceedings of 1st International Workshop on Peer-to-Peer Systems* (IPTPS). 2002.

[82] Cynthia Dwork. “Differential privacy”. In: *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, part II* (ICALP). 2006.

[83] Cynthia Dwork and Aaron Roth. “The Algorithmic Foundations of Differential Privacy”. In: *Foundations and Trends in Theoretical Computer Science*. 2014.

[84] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: *Proceedings of the Third conference on Theory of Cryptography* (TCC). 2006.

[85] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. “Differential privacy under continual observation”. In: *Proceedings of the ACM Symposium on Theory of Computing* (STOC). 2010.

[86] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. “Our Data, Ourselves: Privacy Via Distributed Noise Generation”. In: *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques* (EUROCRYPT). 2006.

[87] Darius M. Dziuda. *Data mining for genomics and proteomics: analysis of gene and protein expression data*. John Wiley & Sons, 2010.

[88] B. Efron and R. Tibshirani. "Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy". In: *Statistical Science*. Institute of Mathematical Statistics, 1986.

[89] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. "Twister: A Runtime for Iterative MapReduce". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*. 2010.

[90] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. "Dynamic graph algorithms". In: *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[91] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-Purpose Approximate Programs". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2012.

[92] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm". In: *AofA: Analysis of Algorithms*. 2007.

[93] *Flink Gelly*. https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/. Accessed: Nov, 2017.

[94] *Flink SQL*. https://ci.apache.org/projects/flink/flink-docs-master/dev/table/sql.html. Accessed: Nov, 2017.

[95] *FlinkML*. https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/ml/. Accessed: Nov, 2017.

[96] James Alan Fox and Paul E Tracy. *Randomized response: a method for sensitive surveys*. Beverly Hills California Sage Publications, 1986.

[97] Arik Friedman, Izchak Sharfman, Daniel Keren, and Assaf Schuster. "Privacy-Preserving Distributed Stream Monitoring". In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2014.

[98] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. "Revisiting Reuse for Approximate Query Processing". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 2017.

[99] Archana Sulochana Ganapathi. "Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning". In: *Technical Report No. UCB/EECS-2009-181*. 2009.

[100] Minos N. Garofalakis and Phillip B. Gibbon. "Approximate Query Processing: Taming the TeraBytes". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 2001.

[101] Johannes Gehrke, Edward Lui, and Rafael Pass. "Towards Privacy for Social Networks: A Zero-Knowledge Based Definition of Privacy". In: *Theory of Cryptography*. 2011.

[102] Johannes Gehrke, Michael Hay, Edward Lui, and Rafael Pass. "Crowd-blending privacy". In: *Proceedings of the 32th Annual International Conference on Advances in Cryptology* (CRYPTO). 2012.

[103] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. "ApproxHadoop: Bringing Approximations to MapReduce Frameworks". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS). 2015.

[104] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *Proceedings of the annual ACM symposium on Theory of computing* (STOC). 1987.

[105] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2014.

[106] Michael T. Goodrich and Michael Mitzenmacher. "Invertible Bloom Lookup Tables". In: *CoRR*. 2011.

[107] S. Dov Gordon, Tal Malkin, Mike Rosulek, and Hoeteck Wee. "Multi-party Computation of Polynomials and Branching Programs without Simultaneous Interaction". In: *Proceedings of the Annual International Conference on Advances in Cryptology* (EUROCRYPT). 2013.

[108] Saikat Guha, Bin Cheng, and Paul Francis. "Privad: Practical Privacy in Online Advertising". In: *Proceedings of the 8th Symposium on Networked Systems Design and Implementation* (NSDI). 2011.

[109] Saikat Guha, Mudit Jain, and Venkata N. Padmanabhan. "Koi: A Location-Privacy Platform for Smartphone Apps". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI). 2012.

[110] Leonidas J. Guibas. "Kinetic data structures: a state of the art report". In: *Proceedings of the third Workshop on the Algorithmic Foundations of Robotics* (WAFR). 1998.

[111] Vincenzo Gulisano, Valentin Tudor, Magnus Almgren, and Marina Papatriantafilou. "BES: Differentially Private and Distributed Event Aggregation in Advanced Metering Infrastructures". In: *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security* (CPSS). 2016.

[112] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2010.

[113] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. "Scalable and Private Media Consumption with Popcorn". In: *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation* (NSDI). 2016.

[114] Peter J Haas. "Data-stream sampling: basic techniques and results". In: *Data Stream Management*. 2016.

[115] Peter J. Haas and Joseph M. Hellerstein. "Ripple Joins for Online Aggregation". In: *Proceedings of the ACM International Conference on Management of Data* (SIGMOD). 1999.

[116] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. "Differential Privacy Under Fire". In: *Proceedings of the 20th USENIX Security Symposium* (USENIX Security). 2011.

[117] Michaela Hardt and Suman Nath. "Privacy-aware personalization for mobile advertising". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (CCS). 2012.

[118] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. "Boosting the Accuracy of Differentially Private Histograms Through Consistency". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2010.

[119] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. "Comet: Batched Stream Processing for Data Intensive Distributed Computing". In: *Proceedings of the ACM Symposium on Cloud Computing* (SoCC). 2010.

[120] *HealthCare.gov Sends Personal Data to Dozens of Tracking Websites*. https://www.eff.org/deeplinks/2015/01/healthcare.gov-sends-personal-data. Accessed: Nov, 2017. url: https://www.eff.org/deeplinks/2015/01/healthcare.gov-sends-personal-data.

[121] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD). 1997.

[122] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm". In: *Proceedings of the 16th International Conference on Extending Database Technology* (EDBT). 2013.

[123] Daniel G Horvitz and Donovan J Thompson. "A generalization of sampling without replacement from a finite universe". In: *Journal of the American statistical Association*. 1952.

[124] T h. Hubert Chan, Elaine Shi, and Dawn Song. "Privacy-preserving stream aggregation with fault tolerance". In: *Proceedings of 16th International Conference on Financial Cryptography and Data Security* (FC). 2012.

[125] Xenofontas Dimitropoulos Marc Stoecklin Paul Hurley and Kind Andreas. "The eternal sunshine of the sketch data structure". In: *Comput. Netw.* 2008.

[126] IDC. *EMC Digital Universe with Research & Analysis*. http://www.emc.com/leadership/digital-universe/2014iview/index.htm. 2004.

[127] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the ACM European Conference on Computer Systems* (EuroSys). 2007.

[128] Stanislaw Jarecki and Vitaly Shmatikov. "Efficient Two-Party Secure Computation on Committed Inputs". In: *Proceedings of the Annual International Conference on Advances in Cryptology* (EUROCRYPT). 2007.

[129] Zbigniew Jerzak and Holger Ziekow. "The DEBS 2015 Grand Challenge". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems* (DEBS). 2015.

[130] *Kafka - A high-throughput distributed messaging system*. http://kafka.apache.org. Accessed: Nov, 2017.

[131] Niranjan Kamat and Arnab Nandi. "A Unified Correlation-based Approach to Sampling Over Joins". In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (SSDBM). 2017.

[132] Niranjan Kamat and Arnab Nandi. "Perfect and Maximum Randomness in Stratified Sampling over Joins". In: *CoRR*. 2016.

[133] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[134] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. "Private analysis of graph structure". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2011.

[135] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. "What Can We Learn Privately?" In: *SIAM J. Comput.* 2011.

[136] Tim Kraska. "Approximate Query Processing for Interactive Data Science". In: *Proceedings of the 2017 ACM International Conference on Management of Data* (SIGMOD). 2017.

[137] Dhanya R. Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. "IncApprox: A Data Analytics System for Incremental Approximate Computing". In: *Proceedings of the 25th International Conference on World Wide Web* (WWW). 2016.

[138] Doug Laney. "3D data management: Controlling data volume, velocity and variety". In: *META Group Research Note*. 2001.

[139] Kevin Lang, Edo Liberty, and Konstantin Shmakov. "Stratified Sampling Meets Machine Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning* (ICML). 2016.

[140] Jaewoo Lee and Chris Clifton. "How Much is Enough? Choosing ϵ for Differential Privacy". In: *Proceedings of the 14th International Conference on Information Security* (ISC). 2011.

[141] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. "ΠBox: A Platform for Privacy-Preserving Apps". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation* (NSDI). 2013.

[142] Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. "Join Processing Using Bloom Filter in MapReduce". In: *Proceedings of the 2012 ACM Research in Applied Computation Symposium* (RACS). 2012.

[143] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. "A Cost Semantics for Self-Adjusting Computation". In: *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2009.

[144] Bingdong Li, Jeff Springer, George Bebis, and Mehmet Hadi Gunes. "Review: A Survey of Network Flow Applications". In: *J. Netw. Comput. Appl.* 2013.

[145] Chao Li, Michael Hay, Vibhor Rastogi, Gerome Miklau, and Andrew McGregor. "Optimizing Linear Counting Queries Under Differential Privacy". In: *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 2010.

[146] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. "Wander Join: Online Aggregation via Random Walks". In: *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 2016.

[147] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2014.

[148] Yehuda Lindell and Benny Pinkas. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *Proceedings of the Annual International Conference on Advances in Cryptology (EUROCRYPT)*. 2007.

[149] Yehuda Lindell and Benny Pinkas. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *J. Cryptology*. 2015.

[150] Shiyao Liu and William Q Meeker. "Statistical Methods for Estimating the Minimum Thickness Along a Pipeline". In: *Technometrics*. 2014.

[151] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin Web, and Ken Yocum. "Stateful bulk processing for incremental analytics". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2010.

[152] Sharon Lohr. *Sampling: Design and Analysis, 2nd Edition*. Cengage Learning, 2009.

[153] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 2012.

[154] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2010.

[155] Sayanta Mallick, Gaétan Hains, and Cheikh Sadibou Deme. "A resource prediction model for virtualization servers". In: *Proceedings of International Conference on High Performance Computing and Simulation (HPCS)*. 2012.

[156] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. "Finding (recently) frequent items in distributed data streams". In: *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. 2005.

[157] Mohammad M Masud, Clay Woolam, Jing Gao, Latifur Khan, Jiawei Han, Kevin W Hamlen, and Nikunj C Oza. "Facing the reality of data stream classification: coping with scarcity of labeled data". In: *Knowledge and information systems*. Springer, 2012.

[158] Commons Math. *The Apache Commons Mathematics Library*. http://commons.apache.org/proper/commons-math. Accessed: Nov, 2017.

[159] Frank McSherry. "Privacy Integrated Queries". In: *Proceedings of* ACM SIGMOD *International Conference on Management of Data* (SIGMOD). 2009.

[160] Frank McSherry and Ratul Mahajan. "Differentially-private Network Trace Analysis". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (SIGCOMM). 2010.

[161] Xiangrui Meng. "Scalable Simple Random Sampling and Stratified Sampling". In: *Proceedings of the 30th International Conference on Machine Learning* (ICML). 2013.

[162] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. "Probabilistically Accurate Program Transformations". In: *Proceedings of the 18th International Conference on Static Analysis* (SAS). 2011.

[163] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. "Measurement and Analysis of Online Social Networks". In: *Proceedings of the ACM SIGCOMM Conference on Internet Measurement* (IMC). 2007.

[164] Sparsh Mittal. "A survey of techniques for approximate computing". In: ACM *Computing Surveys* (CSUR). 2016.

[165] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. "GUPT: Privacy Preserving Data Analysis Made Easy". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (SIGMOD). 2012.

[166] David S. Moore. *The Basic Practice of Statistics*. 2nd. W. H. Freeman & Co., 1999.

[167] Barzan Mozafari. "Approximate Query Engines: Commercial Challenges and Research Opportunities". In: *Proceedings of the 2017 ACM International Conference on Management of Data* (SIGMOD). 2017.

[168] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (SOSP). 2013.

[169] Arjun Narayan and Andreas Haeberlen. "DJoin: Differentially Private Join Queries over Distributed Databases". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2012.

[170] Swaminathan Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[171] Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. "Online aggregation for large mapreduce jobs". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2011.

[172] Sérgio Almeida Paulo, Baquero Carlos, Preguiça Nuno, and Hutchison David. "Scalable Bloom Filters". In: *Information Processing Letters*. 2007.

[173] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (OSDI). 2010.

[174] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation Is Practical". In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology* (ASIACRYPT). 2009.

[175] Odile Pons. "Bootstrap of means under stratified sampling". In: *Electronic Journal of Statistics*. Institute of Mathematical Statistics, 2007.

[176] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. "DryadInc: Reusing work in large-scale computations". In: *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*. 2009.

[177] Navneet Potti and Jignesh M. Patel. "DAQ: A New Paradigm for Approximate Query Processing". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2015.

[178] *Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies*. http://www.wired.com/2010/07/zombie-cookies-lawsuit. Accessed: Nov, 2017. url: http://www.wired.com/2010/07/zombie-cookies-lawsuit.

[179] Davide Proserpio, Sharon Goldberg, and Frank McSherry. "Calibrating Data to Sensitivity in Private Data Analysis: A Platform for Differentially-private Analysis of Weighted Datasets". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2014.

[180] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. "TimeStream: Reliable Stream Computation in the Cloud". In: *Proceedings of the ACM European Conference on Computer Systems* (EuroSys). 2013.

[181] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetze, Volker Hilt, and Thorsten Strufe. "Approximate Stream Analytics in Apache Flink and Apache Spark Streaming". In: *CoRR*. Vol. abs/1709.02946. 2017.

[182] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. *Privacy Preserving Stream Analytics: The Marriage of Randomized Response and Approximate Computing*. https://arxiv.org/abs/1701.05403. Technical Report. 2017. url: https://arxiv.org/abs/1701.05403.

[183] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. "PrivApprox: Privacy-Preserving Stream Analytics". In: *Proceedings of the 2017 USENIX Annual Technical Conference* (USENIX ATC). 2017.

[184] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. "StreamApprox: Approximate Computing for Stream Analytics". In: *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware)*. 2017.

[185] Do Le Quoc, Christof Fetzer, Pascal Felber, Étienne Rivière, Valerio Schiavoni, and Pierre Sutra. "UniCrawl: A Practical Geographically Distributed Web Crawler". In: *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*. 2015.

[186] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. "SnappyData: A Hybrid Transactional Analytical Store Built On Spark". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2016.

[187] Vibhor Rastogi and Suman Nath. "Differentially private aggregation of distributed time-series with transformation and encryption". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2010.

[188] Michael G Reed, Paul F Syverson, and David M Goldschlag. "Anonymous connections and onion routing". In: *IEEE Journal on Selected Areas in Communications*. 1998.

[189] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. "Airavat: Security and Privacy for MapReduce". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2010.

[190] *Sample household electricity time of use data*. https://goo.gl/0p2QGB. Accessed: Jan, 2017.

[191] Adrian Sampson. "Hardware and Software for Approximate Computing". PhD thesis. University of Washington, 2015.

[192] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. "Accept: A programmer-guided compiler framework for practical approximate computing". In: *University of Washington Technical Report UW-CSE-15-01*. 2015.

[193] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. "Approximate Storage in Solid-state Memories". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2013.

[194] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2011.

[195] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services ". In: *Proceedings of the USENIX Security Symposium (USENIX Security)*. 2012.

[196] *SEC Charges Two Employees of a Credit Card Company with Insider Trading*. http://www.sec.gov/litigation/litreleases/2015/lr23179.htm. Accessed: Nov, 2017. url: http://www.sec.gov/litigation/litreleases/2015/lr23179.htm.

[197] Zechao Shang and Jeffrey Xu Yu. "Auto-approximation of Graph Computing". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2014.

[198] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. "Privacy-Preserving Aggregation of Time-Series Data". In: *Proceedings of the Symposium on Network and Distributed System Security* (NDSS). 2011.

[199] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. "Managing Performance vs. Accuracy Trade-offs with Loop Perforation". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (ESEC/FSE). 2011.

[200] Lefteris Sidirourgos, Martin L Kersten, Peter A Boncz, et al. "SciBORQ: Scientific data management with Bounds On Runtime and Quality." In: *Biennial Conference on Innovative Data Systems Research* (CIDR). 2011.

[201] Kapil Singh, Sumeer Bhola, and Wenke Lee. "xBook: Redesigning Privacy Control in Social Networking Platforms". In: *Proceedings of the 18th Conference on USENIX Security Symposium* (USENIX Security). Montreal, Canada, 2009.

[202] *Spark Structured Streaming*. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. Accessed: Nov, 2017.

[203] *SQLite*. https://www.sqlite.org/. Accessed: Nov, 2017.

[204] Kandula Srikanth, Shanbhag Anil, Vitorovic Aleksandar, Olma Matthaios, Grandl Robert, Chaudhuri Surajit, and Bolin Ding. "Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD). 2016.

[205] Steven K. Thompson. *Sampling*. Wiley Series in Probability and Statistics, 2012, pp. 141–144,173–182.

[206] Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. "Building a Hybrid Warehouse: Efficient Joins Between Data Stored in HDFS and Enterprise Warehouse". In: *ACM Trans. Database Syst.* 2016.

[207] Yuanyuan Tian, Tao Zou, Fatma Ozcan, Romulo Goncalves, and Hamid Pirahesh. "Joins for Hybrid Warehouses: Exploiting Massive Parallelism in Hadoop and Enterprise Data Warehouses." In: *In Proceedings of the 2015 International Conference on Extending Database Technology* (EDBT). 2015, pp. 373–384.

[208] *Trident*. https://github.com/nathanmarz/storm/wiki/Trident-tutorial. Accessed: Nov, 2017.

[209] *Twitter Search API*. http://apiwiki.twitter.com/Twitter-API-Documentation. Accessed: Nov, 2017.

[210] Akidau Tyler and et. al. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 2013.

[211] Erlingsson Úlfar, Pihur Vasyl, and Korolova Aleksandra. "RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[212] Bimal Viswanath, Emre Kiciman, and Stefan Saroiu. "Keeping Information Safe from Social Networking Apps". In: *Proceedings of the ACM SIGCOMM Workshop on Social Networks (WOSN'12)*. 2012.

[213] Jeffrey S Vitter. "Random sampling with a reservoir". In: *ACM Transactions on Mathematical Software (TOMS)*. 1985.

[214] Gang Wang, Bolun Wang, Tianyi Wang, Ana Nika, Haitao Zheng, and Ben Y. Zhao. "Defending Against Sybil Devices in Crowdsourced Mapping Services". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2016.

[215] Qian Wang, Yan Zhang, Xiao Lu, Zhibo Wang, Zhan Qin, and Kui Ren. "RescueDP: Real-time Spatio-temporal Crowd-sourced Data Publishing with Differential Privacy". In: *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*. 2016.

[216] S. L. Warner. "Randomized response: A survey technique for eliminating evasive answer bias". In: *Journal of the American Statistical Association*. 1965.

[217] Lucas Waye. "Privacy Integrated Data Stream Queries". In: *Proceedings of the 2014 International Workshop on Privacy & Security in Programming (PSP)*. 2014.

[218] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. "Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud". In: *proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*. 2010.

[219] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. "Conductor: Orchestrating the Clouds". In: *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*. 2010.

[220] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. "Orchestrating the Deployment of Computations in the Cloud with Conductor". In: *proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[221] David P. Woodruff. "Revisiting the Efficiency of Malicious Two-Party Computation". In: *Proceedings of the 26th Annual International Conference on Advances in Cryptology (EUROCRYPT)*. 2007.

[222] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2007.

[223] Andrew Chi-Chih Yao. "Protocols for Secure Computations". In: *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SFCS)*. 1982.

[224] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing". In: *IEEE Design Test*. 2017.

[225] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. "Axilog: Language support for approximate hardware design". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015.

[226] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. "CS2: A New Database Synopsis for Query Estimation". In: *Proceedings of the 2013 International Conference on Management of Data (SIGMOD)*. 2013.

[227] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-Tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[228] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2008.

[229] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2012.

[230] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. "G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2015.

[231] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. "iMapReduce: A Distributed Computing Framework for Iterative Computation". In: *Proceedings of the International Workshop on Data Intensive Computing in the Clouds (DataCloud)*. 2011.