THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

StreamApprox: Approximate Computing for Stream Analytics

Citation for published version:
Quoc, DL, Chen, R, Bhatotia, P, Fetzer, C, Hilt, V & Strufe, T 2017, StreamApprox: Approximate Computing for Stream Analytics. in *ACM/IFIP/USENIX Middleware 2017.* ACM, pp. 185-197, 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, Nevada, United States, 11/12/17.
https://doi.org/10.1145/3135974.3135989

Digital Object Identifier (DOI):
10.1145/3135974.3135989

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM/IFIP/USENIX Middleware 2017

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. Aug. 2022

StreamApprox: Approximate Computing for Stream Analytics

Do Le Quoc[1], Ruichuan Chen[2], Pramod Bhatotia[3],
Christof Fetzer[1], Volker Hilt[2], Thorsten Strufe[1]
[1]TU Dresden, [2]Nokia Bell Labs, [3]University of Edinburgh and Alan Turing Institute

Abstract

Approximate computing aims for efficient execution of workflows where an approximate output is sufficient instead of the exact output. The idea behind approximate computing is to compute over a representative sample instead of the entire input dataset. Thus, approximate computing — based on the chosen sample size — can make a systematic trade-off between the output accuracy and computation efficiency.

Unfortunately, the state-of-the-art systems for approximate computing primarily target batch analytics, where the input data remains unchanged during the course of computation. Thus, they are not well-suited for stream analytics. This motivated the design of StreamApprox— a stream analytics system for approximate computing. To realize this idea, we designed an online stratified reservoir sampling algorithm to produce approximate output with rigorous error bounds. Importantly, our proposed algorithm is generic and can be applied to two prominent types of stream processing systems: (1) batched stream processing such as Apache Spark Streaming, and (2) 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 microbenchmarks and real-world case studies. Our results show that Spark- and Flink-based StreamApprox systems achieve a speedup of 1.15×—3× compared to the respective native Spark Streaming and Flink executions, with varying sampling fraction of 80% to 10%. Furthermore, we have also implemented an improved baseline in addition to the native execution baseline — a Spark-based approximate computing system leveraging the existing sampling modules in Apache Spark. Compared to the improved baseline, our results show that StreamApprox achieves a speedup of 1.1×—2.4× while maintaining the same accuracy level.

1 Introduction

Stream analytics systems are extensively used in the context of modern online services to transform continuously arriving raw data streams into useful insights [20, 34, 47]. 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 [17], 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 [18, 35]. 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 [1, 26, 39], where the input data remains unchanged during the course of computation (see §8 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 (i.e., 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 [21]) 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 §4 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) [46], 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.

To address these limitations, we designed an *online stratified reservoir sampling algorithm* for stream analytics. Unlike existing Spark-based systems, we perform the sampling process "on-the-fly"

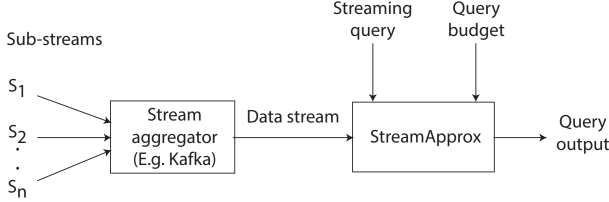

Figure 1. System overview

to reduce the latency as well as the overheads associated in 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 [22], and (2) pipelined stream processing employed by Apache Flink [20].

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 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).

Based on the proposed sampling algorithm, we designed StreamApprox, an approximate computing system for stream analytics (see Figure 1). StreamApprox provides an interface for users to specify streaming queries and their execution budgets. The query execution budget can be specified in the form of latency guarantees or available computing resources. Based on the query budget, StreamApprox provides an adaptive execution mechanism to make a systematic trade-off between the output accuracy and computation efficiency. In particular, StreamApprox employs the proposed sampling algorithm to select a sample size based on the query budget, and executes the streaming query on the selected sample. Finally, StreamApprox provides a confidence metric on the output accuracy via rigorous error bounds. The error bound gives a measure of accuracy trade-off on the result due to the approximation.

We implemented StreamApprox based on Apache Spark Streaming [22] and Apache Flink [20], and evaluate its effectiveness via various microbenchmarks. 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× to 3× 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×—2.4× 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 microbenchmarks and real-world case studies.

StreamApprox's codebase with the full experimental evaluation setup is publicly available: https://streamapprox.github.io/. A detailed version of this paper is available as a technical report [37].

2 Overview and Background

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.

2.1 System Overview

StreamApprox is designed for real-time stream analytics. Figure 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 [24]) 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 uses 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.

2.2 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. *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 [22] adopted this model to process input data streams.

Pipelined stream processing model. In contrast to the batched stream processing model, the pipelined model streams each data

Algorithm 1 **Reservoir sampling algorithm**

```
Input: N ← sample size
begin
    reservoir ← ∅; // Set of items sampled from the input stream
    foreach arriving item x_i do
        if |reservoir| < N then
            // Fill up the reservoir
            reservoir.append(x_i);
        end
        else
            p ← N/i;
            // Flip a coin comes heads with probability p
            head ← flipCoin(p);
            if head then
                // Get a random index in the reservoir
                j ← getRandomIndex(0, |reservoir| − 1);
                // Replace old item in reservoir by x_i
                reservoir[j] ← x_i
            end
        end
    end
end
```

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 [20] implements this model to provide a truly native stream processing engine.

Note that both stream processing models support the time-based sliding window computation [6]. 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.

2.3 Design Assumptions

StreamApprox is based on the following assumptions. We discuss the possible means to address these assumptions in §7.

1. We assume there exists a virtual cost function which translates a given query budget (such as the expected latency 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.

2.4 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 [30]. 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 [41].

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 an 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 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* [2] 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. 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 Design

In this section, we first present the StreamApprox's workflow (§3.1). Then, we detail its sampling mechanism (§3.2), and its error estimation mechanism (§3.3).

3.1 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 §2.2).

For each time interval, we first derive the sample size (*sampleSize*) using a cost function based on the given query budget (see §7). As described in §2.3, we currently assume that there exists a cost function which translates a given query budget (such as the expected latency/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 §7.

We next propose a sampling algorithm (detailed in §3.2) 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 §3.3) to compute the error bounds for the approximate query result in the form of *output* ± *error* bound.

The whole process repeats for each time interval as the computation window slides [7]. Note that, the query budget can change

Algorithm 2 : StreamApprox's algorithm overview

```
User input: streaming query and query budget
begin
    // Computation in sliding window model (§2.2)
    foreach time interval do
        // Cost function gives the sample size based on the budget (§7)
        sampleSize ← costFunction(budget);
        forall arriving items in the time interval do
            // Perform OASRS Sampling (§3.2)
            // W denotes the weights of the sample
            sample, W ← OASRS(items, sampleSize);
        end
        // Run query as a data-parallel job to process the sample
        output ← runJob(query, sample, W);
        // Estimate the error bounds of query result/output (§3.3)
        output ± error ← estimateError(output);
    end
end
```

across time intervals to adapt to user's requirements for the budget.

3.2 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 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 can be 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 §7). 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 2).

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:

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

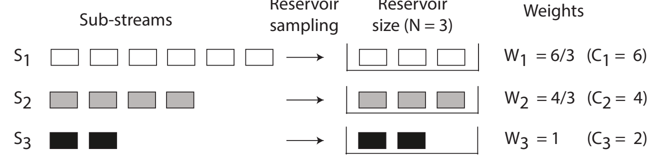

Figure 2. OASRS with the reservoirs of size three.

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 1. Then, the approximate sum SUM_i of all items received from each sub-stream S_i can be estimated as:

$$SUM_i = (\sum_{j=1}^{Y_i} I_{i,j}) \times W_i \tag{2}$$

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

$$SUM = \sum_{i=1}^{X} SUM_i \tag{3}$$

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

$$MEAN = \frac{SUM}{\sum_{i=1}^{X} C_i} \tag{4}$$

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 [11, 12]. It is also worth mentioning that, OASRS not only works for a concerned time interval (e.g., a sliding time window), but also works with unbounded data streams.

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 its beginning [3].

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.

Algorithm 3 : Online adaptive stratified reservoir sampling

```
OASRS(items, sampleSize)
begin
    sample ← ∅; // Set of items sampled within the time interval
    S ← ∅; // Set of sub-streams seen so far within the time interval
    W ← ∅; // Set of weights of sub-streams within the time interval
    Update(S); // Update the set of sub-streams
    // Determine the sample size for each sub-stream
    N ← getSampleSize(sampleSize, S);
    forall S_i in S do
        C_i ← 0; // Initial counter to measure #items in each sub-stream
        forall arriving items in each time interval do
            Update(C_i); // Update the counter
            sample_i ← RS(items, N_i); // Reservoir sampling
            sample.add(sample_i); // Update the global sample
            // Compute the weight of sample_i according to Equation 1
            if C_i > N_i then
                W_i ← C_i / N_i;
            end
            else
                W_i ← 1;
            end
            W.add(W_i); // Update the set of weights
        end
    end
    return sample, W
end
```

3.3 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 §3.2, 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:

$$Var(SUM) = \sum_{i=1}^{X} Var(SUM_i) \tag{5}$$

Further, as items are randomly selected for a sample within each sub-stream, according to the random sampling theory [40], the variance of the approximate sum can be estimated as:

$$\widehat{Var}(SUM) = \sum_{i=1}^{X} \left(C_i \times (C_i - Y_i) \times \frac{s_i^2}{Y_i} \right) \tag{6}$$

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:

$$s_i^2 = \frac{1}{Y_i - 1} \times \sum_{j=1}^{Y_i} (I_{i,j} - \bar{I}_i)^2, \text{ where } \bar{I}_i = \frac{1}{Y_i} \times \sum_{j=1}^{Y_i} I_{i,j} \tag{7}$$

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 4, this approximate mean value can be computed as:

$$\begin{aligned} MEAN &= \frac{SUM}{\sum_{i=1}^{X} C_i} = \frac{\sum_{i=1}^{X} (C_i \times MEAN_i)}{\sum_{i=1}^{X} C_i} \\ &= \sum_{i=1}^{X} (\omega_i \times MEAN_i) \end{aligned} \tag{8}$$

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 [40], the variance of the approximate mean value can be estimated as:

$$\begin{aligned} \widehat{Var}(MEAN) &= \sum_{i=1}^{X} Var(\omega_i \times MEAN_i) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times Var(MEAN_i) \right) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times \frac{s_i^2}{Y_i} \times \frac{C_i - Y_i}{C_i} \right) \end{aligned} \tag{9}$$

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 estimate the variance of the approximate results of any linear queries.

Error bound. According to the "68-95-99.7" rule [45], 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.

4 Implementation

To showcase the effectiveness of our algorithm, we implemented StreamApprox based on two stream processing systems (§2.2): *(i)* Apache Spark Streaming [22], and *(ii)* Apache Flink [20].

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 [21]. 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 evaluated 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 only the Flink native execution.

We next present the necessary background on Spark Streaming (and its existing sampling mechanisms) and Flink (§4.1). Thereafter, we provide the implementation details of our prototypes (§4.2).

4.1 Background

4.1.1 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. Spark Streaming makes use of RDD-based sampling functions supported by Apache

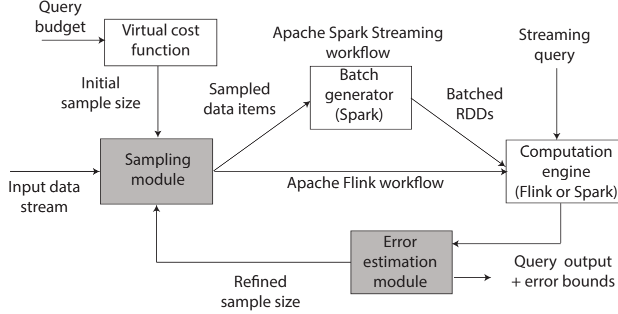

Figure 3. Architecture of StreamApprox prototypes (shaded boxes depict the implemented modules). We have implemented our system based on Apache Spark Streaming and Apache Flink.

Spark [46] 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 [33] 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.

4.1.2 Apache Flink

In contrast to the 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 §3.2).

4.2 StreamApprox Implementation Details

We next describe the implementation of StreamApprox. Figure 3 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.

4.2.1 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/throughput, available computational resources, or acceptable accuracy loss. As noted in the design assumptions (§2.3), we have not implemented the virtual cost function since it is beyond the scope of this paper (see §7 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. In 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 §3.2 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 [32] to implement the error estimation mechanism as described in §3.3.

4.2.2 Flink-based StreamApprox

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

I: Sampling module. We created a sampling operator by implementing the algorithm described in §3.2. 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.

5 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 (§6).

5.1 Experimental Setup

Synthetic data stream. To understand the effectiveness of our proposed OASRS sampling algorithm, we first evaluated StreamApprox using a synthetic input data stream with Gaussian distribution

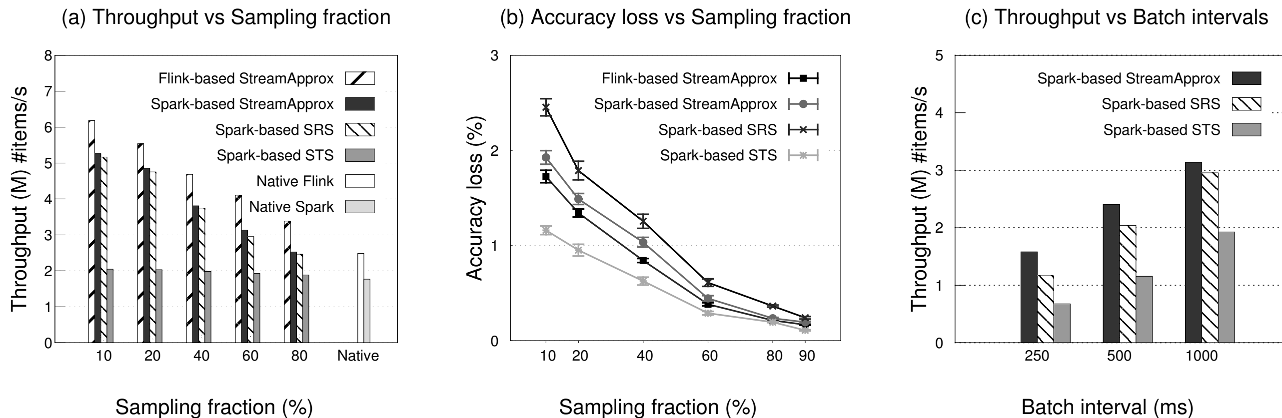

Figure 4. Comparison b/w StreamApprox, Spark-based SRS, Spark-based STS, as well as the native Spark and Flink systems. (a) Throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions. (c) Throughput with different batch intervals.

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 with 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 with 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 §4). 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 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 §5.2, (b) varying batch intervals in §5.3, (c) varying arrival rates for sub-streams in §5.4, (d) varying window sizes in §5.5, (e) scalability in §5.6, and (f) skew in the input data stream in §5.7.

5.2 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 4 (a) first shows the throughput comparison of StreamApprox and the two sampling algorithms in Spark. Spark-based stratified sampling (STS) scales poorly because of its synchronization among Spark workers and the expensive sorting during its sampling process (as detailed in §4.1). Spark-based StreamApprox achieves a throughput of 1.68× and 2.60× higher than Spark-based STS with sampling fractions of 60% and 10%, respectively. In addition, Spark-based simple random sampling (SRS) scales better than STS and has a similar throughput as in StreamApprox, but SRS loses the capability of considering each sub-stream fairly.

Meanwhile, Flink-based StreamApprox achieves a throughput of 2.13× and 3× 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× 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× higher than the native Spark execution, whereas the throughput of Flink-based StreamApprox is 1.65× higher than the native Flink.

Accuracy. Next, we compare the accuracy of our proposed OASRS sampling with that of the two sampling mechanisms with the varying sampling fractions. Figure 4 (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 that 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 §4.1). This leads to a much lower throughput than StreamApprox which only maintains a sample of a fixed size for each sub-stream.

5.3 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%

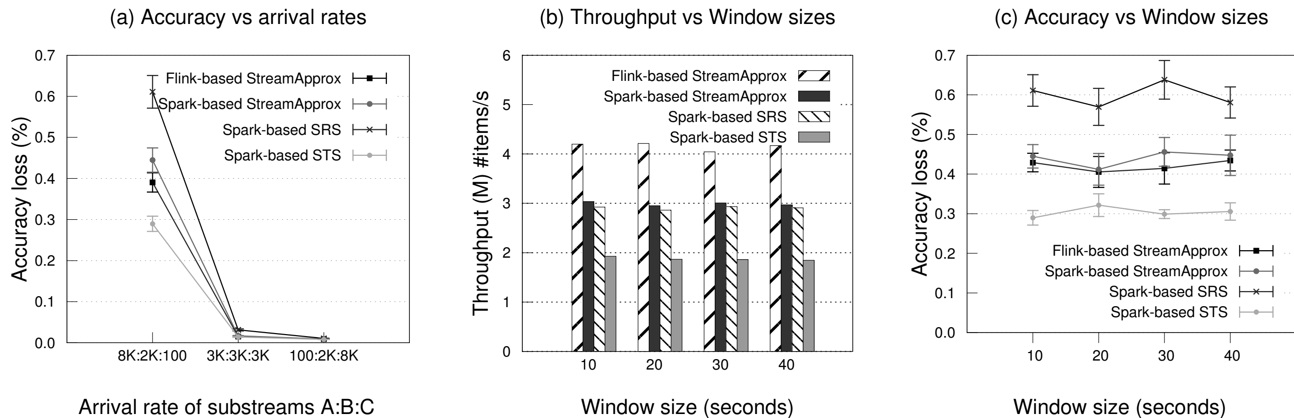

Figure 5. Comparison between StreamApprox, Spark-based SRS, and Spark-based STS. (a) Accuracy loss with varying arrival rates. (b) Throughput with varying window sizes. (c) Accuracy loss with varying window sizes.

and measure the throughput of each system with different batch intervals.

Figure 4 (c) 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× and 1.63× higher than the throughput of Spark-based SRS and STS, respectively; with the 250ms batch interval, the throughput of StreamApprox is 1.36× and 2.33× 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.

5.4 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 §5.1. 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 5 (a) 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 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.

5.5 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 §5.4 with its three sub-streams' arrival rates being 8000, 2000, and 100 items per second. Figure 5 (b) and Figure 5 (c) 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.

5.6 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 6 (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.

5.7 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 stream using 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.

Figure 7 (a), (b), and (c) present the mean values of the received data items produced by the three Spark-based systems every 5 seconds during a 10-minute observation. As expected, Spark-based STS and StreamApprox provide more accurate results than Spark-based SRS because Spark-based STS and StreamApprox ensure

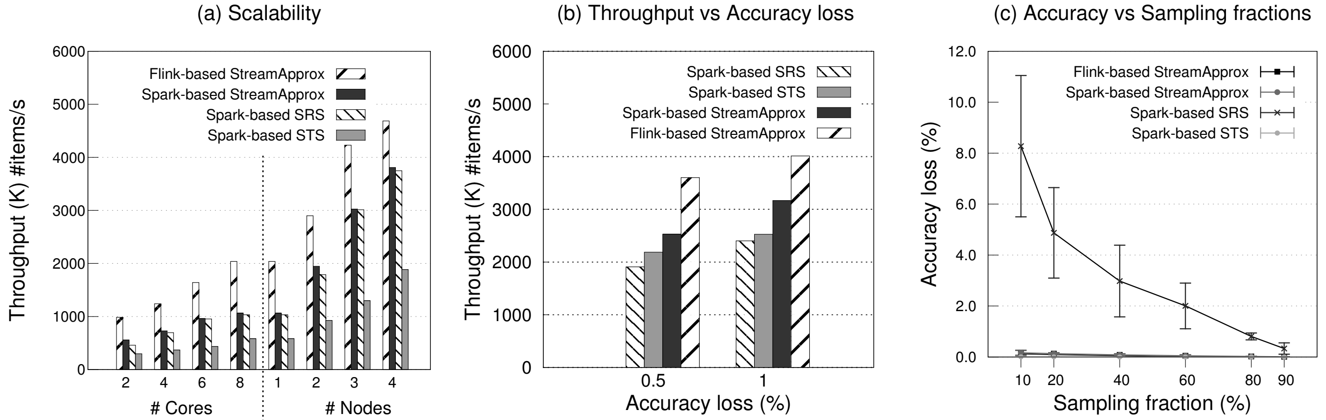

Figure 6. Comparison between StreamApprox, Spark-based SRS, and Spark-based STS. (a) Throughput with different numbers of CPU cores and nodes. (b) Throughput with accuracy loss. (c) Accuracy loss with varying sampling fractions.

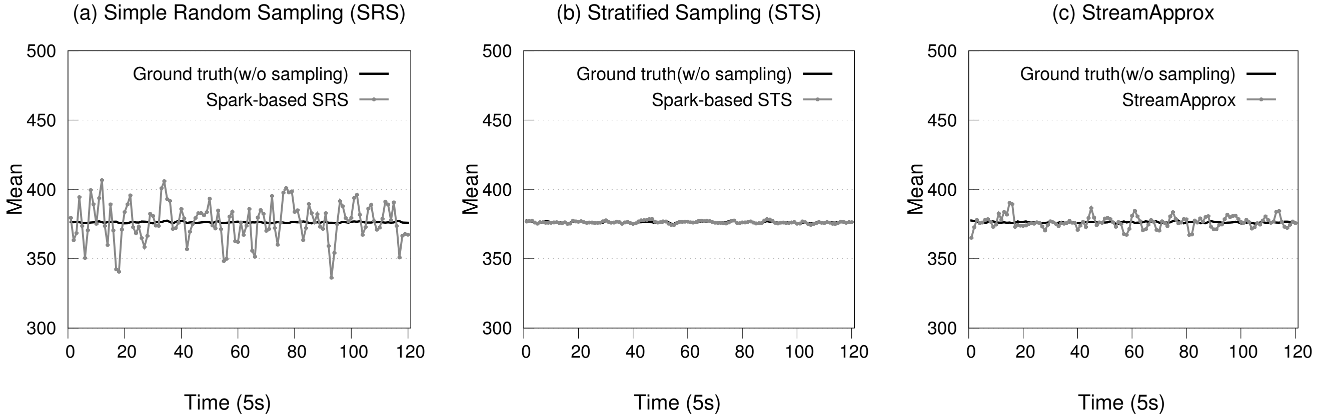

Figure 7. The mean values of the received data items produced by different sampling techniques every 5 seconds during a 10-minute observation. The sliding window size is 10 seconds, and each sliding step is 5 seconds.

that the data items from the minority (i.e., sub-stream C) are fairly selected in the samples.

In addition, we keep the accuracy loss across all four systems the same and then measure their respective throughputs. Figure 6 (b) shows that, with the same accuracy loss of 1%, the throughput of Spark-based STS is 1.05× higher than Spark-based SRS, whereas Spark-based StreamApprox achieves a throughput 1.25× higher than Spark-based STS. In addition, Flink-based StreamApprox achieves the highest throughput which is 1.68×, 1.6×, and 1.26× 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 §5.1. 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 6 (c) 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 our proposed sampling algorithm OASRS over simple random sampling in processing long-tail data which is very common in practice.

6 Case Studies

In this section, we report our experiences and results with the following two real-world case studies: (a) network traffic analytics (§6.2) and (b) New York taxi ride analytics (§6.3).

6.1 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 [23] on the remaining 3 nodes.

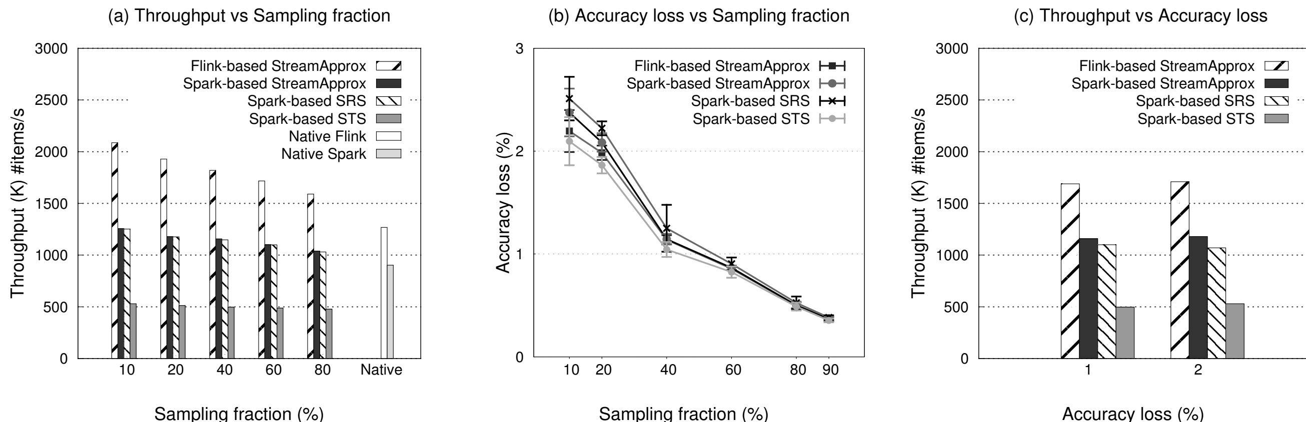

Figure 8. Network traffic analytics case-study: (a) Throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions. (c) Throughput with different accuracy losses.

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.

6.2 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 [13]. These were recorded on the high-speed Internet backbone links in Chicago in 2015. We converted the raw network traces into the NetFlow format [15], and then removed unused fields (such as source and destination ports, duration, etc.) in each NetFlow record to build a dataset for our experiments. The dataset contains 115, 472, 322 TCP flows, 67, 098, 852 UDP flows, and 2, 801, 002 ICMP flows. Each stream data item is a flow record in the dataset.

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

Results. Figure 8 (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× throughput than Spark-based STS, and achieves a similar throughput compared with Spark-based SRS (which loses 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× 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× higher than the native Spark execution, whereas the throughput of Flink-based StreamApprox is 1.35× higher than the native Flink execution. Surprisingly, the throughput of the native Spark execution is even higher than the throughput of Spark-based STS. This is because Spark-based STS requires the expensive extra steps (see §4.1).

Figure 8 (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 performing the expensive *groupByKey* operation as well as synchronization among workers to take samples from the input data stream, whereas StreamApprox performs the sampling operation with a 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 8 (c) shows that, with the accuracy loss of 1%, the throughput of Spark-based StreamApprox is 2.36× higher than Spark-based STS, and 1.05× higher than Spark-based SRS. Flink-based StreamApprox achieves a throughput even 1.46× higher than Spark-based StreamApprox.

Finally, to make a comparison in terms of latency between these systems, we created sampling function *sampleOASRS()* for Spark RDD by implementing our proposed sampling algorithm OASRS,

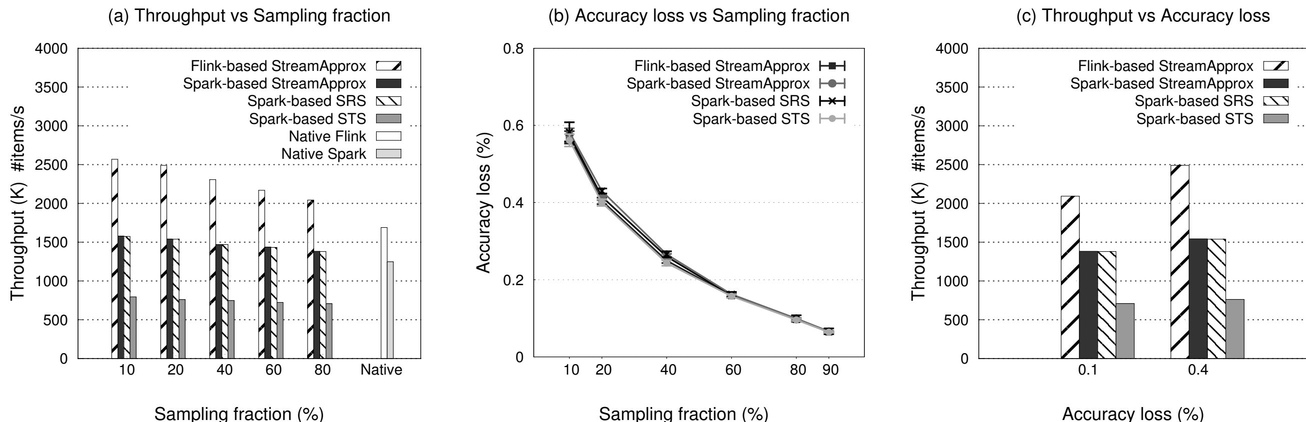

Figure 9. New York taxi ride analytics case-study: (a) Throughput with varying sampling fractions. (b) Accuracy loss with varying sampling fractions. (c) Throughput with different accuracy losses.

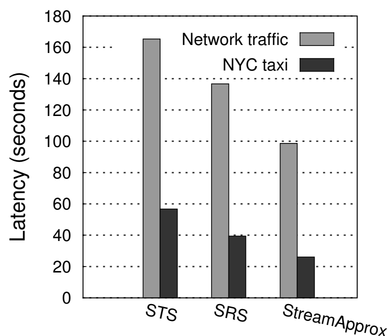

Figure 10. The latency comparison between StreamApprox, Spark-based SRS, and Spark-based STS with the real-world datasets. The sampling fraction is set to 60%.

and then measured the latency in processing the network traffic dataset. Figure 10 indicates that the latency of Spark-based StreamApprox is 1.39× and 1.69× lower than Spark-based SRS and Spark-based STS in processing the network traffic dataset.

6.3 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 [28]. 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 9 (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× higher throughput than Spark-based STS. In addition, due to Flink's pipelined streaming model, Flink-based StreamApprox achieves a 1.5× higher throughput compared to 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× higher than the throughput of the native Spark execution, whereas the throughput of Flink-based StreamApprox is 1.28× 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 the throughput of Spark-based STS.

Figure 9 (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 9 (c) shows that Flink-based StreamApprox achieves the best throughput which is 1.6× higher than Spark-based StreamApprox and Spark-based SRS, and 3× higher than Spark-based STS. Figure 10 further indicates that Spark-based StreamApprox provides the 1.52× and 2.18× lower latency than Spark-based SRS and Spark-based STS.

7 Discussion

The design of StreamApprox is based on the assumptions mentioned in §2.3. 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 as either available computing resources, desired accuracy or latency.

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 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 the 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 [4], and (b) resource prediction model [44] for latency requirements.

Pulsar [4] 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 sample size that can be processed within the given resource budget.

For any given latency requirement, we could employ a resource prediction model [42–44]. In particular, we could build the prediction model by analyzing the diurnal patterns in resource usage [14] 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, we currently assume that the input stream is already stratified based on the source of data items, i.e., the data items within each stratum follow the same distribution — it does not have to be a normal distribution. This assumption ensures that our error estimation mechanism still holds correct since we apply the Central Limit Theorem. 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 follows 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 where we cannot classify strata based on the sources, we need a pre-processing 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 streams: bootstrap [19] and semi-supervised learning [31].

Bootstrap [19] 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 [31] to stratify a data stream. The advantage of this algorithm is that it can work with both labeled and unlabeled streams to train a classification model.

8 Related Work

Given the advantages of making a trade-off between accuracy and efficiency, approximate computing is applied to various domains [38]. Our work mainly builds on the advancements in the databases community.

Over the last two decades, the databases community has proposed various approximation techniques based on sampling [2, 25], online aggregation [27], and sketches [16]. These techniques make different trade-offs w.r.t. the output quality, supported queries, and workload. However, the early work in approximate computing was mainly geared towards the centralized database architecture.

Recently, sampling-based approaches have been successfully adopted for distributed data analytics [1, 26, 29, 36, 39]. In particular, BlinkDB [1] proposes an approximate distributed query processing engine that uses stratified sampling [2] to support ad-hoc queries with error and response time constraints. ApproxHadoop [26] uses multi-stage sampling [30] 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 [39] 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 [29] is a data analytics system that combines two computing paradigms together, namely, approximate and incremental computations [9, 10] for stream analytics. The system is based on an online "biased sampling" algorithm that uses self-adjusting computation [5, 8] to produce incrementally updated approximate output. Lastly, PrivApprox [36] 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.

9 Conclusion

In this paper, 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 selected 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×—3× compared to the native Spark Streaming and Flink executions, respectively. Furthermore, StreamApprox achieves a speedup of 1.1×—2.4× compared to a Spark-based sampling system for approximate computing, while maintaining the same level of accuracy for the query output. Finally, the source code of StreamApprox along with the experimental setup is publicly available: https://streamapprox.github.io/.

Acknowledgments. We thank anonymous reviewers and our shepherd Jan S. Rellermeyer for their helpful comments. This work is supported by the resilience path within CFAED at TU Dresden, the European Unions Horizon 2020 research and innovation programme under grant agreements 645011 (SERECA), Amazon Web Services Education Grant, and Microsoft Azure Grant.

References

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.

[2] Mohammed Al-Kateb and Byung Suk Lee. 2010. Stratified Reservoir Sampling over Heterogeneous Data Streams. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*.

[3] Susanne Albers. 2003. Online algorithms: a survey. *Mathematical Programming* (2003).

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[5] Pramod Bhatotia. 2015. *Incremental Parallel and Distributed Systems*. Ph.D. Dissertation. Max Planck Institute for Software Systems (MPI-SWS).

[6] Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. 2014. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware)*.

[7] Pramod Bhatotia, Marcel Dischinger, Rodrigo Rodrigues, and Umut A. Acar. 2012. *Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis*. Technical Report MPI-SWS-2012-004. MPI-SWS. http://www.mpi-sws.org/tr/2012-004.pdf.

[8] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Bjoern Brandenburg, and Rodrigo Rodrigues. 2015. 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)*.

[9] Pramod Bhatotia, Alexander Wieder, Istemi Ekin Akkus, Rodrigo Rodrigues, and Umut A. Acar. 2011. Large-scale incremental data processing with change propagation. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*.

[10] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. 2011. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[11] Avrim Blum, Cynthia Dwork, Frank McSherry, and Kobbi Nissim. 2005. Practical privacy: the SuLQ framework. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*.

[12] Avrim Blum, Katrina Ligett, and Aaron Roth. 2008. A learning theory approach to non-interactive database privacy. In *Proceedings of the fortieth annual ACM symposium on Theory of computing (STOC)*.

[13] CAIDA. 2017. The CAIDA UCSD Anonymized Internet Traces 2015 (equinix-chicago-dirA). (2017). Retrieved Sep, 2017 from http://www.caida.org/data/passive/passive_2015_dataset.xml

[14] Reiss Charles, Tumanov Alexey, Ganger Gregory, H. Katz Randy, and Kozuch Michael. 2012. *Towards understanding heterogeneous clouds at scale: Google trace analysis*. Techical Report.

[15] Benoit Claise. 2004. Cisco systems NetFlow services export version 9. (2004).

[16] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends databases* (2012).

[17] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[18] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. 2000. On Sequential Monte Carlo Sampling Methods for Bayesian Filtering. *Statistics and Computing* (2000).

[19] Darius M. Dziuda. 2010. *Data mining for genomics and proteomics: analysis of gene and protein expression data*. John Wiley & Sons.

[20] Apache Software Foundation. 2017. Apache Flink. (2017). Retrieved Sep, 2017 from https://flink.apache.org/

[21] Apache Software Foundation. 2017. Apache Spark MLib. (2017). Retrieved Sep, 2017 from http://spark.apache.org/mllib/

[22] Apache Software Foundation. 2017. Apache Spark Streaming. (2017). Retrieved Sep, 2017 from http://spark.apache.org/streaming

[23] Apache Software Foundation. 2017. Apache ZooKeeper. (2017). Retrieved Sep, 2017 from https://zookeeper.apache.org/

[24] Apache Software Foundation. 2017. Kafka - A high-throughput distributed messaging system. (2017). Retrieved Sep, 2017 from http://kafka.apache.org

[25] Minos N. Garofalakis and Phillip B. Gibbon. 2001. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[26] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[28] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 Grand Challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*.

[29] Dhanya R. Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. 2016. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *Proceedings of the 25th International Conference on World Wide Web (WWW)*.

[30] Sharon Lohr. 2009. *Sampling: design and analysis, 2nd Edition*. Cengage Learning.

[31] Mohammad M Masud, Clay Woolam, Jing Gao, Latifur Khan, Jiawei Han, Kevin W Hamlen, and Nikunj C Oza. 2012. Facing the reality of data stream classification: coping with scarcity of labeled data. *Knowledge and information systems* (2012).

[32] Commons Math. 2017. The Apache Commons Mathematics Library. (2017). Retrieved Sep, 2017 from http://commons.apache.org/proper/commons-math

[33] Xiangrui Meng. 2013. Scalable Simple Random Sampling and Stratified Sampling. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*.

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*.

[35] Swaminathan Natarajan. 1995. *Imprecise and Approximate Computation*. Kluwer Academic Publishers.

[36] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2017. PrivApprox: Privacy-Preserving Stream Analytics. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*.

[37] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetze, Volker Hilt, and Thorsten Strufe. 2017. Approximate Stream Analytics in Apache Flink and Apache Spark Streaming. *CoRR* abs/1709.02946 (2017).

[38] Adrian Sampson. 2015. *Hardware and Software for Approximate Computing*. Ph.D. Dissertation. University of Washington.

[39] Kandula Srikanth, Shanbhag Anil, Vitorovic Aleksandar, Olma Matthaios, Grandl Robert, Chaudhuri Surajit, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[40] Steven K. Thompson. 2012. *Sampling*. Wiley Series in Probability and Statistics.

[41] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* (1985).

[42] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2010. 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)*.

[43] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2010. Conductor: Orchestrating the Clouds. In *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*.

[44] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2012. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*.

[45] Wikipedia. 2017. 68-95-99.7 Rule. (2017). Retrieved Sep, 2017 from https://en.wikipedia.org/wiki/68-95-99.7_rule

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*.

[47] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*.