Approximate Computing for Stream Analytics

Do Le Quoc and Ruichuan Chen and Pramod Bhatotia and Christof Fetzer and Volker Hilt and Thorsten Strufe

Abstract Approximate computing has become a promising mechanism to trade off accuracy for efficiency. 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, an online stratified reservoir sampling algorithm is designed to produce approximate output with rigorous error bounds. Importantly, the 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.

Do Le Quoc
TU Dresden, e-mail: do.le_quoc@tu-dresden.de

Ruichuan Chen
Nokia Bell Labs, e-mail: ruichuan.chen@nokia-bell-labs.com

Pramod Bhatotia
University of Edinburgh and Alan Turing Institute, e-mail: pramod.bhatotia@ed.ac.uk

Christof Fetzer
TU Dresden, e-mail: christof.fetzer@tu-dresden.de

Volker Hilt
Nokia Bell Labs, e-mail: volker.hilt@nokia-bell-labs.com

Thorsten Strufe
TU Dresden, e-mail: thorsten.strufe@tu-dresden.de

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 [18, 26, 39]. 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. Since most stream processing systems adopt a data-parallel programming model [15], 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 [16, 27]. 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.

Unfortunately, the advancements in approximate computing are primarily geared towards batch analytics [1, 22, 32], where the input data remains unchanged during the course of computation. 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, one could in principle *repurpose* the available sampling mechanisms in well-known big data processing frameworks such as Apache Spark to build an approximate computing system for stream analytics. In fact, as a starting point for this work, based on the available sampling mechanisms, an approximate computing system is designed and implemented for stream processing in Apache Spark. Unfortunately, Spark's stratified sampling algorithm suffers from three key limitations for approximate computing. 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) [38], 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, this work designed an *online stratified reservoir sampling algorithm* for stream analytics. Unlike existing Spark-based systems, the algorithm performs the sampling process "on-the-fly" to reduce the latency as well as the overheads associated in the process of forming RDDs. Importantly, the algorithm *generalizes* to two prominent types of stream processing models: (1) batched stream processing employed by Apache Spark Streaming [19], and (2) pipelined stream processing employed by Apache Flink [18].

More specifically, the proposed sampling algorithm makes use of two techniques: reservoir sampling and stratified sampling. It performs reservoir sampling for each sub-stream by creating a fixed-size reservoir per stratum. Thereafter, it assigns weights to all strata respecting their 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, STREAMAPPROX—an approximate computing system for stream analytics—is designed.

2 Overview and Background

This section gives an overview of STREAMAPPROX (§2.1), its computational model (§2.2), and its design assumptions (§2.3).

2.1 System Overview

STREAMAPPROX is designed for real-time stream analytics. In this system, the input data stream usually consists of data items arriving from diverse sources. The data items from each source form a *sub-stream*. The system makes use of a stream aggregator (e.g., Apache Kafka [20]) to combine the incoming data items from disjoint sub-streams. STREAMAPPROX then takes this combined stream as the input for data analytics.

STREAMAPPROX 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, the system makes 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. 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. *The 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 pre-defined batch interval, and each such batch is processed via a distributed data-parallel job. Apache Spark Streaming [19] 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 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 [18] implements this model to provide a truly native stream processing engine.

Note that both stream processing models support the time-based sliding window computation [5]. 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. The possible means to address these assumptions are discussed in §4.

1. 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. 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 Design

In this section, first the STREAMAPPROX's workflow (§3.1) is presented. Then, its sampling mechanism (§3.2) and its error estimation mechanism (§3.3) are described (see details in [31, 30]).

3.1 System Workflow

This section shows the workflow of StreamApprox. The system takes the user-specified streaming *query* and the query *budget* as the input. Then it executes the query on the input data stream as a sliding window computation (see §2.2).

For each time interval, StreamApprox first derives the sample size (*sample-Size*) using a cost function based on the given query budget. Next, the system performs a proposed sampling algorithm (detailed in §3.2) to select the appropriate *sample* in an online fashion. This 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, the system executes a data-parallel job to process the user-defined *query* on the selected sample. As the last step, the system performs an error estimation mechanism (as described in §3.3) 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 [6].

3.2 Online Adaptive Stratified Reservoir Sampling

To realize the real-time stream analytics, a novel sampling technique called Online Adaptive Stratified Reservoir Sampling (OASRS) is proposed. 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. The algorithm first stratifies the input stream into sub-streams according to their sources. The data items from each sub-stream are assumed to 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.) The algorithm then samples each sub-stream independently, and perform the reservoir sampling for each sub-stream individually. To do so, every time a new sub-stream S_i is encountered, its sample size N_i is determined according to an adaptive cost function considering the specified query budget. For each sub-stream S_i, the algorithm performs the traditional reservoir sampling to select items at random from this sub-stream, and ensures that the total number of selected items from S_i does not exceed its sample size N_i. In addition, the algorithm maintains a counter C_i to measure the number of items received from S_i within the concerned time interval.

Applying reservoir sampling to each sub-stream S_i ensures that algorithm 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), the algorithm randomly selects N_i items from this sub-stream and each selected item represents C_i/N_i original items on average; otherwise, if $C_i \leq N_i$, the algorithm selects 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, the algorithm assigns a specific weight W_i to the items selected from each sub-stream S_i:

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

StreamApprox supports *approximate linear queries* which return an approximate weighted sum of all items received from all sub-streams. 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.

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.

3.3 Error Estimation

This section describes how to apply OASRS to randomly sample the input data stream to generate the approximate results for linear queries. Next, a method to estimate the accuracy of approximate results via rigorous error bounds is presented.

Similar to §3.2, suppose the input data stream contains X sub-streams $\{S_i\}_{i=1}^{X}$. StreamApprox computes 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: $Var(SUM) = \sum_{i=1}^{X} Var(SUM_i)$.

Further, as items are randomly selected for a sample within each sub-stream, according to the random sampling theory [33], the variance of the approximate sum can be estimated as:

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

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:

$$s_i^2 = \frac{1}{Y_i - 1} \times \sum_{j=1}^{Y_i} (I_{i,j} - \bar{I}_i)^2, \text{ where } \bar{I}_i = \frac{1}{Y_i} \times \sum_{j=1}^{Y_i} I_{i,j} \tag{3}$$

Next, the estimation of the variance of the approximate mean value of all items received from all the X sub-streams is described. This approximate mean value can be computed as:

$$\begin{aligned} MEAN &= \frac{SUM}{\sum_{i=1}^{X} C_i} = \frac{\sum_{i=1}^{X} (C_i \times MEAN_i)}{\sum_{i=1}^{X} C_i} \\ &= \sum_{i=1}^{X} (\omega_i \times MEAN_i) \end{aligned} \tag{4}$$

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

$$\begin{aligned} \widehat{Var}(MEAN) &= \sum_{i=1}^{X} Var(\omega_i \times MEAN_i) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times Var(MEAN_i)\right) \\ &= \sum_{i=1}^{X} \left(\omega_i^2 \times \frac{s_i^2}{Y_i} \times \frac{C_i - Y_i}{C_i}\right) \end{aligned} \tag{5}$$

Above, the estimation of the variances of the approximate sum and the approximate mean of the input data stream has been shown. Similarly, the variance of the approximate results of any linear queries also can be estimated by applying the random sampling theory.

Error bound. According to the “68-95-99.7” rule [37], 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 a quantitative understanding of the accuracy of the proposed sampling technique.

4 Discussion

The design of StreamApprox is based on the assumptions mentioned in §2.3. This section discusses some approaches that could be used to meet the assumptions.

I: Virtual cost function. This work currently assumes 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, the sample size for each sub-stream can be determined based on a desired width of the confidence interval using Equation 5 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, there are two existing techniques that could be used to implement such a cost function to achieve the desired resource target: (a) virtual data center [3], and (b) resource prediction model [36] for latency requirements.

Pulsar [3] 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. The cost function could be adapted for STREAMAPPROX as follows: a data item in the input stream is considered 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 computing the sample size that can be processed within the given resource budget.

For any given latency requirement, resource prediction model [36, 34, 35] could be employed. In particular, the prediction model could be built by analyzing the diurnal patterns in resource usage [13] 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. This work 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 the error estimation mechanism still holds correct since STREAMAPPROX applies 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 STREAMAPPROX cannot classify strata based on the sources, the system needs a pre-processing step to stratify the input data stream. This stratification problem is orthogonal to this work, nevertheless for completeness, two proposals for the stratification of evolving streams, bootstrap [17] and semi-supervised learning [25], are discussed in this section.

Bootstrap [17] 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. A bootstrap-based estimator can be employed for the stratification of incoming sub-streams. Alternatively, a semi-supervised algorithm [25] could be used 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.

5 Related Work

Over the last two decades, the databases community has proposed various approximation techniques based on sampling [2, 21], online aggregation [23], and sketches [14]. 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, 22, 32, 24, 29, 28]. 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. Like BlinkDB, Quickr [32] 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, these "big data" systems target batch processing and cannot provide required low-latency guarantees for stream analytics.

IncApprox [24] is a data analytics system that combines two computing paradigms together, namely, approximate and incremental computations [10, 9, 8] for stream analytics. The system is based on an online "biased sampling" algorithm that uses self-adjusting computation [4, 7] to produce incrementally updated approximate output. Lastly, PrivApprox [29, 28] supports privacy-preserving data analytics using a combination of randomized response and approximate computation. By contrast, STREAMAPPROX supports low-latency in stream processing by employing the proposed "online" sampling algorithm solely for approximate computing, while avoiding the limitations of existing sampling algorithms.

6 Conclusion

This paper presents 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, STREAMAPPROX employs an online stratified reservoir sampling algorithm which ensures the statistical quality of the sample selected from the input data stream. The proposed sampling algorithm is generalizable to two prominent types of stream processing models: batched and pipelined stream processing models.

References

1. S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.
2. M. Al-Kateb and B. S. Lee. Stratified reservoir sampling over heterogeneous data streams. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, 2010.
3. S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.
4. P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.
5. P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware)*, 2014.
6. P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. Technical Report MPI-SWS-2012-004, MPI-SWS, 2012. http://www.mpi-sws.org/tr/2012-004.pdf.
7. P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. 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.
8. P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.
9. P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.
10. P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2011.
11. A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the sulq framework. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, 2005.
12. A. Blum, K. Ligett, and A. Roth. A learning theory approach to non-interactive database privacy. In *Proceedings of the fortieth annual ACM symposium on Theory of computing (STOC)*, 2008.
13. R. Charles, T. Alexey, G. Gregory, H. K. Randy, and K. Michael. Towards understanding heterogeneous clouds at scale: Google trace analysis. Techical report, 2012.
14. G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 2012.
15. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2004.
16. A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 2000.
17. D. M. Dziuda. *Data mining for genomics and proteomics: analysis of gene and protein expression data*. John Wiley & Sons, 2010.
18. A. S. Foundation. Apache Flink, 2017.
19. A. S. Foundation. Apache Spark Streaming, 2017.
20. A. S. Foundation. Kafka - A high-throughput distributed messaging system, 2017.
21. M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

22. I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
23. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.
24. D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *Proceedings of the 25th International Conference on World Wide Web (WWW)*, 2016.
25. M. M. Masud, C. Woolam, J. Gao, L. Khan, J. Han, K. W. Hamlen, and N. C. Oza. Facing the reality of data stream classification: coping with scarcity of labeled data. *Knowledge and information systems*, 2012.
26. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
27. S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.
28. D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. Privacy preserving stream analytics: The marriage of randomized response and approximate computing. `https://arxiv.org/abs/1701.05403`, 2017.
29. D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-Preserving Stream Analytics. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2017.
30. D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. Approximate Stream Analytics in Apache Flink and Apache Spark Streaming. *CoRR*, abs/1709.02946, 2017.
31. D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. StreamApprox: Approximate Computing for Stream Analytics. In *Proceedings of the International Middleware Conference (Middleware)*, 2017.
32. K. Srikanth, S. Anil, V. Aleksandar, O. Matthaios, G. Robert, C. Surajit, and B. 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.
33. S. K. Thompson. *Sampling*. Wiley Series in Probability and Statistics, 2012.
34. A. Wieder, P. Bhatotia, A. Post, and R. 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.
35. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.
36. A. Wieder, P. Bhatotia, A. Post, and R. 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.
37. Wikipedia. 68-95-99.7 Rule, 2017.
38. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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.
39. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.