Strainer: Windowing-based Advanced Sampling in Stream Processing Systems

Nikola Koevski[1], Sérgio Esteves[1], and Luís Veiga[1]

INESC-ID, Instituto Superior Téncico, Universidade de Lisboa

Abstract. Stream Processing is a very effective predominant paradigm for data processing. It provides an efficient approach to extract information from new data, as the data arrives. However, spikes in data throughput, can impact the accuracy and latency guarantees stream processing systems provide. This work proposes data sampling, a type of data reduction, as a solution to this problem. It provides a user-transparent implementation of two sampling methods in the Apache Spark Streaming framework. The results show a reduced amount of input data, leading to decreased processing time, but retaining a good accuracy in the extracted information.

1 Introduction

Big Data has brought a revolution to data processing. With commodity hardware becoming cheaper and widely available, constraints on the amount of data to be collected have been lifted. As a result, useful information, patterns and insights have become far easier to extract. A variety of Big Data processing is on-the-fly data processing called stream processing.

For stream processing systems to provide an efficient service, data needs to be processed as fast as it arrives. When a sudden peak in data throughput occurs, greater than the processing capabilities of the system, several problems arise. When possible, the system will utilize additional computing resources. Next, if the available resources are not enough, the system will try to queue new data while it processes available data. This in turn may lead to a delay in the results, lowered accuracy from an overflowing queue, and an eventual crash of the system.

An obvious solution to the problem is to scale out by adding machines to the system. Next, changing the size of the data to be processed may be attempted [5]. Another approach is to use controlled data reduction methods like load shedding [20,21,19] or sampling [12,9], alternatives to compression [16].

However, additional machines may be unavailable or too costly to provide, and altering the input data size would increase latency. Although effective, load shedding may skew the data distribution lowering the result accuracy. In contrast, sampling decreases data size by producing a subset retaining desired characteristics of the whole data set. This provides lower resource requirements, lower latency, but maintains a good result accuracy.

This paper contributes to the utilization of sampling in stream processing systems. We implemented Strainer, a stream processing sampling framework.

Coupled with the Apache Spark Streaming framework, in Strainer we employ two sampling algorithms. We evaluate the advantages and cost this usage of sampling techniques incurs in the accuracy guarantees of systems like Spark. The result is an early-stage data reduction in the workflow producing a smaller processing load, shorter execution times while keeping a limited result error.

The remainder of this paper is structured as follows. Section 2 presents the necessary background to frame the paper. Section 3 details the design and implementation of Strainer, and its evaluation follows in Section 4. Then, Section 5 reviews and contrasts relevant work within the state-of-the-art, and Section 6 concludes the paper and gives insights on future directions.

2 Background and Assumptions

Sampling methods and their application in Big Data are initially thoroughly analysed in earlier work [4]. As their work suggests, among the varied methods of data reduction available, sampling provides an intuitive and straightforward way to obtain a smaller subset of the data with the same structure. Thus, they show it a valid choice as a method to reduce data for real-time data processing.

Apache Spark Streaming is a mature data processing framework, speeding up processing times by performing in-memory processing. Furthermore, Spark's modular design allows it to integrate with a multitude of different technologies, from Hadoop's HDFS for distributed storage, YARN or Apache Mesos for resource management, to providing libraries for connecting with data sources like SQL, Apache Kafka, Cassandra, Kinesis, as well as Twitter.

As seen in Figure 1, in Spark Streaming, the data is admitted into the system through the Receiver module. The Receiver provides the flexibility to connect with various data sources. Moreover, it allows data items to be pre-processed before being admitted into the workflow. Through the Receiver Supervisor, the Receiver gathers the data items into blocks and then stores them into memory. Furthermore, the Supervisor generates block meta-data and then inserts it into a queue at the Receiver Tracker. Next, Spark Streaming utilizes an interval to build a small batch from the enqueued meta-data. The length of this batch interval determines the size of the micro-batches which are then processed by a user-defined streaming application.

While micro-batches are the reason Spark does not provide "true" real-time stream processing, they are useful to us. Spark Streaming abstracts the data stream into micro-batches, so each micro-batch can be processed as a regular Spark batch application, and have a known size to be considered when sampling.

3 Framework Integration and Sampling Techniques

3.1 Framework Integration

The micro-batch abstraction mentioned in the previous section is what allows a seamless integration of Strainer with Spark Streaming. Figure 1 shows the sampling framework implemented as a wrapper at the Receiver module. Strainer

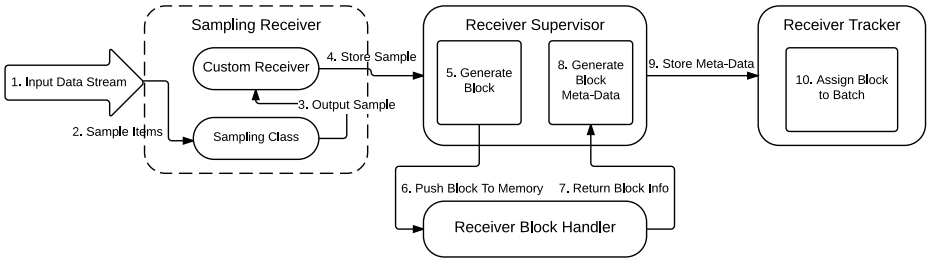


Fig. 1: Basic Architecture of Batching module in Spark Streaming

intercepts each data item before it is stored and passes it through a class implementing a sampling algorithm. Next, before the batch interval passes, the framework outputs the sampled items to the Supervisor. It uses the sampled data to generate blocks, continuing a standard batching operation. Thus, with Strainer the pre-existing functionality of the batching module remains unaltered.

3.2 Sampling Techniques

Before the implementation, several sampling techniques were considered. The following criteria were used for selecting the sampling methods [4,11]. The algorithms needed to implement the reservoir scheme, providing a one-pass sample over an unbounded data stream. The reservoir sampling scheme provides a fixed size sample with a single pass over an arbitrary sized data stream. However, reservoir scheme algorithms use uniform sampling which can skew the data distribution of the sampled set. Thus, algorithms that use techniques that can counter this data distribution skew were required. Finally, an algorithm needed to provie a bounded error guarantee in order to be selected.

Congressional algorithm [1] Congressional sampling is an efficient method of performing sampling when data is partitioned in groups. A considerable number of data processing applications group data by key. The MapReduce paradigm is a relevant example of this. Furthermore, Congressional sampling is a hybrid of uniform and biased sampling. This guarantees that both large and small groups will be represented in the sample, preventing data distribution skew. Algorithm 1 shows the algorithm for Congressional sampling.

As can be seen on lines 5, 6 and 8, in the first stage, the algorithm performs three types of sampling. First, it performs a *house* (standard uniform reservoir) sample. Next, a *senate* sample is performed, which assigns an equal slot of the sample size to each group. Finally, a *grouping* sample is performed for each attribute in the group-by set, where each attribute's "grouping" is assigned a sample slot proportional to the size of the grouping in the data set. Second, in the grouping sample, the slot size for each group is recalculated (line 13).

Equation 1 shows the equation, where S is the sample size, mT is the number of distinct groups, N_g is the number of items for the attribute and N_h represents

Algorithm 1 Congressional algorithm

```
1: initialize(sampleSize, group)
2: sampleCount ← 0
3: houseSample ← ∅
4: senateSample ← ∅
5: groupingSample ← ∅
6: for all item ∈ dataStream do
7:     doHouseSample(item)
8:     doSenateSample(item)
9:     for attribute ∈group do
10:        doGroupingSample(item)
11:    end for
12: end for
13: getFinalCongressionalGroups(groupingSample)
14: calculateSlots(houseSample,senateSample, groupingSample)
15: scaleDownSample()
```

the number of items in the distinct group. In the next stage (lines 14 and 15), the group sizes of the uniform, senate and grouping samples are evaluated and the final slot size for each group is calculated from the house, senate and grouping samples.

$$GroupSize = (S/mT) * (N_g/N_h) \tag{1}$$

In Equation 2, S is the sample size, $max_{g \in G} S_g$ is the size of the largest slot for a group from the house, senate and grouping samples and it is divided by the sum of all the slot sizes for that group. Finally, each group is re-sampled with reservoir sampling to generate a sample slot with the new size. The house sample allocates more space for larger groups. On the other hand, the senate sample allows smaller groups to enter the sample. Finally, the grouping sample optimizes the separate attribute representations inside each group.

$$SlotSize = S * (max_{g \in G} S_g / \sum_{sampletype} max_{g \in G} S_g) \tag{2}$$

Distinct Value algorithm [7] As its name suggests, the Distinct Value sampling method approximates the number of distinct values of an attribute in a given data stream. As with the previous algorithm, determining the distinct values of a certain attribute is frequently used in the optimization of the computation flow. The DV sampling algorithm provides a low, 0-10% relative error, while providing a low space requirement of $O(log_2(D)$, where D is the domain size of the attribute.

Algorithm 2 presents the Distinct Value algorithm. It requires two additional parameters besides the sample size. The second parameter is the maximum sample slot size per value, called the threshold. The third parameter is the domain size, representing the number of possible values that can occur.

The algorithm works as follows. As each data item arrives, the domain size is used to generate a hashed value of the data item. Next, if the hashed value is at least as large as the current level, an attempt to put the item in the appropriate

Algorithm 2 Distinct Value algorithm

```
1: initialize(sampleSize, threshold)
2: level ← 0
3: sampleCount ← 0
4: Sample ← ∅
5: CountMap ← ∅
6: for all item ∈ dataStream do
7:     hashValue ← dieHash(item)
8:     if hashValue ≥level then
9:         if Sample(hashValue) <threshold then
10:            Sample(hashValue).add(item)
11:            CountMap(hashValue) + +
12:            sampleCount + +
13:        else
14:            Sample(hashValue).sample(item)
15:        end if
16:    end if
17:    if sampleCount>sampleSize then
18:        sampleCount− =Sample(level)
19:        Sample(level).remove
20:        level = level + 1
21:    end if
22: end for
```

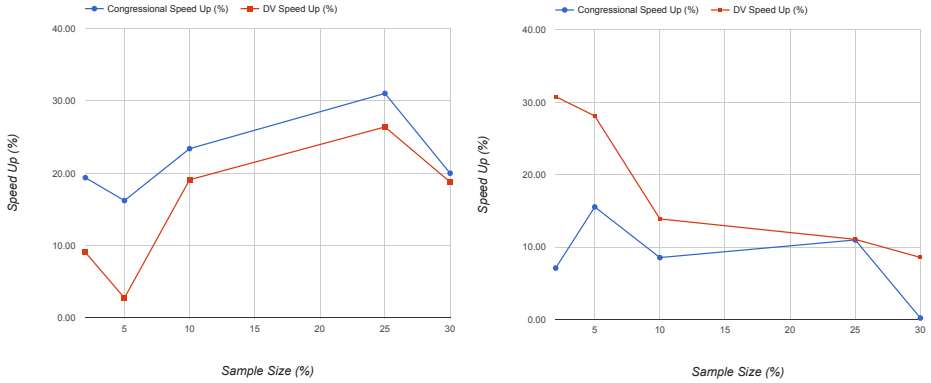
hash value slot is performed. If the slot size is smaller than the threshold value, the item is simply placed in the slot. Otherwise a uniform sample is performed which can result in the new item replacing an item currently in the slot. When the items in the sample exceed the sample size, the slot whose value equals the current level number is removed from the sample and the level is incremented.

By randomly mapping the attribute values to hashed values and only allowing hashed values equal or greater than the current level to enter the sample, the algorithm ensures that the sample contains a uniform selection of the scanned portion of the data stream. As an addition, the threshold value keeps the level from frequently incrementing and skewing the data distribution.

4 Experimental Evaluation

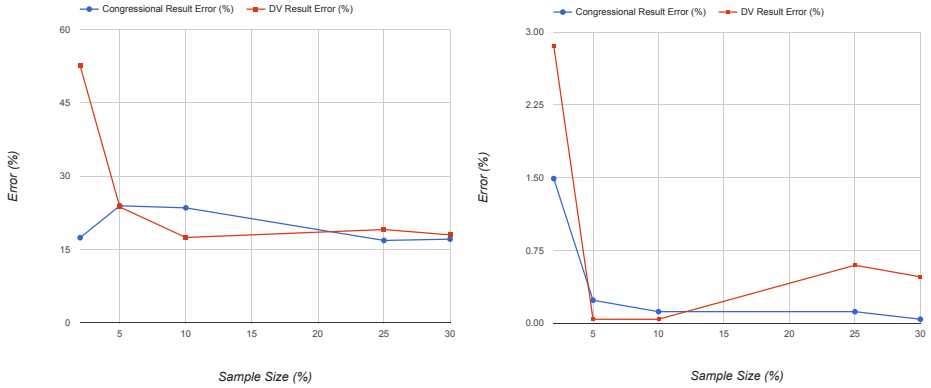
For the experimental evaluation, we employed an instance of a server representative of elements in typical cloud deployments. The server runs on an 8-core, 2.93GHz Intel i7 processor with 12GB of RAM, using 64-bit Ubuntu Server. The system was implemented on Apache Spark, while the data streams were created using the Netcat Linux command-line tool. For measuring the maximum heap memory usage, a light-weight console application was used, called Jvmtop.

Metrics and Benchmarks. In order to understand the gains of the implemented system, four metrics were used. Two are performance metrics, evaluating the speed-up in processing time and the variation in memory consumption. The other two are error metrics, estimating the relative error in the generated sample and the relative error in the results of the benchmark applications. Two benchmark applications were used, usually employed in streaming benchmarks. The first is one that provides the most used payment type in New York taxis. The second provides the country with most customers of an online retail website.



(a) Retail Speed Up (b) Taxi Log Speed Up

Fig. 2: Processing time speed up benchmark applications

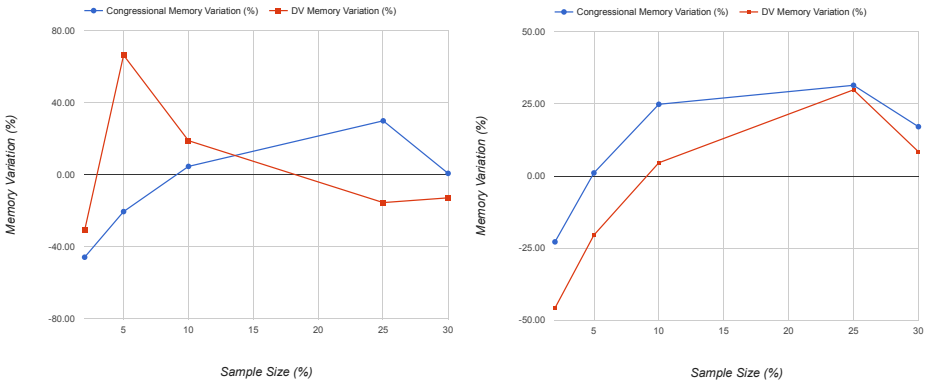


(a) Retail Error (b) Taxi Log Error

Fig. 3: Result error plots for benchmark applications

Results. The speed-up in processing time for both applications is shown in Figure 2. For the Online Retail application (2a), both algorithms show a high speed-up in processing time (20-30%) for sampling sizes of 10, 25 and 30%, but the 2 and 5% sample shows that sampling is rendered ineffective for too small data inputs. However, the Taxi log application (2b), which has a much smaller domain size for the target attribute of the sampling, shows a steady decrease of speed-up, providing high values for the smaller sample sizes.

On Figure 3 the relative error of the application results is presented. As can be seen on Figure 3a, the algorithms in the Online Retail application maintain steadily decreasing error with a maximum of 20%, while the error in the Taxi log application (Figure 3b) is kept bellow 1%. Exceptions are the 2% sampling sizes, where, because of the greatly decreased data size, the differences in the results are much more noticeable.



(a) Retail Maximum Memory (b) Taxi Log Maximum Memory

Fig. 4: Maximum memory usage variation for benchmark applications

The plots in Figure 4 detail the maximum heap memory usage of Spark Streaming during the execution of the benchmark applications. As can be seen, the sampling runs actually consume more memory for most of the sample sizes. The reason behind this is that both algorithms obviously require some constant extra amount of memory for the data structures they use for maintaining the meta-data of the sampled elements; meanwhile, all events are still injected in Spark even if not fully processed, with the garbage collector sometimes freeing their memory only lazily in the background. An exception is the DV algorithm for the Taxi log application, where because of the smaller domain size, the DV sample actually performs better. This is affected by the threshold value which greatly impacts the slot sizes and increment frequency of the level value, thus increasing or decreasing the memory usage.

From the evaluation metrics, it can be summarized that the system can provide significant gains in processing time for the sample sizes between 5 and 25 percent, while maintaining a low error rate. However, this is done at the cost of additional memory consumption.

5 Related Work

Strainer is an approximate computing system that intersects data reduction with data processing platforms. There are several notable works in these areas.

In the area of sampling, several one-pass sampling algorithms can be adapted to streamed data. Reservoir sampling [22] is a uniform sampling algorithm. It provides a bounded error, but may skew data distribution. Count and Weighted sampling [8,2] use biased sampling methods. However, both have no error bounds. Furthermore, Weighted sampling introduces overhead information about the weights of the data items in advance.

Currently, there is an abundance of data processing platforms. Apache Flink, Storm and Samza offer stream processing libraries. In contrast to Spark, they use

a streaming dataflow engine which performs true streaming, thus immediately processing each data element. However, this becomes an obstacle when trying to sample data, since most sampling methods need to first build a sample set.

Approximate computing systems use two approaches in data reduction. Works on Aurora and Borealis [20] tightly integrate load shedding operators that discard tuples throughout their operation paths. Another work on Aurora/Borealis [19] groups tuples into blocks, which then selectively discards. Comparably, the system in [21] divides the input data stream into windows which are probabilistically discarded. Like Strainer, IncApprox [12] uses sampling to reduce the input data. However, it additionally utilizes incremental computing to increase the efficiency of the system. Finally, ApproxHadoop [9] uses multi-stage sampling as the first stage of data reduction, and adds task dropping as a load shedding approach for the second stage.

In the realm of big data analytics, a plethora of research has been outlined [15], delving into the intricacies of resource efficiency within large-scale data processing clusters. A relevant contribution to this field is the utilization of hidden Markov models to predict frequent patterns and approximate computation, a method that has achieved a notable level of accuracy as discussed in [13]. Additionally, the study presented in [18] introduces a model for predicting applications' resource allocation needs by using *a priori* knowledge, which is also beneficial for effective resource mapping. Another significant approach is found in [10], where the focus is on the estimation of actual available resources to prevent scenarios of resource wastage. Furthermore, the innovative application of game theory in managing container allocation for streaming applications, as explored in [17], presents a unique perspective in resource management. Each of these studies offers distinct insights and solutions to the challenges associated with resource management in the context of big data.

Other approaches employ parameter tuning [14] to Map-Reduce and Spark workloads, or employ machine learning in order to determine how to ensure a specific error bound in continuous Map-Reduce workflows [6] while delaying reexecution as much as possible when new input data arrives. This has also been attempted in iterative or continuous graph processing [3].

Overall, none of the previous works employ sampling as a means to ensure application quality-of-service during overcommit/overallocation situations, in a way that does not risk dropping relevant, yet possibly under-represented, types or values of data (as uniform-based sampling approaches typically employed in load-shedding can incur in).

6 Conclusion

Strainer implements the approximate computing paradigm by leveraging the advantages of sampling as a data reduction technique. It utilizes the modularity of the Apache Spark Streaming to create a seamless merging of this established data processing framework with the Congressional and Distinct Value sampling

methods. Thus, it provides a user-transparent framework for the development of approximate computing applications.

The experimental results indicate that the system can be employed in data stream environments and provide a faster execution time while maintaining a low error bound. Although it is fully functional for stable data streams, the introduction of a variable arrival rate in the data stream may impact the accuracy of the results. This is because the sample size would maintain a fixed value while the amount of data fluctuates. Implementing a self-adjusting sampling size depending on the error measurement and processing time would alleviate this problem. Finally, sampling could be performed in a dedicated component with custom memory management or more frequent GC to promote memory savings.

Acknowledgements: This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020). This work was supported by: "DL 60/2018, de 3-08 - Aquisição necessária para a atividade de I&D do INESC-ID, no âmbito do projeto SmartRetail (C6632206063-00466847)". The authors would like to thank colleague Rodrigo Rodrigues for the initial collaboration in this work.

References

1. S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 487–498, New York, NY, USA, 2000. ACM.
2. S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 534–542. IEEE, 2001.
3. M. E. Coimbra, S. Esteves, A. P. Francisco, and L. Veiga. Veilgraph: incremental graph stream processing. *J. Big Data*, 9(1):23, 2022.
4. G. Cormode and N. Duffield. Sampling for big data: A tutorial. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1975–1975, New York, NY, USA, 2014. ACM.
5. T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.
6. S. Esteves, H. Galhardas, and L. Veiga. Adaptive execution of continuous and data-intensive workflows with machine learning. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 239–252, New York, NY, USA, 2018. Association for Computing Machinery.
7. P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 541–550, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
8. P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 331–342, New York, NY, USA, 1998. ACM.

9. 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 '15, pages 383–397, New York, NY, USA, 2015. ACM.
10. S.-H. Ha, P. Brown, and P. Michiardi. Resource management for parallel processing frameworks with load awareness at worker side. In *Big Data (BigData Congress), 2017 IEEE International Congress on*, pages 161–168. IEEE, 2017.
11. W. Hu and B. Zhang. Study of sampling techniques and algorithms in data stream environments. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pages 1028–1034. IEEE, 2012.
12. 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 '16, pages 1133–1144, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.
13. C.-M. Liu and K.-T. Liao. Efficiently predicting frequent patterns over uncertain data streams. *Procedia Computer Science*, 160:15 – 22, 2019. The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops.
14. J. Lu, Y. Chen, H. Herodotou, and S. Babu. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *Proc. VLDB Endow.*, 12(12):1970–1973, Aug. 2019.
15. A. Mohamed, M. K. Najafabadi, Y. B. Wah, E. A. K. Zaman, and R. Maskat. The state of the art and taxonomy of big data analytics: view from new big data framework. *Artificial Intelligence Review*, 53(2):989–1037, 2020.
16. B. Morais, M. E. Coimbra, and L. Veiga. pk-graph: Partitioned k 2-trees to enable compact and dynamic graphs in spark graphx. In *International Conference on Cooperative Information Systems*, pages 149–167. Springer, 2022.
17. O. Runsewe and N. Samaan. Cram: a container resource allocation mechanism for big data streaming applications. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 312–320, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.
18. A. Shukla and Y. L. Simmhan. Model-driven scheduling for distributed stream processing systems. *CoRR*, abs/1702.01785, 2017.
19. L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1115–1126, New York, NY, USA, 2014. ACM.
20. N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 159–170. VLDB Endowment, 2007.
21. N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 799–810. VLDB Endowment, 2006.
22. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.