Advanced Sampling in Stream Processing Systems

(extended abstract of the MSc dissertation)

Nikola Koevski
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Supervisors: Prof. Luís Veiga and Prof. Rodrigo Rodrigues

Abstract

Stream Processing is the new paradigm in 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 [3]. Furthermore, a mini-framework is implemented for the development of additional sampling methods. 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. Now, Big Data processing has moved onto a more on-the-fly type of 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. To cope with this demand, the system will utilize additional computing resources. Next, if the available resources aren't enough, the system will try to queue new data while it processes the 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 add machines to the system. Next, changing the size of the data to be processed may be attempted [10]. Another alternative is to use controlled data reduction methods like load shedding [18, 17, 19, 16] or sampling [15, 13].

However, additional machines may be unavailable or 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 the characteristics of the whole data set. This provides smaller resource requirements, lower latency, but keeps a good result accuracy.

This paper contributes to the utilization of sampling in stream processing systems. For these purposes a single-point, user-transparent sampling framework. Coupled with the Apache Spark Streaming framework [3], this framework was used to enforce two sampling algorithms. Finally, the advantages and cost this usage in advanced sampling techniques incurs in the accuracy guarantees of systems like Spark is evaluated. The result is an early-stage data reduction in the workflow producing a smaller processing load, shorter execution times while keeping a low result error.

The remainder of this paper is structured as follows. Section 2 presents the necessary background on stream processing and "big data" analytics to understand the paper. Section 3 details the design and implementation of the framework, 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 thoroughly analysed in the work by [9]. 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 proved to be the best choice as a method to reduce data for real-time data processing.

Apache Spark Streaming [3] proved to be the best choice to tackle out goals and challenges. Spark 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.

These data sources provide a continuous stream of data which Spark Streaming processes. As seen in figure 1, the data is admitted into the system through the Receiver module. The Receiver provides Spark the flexibility to connect with data sources beyond the ones mentioned previously. 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.

Micro-batches are the reason Spark does not provide "true" real-time stream processing. Spark Streaming abstracts the data stream into micro-batches, so each micro-batch can be processed as a regular Spark batch application. However, a spike in data throughput can cause an increase in the batch size, leading to a delay in batch processing.

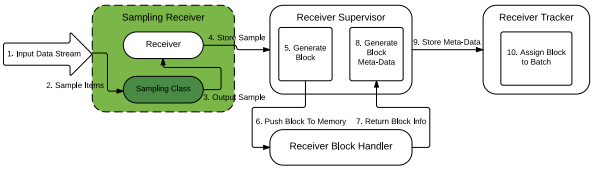


Figure 1: Basic Architecture of Batching module in Spark Streaming

3 Framework Integration and Sampling Techniques

3.1 Framework Integration

The micro-batch abstraction mentioned in the previous section is what allowed a seamless integration of the framework developed in this work with Spark Streaming. Figure 1 shows the sampling framework implemented as a wrapper at the Receiver module. The framework 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. The Supervisor uses the sampled data to generate blocks, continuing a standard batching operation. As a result, the old 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 [9, 14]. 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.

3.2.1 Congressional algorithm [7]

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 considerable proof 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.

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()
```

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 "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).

$$GroupSize = (S/mT) * (N_g/N_h) \quad (1)$$

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 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.

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

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.

3.2.2 Distinct Value algorithm [11]

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 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

Algorithm 2 Distinct Value algorithm

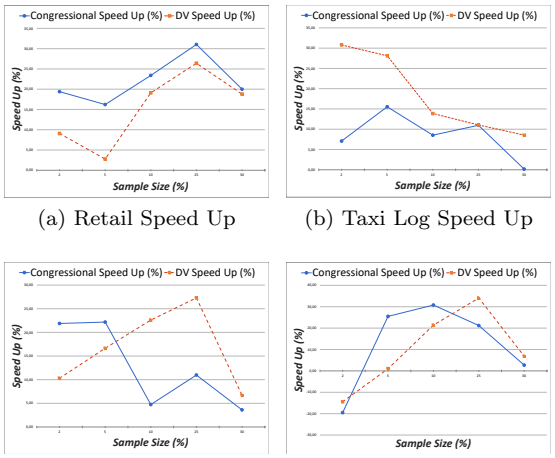
```
1: initialize(sampleSize, threshold)
2: level ← 0
3: sampleCount ← 0
4: Sample ← ∅
5: CountMap ← ∅
6: for all item ∈ dataStream do
7:     hashValue ← dieHash(item)
8:     if hashValue ≥level then
9:         if Sample(hashValue) <threshold then
10:            Sample(hashValue).add(item)
11:            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
```

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

4.1 Experimental configuration

For the experimental evaluation a single server was used. The server runs on an 8-core, 2.93GHz Intel i7 processor with 12GB of RAM, using a 64-bit Ubuntu Server 14.04 LTS operating system. The system was implemented on version 1.6.0 of Apache Spark, while the data streams were created using the Netcat [6] Linux command-line tool. For measuring the heap memory usage, a light-weight console application was used, called Jvmtop [5].



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

(c) Apple NASDAQ Tweets Speed Up (d) US IT Stocks Speed Up

Figure 2: Processing time speed up plots for the Online Retailer (a), Taxi Log Analysis (b), Apple NASDAQ tweets Analysis (c) and US IT Stock Analysis (d) benchmark applications

4.2 Metrics

In order to gain a better understanding of the acquired gains of the implemented system, three metrics were used. From them, two are quantitative metrics, evaluating the speed-up in processing time and the variation in memory consumption. The last is a qualitative metric, estimating the relative error in the generated sampled data for the benchmark applications.

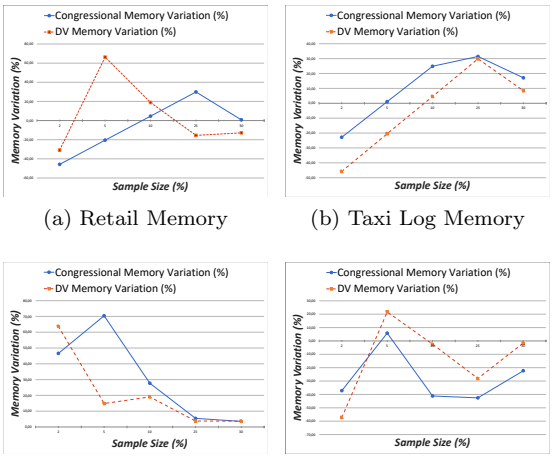
Four benchmark applications were used. The first is an application that analyses Apple NASDAQ tweets and provides the top ten language speaking groups that post statuses on Twitter connected to the Apple stocks. The second is a stock analysis application, providing the company that had the largest positive growth in stock value in a given sliding window interval. The third is an application that provides the most used payment type in New York taxis. The last application provides the country with most customers of an online retail website.

The speed-up in processing time for all applications is shown in Figure 2. For the Online Retail ap-

plication (Figure 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 (Figure 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, 30% and 28% for the 2% and 5% samples of the DV algorithm, and a maximum of 15% speed up for the 5% Congressional sample. The Apple tweet analysis application, shown on Figure 2c, shows two different plot lines for the algorithms. The DV algorithm reports a high speed up in the sample size interval between 5% and 25%, providing a 27% speed up for the 25% sample, 22%, 16% and 10% speed up for the 10%, 5% and 2% samples correspondingly. On the other hand, the Congressional algorithm shows a decreasing speed up, providing a 22% speed up for the 5% sample size, a 21% speed up for the 2% sample and a 10% and lower speed up in processing time for the 25%, 10% and 30% samples. The results of the stock analysis application, shown on Figure 2d, show that in both plot lines the highest speed up is gained between the sample size interval of 5% and 25%. The speed up starts decreasing with sample sizes bigger that 25%. The highest speed up is provided by the Distinct Value algorithm, with a 34% for the 25% sample, and 21% speed up for the 10% sample size. The Congressional sampling algorithm provides a high, 30% speed up for the 10% sample size, with 25% and 21% speed ups for the 5% and 25% samples accordingly.

The plots in Figure 3 detail the heap memory usage of Spark Streaming during the execution of the benchmark applications. The memory usage of the Retail application is shown on Figure 3a. Both algorithms use up more memory to achieve the above mentioned speed up. The Congressional sampling uses the most additional memory, 31%, with the 25% sample, while the Distinct Value algorithm follows closely, with 30% for the same sample size. The Taxi Logs application shows similar plot lines (Figure 3). The Congressional sampling uses the most additional memory, 31%, with the 25% sample, while the Distinct Value algorithm follows closely, with 30% for the same sample size. Figure 3c reports the memory usage of the Apple tweets analysis application. As with the previous applications, both algorithms use up more memory than a normal run. The Congressional algorithm uses up the most memory, with an additional 70% more memory for the 5% sample size, while the Distinct Value algorithm utilizes up to 63% additional memory for the 2% sample size. The reason for this is that the sampling operations in the algorithms have to increase to cope with the decreased sample size. On the other hand, for the Stock analysis application (Figure 3d) both algorithms show that, with sampling, less memory is used, steadily increasing the memory consumption as the sample size is increased. As shown with the plot lines, the Congressional algorithm uses less memory, in general, than the DV algorithm, using 42% and 41% less memory for the 10% and 25% sample sizes appropriately.

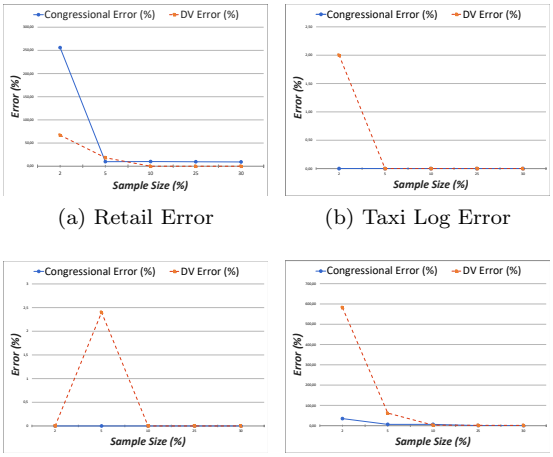
On Figure 4 the relative error of sampled data sets is presented. As can be seen on Figure 4a, the algorithms in the Online Retail application maintain a decreasing error. Both report a high error for the 2% sample, since the small size of this sample distorts the distribution of the data. However, the error drops below 10% for the larger sample sizes for both algorithms. The Taxi log application (Figure 4b) keeps a 0% error for both algorithms, with an exception of 2% error for the 2% DV sample. Simi-



(a) Retail Memory (b) Taxi Log Memory

(c) Apple NASDAQ Tweets Memory (d) US IT Stocks Memory

Figure 3: Memory variation plots for the Online Retailer (a), Taxi Log Analysis (b), Apple NASDAQ tweets Analysis (c) and US IT Stock Analysis (d) benchmark applications



(a) Retail Error (b) Taxi Log Error

(c) Apple NASDAQ Tweets Error (d) US IT Stocks Error

Figure 4: Result error plots for the Online Retailer (a), Taxi Log Analysis (b), Apple NASDAQ tweets Analysis (c) and US IT Stock Analysis (d) benchmark applications

larly, both algorithms report a 0% error rate for the Apple tweets application for all of the sample sizes (Figure 4c). An exception is the 5% sample size of the Distinct Value algorithm. The Stock analysis application shown on Figure 4d shows a very large error, 581%, for the 2% DV algorithm sample and nearly 35% for the appropriate Congressional sample. The error significantly lowers for the larger sample sizes, reporting 6.5% for the 5% and 10% samples, and 0.5% and 0% for the 25% and 30% samples of the Congressional algorithm. The Distinct Value algorithm shows a high error value of 62% for the 5% sample, but decreases the error to 2% for the 10% sample and 1% for the 25% and 30% sample sizes.

From the evaluation metrics, it can be concluded that while both algorithms generally consume more memory than a normal execution, they show different behaviour to different sample sizes. The Congressional sampling algorithm reports a more stable, low error operation for low sampling sizes, suggesting that it is better suited for sample generation when smaller samples are desired. On the other hand, the Distinct Value algorithm shows a better performance for larger sampling sizes, where the error and processing time is better than the Congressional algorithm.

5 Related Work

Strainer, an approximate computing system 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 [20] is a uniform sampling algorithm. It provides a bounded error, but may skew data distribution. Count and Weighted sampling [12, 8] 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 [1], Storm [4] and Samza [2] all 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 [18] and Borealis [17] tightly integrate load shedding operators that discard tuples throughout their operation paths. Another work on Aurora/Borealis [16] groups tuples into blocks, which then selectively discards. Comparably, this system [19] divides the input data stream into windows which are probabilistically discarded. Like Strainer, IncApprox [15] uses sampling to reduce the input data. However, it additionally utilizes the Incremental computing to increase the efficiency of the system. Finally, ApproxHadoop [13] uses multi-stage sampling as the first stage of data reduction and adds task dropping as a load shedding approach for the second stage.

6 Conclusion

The system 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 cre-

ate 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 system showed that current data processing systems can still benefit from advancements made before the Big Data Revolution. The experimental results indicate that the system can be employed in heavy data stream environments and provide a faster execution time while maintaining a low error bound.

6.1 Future work

Future work may address the implementation of a self-adjusting sample size depending on the error measurement and processing time. This may be further expanded by recording the results of prior executions to remember the best parameters of a sampling algorithm and adjust these parameters for each future job. Additionally, allowing the definition of QoS thresholds for error and accuracy overheads would gain the best resource usage for the best available speed up. Finally, a module to detect resource usage and shift the execution of an application from normal to sampled mode would provide the optimal performance and resource utilization.

References

[1] Apache flink. `https://flink.apache.org/`. Accessed: 2016-08-07.

[2] Apache samza. `http://samza.apache.org/`. Accessed: 2016-08-07.

[3] Apache spark streaming. `http://spark.apache.org/streaming/`. Accessed: 2016-08-07.

[4] Apache storm. `http://storm.apache.org/`. Accessed: 2016-08-07.

[5] Jvmtop. `https://github.com/patric-r/jvmtop`. Accessed: 2016-08-07.

[6] Netcat. `http://netcat.sourceforge.net/`. Accessed: 2016-08-07.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[19] 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.

[20] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.