Streaming Machine Learning Algorithms with Big Data Systems

Vibhatha Abeykoon*, Supun Kamburugamuve*†, Kannan Govindrarajan*, Pulasthi Wickramasinghe*,
Chathura Widanage*, Niranda Perera*, Ahmet Uyar†,
Gurhan Gunduz†, Selahattin Akkas* and Gregor Von Laszewski*†
*School of Informatics, Computing and Engineering, IN 47405, USA
{vlabeyko, skamburu, kgovind, pswickra, cdwidana, dnperera,sakkas}@iu.edu
†Digital Science Center Bloomington, IN 47408, USA
{auyar,ggunduz,gvonlasz}@iu.edu

***Abstract*—Designing low latency applications that can process large volumes of data with higher efficiency is a challenging problem. With limited time to process data, usage of online algorithms is becoming important in big data applications. Stream processing is a well-known area that has been studied extensively for a long time. In this research, our objective is to use state-of-the-art big data analytic engines to implement online algorithms and compare the strengths and weaknesses in each system. We use a streaming version of Support Vector Machines (SVM) and KMeans to do the analysis. Apache Flink, Apache Storm and Twister2 streaming frameworks are used to implement these algorithms. Our study focuses on the efficiency of online training of these algorithms, the results of which show higher (better) performance in Twister2 framework overall.**

***Index Terms*—Big Data, Streaming Machine Learning, Dataflow**

I. Introduction

In the modern information technology era, large amounts of data collection is ubiquitous. This in turn means all such collected data needs to be processed to discover meaningful insights hidden within. Data sources range from that collected in social media platforms to signal data collected from small devices such as sensors. Even a simple data processing step such as a filter which simply discards data based on some predefined conditions has become a challenging task due to the amount and the speed at which it is collected.

In recent years, stream processing has become one of the most prominent modes of processing large volumes of data with low latency. Among the applications which produce a large volume of data, internet-of-things-related applications, social media data processing applications, video processing applications, and audio processing applications can be denoted as the most prominent cases. In most such applications the requirements for data processing go beyond simple filter operations, therefore modern stream processing engines need to be able to support more complex machine learning algorithms. While many of the popular stream engines such as Apache Spark, Apache Flink, and Apache Storm provide the basic building blocks needed to develop streaming machine learning applications, the approaches that have been taken by each system vary, resulting in different programming models and varying performance numbers. The objectives in this paper are twofold; First, we aim to analyze the application development styles in each stream processing system to identify subtle differences in the various programming models adopted by popular frameworks; Second, to showcase the performance of each system using streaming machine learning applications.

We believe that applying HPC system design principles to big data systems can make them more efficient. Twister2 is a data processing framework with HPC design principles. The efficiency of Twister2 for streaming applications is showcased in the experiments and results shown in this paper by comparing it with Apache Flink and Apache Storm. We use online K-Means and Support Vector implementations to measure the performance of these systems.

In Section II we discuss the role of stream processing in the big data domain and its importance. Section III highlights two streaming machine learning algorithms in detail. In Section IV the experiments conducted to compare various streaming engines are discussed and the results are presented. Section V describes related work and Sections VI and VII conclude the paper.

II. Stream Processing with Big Data Stack

Stream processing is mostly used with big data-related applications rather than high performance computing applications. In the big data domain, the most prominent and well-known stream processing engines are Apache Flink, Apache Storm and Apache Spark. These stream processing engines have been used by many application developers and researchers to implement applications. Twister2-Streaming is another framework developed by the authors for the same purpose. It can leverage high performance networks and optimized collective operations present in HPC systems for better performance. Twister2 provides the core functionality of the mainstream stream processing systems.

The programming model in Apache Storm is more flexible than most of the other stream processing frameworks due to the lower-level abstraction in its API. This was one of the core features in Apache Storm until the release of version 2.0.0. Still, the user has the capability of developing applications and writing custom APIs on top of the core API. In Apache Spark and Apache Flink, the programming model has been designed on top of a high-level abstraction. This allows the user to develop applications much faster. It also adds a certain level of restriction in application development, which in turn results in lower efficiencies. In Apache Spark, the only way to write a dataflow model is to use a high level API abstraction. Twister2 provides many levels of programming abstractions for user to choose from in order to develop their applications efficiently in terms of usability and performance.

III. Streaming Machine Learning Algorithms

In our research, we use the online versions of two machine learning algorithms, namely SVM and KMeans. In our analysis, we portray how the streaming model is being implemented with Apache Flink, Apache Storm and Twister2 stream processing engines. Here we use the window processing API in each framework to discretize the continuous stream of data.

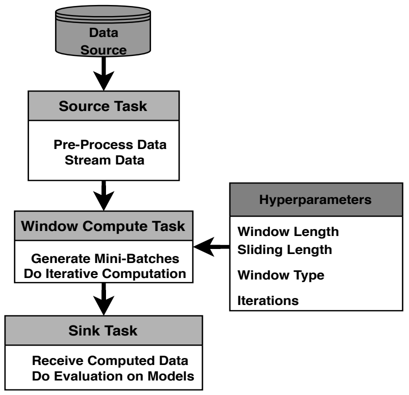

Fig. 1. Workflow of a Streaming ML Application

A standard streaming machine learning algorithm takes the task architecture as shown in Figure 1. The data source can be from a file, HDFS or a message broker like Apache Kafka [1]. In all the experiments, a source task reads data from the file system and performs the data pre-processing. Further down the stream, a window task is designed with some hyper-parameters. The main hyper-parameters involved in a streaming machine learning algorithm include the number of iterations. In KMeans this value was unity, and in SVM it is a finite number set by the user. Window length and sliding window length are important parameters for windowing. Windowing type can be determined as count-based or time-based. In the experiments conducted, we only used count-based windows. Within the window task, a mini-batch is generated using stream discretization which is then processed by the algorithm. Here each algorithm provides a sub-optimal solution respective to the corresponding batch algorithm. This will be further discussed in III-A. Once the streaming algorithm is executed on the windowed elements, the weights or models computed in parallel workers must be globally reduced. The sink task receives such a globally reduced input. Within the sink task, the model evaluation is done using test data sets. Within the sink function, the application developer can decide when to release a stable model for production. This is not addressed in this research, but the applications have been developed in each framework to support it.

A. Motivation for Streaming Machine Learning

With access to a large amount of data, processing and responding with less latency is a significant challenge. Streaming versions of a known batch mode machine learning algorithm always enable the capability to run a model much faster rather than collecting all the data and processing them as a batch. But the main challenge is that all these streaming versions of machine learning algorithms provide a sub-optimal solution. An optimal solution would be the answer we obtain from extensive experiments on a data set with a large number of iterations on the complete batch of data. Most of the machine learning algorithms are in an iterative mode, as they are trying to optimize a set of parameters. In the streaming setting, the model is designed by accepting the nature of having a sub-optimal solution. This can be identified as one of the obstacles in obtaining a better solution with a streaming ML setting. In this research, we focus our work on the evaluation of the state-of-the-art stream processing engines on static conditions to see how each framework performs.

B. Streaming SVM

Support Vector Machine is one of the most prominent classification algorithm used in the machine learning domain. In an online version of this algorithm, we first discretize a stream of data points into a mini-batch or a window and do an iterative computation on each window. Here a variable number of iterations can be used in tuning the application towards expected accuracy in the training period. The core of the algorithm adopted is a stochastic gradient descent-based model. For each window, the weight vector is updated and synchronized to a global value by doing a model aggregation over the distributed setting. Once a model is globally synchronized over all the processes, it is then tested for accuracy. This implementation follows the principle of a batch model developed to evaluate batch-size based performance on SGD-SVM. We adopted the same approach to calculate the weight vector or gradient in the discretized stream (windowed elements) and globally synchronized the calculated weight vector once the computation per window was completed.

$$S = \{x_i, y_i\} \\ \text{where } i = [1, 2, 3, ..., n],\ x_i \in R^d,\ y_i \in [+1, -1] \tag{1}$$

$$\alpha \in (0, 1) \tag{2}$$

$$g(w; (x, y)) = \max(0, 1 - y\langle w|x\rangle) \tag{3}$$

$$J^t = \min_{w \in R^d} \frac{1}{2}\|w\|^2 + C \sum_{x,y \in S} g(w; (x, y)) \tag{4}$$

Equations 1,2,3 and 4 denote the configurations of the sample space, helper functions for gradient calculation and the loss function.

Algorithm 1 Iterative SGD SVM

```
1: INPUT: $[x, y] \in S, w \in R^d, t \in R^+$
2: OUTPUT: $w \in R^d$
3: procedure ISGDSVM($S, w, t$)
4:     for $i = 0$ to $n$ do
5:         if $(g(w; (x_i, y_i)) == 0)$ then
6:             $\nabla J^t = w$
7:         else
8:             $\nabla J^t = w - Cx_iy_i$
9:         $w = w - \alpha \nabla J^t$
       return $w$
```

In algorithm 1, the stochastic gradient descent-based step to update the weights is described as a pseudo-code. This algorithm shows the computation done per data point.

Algorithm 2 Iterative Streaming SVM

```
1: INPUT: $X_\infty, Y_\infty \in S_\infty, w \in R^d, l \in R^+, s \in R^+, m < K, m \in R^+$
2: OUTPUT: $w \in R^d$
3: procedure ISSVM($\bar{S}_i, w, T, l, s$)
4:     In Parallel K Machines $[\bar{S}_1, ..., \bar{S}_b] \subset S_\infty$
5:     procedure WINDOW($\bar{S}_m, w, l, s$)
6:         for $t = 0$ to $T$ do
7:             procedure ISGDSVM($\bar{S}_m, w, t$)
8:         All_Reduce(w)
       return $w$
```

Algorithm 2 shows the complete iterative algorithm with windowing configurations. The l symbol in the algorithm refers to the window length and the s symbol refers to the sliding length. The algorithm encapsulates both tumbling and sliding window-based computations.

C. Streaming KMeans

KMeans is another popular clustering algorithm in the machine learning domain. We apply an online version of this algorithm in our research. In the streaming setting, we use the stream discretization by means of a window operation. In Algorithm 3 we have implemented a basic version of the online-KMeans algorithm. V refers to the cluster centroids, k refers to the number of centroids, and n refers to the number of total data points observed down the stream. The number of data points observed must be at least equal to the number of cluster centroids. In this algorithm, a single data point is observed only once and the closest centroid is located by calculating the euclidean distance. The new centroid is calculated as shown in the algorithm. But in the initialization step, the centroids can be either handpicked from the data set or randomly selected. Here we select it as shown in the algorithm. Our objective is to see how each framework works on global model synchronization when working with machine learning models. Note that an iterative computation is not conducted. In implementing this algorithm we followed the state-of-the-art time notion-based window-less streaming KMeans implemented in Apache Spark. Once the computation related to a window finishes, a global model synchronization is performed. Unlike in a classification algorithm, there is no cross-validation involved during the model generation step.

Algorithm 3 Online KMeans

```
1: INPUT:$X = \{x_1, ..., x_m\}, x_i \in \mathbf{R^m}$
2: $V = \{v_1, ..., v_k\} v_i \in \mathbf{R^m}, k \leq n$
3: OUTPUT: $V$
4: procedure STREAMING-KMEANS($X, V$)
5:     procedure WINDOW($\bar{X}, \bar{V}$)
6:         for $x_j$ in $\bar{X}$ do
7:             if $j \leq k$ then
8:                 $v_i = x_j$
9:                 $k_i = 1$
10:                $i = i + 1$
11:            else
12:                $v_i = argmin_i ||x_j - v_i||$
13:                $v_i = v_i + \frac{1}{n_i + 1}[x_j - v_i]$
14:                $n_i = n_i + 1$
15:        All_Reduce(V)
       return V
```

IV. Experiments

For the experiments, we use a distributed cluster with 8 physical nodes. We schedule 16 tasks per node to run the experiments. Each node consists of Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz with 250GB of RAM capacity. For running an experiment in a finite period, a stream of 49,000 records for training and a stream of 90,000 records for testing are applied. For the experiments, we only test a finite stream to evaluate the training accuracy and performance. In a real world setting, training termination criteria have to be set by considering the objectives of the application. Windowing handles the stream discretization in all the systems. Windowing with the notion of the count of elements then tests the performance of each system. In every framework, the data was loaded from a file source and processing is performed in a distributed manner with overall parallelism of 128. All the experiments were carried out considering tumbling and sliding windowing. Tumbling windowing refers to windowing with non-overlapping elements, and sliding windows refers to windowing with overlapping elements. In the experiments, we adopted a count-based windowing mechanism to conduct a stress test on each framework. For the conducted experiments, the releases used were Apache Storm 1.2.8, Apache Flink 1.9.0 and Twister 0.3.0 (for the basic testing done on Apache Spark, version 2.4.4 was used). Each framework was tested under different internal configurations and we selected configurations that minimized any performance lag. The experiments were conducted for 10-20 rounds and the average results taken to draw our conclusions.

A. Model Synchronization

In the distributed setting, generating a synchronized model is vital. For implementing the online versions of the machine learning algorithms, we adopted strategies specific to each framework. In Apache Flink, the reduce function handles synchronization of the models. This is the only possible way to get an approximation to the all-reduce model as Apache Flink does not support an all-reduce-equivalent communication for synchronizing models globally. For Apache Spark, the reduce function and RDD broadcast performed the synchronization. With Apache Storm, all-grouping was used, while the Twister2-HPC model employed MPI-AllReduce collective communication. Twister2-Dataflow model uses a variation of all-reduce communication with a tree-like communication model. The model synchronization is thus carried out in Twister2.

B. Streaming SVM

For streaming SVM model, we used a dataset with two classes with 22 elements per data point. For the experiments, an iterative computation on windowed elements was deployed. This operation is supported by Apache Flink, Apache Storm and Twister2. We tried this model using Apache Spark streaming engine. With the provided APIs and system constraints, we were able to design an approximate model to that designed with the aforementioned frameworks. The main constraint is that it only provides windowing considering the notion of time. This makes it hard to do a stress test on the stream engine because, by the notion of time, the minimum number of elements that can be set per batch is in millisecond level. Furthermore, it does not support iterative streaming models. This feature is not directly supported with DStream in Apache Spark streaming engine. With the approximate model, the accuracy obtained was comparatively quite low in regard to the other frameworks. A workaround is to use structured streaming in Apache Spark. This implementation works on the SQL engine of Spark, and it only considers the notion of time. We did not implement that model in this research as it is a very different implementation compared to the others. In the conclusion section, this will be explained in detail. Figure 2 shows the experiment results for tumbling window. From these results, it is clear that the Twister2 models outperform both Apache Storm and Apache Flink implementations. Figure 3 illustrates the sliding window related experiments. Similar to tumbling windowing, with sliding windows, Twister2 implementations outperform Apache Flink and Apache Storm implementations. Twister2 possesses a faster stream processing capability through a strong MPI-based backend. This provides a scalable solution for iterative stream processing on a window. With Apache Flink, the main bottleneck is the reduce task doing the model synchronization. In Twister2 and Apache Storm, the all-reduce and all-grouping mechanisms are involved in providing all-to-all model synchronization capability. But in Apache Flink, this process becomes all-to-one and makes a bottleneck in processing the data. In this case, both Twister2 and Apache Storm outperform Apache Flink.

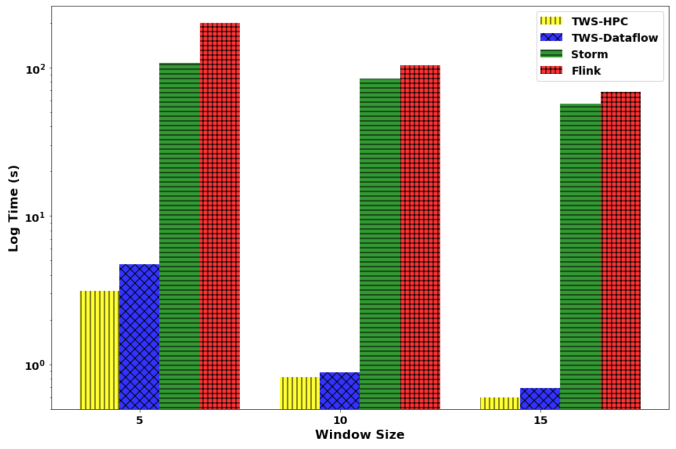

Fig. 2. Streaming SVM with Linear Kernel-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

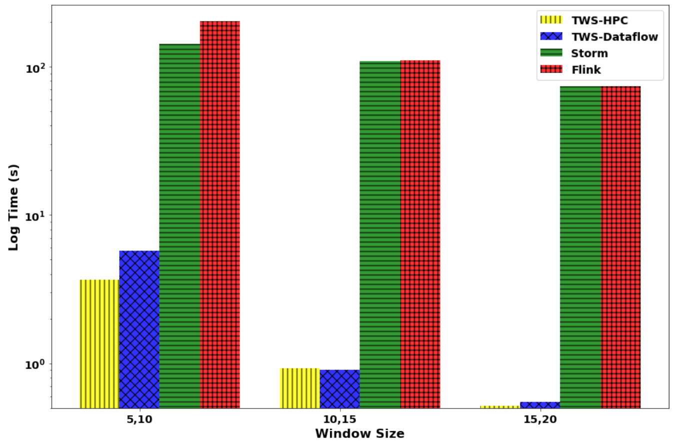

Fig. 3. Streaming SVM with Linear Kernel-based experiments for sliding window is recorded for HPC model and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x-axis in the figure is labeled with the pair of (window length, sliding length).

From all implementations in Apache Flink, Apache Storm and Twister2, 90.49% of test accuracy was obtained after a finite length of the stream was processed. With Apache Spark implementation, we were able to get an average accuracy of 40%-50% with the same number of iterations. We did not include the graphs here since the number of iterations required to get the same accuracy is much higher. The main issue for this is that Spark streaming API is not designed with iteration compatibility. Also, it does not provide a window function to capture the elements belonging to a window. This functionality is available in Apache Storm, Apache Flink and Twister2. Apache Spark only provides basic element operators like map, flatmap, etc. If this was attempted with a forEachRDD function, the user has no capability to synchronize the model as it is a sink function. In addition, Spark only provides a windowing functionality with the notion of time and has no support for windowing based on the count of elements.

C. Streaming KMeans

For the streaming KMeans model, the dataset we used contains 23 elements per data point. Here a non-iterative computation is done. Apache Flink, Apache Storm and Twister2 support the windowing functions to implement an algorithm like this. With Apache Spark streaming, a non-iterative application can be developed, but the count-based notion is not available in the API. In this research, we have only conducted windowed streaming with the notion of the number of elements per window. In achieving the current goal, we used the streaming systems which provide this functionality. Figure 4 shows the tumbling window-based experiments carried out on streaming KMeans model. Figure 3 shows the sliding window-based experiments carried out on streaming KMeans model. Similar to streaming SVM results, Twister2 models outperform both Apache Spark and Apache Flink. Twister2 model synchronization with an all-reduce mechanism provides faster execution than that of regular all-to-all communication in Apache Storm. In Apache Flink, there is no all-to-all communication; the model synchronization happens in an all-to-one setting. This is the same bottleneck as observed in streaming SVM applications. But Apache Flink outperforms Apache Storm. This model is a non-iterative model and the pressure exerted on communication is lower. This leads to much faster data progress from the windowing task to the reduce task.

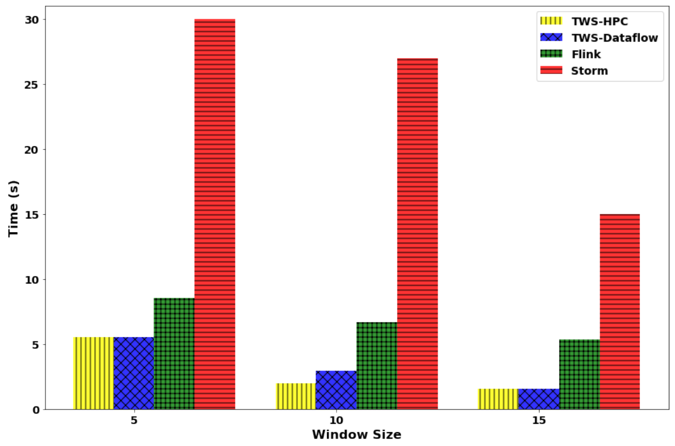

Fig. 4. Streaming KMeans Results for 1000 cluster-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

V. Related Work

Apache Spark [2] considers stream processing as a related event of small-batch computations. It collects the records from the stream in a buffer which is called mini-batch. The main advantage of this technique is to provide effortless fault tolerance. However, a disadvantage is higher latency due to the micro-batch scheduling mechanism. Apache Flink [3] processes the streaming events using the dataflow runtime model rather than processing as micro-batches, which provides lower processing latency. However, a significant disadvantage

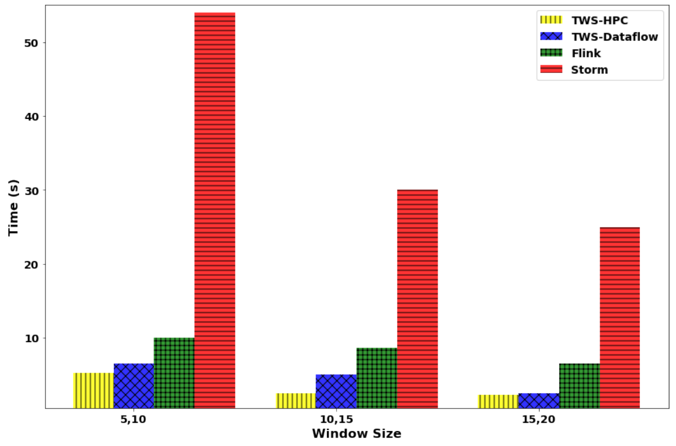

Fig. 5. Streaming KMeans for 1000 cluster-based experiments for sliding window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x axis in the figure is labeled with the pair of (window length,sliding length).

of this model emerges when implementing the fault tolerance mechanism. Apache Storm [4] is a real-time distributed stream processing engine which provides a fault-tolerant and scalable system to process the streaming data. It is implemented with two important processing semantics, namely ”at least once” and ”at most once”, that provide the guarantee of the data which processes it. Twister: Net [5] is a standalone highly optimized dataflow library that defines the dataflow model for big data to process streaming and batch data. Based on the evaluation, it is acknowledged that the communication requirements of big data have been written in a separate library without the integration of any big data framework. Using this library, the user may design highly efficient big data applications. TSet [6] is the highest level of abstraction provided in Twister2 [7] framework which is similar to RDD’s in Apache Spark and DataSets in Apache Flink. S4 (Simple Scalable Streaming System) [8] is a distributed model for processing streaming which has been designed to solve the data mining and machine learning algorithms. It is designed with a simple programming interface along with decentralized and symmetric architecture in which nodes share the same functionalities and responsibilities and there is no overhead to a single node. They have demonstrated the performance of tuning an online search advertising system. Qian et al [9] designed a distributed system known as TimeStream which specifically processes continuous big stream data with low latency. It has provided a powerful abstraction called resilient substitution which is responsible for handling the failure recovery and dynamic reconfiguration corresponding to the load. TimeStream is implemented with a fine-grained data dependency mechanism to enable a re-computation-based failure recovery mechanism that achieves ”at least once” semantics. Derek G. Murray et al [10] designed a timely dataflow system that executes the data-parallel and cyclic dataflow program in a distributed manner. It achieves high throughput batch processing and low latency stream processing using the Timely

Dataflow model. It also enhances the dataflow computation and provides the base for an efficient, lightweight coordination mechanism. Online classification on large scale data sets has also been discussed by Street et al [11] in the early stage of the streaming machine learning research. Hazan et al [12] describe two ways of designing an online SGD algorithm: an adaptive algorithm with a better convergence rate and a standard online algorithm with a descent convergence rate. Zhong et al [13] propose an online version of Kmeans clustering by observing a data point once in the model generation step and assigning it to the closest centroid. Yahoo [14] provides a state-of-the-art stream processing-related benchmark showing the capabilities in each stream processing engine. It uses Apache Storm, Apache Spark and Apache Flink as the streaming engines to draw the comparisons. Krimov et al [14] provide another benchmark on analyzing the capabilities in Apache Storm, Apache Flink and Apache Spark.

VI. Conclusion

Twister2 streaming engine provides state-of-the-art performance for streaming machine learning algorithms. Processing large amounts of data with low latency is critical to streaming frameworks. This paper presented two important machine learning algorithms and showcased their performance. Twister2 outperformed Apache Storm and Apache Flink in all the scenarios considered for both algorithms. With Apache Spark streaming engine, we were only able to design a streaming model based on time. This was not the area of focus in our research. A time-based windowing makes it much harder to run a stress test on the streaming engine. In addition, we observed the importance of windowing functions availability in Apache Storm, Flink and Twister2 for implementing advanced algorithms in the streaming setting. In Apache Spark streaming, we were not able to use the notion of a windowing function. When it comes to doing an iterative computation, the notion of a window-function is highly essential, although Apache Spark provides a solution for this based on the SQL engine. The structured streaming with Apache Spark SQL provides a possible avenue to develop such applications. This involves channeling the capabilities in the SQL engine which can add an overhead to the application. In this research, we paid more attention to the very basic components in a stream engine itself and evaluated the performance for different experiment settings.

VII. Future Work

For future work, we are expecting to design time-notion based experiments. The idea is to analyze event-time and process-time-based stream discretization on state-of-the-art stream processing engines.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054, CINES 1835631 and the Indiana University Precision Health Initiative. We thank Intel for use of the Juliet and Victor systems and extend our gratitude to the FutureSystems team for their support with the infrastructure.

References

[1] "Apache kafka," https://kafka.apache.org/, (Accessed on 10/31/2019).

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] "Apache storm," https://storm.apache.org/, (Accessed on 09/21/2019).

[5] S. Kamburugamuve, P. Wickramasinghe, K. Govindarajan, A. Uyar, G. Gunduz, V. Abeykoon, and G. Fox, "Twister: Net-communication library for big data processing in hpc and cloud environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 383–391.

[6] P. Wickramasinghe, S. Kamburugamuve, K. Govindarajan, V. Abeykoon, C. Widanage, N. Perera, A. Uyar, G. Gunduz, S. Akkas, and G. Fox, "Twister2: Tset high-performance iterative dataflow," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE, 2019, pp. 55–60.

[7] S. Kamburugamuve, K. Govindarajan, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Twister2: Design of a big data toolkit," *Concurrency and Computation: Practice and Experience*, p. e5189, 2017.

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE International Conference on Data Mining Workshops*, Dec 2010, pp. 170–177.

[9] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465353

[10] 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*. ACM, 2013, pp. 439–455.

[11] W. N. Street and Y. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 377–382.

[12] E. Hazan, A. Rakhlin, and P. L. Bartlett, "Adaptive online gradient descent," in *Advances in Neural Information Processing Systems*, 2008, pp. 65–72.

[13] S. Zhong, "Efficient online spherical k-means clustering," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 5. IEEE, 2005, pp. 3180–3185.

[14] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016, pp. 1789–1792.