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*, Gregor Von Laszewski*† and Geoffrey Fox*†,
*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,ggnuduz,gvonlasz,gcf}@iu.edu

***Abstract*—Increasing data volume and designing low latency applications with higher efficiency is a very challenging problem. With the limited storage and limited time to process data, usage of online algorithms has become an emerging trend in the big-data community. In the big-data analytics, stream processing is a well-known area that has been studied for a long time. In this research, our objective is to use state of the art big-data analytic engines to design such online algorithms and compare the strengths and weaknesses in each system. We use a streaming version of Support Vector Machines and KMeans to do the analysis. In our research, we use Apache Flink, Apache Storm and Twister2 to implement the streaming algorithms. Our study focuses on the efficiency of online training by analyzing the inherent features in each stream processing engine. And our experiments show that Twister2 is a streaming engine which can support developing streaming machine learning algorithms with less latency compared to other streaming engines discussed in this paper.**

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

I. Introduction

In the modern information technology era data collection and data processing is now ubiquitous, this has resulted in an explosion of the amount of data that is collected. Which in-turn means all this collected data needs to be processed to gain meaningful insight hidden within the data. Data sources may range from data collected from social media platforms to more basic 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 because of the amount of data that is collected and the speed at which such data is collected. In recent years, stream processing has become one of the most prominent modes of processing large volumes of data with less latency. And these stream processing engines are capable of much more than simple filters. Among the applications which produce a large volume of data, internet-of-things related applications, social media data processing applications, video processing applications, audio processing applications, etc can be denoted as 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, 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 two folds, First is to analyse the application development styles in each stream processing system to identify subtle differences in the different programming models adopted by popular frameworks. Second, to showcase the performance of each system using streaming machine learning applications.
We believe that applying knowledge ganined from the HPC domain into big-data systems allows frameworks to harness the best of both worlds, this is showcased to some extent in the experiments and results that are shown in this paper comparing Twiste2, Apache Flink and Apache Storm. In section II we discuss the role of stream processing in the big-data domain and its importance. Section III talks about several streaming machine learning algorithms in more detail and the motivations behind streaming machine learning algorithms in general.In section IV the experiments conducted to compare various streaming engines are discussed and the results are presented, to this end two algorithms, namely streaming KMeans and streaming SVM are used. Section V discusses related work that can be found in the literature. Finally in sections VI and VII layout the conclusion and future work for this research.

II. Stream Processing with Big Data Stack

Stream processing is majorly involved with big-data related applications rather than traditional 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 streaming applications. Twister2-Streaming is another framework developed by the authors for the same purpose, to apply knowledge gained from the HPC community to provide better performance numbers for stream processing. In our framework, we also provide most of the core functionality provided by these state of the art stream processing systems. The programming model in Apache Storm

is very flexible than most of the other stream processing frameworks due to the lower-level abstraction in the API. This was one of the core features in Apache Storm until the release 2.0.0. But 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. This 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. In Twister2, adopting both programming styles, like Apache Storm, we provide both levels of programming abstractions.

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.

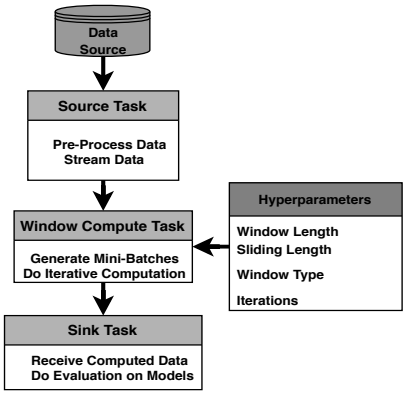

Fig. 1. Workflow of a Streaming ML Application

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 our experiments in all three frameworks, a source task read data from the file system and do the data pre-processing. This source task creates a stream of data. Down the stream, a window task is being designed with some hyper-parameters. The main hyper-parameters involved for a streaming machine learning algorithm includes the number of iterations. I case KMeans this value was unity and in SVM this is a finite value set by the user. Window length, sliding window length a implicit parameters for windowing. Windowing type can be determined as count-based or time-based. In the experiments we conducted, we only used the notion of count. Within the window task, a mini-batch is being generated from stream discretization. This mini-batch is being processed by the streaming model of the machine learning algorithm. Here each algorithm provides a sub-optimial solution with respect 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 the models that is being computed in a distributed manner must be globally synchronized. The sink task gets such a globally reduced input. Within the sink task, the model evaluation is being done using test data sets. Within the sink function, the application developer can decide when to releasing a stable model for production. This is not a section discussed in this research, but the applications have been developed in each framework to support this.

A. Motivation for Streaming Machine Learning

With access to a large amount of data, processing and responding with less latency is one of the challenges. Streaming version of a known batch mode machine learning algorithm always enables 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 being designed by accepting the nature of having a sub-optimal solution. This can be identified as one of the challenges in obtaining a better solution with 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 is performing.

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 it is being synchronized to a global value by doing a model aggregation over the distributed setting. Once a model is being globally synchronized over all the processes, the model is being tested for accuracy. This implementation was followed by an idea related to a batch model developed to evaluate batch-size based performance on SGD-SVM. We adopted the same approach to calculate the weight vector or the gradient in the discretized stream (windowed elements) and globally synchronized the calculated weight vector once the computation per window is completed.

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

$$\alpha \in (0, 1) \quad (2)$$
$$g(w; (x, y)) = \max(0, 1 - y\langle w|x\rangle) \quad (3)$$

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

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

Algorithm 1 Iterative SGD SVM

INPUT: $[x, y] \in S, w \in R^d, t \in R^+$
OUTPUT: $w \in R^d$
procedure ISGDSVM(S, w, t)
  for $i = 0$ to n **do**
    if $(g(w; (x, y)) == 0)$ **then**
      $\nabla J^t = w$
    else
      $\nabla J^t = w - Cx_i y_i$
    $w = w - \alpha \nabla J^t$
  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 a data point.

Algorithm 2 Iterative Streaming SVM

INPUT: $X_\infty, Y_\infty \in S_\infty, w \in R^d, l \in R^+, s \in R^+, m < K, m \in R^+$
OUTPUT: $w \in R^d$
procedure ISSVM($\bar{S}_i, w, T, l, s$)
  In Parallel K Machines $[\bar{S}_1, ..., \bar{S}_b] \subset S_\infty$
  procedure WINDOW($\bar{S}_m, w, l, s$)
    for $t = 0$ to T **do**
      procedure ISGDSVM($\bar{S}_m, w, t$)
    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 s symbol in the algorithm refers to the sliding length. The algorithm encapsulates both tumbling and sliding window based computations.

C. Streaming KMeans

KMeans is another highly used clustering algorithm in the machine learning domain. In this research, we use an online version of this algorithm. In the streaming setting, we use the stream discretization by using a window operation. In the algorithm 3 we have implemented a basic version of online-KMeans algorithm. In this algorithm one data point is observed just once and the closest centroid is located by means of calculating the euclidean distance. And the new centroid is calculated as shown in the algorithm. But in the initialization step the centroids can be either hand picked from the data set or randomly selected. In 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.
But 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 is being completed, a global model synchronization is done. Unlike in a classification algorithm, there is no cross-validation involved during the model generation step.

Algorithm 3 Online KMeans

INPUT: $X = \{x_1, ..., x_m\}, x_i \in \mathbf{R^m}$
$V = \{v_1, ..., v_k\} v_i \in \mathbf{R^m}, k \leq n$
OUTPUT: V
procedure STREAMING-KMEANS(X, V)
  procedure WINDOW($\bar{X}, \bar{V}$)
    for $x_j in \bar{X}$ **do**
      if $j \leq k$ **then**
        $v_i = x_j$
        $k_i = 1$
        $i = i + 1$
      else
        $v_i = argmin_i ||x_j - v_i$
        $v_i = v_i + \frac{1}{n_i + 1}[x_j - v_i]$
        $n_i = n_i + 1$
    All_Reduce(V)
  return V

IV. EXPERIMENTS

For the experiments, we use a distributed cluster with 8 physical nodes. We schedule 16 tasks per each 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 for a finite period, a stream of 49,000 records for training and a stream of 90,000 records for testing is used. For the experiments, we only use a finite stream to evaluate training accuracy and performance. In a real-world setting, application training termination criteria have to be set by considering the objectives of the application use-case. To discretize the stream, we use windowing in all frameworks mentioned in this paper. The windowing with the notion of the count of elements is being used to test the performance in each system. In each framework, the data is being loaded from a file source and processing is done in a distributed manner with overall parallelism of 128. All the experiments were carried out considering tumbling and sliding windowing. Tumbling windowing refers to non-overlapping elements and sliding

windows refers to windowing with overlapping elements. In the experiments, we are adopting a count-based windowing mechanism to conduct a stress test on each framework. For the conducted experiments, Apache Storm 1.2.8, Apache Flink 1.9.0 and Twister 0.3.0 releases have been used. Each framework was tested under different internal configurations and we selected configurations that minimize any performance lag. The experiments have been conducted for 10-20 rounds and average results have been taken to design the plots.

A. Model Synchronization

In the distributed setting, generating a synchronized model for all parallel processes is vital. In implementing the online versions of algorithms, we adopted the strategies specific for each framework. In Apache Flink, the reduce function is used for synchronizing the models. This is the only possible way to do a similar operation to all-reduce in Apache Flink. Apache Flink doesn't support an all-reduce like communication for synchronizing models globally. In Apache Spark, reduce function and RDD broadcast is used to synchronize the model. In Apache Storm, all-grouping is used to generate a synchronized model. Twister2-HPC model uses MPI-AllReduce collective communication to synchronize the models. Twister2-Dataflow model uses a variation of all-reduce communication with a tree-like communication model. The global model synchronization is thus carried out in Twister2.

B. Streaming SVM

For streaming SVM model, we use a dataset with two classes with 22 elements per data point. For the experiments, we used an iterative computation on windowed elements which 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 of design 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 steam engine. Because by time, the minimum number of elements that can be set per batch is in millisecond level. With the approximate model, the accuracy obtained was comparatively very low concerning the other frameworks. The major reasoning was we couldn't implement an iterative computation on windowed data points. This feature is not directly supported with DStream in Apache Spark streaming engine. But there is a workaround for this using structured streaming in Apache Spark streaming. This implementation works on the SQL engine of Spark, and it considers the notion of time. We didn't implement that model in this research as it is a very different implementation concerning the other implementations. In the conclusion section, this will be explained in detail. Figure 2 shows the experiment results for tumbling window is shown. From these results, it is clear that the Twister2 models outperform both Apache Storm and Apache Flink implementations. Figure 3 shows the sliding window related experiments. Twister2 implementations outperform Apache Flink and Apache Storm implementations. Twister2 with a faster stream processing capability through a strong MPI-based backend provides a scalable solution for an 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 involve 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 outperforms Apache Flink.

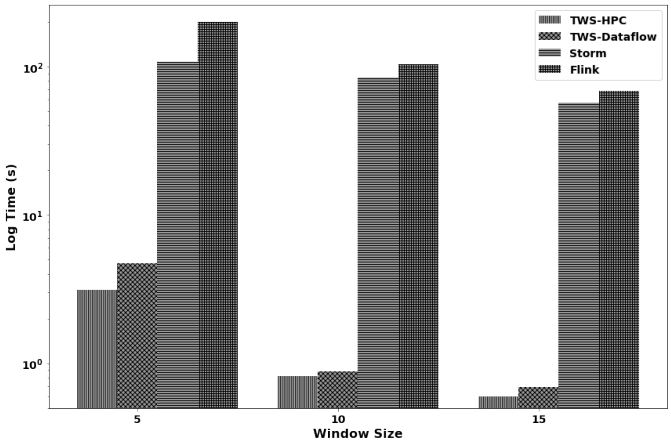

Fig. 2. Streaming SVM with Linear Kernel based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

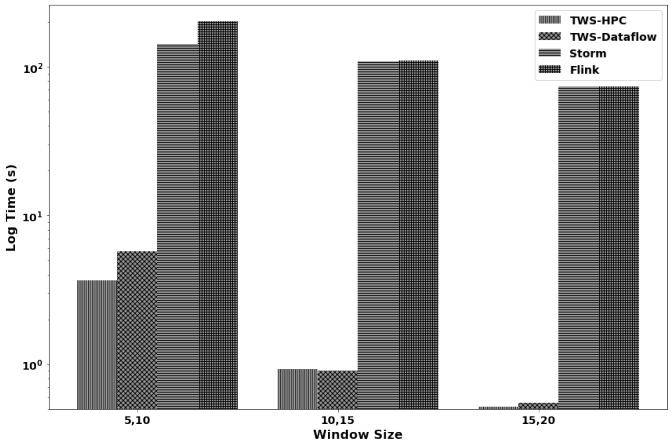

Fig. 3. Streaming SVM with Linear Kernel based experiments for sliding window is recorded for HPC model and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x axis in right figure labelled with the pair of (window length,sliding length).

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 Storm implementation, we were able to get an average accuracy of 40%-50% with the same number of iterations. We didn't include the graphs here, because the number of iterations required to get the same accuracy is much higher.

C. Streaming KMeans

For streaming KMeans model, the dataset we used contains 23 elements per a 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 have used the streaming systems which provide this functionality.

Figure 4 shows the tumbling window-based experiments carried out on streaming KMeans model. And in 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 application. But Apache Flink outperforms Apache Storm. This model is a non-iterative model and the pressure exerted on communication is lesser. This leads to quite faster data progress from the windowing task to reduce task. Experiments were conducted to formulate 1000 cluster centers.

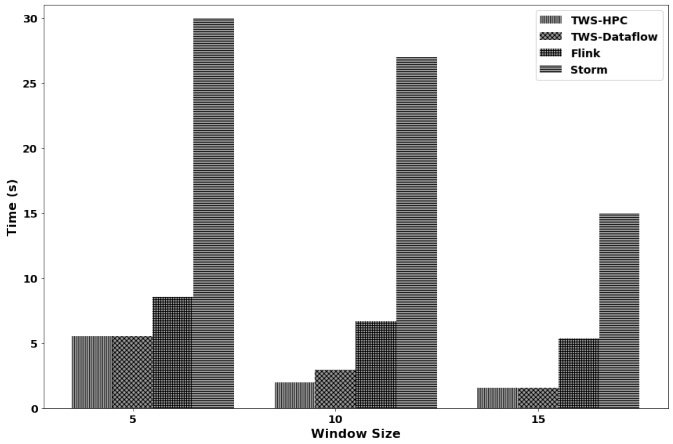

Fig. 4. Streaming KMeans Results for 1000 clusters-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

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 to provide effortless fault tolerance. However, the main disadvantage of this technique is higher latency due to the micro-batch scheduling mechanism. Apache Flink [3] processes the streaming events using the

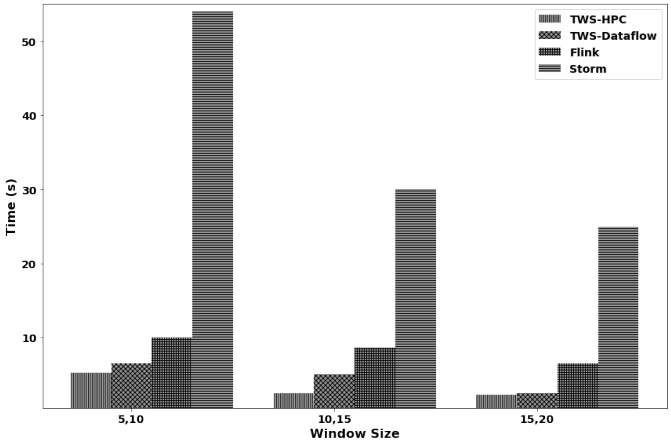

Fig. 5. Streaming KMeans for 1000 clusters-based experiments for sliding window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x axis in right figure labelled with the pair of (window length,sliding length).

dataflow runtime model other than processing as micro-batches which provides lower processing latency. However, the main disadvantage of this model is 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 be able to design a 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 the results for tuning an online search advertising system. Qian et.al [9] designed a distributed system known as TimeStream which is specific to process low latency and continuous big stream data. It has provided a powerful abstraction called resilient substitution which is responsible for handling the failure recovery and dynamic reconfiguration corresponding to the load. It 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 dis-

tributed 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, light-weight coordination mechanism. Online classfication on large scale data sets has been also 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 standard online algorithm with a descent convergence rate. Zhong et.al [13] propose an online version of K-means 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 used to draw the comparisons. Krimov et.al [14] provides another benchmark on analyzing the capabilities in Apache Storm, Apache Flink and Apache Spark.

VI. Conclusion

Twister2 streaming engine provides a state of the art streaming performance for streaming machine learning algorithms. The idea of processing a high throughput data with less time is one of the key features expected in a stream processing system. Here we showcase how this is evident with two of the major machine learning algorithms. In both algorithms with tumbling and sliding window settings, both Twister2 models outperform Apache Storm and Apache Flink. With Apache Spark streaming engine, we were only able to design a streaming model based on time notion. 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. Besides, we observed the importance of windowing functions available 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 do an iterative computation the notion of a window-function is highly essential. All though 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 channelling the capabilities in the SQL engine and this 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 evaluate the performance for different experiment settings.

VII. Future Work

As 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.

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.