A Predictive Scheduling Framework for Fast and Distributed Stream Data Processing

Teng Li, Jian Tang and Jielong Xu

***Abstract*—In a distributed stream data processing system, an application is usually modeled using a directed graph, in which each vertex corresponds to a data source or a processing unit, and edges indicate data flow. In this paper, we propose a novel predictive scheduling framework to enable fast and distributed stream data processing, which features topology-aware performance prediction and predictive scheduling. For prediction, we present a topology-aware method to accurately predict the average tuple processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics. For scheduling, we present an effective algorithm to assign threads to machines under the guidance of prediction results. To validate and evaluate the proposed framework, we implemented it based on a highly-regarded distributed stream data processing platform, Storm, and tested it with two representative applications: word count (stream version) and log stream processing. Extensive experimental results show 1) The topology-aware prediction method offers an average accuracy of 83.7%. 2) The predictive scheduling framework reduces the average tuple processing time by 25.9% on average, compared to Storm's default scheduler.**

I. Introduction

In a distributed stream data processing system, an application is usually modeled using a directed graph, in which each vertex corresponds to a data source or a Processing Unit (PU), and edges indicate data flow. Such a system handles unbounded streams of data tuples, which may last for a long time, therefore, the tuple processing time, i.e., the time between when the tuple comes out of the data source and the time when its processing is completed and acknowledged, is usually used as the primary performance metric. A tuple may traverse multiple PUs and experience various latencies (such as processing latency at a PU and transfer latency between PUs) during its lifetime. How to accurately model and predict tuple processing time in a distributed stream data processing system is quite challenging and has not yet been well studied. Queueing theory has been applied for modeling and prediction in distributed database systems [4]. However, it does not work for general-purpose stream data processing due to the following reasons: 1) Queueing theory can only offer good estimation for queueing delay under a few assumptions (e.g, tuple arrivals follow a Poisson distribution, etc), which, may not hold in a complex distributed data processing system. 2) In queueing theory, many problems in a queueing network (instead of a single queue) remain unknown, while a distributed data processing system represents a fairly complicated multi-point to multi-point queueing network where tuples from a queue may be distributed to multiple downstream queues, and a queue may receive tuples from multiple different upstream queues.

In this paper, we propose a novel predictive scheduling framework to enable fast and distributed stream data processing, which features topology-aware performance prediction and predictive scheduling. Moreover, we validate and evaluate the proposed framework based on an emerging platform, Apache Storm [2], which has been widely used in many companies and organizations, such as Twitter, Groupon, Yelp, etc. We summarize our contributions in the following:

- For prediction, we present a topology-aware method to accurately model and predict the average tuple processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics.
- For scheduling, we present an effective algorithm to assign tasks (threads) to machines under the guidance of the prediction results.
- We implemented the proposed framework based on Storm, and tested it with two representative applications: word count (stream version) and log stream processing. It has potential to be applied to other similar platforms, such as Yahoo!'s S4 [3] Google's MillWheel [5] and Microsoft's TimeStream [6].
- Extensive experimental results well justify accuracy of the proposed prediction method and effectiveness of the proposed scheduling algorithm.

To the best of our knowledge, we are the first to propose a method for modeling and predicting the average tuple processing time in a distributed stream data processing system and validate the proposed method with the widely-used open-source platform, Storm.

II. Design and Implementation of the Proposed Framework

A. Overview

As illustrated in Fig. 1, the proposed framework consists of several modules, including data collector, data store, time synchronizer, data pre-processor, performance predictor, schedule generator, and custom scheduler. The data collector runs as multiple threads to collect runtime statistics from all processes/machines in the cluster, which are then stored into the distributed data store. The data pre-processor prepares input data for prediction and scheduling based on raw data in the data store. The time synchronizer provides necessary synchronization for threads of the data collector.

The most important modules of the proposed framework include performance predictor, schedule generator and the

Teng Li, Jian Tang and Jielong Xu are with the Department of Electrical Engineering and Computer Science at Syracuse University, Email: {tli01, jtang02, jxu21}@syr.edu. This research was supported in part by NSF grants #1218203 and #1443966.

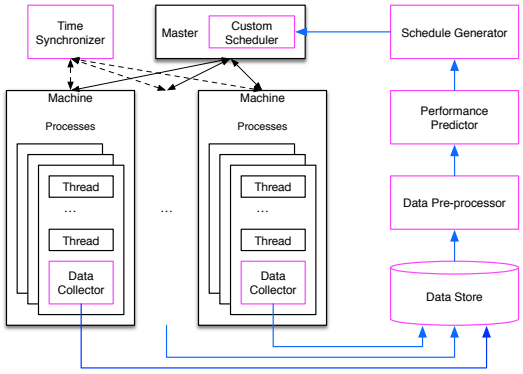


Fig. 1. The proposed predictive scheduling framework

custom scheduler. The performance predictor employs the proposed prediction method to predict the average tuple processing time of an application, which is introduced in Sections II-B. The schedule generator uses the proposed scheduling algorithm (Section II-C) to generate a scheduling solution, which is then deployed to the cluster by the custom scheduler.

B. Topology-aware Performance Prediction

We aim to model and predict the average tuple processing time of an application for a given scheduling solution, which can serve as a guideline to make a wise decision on task scheduling. Our idea is to predict the average tuple processing latency at each PU and the average tuple transfer latency between PUs (including both queueing and communication latencies) and add them up in a certain way according to the topology of the application graph. First, we show how to model the average tuple processing time of a typical application, whose topology is a chain. Then we extend it to graphs with general topologies.

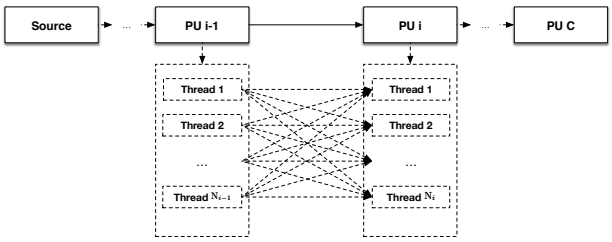


Fig. 2. A chain topology

First, we focus on a typical application graph with a chain topology as shown in Fig. 2. There are a total of $C+1$ components in the topology with C PUs and a data source. Particularly, $i=0$ corresponds to the data source. At runtime, the ith component ($i \in \{0,\cdots,C\}$) runs as N_i threads. Consider latency involving PU i ($i \in \{1,\cdots,C\}$), denoted as t_i^{PU}: after a tuple is processed by a thread of component (source or a PU) $(i-1)$, it is inserted to an outgoing queue to a thread of PU i. After certain waiting time, it is passed to a thread of PU i, which is then inserted to an incoming queue in that thread. Again, after certain waiting time, it is processed by the thread of PU i. This latency can be obtained as follows:

$$t_i^{\text{PU}} = t_i^{\text{Proc}} + t_i^{\text{Tran}}, \tag{1}$$

where t_i^{Proc} and t_i^{Tran} are the average tuple processing latency at PU i and the average tuple transfer latency over edge $(i-1,i)$ respectively, which can be estimated using:

$$t_i^{\text{Proc}} = \sum_{j=1}^{N_i} w(i,j) * t_{ij}^{\text{Proc}}. \tag{2}$$

$$t_i^{\text{Tran}} = \sum_{j=1}^{N_{i-1}} \sum_{k=1}^{N_i} w(i,j,k) * t_{ijk}^{\text{Tran}}. \tag{3}$$

Here, t_{ij}^{Proc} is the average tuple processing latency of jth thread at PU i, which can be predicted using a supervised learning algorithm introduced later. $w(i,j)$ is the weight corresponding to that thread. In our implementation, it was set to the ratio of the number of tuples received at jth thread of PU i to the total number of tuples received by all threads of PU i within the past 10 minutes, which can be obtained using runtime statistics collected by the data collector. t_{ijk}^{Tran} is the average tuple transfer latency between the jth thread of PU $i-1$ and the kth thread of PU i, which can also be predicted using using a supervised learning algorithm introduced later. $w(i,j,k)$ is the corresponding weight, which was set to the ratio between the number of tuples sent from the jth thread of PU $i-1$ to the kth thread of PU i and the total number of tuples sent from PU $i-1$ to i within the past 10 minutes in our implementation. Note that our framework is not restricted to a particular weight function. The other weight functions may also be used here.

Let t_p denote the average tuple processing time of the chain topology, we have:

$$t_{\text{p}} = \sum_{i=1}^{C} t_i^{\text{PU}}. \tag{4}$$

Modeling and predicting the average tuple processing time of an application graph with a general topology are harder. However, an application graph is usually not very complicated so all paths from data sources can be easily enumerated. How to estimate average tuple processing time over *a path (i.e., a chain)* has been described above. Suppose that there are a total of P paths from a data source to the acker and the average tuple processing time of the pth path is t_p, then we have:

$$t_G = \max_{p\in\{1,\cdots,P\}} t_p. \tag{5}$$

In order to use the above model for prediction, we need to predict the average tuple processing latency at a thread, t_{ij}^{Proc}, and the average tuple transfer latency between two threads, t_{ijk}^{Tran}. We propose to employ a supervised learning algorithm to predict the two types of latencies. Intuitively, they can be affected by many factors. We list the set of features selected for predicting the average tuple processing latency of

a thread, t_{ij}^{Proc}, in the following: 1) Machine-CPU: The number of CPU cores of the machine at which the target thread is located. This could be the total capacity of these CPU cores if machines have different CPUs. 2) Machine-Memory: The size of memory (GB) of the machine at which the target thread is located. 3) Thread-Workload: The number of tuples that the target thread processes within a window (30 seconds in the current implementation). 4) Machine-Workload: The number of tuples that the machine (at which the target thread is located) processes within a window (30 seconds in the current implementation). 5) Co-located-Threads: The total number of threads located at the same machine as the target thread. Note that machines could be physical machines or Virtual Machines (VMs).

Predicting the average tuple transfer latency is more difficult since t_{jk}^{Tran} involves threads on both ends. We summarize our feature selection in the following: 1) Machine-CPU-U: The number of CPU cores of the machine at which the upstream thread is located. 2) Machine-Memory-U: The size of memory (GB) of the machine at which the upstream thread is located. 3) Thread-Workload-U: The number of tuples that the upstream thread processes within a window (30 seconds in the current implementation). 4) Machine-Workload-U: The number of tuples that the machine (at which the upstream thread is located) processes during a window (30 seconds in the current implementation). 5) Machine-CPU-D: The number of CPU cores of the machine at which the downstream thread is located. 6) Machine-Memory-D: The size of memory (GB) of the machine at which the downstream thread is located. 7) Thread-Workload-D: The number of tuples that the downstream thread processes within a window (30 seconds in the current implementation). 8) Machine-Workload-D: The number of tuples that the machine (at which the downstream thread is located) processes during a window (30 seconds in the current implementation). 9) Co-location (boolean): if it is 1, the upstream thread and the downstream thread are located at the same machine; 0, otherwise.

Since latency can be any real value, we need to use a regression algorithm to make prediction. This problem has high-dimensional input data with both continuous and categorical attributes, which makes it inappropriate to apply parametric regression algorithms such as linear and non-linear regression. The non-parametric regression algorithms, Support Vector Regression (SVR), is a good candidate for this problem because SVR scales relatively well to high dimensional data and the trade-off between classifier complexity and error can be controlled explicitly [7]. We used cross-validation for kernel function selection and the results show that radial basis kernel function outperforms the linear kernel, the polynomial kernel and the sigmoid kernel.

C. Predictive Scheduling

The prediction serves as a guideline for a schedule algorithm to assign threads to machines. Next, we discuss the scheduling problem and present the proposed algorithm.

Given the set of machines $\mathcal{M}$, the set of processes $\mathcal{Q}$, and the set of threads $\mathcal{N}$, the scheduling problem is to assign each thread to a process of a machine, that is, to find two mappings: $\mathcal{N} \mapsto \mathcal{Q}$ and $\mathcal{Q} \mapsto \mathcal{M}$. Our design ensures that on every machine, threads from the same application are assigned to no more than one process. This is enforced because assigning threads from one application to multiple (instead of one) processes on a machine introduces inter-process traffic and may lead to serious performance degradation, which has been shown in [8]. Therefore the two mappings can be merged to just one mapping: $\mathcal{N} \mapsto \mathcal{M}$, that is, to assign each thread to a machine.

The objective of the scheduling problem is to minimize the average tuple processing time over an application graph G. Let the scheduling solution be $X = \langle x_{ij} \rangle, i \in \{1, ..., N\}, j \in \{1, ..., M\}$, where $x_{ij} = 1$ if thread i is assigned to machine j. Note that different scheduling solutions can lead to different tuple processing and transfer latencies thus different tuple processing time.

The proposed algorithm computes a scheduling solution based on the average tuple processing time predicted by the method described above, which is denoted as $\text{Pred}(\cdot)$ in the following. We formally present the proposed predictive scheduling algorithm as Algorithm 1. The algorithm is essentially a greedy algorithm, which starts from an initial solution and keeps improving it by adjusting the thread-machine scheduling solution according to the prediction results. Then it tries to re-assign threads so as to achieve less average tuple processing time. The algorithm re-assigns one thread at a time in a greedy manner by picking the machine that minimizes $\text{Pred}(t_G(X))$.

Algorithm 1 Predictive Scheduling Algorithm

Input: $\mathcal{M}$, $\mathcal{N}$
Output: $X = \langle x_{ij} \rangle$, $i \in \{1, ..., N\}$, $j \in \{1, ..., M\}$

```
1:  X := X_0, where X_0 can be any feasible solution;
2:  for i = 1 to N do
3:    for j = 1 to M do
4:      X' := X;
5:      if x_ij = 0 then
6:        x'_ij := 1;
7:        x'_ik := 0, ∀k ≠ j;
8:        if Pred(t_G(X')) < Pred(t_G(X)) then
9:          X := X';
10:       end if
11:     end if
12:   end for
13: end for
14: return X;
```

In Algorithm 1, first we initialize X with a feasible solution X_0. In our implementation, X_0 was set to the solution given by Storm's default scheduler, which evenly distributes threads over all machines. Then all threads are traversed (Line 2), and for each thread, the algorithm tries to re-assign it to every possible machine (Line 3–7) that is different from the current solution. While it tries different solutions for threads, the prediction model produces the corresponding (predicted) average tuple processing time (Lines 8–9). Since the prediction model is already trained, Line 8 is considered to take constant time. Therefore the time complexity of Algorithm 1 is $\mathcal{O}(MN)$.

D. Functions of Different Modules

1) Data Collector: The data collector runs as multiple threads throughout all machines in the cluster. When a tuple arrives from another thread of a PU or a data source, the tuple transfer latency is recorded with a unique tuple ID. Then the tuple will be processed and sent to a thread of the next PU, while the tuple processing latency is again recorded by the data collector with the tuple ID. Timestamps are captured and inserted into data records and further sent to the distributed data store (described next). Every collected data record comes with a unique tuple ID, which ensures correct tracking of every tuple's processing and transfer latencies.

2) Data Store: The distributed data store works closely with the data collector. Whenever a data record is collected by the data collector, it will be sent to the distributed datastore, which is a shared data grid throughout the cluster. The data store is implemented by integrating Hazelcast [9] into the proposed framework. Usually, several data records will be generated by different threads at a time, which will be sent to the distributed data store instantly.

3) Time Synchronizer: Threads of the data collector are distributed over processes/machines across the whole cluster, which require time synchronization over the network. The time synchronization is implemented by integrating PTP daemon (PTPd) [10] into the framework.

4) Data Pre-processor: Since the data are collected from threads/processes/machines throughout the cluster, for every single data tuple, there could be several corresponding records in the data store, mixed with records of other data tuples. Those raw data records cannot be used for prediction because they need to be concatenated into new records such that there is a one-to-one correspondence between a data record and a tuple. To achieve this, the data pre-processor congregates data records with the same tuple ID into a new record containing information needed for prediction.

5) Performance Predictor: The performance predictor builds a prediction model using the method described in Section II-B.

6) Schedule Generator: The schedule generator generates a scheduling solution using the predictive scheduling algorithm described in Section II-B.

7) Custom Scheduler: After the scheduling solution is generated by the schedule generator, the custom scheduler deploys it on the cluster via the master. Note that our design handles schedule generation and deployment with two separate modules such that with our framework, the current scheduling algorithm can be replaced by a new one at runtime without shutting shown the cluster.

III. Performance Evaluation

A. Experimental Setup

We implemented the proposed predictive scheduling framework over Apache Storm 0.9.2 release [11] and installed it on top of Ubuntu Linux 12.04. Experiments were performed on a cluster at Syracuse University's Green Data Center for performance evaluation. The Storm cluster consists of 10 IBM blade servers connected by a 1Gbps network, each of which was configured to have 10 slots. Next, we describe the two typical stream applications (topologies) used in our experiments.

Word Count Topology (stream version) (Fig. 3): Widely known as a MapReduce application, Word Count lists every word's number of appearances in a set of files. The stream version used in our experiments does a very similar job but with a stream data source. The topology has a chain-like structure with one spout and three bolts. The topology uses an open-source log agent called LogStash [12] to read data from input source files. The spout produces data stream while the input file is pushed into the queue. The spout is connected with the SplitSentence bolt which splits each input line into individual words and feeds them to a WordCount bolt that counts the number of appearances using fields grouping. The results are further sent to a Mongo bolt which stores them into a Mongo database. In the experiments, the topology was set to have 6 Spout executors (threads), 8 SplitSentence bolt executors, 8 WordCount bolt executors and 8 Mongo bolt executors.

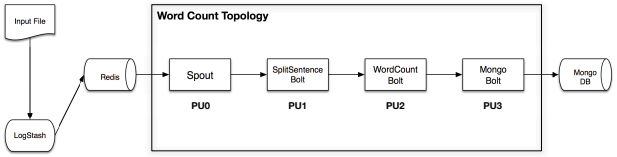


Fig. 3. Word Count Topology (stream version)

Log Stream Processing Topology (Fig. 4): Log stream processing represents one of the most popular use cases for Storm. The topology also uses Logstash to read data from log files. The LogRules bolt performs rule-based analysis on the log stream and delivers values containing a specific type of log entry instance. The log entry is then sent to two separate bolts simultaneously: one is the Indexer bolt which performs index actions and another one is the Counter bolt which performs counting actions on the log entries. For testing purpose, the original topology was slightly modified by introducing two Mongo bolts, one for the Indexer bolt and another one for the Counter bolt. These two Mongo bolts store the results into separate collections in a Mongo database for verification. In the experiments, the Storm cluster was configured to have 3 Spout executors (threads), 6 LogRules bolt executors, 6 Indexer bolt executors, 6 Count bolt executors and 6 executors for each Mongo bolt.

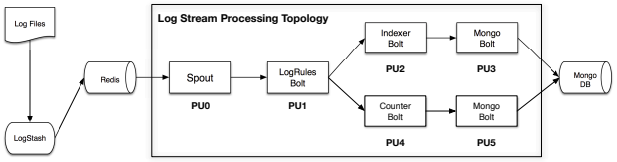
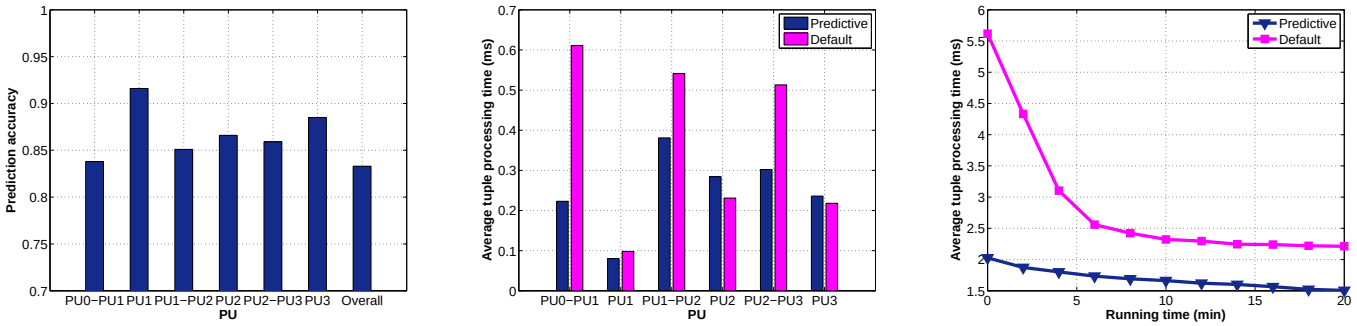
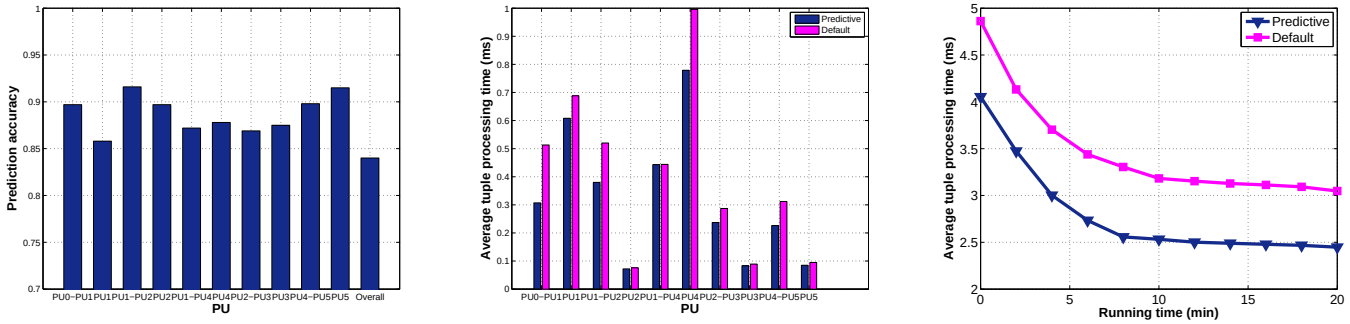


Fig. 4. Log Stream Processing Topology



(a) Prediction accuracy (b) Average tuple processing/transfer latencies at/between PUs (c) Average tuple processing time (over a topology)

Fig. 5. Performance on a word count topology (stream version)



(a) Prediction accuracy (b) Average tuple processing/transfer latencies at/between PUs (c) Average tuple processing time (over a topology)

Fig. 6. Performance on a log stream processing topology

B. Experimental Results and Analysis

We used two commonly used metrics for performance evaluation, including prediction accuracy and average tuple processing time (of a topology after stabilization) [8]. To justify the effectiveness of the proposed scheduling algorithm, Storm's default scheduler was used as the baseline solution for comparison.

For each topology, three figures are presented to show the experimental results. The first one shows the prediction accuracy, which is given by $(1 - \frac{|\text{Pred}-\text{Act}|}{\text{Act}})$, where Pred is the predicted value and Act is obtained from actual readings collected by data collector. For prediction, we used the readings of various features (described above) within the first 20 minutes after stabilization as the training data. During the training phase, randomly generated scheduling solutions were used to obtain sufficient amount of diverse data for training. In the corresponding figures, "PU0-PU1", "PU1" and "Overall" refer to the prediction accuracy of average (over 30 seconds) tuple transfer latency between PU0 and PU1, average tuple processing latency at PU1 and average tuple processing time of the whole topology after stabilization, respectively. The second figure shows the average (over 30 seconds) tuple processing time at a PU and average tuple transfer latency between PUs (after stabilization) given by the proposed scheduling framework and the default scheduler respectively. The third figure shows the average tuple processing time of the whole topology (given by the proposed scheduling framework and the default scheduler respectively) over a period of 20 minutes. From these experimental results, we can make the following observations.

Word Count Topology (stream version): The experimental results of this topology are shown in Fig. 5. From Fig. 5(a), we can see that the prediction accuracies for individual PUs vary from 83.8% (at SplitSentence bolt) to 91.6% (between the Spout and SplitSentence bolt). The prediction accuracy for the whole topology is 83.3%, which is slightly lower than individual prediction accuracies.

From Fig. 5(b), we can see that the predictive scheduling framework outperforms the default scheduler in terms of average tuple transfer latency. For example, the improvement on the average tuple transfer latency between PU0 and PU1 is the most significant one, which is over 60%. The improvements are around 30% for both PU1-PU2 and PU2-PU3. However, we also notice that the predictive scheduling framework performs slightly worse than the default scheduler at PU2 and PU3. This is because compared to the default scheduler (that leads to almost even distribution of workload), the predictive scheduling framework tends to consolidate workload to reduce

tuple transfer latency by assigning more threads to a machine, which, however may lead to longer tuple processing latency at some PUs. Even though some PUs experience longer tuple processing latency, the predictive scheduling framework reduces the overall tuple processing time of the whole topology, which can be observed from Fig. 5(c). From Fig. 5(c), we can observe that after a short initial period of about 8 minutes, the average tuple processing time (given by both schedulers) stabilizes at a lower value (compared to its initial value). The default scheduler's initial value is relatively high (5.6 ms) and then decreases significantly to about 2.2 ms. The predictive scheduling framework starts with a much lower initial value (2.0 ms) and stabilizes at about 1.5 ms, which represents a reduction of 31.8%.

Log Stream Processing Topology: The experimental results of this topology are shown in Fig. 6. From Fig. 6(a), we can see that the prediction accuracies for individual PUs vary from 85.8% (at LogRules bolt) to 91.5% (between LogRules bolt and Indexer bolt). Similarly, the prediction accuracy for the whole topology is 84%, which is slightly lower than individual prediction accuracies.

From Fig. 6(b), we can discover that the predictive scheduling framework outperforms the default scheduler in terms of tuple transfer latency, while for different PUs, the improvement can vary, from 40% for PU0-PU1, to nearly flat for PU1-PU4. From Fig. 6(c), we can see that both schedulers stabilizes after $8-10$ minutes. Two curves show similar trends. For the default scheduler, it starts with an initial value of 4.9 ms and stabilizes at about 3.0 ms. For the predictive scheduling framework, the initial value is lower (at 4.0 ms) and stabilizes at about 2.4 ms, which shows a 20% improvement.

IV. Related Work

In [14], scheduling strategies aiming at minimizing tuple latency and total memory requirement were proposed for a Data Stream Management System (DSMS). In [15], an offline scheduler and an online scheduler were presented for Storm. Xu *et al.* presented T-Storm in [8], which focused on traffic-aware scheduling that minimizes inter-node and inter-process traffic in Storm while ensuring no worker nodes were overloaded. It also enables fine-grained control over worker node consolidation. In [16], Bellavista *et al.* proposed a general and simple technique to design and implement priority-based resource scheduling in flow-graph-based distributed stream processing systems. In an early work [4], Nicola and Jarke provided a survey of performance models for distributed and replicated database systems, especially queueing theory based models. In [17], Wei *et al.* proposed a prediction based Quality-of-Service (QoS) management scheme for periodic queries over dynamic data streams. In a recent work [18], the authors presented a novel technique for resource usage estimation of data stream processing workloads in the cloud.

Different from these related work, we present a novel topology-aware method to predict average tuple processing time particularly for the programming model used by emerging big stream data processing platforms such as Storm and S4.

V. Conclusions

In this paper, we presented the design, implementation and evaluation of a predictive scheduling framework aiming at fast and distributed stream data processing. For prediction, a topology-aware method was presented to accurately predict the average tuple processing time of an application for a given scheduling solution, using runtime statistics. For scheduling, an effective algorithm was presented to assign threads to machines under the guidance of prediction results. We implemented it based on Storm, and tested it in a cluster with two representative applications: word count (stream version) and log stream processing. Extensive experimental results show 1) The topology-aware prediction method offers an average accuracy of 83.7%. 2) The predictive scheduling framework reduces the average tuple processing time by 25.9% on average, compared to the Storm's default scheduler.

References

[1] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Proceedings of USENIX OSDI'2004*.

[2] Apache Storm, *http://storm.apache.org/*

[3] S4, *http://incubator.apache.org/s4/*

[4] M. Nicola and M. Jarke, Performance modeling of distributed and replicated databases, *IEEE Transactions on Knowledge Discovery and Data Engineering*, 2000, Vol. 12, No. 4, pp. 645–672.

[5] T. Akidau, A. Balikov, K. Bejiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom and S. Whittle, MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 2013, pp. 1033–1044.

[6] 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, *Proceedings of EuroSys'2013*.

[7] V. Jakkula, Tutorial on support vector machine (SVM), School of EECS, Washington State University, 2006.

[8] J. Xu, Z. Chen, J. Tang and S. Su, T-Storm: Traffic-aware online scheduling in Storm, *Proceedings of IEEE ICDCS'2014*.

[9] Hazelcast, *http://hazelcast.com/products/hazelcast/*

[10] PTP daemon - Precision Time Protocol daemon, *http://ptpd.sourceforge.net/*

[11] Storm 0.9.2 on Apache Software Foundation, *https://storm.apache.org/2014/06/25/storm092-released.html*

[12] Logstash - Open Source Log Management, *http://logstash.net/*

[13] P. Bakkum and K. Skadron, Accelerating SQL database operations on a GPU with CUDA, *Proceedings of the 3rd Workshop on General-Purpose Computation on GPU (GPGPU'10)*, pp. 94–103.

[14] Q. Jiang and S. Chakravarthy, Scheduling strategies for a data stream management system, *Technical Report CSE-2003-30*.

[15] L. Aniello, R. Baldoni and L. Querzoni, Adaptive online scheduling in Storm, *Proceedings of ACM DEBS'2013*.

[16] P. Bellavista, A. Corradi, A. Reale and N. Ticca, Priority-based resource scheduling in distributed stream processing systems for big data applications, *IEEE/ACM International Conference on Utility and Cloud Computing*, 2014, pp. 363–370.

[17] Y. Wei, V. Prasad, S. Son and J. Stankovic, Prediction-based QoS management for real-time data streams, *Proceedings of IEEE RTSS'2006*.

[18] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, P. P. Jayaraman, J. Hosking and E. Abbasnejad, Resource usage estimation of data stream processing workloads in datacenter clouds, 2015, *http://arxiv.org/abs/1501.07020*.