Elastic Resource Provisioning for Batched Stream Processing System in Container Cloud

Song Wu[1], Xingjun Wang[1], Hai Jin[1], and Haibao Chen[2]

[1] Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
{wusong,wangxingjun,hjin}@hust.edu.cn
[2] School of Computer and Information Engineering
Chuzhou University, Chuzhou, 239000, China
chb@chzu.edu.cn

Abstract. Batched stream processing systems achieve higher throughput than traditional stream processing systems while providing low latency guarantee. Recently, batched stream processing systems tend to be deployed in cloud due to their requirement of elasticity and cost efficiency. However, the performance of batched stream processing systems are hardly guaranteed in cloud because static resource provisioning for such systems does not fit for stream fluctuation and uneven workload distribution. In this paper, we propose *EStream*: an elastic batched stream processing system based on Spark Streaming, which transparently adjusts available resource to handle workload fluctuation and uneven distribution in container cloud. Specifically, *EStream* can automatically scale cluster when resource insufficiency or over-provisioning is detected under the situation of workload fluctuation. On the other hand, it conducts resource scheduling in cluster according to the workload distribution. Experimental results show that *EStream* is able to handle workload fluctuation and uneven distribution transparently and enhance resource efficiency, compared to original Spark Streaming.

Keywords: Elastic, Resource provisioning, Stream processing, Container

1 Introduction

Due to data explosion in recent years, massive stream data that require to be processed in real time are increasingly common in business community and academia. Parallel stream processing systems play a critical role in such scenarios and provide low latency and high throughput. Traditional parallel stream processing systems, such as S4 [10] and Storm [15], are mostly designed based on operator model. In these systems, an application is represented with a topology of operators. Each operator implements a particular processing logic and is usually accelerated with a number of instances. Records in stream are processed by the operators within the topology. In that way, records are handled one by one

with latency in tens of milliseconds. Batched stream processing model, which combines the features of batch processing with stream processing, is proposed to meet demands of higher throughput and fault tolerance in recent years [20].

With the prevalence of cloud computing, cloud data center is increasingly being explored to deploy big data applications, such as parallel stream processing application, instead of building a local cluster. Recently, container technology is on the road to be a lightweight alternative to virtual machine (VM). Compared to VM, container imposes less overhead for resource virtualization since they share the OS kernel of physical machine. Until now, most researches discuss the elasticity of stream processing system in VM-based cloud. For example, Stela [18] is presented to adjust parallelism of operators in Storm with VMs. Unfortunately, the long startup time (usually in seconds) prevents VM from providing real-time resource provisioning. Besides, resource adjustment in VM is also complex.

Since stream workload fluctuation is common in production and it is difficult to make accurate estimation of resource requirements to handle the workload in advance or adjust resource configuration on the fly. As a result, there will be extra payments for the waste of cloud resource for customers. Many efforts have been paid to estimate reasonable resource provisioning to optimize performance for parallel distributed applications in cloud [6, 14]. Besides, in batched stream processing systems, sometimes blocks in batches are in different sizes, which leads to uneven execution time of tasks. As a result, the parallelism of stream processing is influenced. Similar problem exists in Storm due to different traffics on operators and adjusting the number of operators or slot reallocation is common in existing researches [3, 7]. Some researches also pay attention to handle this problem with dynamic operator placement or parallelism adjustments [17, 18]. However, few researches focus on batched stream processing system, thus, an elastic resource provisioning method for bathed stream processing systems in cloud is promising and still in demand.

In this paper, we solve the above problems and propose *EStream*, an elastic distributed batched stream processing system which bridges the gap between stream processing applications and resource management in cloud.

In summary, we make the following contributions in this paper:

- We build an optimization model of batched stream processing system and determine when it needs resource adjustment.
- We introduce cluster scaling and local resource scheduling with historical execution information, targeting to satisfy the requirement of system stability and higher resource efficiency under fluctuant workload. Then we design and implement *EStream* based on Spark Streaming and container technology (i.e., Docker).
- We conduct a series of experiments to evaluate the performance of *EStream*, the results show that *EStream* is able to adapt to workload fluctuation and uneven distribution and the system resource efficiency is also enhanced.

The rest of this paper is organized as follows: In Section 2, we review and discuss some related work. Section 3 describes background of our design. We

introduce an optimization mathematical model which guides our design in Section 4. Section 5 shows the overall system design of *EStream* and explains how it works. Performance evaluation results are presented in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work

There are a great deal of researches that concentrate on resource managements of parallel stream or batch processing systems. Many researches on stream processing systems mainly focus on operator-based systems. For instance, Fu et al. [3] propose a resource scheduler which dynamically allocates processors to operators to ensure real-time response. They model the relationship between resources and response time and use optimization models to get processor allocation strategy. Xu et al. [18] present a VM operator instance scaling technique according to throughput of operators. Their approach does not work on batched stream processing system, which is not operator-based. What is more, the time and monetary cost of VM scaling is expensive. Cervino et al. [1] propose an adaptive approach for provisioning VMs for stream processing system based on input rates but latency guarantees are ignored. In addition, Madsen et al. [9] integrate fault-tolerance and dynamic resource managements to avoid waste of excessive processing delay. Wu et al. [16] introduce ChronoStream, which provides vertical and horizontal elasticity. In a word, there are few researches on resource management of containerized batched stream processing system.

As for batch processing systems, a flexible slot management scheme to accelerate task execution for off-line MapReduce jobs is shown by Guo et al. [4]. Their work partly inspires our design. Ruan et al. [14] present an approach based on monetary cost model for cloud resource provisioning for Spark with sample running, which inspires us to leverage the historical information of previous batches in batched streaming system to make resource allocation decision in our design. Park et al. [12] dynamically increase or decrease the computing capability of each node to enhance locality-aware task scheduling with dynamic virtual CPU number configurations.

There are some recent studies that aim to improve performance of batched stream processing system. For example, Das et al. [2] ensure system stability and low latency with dynamic batch sizing, which changes task parallelism in runtime. However, resource efficiency is ignored in their work.

Our paper is different from previous work in several aspects. First, we model batched stream processing system with considering the features of both batch and stream and get the scheduling target which guides our design. Second, we design our elastic resource provisioning approach with execution information of previous processed batches. The approach automatically scales containerized cluster and carries out resource scheduling with the help of container technology. Finally, we implement *EStream* which takes full advantages of containerized environments.

3 Background

3.1 Batched Stream Processing

In this subsection, we take Spark Streaming as an example to describe batched stream processing.

Data Model. *Discretized stream* (DStream) is the abstraction of data stream in Spark Streaming which is a series of *resilient distributed datasets* (RDDs)[19]. RDD is introduced in Spark framework to persist data in memory or disk. Hot data can be cached in memory to avoid frequent IO operations to improve performance of batch jobs. As shown in Fig.1, each RDD in DStream is the received stream data in a batch interval and stored with a number of blocks. The block size is determined by block interval in application.

Execution Model. Fig.1 also demonstrates the execution model of Spark Streaming. In Spark Streaming, user's application gives the definition of transformations on RDDs. When a batch is received and persisted as an RDD, a job is accordingly generated and submitted to the job scheduler. Each job waits in a scheduling queue to be executed if there are former jobs that are not completed. Jobs will finally be transformed into tasks and each task is executed on a certain executor in cluster.

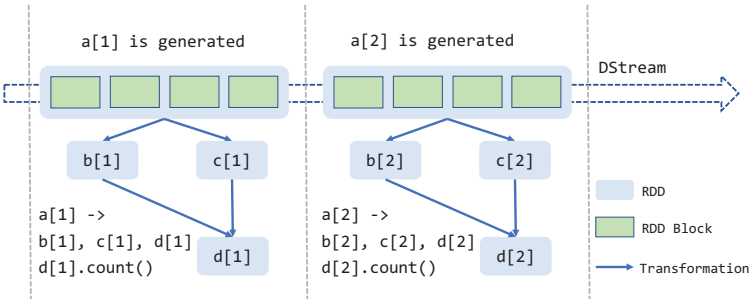


Fig. 1. Batched stream processing model with a sample RDD transformation definition in Spark Streaming. In the example, RDD a[i] is generated in batch[i]. RDD b[i], c[i], d[i] are transformed by different APIs from a[i]

The characteristics of batched stream processing system are: 1) Jobs are tiny and latency sensitive. Due to this feature, resource provisioning method should be proactive, simple, and efficient; 2) Jobs are recurrent [5, 13], this nature inspires us to use runtime information of former processed batches to make workload prediction for further resource provisioning decision.

3.2 Container Technology

Container is an OS-level virtualization technology and gaining great much attention in recent years. One of the core technologies in container is CGroup, which is a mechanism for grouping and limiting resource of tasks, and all their future children, into hierarchical groups with specialized behavior in Linux kernel. With container, users can build a more flexible and lightweight platform for applications. There is a trend for container technology to replace traditional VM-based application deployment mode due to its lightweight, fast startup time, and near-native performance. Since container provides near-native performance and millisecond-level startup time, it is suitable for elastic stream processing system, which requires fast and efficient resource provisioning. In this paper, we use Docker[1] in our design to provide elasticity because it is one of the most popular container technologies and is widely used for application in production system.

4 Modeling

Before modeling the batched stream processing, some basic concepts are described as follows.

- *Batch interval*: the time period used to divide real-time data stream into batches, denoted by I.
- *Batch processing time*: the time consumption for processing a batch, denoted by p.
- *Scheduling delay*: the waiting time for a batch from when it is received to the time when it begins to be processed, denoted by s.
- *Latency*: the time cost for a batch from when it is received and prepared to be processed to the time when it is completely processed, denoted by l. It is obvious that $l = p + s$.

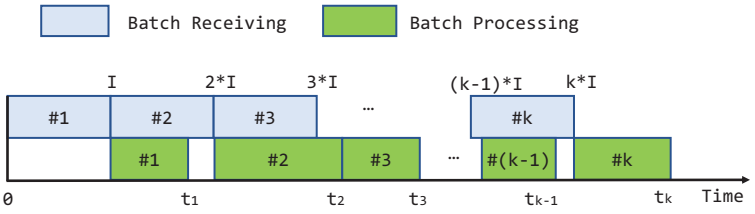


Fig. 2. A scenario of batched stream processing. Batched stream processing application is started at time t_0=0, and $batch_k$ is received and ready to be processed at time $k * I$, then it is completely processed at time t_k

In this section, we discuss a scenario of a batched stream processing system which starts an application at time t_0=0 and n batches are handled in the system

[1] http://docker.com/

until now. The i_{th} batch $batch_i$ is completely finished at time t_k. Processing time of $batch_i$ is p_k and its scheduling delay is s_k. Fig. 2 displays the scenario. In order to simplify our discussion, we make the assumption that the system in our discussion always runs normally and no crash or other faults exist.

If $t_{k-1} > k * I$, it means when $batch_k$ is received and ready for processing, the system is busy in processing $batch_{k-1}$, in other words, $batch_k$ is blocked in the queue, and its block time is $BT_k = t_{k-1} - k * I$. When $t_{k-1} < k * I$, $batch_{k-1}$ is processed at once when it is ready and system is idle before processing $batch_k$, and the system idle time is $ST_k = k * I - t_{k-1}$. Things are quite simple when $t_{k-1} = k * I$: $batch_k$ begins to be processed at once when it is ready. Therefore, there is:

$$t_k = max\{t_{k-1}, k * I\} + p_k \tag{1}$$

In order to obtain better resource utilization, the system idle time should be as less as possible. On the other hand, the system should process batches as soon as possible when they are received. Therefore, the goals of our system design are to minimize the sum of batch blocked time (BT) and the sum of system idle time (ST). This can be modeled as an optimization problem:

$$\min_{p_i} \quad \sum_{i \in B} BT_i, \sum_{i \in D} ST_i$$

$$s.t. \begin{cases} t_k = max\{t_{k-1}, k * I\} + p_k, & 2 \le k \le n \\ t_1 = I + p_1 \end{cases} \tag{2}$$

where B is the set of batches that are blocked before processing and D is the set of batches that begin to be processed at once when they are received. There is $U = B \cup D$, where U is the set of all n batches in the scenario.

Since this is a multiple-objective and non-linear optimization model and cannot be solved directly, it is common to convert the original problem with multiple objectives into a single-objective optimization problem. Considering the two objectives in the model are all urged to be minimized, a linear combination is suitable for this conversion. With linear combination, each objective is assigned with a weight range from 0 to 1, and the sum of weights is 1. With this method, we get the transformed model:

$$\min_{p_i} \quad \alpha \cdot \sum_{i \in B} BT_i + \beta \cdot \sum_{i \in D} ST_i \quad (\alpha + \beta = 1)$$

$$s.t. \begin{cases} t_k = max\{t_{k-1}, k * I\} + p_k, & 2 \le k \le n \\ t_1 = I + p_1 \end{cases} \tag{3}$$

To simplify the problem, we give the same importance to the two objectives and choose $\alpha = 0.5$ and $\beta = 0.5$. Note that BT_i and ST_i can be unified to be demonstrated with $|t_{i-1} - i * I|$. We finally transform the above model to:

$$\min_{p_i} \quad \sum_{i=1}^{n} |t_{i-1} - i * I|/2$$
$$s.t. \begin{cases} t_k = max\{t_{k-1}, k * I\} + p_k, & 2 \leq k \leq n \\ t_1 = I + p_1 \end{cases} \tag{4}$$

By solving this model, we can get the solution:

$$p_1 = p_2 = p_3 = \cdots = p_n = I \tag{5}$$

The result indicates that if processing time of batches is equal to batch interval, there will be $l_k = t_k - k * I = p_k = I$ and $s_k = 0$. In that case, batches in data stream are processed in time and the system is never idle. This conclusion means that our resource provisioning mechanism should target to make processing time of batches reach as closer to batch interval as possible. Therefore, when we find $p \neq I$, we should adjust resource provisioning of the system.

5 System Design

5.1 Overview

EStream aims to make proactive resource provisioning by collecting and analyzing the execution information of processed batches. The overview of *EStream* is shown in Fig. 3. Two main components that run in Spark driver are: *Execution Information Collector* (EIC) and *Elasticity Manager* (EM). Executors of batched stream processing system run in Docker containers and they are managed by NodeManager and ContainerDaemon on the host server. NodeManager is in charge of starting or shutting down containers according to the requests from ClusterResourceManager. ContainerDaemon receives resource scheduling request and adjusts resource allocation of containers. There are mainly three steps to elastically provision resource:

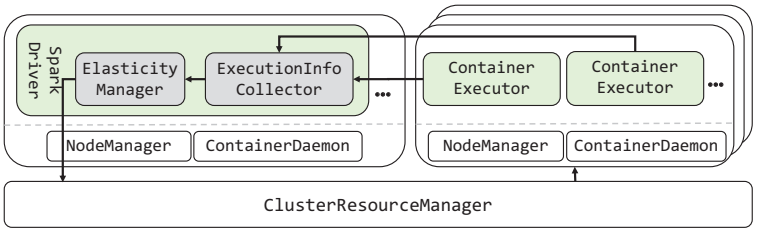


Fig. 3. Overview of *EStream*

1. *Execution information collection*: EIC gathers historical execution information about the workload handled on each container executor and execution time in previous processed batches. Processing time of last batch is also recorded by EIC. Then EIC sends the information to EM.

2. *Workload prediction and execution analysis*: After receiving execution information from EIC, EM firstly makes a workload prediction for each executors. Then it analyzes execution of the last batch and determines a resource provisioning plan. The resource provisioning plan is sent to ClusterResourceManager.
3. *Resource allocation*: When ClusterResourceManager receives the resource provisioning plan from EM, it will send this request to each NodeManager and ContainerDaemon. Then NodeManagers start or shutdown containers in response to the request from ClusterResourceManager. ContainerDaemon performs resource scheduling among containers.

In summary, EIC collects historical execution information and transmits it to EM. EM analyzes the historical information and determines whether a resource adjustment is needed. If so, EM triggers elastic resource provisioning to prevent the system from performance degradation in the future. In the next, we are going to explain more details of EIC and EM.

5.2 Execution Information Collector

EIC is designed to perform collection of historical execution information for later analysis. It is activated every time when a batch or a task is completed to obtain runtime information. The EIC collects the following information:

1. Workload handled by each executor in the last N batches, a window of size N is maintained in EIC. N is set to a default value of 5 in *EStream* according to our experimental results.
2. Total execution time on each executor in the last processed batch. It is calculated with $theLastTaskFinishTime - theFirstTaskStartTime$.
3. Processing time of last batch.

We implement EIC with customized event listeners that will be notified with the information we need when batches are processed.

5.3 Elasticity Manager

EM is in charge of analyzing historical information and making resource provisioning decision. When an application is started, EM is started as a daemon process in Spark driver at the same time. Spark driver is responsible for task scheduling of applications in Spark Streaming.

First, EM makes a workload prediction. For each executor in the cluster, EM predicts its workload in next batch with *Exponential Weighted Moving Average* (EWMA), which is a simple but efficient prediction algorithm. As shown in Eq. 6, workload on an executor in i_{th} batch w_i is used to predict w_{i+1}. $w^{'}$ is the difference of workload handled by this executor between the last two batches. λ indicates the weights of w_i in prediction of w_{i+1}, ranging from 0 to 1. We set $\lambda = 0.8$ in our design and experimental results show it works well.

$$w_{i+1} = \lambda * w_i + (1 - \lambda) * w^{'} \tag{6}$$

Second, EM conducts execution analysis on last processed batch. As discussed in Section 4, our target is to make batch processing time as closer to batch interval I as possible. Since it is infeasible to guarantee that batch processing time is equal to I all the time and we should detect the resource problem in advance to make a proactive resource adjustment in order to avoid future performance degradation. A threshold μ $(0 < \mu < 1)$ is given to address this problem. When processing time of the last batch exceeds $\mu * I$, EM will try to prevent the risk of performance degradation through elastic resource provisioning. We set 0.9 as the default value of μ in our design according to lots of experiments. We introduce an *Elastic Resource Provisioning* (ERP) algorithm in EM which aims to handle two situations as follows:

Workload Fluctuation. When facing workload fluctuation, the system may suffer a performance degradation from resource insufficiency or it is under low resource utilization. ERP algorithm recognizes these two situations with analysis on execution time on executors and last batch processing time. We introduce a cluster scaling strategy in ERP to ensure performance and resource utilization of the system. If resource over-provisioning occurs, we scale in the cluster to save resource usage. On the contrary, if resource is insufficient, we scale out the cluster in response to increasing workload. The details of two situations are as follows:

1. *Scaling out*: If processing time of last completed batch is larger than $\mu * I$, executors whose execution time is larger than $\mu * I$ are regarded as slow executor. If the number of slow executors exceeds $\delta * clusterSize$, where δ is a threshold with default value of 0.4 according to our experimental results. ERP scales out the cluster to handle the increasing workload.
2. *Scaling in*: If processing time of last completed batch is smaller than $\mu * I$, executors whose execution time is smaller than $\gamma * I$ are marked as fast executor, where γ is a threshold with default value of 0.5 to identify fast executor. If the number of fast executors exceeds $\delta * clusterSize$, ERP scales in the cluster to save resource.

When cluster scaling is determined, ERP firstly calculates the average processing speed ($avgSpeed$) of each executor with:

$$avgSpeed = \frac{totalWorkloadInLastBatch}{clusterSize * processingTimeOfLastBatch} \tag{7}$$

Then the new cluster size is estimated with:

$$clusteSize^{'} = \lceil \frac{workload_{predict}}{\mu * I * avgSpeed} \rceil \tag{8}$$

where $workload_{predict}$ is predicted workload of next batch.

Uneven Workload Distribution. When RDD blocks of batches are in different sizes, execution time on executors in the cluster is influenced. Some executors may suffer heavy workload which results in long task execution time. As a result, total execution time is increased on these executors and batch processing time is increased. In such situation, a common idea is to make load balancing across executors in the cluster, but this cannot be completed in real time and will lead to extra overhead, such as network bandwidth. Fortunately, container technology provides another approach to this problem. It is possible to adjust computing resource among executors in containers (Docker). ERP leverages this mechanism to make resource scheduling to handle uneven workload distribution.

Specifically, when last batch processing time is larger than $\mu * I$, ERP algorithm calculates the number of slow executors. If the number of slow executor is smaller than $\delta * clusterSize$, i.e., there exists an uneven workload distribution, ERP will conduct resource scheduling among executors. In this paper, our ERP only focuses on CPU resource allocation, because it is considered as the main bottleneck of Spark framework [11]. Docker supports CPU resource limitation with the help of CGroup and it is easy to carry out an adjustment with RESTful APIs. In our design, we introduce a proportional allocation strategy for CPU resource scheduling. ERP allocates CPU usage among container executors with:

$$cpuRatio_i = \frac{workload_i}{\sum workload_i} \cdot totalCpuRatio \tag{9}$$

The pseudo-code of ERP algorithm is shown in Algorithm 1. The key idea is to scale cluster when over-provisioning or resource insufficiency exists and adjust resource allocation to handle uneven workload distribution. More specifically, if the last batch processing time exceeds the threshold $\mu * I$, executors whose execution time is larger than $\mu * I$ is regarded as slow executors (line 3-6). If the number of slow executor is larger than $\delta * clusterSize$, EM decides to scale out the cluster (line 7-10). Otherwise, *EStream* will carry out a CPU resource reallocation among container executors (line 11-16). The details of cluster scaling are as follows: 1) estimating average handling speed on executor (line 8); 2) calculating the new cluster size according to predicted total workload and the average handling speed (line 9). If the batch processing time is less than the threshold $\mu * I$, EM counts the number of fast executors (line 18-20). If the number of fast executors is larger than $\delta * clusterSize$, EM will scale in the cluster (line 21-24).

The cost analysis. Suppose the cluster size is m when ERP algorithm is called. The space complexity of ERP algorithm is $O(m)$ and the time complexity is $O(m)$. This is because the amount of information of each executor gathered by EIC is fixed and the amount of space needed only depends on m. When the cluster needs to be scaled, the average handling speed of executor is calculated in $O(m)$ time. Then ERP calculates the desired cluster size in $O(1)$ time. As for CPU resource scheduling, ERP firstly gets total CPU ratio in $O(m)$ time. Then new CPU ratio of each executor is calculated in $O(1)$ time.

Algorithm 1 Elastic resource provisioning algorithm

Input: 1) Batch processing time in last batch: pt; 2) Batch interval: I; 3) Workload handled on executors in last N batches: $W[N][excutorId : workload]$; 4) CPU usage of executors: $cpuRatio[executorId : percentage]$; 5) Execution time on executors in last batch: $d[executorId : executionTime]$;

Output: New cluster size or CPU resource allocation plan.

$slowExecutorNum \leftarrow 0$, $fastExecutorNum \leftarrow 0$, $clusterSize \leftarrow executorIds.size$, $hostServers \leftarrow$ host servers of container executors, $w \leftarrow W[N-1]$;
Predict workload on each executor: $w^{'}[executorId : workload] \leftarrow EWMA(W)$
if $pt > \mu * I$ **then**
  for each $id \in executorIds$ **do**
    if $d[id] > \mu * I$ **then**
      $slowExecutorNum++$
  if $slowExecutorNum > \delta * clusterSize$ **then**
    $avgSpeed = \frac{\sum w[i]}{clusterSize * pt}$
    $clusterSize^{'} = \lceil \frac{\sum w^{'}[i]}{\mu * I * avgSpeed} \rceil$
  return $clusterSize^{'}$
  if $0 < slowExecutorNum < \delta * clusterSize$ **then**
    for each $host \in hostServers$ **do**
      $executors \leftarrow$ executors on $host$.
      for each $id \in executors$ **do**
        $cpuRatio^{'}[id] = \frac{w^{'}[id]}{\sum w^{'}[id]} * \sum cpuRatio[id]$
  return $cpuRatio^{'}[]$
else
  for each $id \in executorIds$ **do**
    if $d[id] < \gamma * I$ **then**
      $fastExecutorNum++$
  if $fastExecutorNum > \delta * clusterSize$ **then**
    $avgSpeed = \frac{\sum w[i]}{clusterSize * pt}$
    $clusterSize^{'} = \lceil \frac{\sum w^{'}[i]}{\mu * I * avgSpeed} \rceil$
  return $clusterSize^{'}$

6 Evaluation

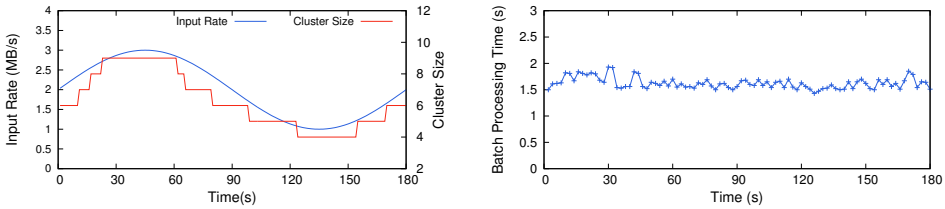
6.1 Experiments Setup and Method

Our experiments are conducted on a private container cloud. Each physical host is comprised of two quad-core 2.4GHz Intel CPU, 24GB memory, 1TB disk and 1Gbps Ethernet card. We implement *EStream* on Spark-1.6.0 and the version of Docker is 1.12.3. Each container executor is configured with 2GB memory, 2 cores and 5% CPU usage limitation when it is started. We use WordCount, a widely used benchmark application [8, 20] which counts the number of words in a fixed time interval in our experiments. The batch interval of WordCount application is 2 seconds.

6.2 Experiment with Workload Fluctuation

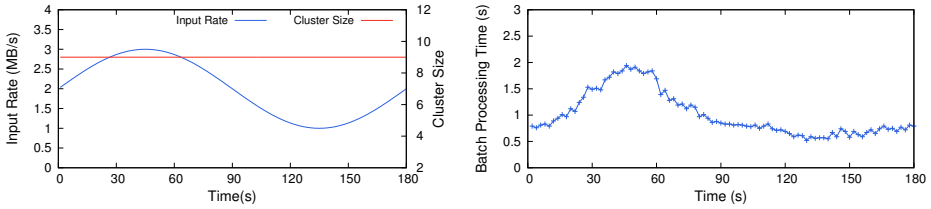
In this experiment, we simulate a data stream with a sinusoidal input rate ranging from 1 to 3 MB/s. We generate records from a set of words with the same length as the seed of input stream. The cluster in this experiment is initialized with 6 container executors. We run *EStream* with simulated data stream for 180 seconds. Fig.4(a) shows that, with the fluctuation of input rate, the cluster size of *EStream* changes accordingly, the maximum and minimal cluster size are 9 and 6, respectively. Fig.4(b) shows batch processing time in *EStream*. As shown in the figure, when input rate changes, batch processing time in *EStream* is relatively stable.

As for Spark Streaming, we deploy a cluster of 9 container executors, which is the maximum cluster size needed to handle peak traffic in *EStream* test. Fig.5(a) shows the input rate and cluster size in this test. Fig.5(b) shows batch processing time in original Spark Streaming test. Batch processing time in Spark Streaming suffers a fluctuation with the variation of input rate because of resource reservation configuration of Spark Streaming and it cannot elastically scale its container cluster. As we know, cloud payment is calculated by the number of instances and the runtime. We build a function $f(t)$ to denote the relationship between cluster size and time in our test, thus, resource utilization in this test can be expressed with $\int_0^{180} f(t)dt$. With this formulation, we can figure out that *EStream* saves a resource usage of about 30%, compared to Spark Streaming.



(a) input rate and cluster size (b) batch processing time

Fig. 4. Performance of *EStream* with sinusoidal input rate



(a) input rate and cluster size (b) batch processing time

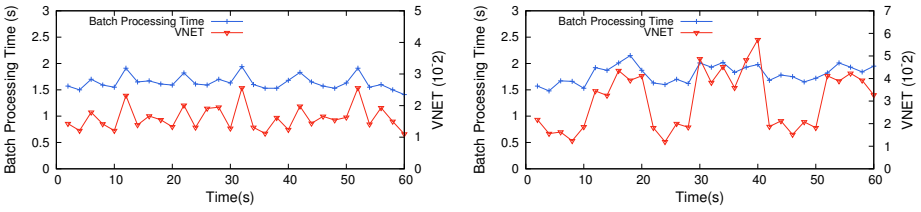
Fig. 5. Performance of Spark Streaming with sinusoidal input rate

6.3 Experiment with Uneven Workload Distribution

To evaluate the adaptability of *EStream* to uneven workload distribution, we simulate a data stream with constant input rate of 3 MB/s. We generate two files with the same size of 3 MB as the seed of our simulated data stream. One file $file_1$ is a set of words with same length, while the other one $file_2$ is a set of words with random length. This means the number of words in these two files are different. The simulated data stream changes its seed in every 10 seconds. In each round, the data source sends the data from $file_1$ for 10 seconds, then $file_2$ is selected as the seed for the next 10 seconds. We setup a cluster of 9 container executors to carry out this experiment. We run WordCount in both *EStream* and Spark Streaming for 60 seconds. As mentioned above, uneven workload distribution leads to the difference of execution time on executors. We collect and normalize the execution time on each executor. We use the *variance of normalized execution time* (VNET) to measure the impact on parallelism caused by uneven workload distribution. Fig. 6 shows the results of *EStream* and Spark Streaming under the simulated data stream in this experiment.

As we can observe in Fig. 6(a), when workload changes every 10 seconds, the VNET suddenly increases, due to the uneven workload distribution, as well as batch processing time. However, the two metrics decreases soon. This is because *EStream* makes resource adjustment among executors to deal with the uneven workload distribution. Note that when the seed of data stream is changed to $file_1$, the two metrics increase because the CPU allocation in the previous 10 seconds does not match the current workload distribution. But *EStream* will soon solve this problem.

Fig. 6(b) shows the performance of Spark Streaming under the same condition. It is obvious that after the changing of data stream seed to $file_2$, both of the VNET and batch processing time no longer decrease in the next 10 seconds. By comparing the batch processing time when the system is under uneven workload distribution in this test, *EStream* reduces batch processing time by 19% on average.



(a) *EStream* (b) Spark Streaming

Fig. 6. Batch processing time and *variance of normalized execution time* (VNET) in *EStream* and Spark Streaming with uneven workload distribution

7 Conclusion

In this paper, we concentrate on common problems in stream data processing: workload fluctuation and uneven distribution. In order to achieve higher resource efficiency while provide performance guarantees in cloud, we design and implement *EStream*, which providing elastic resource provisioning for batched stream processing applications. We exploit container technology to realize fast cluster scaling and efficient resource reallocation in runtime to handle workload fluctuation and uneven workload distribution. In contrast to the original Spark Streaming, experimental results show *EStream* is able to adapt to varying workload and save a resource usage of about 30%. Besides, *EStream* can transparently schedule CPU resource among executors to enhance parallelism and reduce batch processing time.

Acknowledgments. This research is supported by National Key Research and Development Program under grant 2016YFB1000501, 863 Hi-Tech Research and Development Program under grant No. 2015AA01A203, and National Science Foundation of China under grants No. 61232008. This work is also supported by the Anhui Natural Science Foundation of China under grant (No. 1608085QF147), and Key Project of Support Program for Excellent Youth Scholars in Colleges and Universities of Anhui Province (No. gxyqZD2016332).

References

1. Cervino, J., Kalyvianaki, E., Salvachua, J., Pietzuch, P.: Adaptive provisioning of stream processing systems in the cloud. In: Proceedings of 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW). pp. 295–301. IEEE (2012)
2. Das, T., Zhong, Y., Stoica, I., Shenker, S.: Adaptive stream processing using dynamic batch sizing. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). pp. 1–13. ACM (2014)
3. Fu, T.Z., Ding, J., Ma, R.T., Winslett, M., Yang, Y., Zhang, Z.: Drs: dynamic resource scheduling for real-time analytics over fast streams. In: Proceedings of 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS). pp. 411–420. IEEE (2015)
4. Guo, Y., Rao, J., Jiang, C., Zhou, X.: Flexslot: moving hadoop into the cloud with flexible slot management. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 959–969. IEEE (2014)
5. Jyothi, S.A., Curino, C., Menache, I., Narayanamurthy, S.M., Tumanov, A., Yaniv, J., Goiri, Í., Krishnan, S., Kulkarni, J., Rao, S.: Morpheus: towards automated slos for enterprise clusters. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). p. 117. USENIX (2016)
6. Kambatla, K., Pathak, A., Pucha, H.: Towards optimizing hadoop provisioning in the cloud. In: Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). vol. 9, p. 12. USENIX (2009)

7. Kumbhare, A., Frincu, M., Simmhan, Y., Prasanna, V.K.: Fault-tolerant and elastic streaming mapreduce with decentralized coordination. In: Proceedings of 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS). pp. 328–338. IEEE (2015)
8. Lin, W., Qian, Z., Xu, J., Yang, S., Zhou, J., Zhou, L.: Streamscope: continuous reliable distributed processing of big data streams. In: Proceedings of USENIX Symposium on Networked System Design and Implementation (NSDI). pp. 439–454. USENIX (2016)
9. Madsen, K.G.S., Zhou, Y.: Dynamic resource management in a massively parallel stream processing engine. In: Proceedings of the 24th ACM International Conference on Information and Knowledge Management (CIKM). pp. 13–22. ACM (2015)
10. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: Proceedings of 2010 IEEE International Conference on Data Mining Workshops (ICDMW). pp. 170–177. IEEE (2010)
11. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.G.: Making sense of performance in data analytics frameworks. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 293–307. USENIX (2015)
12. Park, J., Lee, D., Kim, B., Huh, J., Maeng, S.: Locality-aware dynamic vm reconfiguration on mapreduce clouds. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC). pp. 27–36. ACM (2012)
13. Rasley, J., Karanasos, K., Kandula, S., Fonseca, R., Vojnovic, M., Rao, S.: Efficient queue management for cluster scheduling. In: Proceedings of the 11th European Conference on Computer Systems (EuroSys). p. 36. ACM (2016)
14. Ruan, J., Zheng, Q., Dong, B.: Optimal resource provisioning approach based on cost modeling for spark applications in public clouds. In: Proceedings of the Doctoral Symposium of the 16th International Middleware Conference. p. 6. ACM (2015)
15. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 147–156. ACM (2014)
16. Wu, Y., Tan, K.L.: Chronostream: Elastic stateful stream computation in the cloud. In: Proceedings of 2015 IEEE 31st International Conference on Data Engineering (ICDE). pp. 723–734. IEEE (2015)
17. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic load distribution in the borealis stream processor. In: Proceedings of 2005 21st International Conference on Data Engineering (ICDE). pp. 791–802. IEEE (2005)
18. Xu, L., Peng, B., Gupta, I.: Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In: Proceedings of IEEE International Conference on Cloud Engineering (IC2E). pp. 22–31. IEEE (2016)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 2–2. USENIX (2012)
20. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). pp. 423–438. ACM (2013)