© 2016 Anirudh Jayakumar

ADAPTIVE BATCHING OF STREAMS TO ENHANCE THROUGHPUT AND TO SUPPORT DYNAMIC LOAD BALANCING

BY

ANIRUDH JAYAKUMAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Tarek F. Abdelzaher

ABSTRACT

As data permeates all disciplines, the role of big data becomes increasingly important. Sensors, IoT devices, social networks, and online transactions are all generating data that can be monitored constantly to enable a business to identify opportunity to enhance customer service and increase revenue. This need for real-time processing of big data has led to the development of frameworks for distributed stream processing in clusters. It is important for such frameworks to be resilient against variable operating conditions such as server load variation, changes in data ingestion rates, and workload characteristics. In this thesis, we explore the effects of the batch size on the performance of streaming workloads by developing an adaptive batching framework and building load-balancing algorithms on top of this framework. We explore the idea of using a combination of adaptive batching of tuples and dynamic tuple dispatching to improve the throughput and load-distribution of the workload. We show through experiments that the system is able to be resilient and robust under varying operating conditions.

To my mother, Shobhana.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Tarek Abdelzaher, for letting me explore different research ideas. He was patient with me and gave me invaluable insights throughout my MS program. He also provided me with all the support needed to achieve my goals. I would also like to thank my mentor, Dr Shen Li for his invaluable and continuous guidance during the thesis right from its conception and for suggesting many ideas to overcome the problems I encountered along the way. My parents and family have made me the person I am today and I cannot thank them enough for being supportive of all my endeavors. Finally, I would like to thank Banu, Ankita, Varsha, Avinash and Aashik for making my life infinitely more interesting in the otherwise quiet town of Urbana.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Distributed stream-processing system (DSPS) such as Apache Storm [1], Apache Flink [2], Spark Streaming [3], Samza [[4] are getting popular for their ability to process massive amount of real-time data in parallel with low latency on commodity hardware. Such an infrastructure can aid organizations that need to quickly assess large volumes of incoming information to make business decisions. In the past decade, Hadoop has been the de facto open-source tool for most big data processing. Hadoop was designed primarily as a batch processing system implementing the MapReduce framework and can have a high latency when deployed to process high-volume, high-velocity workloads.

Most DSPS define the data model as a directed-acyclic graph (DAG), with vertices representing processing elements (PEs) and edges representing the flow of data. The Some DSPS like Apache Storm also support cycles in the graph. For scalability, the streams are partitioned into sub-streams and executed by multiple instances of a processing element - processing element instance (PEI). The current parallel stream processing systems are geared towards low-latency processing resulting events generated for each element in the stream. For example, in Apache Storm, each tuple (infinite tuples constitute a stream) is transported to the next PEI immediately in order to reduce the end-to-end processing latency for each tuple. Sending these tuples in the aforementioned manner can result in huge overheads. A major slice of the CPU is spend context-switching and waking up threads that are responsible for data transfer between PEIs. This scheme works well for high latency-sensitive applications, but reduces the throughput considerably.

As with any distributed data processing system, running in a multi-tenant environment, stream processing systems are also prone to load imbalance. It is well known that load-imbalance can happen due to various causes. In a typical production cluster, there could be multiple applications running using a resource scheduler like YARN [5] or Mesos [6]. Multiple PEIs belonging to different applications share the same physical node due to which performance of each PEI is dependent on the load on other PEIs in the node. To add more complexity, schedulers like YARN can manage applications of different types. So there could be a cluster running Hadoop jobs, storm topologies and even a NoSQL database like Cassandra, all sharing the resources of the cluster. This can create CPU, memory and network resource contention. Load imbalance could also be due to the nature of the application. Big-data applications [7, 8] in data mining and machine learning require gathering and storing state during the life time of the application. The data is partitioned based on key and all the data (streams) with the same key are processed by the same PEI. This represents a single choice paradigm as it does not consider the popularity of the key. The workload might be skewed towards few keys leading to some PEIs being more loaded than others.

Load distribution is a classical problem in distributed and parallel systems. Traditionally, most systems handle load distribution by intelligently allocating replica tasks to processing units statically [9]. Dynamic load distribution is achieved by migrating processing units and the associated state to less loaded nodes [10, 11]. This can be an expensive operation as the execution will have to be paused while the migration is taking place. Other popular techniques include speculative execution in Hadoop where the system monitors the progress of the task and may assign a replica task for any stragglers in the system. In this scheme both the processing units continue to process the data till one among them completes the execution.

In this work we explore the idea of using a combination of Adaptive batching of tuples at the PEI level and dynamic tuple dispatching to improve the throughput and load-distribution of the workload. In the adaptive batching scheme, the system can control the batch size of each PEI and the processing starts only when the batch size is full. For dynamic tuple dispatching, the system can set an outgoing load-proportion which the tuple forwarder can read and send a proportional amount of tuples to the receiving PEI. This way the infrastructure provides a mechanism to vary the data rates at each PEI. From previous work on batching at middleware-level [12] and application-level [13], we know that batching can improve throughput by several folds. The concept of tuple batching has not been studied in the context of load-balancing as per our knowledge. With our technique we foresee no downtime during load balancing and also believe that a range of algorithms related to various topics like load-balancing, throughput and latency improvement and energy optimization can be developed on top of our adaptive batching framework.

The rest of the thesis is organized as follows: We look at the architecture of Apache Storm and Spark Streaming in Chapter 2 and investigate the effects of batching on performance in Chapter 3. We present the design of our adaptive batching framework in Chapter 4. The load balancing algorithms are presented in Section 5 with experimental results in Section 6. We outline the related work in Section 7 and conclude in Section 8.

CHAPTER 2

BACKGROUND

There are a multiple stream processing systems, both production systems as well as research systems [1, 2, 3, 4, 14, 15]. The most popular stream processing systems are Apache Spark with its new Streaming API [3] and Apache Storm [1], which was initially developed at Twitter Inc. As of this writing, Apache Flink [16] is also gaining a lot of momentum in the stream processing space especially due to its higher-level APIs and native support for batch processing. For our work, we implement the adaptive batching scheme within Apache Storm. Our decision was based on the fact that Storm is a pure stream processing system, unlike Spark Streaming, which is an abstraction build on top of its batch-processing engine. In the below sub-section we very briefly discuss the internal architecture of Spark Streaming and discuss in detail the architecture of Apache Storm.

2.1 Spark Streaming

Spark streaming represents a continuous stream of input data as a discretized stream, or DStream[3]. Internally, Spark stores and processes DStream as a sequence of Resilient Distributed Datasets (RDD). RDD is one of the main abstraction of Spark. RDD is a collection of elements partitioned across the nodes of the cluster that can be operated in parallel. Each of these RDDs in a streaming setup is a snapshot of all the data ingested during a specified time period. This allows Sparks existing processing capability to operate on the data. This model of chopping the stream into chunks is called micro-batching. The batch interval generally ranges from as little as 500ms to about 5000ms (can be higher). The batch interval is static and cannot be changed while the application is running. Reducing the time interval will bring the system closer to real-time, but will result in more overhead due to

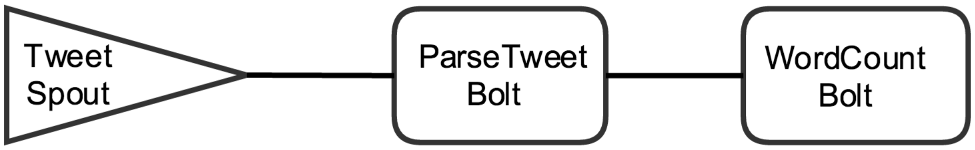


Figure 2.1: Tweet word count topology

a large number of RDDs. Spark Streaming is an excellent choice for users who require both streaming and batching capabilities in the same system and users who do not mind higher-latencies.

2.2 Apache Storm

Apache Storm is a real-time distributed stream data processing engine that was originally developed at Twitter. Various companies power their critical real-time data management tasks using Storm. Storm is designed to be scalable, resilient, extensible, efficient and easy to administer. The basic Storm data processing architecture consists of streams of tuples flowing through topologies. Topologies in Storm are directed graphs where vertices represent the processing elements and the edges represent the stream of data flowing between processing elements. There are two kinds of processing elements in Storm Spouts and Bolts. Spouts are the source of data. They are responsible for injecting tuples into the topology. Spouts often pull data from queues such as Kafka [17] or Kestrel [16]. Bolts form the main processing component comprising of the business logic of the application. Bolts process the incoming tuples from spout or other bolts and passes new set of tuples to downstream bolts. Figure 2.1 shows an example topology that counts words occurring in a stream of tweets and reports that stats every 5 minutes. The *TweetSpout* injects tweets into the topology using Twitter's APIs, *ParseTweetBolt* tokenizes the tweets and emits a tuple <word,1>for each word and *WordCountBolt* receives these tuples and maintains a table of word counts. After every 5 minutes, the *WordCountBolt* reports the statistics and clears its internal state.

Figure 2.2 shows the high level execution architecture of storm. The execution architecture of Storm consists of a master node called Nimbus. Nimbus

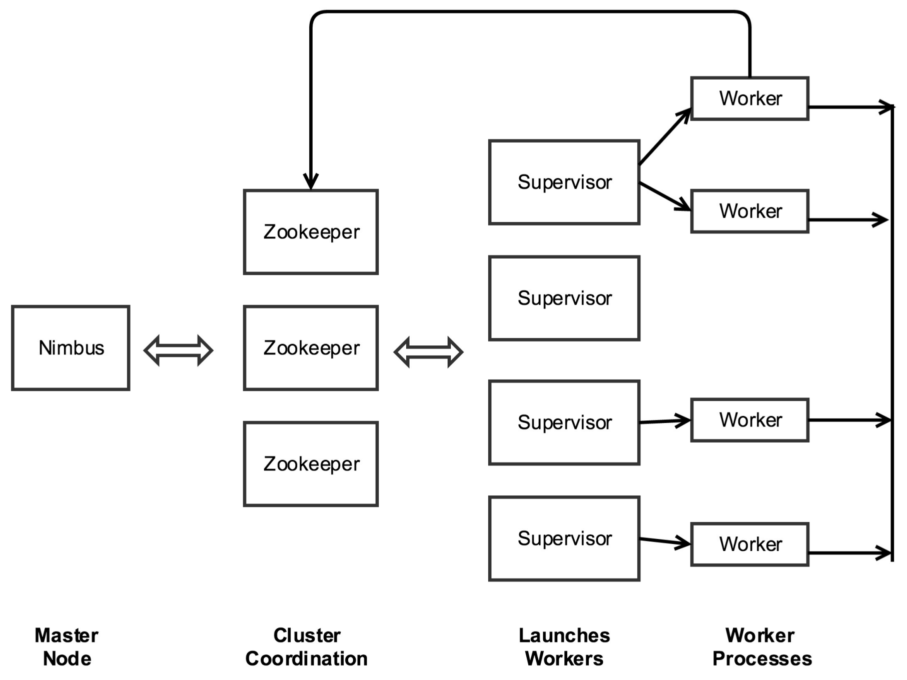


Figure 2.2: Apache Storm Architecture

is responsible for distributing and coordinating the execution of the topology. The actual work is done on worker nodes. Each worker node runs a Supervisor that connects with Nimbus and sends periodic heartbeat, advertising topologies that are currently running and any resource availability to run more topologies. The supervisor manages one or more worker process, with each worker mapped to a single topology. All coordination between Nimbus and Supervisor is done using Zookeeper. Each worker runs a JVM, inside which there are executor threads. Each executor runs one or more tasks. The user defined spouts and bolts are mapped to the tasks. By default, each executor only runs one task. Thus, tasks provide intra-bolt/intra-spout parallelism, and the executors provide intra-topology parallelism.

Buffering of tuples in Storm happen at two levels inter-worker transfer queues and intra-worker LMAX disruptor queues [18]. The inter-worker buffering happens again at two places. The first place is at the client side where messages are buffered to be transferred across the wire and second place is at the server-side where a dedicated thread forwards the tuples to downstream components. There are two pairs (send/receive) queues at the executor level, which forms the basis for intra-worker tuple transfer. A re-

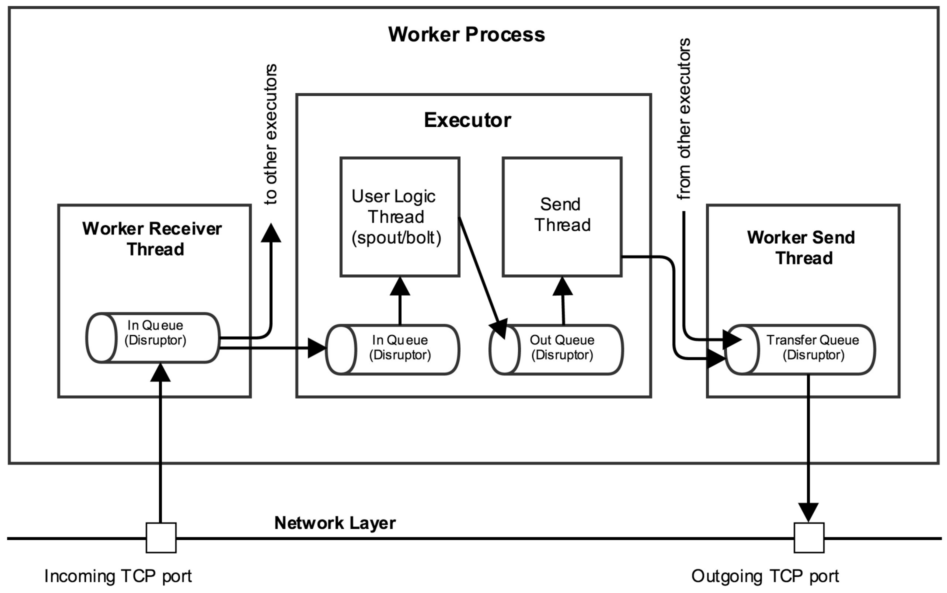


Figure 2.3: Overview of a workers internal message queues in Storm

cent patch[19] added static batching of tuples at the disruptor queue level using two parameters – batch size and batch interval. Batch size is the total tuples that can be accumulated before being processed and batch interval is the time period after which he accumulated tuples will be processed. Either of these parameters can trigger the message flush depending on which event happens first (either batch size getting filled or batch interval ending).

CHAPTER 3

EFFECT OF BATCHING ON PERFORMANCE

In this section we look at the effect of batching on critical performance parameters like latency, throughput and CPU usage. We begin by looking at the the variation in context-switches while batching tuples before processing. All the experiment in this section was done on one node with the same configuration as mentioned in Section 6 running the word count topology as shown in 2.1. The topology is run with one worker and two instances for each of the PEs (*TweetSpout, ParseTweetBolt* and *WordCountBolt*).

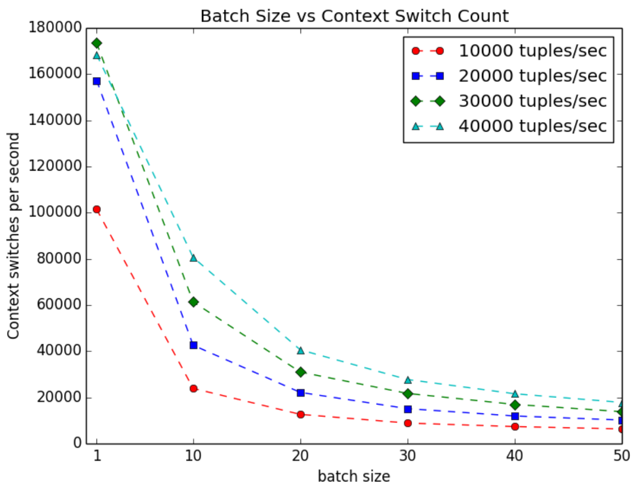


Figure 3.1: The effect of batch size on context-switches

3.1 Context Switches

In Storm, the usual behavior is to process an incoming tuple right away by one of the thread in the disruptor queue thread pool. Hence, when the data rate is high, large number of thread are waiting for the CPU to process the tuple. There is also a system overhead of waking up, context switching and book-keeping the threads which can use up a significant portion of the CPU, especially when the data rate is high. In our batching scheme, a thread is woken up only when the batch is filled to the configured size, hence reducing the number of active threads in the system and reducing the thread maintaining overhead.

In Figure 3.1 we look at the effect of batching on the total context switches in the application. There is a significant drop in the context-switch count when batching is introduced. When we set the batch size to 10, we see a decrease of 80%, 74%, 65% and 52% for incoming data-rate of 10k, 20k, 30k and 40k tuples per seconds respectively. The decrease in context-switched saturates when the batch size increase. We don't see a significant decrease when moving from batch size 40 to 50. What is also interesting is the higher context-switches when the date-rate is high. So, for every system, there is a data rate limit above which the system is not capable of handling the incoming data without dropping some of the traffic.

3.2 CPU Usage

The decrease in the context-switch means that the amount of CPU time spend by the kernel to schedule the threads and also the additional overhead of data movement to and from the cache can be avoided. This should result in lesser CPU usage by the application. We look at this in more detail by plotting the CPU time spend by the application for every 30 seconds of wall time. Figure 3.2 shows the variation in CPU times as the batch size changes. As expected, the CPU time variation has a similar pattern to the context switch count variation. The CPU time decreases drastically on the introduction of batching but saturates around batch size of 30. This CPU time that is saved can be used to either process more tuples hence, increasing the throughput of the system or can be used to schedule other jobs by the scheduler.

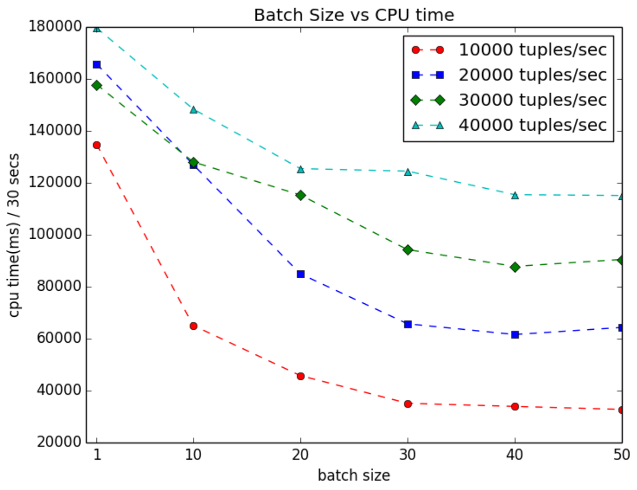


Figure 3.2: The effect of batch size on CPU time

3.3 Latency and Throughput

Now we look at the impact on latency. In the context of this work, we define latency as the time difference between the spout emitting a tuple and the spout receiving an *ack* from storm for the same tuple. When the tuple gets sent to consuming bolts, Storm tracks the tree of messages that are created. If Storm detects that a tuple is fully processed, Storm will call the ack method on the originating Spout task with the message id that the Spout provided to Storm. This is how Storm guarantees message processing. Storm developers are working to bring more accurate latency measurements within storm, but as of this writing such metrics don't exist.

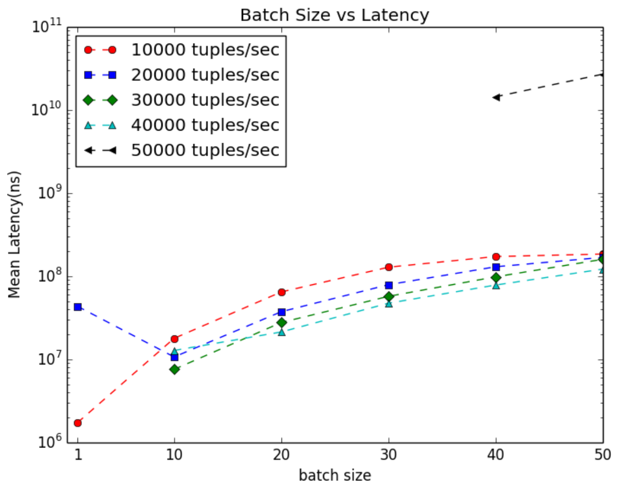


Figure 3.3: The effect of batch size latency

Figure 3.3 shows the impact of batching on latency. There is a multifold increase in latency when we introduce batching. We see a 10x increase in latency when increasing the batch size from 1 to 10. Similarly, there is a 37x, 74x, 99x and 106x increase for batch size 20, 30, 40 and 50 respectively when the data rate is 10k tuples/second. This is a significant increase considering that some of the streaming applications expects sub-second latencies. The other observation is that when the data-rate is high, the latency decreases. This is because, the batch is filled faster and the frequency of thread waking up to service the batch increases.

There are two other important takeaways from this plot. The first is the anomaly in the latency of the 20k data-rate job when there is no batching. This is surprising because all the reasoning till now suggested that batching increases the latency, but in the case of the 20k data-rate job, the latency decreased when the batch size was increased to 10. The reason for such behavior is that the system is unable to handle such a high data rate with no batching. Due to the high over head of processing each tuple, the threads wait for the CPU. If the incoming tuple count exceed the number of threads in the pool, then the tuples are stored in the disruptor queue till some thread is free to process it. Due to this, there is a delay in processing tuples. If this job was run for a longer time, the JVM would have hit the memory wall and the process would have crashed. The second major observation is the absence of any latency numbers for higher tuple rates at certain smaller batch sizes.

This is because of the same reason that the system was unable to handle the speed of the data injection in to the system. These jobs failed within 30 seconds from the start time due to which no metrics could be collected.

In our work, we use the advantages of batching, like higher-throughput and less CPU usage to adaptively load-balance the system. In the next chapter we describe the design and implementation of the adaptive batching framework. In chapter 5 we look at the various load-balancing schemes that we build on top of our batching infrastructure.

CHAPTER 4

ADAPTIVE BATCHING FRAMEWORK

The overall working of the Adaptive Batching Framework (ABF) is shown in Figure 4.1. The framework consists of three major components: 1) Metric Collector, 2) Adaptive Tuple Batching and 3) Adaptive grouping. The below sections explain each of these components in detail. The load-balancing schemes that we develop are build on this framework.

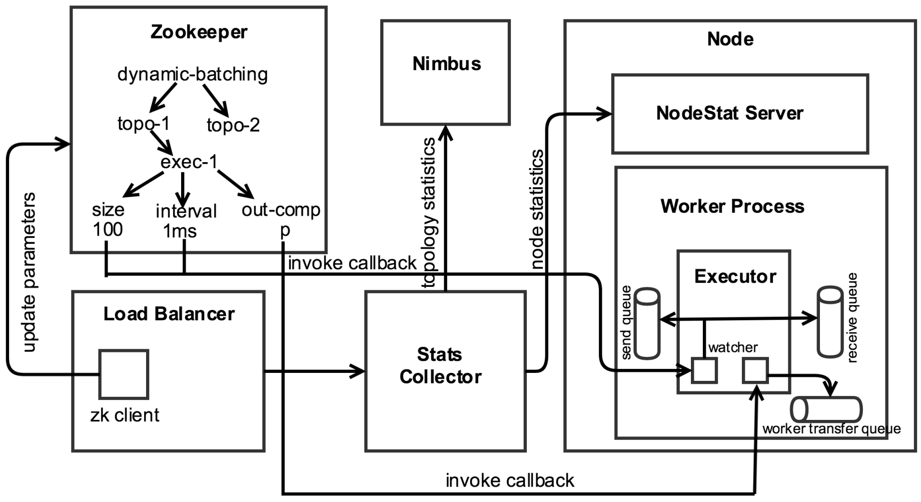


Figure 4.1: The Adaptive Batching Framework

4.1 Metric Collector

The Metric Collector (MC) is responsible collecting live metrics from the system. These metrics are of two types: a) topology statistics and b) node statistics. Topology statistics include metrics that define the performance of a topology. These include latency, throughput, input data rate and failed tuple count. All these statistics are gathered from Nimbus, the master component of Storm. The MC periodically polls Nimbus to get these statistics. Node

Statistics are metrics that are specific to each computing node. We run a Thrift service called *NodeStatServer* on each of the worker nodes to which the MC connects and periodically gather metrics from all the worker nodes in the system. The metrics gathered are memory usage, cpu usage, network usage and power consumed by the node. All these metrics are stored by MC and a set of APIs are provided to access various statistical information about the metrics.

4.2 Adaptive Tuple Batching

Tuple batching can be achieved by two mechanism, one using batch size other using a batch interval. ABF support both these mechanisms. The batch size/interval is updated by exploiting the callback scheme in Zookeeper /ref. Zookeeper is already used extensively in Storm for coordination. Every executor creates znodes with path "/dynamic-batching/ <topology-id >/ <exec-id >/interval" and "/dynamic-batching/ <topology-id >/ <exec-id >/size" for batch interval and batch size respectively. Each executor also registers a callback (also called watches) with the zookeeper, which gets triggered when the data in the znode is updated. There is separate call back for each of the two znodes per executor. On receiving this trigger, the disruptor queue is updated with either the size or the interval parameter. In the disruptor queue, the batch size is checked when a new tuple arrives and if the size reaches the limit, the tuples are forwarded for processing. A thread also is woken up every batch interval time and flushes out the tuples for processing. The tuples are flushed by either one of these techniques depending on the size, interval and data-rate values. If the data rate is high the batch fills up to the set size. The interval setting is a good way to enforce the minimum latency for processing. The components that are build on top of the ABF is expected to connect to Zookeeper and update the required fields.

4.3 Adaptive Stream grouping

The other important infrastructure feature to leverage adaptive batching is to have the flexibility to change the data-rate of tuple between instances of the same component. Part of defining a topology is specifying for each bolt which streams it should receive as input. A stream grouping defines how that stream should be partitioned among the bolt's tasks. Storm provide a number of grouping options out of which the most popular are shuffle grouping and fields grouping. We add some minimal changes to shuffle and fields grouping to make it forward tuples in different proportions so that framework can control the data rates on each of the PEIs.

4.3.1 Adaptive Shuffle Grouping

Shuffle grouping (SG) routes messages independently in a round-robin manner. SG tries to load balance the outgoing tuples among the processing instances by assigning almost equal number of messages to each instance. To make the shuffle grouping more dynamic we introduce Adaptive Shuffle Grouping (ASG) which routes the messages to different instances based on a custom proportional distribution. The distribution can be updated dynamically, enabling ASG to route messages based on the current proportional distribution. In the word count topology, if the *ParseTweetBolt* has five instances, then using ASG we can control the proportion of tuples going into each instance. For example, we can configure instance 0 to receive 50% of the the incoming tuples and other instances to receive 10% each.

4.3.2 Adaptive Partial Key Grouping

Fields Grouping (FG) ensures that messages with the same key are handled by the same processing instance. This is similar to message passing in MapReduce. The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks. It is very challenging to build adaptive fields grouping techniques due to the extensive use of fields grouping in state-full operations. Sending tuple with the same key to different PEI can decrease the accuracy of data mining/machine learning algorithms and also requires additional aggregator elements to consolidate the results. In our approach we build on the "power of both choices" approach studied in

[20]. The authors of this work use two downstream instances and route the message to the least loaded instance among the two based on local load estimate. The authors call this method partial-key grouping (PKG). We tweak this method by removing the local load estimator and replacing it with the custom proportional distribution technique use in ASG. Here, the choice is always between two instances as opposed to ASG, where the number of downstream instances depends on the parallelism of the component. We call this method adaptive partial key grouping (APKG). The limitations of PKG are detailed in [20]. The same limitations apply to APKG.

CHAPTER 5

LOAD BALANCING ALGORITHMS

In this section we look at the load balancing algorithms that can be build on top on our Adaptive Batching Framework (ABF). For this thesis we look at a data aware load balancing algorithm which exercises the dynamic batch size update feature of ABF. Also, as a continuation of this work we are implementing load-aware and energy-aware LB techniques which will exercise both the dynamic batch size update and the adaptive grouping techniques. More details about these algorithms are provided in the end of this chapter. In this chapter we look at *DataRateLB* in detail.

5.1 Data Rate Aware Load-Balancer

The Data Rate Aware Load Balancing algorithm (*DataRateLB*) helps Storm to adaptively react to the fluctuations in the input data rates. When topologies are deployed, it is often deployed taking into account the maximum load that the topology is expected to handle and as a result the resources are over-provisioned. In spite of this, in some cases the allocated resources still fall short. *DataRateLB* helps topologies to manage such fluctuations by allowing higher throughput processing when the incoming tuple rate exceeds the resources allocated to handle them as discrete streams. In a nutshell, the algorithm monitors the lag between the tuple generated count and the tuple completed count and increases the batch size of the PEs (executors) if the lag is above a certain administrator defined threshold. Algorithm 1 shows the complete algorithm followed by more detailed explanation. The description of all the variables used in the algorithm is given in Table 5.1.

Lines 1-6 initializes the variables. *diff_max* is initialized to START_MAX which is a administrator defined limit. Systems with more memory and higher latency SLAs can have a higher *diff_max*. *diff_min* is always *min_p* times *diff_max*. *diff_max* and *diff_min* are updated by the algorithm based on the measured *diff* of the topology. If the diff goes above *diff_max* then the range is adjusted to values between the current diff. *curr_update_value* fluctuates between [0,MAX_UPDATE] and helps control the granularity of batch size update. The batch size is always updated by adding or subtracting the value $2^{curr\_update\_value}$ from the current batch size. This helps in quickly reacting to continuous increase/decrease in the input data rates hence help

Algorithm 1: DATA RATE AWARE LOAD BALANCER

```
1  $batch\_action \leftarrow NONE$
2  $diff\_max \leftarrow START\_MAX$
3  $diff\_min \leftarrow min\_p * diff\_max$
4  $curr\_batch\_size \leftarrow 1$
5  $curr\_update\_value \leftarrow 0$
6  $reading\_count \leftarrow lb\_period/stats\_collection\_period$
7  while $true$ do
8      $acked \leftarrow stat\_collector.get\_ack\_count()$
9      $emitted \leftarrow stat\_collector.get\_emit\_count()$
10     $diff \leftarrow emitted - acked$
11     $bSLA \leftarrow sla\_satisfied()$
12     if $diff > diff\_max$  $AND$  $bSLA$ then
13         $curr\_update\_value \leftarrow (batch\_action ==$
             $INC)\ ?\ min(curr\_update\_value + 1, MAX\_UPDATE) : 0$
14         $curr\_batch\_size \leftarrow$
             $min(MAX\_BATCH, curr\_batch\_size + 2^{curr\_update\_value})$
15         $diff\_max \leftarrow diff * inc\_p$
16         $diff\_min \leftarrow min\_p * diff\_max$
17         $batch\_action \leftarrow INC$
18         $update\_batch\_size(curr\_batch\_size)$
19     else if $diff < diff\_min$ then
20         $curr\_update\_value \leftarrow (batch\_action ==$
             $DEC)\ ?\ min(curr\_update\_value + 1, MAX\_UPDATE) : 0$
21         $curr\_batch\_size \leftarrow max(1, curr\_batch\_size + 2^{curr\_update\_value})$
22         $diff\_max \leftarrow max(START\_MAX, diff\_max * dec\_p))$
23         $diff\_min \leftarrow min\_p * diff\_max$
24         $batch\_action \leftarrow DEC$
25         $update\_batch\_size(curr\_batch\_size)$
26     else
27         $batch\_action \leftarrow NONE$
28     $sleep(lb\_period)$
```

in keeping the system stable. The system could go into an unstable state if the batch size increase is constant while the data rates increase at a much higher

Table 5.1: Description of the variables defined in Algorithm 1

Variable	**Description**
batch_action	enum (NONE, INC, DEC) representing the action performed on the batch size
diff_max	the diff threshold above which the batch size will be increased
diff_min	the diff threshold below which the batch size will be decreased
curr_batch_size	represents the current size of the disruptor queue
curr_update_value	$2^{curr_update_value}$ is the size added to or subtracted from the batch size
reading_count	number of previous stats reading to be considered while making LB decisions
lb_period	the interval between each LB action
stats_collection_period	the interval for statistics collection
acked	the total number of tuples that have finished processing
emitted	the total number of input tuples generated
diff	*emitted* - *acked*
bSLA	boolean value describing if the SLA is met or not

rate. This scheme helps maintain stability of the topology. Similarly, if there is a quick drop in the data rate it would be better to get the system to process the tuples discretely that in a batch. Having an exponential increase/decrease in the batch size can help the system adapt faster to external changes. Line 7 starts an infinite loop that constantly monitors the topology and applies batch size updates. At the end of the loop the algorithm sleeps for *lb_period* seconds before resuming to load balance again. In lines 8-11, the parameters needed to take load balancing decisions are initialized. *diff* gives us the difference between the total *emitted* and total *acked* tuples hence giving an indication of the processing speeds of the topology relative to the input tuple rate. *bSLA* tells us if the topology is meeting the latency SLA. The Metric Collector provides three latency metrics a) mean, b) 99.9% latency and c) 99.99% latency. For all the experiments in this thesis we use the mean latency as the SLA metric. *sla_satisfied()* returns true if the mean latency is less than the SLA provided to the job.

Lines 12-27 contains the core elements of the algorithm. Line 12 checks if the *diff* is greater than *diff_max* in which case the algorithm increases the batch size so that PEs can catch up with the incoming tuples. But before we do that we check if the SLA is met. We know from section 3.3 that the latency increases on increasing the batch size, so we only increase the batch size if the latency is met otherwise the algorithm does nothing during the current iteration. In case where the SLA is not met, it is a good idea to drop tuples to avoid memory being used up due to tuple queuing. For our experiments we avoid dropping any tuples. In lines 13-14 we set the new batch size by adding $2^{curr\_update\_value}$ to the curr_batch_size. If the previous batch action was also to increment (action INC) the size then the current increment will be an exponential. In line 15, the *diff_max* is adjusted to be slightly higher than the current diff by multiplying diff with *inc_p*. *diff_min* is always *min_p* times *diff_max* (lines 3,16 and 23). In line 19, the algorithm checks to see if there is an opportunity to reduce the batch size by comparing with *diff* with *diff_min*. In lines 20-21 batch size is calculated similar to the INC operation by checking if there has been a DEC operation in the previous iteration and accordingly manipulating *curr_update_value* and *curr_batch_size*. If the *diff* is between the *diff_min, diff_max* range then we avoid any batch action to reduce the load on the zookeeper. This is algorithm is a simple heuristic and can be configured and extended for various other scenarios.

As part of the ongoing work, we are implementing two additional load balancing algorithms a) *LoadAwareLB* and b) *EnergyAwareLB*. Both these algorithms will exercise the complete ABF including the adaptive grouping schemes. *LoadAwareLB* will use the stats collector to monitor the load on each machine and intelligently routing traffic using the adaptive grouping schemes and dynamically updating the processing throughputs of PEs in nodes by varying the batch size. Similarly, *EnergyAwareLB* monitors the energy consumption of each node and routes the traffic such that the total energy consumption of the cluster can be optimized. With this range of algorithms implemented on top of ABF we showcase the versatility and flexibility of ABF and we encourage readers to develop more advanced algorithms on top of the ABF.

CHAPTER 6

EXPERIMENTS AND RESULTS

We evaluate the load balancing algorithms using two micro-benchmark applications [21] namely *WordCount* and *RollingSort*. Figure 6.1 shows the topology of each of two benchmarks. The *WordCount* topology is composed of one spout (*FileReadSpout*) and two bolts (*SplitSentence*, *CountBoltNum*). *FileReadSpout* reads the contents of the input file and emits all the lines in the file. For experiments we read infinitely the contents of the novel 'A tale of two cities'. The *SplitSentence* bolt tokenizes each line into words and emits them. The *CountBoltNum* counts the occurrence of each of the word. *WordCount* is a CPU sensitive benchmark. The *RollingSort* topology consists of one spout (*RandomMessageSpout*) and one bolt (*SortBolt*). *RandomMessageSpout* selects a random string message from a given static list and emits it to the topology. The *SortBolt* keeps a sorted list of all the incoming messages. When the sorted message count reaches a specific value the the list is emitted. *RollingSort* is a memory-sensitive benchmark.

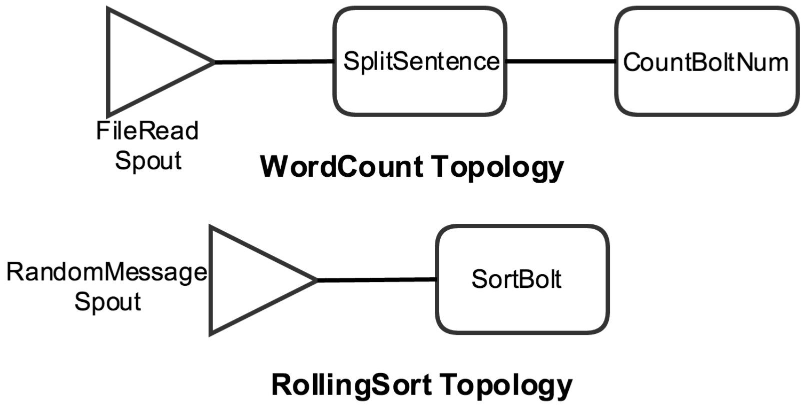
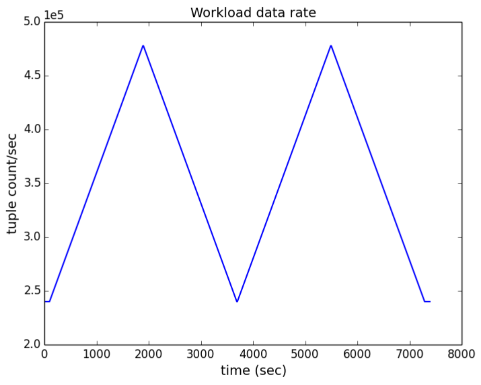


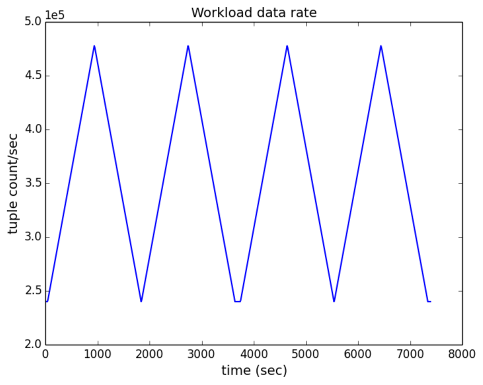
Figure 6.1: Micro-benchmark topologies

The experiments are carried out in a cluster consisting of 20 nodes with each node having 6-core Intel Xeon CPU with a clock speed of 2.0GHz and

16GB memory. The nodes are connected by a ToR switch with each node connected with a 1000 Mbit/sec ethernet interface card. For our experiments we use one node as Nimbus, 8-12 nodes as workers running the supervisor daemon, 3 zookeeper nodes and 1 node running both the Stats Collector and the Load-balancing algorithms.



(a) Workload D1



(b) Workload D2

Figure 6.2: Workload data-rates

6.1 Data Rate Aware Load-Balancer

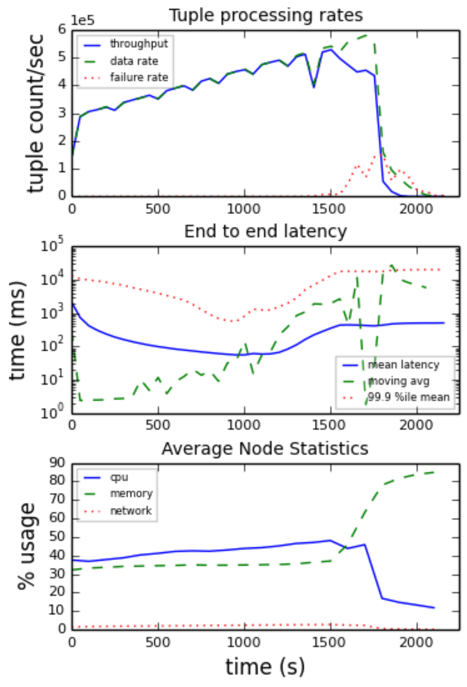
Variable	**Value**
period	30 secs
sla_type	mean
sla	750 ms
START_MAX	10000
MAX_BATCH	130
stat_period	10 secs
min_p	0.90
inc_p	1.10
dec_p	0.95

Table 6.1: DataRateLB parameters

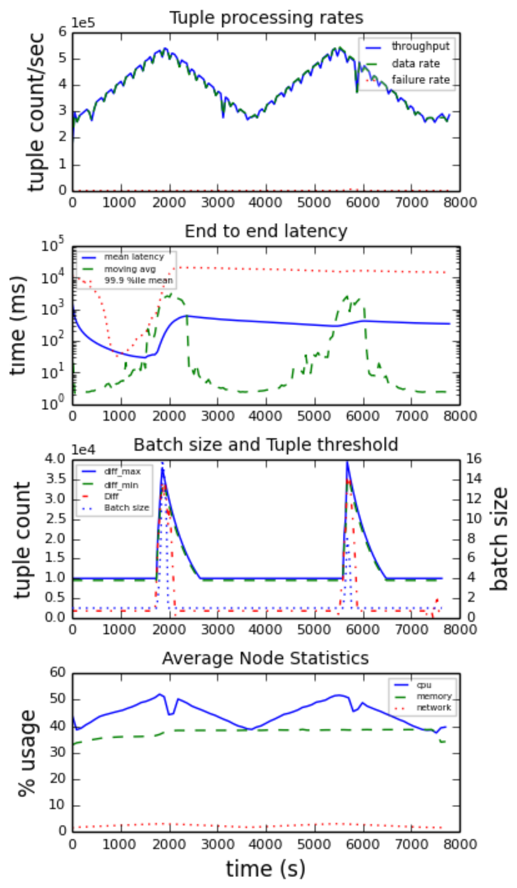
Variable	**Value**
spout/worker	2
SortBolt/worker	2
msg_size	1KB
chunk_size	400
emit_frequency	3 secs

Table 6.2: RollSort Topology Parameters

We look at the experimental results for the *DataRateLB* described in Algorithm 1. For the experiments we run the *RollingSort* topology with two time-varying input data-rates. Figure 6.2 shows the workload data rates used



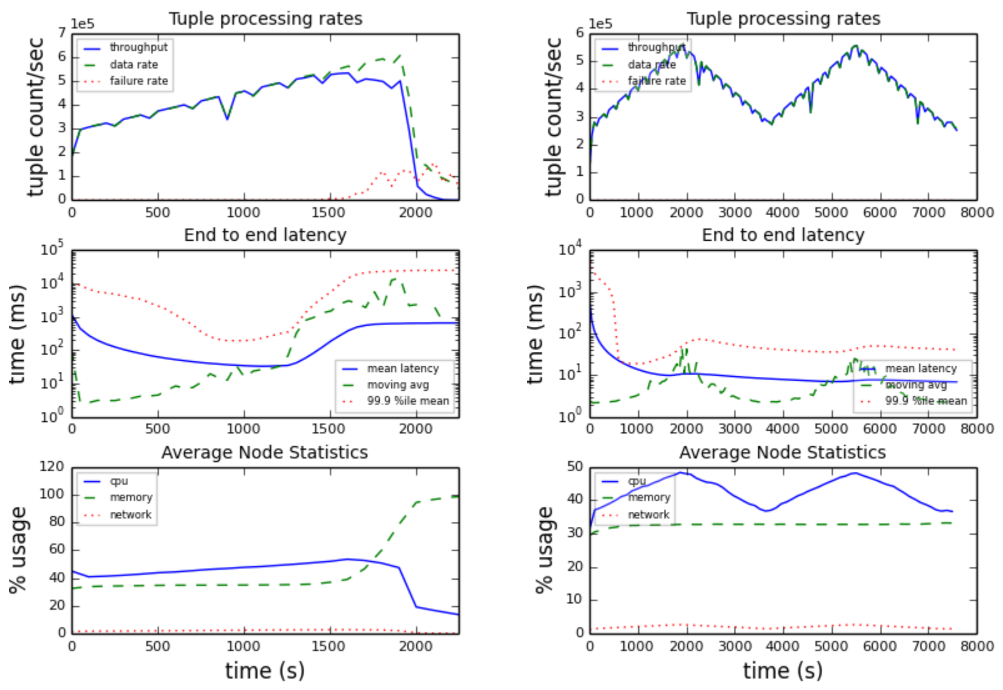
(a) Workload D1 on 9 workers without *DataRateLB*



(b) Workload D1 on 9 workers with *DataRateLB*

Figure 6.3: Workload D1 on 9 workers

in the experiments. D1 varies less as compared to D2 which fluctuates at a much higher rate. For both the workloads the input rate varies from 240,000 tuples/sec to 480,000 tuples/sec. For the experiments in this section the topologies are run for 2 hours. Table 6.1 shows the values of various parameters used by the algorithm and table 6.2 shows the values of the parameters used by RollSort topology. All the following experiments use these values unless specified otherwise.



(a) 11 workers (b) 12 workers

Figure 6.4: Workload D1 without *DataRateLB*

6.1.1 Effect on running topologies

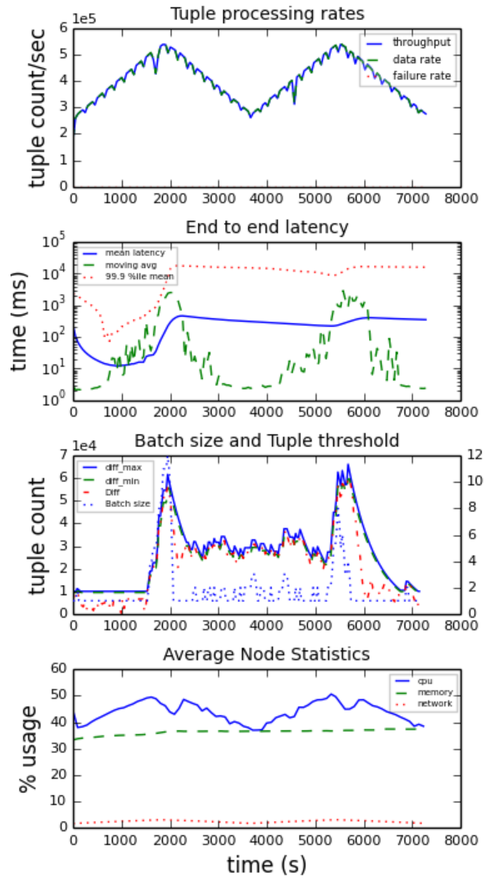
In this section we look at how *DataRateLB* works when applied to a running topology. For this purpose we first run the *RollSort* topology with workload D1 on 9 workers without *DataRateLB* enabled. Then we run the same topology with the same workload but with *DataRateLB* enabled. Figure 6.3a shows how the topology progresses without *DataRateLB*. At around 1500 seconds the system starts to become unstable and the throughput starts dropping drastically and reaches 0 around 1800 seconds. The failure rate of the tuples also starts to go up around the same time as when the throughput decreases. The job finally stops when the memory reaches the upper limit as seen in the node statistics plot. The latency behavior is in line with the observation in Section 3.3. The latency goes up when the system is unable to keep pace with the input data rate due to resource unavailability to process the tuples.

Figure 6.3b shows how the same topology behaves when *DataRateLB* is enabled. In the tuple processing rates plot, you can see that the throughput is able to keep up with the incoming tuple rate. The mean latency for the job varies from 320ms to 450ms and doesn't shoot up because the queuing time is limited due to batching when the input rates are high. The batch size increases to 16 during both the input rate peaks and comes down to 1 when the data rates come down and the system can now manage with a smaller batch size. The system remains stable as the memory usage is always less than 40%. As you can see the batch size update is only done once the value of *diff* goes above 10000 as configured before the job was run. This value of START_MAX can be set by the administrator to be small enough so that the latency doesn't go up due to queuing and the memory doesn't run out and large enough for the load balancer to be exercised less frequently. Such a scheme of increasing batch size to increase throughput and to keep the system stable is very useful for production jobs that do not have a sub-microsecond latency requirement thereby having a margin of latency large enough to apply batching.

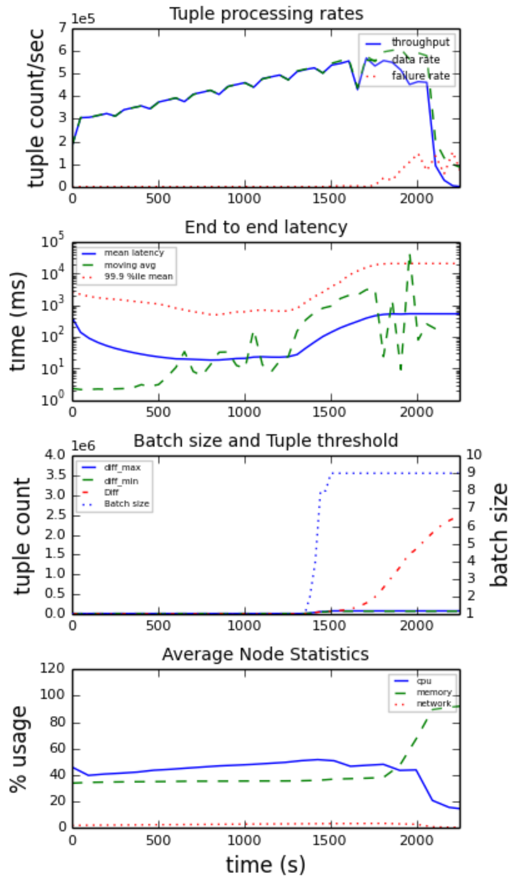
Now, to understand how much resources are needed to run the job without *DataRateLB*, we run the same topology on 10, 11 and 12 workers. The jobs fail on 10 and 11 workers but runs successfully on 12 workers. Figure 6.4 illustrates the behavior of the topology when run on 11 and 12 workers. As expected, the memory usage for the failed run reached 100% before failing and exiting. As compared to the 9 worker run, the throughput starts dropping around 2000 seconds for the 11 worker job where as the throughput started dropping around the 1500 seconds into the 9 worker job. The topology works when run on 12 workers with the memory usage almost constant around 30%. The significant advantage while running with 12 workers is the latency which stays at an average of 10ms for the 2 hour run. As discussed earlier, the batching approach is predominantly useful for higher order latency SLAs (>500 ms) than lower latency SLAs (<10ms).

6.1.2 Understanding the behavior under varying SLAs

DataRateLB avoids increasing the batch size when the SLA is not met. We look at the effect this decision has on the topologies under more strict latency SLAs. The run in Fig 6.3a had an SLA of 750ms. We run the same topology with 500ms and 350ms latency as the SLA to study the behavior of the topology. The job with 500ms latency SLA behaves similar to the one with 750ms SLA. During no time during the run the mean latency goes above 500ms thereby allowing the load-balancer to apply batch size updated during the complete life time of the topology. Fig 6.5a shows the execution of the 500ms latency job. It runs very similar to the 750ms latency job with the batch size increasing during high input data-rates. Here we also notice that there are some small batching (2-4 batch size) between the peaks. On the other hand, for the job with 350ms latency the load-balancer stops applying batch size updates after around 1500 seconds into the job because the latency shoots above 350ms.



(a) Workload D1 on 9 workers with *DataRateLB* and SLA 500ms



(b) Workload D1 on 9 workers with *DataRateLB* and SLA 350ms

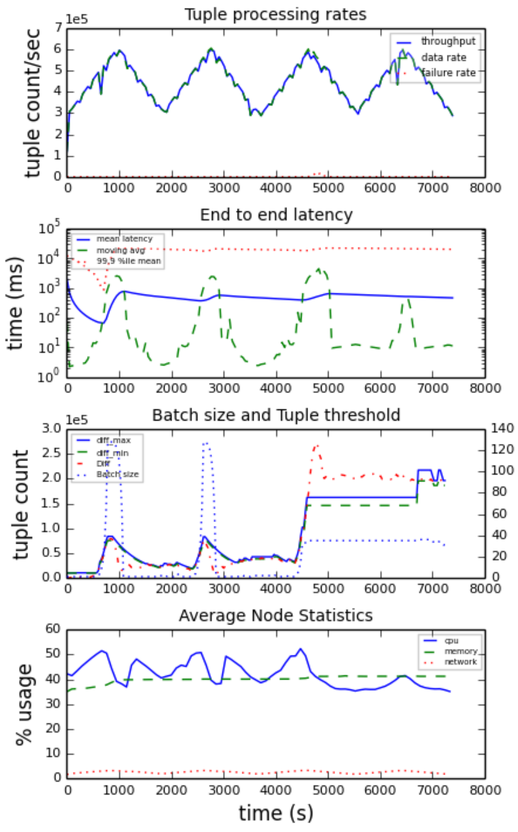
Figure 6.5: Workload D1 run under different SLAs

Once the algorithm is disabled, the *diff* goes up and the latency further increases due to additional queuing. Fig 6.5b shows the complete execution of the job. The job finally ends when the memory cannot hold all the pending tuples and crashes. A possible way to avoid this scenario is to spawn new workers and re-direct the load to these new workers. We wish to integrate this feature as future work. In this work we avoid any migration of existing PEs or creation of new PEs and look at only the use of ABF to find solutions to load balancing problems.

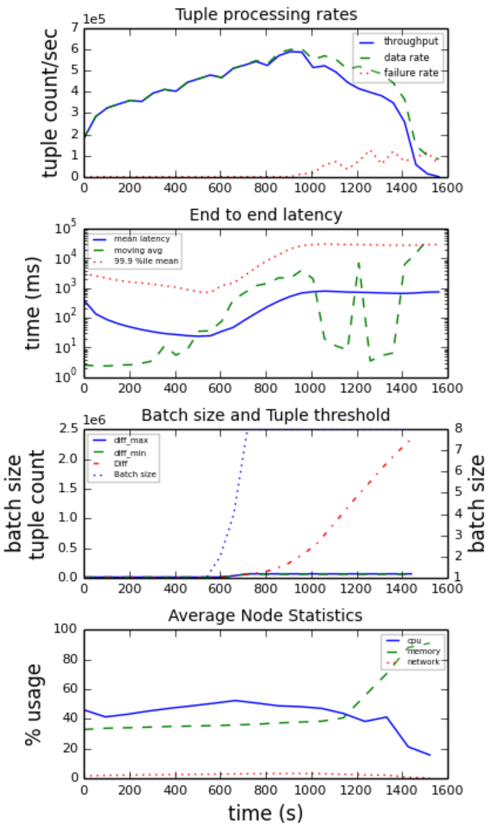
6.1.3 Significance of the load balancing interval

In this section we look at how the load-balancing interval impact the execution of topologies under *DataRateLB*. For the experiments in this section we run the D2 workload where the date-rate changes at a higher rate. We run the workload with period of 30secs and 60secs. Fig 6.6a shows the complete execution of topology with 30 second interval.

The throughput of the topology is able to keep up with the input data-rate but as you can see the batch size increases considerably more than the previous runs. The batch size goes up to 130 during peak data-rates which is the set MAX_BATCH value. The CPU and memory usage have as expected gone slightly higher when compared with the D1 workload. Now, when the same topology is run with an interval of 60 secs, the throughput starts to drop around 1000 seconds into execution. *DataRateLB* is unable to make adequate batch size adjustments to handle the fast increase in the data rate. By the time *DataRateLB* increases the batch size a considerable amount of tuples are already queued up for processing. In a few iterations of the algorithm this lag ends up consuming the available memory and the job crashes. From these results we understand that the load balancing interval should be in tune with the expected data rates. As future work, we wish to make the interval adaptive based on the value of *diff*. A higher value of *diff* could mean a lower *lb_period*.



(a) Workload D2 on 9 workers with interval of 30secs



(b) Workload D2 on 9 workers with interval of 60secs

Figure 6.6: Workload D2 with different intervals

CHAPTER 7

RELATED WORK

There have been limited work in the area of load balancing for stream processing systems as this is a newer field. In this section we go through some of the recent work in the load balancing and load distribution space in the context of distributed stream processing systems. Resource aware scheduling has been studied in a recent work by Peng et al. [22]. In this work, additional functionality is added to the scheduler to be aware of the current available resources in the cluster before scheduling the workers on specific nodes. In our work the load balancer is aware of the dynamic changes in the resource availability and hence is better equipped to make smarter choices than the more static scheduling done by R-storm.

Aniello et al. in their paper [23] proposes two schedulers for Storm. The first scheduler is used in an offline manner prior to executing the topology and the second scheduler is used in on online fashion to potentially reschedule after a topology has been running for a duration. The offline scheduler has the same disadvantages as R-Storm but the online scheduler incorporates dynamic changes to the resource availability. Our method is different because the rebalancing does not include migrating PEs to other nodes which can lead to downtimes or extra usage of resources during the migration to avoid any downtimes. The method of "The Power of Two Choices" (PoTC) [20] continuously defines two hash functions h1(x) and h2(x), such that each key x can be sent to one of two alternative downstream operator instances. Each operator instance tries to balance the amount of work sent downstream, such that all operator instances downstream receives an even workload. In our we work we build on this method but instead of using local estimation of load, our load balancer has a more global understanding of the cluster and hence can accommodate changes in resource usage due to the presence of other systems in the cluster. Additionally, our work also balances shuffle grouped components as well.

Das et al. in their work on stream processing using dynamic batching [12] proposes a micro-batch solution for throughput improvements. Here, a batch interval is selected and the incoming data is batched together and fed to the Spark batch processing system. Since, the batch size is small, the performance can be similar to steaming systems, but with higher latency. The interval can change depending in the data-rate. In our work, there is no micro-batching involved since the batching is not done to the incoming data, instead batching is done at individual PEIs. In [12] once the data is batched, the batch remains as one data entity throughout the life cycle of the program. So, a tuple will always be part of the same batch whereas in our case the tuple is only temporarily batched at each PEI giving flexibility to do more adaptive configuration of the data movement. In [24], Gulisano et al. propose a system called StreamCloud in which they describe a high scalable and elastic stream engine with novel parallelization approaches and an effective resource management approach. The resource management is based on provisioning and decommissioning instances leading to smaller overheads.

CHAPTER 8

CONCLUSION

In this thesis, we have presented a novel framework for dynamically adapting the batch size and batch interval in stream processing systems. We then implemented a data-rate aware algorithm which with minimal workload specific configuration is able to adapt to variations in input data rates. As part of this work we also studied in detail the effect of batching on performance of the system under various operating conditions. We additionally proposed future work of implementing more advanced load-balancing algorithms using the adaptive batching framework.

REFERENCES

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl et al., "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[3] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud CComputing*, 2012.

[4] N. Ramesh, "Apache Samza, LinkedIns Framework for Stream Processing," http://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/, 2015, [Online; accessed 29-Aug-2016].

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth et al., "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[7] Y. Ben-Haim and E. Tom-Tov, "A streaming parallel decision tree algorithm," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 849–872, 2010.

[8] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 4, p. 26, 2010.

[9] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Society Press, 1995.

[10] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic duo distribution in the borealis stream processor," in *21st International Conference on Data Engineering (ICDE'05).* IEEE, 2005, pp. 791–802.

[11] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations.* Army High Performance Computing Research Center, 2000.

[12] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing.* ACM, 2014, pp. 1–13.

[13] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu, "Performance optimization for distributed intra-node-parallel streaming systems," in *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on.* IEEE, 2013, pp. 62–69.

[14] T. Akidau, A. Balikov, K. Bekiroğlu, 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*, vol. 6, no. 11, pp. 1033–1044, 2013.

[15] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: fault-tolerant and scalable joining of continuous data streams," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* ACM, 2013, pp. 577–588.

[16] "Apache Flink, Scalable Batch and Stream Data Processing," https://flink.apache.org/, [Online; accessed 29-Aug-2016].

[17] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[18] "LMAX Disruptor: High Performance Inter-Thread Messaging Library," https://lmax-exchange.github.io/disruptor/, [Online; accessed 29-Aug-2016].

[19] "Batching in DisruptorQueue," https://github.com/apache/storm/pull/765, [Online; accessed 29-Aug-2016].

[20] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *2015 IEEE 31st International Conference on Data Engineering.* IEEE, 2015, pp. 137–148.

[21] "Storm Benchmark," https://github.com/intel-hadoop/storm-benchmark, [Online; accessed 29-Aug-2016].

[22] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference.* ACM, 2015, pp. 149–161.

[23] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems.* ACM, 2013, pp. 207–218.

[24] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, "Streamcloud: A large scale data streaming system," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on.* IEEE, 2010, pp. 126–137.