Leron : Self-Mitigating and Load Balancing in Stream Processing for Twitter Heron

Jialin Liu Xiaofo Yu Kaishen Wang

Abstract

Twitter Heron is one of the most popular stream processing distributed systems at scale. It makes many innovations and introduces the idea of backpressure, which dynamically adjust the rate at which data flows through the topology. However, we believe that such backpressure mechanism is not optimal and may not fully explore the computing ability of each HI (Heron Instance) for all time. Thus, we build Leron, whose bolts are versatile and are capable of helping each other. Experiments show that Leron significantly improved the efficiency and utilization of computing resources (by up to 40% in throughput) compared to Heron.

1 Introduction

Twitter Heron employs a backpressure mechanism, which is a state-of-art technique for distributed streaming systems, to dynamically adjust the rate at which data flows through the topology. This mechanism is important in topologies where different components can execute at different speeds (and the speed of processing in each component can change over time). For example, consider a pipeline of work in which the later/downstream stages are running slow, or have slowed down due to data or execution skew. In this case, if the earlier/upstream stages do not slow down, it will lead to buffers building up long queues, or result in the system dropping tuples. If tuples are dropped mid-stream, then there is a potential loss in efficiency as the computation already incurred for those tuples is wasted. A backpressure mechanism is needed to slow down the earlier stages. [6] Heron implements it using the approach of Spout Backpressure and finds that it has good performance.

When a topology is in backpressure mode, it goes as fast as the slowest component.[6] In other words, at this time, the computing ability of all the other components are not fully utilized.

Our motivation is mainly from the backpressure mechanism in Heron. There can be a lot of different causes that bring the entire Heron topology into backpressure mode, including but not limited to, long logical or physical plans, insufficient resources allocated for instances, non-uniform random distribution of tuples, stragglers and data skew from the spout. Specifically, such backpressure mechanism does not work well to fully exploit resources in a lot of different scenarios.

Dhalion[5] is a self-regulating framework built on top on Heron system which borrows ideas from control theory to self-tune the workload structure by detector feedbacks. However, the rescheduling and reallocation process is time consuming and would be an overkill for the circumstances where slight backpres-

sures happen. At the same time, Dhalion is not suitable for frequent changes in data stream which could cause backpressure on different bolts overtime.

In this paper, we present Leron, which deals with these circumstances as a supplement for Dhalion. Leron intends to improve the elasticity and robustness of Twitter Heron system by enabling busy bolts get help from upstream ones and reducing the times of backpressure happens. There's no previous work (Flink, Storm or Heron) that uses similar smart technique which enables different botls helping each other. Such smart technique should reduce the number of backpressure (when Heron system is limited by its slowest component) and utilize all the computing and storing resources more effectively. Consider the scenarios where the additional work for some bolts which should trigger backpressures in original Heron system can be completed by bolts from upstreams. As a result, it is possible to eliminate the backpressures in such scenarios. Further more, bolts in Leron can anticipate which bolts are likely to have backpressure by learning previous experience and help doing jobs for those potential busy bolts during normal time. Such automatic load balancing mechansim can further reduce the times of backpressure happens. In this paper, we present the full design and implementation of Leron, and show that Leron can improve the throughput of Heron by 0% (no backpressure) to 40% (with backpressure) in different circumsatnces.

2 Related Work

2.1 Stream Processing

Stream processing aims to process the data stream in real time. Stream processing is studied even before large scale batch processing comes into attention. Some database systems, including Aurora[2] and TelegraphCQ[4] process streaming data on single machine.

Since the input event data rate in production can be intolerable for a single machine to handle, many researches has focused on large-scale distributed stream processing systems. Two computation models for stream processing frameworks has emerged in recent years. Systems like Storm[8], Heron[6] and Flink[3] use continuous operator model as their building block. This model converts user application code into a DAG of operators, with each operators deployed on one or more processors to handle incoming data tuples.

2.2 Cluster Scheduling

In order to maximize the throughput in the stream processing systems, some research work has been done to optimize resource usage and avoid unnecessary cost. StroMAX[7] uses graph-partitioning based approach to schedule streaming workload during job submission and re-balance workload in the middle of execution. Twitter Heron uses Aurora [1], one dominant cluster management system in use. Dhalion[5] is a self-regulating framework built on top on Heron system which borrows ideas from control theory to self-tune the workload structure by detector feedbacks.

2.3 Overview of Heron Architecture

In Heron, a topology will have spouts and bolts which are sources of input data and computation units on the stream. Each topology is run as an Aurora job consisting of several containers, as shown in Figure 1. The first container runs a process called the **Topology Master**. The remaining containers each run a **Stream Manager**, a Metrics Manager, and a number of processes called **Heron Instances** (spouts/bolts that execute logic submitted by users). Multiple

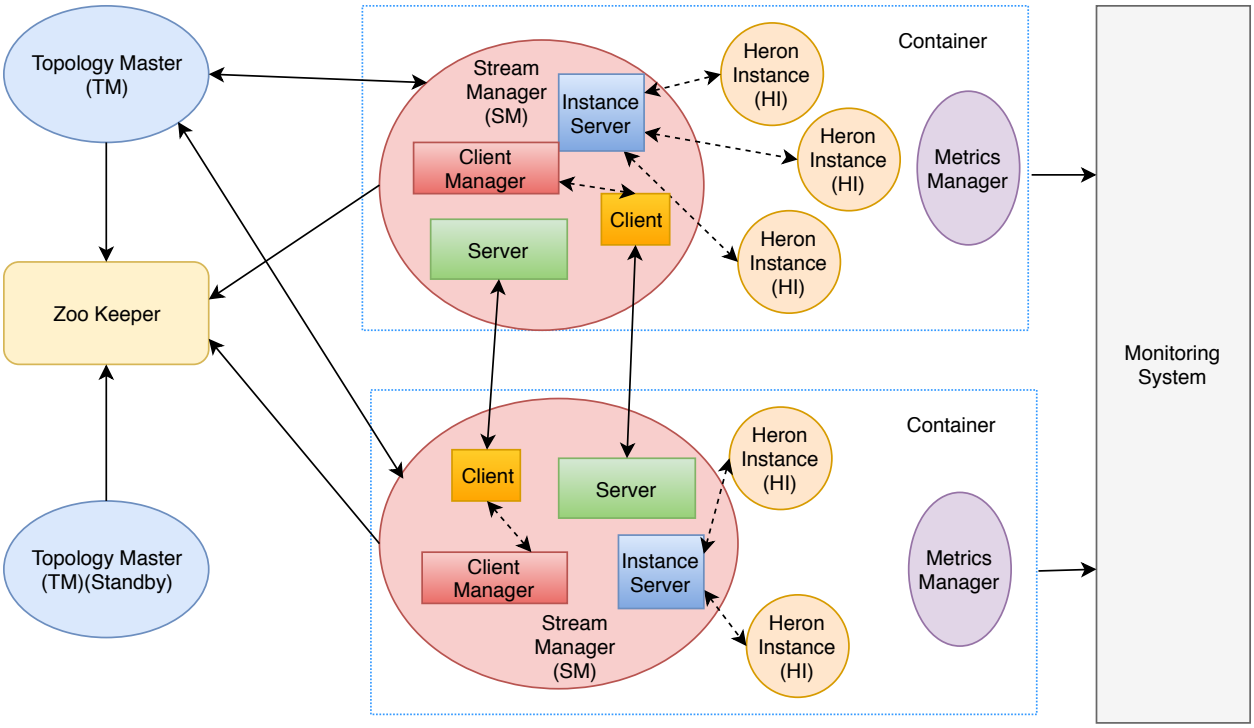


Figure 1: Heron Architecture

containers can be launched on a single physical node. These containers are allocated and scheduled by Aurora based on the resource availability across the nodes in the cluster. There will also be a standby Topology Master to ensure availability. The metadata of the topology (which includes information about the user who launched the job, the time of launch, and the execution details) are kept in Zookeeper.[6]

Each Stream Manager includes several important components. There is an **Instance Server**, which are responsible for communicating with different Heron Instances, as well as sending data to and receiving data from all the Instances belong to the Stream Manager. There is also a **Server**, a **Client Manager** and several **Client**s. The Client Manager manages different Clients, each of which is connected to the Server belongs to another Stream Manager. Server, Client Manager and Clients as a whole enable message and data delivery between different Stream Managers.

2.4 Congestion Control

In stream processing systems, congestion often happens because different components in topologies can execute at different speeds and the speed of processing in different components can change overtime. Heron proposed a backpressure mechanism to adjust the rate at which data flows through the topology to address the congestion problem.[6]

Specifically, **TCP Backpressure**, **Spout Backpressure** and **Stage-by-Stage Backpressure** are proposed for the Heron architecture(The latter two mechanisms work in conjunction with the first one). In Heron the Spout Backpressure mechanism is implemented. The backpressure mode is triggered when the size of application-level buffer reaches the high water mark. When the topology is in the backpressure mode, everything goes as fast as the slowest component in the topology. In this mode, tuples emitted from the spouts don't get dropped for slowness except machine failures.

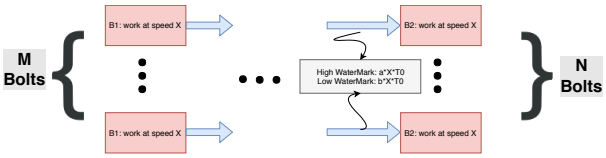


Figure 2: Simulation

However, if the slow persists for too long, data would build up in the upstream queues where spouts are reading data from. The backpressure ends when the size of application-level buffer reaches the low water mark.

3 Explanation and Simulation of Idea

In order to illustrate the soundness and validity of the helping mechanism from a theoretical perspective, we will present a simulation of stream processing flow using a very basic example (Figure 2). Consider there are M instances of type B_1 bolt and N instances of type B_2 bolts where M is greater than N. Assume all the bolt instances have the same capability of processing X tuples of data in a unit of time T_0.

Although it is possible to reallocate the computing resources more effectively, optimal resource allocation is not guaranteed to work in all cases. In a very naive example where $M = 2$ and $N = 1$, it's easy to see that a single bolt instance is not partitionable. Generally speaking, properties of input data stream are constantly changing over time. Thus, the speed of processing data in different bolts could change and the bottleneck of the whole system could change, which makes frequent reallocation of resources impossible.

The work can be done by all B_1 bolts in time T_0 is MXT_0. The work can be done by all B_2 bolts in time T_0 is NXT_0. As a result, there will be $\frac{MX-NX}{N}$ work that cannot be processed by each B_2 bolt in time T_0. The High Water Mark for the incoming queue is aXT_0 and the Low Water Mark is bXT_0. Without loss of generality, we assume the work starts when queue size is at low watermark. Then the work to trigger backpressure for B_2 bolts is $(a-b)XT_0$ and it will take $(a-b)XT_0\frac{N}{(MX-NX)} = \frac{(a-b)NT_0}{(M-N)}$ time for backpressure to happen. After that, B_2 bolts will continue processing the data in queue and it should take $\frac{(a-b)XT_0}{X} = (a-b)T_0$ time to reach Low Water Mark again, which will make the backpressure stop. So the total time for this period cycle is $\frac{(a-b)NT_0}{(M-N)} + (a-b)T_0 = (a-b)T_0\frac{M}{M-N}$. During the period, all the B_2 bolts keeps working at maximum speed. The total amount of data can be processed in the period is $(a-b)XT_0\frac{M}{M-N}$. Consider the scenario where B_1 bolts can help with B_2 bolts during backpressure. During the $(a-b)T_0$ time, all B_1 bolts will spend half of their computing resources working on B_2 bolts' jobs. B_1 bolts will instead work at speed $\frac{X}{2}$ instead. During the backpressure time, there will be additional $\frac{(a-b)T_0MX}{2}$ data processed. In summary, the helping mechanism can improve the total efficiency by $\frac{\frac{(a-b)T_0MX}{2}}{(a-b)XT_0\frac{M}{M-N}} = \frac{M-N}{2} * 100\%$.

However, this is the ideal case. In fact, the capacity of incoming queues for type B_1 bolts are also limited. Backpressure are likely to happen on B_1 bolts while they are helping with B_2 bolts. On one hand, upstream bolts of B_1 bolts will stop working during backpressure and becomes a waste of computing resources. On the other hand, if the incoming queue size of B_2 bolts are very close to low weater mark when backpressure on B_1 bolts happens, B_2 bolts may consume all the data tuples in their incoming queues before B_1 bolts produce new data tuples for them to execute. The waiting time for B_2 bolts is also a waste of computing resources. In order to further deal with these inefficiencies, we introduce a new self-mitigating algorithm. Bolts will dynamic anticipate if their downstream bolts need help from learning past experience, and adjust a ratio. Based on the ratio, Bolts will choose some data tuples and execute

them with the logic of itself and its downstream bolts. By dynamically adjusting the ratio and load balancing the workload, Leron could further decrease the number of real backpressure happens.

4 Leron Design

4.1 Efficient Backpressure Mechanism

We introduce a new term, Leron Backpressure (written as **LB**), to describe the following situation: There are backpressures on some bolts, but those bolts get help from their upstream bolts. Thus, there are no spouts/bolts stop working during LB. LB can be a pre-stage for real backpressure, although in some ideal cases, it is not necessary for real backpressure to happen. If the system fails to find available bolts to help the busy bolts under backpressure, it will stop LB (if there are any) and enter the normal backpressure stage. Besides the normal broadcasting backpressure message and stop spouts working, the system will also ask all the bolts which are doing the helper work (if there are any) to stop help.

4.2 Roles for Different Components

In this section, we will briefly discuss the roles for stream manager and bolt instance.

Stream managers are responsible for starting and stopping LB. If it observes any backpressure, it will try to find any available upstream bolts in the same container. There is a reason why it is necessary to have at least one available bolt in the same container. Because the stream manager cannot know whether the remote upstream bolts in other containers are under their own LB, which makes those bolts unable to offer help. It will be unrealistic to wait for such information as backpressure has already happened. In consequence, it is necessary to have a local upstream bolt available (not under its own LB) to guarantee that busy bolts can obtain help. If such condition is true, stream manager will start LB by asking local upstream bolts to help. It will also broadcast this message to other stream managers and ask upstream bolts in other containers to help. The reason why broadcasting is necessary is that, the incoming queues for busy bolts (which are under backpressure) are running out of room. If the upstream bolts from other containers are not aware of this, they will keep sending data tuples to those busy bolts under backpressure. As a result, many data tuples are likely to be dropped.

There are two cases that can trigger a LB to stop. The first is the stream manager observes backpressure on a bolt which is currently helping other bolts. On one hand, we don't want to have complex dependencies of helping where all the bolts are doing jobs for their downstream bolts. On the other hand, downstream bolts of that busy bolt are also busy and are incapable of receiving and executing data tuples from two layers ahead under their own LB. So the stream manager will stop LB for this case. The second case is the connections which were under backpressure have reached the condition to stop the backpressure (the size of incoming queue reaches low water mark). As there are no more connections under backpressure, stream manager will stop the LB. As stated in the previous paragraph, in order to stop an LB, the stream manager will ask all the bolts which are helping that LB to stop helping, including notifying local bolts though Instance Server and broadcasting to remote bolts though Clients.

For the bolt instance, it will execute both its own logic and the logic of its downstream bolts if it received the"asking for help" message. In other time, the bolt will anticipate whether there will backpressure happen on its downstream bolts and how much work it should complete for the downstream bolts in order to minimize the times of LB and real backpressure

on both itself and its downstream bolts. Details about this will be illustrated in the next subsection.

4.3 Load Balancing Mechanism Design

In this section, we will describe a naive load balancing function that tries to mitigate imbalanced loads among different stages in a topology. As described in the simulation, it is often impossible for upstream bolts to help downstream bolts to do all the computation because upstream bolts will also trigger backpressure due to the burst of heavy loads. However, in many cases downstream bolts that become bottleneck can share some workload with upstream bolts that have not fully exploited their computational capabilities. Therefore, a load balancing mechanism is naturally suitable with the new helpful LB mechanism we introduced earlier.

The basic idea of the load balancing mechanism is pretty simple. Upstream bolts will automatically help their downstream bolts by working on a percentage of data items and leaving the rest as usual for the downstream bolts. Such percentage is defined using the term **ratio** or r and it is shared among all bolts in the same unit in logical plan. Then, there needs to be a function that modifies this ration according to different real time events, which are basically trigger/stop of backpressure in our system.

There are several limitations and constraints need to be addressed by the load balancing function that is responsible for adjusting the ratio. First of all, the initial value of the ratio should be 0 since in the beginning it is assumed that both the designated logical and physical plan should work well with the anticipated workload. On the other hand, the load balancing function should be able to find out an optimal value of the ratio r if there exists one. In the end, the load balancing function should also be adaptive to changes because cluster and data stream properties are constantly changing and so does the optimal value of the ratio r. Thus, we propose the following function to address these basic concerns about load balancing.

$$r = \begin{cases} r + (1-r) * \Delta & \text{observe a help request} \\ r - r * \Delta & \text{observe a self bp} \\ r & \text{otherwise} \end{cases} \tag{1}$$

In this function, $r = 0$ when bolt instances are initialized and Δ is a parameter that controls the size of step of learning the value of r. The main idea is that the function will increment the value of r if downstream bolts still need more help and decrease r when the bolt already gets too much workloads. If both bolts do not invoke backpressure and need help any more, then r will not change because it is considered as an optimal value.

It is clear that $0 \leq r < 1$ and r is always updated towards optimal values. We can throw some light on its proof. If r is less than the optimal value, it keeps receiving the "asking for help" message and thus $r + (1-r) * \Delta$ will be invoked. It can be viewed as a recurrence function: $r_n = r_{n-1} + (1 - r_{n-1}) * \Delta$. The closed form solution of the recurrence function is $r_n = (c_1 + \Delta - 1) * (1-\Delta)^{n-1} + 1$ where c_1 is a constant paramter. As $0 < \Delta < 1$, $0 < 1 - \Delta < 1$ holds, $(c_1 + \Delta - 1) * (1-\Delta)^{n-1}$ converges to 0 if n increases and r_n converges to 1. If r is less than the optimal value, it will keep increasing and converge to 1. As the optimal value is less than 1, so r will increase in this case. At the same time, one can easily prove that r will decrease when it is greater than the optimal value. This property ensures that the load balancing function can adapt to changes of the optimal values of r when either data stream or cluster properties change and r can be adjusted using the function accordingly.

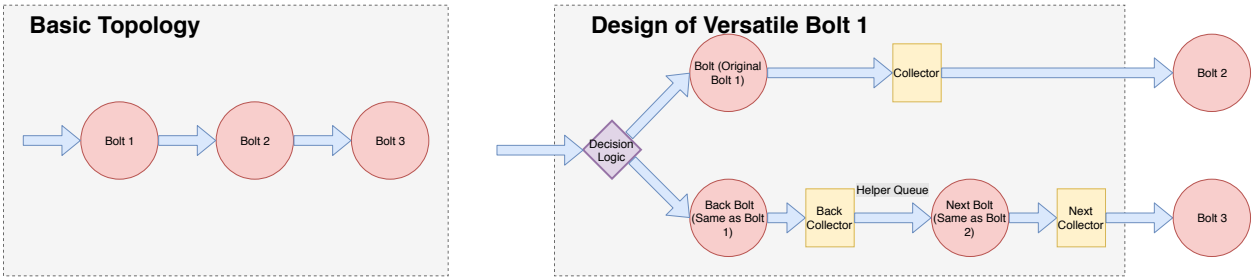


Figure 3: Versatile Bolt Design

5 Implementation

Leron is implemented in around 2000 lines of code in Java, C++ and gRPC[1] Protobuf. It merely changes any APIs of Heron and the existing metrics for Heron also work well on Leron. We followed Twitter Heron's design principle and exploited gRPC Protobuf's extensibility and easy language interoperability as a medium of backpressure message communication across containers and instances in a distributed Leron Topology. We will discuss about the implementation of versatile bolt as it is an important point in our implementation choices.

5.1 Versatile Bolt

One major concern in implementing the versatile bolt is how to transit between different status smoothly. When the bolt receives the "asking for help" message, it should start helping as soon as possible. However, there are some data tuples remaining in the outgoing queues to its downstream bolts. The same situation also applies when the bolt receives "stop helping" message as there are some remaining data tuples which have already been executed with the logic of its downstream bolts. In addition, as we want to have the versatile bolts keep helping with their downstream bolts during normal time, we need to have an efficient implementation that enables the versatile deals with two kinds of work (one is its own logic, another is its own logic and the logic from its' downstream bolts). We will use a simple example to help elaborate this.

As shown in Figure 3, part of the topology is composed by three bolts (Bolt 1, Bolt 2 and Bolt 3), whose relationship are obvious in the figure. For Bolt 1, Heron only implements Bolt, which will execute the data tuple by the logic of Bolt 1, and Collector, which is a transition container to deliver data tuples to Bolt 2. In Leron, we keep this part and add another pipeline. The new pipeline has a Back Bolt which is identical to the Bolt, and will output the data tuples into a Back Collector, which is followed by a queue called Helping Queue. There is another Next Bolt in the new pipeline which will try to get all the data tuples in the Helping Queue and execute them by the logic of Bolt 2. After executed by Next Bolt, data tuples will be delivered to another component called Next Collector, which targets at Bolt 3. We can see that, the new pipeline completes both the logic of Bolt 1 and Bolt 2, and output the data tuples directly to Bolt 3. The advantage of this implementation is that, we only need to make a decision about which component (Bolt or Back Bolt) that a data tupleset should enter when it comes. We no longer need to worry about transitions between different stages as the versatile bolt will executes and conveys the data tuples in all the components including Bolt, Collector, Back Bolt, Back Collector, Helping Queue, Next Bolt and Next Collector.

[1] https://grpc.io/

The reason that we don't invent a super bolt that does all the logic and get rid of the back collector and helping queue is that we don't want to change the APIs of a bolt instance. The existing APIs of a bolt instance limits that it can only read from a queue, output to a collector and execute one logic from topology.

6 Experiments

In this section we experimentally verify the performance of Leron. We will first introduce the setting of our experiments, after that we introduce the topology we use in experiments and discuss the meaning of choosing this topology. Finally, we compare with baseline (Heron) to show the effectiveness of our approach.

6.1 Experiment Setting

As introduced in the section 5, we implemented Leron for evaluation, and to compare with the original version, we use the Apache Heron 0.17.8 as baseline. We use Amazon EC2 as our experiment platform to perform the whole experiment since it provides a large variety of instance types and lower monetary cost compared to Microsoft Azure cloud. We deployed a small cluster with 3 nodes due to the limited budget. The cluster is consist of 1 m4.xlarge node and 2 m4.2xlarge nodes. The m4.xlarge instance contains 4 virtual CPU cores and m4.2xlarge instance contains 8 virtual CPU cores. All the instances use Intel Xeon E5-2686 v4 CPU and 16 Gigabyte memory.

We use Apache Aurora as the Heron scheduler and Zookeeper as the state manager. The Apache Aurora scheduler is deployed on the m4.xlarge node as the master node and the rest 2 m4.2xlarge nodes serve as slaves containing multiple containers. Deploying the Aurora scheduler on a single machine can avoid the performance downgrade on worker containers. Here we use m4.2xlarge as worker instance type rather than more m4.xlarge instances because it can provide more flexibility on the choice of the size of a single Heron container.

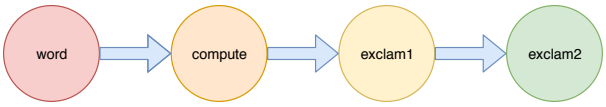


Figure 4: Experiment Topology

6.2 Topology Setting

The original Heron paper[6] uses the "WordCount" topology as the experiment topology. The idea of this topology is to generate random fix-length words from the Spout and count the appearance of each word in Bolt. The bolt use in-memory map structure to keep the record of each word. To maintain the accurate count of each word, the topology uses fields grouping (hash grouping on specific field) to distribute the word generated from the Spout.

We claim that this topology is not suitable for our evaluation for the following two reasons: Firstly, the topology uses in-memory key value map to maintain the state, while our optimization is designed for tackling the performance issue in stateless topology. Secondly, the topology contains only one Spout and one Bolt in logical plan, which is too simple to apply our optimization technique.

To resolve this, we propose a multi-stage topology containing different bolt logic and various parallelism levels, as shown in Figure 4. It starts with "word" spout, and connects the "compute" bolt, following two "exclam" bolt marked as "exclam1" and "exclam2" respectively. The "word" spout generates random string that may contain '!'. The "compute" bolt counts the number of appearance of '!' in the string, simulating the real-world logic with stateless computation such as model inference in machine learning systems. The "exclam1" and "exclam2" bolts append a fixed number of

'!' at the end of the incoming strings and output new ones. We can compare the total count of executed tuples by "exclam2" bolts and "compute" bolts to make sure no tuple is missing during the execution. As "compute" bolt has heavier workload compared to "exclam" ones, we set the parallelism of "compute" bolt to 2 and parallelism of "exclam" bolts to 1. However, with such parallelism setting, "exclam1" component cannot work as fast as "compute" component and will suffer from backpressure. Such unbalance workload is deliberate as we want to trigger backpressure repeatedly.

6.3 Bolt Performance Measurement

In this subsection, we tried to measure the performance of different bolt tasks. Having the statistics of these bolts will help us understand the behavior and performance difference between Heron and Leron. In order to record the statistic, we add the logging command in the each bolt logic to collect the speed of each bolt logic. We extract logs from each bolt and compute the average number of processed tuple count per microsecond. The result is shown in Table 1.

bolt	# of tuples/ms
compute	113.76
exclam1	146.58
compute(LB)	86.78

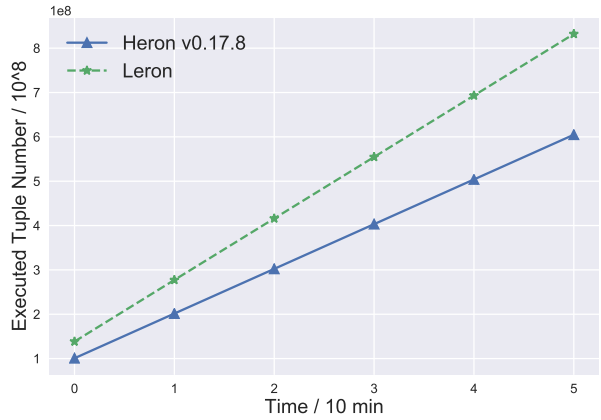
Table 1: Processing speed of each bolt logic

From the table we can see that a single "compute" bolt is slower than "exclam1" bolt, but two "compute" bolts will process more tuples than downstream "exclam1" bolt at the same time. Also, when "compute" bolt is turned into LB mode, the processing speed can further decrease, but we must be aware that in helper mode the tuples have been processed by two consecutive bolt logic and will be sent to the downstream of "exclam1". This setting is consistent with the parallelism setting described in the last subsection as the "compute" bolts will be slightly faster than "exclam1" bolt, and will eventually fill up the input queue of "exclam1" bolt. We expect that backpressure will be triggered in the original Heron version while in Leron implementation we will use load balancing technique to shift the workload to other bolts and ease the pressure.

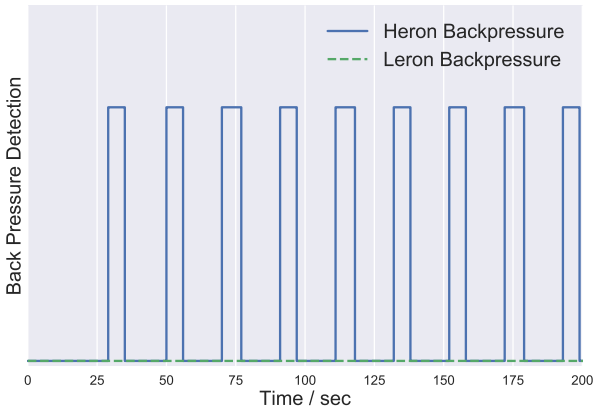
6.4 Experiment Result

We run Heron and Leron using previous mentioned topology on our cluster. We run each implementation for more than one hour and measure the statistics for each version every ten minutes. We first explore the cumulative count of execution tuples for the whole topology with the length of one hour, the result is shown in Figure 5(a). From the figure we can tell that the Leron is about 1.4 times faster than the original Heron implementation. This is because the Heron version will enter back pressure and clamp down the spouts when input queue of "exclam1" bolt is filled up, and the "compute" bolts will stop processing when the back pressure time is long enough.

To further explore this, we plot the back pressure time of Heron during a period of execution(about 200 seconds) using data from the log of stream manager. The result is shown in Figure 5(b). From the figure we can see that the back pressure is detected routinely by stream manager, and it takes up about 30% of total time to resolve back pressure. While in the Leron implementation, when a full input queue problem is detected by the stream manager, it sends out help request to upstream bolts to let them help working on downstream logic and bypass downstream bolt. From the same figure we can tell that no back pressure is trig-



(a) Throughput of Heron and Leron



(b) Backpressure Detection in Heron and Leron

Figure 5: Performance of Leron VS Heron

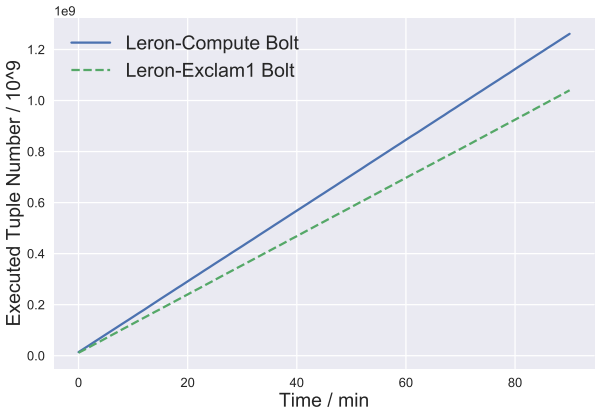


Figure 6: Bolt Throughput Difference in Leron Topology

gered during the execution. Moreover, Figure 6 shows difference of total tuples executed in "compute" bolts and "exclam1" bolt collected from metric manager at the length of one hour during execution. From the figure we can see that, during LB phase, "compute" bolts process more tuples, they bypass the "exclam1" bolt and are sent to "exclam2" bolt directly.

In addition, we compared Leron with Heron by running some topologies that will not trigger any backpressure on them. The experimental results show that Leron can achieve the same performance as Heron does in these circumstances.

7 Future Work

Although we experimentally verify that LB can help avoid some backpressures and improve the total throughput, there are still some more fields to explore in the current Heron architecture design.

From the design we can see that Leron can only ask upstream bolts who is directly to the target bolt, which refers to the direct ancestor bolt(s) of target bolt in the logical plan. This will limit the power of helping logic since upstream bolts may not have extra resources to compute downstream logic without sacrifices its own performance, while bolts in other part of this topology might have more extra computation power. To solve this, each instance will need to have all the bolt logics (executables) in the topology, and ready to execute extra bolt logic when needed. This requires stream managers to have more knowledge on the state of each instance and have on-time sync up to exchange local states. We refers this as "tuple flow redirection" and is one step further from our current implementation.

The other potential idea is to have flexibility over topology representation. Currently, the Heron community proposed a new declarative language to represent the Heron topol-

ogy. This declarative language will make the user to apply their idea faster and let the backend framework have extra space to optimize topology performance, which is of the similar role as SQL language in database system. Since this language is still in beta and may have unknown errors during topology translation phase, we didn't optimize over this but use on original topology API written purely in JAVA language. When translating this topology, an optimization can be applied: Compress some simple consecutive bolt tasks into a single bolt containing multiple step when possible. This will definitely improve performance of topology in some situations since it save multiple tuple passing phases over networks. We can assign different cost values on different logic and network overhead and transform the topology into a (dynamic) linear programming problem that aims to maximize throughput.

References

[1] Apache aurora. http://aurora.incubator.apache.org/.

[2] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: a new model and architecture for data stream management. *The VLDB Journal 12*, 2 (Aug 2003), 120–139.

[3] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[4] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 668–668.

[5] FLORATOU, A., AGRAWAL, A., GRAHAM, B., RAO, S., AND RAMASAMY, K. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1825–1836.

[6] HERON, M. J., AND BELFORD, P. Ethics in context: A scandal in academia. *SIGCAS Comput. Soc. 44*, 2 (July 2014), 20–51.

[7] JIANG, J., ZHANG, Z., CUI, B., TONG, Y., AND XU, N. Stromax: Partitioning-based scheduler for real-time stream processing system. In *Database Systems for Advanced Applications* (Cham, 2017), S. Candan, L. Chen, T. B. Pedersen, L. Chang, and W. Hua, Eds., Springer International Publishing, pp. 269–288.

[8] 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., AND RYABOY, D. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.