Streaming@Twitter

Maosong Fu, Sailesh Mittal, Vikas Kedigehalli, Karthik Ramasamy, Michael Barry, Andrew Jorgensen, Christopher Kellogg, Neng Lu, Bill Graham, Jingwei Wu

Twitter, Inc.

Abstract

Twitter generates tens of billions of events per hour when users interact with it. Analyzing these events to surface relevant content and to derive insights in real-time is a challenge. To address this, we developed Heron, a new real time distributed streaming engine. In this paper, we first describe the design goals of Heron and show how the Heron architecture achieves task isolation and resource reservation to ease debugging, troubleshooting, and seamless use of shared cluster infrastructure with other critical Twitter services. We subsequently explore how a topology self adjusts using back pressure so that the pace of the topology goes as its slowest component. Finally, we outline how Heron implements at-most-once and at-least-once semantics and we describe a few operational stories based on running Heron in production.

1 Introduction

Stream-processing platforms enable enterprises to extract business value from data in motion, similar to batch processing platforms that facilitated the same with data at rest [42]. The goal of stream processing is to enable real-time or near real-time decision making by providing capabilities to inspect, correlate and analyze data as it flows through data-processing pipelines. There is an emerging trend to transition from predominant batch analytics to streaming analytics driven by a combination of increased data collection in real-time and the need to make decisions instantly. Several scenarios in different industries require stream processing capabilities that can process millions and even hundreds of millions of events per second. Twitter is no exception.

Twitter is synonymous with real-time. When a user tweets, his or her tweet can reach millions of users instantly. Twitter users post several hundred million tweets every day. These tweets vary in diversity of content [28] including but not limited to news, pass along (information or URL sharing), status updates (daily chatter), and real-time conversations surrounding events such as the Super Bowl, and the Oscars. Due to the volume and variety of tweets, it is necessary to surface relevant content in the form of break-out moments and trending #hashtags to users in real time. In addition, there are several real-time use cases including but not limited to analyzing user engagements, extract/transform/load (ETL), and model building.

In order to power the aforementioned crucial use cases, Twitter developed an entirely new real-time distributed stream-processing engine called Heron. Heron is designed to provide

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- **Ease of Development and Troubleshooting**: Users can easily debug and identify the issues in their topologies (also called standing queries), allowing them to iterate quickly during development. This improvement in visibility is possible because of the fundamental change in architecture in Heron from thread based to process based. Users can easily reason about how their topologies work, and profile and debug their components in isolation.
- **Efficiency and Performance**: Heron is 2-5x more efficient than Storm [40]. This improvement resulted in significant cost savings for Twitter both in capital and operational expenditures.
- **Scalability and Reliability**: Heron is highly scalable both in the ability to execute large numbers of components for each topology and the ability to launch and track large numbers of topologies. This large scale results from the clean separation of topology scheduling and monitoring.
- **Compatibility with Storm**: Heron is API compatible with Storm and hence no code change is required for migration.
- **Simplified and Responsive UI**: The Heron UI gives a visual overview of each topology. The UI uses metrics to show at a glance where the hot spots are and provides detailed counters for tracking progress and troubleshooting.
- **Capacity Allocation and Management**: Users can take a topology from development to production in a shared-cluster infrastructure instantly, since Heron runs as yet another framework of the scheduler that manages capacity allocation.

The remainder of this paper is organized as follows. Section 2 presents related work on streaming systems. The following section, Section 3 describes the Heron data model. Section 4 describes the Heron architecture followed by how the architecture meets the design goals in Section 5. Section 6 discusses some of the operational aspects that we encountered while running Heron at Twitter specifically back-pressure issues in Section 6.1, load shedding in Section 6.2, and Kestrel spout issues in Section 6.3. Finally, Section 7 contains our conclusions and points to a few directions for future work.

2 Related Work

The importance of stream-processing systems was recognized in the late 1990s and early 2000s. From then on, these systems have gone through three generations of evolution. First-generation systems were either main-memory database systems or rule engines that evaluate rules expressed as condition-action pairs when new events arrive. When a rule is triggered, it might produce alerts or modify the internal state, which could trigger other rules. These systems were limited in functionality and also did not scale with large-data-volume streams. Some of the systems in this generation include HiPAC [29], Starburst [43], Postgres [37], Ode [31], and NiagaraCQ [27].

Second-generation systems were focused on extending SQL for processing streams by exploiting the similarities between a stream and a relation. A stream is considered as an instantaneous relation [22] and streams can be processed using relational operators. Furthermore, the stream and stream results can be stored in relations for later querying. TelegraphCQ [25] focused on developing novel techniques for processing streams of continuous queries over large volume of data using Postgres. Stanford stream manager STREAM [21] proposed a data model integrating streams into SQL. Aurora [18] used operator definitions to form a directed acyclic graph (DAG) for processing stream data in a single node system. Borealis [17] extended Aurora for distributed stream processing with a focus on fault tolerance and distribution. Cayuga [30] is a stateful publishe-subscribe system that developed a query language for event processing based on an algebra using non-deterministic finite state automaton.

Because these second-generation systems were not designed to handle incoming data in a distributed fashion, a need for a third generation arose as Internet companies began producing data at a high velocity and volume. These third-generation systems were developed with the key focus on scalable processing of streaming data. Yahoo S4 [3] is one of the earliest distributed streaming systems that is near real-time, scalable and allows for easy implementation of streaming applications. Apache Storm [40] is a widely popular distributed streaming system open sourced by Twitter. It models a streaming analytics job as a DAG and runs each node of the DAG as several tasks distributed across a cluster of machines. MillWheel [19] is a key-value based streaming system that supports exactly once semantics. It uses BigTable [26] for storing state and checkpointing. Apache Samza [4] developed at LinkedIn, is a real-time, asynchronous computational framework for stream processing. It uses several independent single-stage computational tasks for stitching together a topology similar to Storm. Each stage reads one or more streams from Apache Kafka [32] and writes the output stream to Kafka for stitching together a processing DAG.

Apache Spark [5] supports streaming using a high-level abstraction called a discretized stream, Spark runs short tasks to process these discretized streams and output results to other systems. In contrast, Apache Flink [2] uses a distributed streaming dataflow engine and asynchronous snapshots for achieving exactly once semantics. Pulsar [35] is a real time analytics engine open sourced by eBay and its unique feature is its SQL interface. Some of the other notable systems include S-Store [34] Akka [1], Photon [20], and Reactive Streams [11]. In addition to these platforms, several commercial streaming systems are available in the market [7], [8], [9], [12], [13]i, [14], and [15].

3 Heron Data Model

Heron uses a directed acyclic graph (DAG) for representing a real-time computation. The graph is referred to as a *topology*. Each node in the topology contains processing logic, and the links between the nodes indicate how the data flows between them. These data flows are called *streams*. A stream is an unbounded sequence of tuples. Nodes take one or more streams and transform them into one or more new new streams. There are two types of nodes: *spouts* and *bolts*. Spouts are the sources of streams. For example, a Kafka [32] spout can tap into a Kafka queue and emit it as a stream. A bolt consumes tuples from streams, applies its processing logic and emits tuples in outgoing streams. Typical processing logic includes filtering, joining and aggregation of streams. An example topology is shown in Figure 1.

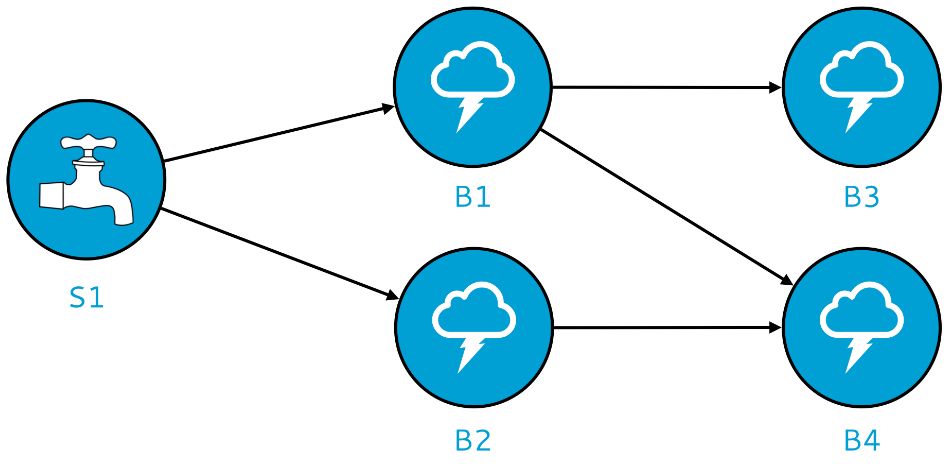

Figure 1: Heron Topology

In this topology, the spouts S1 taps into its data source and emits two streams consumed by the first stage bolts B1, and B2. These bolts transform the streams and emit three new streams feeding bolts B3 and B4. Since the incoming data rate might be higher than the processing capability of a single process or even a single machine, each spout and bolt of the topology is run as multiple tasks. The number of tasks for each spout and bolt is specified in the topology configuration by the programmer. Such a task specification is referred to as the degree of parallelism. The topology shown in Figure 1, when instantiated at run time is illustrated in Figure 2. The topology, the task parallelism for each node and the specification about how data should be routed form the physical execution plan of the topology.

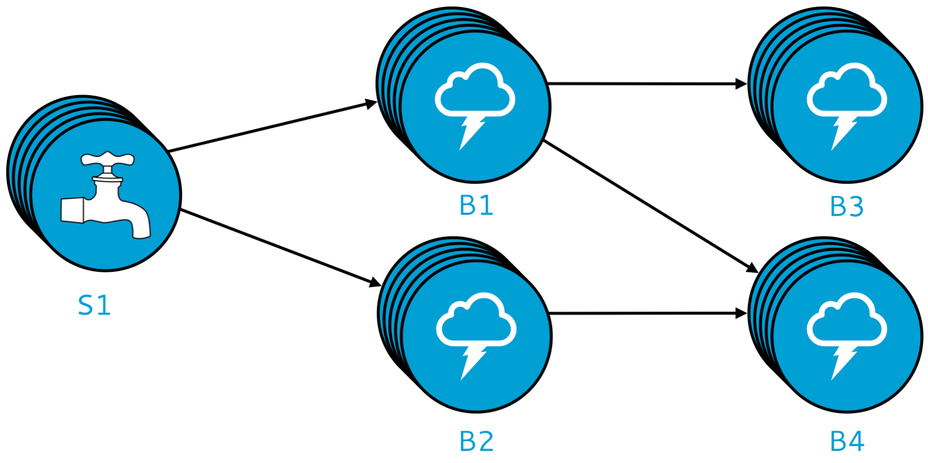

Figure 2: Physical Execution of a Heron Topology

4 Heron Architecture

The design goals for Heron are multifold. First, the spout and bolt tasks need to be executed in isolation. Such isolation will provide the ability to debug and profile a task when needed. Second, the resources allocated to the topology should not be exceeded during the execution of the topology. This requirement enables Heron topologies to be run in a shared cluster environment alongside other critical services. Third, the Heron API should be backward compatible with Storm and a migrated topology should run unchanged. Fourth, Heron topologies should adjust themselves automatically when some of their components are executing slowly. Fifth, Heron should be able to provide high throughput and low latency. While these goals are often mutually exclusive, Heron should expose the appropriate knobs so that users can balance throughput and latency needs. Sixth, Heron should support the processing semantics of at most once and at least once. Finally, Heron should be able to achieve high throughput and/or low latency while consuming a minimal amount of resources.

To meet the aforementioned design goals, Heron uses the architecture as shown in Figure 3. A user writes his or her topology using the Heron API and submits to a scheduler. The scheduler acquires the resources (CPU and RAM) as specified by the topology and spawns multiple containers on different nodes. The first container, referred to as the *master container*, runs the *topology master*. The other containers each run a *stream manager*, a *metrics manager* and several processes called *instances* that execute the processing logic of spouts and bolts.

The topology master is responsible for managing the entire topology. Furthermore, it assigns a role or group based on the user who launched the topology. This role is used to track the resource usage of topologies across different teams and calcuate the cost of running them for reporting. In addition, the topology master acts as the gateway to access the metrics and status of the topology. Once the topology master comes up in the master container, it advertises its location in the form of a host and port via an ephemeral Zookeeper [6] node. This node allows other containers to discover the location of the topology master and also prevents multiple topology

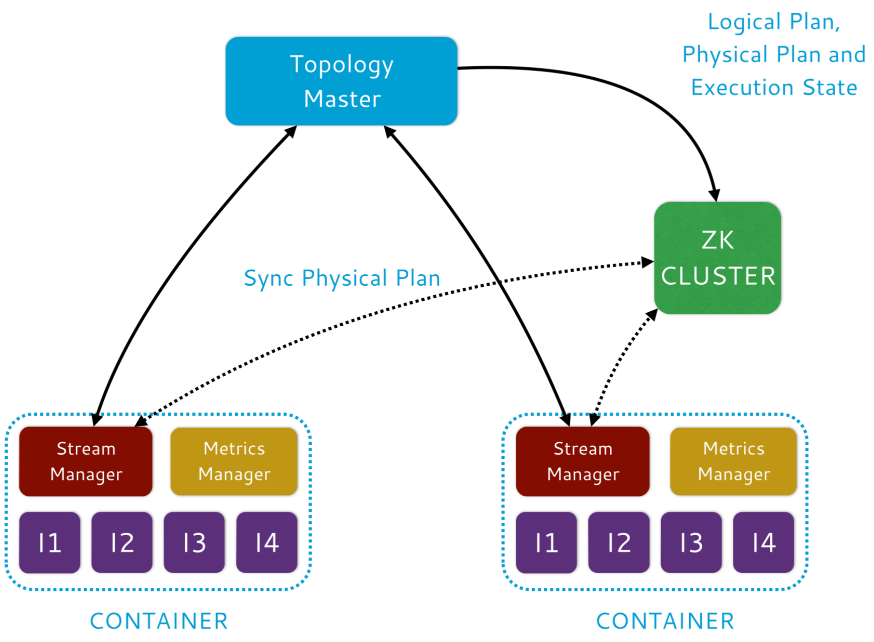

Figure 3: Heron Topology Architecture

masters becoming master during network partitioning. We use an ephemeral node in Zookeeper because when the topology master dies, it detects the loss of session and automatically removes the node.

A network of stream managers route data tuples from one Heron instance to other Heron instances. Each container has a stream manager and the Heron instances in that container send and receive data from it. Even data tuples destined for local Heron instances in a container are routed through the stream manager. When a container is scheduled, the stream manager comes up and discovers where the topology master is running. The stream manager forms a handshake request that includes the host and port on which it is listening and sends it to the topology master. This host and port information allows the topology master to assemble the physical plan and push the plan to all the stream managers. Once stream managers get the physical plan, they connect with other stream managers to form a fully connected graph, as shown in Figure 3.

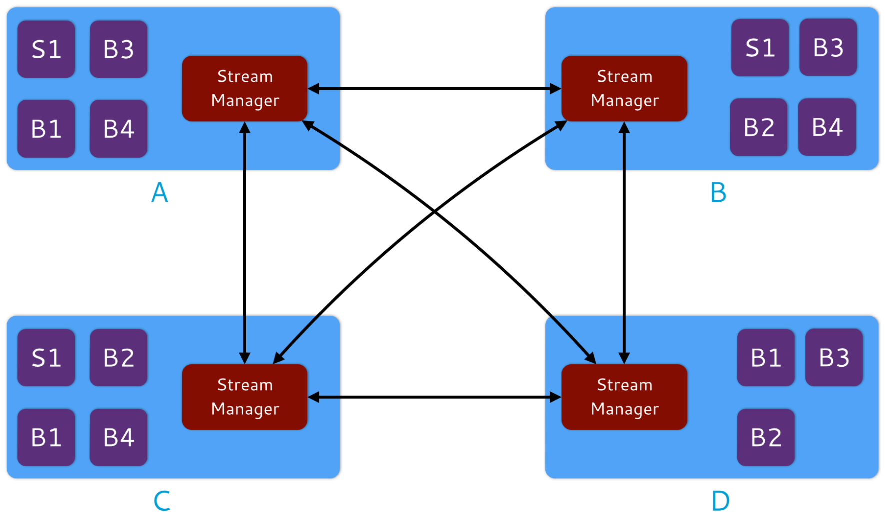

Figure 4: Dataflow in Heron

A Heron instance runs the processing logic in spouts or bolts. Each Heron instance is a process running a single spout task or a bolt task. The instance process runs two threads –the gateway thread and the task-execution thread. The gateway thread communicates with the stream manager to send and receive data tuples from the stream manager. The task-execution thread runs the user code of the spout or bolt. When the gateway thread receives tuples, it passes them to the task-execution thread. The task-execution thread applies the processing logic and emits tuples, if needed. These emitted tuples are sent to the gateway thread, which passes them to the stream manager. In addition to tuples, the task-execution thread collects several metrics. These are passed to the gateway thread, which routes them to the metrics manager.

The metrics manager is responsible for collecting metrics from all instances and exporting them to the metrics-collection system. The metrics-collection system stores those raw metrics and allows for later analysis. Since there are several popular metrics-collection systems, the metrics manager exposes a generic abstraction. This abstraction facilitates ease of implementation for routing metrics to various different metrics-collection systems.

5 Achieving Design Goals

As mentioned in the previous section, Heron was developed with certain design goals in mind. In this section, we examine how we achieved each one of them in detail.

5.1 Task Isolation

Since a Heron instance executes a single task in a dedicated process, it is entirely isolated from other spout and bolt tasks. Such task isolation provides several advantages. First, it is easy to debug an offending task, since the logs from its instance are written to a file of its own providing a time ordered view of events. This ordering helps simplify debugging. Second, one can use performance-tracking tools (such as YourKit [16], etc) to identify the functions consuming substantial time, when a spout or bolt task is running slowly. Third, it allows examination of the memory of the process to identify large objects and provide insights. Finally, it facilitates the examination of execution state of all threads in the process to identify synchronization issues.

5.2 Resource Reservation

In Heron, a topology requests its resources in the form of containers, and the scheduler spawns those containers on the appropriate machines. Each container is assigned the requested number of CPU cores and memory. Once a certain amount of resources (CPU and RAM) are assigned to a topology, Heron ensures that they are not exceeded. This monitoring is needed when Heron topologies are run alongside other critical services in a shared infrastructure. Furthermore, when fragments of multiple topologies are executing in the same machine, resource reservation ensures that one topology does not influence other topologies by consuming more resources temporarily. If resource reservation is not enforced, it would lead to unpredictability in the behavior of other topologies, making it harder to track the underlying performance issues. Each container is mapped to a Linux cgroup. This ensures that the container does not exceed the allocated resources. If there is an attempt to temporarily consume more resources, the container will be throttled, leading to a slowdown of the topology.

5.3 Self Adjustment

A typical problem seen in streaming systems, similar to what is seen in batch systems, is that of stragglers. Since the topology can process data only as fast as its slowest component, stragglers cause lag in the input data to build up. In such scenarios, a streaming system tends to drop data at different stages of the DAG. This dropping of results in either data loss or replay of data multiple times. A topology needs to adjust its pace depending on the prevailing situations. Some of these situations are data skew, where a bolt instance is receiving more data than it can process, and when a fragment of the topology is scheduled on a slow node.

During such scenarios, some feedback mechanism should be incorporated to slow down the topology temporarily so that the data drops are minimized. Heron implements a full fledged back-pressure mechanism to ensure that the topology is self adjusting. We investigated two back-pressure approaches –TCP-based back pressure and spout-based back pressure.

The TCP protocol uses slow-start and sliding-window mechanisms to ensure that the sender is transmitting at the rate the receiver can consume. Hence it is natural to ask whether Heron could leverage the TCP protocol for back pressure. But due to the multiplexing nature of the stream manager, where multiple logical transport channels are mapped on a single physical channel, TCP-based back pressure could slow upstream or downstream spouts or bolts. To illustrate this possibility, consider the physical execution of the topology in Figure 1 with four containers as shown in Figure 5. Assume that an instance of Bolt B3 in Container A is going slow. As shown in Figure 1, Bolt B3 receives input from Bolt B1 which means all instances of Bolt B3 will receive input from all instances of B1. Hence, the stream manager in Container A will receive input from bolt instances of B1 running in Containers C and D. Since the instance of Bolt B3 in Container A is going slow, its stream manager will not take any additional input from the stream managers of the containers C and D. Since the connection between stream managers use TCP sockets, eventually the socket send buffers in stream managers in Containers C and D will fill up. As a result, the data exchange between Bolt B1 and B2 (shown in green) in containers C and D with bolt B4 (shown in green) in Container A is affected. We found that for some topologies, such situations could eventually drive the throughput to zero.

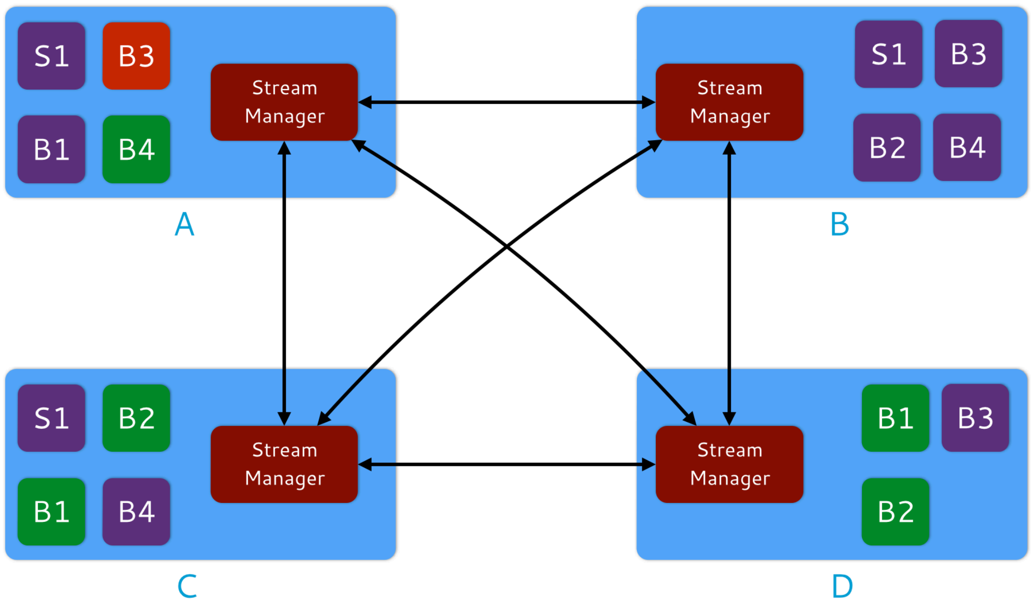

Figure 5: TCP Back Pressure

We considered another approach called spout-based back pressure. This approach is based on the observation that spouts are the sources of data and we can manage when they emit or suspend the injection of data. In other words, whenever a stream manager detects one of the instances is going slow, it will explicitly send an initiate-back-pressure message to all the other stream managers. When a stream manager receives this message, it examines the physical plan and, if there are any spouts running in the container, it will not consume data from them. To illustrate, again consider the physical execution of topology in Figure 1 as shown in Figure 6. When the Bolt B3 in Container A goes slower, its stream manager sends the initiate-back-pressure message to stream managers of all the containers. Upon receiving this message, the stream managers in Containers B and C do not consume data from their spouts, in this case, Spout S1 (shown in blue). This action reduces the data inflow into the topology thereby self adjusting. Once the Bolt B3 picks up pace, its stream manager sends a relieve-back-pressure message to all other stream managers. They act on this message by starting to consume from their local spouts. More details about the back pressure mechanism can be found in Kulkarni, et al. [33].

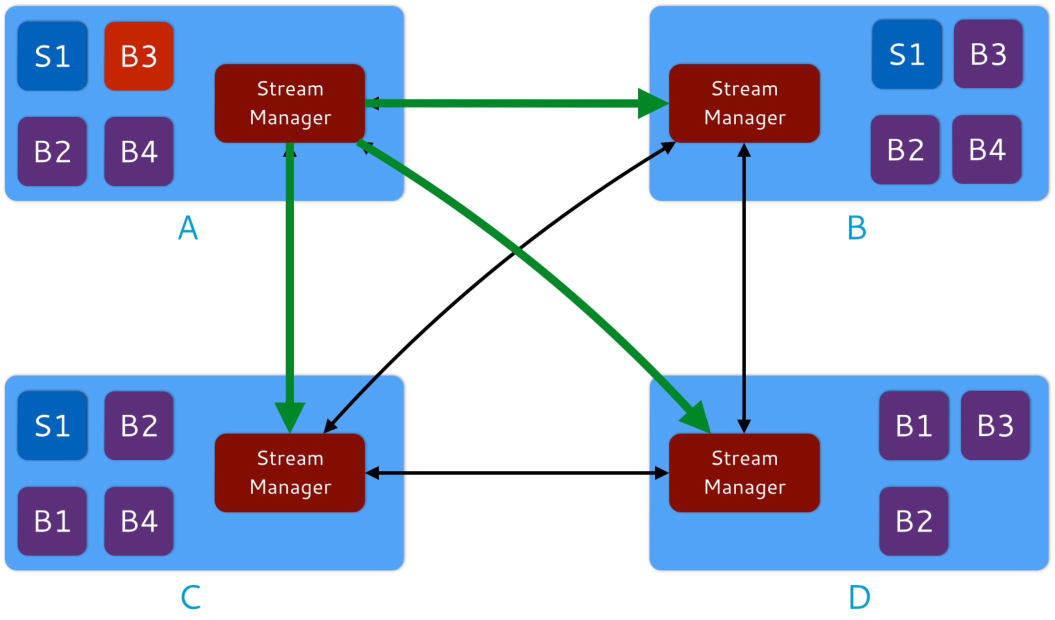

Figure 6: Spout Back Pressure

5.4 Processing Semantics

In order to provide predictability, a stream processing system needs to provide guarantees on the data that passes through it. Heron supports two different types of processing semantics:

- **At most once:** In this semantics, the processing is best effort. In the presence of node or process failures, the data processed by the streaming system could be lost. Hence, the number of data tuples processed might be lower than the actual number of data tuples, which could affect the results.
- **At least once:** In this semantics, the system guarantees that the data is processed at least once. If the data is dropped during node or process failures, it is reprocessed. It is possible that the same data tuple is processed more than once. Hence, the number of data tuples processed might be higher than the actual number of data tuples, again potentially affecting the results.

Incorporating at-most-once semantics in Heron is straight forward. A Heron topology continuously processes data and, during processing, the data moves from instance to stream manager and between stream managers. When an instance in a container fails, the state accumulated by the bolt or spout is lost. After restart, it connects with the stream manager and continues to receive and process data thereby, accumulating new state. Similarly, when a stream manager in a container dies, it restarts and reconnects to other stream managers and resumes processing. If an entire container fails due to node failure, the container is relocated to another node. Once the stream manager and instances in the relocated container come up, the data processing continues. During relocation, the data intended for the failed stream manager from other stream managers could be dropped or if the data is buffered, the buffers could overflow, eventually dropping data.

6 Heron in Practice

Heron has been in production at Twitter for over two years. It is used for diverse use cases such as real-time business intelligence, real-time machine-learning, real-time classification, real-time engagements, computing real-time trends, real-time media, and real-time monitoring. In this section, we will explore some of operational issues that occur in practice and how we solve them.

6.1 Back Pressure

Spout-based back pressure helped us reduce data loss significantly as stragglers are the norm in multi-tenant distributed systems. The Heron back-pressure recovery mechanism allows us to process data at a maximal rate such that the recovery times are very low. Since most topologies are provisioned with extra capacity to handle increased traffic during well-known events (such as the Super Bowl and the Oscars), the recovery rate is usually much higher than the steady state. In cases where the topologies have not been provisioned to handle increased traffic, the back pressure mechanisms act as a shock absorber to handle any temporary spikes. In cases where these spikes are not temporary, back pressure also allows users to add more capacity and restart their topologies with minimal loss of data.

We have encouraged topology writers to test their back pressure (and recovery) mechanism in staging environments by artificially creating traffic spikes (e.g., by reading from older offsets in Kafka). This practice allows them to understand the dynamic behavior of back pressure and measure the recovery time. To monitor this process in real time, several metrics have been exposed on the dashboard. Back pressure also helps topology writers in tuning their topology. Since we do not have auto tuning (yet), users are required to use trial and error to get the correct values for resource allocation and parallelism of the components. By looking at the back pressure metrics, they can identify which of the components are under back pressure and correspondingly increase the resources or parallelism until there is no back pressure in steady state.

In our experience, we have found that in most scenarios, back pressure recovers without manual intervention. However, there are cases where a particular component in topology gets scheduled on a faulty host or goes into irrecoverable garbage-collection cycles (for various reasons). Under such scenarios, users get paged, upon which they usually restart those components to get the problem fixed. While most users see back pressure as a requirement, some users prefer dropping data as they only care about the latest data. To handle such cases, we added the *load-shedding* feature in spouts as decribed in the following section.

6.2 Load Shedding

Load shedding has been studied extensively in the context of second-generation streaming systems [23, 24, 36, 38, 39, 41]. Most of the proposed alternatives fall into two broad categories, sampling-based approaches and data-dropping-based approaches. The idea behind sampling-based approaches is that if the system can automatically downsample an incoming stream in a predictable way, the user can potentially scale up the results of the computation in order to compensate. For example, if a Heron topology is counting widgets and the stream is being downsampled by 50%, the user can simply multiply the widget counts by two for each widget that is present in the stream and therefore still get approximately correct results.

The common theme of sampling approaches is that a more uniformly sampled stream is easier to reason about and a user could also use the information about the sampling rate to scale the output of the computations, which is a very desirable property. However, for sampling to be useful to applications, it would be important that the sampling was done on a global level.

If each spout instance was individually sampling at different times and different rates the value of uniform sampling to applications programmers is pretty much negated. The system would lose the property that it is easy to reason about the sampling that is happening and also the ability to properly scale the output of the computation based on the sampling rate. Due to these limitations and its considerably higher complexity, we did not implement the sampling-based approach.

On the other hand, the idea behind dropping-based approaches is that the system will simply drop older data and prefer more recent data when the Heron topology is unable to keep up. Heron spouts are modified such that the user can configure a *lag threshold* and a *lag-adjustment value*. The lag threshold will indicate how much lag is tolerable before the spout drops any data. The lag-adjustment value will indicate how much of the old data the system will drop when this threshold is reached.

Given the two values described above, the system will monitor the lag for each individual spout instance and periodically skip ahead by the lag adjustment value whenever the lag is above the threshold value. A key point here is that the decision to drop data is a completely local decision in each spout instance. There will be no attempt made to synchronize amongst different spouts or otherwise coordinate such that the spouts work together in deciding what data to drop. Each spout drops data from its associated Kafka or Eventbus partition and no communication between spouts will occur.

6.3 Kestrel Spout

Kestrel [10] is a simple distributed message-queuing system. Each Kestrel host handles a set of reliable, and ordered, message queues. A Kestrel cluster consists of several such hosts with no communication between them. Whenever a client is interested in enqueuing or dequeuing an item, it randomly picks a host, thereby providing reliable, loosely ordered message queue behavior. An attractive property of Kestrel is its ability to scale, since servers do not communicate with each other and have no need for any coordination.

Unlike Kafka [32], Kestrel is stateful. In order to maintain state, Kestrel replicates data for every consumer. In other words, Kestrel assumes only one consumer per physical queue. An item in the queue is removed only after a client dequeues and then acknowledges it. If two different instances of a consumer are consuming from the same Kestrel queue, it is guaranteed that they will never receive same item, given that they acknowledge their respective items. If the item is not acknowledged within a specified amount of time, it is placed back in the queue for the next instance to receive.

We started with the open source Kestrel spout and it worked reasonably well. However, as traffic grew, Heron topologies using Kestrel spouts faced several issues, such as:

- One or more Kestrel hosts would start accumulating data and not drain. The immediate resolution is to manually mark those servers as read only until they drain, and enable writes once the number of items to be consumed goes below a certain threshold. This approach presents an operational challenge, especially during non-working hours. When a host is not getting drained, it affects the performance of other queues it needs to service as well. One possible solution is to set maxItems (the maximum number of items held in queue) and maxAge (maximum amount of time an item stays in the queue before it is deleted) limits on the queues to be small, so that the size of queue does not grow to affect other queues on the host. But this solution results in data loss for the job consuming this queue.

- A Kestrel spout would pack the Kestrel client (or connection) along with the data in a tuple. This would cause the spout to become stateless, because when the tuple came back to the spout to get acknowledged, it just extracted the client from the tuple and acknowledged it back to Kestrel host to retire the tuple. The problem with this approach was that the tuple size grew, and it carried extra load for no reason, which resulted in extra data transfers, and more serialization and deserialization costs.

- A Kestrel spout would create a new connected client every time it requested the next batch of items from Kestrel. While this behavior has no effect on topologies with low throughput, for more data-heavy topologies, the number of connections to a host grew without bound. Some of the spout-related configurations, such as maximum spout pending (limits the number of tuples in flight in a topology, so the spouts do not request an unbounded number of tuples) often hid this problem. Furthermore, creation of many connections exacerbated garbage-collection issues.

The root cause for one or more Kestrel hosts not draining was triggered by the use of Zookeeper to discover Kestrel hosts. Specifically, the Kestrel spout used a service factory for creating a connection to one of the Kestrel hosts in the server set, The factory did not provide any guarantees that all the hosts would be connected and read evenly. As a result, some of the servers were occasionally left out, causing items from those servers to not be consumed. Our initial solution was to fetch all the hosts from the Kestrel server set, and read from each server in a round-robin fashion. This practice ensured that no server is left unread, while giving all the hosts equal priority. This approach worked even during times of high load, because it is assumed that to achieve steady state, the read rate has to be higher than the write rate. So even in case of high load, round robin would drain the full queues, and bring the system to steady state.

Soon we saw an issue where instead of one Kestrel host lagging, all of the hosts were backing up. This issue was traced to one host being unable to respond and because of the round robin policy, all the hosts were read at the pace of the slowest one. The actual slow down of the host was due to disk writes for logging. Hence, an approach was needed to decouple a slow host from others temporarily. To solve the issue, each spout instance is assigned a configurable number of Kestrel hosts. These assignments were not mutually exclusive, and had overlaps. The three main properties of these assignments are:

- Each spout instance reads from a subset (more than one) of Kestrel hosts.
- Each Kestrel host is read by a subset (more than one) spout instances.
- If any two Kestrel hosts, A and B, are read by one spout instance, then there exists a spout instance that reads from host A and not B, and another instance that reads from host B and not A.

The last property ensures that if one Kestrel host slows down, the rest of the hosts will be read without any penalties. And using round-robin reads ensures that the slow host will not be left out, and will still be drained.

The issue of passing a Kestrel client was fixed by mapping each tuple to its Kestrel client using a combination of a generated unique identifier and the original item identifier provided by the Kestrel host. This approach also prevented the creation of several client objects by reusing existing Kestrel client objects. Finally, we added configuration parameters to control both the number of connections per Kestrel host from a spout instance and the number of pending items per connection, which helped in playing nice with Kestrel.

7 Conclusion

Heron has become the de-facto real-time streaming system at Twitter. It runs several hundred development and production topologies and been in production for more than two years. Several teams in Twitter use Heron for making real-time data-driven decisions that are business critical. Heron is used for several diverse use cases ranging from ETL to building machine-learning models and is expanding rapidly. These use cases require additional future work to evolve Heron.

First, manual resource assignment for a topology when it goes production currently requires several iterations. Each iteration involves changing the configuration parameters, recompiling and redeploying. For large topologies, each iteration is very expensive. We want to explore an elegant solution for estimating initial resource requirements using a combination of data-source characteristics, sampling and linear regression. Second, the topologies are often overprovisioned to accommodate peak loads during popular events to avoid manual intervention. This policy led to resource wastage and hence we are investigating approaches where the topology can expand automatically and shrink depending on traffic variations. Third, we want to support a declarative query paradigm that allows users to write queries faster and be more productive. Fourth, in some uses cases, we have to guarantee data processing by the topology is exactly once. The problems of auto-scaling and exactly once will require distributed partitionable state and additional Heron APIs.

8 Acknowledgements

Thanks to David Maier and Kristin Tufte for providing comments on the initial draft of the paper that helped improved its presentation. Thanks to Jeff Naughton, Deep Medhi and Jignesh Patel for reading the pre-final draft and help improve the presentation. Thanks to Arun Kejariwal for help with LaTeX including setting it up, patiently answering several questions and providing several comments on the first draft.

References

[1] Akka. `http://akka.io/`.

[2] Apache Flink. `https://flink.apache.org/`.

[3] Apache S4. `http://incubator.apache.org/s4/`.

[4] Apache Samza. `https://samza.apache.org/`.

[5] Apache Spark. `https://spark.apache.org/`.

[6] Apache Zookeeper. `http://zookeeper.apache.org/`.

[7] Apama Streaming Analytics. `http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/default.asp`.

[8] Informatica Vibe Data Stream. `https://www.informatica.com/products/data-integration/real-time-integration/vibe-data-stream.html#fbid=v8VRdfhc8YI`.

[9] InfoSphere Streams: Capture and analyze data in motion. `http://www-03.ibm.com/software/products/en/infosphere-streams`.

[10] Kestrel: A simple, distributed message queue system. `http://twitter.github.io/kestrel`.

[11] Reactive Streams. `http://incubator.apache.org/s4/`.

[12] SAP Event Stream Processor. `http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html`.

[13] SQLstream Blaze. `http://www.sqlstream.com/blaze/`.

[14] TIBCO StreamBase. `http://www.streambase.com/`.

[15] Vitria OI For Streaming Big Data Analytics. `http://www.vitria.com/solutions/streaming-big-data-analytics/benefits/`.

[16] YourKit. `https://www.yourkit.com/`.

[17] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 277–289, 2005.

[18] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2), Aug. 2003.

[19] 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*, 6(11):1033–1044, Aug. 2013.

[20] 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 International Conference on Management of Data*, pages 577–588, 2013.

[21] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 665–665, 2003.

[22] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, 2002.

[23] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems. In *Proceedings of the 2003 Workshop on Management and Processing of Data Streams MPDS*, 2003.

[24] B. Babcock, M. Datar, and R. Motwani. Load shedding in data stream systems. In C. Aggarwal, editor, *Data Streams*, volume 31 of *Advances in Database Systems*, pages 127–147. Springer US, 2007.

[25] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[27] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. Niagara CQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[28] S. Dann. Twitter content classification. *First Monday*, 15(12), December 2010. `http://firstmonday.org/ojs/index.php/fm/article/view/2745/2681`.

[29] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, March 1988.

[30] A. Demers, J. Gehrke, M. Hong, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[31] N. Gehani and H. V. Jagdish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

[32] N. N. Jay Kreps and J. Rao. Kafka: A distributed messaging system for log processing. In *SIGMOD Workshop on Networking Meets Databases*, 2011.

[33] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Streaming at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, 2015.

[34] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: streaming meets transaction processing. *Proceedings of VLDB Endowment*, 8(13):2134–2145, Sept. 2015.

[35] S. Murthy and T. Ng. Announcing Pulsar: Real-time Analytics at Scale. `http://www.ebaytechblog.com/2015/02/23/announcing-pulsar-real-time-analytics-at-scale`, Feb. 2015.

[36] S. Senthamilarasu and M. Hemalatha. Load shedding using window aggregation queries on data streams. *International Journal of Computer Applications*, 54(9):42–49, September 2012.

[37] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[38] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 309–320, 2003.

[39] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06).*

[40] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.

[41] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 787–798, 2006.

[42] J. Vijayan. Streaming Analytics: Business Value from Real-Time Data. `http://www.datamation.com/data-center/streaming-analytics-business-value-from-real-time-data.html`.

[43] J. Widom. The Starburst rule system: Language design, implementation, and applications. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15:1–4, 1992.