Performance Evaluation of Stream Data Processing Systems

ABSTRACT

Over the past years, Stream data processing is gaining compelling attention both in industry and in academia due to its wide range of applications in various use-cases. To fulfil the need for efficient and high performing Big data analytics, numerous open source stream data processing systems were developed. Processing data with high throughput while retaining low latency is key performance indicator for such systems. In this paper, we propose a benchmarking system to evaluate the performance of stream data processing systems, Storm, Spark and Flink in terms of indicators shown above. More specifically, the latency of windowed aggregation and windowed join operators is evaluated jointly with the throughput of a system.

1. INTRODUCTION

Streaming Data Processing (SDP) has been gaining significant attention due to its wide range of uses in Big data analytics. The main reason is that processing big volumes of data periodically is not enough anymore and data has to be processed fast to enable fast adaptation and reaction to changed conditions. Several engines are widely adopted and supported by open source community, such as Apache Storm [**?**], Apache Spark [**?**] , Apache Flink [**?**]. There are definitely, different ways to process stream data before it is persisted in database. For example, while Storm and Flink provide record-by-record stream processing, Spark has a different approach to collect events together and process all in a series of mini-batches.

Determining the right Stream Data Processing System (SDPS) for use case is key to get best performance. Latency and throughput are two main KPIs to measure the performance of SDPS. Latency is the time required to calculate the final result. Because the stream is theoretically infinite and therefore there is no final result, defining latency in SDPS can be tedious. Throughput on the other hand determines the number of successful calculations per unit time. In stream data processing, it is number of tuples source operator can ingest for further process per unit time.

In this work we propose the benchmark system to assess the performance of three major, open source and community driven streaming engines, being Apache Storm, Apache Spark and Apache Flink. Throughout the tests, we use latency and throughput as the major performance indicators. The benchmarks are designed on top of real world use cases, specifically the ones from *Rovio Entertainment*. The first use case is finding an average price over numerous streaming data sources by dividing them into sliding windows. Here, we aim to evaluate the performance of partitioned windowed aggregations. The second use case is comparison of prices which originated in same geo locations by dividing them in sliding windows. Here, we focus on performance of partitioned windowed joins.

There are numerous works that do benchmarks among different SDPSs with specific Key Performance Indicators (KPI) [**?**, **?**]. However, researchers in previous works, either don't clearly specify the semantics of KPIs or the determined KPIs are affecting system's actual performance. That is, the definition and calculation of KPIs should be clear and those should be kept separate from system under test. Moreover, keeping benchmark design simple and with possible less systems, makes the results less biased and reproducible. For example, if one of underlying benchmarking systems is a bottleneck for measurements, then the actual results of system under test, can be interpreted wrongly.

TODO: STATEFUL AND STATELESS OPERATORS LATENCY

In this paper, we overcome all of possible bottlenecks listed above. Our system is generic, has simple design with clear semantics and can be applied to any streaming system. It consists of predefined number of data generators and actual System Under Test (SUT). The link between SDPS and data generator is a socket. To overcome with extra latency of persistent queues, we removed persistent queue as a connector and used sockets. The main intuition is to simulate an environment to get actual latency of a tuple with minimum affections of other factors. Each tuple has its timestamp field which indicates the time it was generated by data generator. Rather than connecting the SDPS to directly data generator, we put a queue between them as the SDPS may not ingest all generated data. Waiting in queue increases the latency of a tuple, so the sooner the system pulls data from queue, the lower the latency will be. The throughput on the other hand, is calculated by number of pulls of SDPS to queue in a unit time.

The main contributions of this paper are listed above:

- In this paper, we introduce the first mechanism to measure the latency in stateful operators is SDPS. We applied the proposed method with partitioned windowed aggregation and partitioned windowed join use cases.
- The proposed benchmarking system measures the throughput of a system and the latency of an operator out of the box. That is, the throughput is associated with the system and therefore we measure throughput outside the system, in data generator module. Moreover, the latency is linked with operator, therefore, we assess it outside of operator. The main goal is, to eliminate side factors affecting the measurements.
- Because we are testing SDPSs, the backpressure is an inportant feature for such systems and should be considered. We introduce the first solution to check the SUT's upper limit of throughput taking into consideration backpressure and system initialization delays. If the system cannot sustain the data rate, depending on user's configuration, data-queue module will tolerate it for some time.
- We test the SDPSs with different configurations and cluster size and provide the analysis of the experimental results.

The remainder of paper is organised as follows. In Section ... TODO

2. RELATED WORK

The main concepts and methodologies used in benchmarking SDPS were inherited from big data benchmarks. Now with emerging next generation stream data processing engines, batch processing is seen as a special case of stream processing where the data size is bounded. Huang et.al. propose HiBench, the first benchmark suite for evaluation and characterisation of Hadoop [**?**]. Authors conduct wide range of experiments from micro-benchmarks to machine learning algorithms [**?**]. Covering end-to-end big data benchmark with all major characteristics such as three Vs in the lifecycle of big data systems is the main intuition behind BigBench [**?**]. Wang et.al. introduce BigDataBench, a big data benchmark suite for Internet Services, characterising the 19 big data benchmarks covering broad application scenarios and diverse and representative data sets [**?**].

Benchmarks on SDPS are Researchers from Yahoo Inc. have done benchmarks on streaming systems to measure latency and throughput [**?**]. They used Apache Kafka [**?**] and Redis [**?**] for data fetching and storage. Later on, on the other hand, Data Artisans, showed those systems actually being a bottleneck for SUT's performance [**?**]. The extensive analysis of the differences between Apache Spark and Apache Flink in terms of batch processing is done by correlating the operators execution plan with the resource utilization and the parameter configuration [**?**]. In another benchmark, authors compare the performances of Apache Spark and Apache Flink to provide clear, easy and reproducible configurations that can be validated by community in clouds [**?**]. Authors conducted benchmarks to assess the fault tolerance and throughput efficiency for open source stream data processing engines [**?**]. In another benchmark, authors motivate IoT as being main application area for SDPS and perform common tasks in particular are with different stream data processing engines and evaluate performance [**?**]. One of the pioneers in SDPS benchmarks, developed framework StreamBench analysing the current standards in streaming benchmarks and proposing a solution to measure throughput, latency considering the fault tolerance of SUT [**?**]. Authors put a mediator system between data generator module and SUT and define the latency as the average time span from the arrival of a record till the end of processing of the record. LinearRoad benchmarking framework was presented by Arasu et al. to measure performance of standalone stream data management systems such as Aurora [**?**] by simulating a scenario of toll system for motor vehicle expressways. Several stream processing systems implement their own benchmarks to assess the performance [**?**, **?**, **?**]. SparkBench is a benchmarking framework to evaluate machine learning, graph computation, SQL query and streaming application on top of Apache Spark [**?**].

3. PRELIMINARY AND BACKGROUND

In this section we provide preliminary and background information about the stream data processing engines used throughout this paper. Initially, the general information about the working principles of particular system is given. Afterwards, we provide more use case specific info for each system. Because we test engines' partitioned windowed join and aggregation operators, basic semantics of particular operators , computational model and back-pressure mechanism are analysed.

3.1 Apache Storm

Apache Storm is a distributed stream processing computation framework which was open sourced after being acquired by Twitter.

Computational model Storm operates on tuple streams and provides record-by-record stream processing. It supports at-least-once processing (when there are failures events are replayed) mechanism and guarantees all tuples to be processed. Storm also supports exactly-once semantics with its Trident abstraction. The core of Storm data processing is a computational topology which consists of spouts and bolts. Spouts are source operators whereas bolts are processing and sink operators. Because Storm topology is DAG structured, where the edges are stream tuples and vertices are operators (bolts and spouts), when a spout or bold emits a tuple, the ones that are subscribed to particular spout or bolt receive input. Storm's parallelism model is based on *tasks*. Each task runs in parallel and by default single thread is allocated per task.

Storm's lower level API's provide little support for managing the memory and state. Therefore, choosing the right data structure for state management, utilizing memory usage efficiently my making computations incrementally is up to the user. memory management. Storm supports cashing and batching the state transition. However, the efficiency of particular operation degrades as the size of state grows. Storm supports back-pressure.

Windowing. Storm has built-in support for windowed calculations. That is, partitioned windowed joins and aggregations are supported internally. Although the information of expired, new arrived and total tuples within window is provided through APIs, the state management and making

computations incremental should be done manually. Storm supports processing and event-time windows with sliding and tumbling window features. Processing time windows include time and count based semantics. For event-time windows, tuples should have separate timestamp field so that the engine can create periodic watermarks. One of the downsides of Storm's relying heavily on ackers, is that tuples can be acked once they completely flush out of window. This can be an issue specially, on windows with big length and small slide.

3.2 Apache Spark

Apache Spark is an open source data processing engine, originally developed at the University of California, Berkeley.

Computational model Spark internally is batch processing engine. It handles the stream processing by micro-batches. As can be seen from Figure 1, Spark Streaming resides at the intersection of batch and stream processing. Resilient Distributed Dataset (RDD) is a fault tolerant abstraction which enables in memory parallel computation in distributed cluster environment [**?**]. Unlike Storm and Flink, which support one record at a time, Spark Streaming inherits its architecture from batch processing which support processing records in micro-batches.

One of Spark's features is that it supports lazy evaluation and tries to limit the amount of work it has to do. This enables the engine to run more efficiently. Spark also supports DAG based execution graph which works implementing stage-oriented scheduling. Unlike from Flink and Storm, which also work based on DAG exetuion graph, Spark computing unit in graph is data set rather than streaming tuple and each vertex in graph is a stage rather than operators. RDDs are guaranteed to be processed in order in a single DStream. However, the order guarantee within RDD is not provided since each RDD is processed in parallel.

Spark Streaming has improved significantly its memory management in recent releases. The memory is shared between execution and storage. This unified memory management supports dynamic memory management between the two modules. Moreover, Spark supports dynamic memory management throughout the tasks and within operators of each task.

Windowing Spark Streaming has a built-in support for windowed calculations. Processing time windows with sliding and tumbling versions are supported in Spark. The operations done with sliding windows, are internally incrementalized transparent to the user. However, choosing the length batch interval can affect the window based analytics. Firstly, the latency and response time of windowed analytics is strongly replying on batch interval. Secondly, supporting only processing time windowed analytics, can still be a bottleneck in some use cases. Spark supports back pressure which is very useful in windowed calculations. The window size must be a multiple of the batch interval, because window keeps the particular number of batches until it is purged.

3.3 Apache Flink

Apache Flink which was started off as an academic open source project (Stratosphere [**?**]) in Technical University of Berlin.

Computational model Distributed dataflow engine is standing in the core of Flink. It is responsible for executing the dataflow programs. Like Storm, A Flink runtime program is a DAG of stateful operators connected with data streams. Flink's runtime engine supports unified processing of batch (bounded) and stream (unbounded) considering former as being the special case of the latter.

Flink provides its own memory management to avoid long running JVM's garbage collector stalls by serialising data into memory segments. The data exchange in distributed environment is done via buffers. So, producer takes a buffer from the pool, fill it up with data, and the consumer receives data and frees the buffer informing the memory manager. There are different mechanisms such as sending when buffer is full or sending when timeout is reached to link buffers between consumer and producer or sending locally or remotely. Flink provides wide range of high level and user friendly APIs to manage the state. The incremental state update, managing the memory or checkpointing with big states is done automatically, transparent to user.

Windowing Flink owns strong feature set for building and evaluating windows on data streams. With wide range of pre-defined windowing operators, it supports user defined windows with custom logic. The engine provides processing time, event time and ingestion notion of time. In processing time, like Spark, windows are defined with respect to the wall clock of the machine that is responsible for building and processing a window. In event time on the other hand, the notion of time is determined when the event are created. Like in Storm, the timestamps must be attached to each event record as a separate field. In ingestion time, the system still processes with event time semantics but on the timestamps which were assigned when tuples arrive the system. Flink has a support for out-of-ordered streams which gained popularity after Googles MilWheel and Dataflow papers [**?**, **?**]

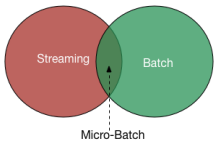


Figure 1: Conceptual view of micro-batching

4. CHALLENGES

There are several challenges to be taken into consideration while designing a benchmark for SDPSs. In this section we analyse the challenges and provide solutions.

Simple is beautiful. The first challenge is to design a simple system, because simple is beautiful. As the number of systems included inside benchmark increases, the complexity escalades at the same time. One of downsides of the complex system is, difficult to determine the bottleneck. For example, to test the stream data processing engine, the data generator component is essential, to simulate the real life scenarios. Figure **??** shows possible three cases to link data generator and SUT. The simplest design would be connecting the SDPS directly to data generators as shown in Figure 8e. Although this is perfectly acceptable case, it confronts with real life use cases. That is, in real life, the

stream data processing engines, do not connect to pull based data sources unless there is a specific system design. Usually, SDPSs pull data from the distributed message queues which reside between data generators and SUT. One bottleneck on this option is throughput which is bounded by the maximum throughput of message queueing system. We selected the third option which stands between the first two. As can be seen from Figure 8h, we embedded the queues as a separate module in data generators. In this way, the throughput is bounded only by network bandwidth, the system works more efficient as there are no se/deserialisation overheads and finally it has a simple design. Moreover, while measuring the performance of SUT, using extra systems for checkpointing the current state of measurements or for saving the intermediate evaluation results can affect the SUT's performance.

Keep driver and test unit as separate as possible. The second challenge is to isolate the benchmark driver and SUT as much as possible. For example, in benchmarking SDPSs, it is common to measure the throughput inside the SUT. Because both computations can affect each other the results can be biased which we discuss below. We solved this problem by categorising the test unit and pointing measurements accordingly. The first evaluation is throughput. Throughput is associated with the system. So, we kept the throughput assessment outside the SUT, inside data generator module. The second evaluation is latency. The latency is linked with an operator inside system. So, we kept all latency measurements outside the particular operator.

Avoid misleading tests. The third challenge is abstain from biased test results and keep the evaluation semantics clear. One example for this is, throughput measurement. In the previous SDPS benchmarks, the throughput of a SUT is measured by either taking quantiles over test time, or showing max, min and average assessments. From user's perspective on the other hand, the system's throughput is the one with upper bound that it can sustain the data processing. When conducting the experiments with stream data processing engines, back pressure is another factor affecting the throughput, that should be taken into consideration. Another example for this challenge is latency measurement. In the previous SDPS benchmarks, the latency of an operator is measured by taking difference of tuple's ingestion timestamp to system and output timestamp from system. However, from user's perspective, the start timestamp of a tuple is once it is created and latency of particular tuple should be calculated by taking into consideration its event timestamp. We solve the first issue by measuring the SUT's max throughput that it can sustain under back pressure with a given amount of input. We developed the mechanism for handing the back pressure. For the second issue, we measure the latency of a tuple by differentiating the time when output is done and its event timestamp.

Latency of stateful operator? The fourth challenge is measuring the latency of stateful operator. Up to this point, the related works in literature either concentrated on stateless operators or evaluated the latency of stateful operators by checkpointing to external systems like lightweight distributed databases. As we discussed above, this approach can be a bottleneck in some cases. In this paper, we provide a solution for this problem without any external system.

5. BENCHMARK SYSTEM DESIGN

We keep overall design of benchmark simple. Figure 3 shows the overall intuition behind the benchmark system. There are two main components of a system:

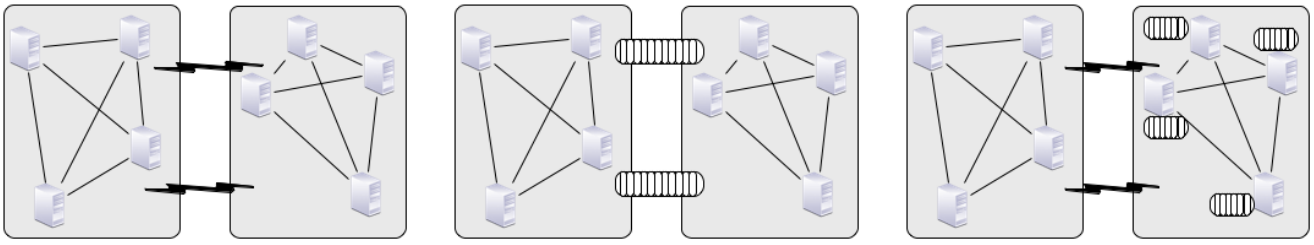
- SUT
- Data Engine
  - Data Generator
  - Data Queue

The Data Engine component is responsible for generating and queueing the synthetic data. It has two subcomponents: Data Generator and Data Queue. Both of them reside in the same machine to avoid network overhead and the data is kept in memory to circumvent the disk write/read overhead. First subcomponent, data generator, generates data with a given speed. To ensure high throughput we kept the number of fields in an event minimal. To assure the scalability and each Data Generator connects to its own Data Queue rather than to common and centralized message queue. Queue is based on FIFO semantics. In this way, we avoid data de-/serialization, disk read/write and network overhead. Theoretically, this approach has no difference from the one which includes centralized and distributed message queue. The following analogy can be made: the topic in distributed message queueing system is analogous to Data Queues, and partition is single Data Queue. The Data Generator appends the current time to timestamp field of an event. The event's latency is calculated from this point and the more it stays in queue, the more the latency is. The number of Data Engines can be arbitrary and its overall throughput is only bounded by network bandwidth. As we discussed above, the throughput assessment which is associated with the SUT as a whole, is done in Data Engine component, which has a clear separation from SUT.

The SUT is second main component, which processes the data. So the only interface of Data Engine to outer world is Data Queue subcomponent. SUT pulls data from Data Queue. The connection is pull based because we let the SUT to decide when and how much data it can ingest. The SUT connects predefined number of Data Queues. Firstly it combines the input data sources. There can be one or two unions, depending on the operator semantics. For example, windowed aggregation is a single input stream (union of streams) operator but join operator accepts two streams (union of streams) as an input. The latency calculation is done inside SUT. This measurement is associated with the operator inside SDPS, we clearly separate the latency assessment from Operator Under Test (OUT). Placing extra stateless operator just after OUT and measuring the latency between the event time and current time gives the latency of an element. The calculation of latency in windowed aggregation and windowed join operators is shown in Section 5.2.2.

5.1 Use case

The use case for this benchmark is provided by *Rovio Entertainment*. It is known as a video game development company. To get higher customer satisfaction, it is mandatory to analyse the game usage statistics. For example, company releases a new feature for a particular video game. It is essential to have a quick overview of customer's opinion by analysing the usage statistics. For that reason, Rovio uses



(a) Without message queue (b) With message queue (c) Partial message queue

Figure 2: Different system designs to link data generator and SUT.

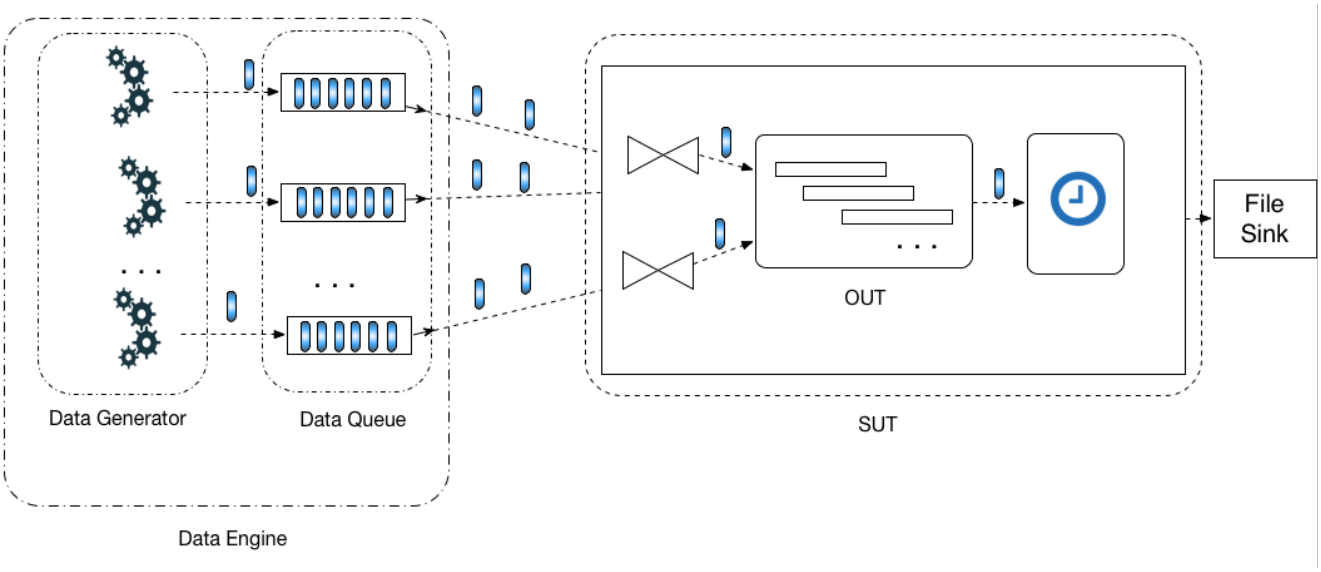


Figure 3: Design of benchmark system.

stream data processing analytics rather than batch based periodic analysis. In all use cases an event has following fields: event timestamp, key and value.

Windowed Aggregation Windowed aggregations are important part of the statistics analytics from the feed stream. One use case is computing the user average session time within windows. Each event has its geo ID, which indicate the location in which an event was generated. A use case includes partitioning the events by their ID and finding the average session time within windows. Here, the key field is geo location and value field is session time. Another use case is computing the average ads screening time within window for each geo location.

Windowed Join Joining several stream feeds based on key field within window is another general use case for user statistics analytics. One specific use case is joining two user statistics stream feed within window by geo field and calculating an event with higher session time and the difference between them. In this use case the key field would be geo location and value would be the session time. Another use case would be doing the same procedure with ads watch time.

5.2 Key Performance Indicators

The Key Performance Indicators (KPIs) for this benchmark are latency and throughput. The throughput indicator is related with SUT, on the other hand, the latency is associated with OUT.

5.2.1 Throughput

Definition. Let $c_i \in C$ be a configuration for a Data Engine, $d_i \in D$, c_i^{sp} be the data generation speed configuration which can be sustained by SUT. If $\exists c_i \in C$ such that $c_1 = c_2 ... = c_i$ and $T = \sum_i c_i^s$ is maximum, $\forall c_i \in C, \forall i \in \{1, 2, 3...|C|\}$, then T is a maximum sustainable throughput of SUT.

Throughput of a system is calculated as summing the throughout of each Data Engine because the SUT is expected to pull the data from all Data Queue subcomponent of Data Engine approximately with same rate. We restricted the configuration of all Data Engines to be the same, to ease calculating the maximum sustainable throughput. It is crucial to note that the maximum sustainable throughput is not the same as maximum throughput of Data Engine but the one for SUT.

To examine the system's sustainability with a given throughput, we divide the queue used in Data Queue subcomponent into three parts: c^a , c^b and c^n. The Figure 4, shows the example partitioning of queue. If the size of the queue is less than or equal to c^a then this is acceptable and means, the SUT can sustain the given throughput. If the queue size is less than or equal to c^b on the other hand, the SUT

cannot sustain the given data rate but we can tolerate it for some time, in case the increase in queue size is an outcome of back-pressure. However, if the queue size is bigger than the c^b then the SUT cannot sustain the given throughput and there is no need to do benchmarks with particular data rate.

The semantics behind examination of SUT's sustainability with a given throughput must be clear. Moreover, it should support the system specific behaviours like back-pressure. Algorithm 1 show an algorithm to check if the SUT can sustain the given throughput of a single Data Engine. It gets the configuration c of Data Engine as an input. After firing the Data Engine with configuration c, in line 3, it is put to *idle* position, meaning no data is generated until the SUT makes its first data pull request from Data Queue subcomponent. c^{input} is the input count that must be generated in particular Data Engine. In lines $10-11$ the events are generated with speed c^{sp} in Data Generator subcomponent of Data Engine component and put into the queue of Data Queue subcomponent. We check the queue size periodically, once per c^a element and not for each iteration. The first reason is that the data pull rate of stream data processing engine is not steady and therefore checking the queue size with little delays makes more sense. The second reason is that, c^a can be thought of the $confidencelimit$ as shown in Figure 4. While checking the queue size there are three possibilities. The first is (lines $13-15$), the size is bigger than c^b, the back-pressure limit. In this case, the Data Engine is stopped and the $false$ is returned meaning the SUT is not sustainable with given throughput. The second is (lines $16-18$), queue size is less than c^a, the acceptable queue size. In this case, we set back-pressure counter to zero, in case there was one and continue generating data. The third is (lines $19-23$), the size of queue is within boundaries of c^a and c^b. In this case, we can tolerate the SUT for at most $\frac{c^b}{c^a}$ times. If the system can pull the data in queue and set the size of queue within *confidence* boundaries (c^a) in a given period, then it continues, else, the application returns $false$ meaning, the SUT cannot sustain the given throughput.

Definition. Let $c_i \in C$ be a configuration for a Data Engine, c_i^{dr} be a data generation rate of particular Data Generator, $d_i \in D$ and n be the number of Data Engines being $n = |D|$. The SUT is sustainable with given throughput $n * c_{dr}$, iff $isSustainable(c_i) == true \ \forall \ i \in \{1, 2, 3, ...n\}$

The above definition states that the SUT is sustainable with a given data generation rate iff, it can sustain all Data Engines at the same time. If one of the Data Engines cannot be sustained, then SUT is said is not sustainable with a given data generation rate.

Algorithm 1: Throughput sustainability test of single data engine

```
1  function isSustainable (c);
   Input  : c is configuration of Data Engine
   Output: return true if is sustainable, false otherwise
2  Fire Data Engines with configuration c.
3  c^st ← idle ;                  // wait for SUT to pull data
4  while There is no pull request from SUT do
5  |  wait
6  end
7  c^st ← active ;                  // start generating data
8  bp_index ← 0 ;   // initialize back-pressure index
9  for i ← 0; i < c^input; i++ do
10 |  Generate e_i ∈ E with speed c^sp
11 |  queue.put(e_i) ; // put generated event to queue
   |  /* check the queue once per c^a elements    */
12 |  if i % c^a == 0 then
13 |  |  if queue.size > c^b then
14 |  |  |  Stop Data Engine
15 |  |  |  return false
16 |  |  else if queue.size < c^a then
17 |  |  |  bp_index ← 0 ;        // no back-pressure
18 |  |  |  continue ;      // SUT can sustain so far
19 |  |  else
   |  |  |  /* Tolerate for back-pressure        */
20 |  |  |  bp_index ← bp_index + 1
21 |  |  |  if bp_index == c^b/c^a then
   |  |  |  |  /* This is not back-pressure      */
22 |  |  |  |  Stop Data Engine
23 |  |  |  |  return false
24 end
```

5.2.2 Latency

Latency is another KPI for this benchmark and defining the latency needs clear semantics. There are several points that needs to be clarified: *i)* the aggregation or join of timestamp fields of tuples and *ii)* clear boundaries (start and end timestamp) of latency.

The first point is the aggregation or join of tuples with timestamp fields. While the use case provides the semantics for aggregating or joining the tuples' *value* fields, the one for timestamp field is unclear. For example, in windowed aggregation operator, which calculates the average of elements' *value* field, the aggregation semantics with tuples' timestamp field is unclear. The Equation 1 addresses this

issue. Let $t[k]$ and $t'[k]$ denote the timestamp field for tuples t and t' respectively, $\equiv$ be an operator checking for the type and TS be tuple field of type timestamp. Here, f_s is an stateful operator which takes a set of tuples $t \in T$ and converts it to tuple t'. Then there exists k and m such that the respective fields of input and output tuples have the same type being timestamp and the output tuple's timestamp is calculated taking maximum among input tuples. Here input and output tuples are associated with operator f_s and not with the SUT.

To calculate the latency, it is crucial to have clear boundaries of when to start and stop the stopwatch for each tuple. Equation 2 defines the basic semantics behind this. This is basically a follow-up for Equation 1. Let $t_i \in I$ be a tuple in input set and $t_o \in O$ be a tuple in output set. The latency is associated with output tuples. So, the latency of tuple t_o is calculated by extracting the $t_o[m]$, the timestamps field from current time. The calculation of output tuples' timestamp field is shown in Equation 1.

$$\begin{gathered} \textbf{Let} f_s : \{t | t \in T\} \rightarrow t', \\ \textbf{then}, \exists k, m \ s.t. \ t[k] \equiv t'[m] \equiv \ TS \ \forall t \in T \\ \textbf{and} \ t'[m] \leftarrow \text{argmax}\{t[k] \mid t \in T\} \end{gathered} \tag{1}$$

$$\begin{gathered} \textbf{Let} \ t_i \in I, t_o \in O \\ \textbf{then} \ Latency_{\ t_o} = time_{now} - t_o[m] \\ s.t. \ f_s : \{t_i \ | t_i \in I\} \rightarrow \{t_o \ | t_o \in O\} \\ \textbf{and} \ t_o[m] \equiv TS \end{gathered} \tag{2}$$

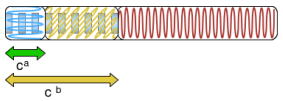


Figure 4: Basic intuition behind *back-pressure-compatible queue*

6. EVALUATION

6.1 Configuration

The following configurations are used throughput experiments:

- Cluster size: 2,3,4 and 8 node clusters
- Parallelism within single node: number of cores, which is 16.
- Parallelism within cluster: (Parallelism within single node) * (number of nodes)
- Backpressure: enabled in all systems
- Network bandwidth: 1Gb
- Number of Data Engines running in parallel: 16
- Allocated memory: 16GB
- Cluster type: Standalone
- Input size for aggregation use case: 150M * 16

	2 Node	**3 Node**	**4 Node**	**8 Node**
Storm	345	235	345	345
Spark(2sec)	8K	15K	23K	23K
Spark(4sec)	240K	480K	656K	920K
Flink	1072K	1232K	1232K	1232K

Table 1: Sustainable throughput

	2 Node	**3 Node**	**4 Node**	**8 Node**
Storm	345	235	345	345
Spark(2sec)	376	707	1.1K	1.1K
Spark(4sec)	12K	24K	33K	47K
Flink	53K	58K	58K	58K

Table 2: Average Window size

- Input size for join use case:
- Spark batch size: 4 seconds and 2 seconds
- Window type: Processing time
- Number of distinct keys in input: 160
- Join inputs selectivity:
- c_a, acceptable queue limit: 1M
- c_b backpressure tolerated queue limit: 15M

6.2 Keyed Windowed Aggregations

6.2.1 Storm

6.2.2 Spark

6.2.3 Flink

6.3 Joins

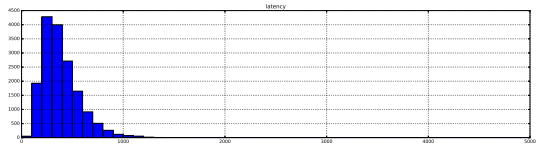
same as above

7. CONCLUSIONS

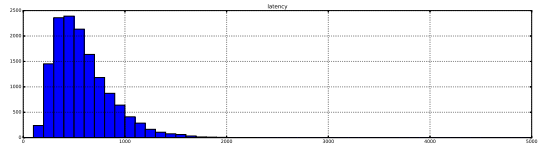
8. ACKNOWLEDGMENTS

	2 Node	**3 Node**	**4 Node**	**8 Node**
Storm	345	235	345	345
Spark(2sec)	1.6s	1.7s	1.7s	1.6s
Spark(4sec)	6.9s	3.6s	4.4s	3.9s
Flink	0.3s	0.4s	0.35s	0.28s

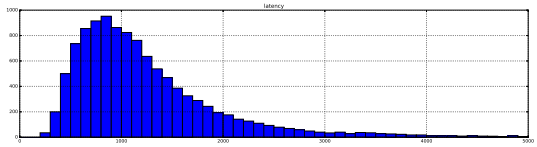
Table 3: Average Latency



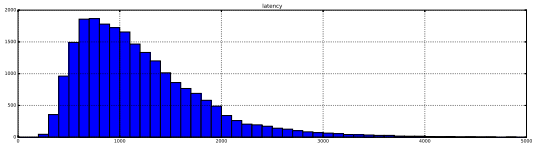
(a) 2 Node latency histogram



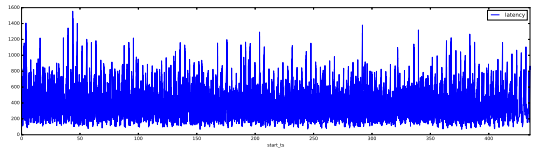
(b) 3 Node latency histogram



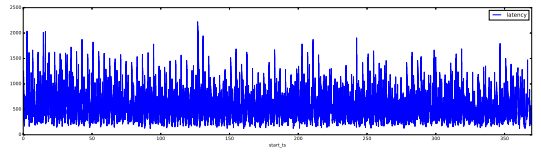
(c) 4 Node latency histogram



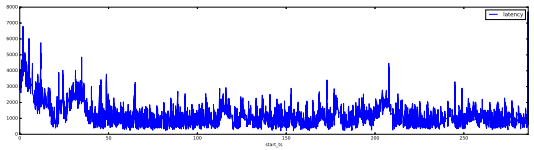
(d) 8 Node latency histogram



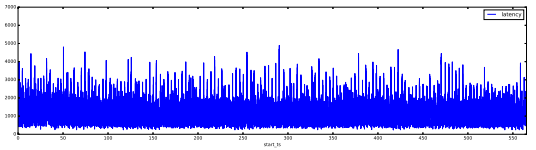
(e) 2 Node latency time series



(f) 3 Node latency time series

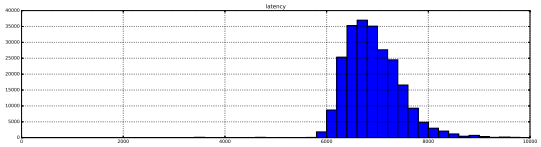


(g) 4 Node latency time series

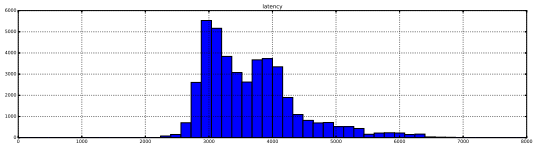


(h) 8 Node latency time series

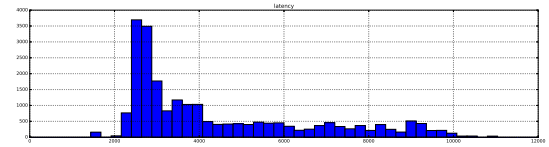
Figure 5: Latency of windowed aggregations for Storm



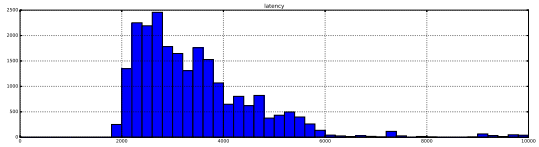
(a) 2 Node latency histogram



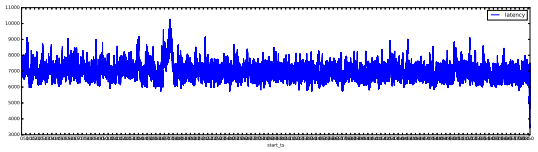
(b) 3 Node latency histogram



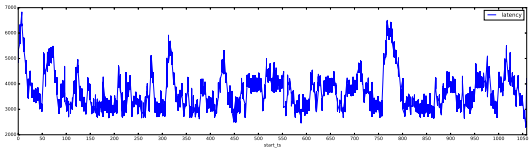
(c) 4 Node latency histogram



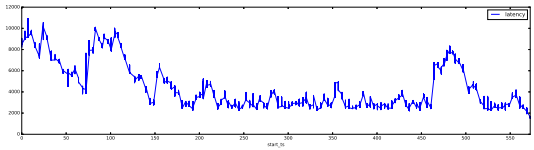
(d) 8 Node latency histogram



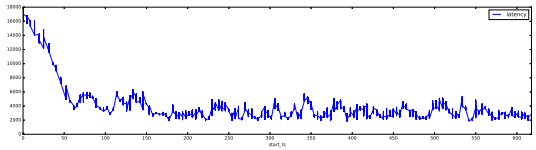
(e) 2 Node latency time series



(f) 3 Node latency time series

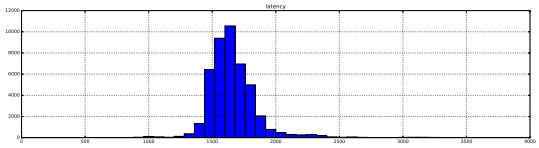


(g) 4 Node latency time series

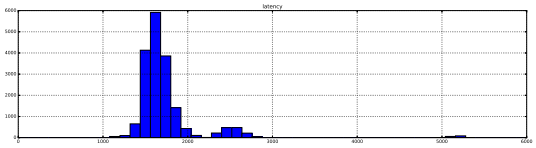


(h) 8 Node latency time series

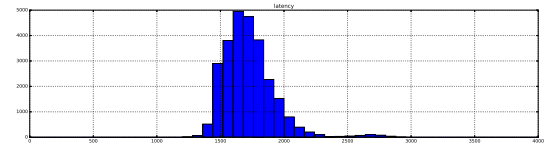
Figure 6: Latency of windowed aggregations for Spark (4 sec batch).



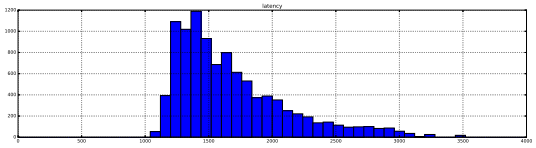
(a) 2 Node latency histogram



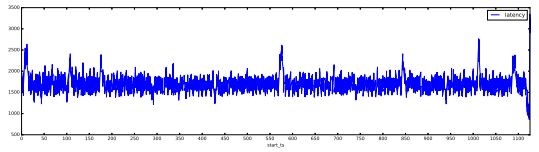
(b) 3 Node latency histogram



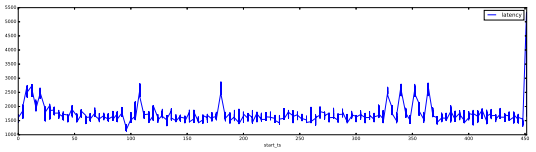
(c) 4 Node latency histogram



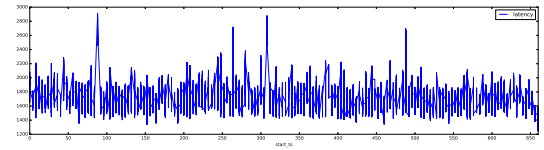
(d) 8 Node latency histogram



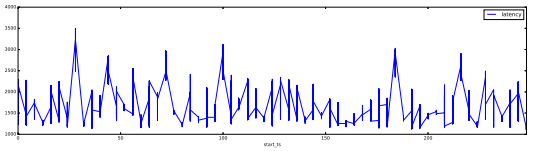
(e) 2 Node latency time series



(f) 3 Node latency time series

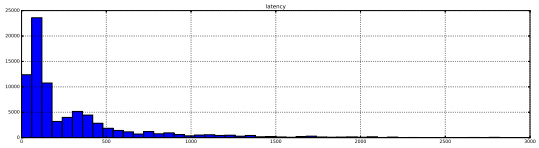


(g) 4 Node latency time series

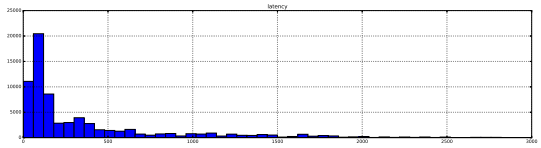


(h) 8 Node latency time series

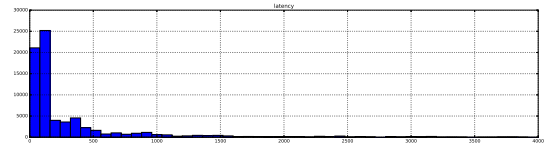
Figure 7: Latency of windowed aggregations for Spark (2 sec batch).



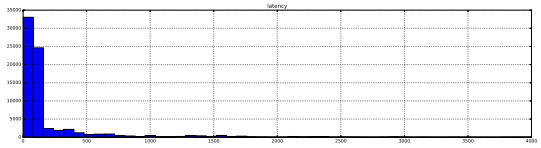
(a) 2 Node latency histogram



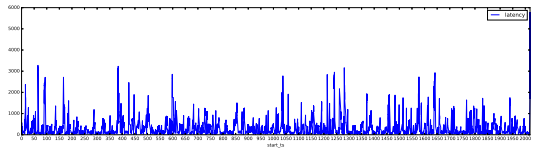
(b) 3 Node latency histogram



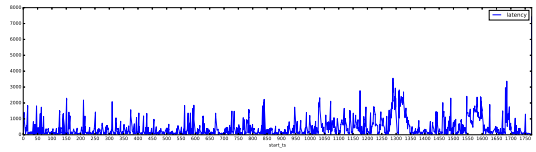
(c) 4 Node latency histogram



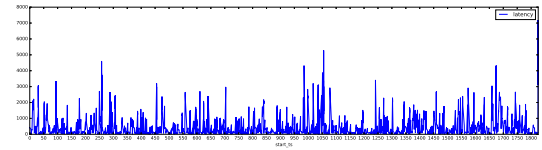
(d) 8 Node latency histogram



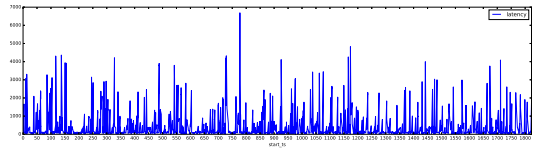
(e) 2 Node latency time series



(f) 3 Node latency time series



(g) 4 Node latency time series



(h) 8 Node latency time series

Figure 8: Latency of windowed aggregations for Flink.