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 (SDPS) 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 SDPSs, Storm, Spark and Flink in terms of latency and throughput. More specifically, the latency of windowed aggregation and windowed join operators is evaluated jointly with the throughput of a system. The main goal of this work is to build a benchmarking system for SDPS which is simple because simple is beautiful. The main contribution of this work is threefold. Firstly, we first define a definition of latency and throughput for stateful operators, based on the industry use-case. Secondly, we completely separate the unit under test and driver. This enables us not only building a simple and transparent system but also empowers the system to be flexible. Finally, we first build a driver to test the sustainability of a unit under test which supports various user defined policies.

1. INTRODUCTION

Stream data processing has been gaining significant attention due to its wide range of uses in big data analytics. The main idea is that processing big volumes of data with batch processing engines 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 [20], Apache Spark [24] , Apache Flink [5].

One of application areas of stream data processing is video games. Processing of large scale online data feeds from different sources swift is imperative for online video game companies in industry. For example, once the company releases the new feature of particular video game, fast and efficient analysis of customer usage statistics is a must to determine the their attitudes and satisfaction. In this paper, we collaborate with *Rovio Entertainment*, the video game company. We build the benchmarking system on top of *Rovio* use-case. One use-case is related with analytics on windowed aggregation, calculating the average session time of users within specified geo locations. Another use-case is associated with windowed join, calculating the time lag between clients and company services.

In this work we propose the benchmarking 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. Latency is the time required to calculate the final result, throughput on the other hand determines the number of successful calculations per unit time. In stream data processing, it is number of tuples per unit time the engine can ingest for further process. We use two *Rovio* use-cases for experiments.

There are numerous open challenges related with measuring main Key Performance Indicators (KPIs) in stream data processing engines. For example, there is not clear definition for the latency of stateful operator within SDPS. That is, the stream is theoretically infinite and there is no final output. While the previous works use checkpointing each event's ingestion and output time, this can lead to misleading results and escalade the driver's complexity. Moreover, while benchmarking SDPSs, it is crucial to separate completely the driver and units under test as otherwise, the results can be biased. Furthermore, the driver should be flexible enough to support different policies to handle system specific features. For example, back-pressure is characteristic feature of SDPSs and while calculating system's sustainability it must be taken into consideration. Most importantly, keeping the benchmarking system simple while solving the challenges listed above is imperative.

In this paper, we overcome the challenges listed above. The proposed solution is generic, has simple design with clear semantics and can be applied to any SDPS. We keep the design of a solution as simple as possible. The main intuition is to simulate an environment in which we can calculate the KIPs more precisely and with minimum influence of side factors. We design experiments on top of *Rovio* use-cases.

The main contributions of this paper are listed above:

- We first introduce the term sustainability of SDPS and give its definition. The stream data processing engine is sustainable with a given input if it can process it

within user-defined quality boundaries. For example, user can define custom logic which can handle back-pressure while measuring system's sustainability.

- We first introduce the 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.
- We accomplish the above contributions with a simple system design in which the KPI measurements are clearly separated from units under test.
- We test the proposed benchmark on Storm, Spark and Flink with industry use-cases of *Rovio*.

The remainder of paper is organized as follows. In Section 2, related benchmarks in stream data processing and in general big data analytics are studied. The preliminary and background information about the engines being tested are analyzed in Section 3. The detailed interpretation of open challenges and their importance are discussed in Section 4. We provide the design of benchmark system, the use-cases and KPIs in Section 5. After evaluations in Section 6 we conclude in Section 7.

2. RELATED WORK

The main concepts and methodologies used in benchmarking SDPSs are inherited from benchmarks of batch processing engines. 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 [10], the first benchmark suite for evaluation and characterisation of Hadoop [22]. 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 [9]. 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 [21].

Benchmarks on SDPSs extend the batch data processing benchmarks and pose unique challenges. Researchers from Yahoo Inc. have done benchmarks on stream data processing engines to measure latency and throughput [7]. They used Apache Kafka [11] and Redis [6] for data fetching and storage. Later on, on the other hand, Data Artisans, showed those systems actually being a bottleneck for SUT's performance [8]. 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 [15]. 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 [17]. Benchmarks to assess the fault tolerance and throughput efficiency for open source stream data processing engines show that the micro-batch stream processing system, Spark, is more robust to node failures on the other hand, it is up to 15 time worse than naive stream processing systems in terms of performance [13]. In another benchmark, authors motivate IoT as being main application area for SDPS and perform common tasks in particular area with different stream data processing engines and evaluate performance [19]. Yet another SDPS benchmarks pioneers, developed framework StreamBench analysing the current standards in streaming benchmarks and propose a solution to measure throughput, latency considering the fault tolerance of SUT [14]. 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 [1] by simulating a scenario of toll system for motor vehicle expressways. Several stream processing systems implement their own benchmarks to assess the performance [16, 18, 24]. SparkBench is a benchmarking framework to evaluate machine learning, graph computation, SQL query and streaming application on top of Apache Spark [12].

3. PRELIMINARY AND BACKGROUND

In this section we provide preliminary and background information about the stream data processing engines used throughout this paper.

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 mechanism and guarantees all tuples to be processed. However, when there are failures events are replayed. 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. Storm supports cashing and batching the state transition. However, the efficiency of particular operation degrades as the size of state grows. Back-pressure is also supported by Storm although not being mature yet.

Windowing. Storm has built-in support for windowed calculations. Although the information of expired, new arrived and total tuples within window is provided through APIs, the incremental state management is not transparent to user in core Storm. Trident on the other hand, has built-in support for partitioned windowed joins and aggregations. 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 4, 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 [23]. 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 [4]) in Technical University of Berlin, moved to Apache afterwards.

Computational model Distributed dataflow engine is standing in the core of Flink. It is responsible for executing the dataflow programs. Like Storm, Flink runtime program is a DAG of operators connected with data streams. Flink's runtime engine supports unified processing of batch (bounded) and stream (unbounded) data, 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 [2, 3]

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 for the driver, because this avoids extra overheads and bottlenecks and of course, simple is beautiful. As the number of sub-systems included in driver increases, the complexity escalades at the same time. One of downsides of the complex system is, difficult to determine the bottleneck. One example is the connection system between data generator and SUT. To test the stream data processing engine, the data generator component is essential, to simulate the real life scenarios. Figure 1 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 1a. 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 sources and SUT as shown in Figure 1b. One bottleneck of this option is throughput which is bounded by the maximum throughput of message queueing system. Moreover, there is another de-/serialization layer which *artificially* increases SUT's latency. We selected the third option which stands between the first two. As can be seen from Figure 1c, 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 efficiently as there are no de-/serialisation overheads, it can scale out easily and finally it has a simple design.

Isolate driver and test units. The second challenge is to isolate the benchmark driver and SUT as much as possible. For example, in SDPSs benchmarks, it is common to measure the throughput inside the SUT. Because both computations (driver and test) can affect each other the results can be biased which is discussed 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.

Artificial latency and throughout. The third challenge is to 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 maximum throughput is defined as upper bound for processing the given workload. When conducting the experiments with stream data processing engines, back pressure is one factor affecting the throughput, that should be taken into consideration. We solve the first issue by measuring the SUT's max throughput with maximum workload that it can sustain. We give the definition of SUT's sustainability with given workload. The mechanism we provide for measuring sustainability is customisable based on user's preferences. For example, it can take user-defined function and measure sustainability according to particular function. In this way, we can easily plug the custom logic for back-pressure handling.

Another example for this challenge is latency measurement. Latency is a time interval between the stimulation and response. For batch data processing systems, we assume that data is already in the system so, the latency is the interval between tuples' read and output. For SDPSs on the other hand, the latency is the time between tuple's event time and output. If there are extra systems between SDPS and data source, then those are likely to add *artificial latency* for each tuple. Some researchers circumvented this problem by measuring latency in SDPS the same as in batch data processing systems. In this case, the number and complexity of the systems between SDPS and data source is regardless as latency is the interval between tuples' ingestion and output. However, this measure of latency is *optimistic* as latency for SDPSs is between tuple's event time and output time. Another factor triggering the *artificial latency* is input data rate. If the input data rate is higher than the SDPS can sustain, the results can be biased. Even if the system supports back-pressure, if data rate is more than it can sustain, initially the system will try to ingest as much data as possible. Then, the calculation time will take longer and the latency will increase for every tuple. For this case, it is crucial to measure system's sustainability rate and calculate the latency in that specific configuration, otherwise the measurements will be biased. We solve this issue by clearing the systems creating artificial latency between data source and SDPS and measuring the latency with the sustainable throughput only.

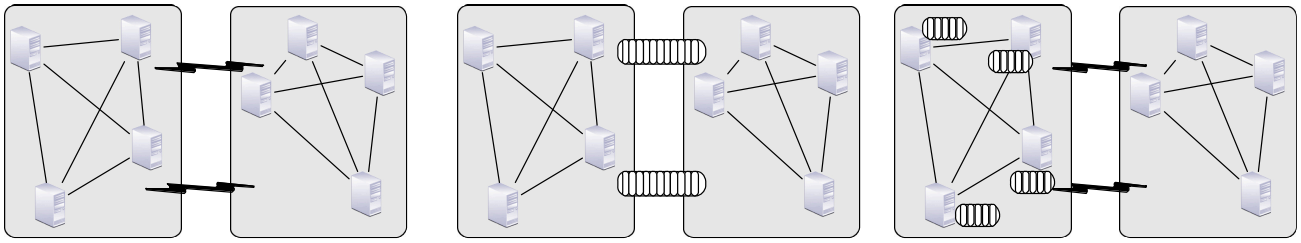
Latency of stateful operator The fourth challenge is measuring the latency of stateful operator. Up to this point, the related works in the literature either concentrated on stateless operators or evaluated the latency of stateful operators by checkpointing to external systems like lightweight distributed databases. As we discussed above, this approach can be a bottleneck in some cases. In this paper, we define the latency of stateful operator formally and provide a solution for this problem without any external system. Basically, the latency of the target operator can be calculated by just putting extra operator adjacent to it and the difference in tuple's timestamp and current timestamps are calculated in particular operator.

5. BENCHMARK SYSTEM DESIGN

We keep overall design of benchmark simple. Figure 2 shows the overall intuition behind the benchmark system. There are two main components of a system: *i)* SUT and *ii)* Data Engine. Data Engine has 2 subcomponents being *i)* Data Generator and *ii)* Data Queue.

The Data Engine component is responsible for generating and queueing the data. Both of its subcomponents reside in the same machine to avoid network overhead and to ensure the data locality. The data is kept in memory to circumvent the disk write/read overhead. First subcomponent, Data Generator, generates data with a given speed. The speed is constant throughout the benchmark. To ensure high throughput we keep the number of fields in an event minimal. To assure the data locality each Data Generator connects to Data Queue residing in the same machine. Because of the bottlenecks explained in Section 4, we avoid queueing data in centralized message queues. Queues in Data Queue subcomponent are based on FIFO semantics. Theoretically, this approach has no difference from implementing the centralized and distributed message queues between SUT and data source. The following analogy can be made: the topic in distributed message queueing system is analogous to set of all Data Queues, and partition is the 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 another component of a system, which processes the data. Because 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 the frequency of data ingestion. The SUT connects predefined number of Data Queues. The pulled data from multiple Data Queues is combines inside SUT. There can be one or two unions, depending on the operator semantics. For example, windowed aggregation is a single input stream operator but join operator accepts two streams as an input. So, for windowed aggregation operator we union all input streams and give as an input. On the other hand, for windowed join operator, we make two unions of input streams and give them 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



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

Figure 1: Different system designs to link data generator and SUT.

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 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 3, 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 3. 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

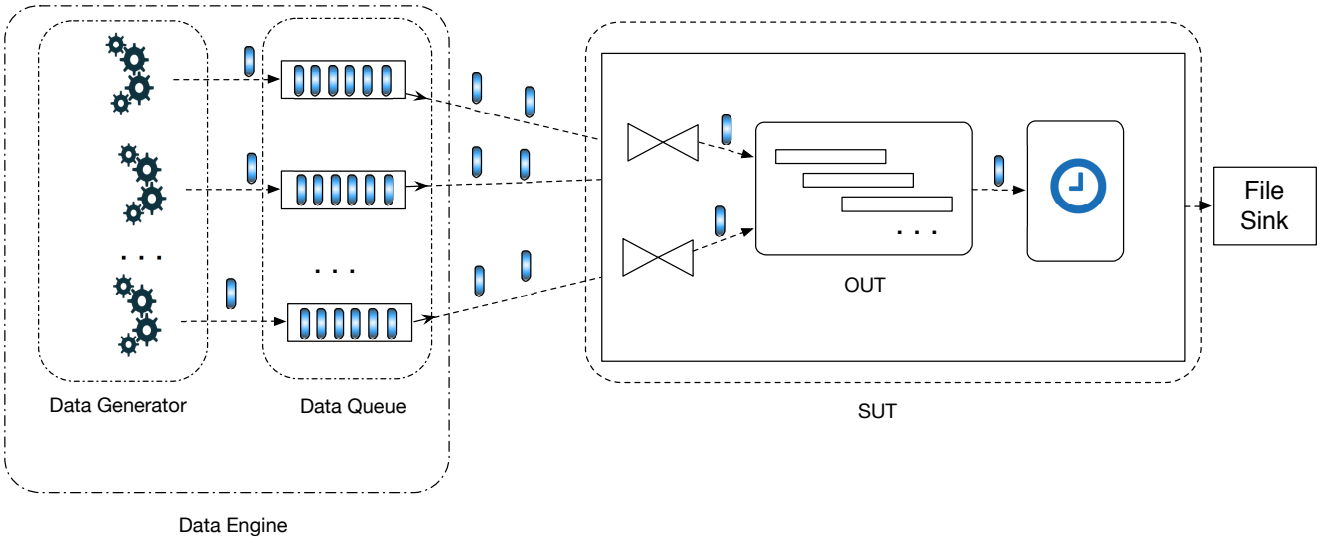


Figure 2: Design of benchmark system.

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.

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

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
```

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.

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

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

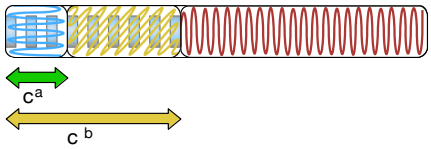


Figure 3: Basic intuition behind *back-pressure-compatible queue*

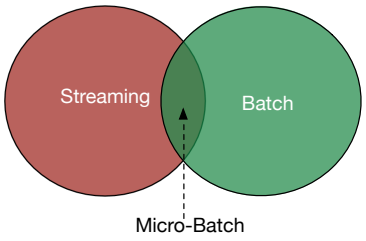


Figure 4: Conceptual view of micro-batching

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
- 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

	2 Node	4 Node	8 Node
Storm	408K	696K	XXX
Spark	379K	642K	912K
Flink	1230K	*1260K	*1260K

Table 1: Sustainable throughput for windowed aggregations

	2 Node	4 Node	8 Node
Storm	1424ms	2043ms	XXX
Storm(90%)	1109ms	1669ms	XXX
Spark	3759ms	4138ms	3152ms
Spark(90%)	3418ms	2846ms	2798ms
Flink	576ms	259ms	243ms
Flink(90%)	299ms	-	-

Table 2: Average Latency for windowed aggregations

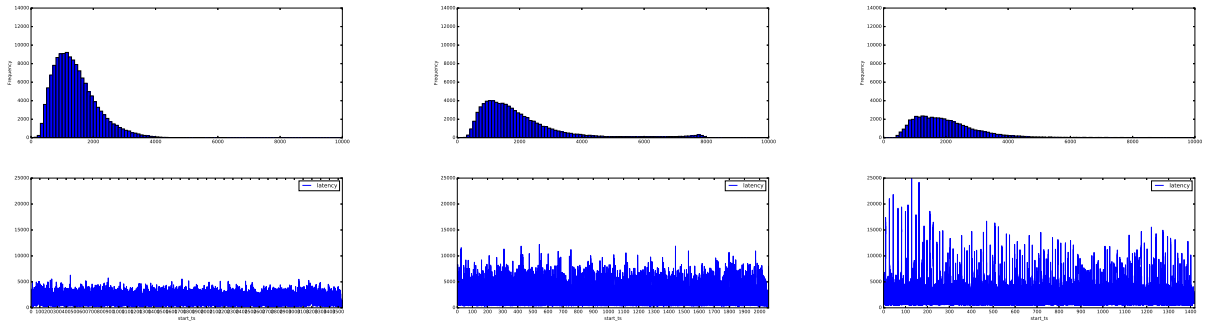
	2 Node	4 Node	8 Node
Storm	xxx	xxx	XXX
Spark	365K	632K	947K
Flink	851K	1128K	*1260K

Table 3: Sustainable throughput for windowed joins

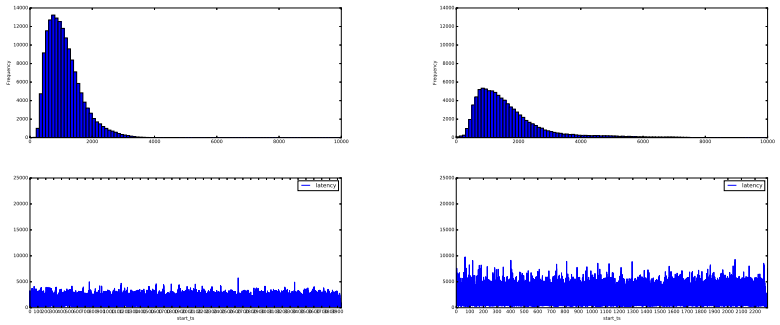
	2 Node	4 Node	8 Node
Storm	1424ms	2043ms	XXX
Storm(90%)	1109ms	1669ms	XXX
Spark	8417ms	7823ms	7590ms
Spark(90%)	7538ms	5825ms	6015ms
Flink	4825ms	4255ms	*3748ms
Flink(90%)	3872ms	3285ms	-

Table 4: Average Latency for windowed joins

6.2.1 Storm



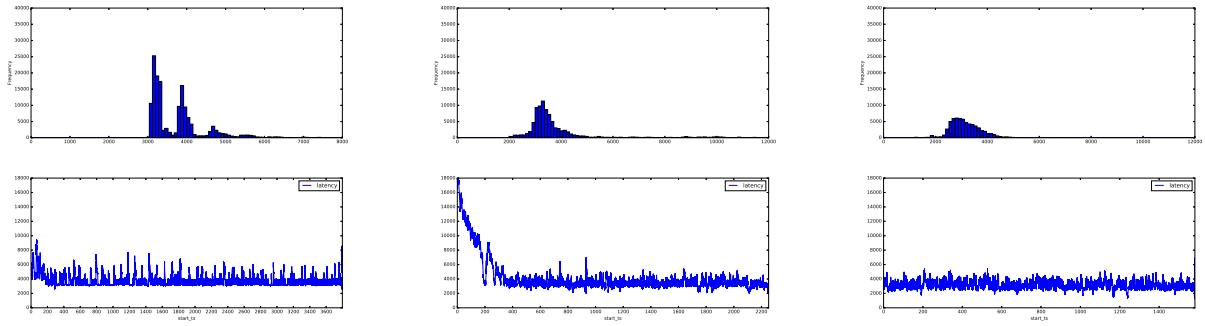
(a) 2 Node latency with max throughput (b) 4 Node latency with max throughput (c) 8 Node latency with max throughput



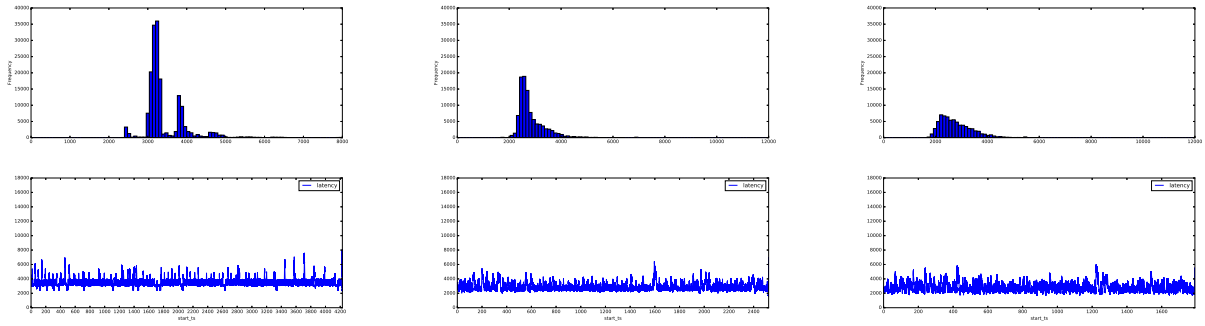
(d) 2 Node latency with 90% throughput (e) 4 Node latency with 90% throughput

Figure 5: Latency of windowed aggregations for Storm

6.2.2 Spark



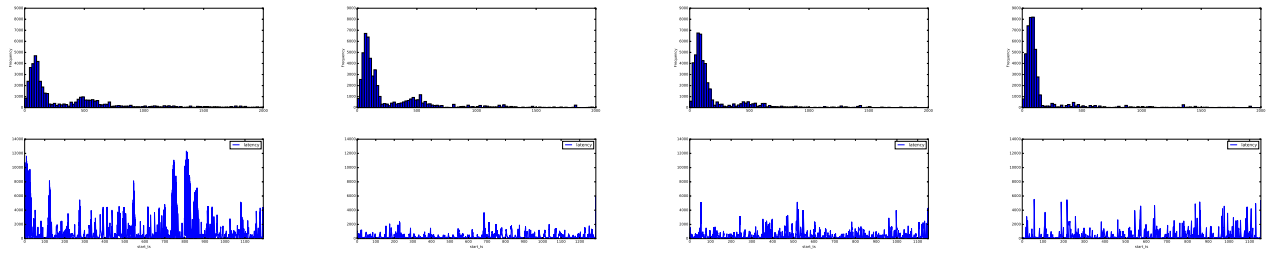
(a) 2 Node latency with max throughput (b) 4 Node latency with max throughput (c) 8 Node latency with max throughput



(d) 2 Node latency with 90% throughput (e) 4 Node latency with 90% throughput (f) 8 Node latency with 90% throughput

Figure 6: Latency of windowed aggregations for Spark

6.2.3 Flink



(a) 2 Node latency with max throughput

(b) 2 Node latency with 90% throughput

(c) 4 Node latency with network bounded thorughput

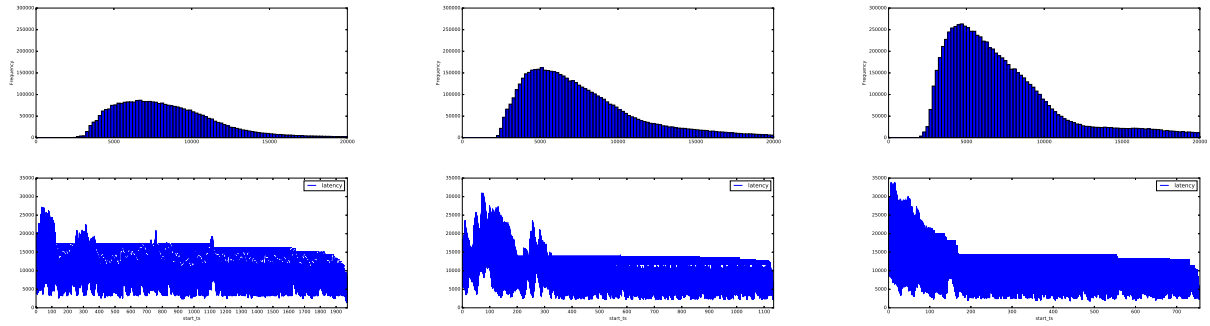
(d) 8 Node latency with network bounded thorughput

Figure 7: Latency of windowed aggregations for Flink

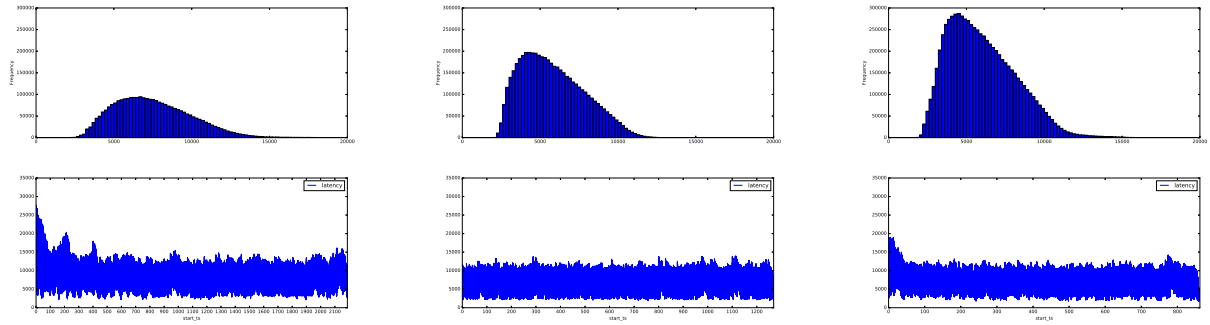
6.3 Joins

6.3.1 Storm

6.3.2 Spark



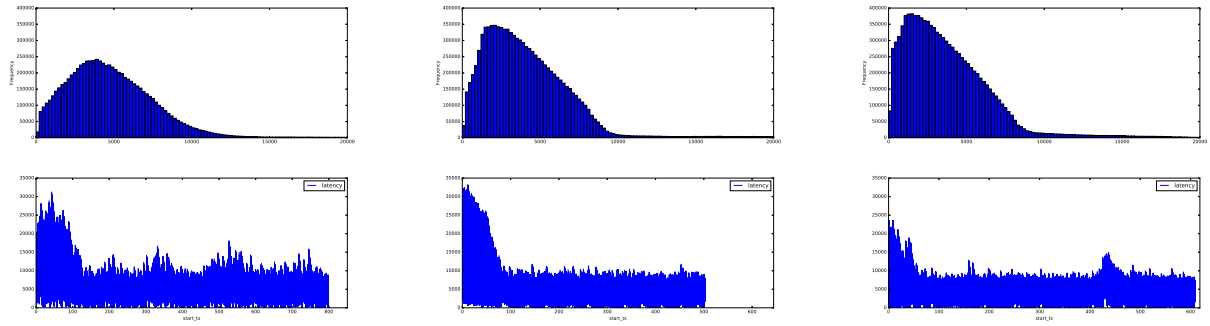
(a) 2 Node latency with max throughput (b) 4 Node latency with max throughput (c) 8 Node latency with max throughput



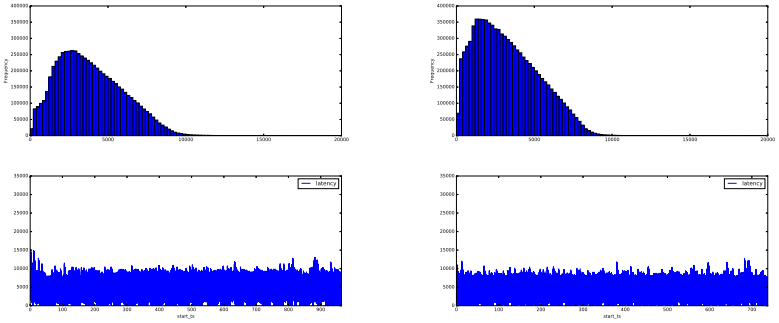
(d) 2 Node latency with 90% throughput (e) 4 Node latency with 90% throughput (f) 8 Node latency with 90% throughput

Figure 8: Latency of windowed join for Spark

6.3.3 Flink



(a) 2 Node latency with max throughput (b) 4 Node latency with max throughput (c) 8 Node latency with max throughput



(d) 2 Node latency with 90% throughput (e) 4 Node latency with 90% throughput

Figure 9: Latency of windowed join for Flink

7. CONCLUSIONS

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] 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—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[2] 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, 2013.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[4] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[5] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[6] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[7] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[8] DataArtisans. Extending the Yahoo! Streaming Benchmark. `http://data-artisans.com/extending-the-yahoo-streaming-benchmark/`, 2016. [Online; accessed 19-Nov-2016].

[9] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208. ACM, 2013.

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[11] A. Kafka. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of*, 5(1), 2014.

[12] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[13] M. A. Lopez, A. Lobato, and O. Duarte. A performance comparison of open-source stream processing platforms. In *IEEE Global Communications Conference (Globecom), Washington, USA*, 2016.

[14] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[15] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez. Spark versus flink: Understanding performance in big data analytics frameworks. In *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*, 2016.

[16] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[17] S. Perera, A. Perera, and K. Hakimzadeh. Reproducible experiments for comparing apache flink and apache spark on public clouds. *arXiv preprint arXiv:1610.04493*, 2016.

[18] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[19] A. Shukla and Y. Simmhan. Benchmarking distributed stream processing platforms for iot applications. *arXiv preprint arXiv:1606.07621*, 2016.

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

[21] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, 2014.

[22] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[24] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.