Benchmarking Distributed Stream Processing Engines [Experiments and Analyses]

Jeyhun Karimov[1,4] Tilmann Rabl[1,2,5] Asterios Katsifodimos[1,5] Roman Samarev[2,6]
Henri Heiskanen [3,7] Volker Markl[1,2,5]

[1]DFKI, Germany, [2]TU Berlin, Germany, [3]Rovio Entertainment

[4]jeyhun.karimov@dfki.de, [5]{rabl, asterios.katsifodimos, volker.markl}@tu-berlin.de, [6]samarev@acm.org,

[7]henri.heiskanen@rovio.com

ABSTRACT

Over the last years, stream data processing has been gaining attention both in industry and in academia due to its wide range of applications. To fulfill the need for scalable and efficient stream analytics, numerous open source stream data processing systems (SDPSs) have been developed, with high throughput and low latency being their key performance targets. In this paper, we propose a framework to evaluate the performance of three SDPSs, namely Apache Storm, Apache Spark, and Apache Flink. Our evaluation focuses in particular on measuring the throughput and latency of windowed operations, such as windowed aggregation and join. For this benchmark, we design workloads based on real-life, industrial use-cases. The main contribution of this work is threefold. First, we give a definition of latency and throughput for stateful operators. Second, we completely separate the system under test and driver, so that the measurement results are closer to actual system performance under real conditions. Third, we build the first driver to test the actual sustainable performance of a system under test. Our detailed evaluation shows that there is no single winner, but rather, each system shines in individual use-cases.

1. INTRODUCTION

Processing large volumes of data in batch is often not sufficient in cases where new data has to be processed fast to quickly adapt and react to changes. For that reason, stream data processing (SDP) has gained significant attention. The most popular engines used for SDP, with large-scale adoption by industry and the research community, are Apache Storm [25], Apache Spark [29], and Apache Flink [10].

One important application area of SDP is online video games. These require the fast processing of large scale online data feeds from different sources. Windowed aggregations and windowed joins are two main operations that are used to monitor user feeds. One typical use-case is tracking the in-application-purchases (IAPs) per application, per distribution channel, or per product item (in-app products). Another typical use-case is the monitoring of advertising: making sure that all campaigns and advertisement networks work flawlessly. Yet another use-case is comparing different user feeds by joining them. For example, monitoring the IAPs of the same game downloaded from different distribution channels and comparing users' actions are essential in online video game monitoring.

In this work, we propose a benchmarking framework to accurately measure the performance of stream data processing engines. For our experimental evaluation, we test three publicly available open source and community driven engines, namely Apache Storm, Apache Spark, and Apache Flink. We measure the latency and throughput as the major performance indicators. Latency, in SDP, is the time difference between data production at the source (e.g., the mobile device) to result output at the sink of the data flow graph describing the stream processing operations (e.g., the monitoring solution on the game server). Throughput, in this scenario, determines the number of ingested and successfully processed records or events per time unit.

Even though there have been several comparisons of the performance of SDPS recently, these do not measure the latencies and throughput that can be achieved in a production setting. One of the repeating problems in the previous evaluations is a missing definition and inaccurate measurement procedure for latency of stateful operators in SDPS. Another challenge is a missing separation between the system under test (SUT) and the benchmark driver. Frequently, the performance metrics are measured and calculated within SUT; this means that the results will be influenced through the measurements and, thus, can be biased. We discuss in detail these challenges as well as our solutions in Section 4.

In this paper, we address the above mentioned challenges. The proposed solution is generic, has a clean design with clear semantics, and can be applied to any SDPS. The main goal is to stimulate an environment in which we can measure the metrics more precisely and with minimum influence of side factors.

The main contributions of this paper go as follows:

- We introduce a mechanism to accurately measure the latency of stateful operators in SDPSs. We apply the

proposed method to use-cases featuring windowed aggregations and windowed joins.

- We accomplish the complete separation of the test driver from the system under test (SUT).
- We measure the maximum sustainable throughputs of the SDPSs. Our benchmarking framework handles system specific features like backpressure to measure the maximum sustainable throughput of a system.
- We use the proposed benchmarking system for an extensive evaluation of Storm, Spark, and Flink with practical use-cases.

The remainder of this paper is organized as follows. In Section 2, we survey the related work. We give an overview of the stream data processing engines benchmarked in this paper in Section 3. In Section 4, we discuss the detailed interpretation of stream benchmarking challenges and their importance. We provide the design of our benchmarking system, the use-cases, and the metrics in Section 5. After a detailed evaluation in Section 6, we conclude in Section 7.

2. RELATED WORK

Benchmarking distributed data processing systems has been an active area of research. Huang et.al. proposed HiBench [15], which was the first benchmark suite to evaluate and characterize the performance of Hadoop. In a more recent version, the benchmark also features a streaming component [27]. The authors conduct a wide range of experiments from micro-benchmarks to machine learning algorithms. A highly related benchmark is SparkBench, which features machine learning, graph computation, SQL queries, and streaming applications on top of Apache Spark [17]. Building an application level, end-to-end big data benchmark with all major characteristics in the lifecycle of big data systems is the main goal of BigBench [14]. Wang et al. introduce BigDataBench, the big data benchmark suite for Internet services. The BigDataBench suite contains 19 scenarios covering a broad range of applications and diverse and representative data sets [26].

Benchmarks on SDPSs extend the batch data processing benchmarks and pose unique challenges. Recently, researchers from Yahoo! conducted a series of experiments on stream data processing engines to measure their latency and throughput [12]. They used Apache Kafka [16] and Redis [11] for data fetching and storage. Later on, on the other hand, it was shown that those systems actually are a bottleneck for the tested systems' performance [13]. In this paper we overcome this bottleneck and measure event time latency. Marcu et al. did an extensive analysis of the differences between Apache Spark and Apache Flink by correlating the execution plan with resource utilization and parameter configuration [20]. The authors conduct experiments with batch and iterative workloads on up to 100 nodes. Their key finding is similar to ours that none of the systems outperforms the other for all data types, sizes and job patterns. However, the authors leave benchmarking the SDP features of the systems for the future work, which we present in this paper.

Perera et al. use Karamel [4] to provide reproducible batch and SDP benchmarks of Apache Spark and Apache Flink in a cloud environment [22]. We can identify the similarity in the results in Spark and Flink's memory usages between that particular work and our paper. However, for other metrics such as CPU usage and latency, we notice significant differences. The authors used the Yahoo stream benchmarks for evaluation and it was shown above that the particular benchmarking framework have bottlenecks; thus, this can be a possible reason for differences in evaluation results. In Section 4 we analyze this challenge and provide our low-overhead solution to overcome this bottleneck. Lopez et al. propose a benchmarking framework to assess the fault tolerance and throughput efficiency for the open source stream data processing engines Storm, Spark and Flink [18]. The key finding of the paper is that the micro-batch stream processing system, Spark, is more robust to node failures, i.e., it efficiently stores the full processing state of the micro-batches and distributes the interrupted processing with low-overhead among other worker nodes. On the other hand, Spark performs up to 15 times worse than Storm and Flink. Compared the particular paper with our work, there are differences in terms of the throughputs of the systems. In our paper, we analyze the *unreal throughput* as being one of the challenges and provide our solution, *sustainable throughput*. Shukla et al. perform common IoT tasks with different stream data processing engines, evaluating their performance [24]. Firstly, the authors define the latency being an interval between source operator's ingest and sink operator's emitting time. As we will discuss in Section 4, this approach may lead to unrealistic results. Thus, we provide a solution for the event time latency. Secondly, the authors define throughput as the rate of output messages emitted from the sink operators in unit time. However, the output throughput may not be useful in various real-life use-cases. In our paper, we introduce sustainable throughput and provide a low-overhead solution to measure it.

Lu et al. propose StreamBench, a solution to measure the throughput and latency of a SDSP by putting mediator system between the data source and SUT [19]. In Section 4, we show that the mediator system can be a bottleneck and resulting benchmarks may not exhibit precise results. Thus, we provide an efficient solution to use local, distributed, and efficient mediator system. The LinearRoad benchmark was proposed by Arasu et al. to measure the performance of standalone stream data management systems such as Aurora [6] by simulating a toll system scenario for motor vehicle expressways [9]. The authors use response time, which is the time difference between input's arrival and output from SDPS, as a metric. As stated above, this definition of latency can lead to unrealistic results; thus, we provide low-overhead solution to measure event time latency. Several stream processing systems implement their own benchmarks to measure the system performance without comparing them with any other system [21, 23, 29].

3. PRELIMINARIES AND BACKGROUND

In this section, we provide background information about the stream data processing engines and their features used in this paper. We analyze Apache Storm, Apache Spark, and Apache Flink as they are the most mature and accepted ones in both academia and industry.

3.1 Apache Storm

Apache Storm is a distributed stream processing framework, which was open sourced after being acquired by Twitter [2]. Storm operates on tuple streams and provides record-by-record stream processing. It supports an at-least-once processing semantics and guarantees all tuples to be processed.

In failure cases, events are replayed. Storm also supports exactly-once processing semantics with its Trident abstraction [5]. Stream processing programs are represented by a computational topology, which consists of spouts and bolts. Spouts are source operators and bolts are processing and sink operators. A Storm topology forms a directed acyclic graph (DAG), where the edges are tuple streams and vertices are operators (bolts and spouts). When a spout or bolt emits a tuple, the bolts that are subscribed to this spout or bolt receive input.

Storm's lower level APIs provide little support for automatic memory and state management. Therefore, choosing the right data structure for state management and utilizing memory efficiently by making computations incrementally is up to the user. Storm supports caching and batching the state transition. However, the efficiency of a particular operation degrades as the size of the state grows.

Storm has built-in support for windowing. Although the information of expired, newly arrived, and total tuples within a window is provided through APIs, the incremental state management is not transparent to the users. Trident, on the other hand, has built-in support for partitioned windowed joins and aggregations. Storm supports sliding and tumbling windows on processing time and event time. For event-time windows, tuples need to have a dedicated timestamp field so that the engine can create periodic watermarks. Any worker process in Storm topology sends acknowledgements to the source executor for a processed tuple. In case of failure Storm sends the messages again. One of the downsides of Storm's use of acknowledgments is that the tuples can be only be acknowledged once a window operator completely flushes them out of a window. This can be an issue on windows with large length and small slide.

Backpressure is one of the key features of SDPSs. It refers to the situation where a system is receiving data at a higher rate than it can process. For example, this can occur during temporary load spikes. Storm supports backpressure although the feature is not mature yet [3]. This was confirmed throughout our experiments as well. Storm uses an extra backpressure thread inside the system. Once the receiver queue of an operator is full, the backpressure thread is notified. This way Storm can notify all workers that the system is overloaded. Due to its high complexity, Storm's backpressure feature can stall the system and, therefore, it is not enabled by default in the current version.

3.2 Apache Spark

Apache Spark is an open source big data processing engine, originally developed at the University of California, Berkeley [1]. Unlike Storm and Flink, which support one record at a time, Spark Streaming inherits its architecture from batch processing, which supports processing records in micro-batches. Throughout this paper, we refer to Spark Streaming as simply Spark. The Resilient Distributed Dataset (RDD) is a fault-tolerant abstraction of Spark, which enables in-memory, parallel computation in distributed cluster environments [28].

One of Spark's features is its support of lazy evaluation. This enables the engine to run more efficiently.

Spark supports stage-oriented scheduling. Initially, it computes a DAG of stages for each submitted job. Then it keeps track of materialized RDDs and outputs from each stage, and finally finds a minimal schedule.

Unlike Flink and Storm, which also work based on DAG execution graphs, Spark's computing unit in a graph (edge) is a data set rather than streaming records and each vertex in a graph is a stage rather than individual operators. RDDs are guaranteed to be processed in order in a single DStream (Discretized Stream), which is a continuous sequence of RDDs. However, there is no guaranteed ordering within RDDs since each RDD is processed in parallel.

Spark has improved its memory management significantly in the recent releases. The system shares the memory 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.

Spark has a built-in support for windowed calculations. It supports only windows defined by processing time. The window size must be a multiple of the batch interval, because a window keeps a particular number of batches until it is purged. Choosing the batch interval can heavily affect the performance of window-based analytics. First, the latency and response time of windowed analytics is strongly relying on the batch interval. Second, supporting only processing time windowed analytics, can be a severe limitation for some use-cases.

Spark also supports backpressure. It handles backpressure by putting a bound to the block size. Depending on the duration and load of each mini-batch job, the effectiveness of backpressure signal handling from source to destination may vary.

3.3 Apache Flink

Apache Flink started off as an open source big data processing system at TU Berlin, leveraging in major parts the codebase of the Stratosphere project [9].

At its core, Flink is a distributed dataflow engine. Like in Storm, a 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 serializing data into memory segments. The data exchange in distributed environments is achieved via buffers. A producer takes a buffer from the pool, fills it up with data, then the consumer receives the data and frees the buffer informing the memory manager. Flink provides a wide range of high level and user friendly APIs to manage state. Incremental state update, managing the memory, or checkpointing with big states are performed automatically and transparently to user.

Flink has a strong feature set for building and evaluating windows on data streams. With a wide range of pre-defined windowing operators, it supports user-defined windows with custom logic. The engine provides processing-time, event-time, and ingestion-time semantics. When using 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. When using event time, on the other hand, the notion of time is determined by the timestamp attached to an event, which typically resembles its creation time. Like in Storm, the timestamps must be attached to each event record as a separate field. At ingestion time, the system processes records with event time semantics on these timestamps. Flink

has support for out-of-ordered streams, which were motivated from Googles MilWheel and Dataflow papers [7, 8]. Flink also supports backpressure. It uses blocking queues. Once the congestion is detected this information is automatically transferred to upstream operators at negligible cost.

4. CHALLENGES

There are several challenges to be addressed when designing a benchmarking framework for SDPSs. In this section, we analyze these challenges and explain our solutions.

Separate driver and tested system. The first challenge is to isolate the driver from the SUT as much as possible. In previous works, researchers measured the throughput either inside the SUT, or used internal statistics of the SUT. However, if the measurements and SUT are not separated, then the measurement computations can influence the results of benchmark. In our benchmarking framework, we separate the driver and SUT for each measurement metric and perform measurements separately from the SUT.

The first metric is throughput. If the system can ingest and process all the data produced by the driver instances during the whole experiment, then the system can *sustain* the given throughput and we call it *sustainable throughput*. If the system sustains the given throughput and cannot process more, then we call it *maximum sustainable throughput*. In this paper, we analyze the maximum sustainable throughput of SDPSs which is sum of the throughputs of all driver instances. So, we keep throughput assessment inside driver instances and sum them at the end of experiment.

The second evaluation is latency. We define the latency of a tuple to be the interval between tuple's event time and emission time from SUT's sink operator. We develop a model to calculate the latency of stateful operator which we analyze in Section 5.1.2.

Many systems continuously collect metrics inside the SUT while the application is running. Then, the question arise whether we need the driver to be separated from SUT. Firstly, once we accept metric measurement inside the SUT, benchmarking different systems becomes a challange especially if the semantics of metric measurements are different among systems or even the semantics is unknown if the system is blackbox. Secondly, the measuring metrics, latency and throughput in our case, inside the system and performing so with clear separation from system gives completely different results, as we can see in Section 6.

Simplicity is key. The second challenge is to design a simple benchmarking framework as complex benchmarking frameworks can cause additional overheads.

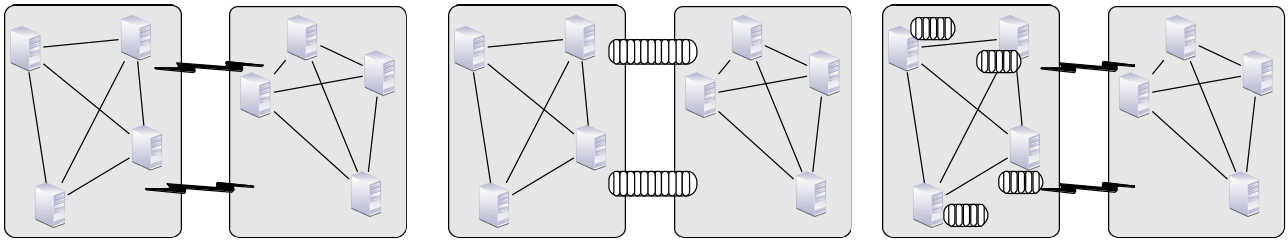
For example, the connection between the data generator and the SUT can cause large additional overheads when benchmarking. To test the stream data processing engine, a data generator component is essential, to provide large amounts of streaming data. Figure 1 shows three possible cases to link the data generator and the SUT. The simplest design is to connect the SDPS directly to the data generators as shown in Figure 1a. Although this is a perfectly acceptable design, it does not match real-life use-cases. In large-scale setups, stream data processing engines do not connect to push-based data sources but pull data from distributed message queues. A pull based design, where data sources and SUT are connected through queues, is in Figure 1b. A common bottleneck of this option is the throughput of the message queuing system. Also, this adds a de-/serialization layer between the SUT and the data sources. Therefore, we use a third option, which is a hybrid of the first two. As can be seen in Figure 1c, we embed the queues as a separate module in the data generators. This way, the throughput is bounded only by the network bandwidth and the systems work more efficiently as there are no de-/serialization overheads.

The pull and push based data ingestion approaches perform theoretically the same with sustainable workloads. However, there are two issues to consider from practical perspective. Firstly, as we mentioned above, pull based approach is the one used in industrial setup. No matter how high the workload a system can sustain, it always has an upper limit. As a result, using push based approaches leaves the user with no guarantee for possible high workloads. Secondly, it is not trivial to find the maximum sustainable workload for a particular system with push based data ingestion approach. As a result, if the data ingestion rate is above the maximum sustainable workload the engine may fail to perform the experiments. If we are lucky, we can detect this type of exception within benchmarking time interval. However, ideally this case requires infinite-length experiments.

Unreal latency and throughout. The third challenge is to measure the metrics with close to real-life scenarios. One example is the throughput measurement. In the previous SDPS benchmarks, the throughput of a SUT is measured by either taking quantiles over the test duration, or showing max, min, and average results. From a user's perspective on the other hand, the system's throughput is the upper limit throughput that it can sustain in a realistic setup. We propose user-defined sustainability policies in our benchmarking framework. For example, the benchmarking framework can take into account the backpressure of the SUT.

Another example for this challenge is latency measurement. If there are additional systems between the SDPS and data source (driver), then those are likely to add an extra latency for each tuple. One solution is to measure the latency in the SDPS the same way as it is measured in batch data processing systems. In this case, the number and complexity of the systems between SDPS and data source is irrelevant as the latency is the interval between tuples' ingestion and output from SUT. However, this measurement of latency is too *optimistic* and does not result in the real latency of the tuple. The reason is that the actual latency is based on tuples' event generation time. For example in online games, the event time can be a user's specific action in a game, the latency then is the time interval between user's action and the result being emitted from the SDPS. Therefore, the usual method of measuring the latency within SDPS does not conform to real world scenarios especially in pull-based SDPSs. Depending on their backpressure mechanism, a pull-based SDPS will reduce the input rate on high loads, which will not be reflected in the measured latency. To solve this, we define the latency for SDPSs to be the difference between tuple's event time and output time.

Another factor triggering the unrealistic latency is the data generation rate. If the input data rate is higher than the SDPS's *maximum sustainable throughput* and we measure the event time latency, the results may not exhibit the real latency. Initially the system will try to ingest as much data as possible. As a result, the SUT's processing time will take longer and the latency will increase for every adjacent tuple. To solve this issue, we conduct experiments with the maximum sustainable throughput for each SDPS.



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

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

Latency of stateful operators The final challenge is measuring the latency of stateful operators. Up to this point, the related work either concentrated on stateless operators or evaluated the latency of stateful operators by checkpointing to external systems. As we discussed above, this approach can be a bottleneck and will influence the measurements. In our benchmarking framework, we propose a solution to this problem. We aggregate tuples by selecting their maximum timestamp and append the resulting timestamp to the emitted tuple from stateful operator. In this way, we measure the latency of stateful computation time, excluding the tuples' waiting time (in window) in stateful operator.

5. BENCHMARKING FRAMEWORK DESIGN

We discuss the design of the benchmarking framework in this section. As shown in Figure 2, there are two main components of test deployment: *1)* the SUT and *2)* the driver. The driver has two subcomponents being *i)* Data Generator and *ii)* Data Queue.

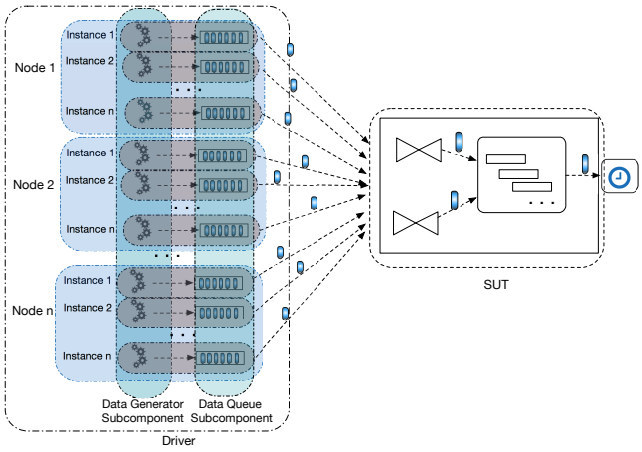


Figure 2: Design of the benchmark framework.

The Driver Instance (DI) is a combination of the Data Generator Instance (DGI) and the Data Queue Instance (DQI). The driver is the combination of all DIs. Similarly, the Data Generator Subcomponent (DGS) and Data Queue Subcomponent (DQS) are the combination of all DGIs and DQIs, respectively. The driver is responsible for generating and queuing the data. It is composed of finite number of instances which are distributed evenly to the worker nodes in the cluster. The driver nodes are separate from SUT nodes in the cluster deployment. The DGI and the DQI reside in the same machine to avoid any network overhead and to ensure data locality. The data is kept in memory to avoid disk write/read overhead. The first subcomponent, the DGS, generates data with a constant speed. Because of the bottlenecks explained in Section 4, we avoid using mediator data queueing systems between the driver and SUT but use distributed local queues (DQS). Data is queued in first-in/first-out manner in the DQS. As explained earlier, this approach is not different from implementing centralized and distributed message queues between the SUT and the driver. The following analogy can be made: the queue topic in a distributed message queuing system is analogous to DQS, and the queue partition is analogous to DQI.

The data generator timestamps every event. The event's latency is calculated from this point and the longer it stays in a queue, the higher its latency. The number of DIs can be arbitrary and the overall throughput that can be generated is only bounded by the network bandwidth.

The use-cases for this benchmark are derived from an online video game applications. Online video game companies continuously monitor the user actions in a game and ensure that the system works as expected. For example, the company tracks the number of active users and take actions on a sudden drop. Moreover, once the new feature is added to an online game or the game is updated to a new version, the company monitors the application to see the newly added feature is working without failures.

Windowed aggregations are an important part of the user monitoring in online video gaming. Online video game companies track the IAPs per application, per distribution channel(the application store, e.g. iTunes/appStore, Google Play), per product item (the actual IAP item like gem pack) and etc. In this benchmark, we use similar use-case monitoring the IAPs per product item.

Windowed joins are another important part of user monitoring in online video gaming. Analysing the IAPs within a game and comparing the results based on distribution channel, user ID, and etc. is another use-case in online video game industry. In our benchmark, we get user feeds per distribution channel and compare the IAPs of the same product item between streams. That is, we divide the input streams into windows and join them per key (product item).

5.1 Metrics

The metrics for this benchmark are latency and throughput. In this section, we give the definition of each metric and explain our solution to measure it.

5.1.1 Throughput

As we defined above, we use the maximum sustainable throughput to measure system's workload. Throughout the experiments, the data generation speed in all DIs are equal and constant. To examine if a system can sustain a given throughput, we divide the queue used in DQI into three parts: q^a , q^b and q^n. Figure 3, shows the example partitioning of the queue. If the size of the queue is less than or equal to c^a throughout the experiment, then this is acceptable, meaning the SUT can sustain the given throughput. If the queue size is between q^a and q^b on the other hand, the SUT cannot sustain the given data rate, but the driver might tolerate it for some time. A longer queue can be caused by slow system initialization, backpressure, etc. However, if the queue size is longer than q^b then the SUT cannot sustain the given throughput and the latency is expected to continuously increase. In this case we end the experiment.

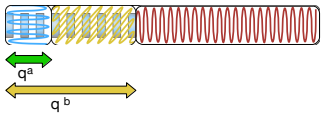


Figure 3: Basic intuition behind *backpressure-compatible queue*

```
S_1 = UNION  {s_1, s_2, ..., s_m}
S_2 = UNION {s_m, s_m+1, ..., s_n}
S_3 = UNION  {s_1, s_2, ..., s_n}

Query_1 = SELECT AVG(S_1.price)
FROM S_1 on window(1, s)
GROUPBY S_1.geo

Query_2 = SELECT MAX(S_2.price, S_3.price)
FROM S_2, S_3 on window(1, s) ,
WHERE S_2.geo = S3.geo and S_2.ts = S3.ts

Query_3 =
1.UPDATE S_1, S2: enrich with a new field =
   cnt,  the country name of geo location
2.SELECT AVG(R.price)
  FROM
   (SELECT MAX(S_2.price, S_3.price)
    FROM S_2, S_3 on window(1, s),
    WHERE S_2.cnt = S3.cnt and S_2.ts = S3.ts)
    as R on window(1, s)
  GROUPBY R.geo
  WHERE R.price > 100
```

Listing 1: Queries

Our system supports user-defined policies to test the sustainability. For example, one policy tolerates the q^b part of the queue for a given time period (time based policy). Another policy tolerates the q^b part of the queue for a given number of pushes into the queue (count based policy).This policy is customizable and can be easily plugged into the driver.

We choose a policy that is a combination of time and count based policies. Once the queue size exceeds the q^a and is less than q^b, we wait until $\frac{q^b-q^a}{2}$ elements are pushed into the queue and check again. If the queue size is still bigger than c^a we end the benchmark concluding the SUT cannot sustain the given throughput. This process is done continuously in all DIs and if the SUT fails to sustain one instance of the driver, then the experiments are halted meaning the SUT cannot sustain the given throughput. That is, the SUT can sustain a given workload if it can sustain the throughput of all DIs.

It is crucial that separation of driver and SUT and handling backpressure in the driver side does not cause side effects in metric measurements. To emplasize again, the driver does not stop the benchmark when the system exhibits backpressure. Firstly, accurate selection of q^a, q^b and q^n is crucial. In our experiments, first we assign 5%, 15% and 30% of input size for variables q^a, q^b and q^n respectively. In this case, the system can "handle" the given input; however, with longer experiments it fails, as the size of the queue keeps increasing. Moreover, we get very big latencies for the given input. As a result, we keep decreasing the variables until we find the best thresholds for all systems. Secondly, it is crucial for the driver to distinguish unsustainable throughput and backpressure. As we discussed in above paragraph, the driver can successfully handle this issue. If the queue size gets beyond q^a and less than q^b, the driver starts to inspect the queue for the next $\frac{q^b-q^a}{2}$ tuples. If the queue size keeps increasing, for every such tuple, then we conclude that this is not a backpressure. In fact in experiments, we show that our driver can successfully handle the backpressure.

5.1.2 Latency

While the use-case provides the semantics for aggregating or joining tuples, the semantics for measuring the latency of stateful operators is unclear. We provide efficient solution for this problem in our benchmarking framework. When the tuples arrive to a stateful operator, it does the aggregation that is defined in the use-case. Besides the aggregation defined in the use-case, we aggregate the timestamp fields of tuples within stateful operator. That is, the timestamp of stateful operator's output tuple is the maximum timestamp of all tuples inside the operator. The main intuition is that we exclude the tuples' waiting time in the window by taking the timestamp of the latest arrived tuple in each window. When a record is emitted from the sink operator of the SUT, we calculate the latency of a tuple by subtracting its timestamp from the current time.

We are aware of the fact that new definition of latency can be applied when the data is pushed to the system.

6. EVALUATION

In this section, we discuss our experimental results. We run experiments in 2-, 4-, and 8-node clusters to measure the scale out feature of the systems under test. Throughout the experiments we used Storm 1.0.2, Spark 2.0.1 and Flink 1.1.3. The experiments are done with the maximum and 90% sustainable throughputs. Throughout the experiments, each DI generates 100M events and we use 16 parallel DIs in the cluster. We allocate 16 CPUs and 16GB RAM for every node. All nodes' system clocks in the cluster are synchronized via an NTP server. We generate events with normal distribution.The network bandwidth is 1Gb/s. We only use processing time windows in the experiments as it is the only window type that all three systems support. We conducted experiments with event time windows with Storm and Flink; however, the results were very similar to the experiments with processing time windows; therefore, we provide only latter in this paper. We select q^a to be 5% and q^b 10 % of the overall input. Although we did experiments with lower values of the

particular variables, it was hard to detect the systems' maximum sustainable throughput. The reason is that the systems under test use pull-based mechanism to get data from DQS and and the smaller the queue size the more difficult to test SUT's upper limit workload. We use approximately 25% of the input data as a warmup. So, in all experiments we exclude the first 25% of output. The backpressure is enabled in all systems to ensure the durability of experiments. That is, we do not want the systems to ingest more input than they can process and crash during the experiment. In preliminary experiments, measuring the maximum throughput without backpressure mechanisms lead to various exceptions. If the SUT drops one or more connections to the DI, then the experiment is halted with the conclusion that the SUT cannot sustain the given throughput. Similarly, in real-life if the system cannot sustain the user feed and drops connection, then this is considered as a failure.

Tuning the systems. Tuning the engines' configuration parameters is important to get a good performance for the given use-cases. There are several properties for each SDPS, that need to be tuned to customize the given use-case. For example, in Flink the buffer size has to be adjusted properly to ensure a good balance between throughput and latency. Although selecting low buffer sizes can result in low system latency, the actual latency of tuples may increase as they will be queued in the DQI instead of the buffers inside the system. In Spark the block interval for partitioning the RDDs is one key aspect to tune the system for the given use-case. The number of RDD partitions a single mini-batch is bounded by $batchInterval$ / $blockInterval$. As the cluster size increases, decreasing the block interval can increase the parallelism. One of the main reasons that Spark scales up very well is the partitioning of RDDs. However, depending on the use-case, the optimal number of RDD partitions can change. In Storm the number of workers, executors and buffer size are the the configurations (among many other) that needs to be tuned to get the best performance. Similar to Flink, choosing the buffer size is a key to balance between latency and throughput. For all systems, choosing the right parallelism level is essential to balance between a good resource utilization and network or resource exhaustion.

Windowed operations. When talking about the windowed queries, we should take the system architecture into the consideration. For example, Spark gathers the data in window and then parallelizes the computation within a single window. Basically, it enables users to query a single window. We call it global windows. Flink and Storm on the other hand, first partitions the data and then creates windows. As a result, user queries much smaller(depending on data skew) partitioned windows. However, there is no coordination between partitioned windows. As a result, there is no guarantee that one set of windows will be processed after another. We call this approach partitioned windows. In all previous works, this difference was neglected. Spark supports only global windows but Flink and Storm supports both types. Throughout our evaluations, we evaluate both cases for Flink and Storm and compare with Spark.

Partitioned Windowed Aggregations. In the first use-case, we use partitioned windowed aggregations in Storm, Spark, and Flink. We calculate the maximum sustainable throughput for each SDPS. To confirm that the engines are saturated, we performed the same experiments with 90% of the maximum throughput. We calculate the latency measurements for each system over the respective throughputs.

The maximum sustainable throughput of the SDPSs are shown in Table 2. We use a four second batch-size for Spark, as it can sustain the maximum throughput with this configuration. We identified that for 4- and 8-node configurations, Flink's performance is bounded by network bandwidth. Storm's and Spark's performance in terms of throughput are comparable, with Storm outperforming Spark by approximately 8% in all configurations. One reason for Spark's worse performance can be the overhead of starting periodic mini-batch jobs. Storm and Flink, on the other hand, are operating on tuples and, therefore, can sustain higher sustainable throughputs. We want to stress that the sustainable throughput is not the same as an engine's peak throughput. As stated above, we are interested in measuring maximum sustainable throughputs of SUTs in this paper. We adjusted Spark's block interval to achieve a better parallelism on RDD level and the highest sustainable throughput. Storm introduced backpressure feature in recent releases; however, it is not mature yet. With high workloads, it is possible that the backpressure stalls the topology, causing spouts to stop emitting tuples. Moreover, we notice that Storm drops some connections to the DIs when tested with high workloads with backpressure disabled, which is not acceptable according to the real world use-cases. Dropping connections due to high throughput is considered a system failure.

Table 1 shows the latency measurements of windowed aggregations. We conduct experiments with the maximum and 90%-workloads and report avg, min, max, and quantiles (90,95,99) over the results. The latencies shown in this table are computed with the workloads given in Table 2. In most cases, where the network bandwidth is not a bottleneck, we can see a significant decrease in latency when lowering the throughput by 10%. This shows that the maximum throughput saturates the system. For example, Flink's metrics shown in Table 1 change more than the metrics of other systems when lowering the throughput. The reason is that it pulls the data from DQS more periodically than other systems so that the queue size does not get beyond the limits. Spark's 2-node configuration, on the other hand, exhibits negligible difference in latency measurements between the maximum and 90% sustainable throughputs. The reason is that for that configuration, Spark cannot pull the data from DQS periodically and therefore the queue size goes beyond accepted limits. As a result, we do the benchmarks with lower throughputs. This also shows the efficiency of backpressure feature in systems under test. The smoother the backpressure and the lower the overhead of backpressure, the easier to saturate the system with sustainable throughput.

Flink has the best min and avg latencies. Although its max latency is way above its min, from quantile values we can conclude that those values can be considered as outliers. The main reason for having such a high max latency is associated with the buffer size. The large buffer size enables high throughput; on the other hand, it can cause some tuples to have high latencies. For example, with a larger buffer size the tuple resides in buffer longer until the buffer gets filled and flushes. In the 4- and 8-node cluster configurations, we can see that there is a slight difference in Flink's latency statistics between the maximum and 90%-throughputs. The reason is that this workload is not the maximum sustainable throughput but is bounded by the network bandwidth.

	2-node	4-node	8-node
Storm	1407, 66, 5758, (2330, 2721, 3477)	2058, 115, 12216, (3724, 5856, 7767)	2253, 179, 17762, (3818, 6467, 9245)
Storm(90%)	1100, 84, 5752, (1832, 2176, 2859)	1676, 43, 9280, (2985, 4140, 6330)	1932, 182, 11040, (3324, 5005, 7630)
Spark	3673, 2537, 8596, (4603, 4953, 5990)	3394, 1987, 6994, (4076, 4315, 4955)	3139, 1223, 6946, (3896, 4162, 4711)
Spark(90%)	3416, 2354, 8005, (3936, 4550, 5443)	2822, 1630, 6925, (3473, 3767, 4804)	2781, 1702, 5961, (3624, 3948, 4834)
Flink	563, 4, 12328, (1448, 2282, 5233)	265, 4, 5132, (612, 1242, 2479)	258, 4, 5469, (574, 1228, 3975)
Flink(90%)	310, 3, 5826, (698, 1185, 2009)	250, 4, 5112, (601, 1334, 2430)	219, 2, 5405, (482, 843, 3458)

Table 1: Latency statistics, avg, min, max, and quantiles (90, 95, 99) in milliseconds for windowed aggregations.

	2-node	4-node	8-node
Storm	408K	696K	992K
Spark	379K	642K	912K
Flink	1230K	1260K	1260K

Table 2: Sustainable throughput for windowed aggregations.

	2-node	4-node	8-node
Spark	365K	632K	947K
Flink	851K	1128K	1190K

Table 3: Sustainable throughput for windowed joins

As we see from Table 1, Spark has more latency than Storm and Flink but it exhibits less of a difference among the *avg*, *min*, and *max* latency measurements. Because it processes tuples in mini-batches, the tuples within the same batch have similar latencies; therefore, there is no huge difference among measurements. Moreover, transferring the data from Spark's block manager to DStream by creating RDDs is another overhead that results in Spark's higher *avg* latency compared to Flink and Storm. Although the *avg* and *max* latencies increase in Storm with increasing workload and cluster size, in Spark we see the opposite behavior, which means Spark can partition the data (RDDs) in bigger distributed environments. However, from the quantile values we can conclude that the *max* latencies of Storm can be considered as outliers.

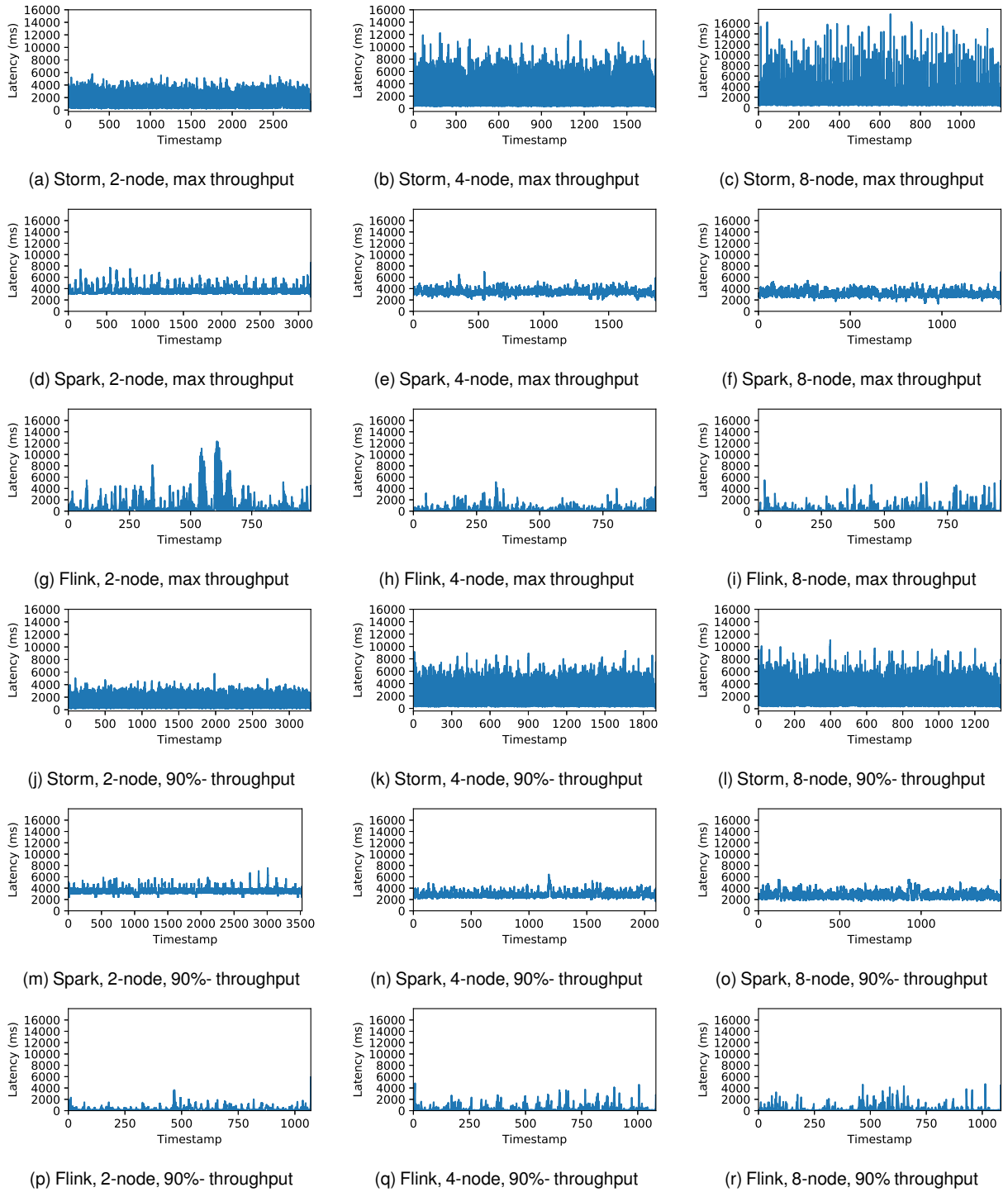
Global Windowed Aggregations. As we discussed above, in global windowed aggregations the system collects the data and parallelizes the query for the data inside window. As a result, the system should be able to manage and parallelize large size windows. Because Spark only supports global windowed aggregations, we enable the global window aggregation semantics in Flink and Storm and compare with Spark. In Flink and Storm, for all types of windowed queries, the performance of the system is bounded by the performance of a single slot of a machine, meaning it does not scale. The reason lies under the design option of a system. That is, supporting querying querying big windows and coordinating the sub-queries of a given query is costly. For Query- 1, we measured the throughput 480K for Flink and 200K for Storm. We can see that on 2 nodes Flink's single threaded global windowed aggregation beats over Spark and Storm. However, Spark imrpoves the performance as it scales as we can see from Table 2.

Figures **??** and 4 show the windowed aggregation latency distributions as histogram and time-series, respectively. In all cases we can see that the fluctuations are lowered when decreasing the throughput by 10%. While in Storm and in Flink it is hard to detect the lower bounds of latency as they are close to zero, in Spark the upper and lower boundaries are more stable and clearly noticeable. The reason is that a Spark job's characteristics are highly dependent on the batch size and this determines the clear upper and lower boundaries for the latency. The lower the batch size, the lower the latency and throughput. To have a stable and efficient configuration in Spark, the mini-batch processing time should be less than the batch interval. We determine the most fluctuating system to be Flink in 2-node setup and Storm in 8-node setup as shown in Figures 4g and 4c. Those fluctuations show the behaviour of backpressure. For the same experiment with 90% of the maximum throughput, we notice that the fluctuations are significantly reduced as shown in Figures 4p and 4l. As we discussed above, 4- and 8-node experiments in Flink are network bound; therefore, these experiments do not fully utilize the Flink nodes and thus have a lower variance in latency.

Figure 5 shows the resource usages of the SUTs. The upper figure shows the CPU load during the experiment. The CPU load is the number of kernel level threads that are runnable and queued while waiting for CPU resources, averaged over one minute. The next figure shows the network usage of the SUTs. Because the overall result is similar, we show the of systems' resource utilization graphs with windowed aggregation case in 4-node cluster. Because Flink's performance is bounded by network, we can see that CPU utilization is minimal among others. Storm, on the other hand, uses approximately 50% more CPU clock cycles than Spark. This results in having competitively higher throughout than Spark. One reason behind Spark's efficient usage of CPU is that Spark automizes most of the work, handles resource usages transparent to user, and does a lot of optimizations internally. For example, Spark handles incremental state management, optimizations with code generation and dynamic memory management efficiently and transparent to user.

Event time vs System time latency. In this paper we introduce new definition of latency for SDPSs. It is crucial to conduct comparative analysis of event time and system time latency. System time latency is the time interval between tuple's ingestion to SUT and its emission from the sink operator of SUT. Event time latency, on the other hand, is the end-to-end latency between tuple's tuple's event time and its emission time from the sink operator of SUT. Figure 6 shows the comparison between the two cases. We conducted this experiments with Query-1, window length = 8 sec, window slide = 4 sec. Event with small cluster size we can see, there is significant difference between event and system time latency. To be more specific, Storm's *avg* system time latency is 1234ms (event time is 1407ms), Spark's is 1040ms (even time is 3394ms) and Flink's *avg* system time latency is 254ms (event time is 563ms). We examined the similar behavior with large cluster size and with join queries as well.

More throughput than a system can sustain. Providing maximum sustainable throughput to SDPSs is essential as otherwise the performance of the systems can degrade. We experienced on average 5-10 % performance decrease on 20%



(a) Storm, 2-node, max throughput (b) Storm, 4-node, max throughput (c) Storm, 8-node, max throughput

(d) Spark, 2-node, max throughput (e) Spark, 4-node, max throughput (f) Spark, 8-node, max throughput

(g) Flink, 2-node, max throughput (h) Flink, 4-node, max throughput (i) Flink, 8-node, max throughput

(j) Storm, 2-node, 90%- throughput (k) Storm, 4-node, 90%- throughput (l) Storm, 8-node, 90%- throughput

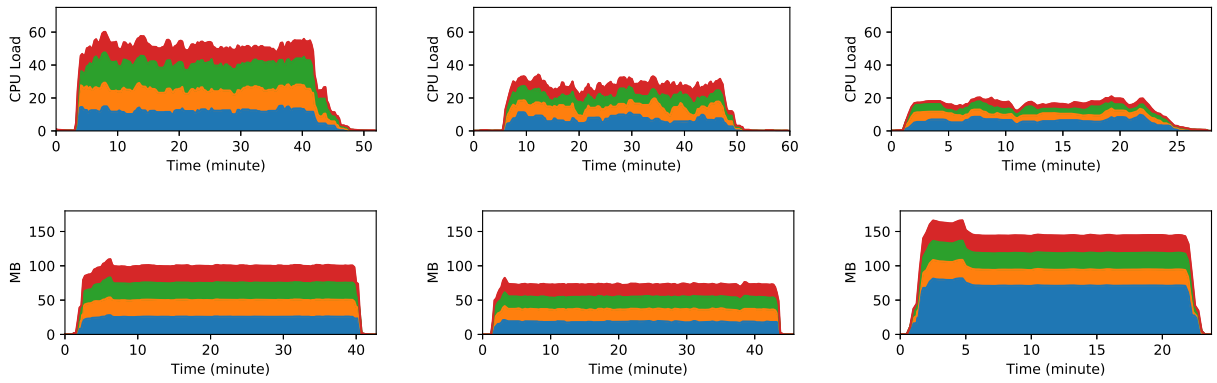
(m) Spark, 2-node, 90%- throughput (n) Spark, 4-node, 90%- throughput (o) Spark, 8-node, 90%- throughput

(p) Flink, 2-node, 90%- throughput (q) Flink, 4-node, 90%- throughput (r) Flink, 8-node, 90% throughput

Figure 4: Windowed aggregation latency distributions in time series

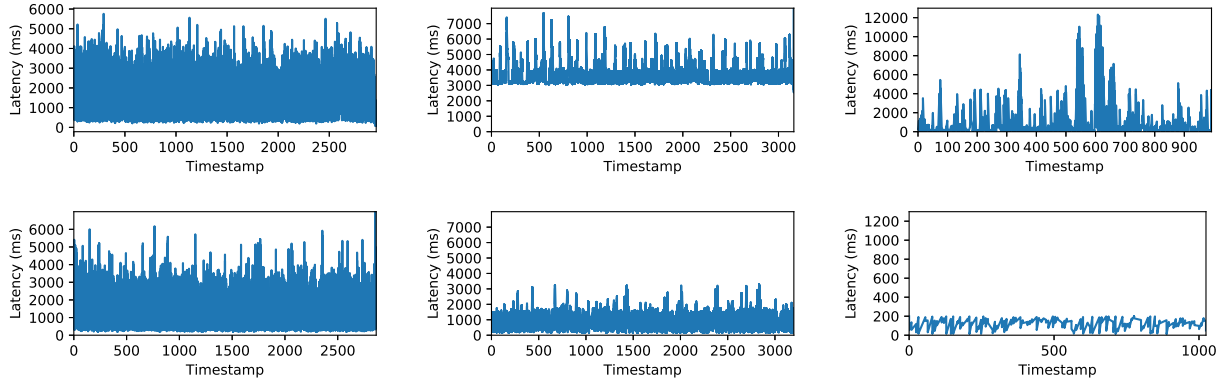
	2-node	**4-node**	**8-node**
Spark	7724, 1376, 21600, (11297, 12409, 14718)	6730, 2139, 23647, (10279, 11788, 15410)	6263, 1841, 19993, (9465, 10469, 13242)
Spark(90%)	7195, 2157, 17981, (10316, 11178, 12702)	5834, 1861, 13956, (8741, 9525, 10729)	5788, 1721, 14197, (8678, 9454, 10625)
Flink	4367, 17, 18230, (7601, 8570, 10554)	3671, 19, 13872, (6756, 7531, 8664)	3275, 22, 14934, (6241, 7045, 8436)
Flink(90%)	3866, 19, 13027, (6786, 7569, 8759)	3243, 16, 12702, (6130, 6901, 8011)	3240, 16, 14978, (6199, 6995, 8304)

Table 4: Latency statistics, avg, min, max and quantiles (25, 50, 75) in milliseconds for windowed joins.



(a) CPU and network usage of Storm (b) CPU and network usage of Spark (c) CPU and network usage of Flink

Figure 5: CPU and Network usage of systems in a 4-node cluster. Colors indicate nodes in a cluster

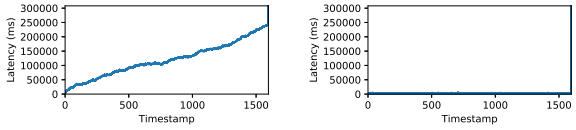


(a) TODO (b) Event vs system time latency in Spark (c) Event vs system time latency in Flink

Figure 6: Comparison between event and system time latency among systems

more throughput than the system can sustain. We did experiments with Query-1 and Query-2 with window length between 5-10 second, and window slide between 1-5 seconds. We experienced that as the window size gets large and window slide and buffer size gets smaller, Storm's probability to drop the sockets increases. In general the decrease in overall throughput is dominated by the time interval where the system is unstable, meaning the system cannot fire the backpressure effectively. For example, we executed Query-1 with window length = 8 seconds and window slide = 4 seconds on 2 nodes. The overall throughput of the Storm, Spark and Flink decreased by 6%, 5%, 3% respectively. With the same parameters on Query-2, we examined approximately 1.5 times more performance decrease with 20% more throughput. Moreover, if the data is skewed the performance loss can be in orders of magnitude (in Spark, Figure 7) or even system failure (in Flink).

Impact of window size to systems' performance. The window size and window slide has significant impact on system's performance with windowed operations. Here we talk about partitioned windowed operations as Spark is single winner in global windows in scale. One interesting fact we inspected is that with the increasing window size Spark's throughput decreases significantly given the same batch size. For example, with window length and slide 60 seconds, Spark's throughput decreases by 2 times and avg

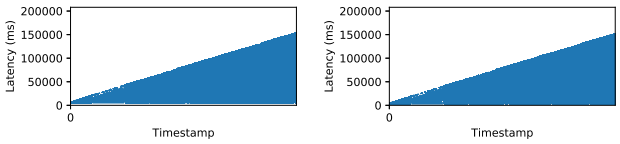


(a) System time latency (b) Event time latency

Figure 7: Event vs system latency with more workload than Spark can sustain

latency increases by 10 times given the batch size 4 seconds. However, with higher batch sizes, Spark can handle much higher workloads and big windows. Storm on the other hand, can handle the big window sized operations if the user has advanced data structures that can spill to disk when needed. Otherwise, we encountered memory exceptions. Flink (as well as Spark) has built-in data structures that can spill to disk when needed. However, this do not apply for the operations inside the UDFs, as Flink treats UDFs as blackbox.

Impact of data skew to systems' performance. Data skew is another factor affecting the system performance. Especially in industrial scenarios one should not expect purely equal data distribution. We tested the systems with extreme skew, which is with single key. Here, we treat Flink's global



(a) Event time latency (79175.5) (b) System latency (77181.6)

Figure 8: Event vs system latency with skewed data

windowed performance equivalently for skewed data with one key. The reason is that, Flink internally uses static dummy key to partition global windows. The similar approach is used in Storm as well. Spark on the other hand, can handle skewed data efficiently. We experienced Spark outperforming both engines with Query-1 (8sec, 4sec) in a cluster with 4 nodes or more. For skewed joins on the other hand, both Spark and Flink stugle to handle. Flink often got stuck and stayed unresponsive in this case. Spark on the other hand, performed with very high latencies. We decreased the join operator's selectivity to make sure that data transfer is not a bottleneck. The main reason is that the memory is consumed quite fast and backpressure mechanism lacks to perform efficiently. As can be seen from Figure 8, although very high latency the system, Spark in this case, ingests the input continuously.

Data pulling from data source. While analyzing the systems' performance it is essential to inspect their data pull graphs from data sources. We retrieved this metrics from driver side to ensure that we are not affecting the system performance. Figure 9 shows the corresponding behavior of systems with maximum sustainable throughout on Query-1 (8sec, 4sec). However we examined the similar behavior in other window settings and windowed operations as long as the workload is maximum sustainable. As we can see, Spark and Storm have fluctuating data pull rates. Despite having high data pull rate, Flink have less fluctuations. Once we lower the workload both Flink and Spark gets to monotonic data pull rate; however, Storm still exibits significant fluctuations.

Peaks. Especially in industrial scenario, peaks are inevitable. We simulated the peaks for all systems both for Query-1(8 sec, 4 sec) and Query-2 (8sec, 4sec). We start the benchmark with 840K per second workload then decrease it to 280K and increase again after a while. As can be seen from Figure 10, Storm is the most susceptible system for peaks. Spark and Flink have comparative behavour with partitioned windowed aggregations case. However, for partitioned windowed join case, Flink can handle peaks better. To emphasize the event and system time latency again, we show the system time latency of Figure 10e in Figure 10d.

Another scenarios To test the systems' behavour deeply, we conducted more complex scenario with Query-3. The main purpose is to inspect engines' query optimization with non-trivial queries. Examining the logical and physical query graph, we inspect that Flink and Spark perform query optimization techniques to increase the performance of system. Therefore, the performance of systems stay approximately with the same proportion, with a more complex query. The same can be said about lower workloads. We tested systems with lower workloads than they can sustain. One interesting behaviour was that Flink's avg latency increase with the smaller workloads. So we had to tune the buffer size for each workload to lower the latency; however, in real industrial scenario this is not the case. In Spark however, the avg latency improved with decreasing workloads. However, it can be further improved with smaller batch size. For this, again we have to stop the job and tune the system for each workload, which is unrealistic in real industrial scenarios. Number of keys

Windowed Joins In the second use-case, we use partitioned windowed joins in Spark and Flink. Storm provides a windowing capability but there is no built-in windowed join operator. Initially, we tried Storm's Trident abstraction, which has built-in windowed join features. However, Trident computed incorrect results as we increased the batch size. Moreover, there is a lack of support for Trident in the Storm community. As an alternative, we implemented a simple version of a windowed join in Storm. However, comparing it with Spark and Flink, which have advanced memory and state management features, leads to unfair comparisons. We implemented a naïve join in Storm and examined the maximum sustainable throughput to be 140K events per second and measured an average latency of 2349 milliseconds on a 2-node cluster. However, we faced memory issues and topology stalls on larger clusters. As a result, we focus on Flink and Spark in our windowed join benchmarks.

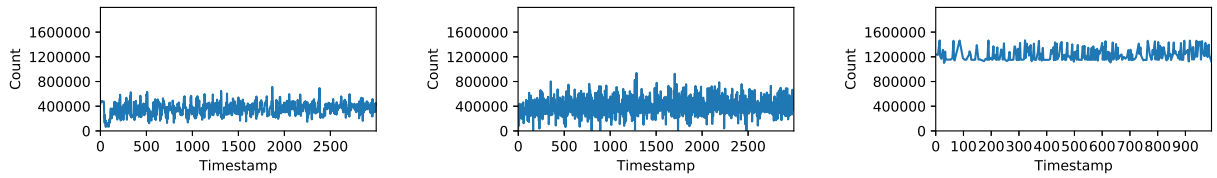
Throughout the experiments, we found important points that need to be taken into consideration while performing windowed joins over streams. First, depending on the selectivity of the input streams to the join operator, vast amount of results can be produced which can limit the speed of the engine to the disk I/O. We are assuming that the result sink is a file system. Second, the vast amount of results of a join operator can cause the network to be a bottleneck. This can happen if we output the results to shared file system. If the result size is large, then the network becomes a bottleneck. To address this issue, we decreased the selectivity of the input streams. In general, the experimental results for windowed joins are similar to the experiments with windowed aggregations.

Table 3 shows the maximum throughput that the systems under test can sustain. Flink's throughput for 8-node configuration is bounded by network bandwidth. This is 1256K in windowed aggregations. The reason for the difference is that there is more network traffic as the result size is larger in windowed joins than in windowed aggregations. Table 4 shows the latency statistics for windowed joins. We can see that in all cases Flink outperforms Spark in all parameters. To ensure the stability of the system, the run time of each mini-batch should be less than batch size in Spark. Otherwise, the queued mini-batch jobs will increase over time and the system will not be able to sustain the throughput. However, we see from Table 3 that the latency values are higher than mini-batch duration (4 sec). The reason is that we are measuring the event time latency.Therefore, if the latency is higher than mini-batch duration, then the tuples wait in the DQS.

Figures **??** and 11 show the latency for the windowed joins case as histogram and time-series . In contrast to windowed aggregations, we see the substantial fluctuations in Spark in Figures 11a, 11b and 11c. Also we can see a significant latency increase in Flink when compared to windowed aggregation experiments. The reason is that windowed joins are more expensive than windowed aggregations. However, the spikes are significantly reduced with 90% workload.

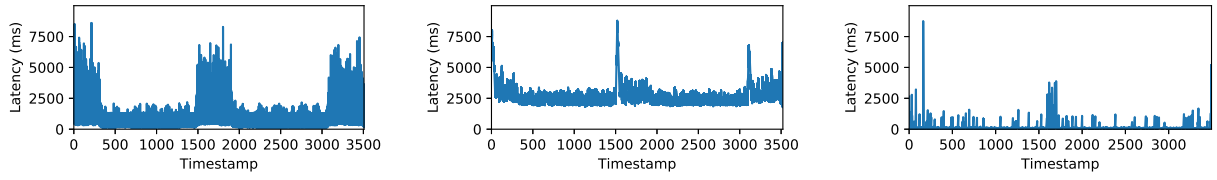
6.1 Discussion

Throughout the experiments, we notice that Storm has a comparable performance to Spark. Storm could provide more high level abstractions with transparent on/of-heap memory

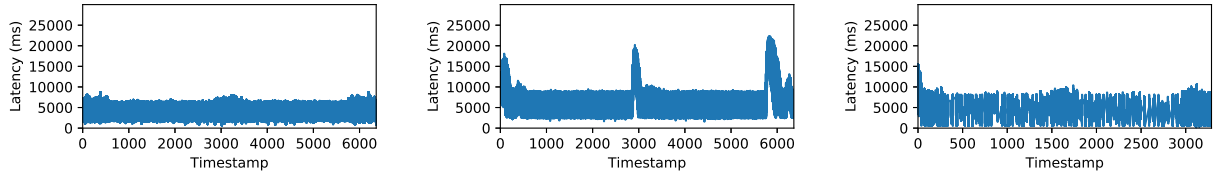


(a) Storm (b) Spark (c) Flink

Figure 9: Data pull graphs of systems



(a) Storm aggregation (b) Spark aggregation (c) Flink aggregation



(d) Spark system time latency (e) Spark join (f) Flink join

Figure 10: Peaks

management in the engine. We can see from resource utilization graphs that the usage of low level APIs results to high CPU and network usage in system. Although Trident is a high level abstraction on top of Storm, its efficiency is much lower than Storm and with large batch sizes, it produces wrong results. Moreover, Storm's backpressure mechanism needs further enhancements as it can stall the topology with high throughputs.

Although Spark performs worse than Storm in terms of throughput, its high level APIs and advanced and the dynamic memory management is better than Storm and competitive with Flink. One downside of Spark is that it schedules one job at a time. So there is only one active job at a time. Although there is an experimental feature to start multiple mini-batch jobs at a time, this can lead to unexpected sharing of resources. Moreover, Spark's backpressure mechanism was significantly improved in recent releases but there is still need for further improvement.

Flink performs best in throughput and in *avg* latency. However, its *max* latency is worse in comparison to other systems. Flink could add a dynamic buffer management feature to change the size of buffers at runtime to reduce the latency. We noticed the performance of Flink's backpressure to be the best because of its robustness and its low overhead.

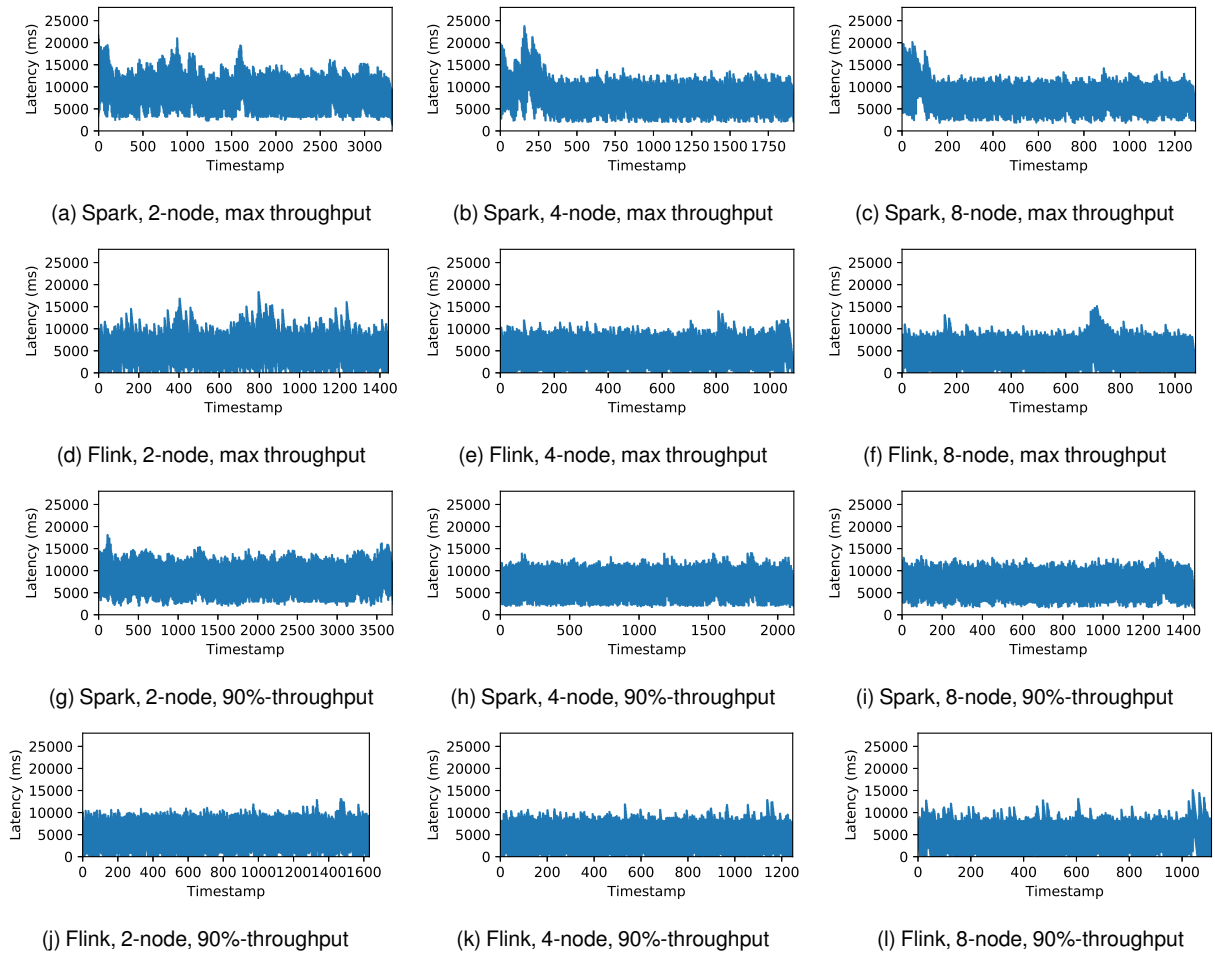
There is no single winner according to the experimental results. Storm's and Spark's throughputs in windowed aggregations are comparable but in terms of *avg* latency Storm is better. So if a user needs to use lower level APIs to implement custom business logic, then Storm is the best engine for windowed aggregation. However, Spark reduces latency when scaling out. So, when working with very large clusters Spark can outperform Storm. Flink on the other hand, provides the highest throughput and the lowest *avg* latency with its transparent and high level APIs. However, its high *max* latency is a limitation when compared to other systems.

7. CONCLUSIONS

Responding to an increasing use of real time data processing in industry and in academia, we build a novel framework for benchmarking stream data processing engines. We identify the current challenges in this area and build our solution to solve them. First, we give the definition of latency of a stateful operator and a methodology to measure it. The solution is lightweight and does not require the use of additional systems. Second, we completely separate the systems under test from the driver. Third, we introduce a simple and novel technique to conduct experiments with the highest sustainable workloads. We conduct extensive experiments with the three major distributed, open source stream processing engines - Apache Storm, Apache Spark, and Apache Flink. In the experiments, we can see that there is no single winner. Each system has specific advantages and challenges. Thus, users need to take use-case requirements into account when choosing the right system. We are currently extending the benchmarking framework to handle several other metrics like query throughput. In addition, we are developing a generic interface that the users can plug into any stream data processing system, in order to facilitate and simplify benchmarking.

8. REFERENCES

[1] Apache Spark. `http://spark.apache.org`. Accessed: 2017-01-28.
[2] Apache Storm. `http://storm.apache.org`. Accessed: 2017-01-28.



(a) Spark, 2-node, max throughput (b) Spark, 4-node, max throughput (c) Spark, 8-node, max throughput

(d) Flink, 2-node, max throughput (e) Flink, 4-node, max throughput (f) Flink, 8-node, max throughput

(g) Spark, 2-node, 90%-throughput (h) Spark, 4-node, 90%-throughput (i) Spark, 8-node, 90%-throughput

(j) Flink, 2-node, 90%-throughput (k) Flink, 4-node, 90%-throughput (l) Flink, 8-node, 90%-throughput

Figure 11: Windowed join latency distributions in time series

[3] Apache storm issue: Disable backpressure by default. `https://issues.apache.org/jira/browse/STORM-1956`. Accessed: 2017-01-17.

[4] Karamel, Orchestrating Chef Solo. `http://storm.apache.org`. Accessed: 2017-01-28.

[5] Storm's Trident abstraction. `http://storm.apache.org/releases/2.0.0-SNAPSHOT/Trident-state.html`. Accessed: 2017-01-28.

[6] 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.

[7] 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.

[8] 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.

[9] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] R. Lu, G. Wu, B. Xie, and J. Hu. Streambench: 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.