StreamBench: Towards Benchmarking Modern Distributed Stream Computing Frameworks

Ruirui Lu[1], Gang Wu[1], Bin Xie[2], Jingtong Hu[3]
ShelleyRuirui@sjtu.edu.cn , wugang@cs.sjtu.edu.cn, xiebin_sh@163.com, jthu@okstate.edu
Shanghai Jiao Tong University[1], Nanjing University of Science and Technology[2], Oklahoma State University[3]

***Abstract*—While big data is becoming ubiquitous, interest in handling data stream at scale is also gaining popularity, which leads to the sprout of many distributed stream computing systems. However, complexity of stream computing and diversity of workloads expose great challenges to benchmark these systems. Due to lack of standard criteria, evaluations and comparisons of these systems tend to be difficult. This paper takes an early step towards benchmarking modern distributed stream computing frameworks. After identifying the challenges and requirements in the field, we raise our benchmark definition StreamBench regarding the requirements. StreamBench proposes a message system functioning as a mediator between stream data generation and consumption. It also covers 7 benchmark programs that intend to address typical stream computing scenarios and core operations. Not only does it care about performance of systems under different data scales, but also takes fault tolerance ability and durability into account, which drives to incorporate four workload suites targeting at these various aspects of systems. Finally, we illustrate the feasibility of StreamBench by applying it to two popular frameworks, Apache Storm and Apache Spark Streaming. We draw comparisons from various perspectives between the two platforms with workload suites of StreamBench. In addition, we also demonstrate performance improvement of Storm's latest version with the benchmark.**

***Keywords*-distributed stream computing; benchmark; big data**

I. Introduction

Data stream refers to a specific kind of data that continuously arrive to be processed without being persisted either in disk or in memory for future random access. Data stream computation has a wide range of scenarios that has been witnessed for many years. Examples of such applications include financial applications, network monitoring, sensor networks and web log mining[1], [2], [3]. Nowadays it is experiencing rapidly expanding data scale because of the growing coverage of sensors, prevalence of the mobile Internet and the climbing popularity of social media such as Twitter and Facebook[4]. Facebook reports handling 10^6 events/s within latency of 10–30 seconds for giving advertisers statistics about users clicking their pages and that their data rate can be up to 9GB/sec at peak[5]. Social networks such as Twitter may also need to identify spam urls in real time at a rate of 15 million URLs/day[6].

This work is done during Ruirui Lu's internship in Intel APAC R&D Center.

Although distributed stream processing systems are under research by academia for over a decade, for example the Medusa projects[7], these efforts are extensions to their single node ancestor Aurora[8] to achieve distributed features. As MapReduce concept was introduced to separate programming logic from concerns of distributed system[9], many distributed stream computing frameworks and systems borrowing concepts from MapReduce more or less have emerged to accommodate increasing data stream volume for better scalability and fault tolerance by both academia and industry. Examples of this kind are S4 from Yahoo[10], Apache Storm[11], Apache Samza[12], Apache Spark streaming[13] and TimeStream from Microsoft[4]. As these systems keep sprouting and maturing, the necessity to evaluate and compare them rises. As with standalone Stream Data Management Systems, Linear Road benchmark[14] is proposed and adopted. However, little evident efforts have been devoted to establishing criteria for these modern distributed stream processing frameworks.

This paper aims to take an early step towards establishing a benchmark for them. In the first place it analyzes the challenges and requirements of building the benchmark. It identifies the wide range of workloads to address and categorizes them in three dimensions, namely target data type, workload complexity and stored state update involvement. Moreover it highlights the challenge of massive data generation on the fly and points out to take both burstiness and durability of stream computing as the target of measurement.

In the second step, we propose our benchmark definition named StreamBench. The definition includes data set selection, data generation methodologies, program set description, workload suites design and metric proposition. We select two real world data sets as seed covering both text and numeric data. As with on-the-fly stream input generation, a message system is utilized as mediator between data generation and consumption to decouple them. The program set containing 7 programs is chosen with regards to the requirements identified and covers all categories mentioned above. Moreover, four workload suites, performance workload suite, multi-recipient performance suite, fault tolerance suite and durability suite are proposed based on the program set to evaluate not only performance, but also other characteristics of modern stream computing frameworks including fault tolerance ability and durability as well. Metrics are also defined in conjunction with workload suites.

We finally implement the program set on Apache Storm and Apache Spark streaming and report the evaluations of the two frameworks applying StreamBench workload suites. According to the results of performance suites, throughput of our Spark cluster can reach to nearly 100MB/s and is about 5 times that of Storm's both under default configurations without tuning while Storm's throughput may catch up when record size grows. Storm's latency in most cases is far less than Spark's, but will exceed Spark when workload complexity and data scale grows. Spark tends to outperform Storm in terms of fault tolerance ability and both two frameworks have successfully gone through their two-day durability test. In addition, our experiment shows Storm 0.9.3 has an average throughput growth of 26% and an average latency drop of 40%, which demonstrates the expected performance improvement between versions.

In summary, our key contributions are as follows:

1) As pioneer work, we point out challenges and state requirements for benchmarking distributed stream computing frameworks. Based on this we present the first formal benchmark in this field named StreamBench.
2) We devise novel ways for stream data preparation and workload selection.
3) Apart from measuring performance, we also take fault tolerance ability and durability into account by introducing various workload suites.
4) We apply StreamBench to Spark Streaming and Storm to conduct a proof of our concept and provide more insights of the two platforms as well.

The remainder of this paper is organized as follows. Section II states the challenges and requirements of stream benchmarks. The definition of StreamBench is described in detail in Section III. Section IV presents the benchmark results carried out on Spark Streaming and Storm as well as different versions of Storm. Section V covers previous related work. Finally, Section VI summarizes the paper and suggests future directions.

II. Challenges and Requirements

As research in this area is relatively in its infancy, it is necessary to clarify and state the challenges faced with benchmarking modern distributed stream computing frameworks. Besides, this section also discusses requirements of benchmarks in this field.

A. Challenges

The first challenge comes from the wide range of operations on data stream. Apart from basic operations including sampling, sketching and filtering[1], [15], different scenarios such as network monitoring[4] and web log mining[1] may involve various operations. Taking all typical workloads into account tend to be not quite practical, especially when most workloads are not published in detail.

Next challenge relates to data generation and consumption. Unlike big data benchmarks whose data can be generated and persisted in advance, stream data are obliged to be generated on the fly. As stream computation frameworks become distributed and may grow to the size of cloud, if the stream input is not scaled or data reception capability of stream processing frameworks is trivially utilized, the computation ability of the whole system may be overshadowed by network bandwidth bottleneck or undersized input, which results in loss of the preciseness of benchmark.

The final aspect to address is both burstiness and durability of stream computation are worth consideration. By burstiness we mean that occasionally the speed of data arrival is fast but lasts for a short period of time while durability means that the stream data constantly arrive to be processed.

B. Benchmark Requirements

This subsection discusses requirements of benchmarking modern distributed stream computing frameworks.

(1) Evaluating and comparing modern distributed stream computing systems and frameworks. First and foremost, the purpose of the benchmark is to evaluate and draw comparisons between different distributed stream computing frameworks. Apart from performance measurements, benchmarks should also highlight the distributed nature of the target frameworks, hence it may take fault tolerance ability, availability and scalability into account.

(2) Identifying the characteristics of stream computation. Stream computation possesses many unique features. For one thing, besides throughput, latency is also a crucial metric of performance. Another feature of stream computation is that it is continuously handling arriving data while in the meantime, it witnesses burstiness, so it's important to profile both durability and maximum capability.

(3) Addressing diverse and representative workloads. As stream computing has a wide range of use cases, establishing fair benchmarks demands incorporating diverse application scenarios and workloads.

(4) Feasibility. Stream computing frameworks are diverse and complex, so are stream workloads. In addition, scaled live data generation is by no means an easy task. Hence, attention shall be paid to making the benchmark easy for operation, that is, easy to deploy and run as well as obtain metrics.

III. Benchmark Definition

In this section, we present the definition of our benchmark of modern distributed stream computing frameworks named StreamBench as well as the rationality of its design.

A. *Our Benchmark Methodology*

As benchmark efforts in the field are relatively immature, we may draw an analogy between big stream computation with big data computation. Big data benchmarks can be roughly classified into two types. The first type is to select relatively easy, representative workloads and data sets from different scenarios to form a synthetic benchmark. Examples of this kind are BigDataBench[16] and HiBench[17]. The second type is end to end benchmarks, that is, they focus only on one scenario but with detailed background from which all the workloads and data sets are born. The representative of this type is BigBench[18], which reuses scenario from TPC-DS[19], a data warehouse benchmark by TPC committee. As with stream related benchmarks, Linear Road benchmark[14], the benchmark targeting at single node stream data management systems, takes the second approach by emulating a highway toll system scenario. For StreamBench, we take the first approach, since stream computing has a wide range of scenarios that need to be addressed after rapidly growing utilization for years and it's also simpler as an early step.

StreamBench selects a program set containing 7 benchmark programs covering three dimensions of categories identified. Based on the benchmark programs, four workload suites are defined targeting at measuring different aspects of stream computing frameworks. As with data generation, StreamBench leverages a message system for data feed and generates both textual and numeric input data at different scales based on two seed data sets.

B. *Seed Data Sets*

Stream data are sent to target systems on the fly. To ensure the input is identical and also closer to real-world applications, StreamBench chooses to generate streams from seed data sets gathered from two main stream computing scenarios, real time web log processing and network traffic monitoring. The data sets cover both text and numeric data. In addition, the record size of the two data sets bears significant difference as record size may have an impact on computation as well as throughput. A summary is in Table I.

Table I: Data Set Summary

Data set	Data type	Record count	Average record size
AOL Search Data	Text	100000	60 bytes
CAIDA Anonymized Internet Traces Dataset	Numeric	11942	200 bytes

AOL Search Data[20]. The AOL Search Data set is a collection of real query log data from real users. The original data set consists of 20M web queries from 650k users. We truncate it to 10,000 lines to ensure two data sets are at the same data scale. Each record consists of 5 fields, namely an anonymous user ID number, query, query time, item rank and click URL.

CAIDA Anonymized Internet Traces Dataset[21]. This data set consists of statistical information of an hour-long internet package trace collected once a month. We concatenate records for 9 months from September 2013 to May 2014. It contains 17 fields, two examples of which are internet data package size and the number of packets of this size using IPv4 protocol.

Although the sizes of the data sets are by no means grand as they only serve as the seed, the actual input stream size is far beyond this, which will be discussed in detail in data generation section.

C. *Workload*

In this section, we first categorize real-world stream computing workloads and then we give the selection of program set based on these categorizations. We also define four workload suites to measure different aspects of distributed stream computing systems.

1) Workload Categorization: To meet the challenge of various workloads, our approach is to categorize the workloads based on features of stream computation and select typical and representative program set that will cover most categories.

The first dimension of categories is concerned with the target data type of computation. Two data types are involved: numeric data and textual data. Numeric data are encountered in some scenarios including financial calculation, sensor monitoring and network monitoring and so on while textual data are involved in web log analysis, social media scenarios and others[1], [2], [3].

The second category is related to the complexity of computation. Some basic stream operations are shared across a bunch of applications, including sample, projection and filter[1], [15]. Extracting these common operations out for testing is essential for understanding any data stream computation frameworks. However, simple operations alone are not sufficient to profile sophisticated applications and thus multi-step operations shall also be considered.

As with the third dimension, one feature of stream processing is integrating stored and streaming data[3], meaning that handling realtime data may require referring to historical information and may result in updating global status either in disk or in memory, hence the benchmark shall take this feature into account as well.

2) Program set selection: StreamBench defines 7 benchmark programs covering the categories identified above, as shown in Table II. The first four benchmarks are relatively atomic. Identity benchmark simply reads input and takes no operations on it. It intends to serve as the baseline of other benchmarks. Sample benchmark samples the input stream according to specified probability. It's a basic operation widely utilized [1], [15]. Projection benchmark extracts a

Table II: Workload Categorization Mapping

Benchmark	Data type	Complexity	Stored state involvment
Identity	Text	Single Step	Not Involved
Sample	Text	Single Step	Not Involved
Projection	Text	Single Step	Not Involved
Grep	Text	Single Step	Not Involved
Wordcount	Text	Multi Step	Involved
DistinctCount	Text	Multi Step	Involved
Statistics	Numeric	Multi Step	Involved

certain field of the input. It is the pre step of a bunch of real world applications as well as some of our workloads such as DistinctCount and Statistics. Grep checks if the input contains certain strings. It's a simple yet common operation also adopted in the evaluation of Spark Stream[13].

Last three benchmarks are more complex. Wordcount benchmark first splits the words, then aggregates the total count of each word and updates an in-memory map with word as the key and current total count as value. It's widely accepted as a standard micro-benchmark for big data[17]. Authors of Spark Stream[13] implemented a stream version of it for evaluation. DistinctCount benchmark first extracts target field from the record, puts it in a set containing all the words seen and outputs the size of the set which is the current distinct count. It's also regarded as one of the common use cases of stream computation and is chosen to test TimeStream framework[4]. Statistics benchmark calculates the maximum, minimum, sum and average of a field from input.

3) Workload suites definition: The complexity of stream computing systems poses great difficulty for evaluating them comprehensively. To profile the performance of stream computing frameworks, previous way is to measure its response time and supported query load[14]. It is quite straightforward to obtain response time by simply calculating the average latency of all input. On the contrary, supported query load is quite hard to obtain since multiple input stream speeds need to be tried to figure out this metric, especially when capabilities of different platforms under different configurations vary vastly. Hence, we take an easier approach by leveraging a message system where generated input records are placed so that the computing frameworks can obtain as fast as they can and we measure the throughput to understand the its best computation velocity. This provides insight for understanding how fast the frameworks can be when faced with bursts. Our basic performance workload suite is designed based on this with only one node in the frameworks serving as data recipient. To further profile the actual processing capability of the stream frameworks and avoid inaccuracy from limitations of the undersized input stream, as has been pointed out previously as one challenge, we propose the multi-recipient performance workload suite.

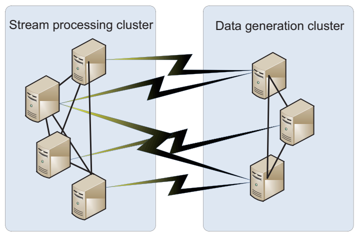
In addition, fault tolerance is one of the key concerns for distributed computing, especially for stream computing where data are not stored. Modern distributed stream computation systems handle fault tolerance with high concern[4], [13], [11], [12] and the recovery ability is worth inclusion. The fault tolerance workload suite is designed for this goal. Besides, stream computing frameworks in most cases aim to provide service constantly, thus the durability workload suite is raised to verify this capability.

Performance workload suite: The performance workload suite incorporates all of the seven benchmark programs. It demands all input data be pushed to the message system where stream computing systems can obtain data as fast as they can and metrics are collected during its computation. Four input data scales are defined, namely 5 million records(baseline), 10 million records(2X), 20 million records(4X) and 50 million records(10X), which are based on the volume of Twitter and Facebook stream computations[5], [6]. Data scale can be extended if necessary for further profiling.

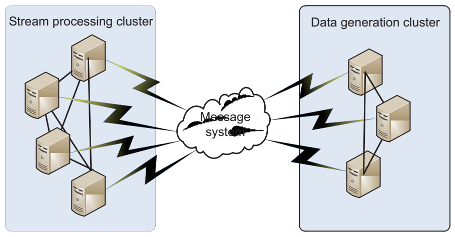
Multi-recipient performance workload suite: This suite contains seven benchmark programs too, but selects only one data scale. It defines a new descriptor of the stream computation cluster, **reception ability**, to be the proportion of nodes that receive input data out of the whole cluster, ignoring their differences of network and other components. The workload suite consists of three reception ability level, single node as recipient, half of total nodes as recipients, that is with the reception ability of 0.5 and all nodes as recipients, which means the reception ability is 1. This workload measures performance under these different configurations to further capture the behavior of the computing frameworks.

Fault tolerance workload suite: This suite also includes all seven benchmark programs and considers only one node failure. It is based on multi-recipient performance workload suite with reception ability of 0.5, which means half of total nodes are recipients. It requires intentionally failing one node which is not a recipient in about the middle of the executions to capture penalty brought by the loss of the compute resource. Since measuring the time to recover tend to be complicated as the system needs to be constantly inspected and judged whether it has been recovered, we take an easy approach for feasibility by simply collecting the performance metrics through the whole process and comparing it with failure free executions. This is a relatively rough measurement as it's also quite hard to tell when the middle of the execution is. We simply consider half of the run time of the failure free experiment done in the previous suite as the expected middle of this suite.

Durability workload suite: The durability workload suite contains only one benchmark program, the Wordcount benchmark. It has two data scales, 10,000 records and 1 million records per minute lasting for two days, which is nearly 30 million and 3 billion records totally. It then reports the percentage of available period of the frameworks.



(a) Without message system



(b) With message system

Figure 1: Two data generation architectures

D. Data Generation Approaches

Stream data is demanded to be generated live, thus separate data sending nodes are needed inevitably. One possible data transmission means may be specifying the mapping of each generator node to computation node, as illustrated in Fig.1(a).

However, we handle it by applying a different means. We propose using a message system as a mediator for data generation and consumption for two main reasons. First, this design provides higher abstraction and decouples data generation from data consumption for better feasibility. Second, it's the actual usage pattern of many companies[22] and LinkedIn Samza[12] even relies on a message system Apache Kafka as real time data feed. It's also behind our performance workload design, as has been explained in Workload suites definition section. The new architecture is shown in Fig.1(b).

In this architecture, two factors may impose a negative impact on reception speed of stream computing frameworks. The first factor is the data generation and publication speed. If stream input data are not published rapidly enough, it will become the bottleneck leading to lame performance measurement of target systems. Hence, we clarify the first requirement of data generation to be (i)data publishing speed should be faster than consumption speed. In an effort to achieve this, our approach to generate input stream data from seed data set is to load the data set into memory with each line as one record and fetch record sequentially in the set and restart from beginning when coming to the end of the set until it meets the required input scale.

The second factor is the message system capability. The message system shall be chosen and configured to be powerful enough in terms of serving data feeds to streaming computation frameworks. So we state the second requirement of data generation to be (ii)Message system should be able to serve data faster than the consumption speed. Data generation process and message system that meets these two requirements regardless of their implementation will prevent them from hurting the preciseness of the benchmark.

This architecture uniforms data generation and serving process. It is possible that Stream computation frameworks may achieve greater throughput if the input source is not the message system. However, if we can ensure that under our current environment and settings maximum reception speed is reached by obeying the two requirements stated above, we may claim that our results are relatively fair since the message system is the common data feed and its potential negative impact is eliminated.

E. Metrics

Metric definition is correlated with workload suite. For performance related work suite, two main metrics, throughput and latency, are proposed to describe the capability of the systems. Throughput metric is the average count of records as well as data size in terms of bytes processed per second. Throughput per node is its derivative metric to capture average capability of each node. The second metric is latency, which is the average time span from the arrival of a record till the end of processing of the record.

New metrics are raised for the fault tolerance workload suite. Suppose originally N nodes are serving, and as described in the workload definition, one node is deliberately failed approximately in the middle of execution and remaining N-1 nodes continue working. Denote the throughput for N nodes without failure to be T_N, throughput for N-1 nodes without failure to be T_{N-1} and the throughput of the workload with failure is $T_{failure}$. We define the throughput penalty factor TPF to be:

$$TPF = \frac{T_{failure}}{\frac{1}{2}(T_N + T_{N-1})}.$$

It's easy to tell that TPF is usually less than 1 and the closer to 1 the better. Similarly latency penalty factor LPF is:

$$LPF = \frac{L_{failure}}{\frac{1}{2}(L_N + L_{N-1})}.$$

where L stands for latency. LPF is likely to be greater than 1 and the closer to 1 the better.

For durability workload suite, the percentage of its available period during the two day run is briefly defined as its durability index. If the service keeps up throughout the workload, the index is 100%.

IV. Evaluations

StreamBench is defined in the conceptual level and is decoupled from implementation. Thus it can be adopted to any stream computing frameworks as long as the program set is implemented according to the programming interfaces of the target frameworks and that all the workloads are applied. In our experiments, Apache Spark Streaming[13] and Apache Storm[11] are chosen as target frameworks. As with message system, we adopt Apache Kafka[23]. This section briefly dips into the configurations of the clusters followed by the evaluation of the systems in terms of StreamBench workload suites. Besides, comparisons of different versions of Storm are also carried out.

A. Experiments Configurations

Table III: Node Configuration Summary

Type name	CPU	Memory
type1	Xeon X5570@2.93GHz 16cores	48G
type2	Xeon E5-2680@2.70GHz 16cores	64G
type3	Xeon E5-2660@2.20GHz 32cores	192G

Two clusters with 12 nodes in total are utilized for the evaluation, one computation cluster for deploying Apache Spark as well as Apache Storm and one input generator cluster for Apache Kafka to send data to computation cluster. The computation cluster contains 6 nodes, one functioning as master (or nimbus for Storm) and others as slaves (or supervisors for Storm). Half of the cluster is type1 and half type2 of Table III. The input generator cluster also has 6 machines, one for Apache Zookeeper providing services for Apache Kafka and Apache Storm and the rest 5 for Kafka broker servers. They are identical in terms of node configuration which belongs to type3 in Table III. Maximum Ethernet link speed of all nodes is 1000Mb/s. Hyper threading is enabled for all tests.

We target at comparing Apache Spark version 0.9.0-incubating and Apache Storm version 0.9.1-incubating. As Apache Storm github master version has evolved to 0.9.3-incubating, we also briefly test on this version to draw comparisons between new and old versions. One thing to note is that we carry out the experiments with default configuration of Spark and Storm without tuning except that we set Spark executor memory size to 35G in our benchmark code. Batch duration of Spark Streaming in our experiments is 1 second. Our reports simply reveal the difference between the frameworks under these configurations for a proof of concept instead of a thorough comparison of the frameworks. We also ensure that the two requirements stated in Data Generation Approaches section are met in our experiments.

B. Workload Suite Execution

All four workload suites are executed on both Apache Storm and Apache Spark Streaming platforms. This section presents the results as well as comparisons of them.

1) Performance workload suite: For Spark, all 4 data scales defined are tested while for Storm 10x data scale is omitted due to too long execution time. As the suite definition indicates, only one node in the computation cluster will receive input data.

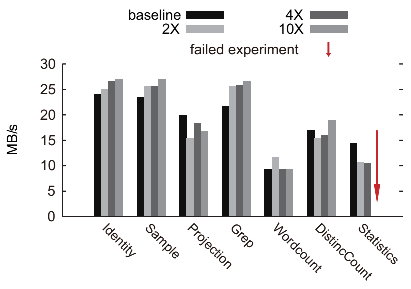
Fig.2 shows the throughput of the cluster in terms of MB/s of the two platforms while throughput in terms of records/s and per node is omitted. One thing to note is that the stream computation frameworks may encounter **saturate volume**, which is the data volume that will overwhelm the frameworks. If the data scale exceeds the saturate volume, the two frameworks both behave abnormally. Saturate volume varies with benchmark program, platform and configuration. In this workload suite, 10X data scale for Statistics benchmark run on Spark meets the saturate volume, thus its throughput is not obtained, as annotated in Fig.2(a).

This suite leads to two observations regarding throughput. **[Observation 1]** Spark's throughput is roughly 4.5 times greater than Storm's on average. This throughput gap is more obvious for basic operations than for more complex workloads. For Statistics, their throughputs are quite close. **[Observation 2]** Spark streaming throughput varies with benchmark programs and the maximum(Identity) throughput is over 2.5 times that of minimum(Wordcount) throughput. In contrast, for Storm it's almost indifferent to various benchmark programs except Statistics. Its throughput is about 3 times that of others due to larger record size.

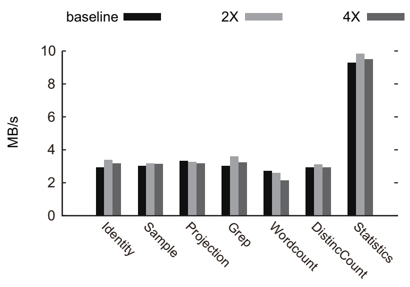
This suite helps to illustrate the throughput gap between the two platforms under current configurations. Different behaviors of Spark and Storm when running various programs further highlight the rationality to include diverse programs in the program set. Besides, according to the observations, record size also has influence on throughput especially for Storm, which supports to incorporate data sets with different record sizes.

Latency for Spark streaming and Storm appears with totally different patterns. Spark latency for different scales is plotted in Fig.3(a) and Storm latency is listed in Fig.3(b) as well as Table IV as the latency for Wordcount of Storm is far larger than others.

We may make the following observations from the graph and table. **[Observation 1]** For most operations, Storm has a much lower latency (less than 20ms) than Spark whose latency is at the scale of 1000ms. However, for workloads like Wordcount under data scale 4x, Storm's latency is about 4 times that of Spark. **[Observation 2]** Spark latency varies for different programs, but in the same scale. Latency of Statistics, which is the highest, is about 5 times of Identity's latency. As with Storm, latency may grow dramatically

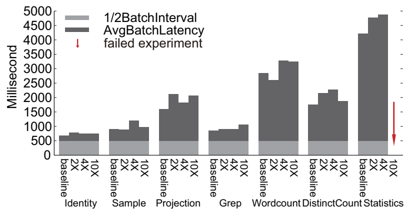


(a) Spark throughput

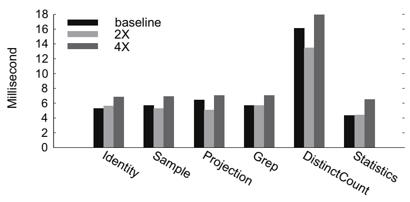


(b) Storm throughput

Figure 2: Throughput of two platforms under multiple data scales



(a) Spark latency



(b) Storm latency

Figure 3: Latency of two platforms under multiple data scales

Table IV: Storm Latency For Wordcount

Program	Baseline(ms)	2X(ms)	4X(ms)
Wordcount	166.827	1900.146	12019.256

with program complexity. Most benchmark programs witness small latency, but Wordcount latency is hundreds even thousands of times of the majority. **[Observation 3]** Data scale does not significantly affect latency on Spark platform as well as most Storm workloads. However for Wordcount, latency grows by a factor of almost 10 along with the increase of data scale.

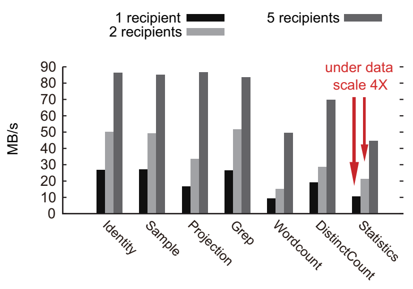
Latency metric of this workload suite reflects another trait of the stream computing platforms in terms of the life cycle of single records, which contributes to revealing differences between the two platforms from another perspective. Besides, data scales impose obvious impacts for Wordcount of Storm, which implies multiple data scales may help to capture diverse behaviors of workloads on different platforms.

2) Multi-recipient performance workload suite: Our cluster of 6 nodes running Spark and Storm includes 1 node as master as well as nimbus and only rest 5 nodes are leveraged for receiving input data and computing. We then did experiments with 2 and 5 nodes as input recipients since reception ability is configured in terms of the 5 nodes rather than 6 nodes. Data scale for this suite is 10X for Spark and 4X for Storm. For Statistics on Spark platform with 1 and 2 recipients, 10X exceeds the saturate volume. Workloads under these two configurations are carried out with 4X scale while for 5 recipients 10X scale runs healthily, as illustrated in Fig.4(a) and Fig.5(a). Some Storm workloads also meet the saturate volume, which is annotated in Fig.4(b), Fig.5(b) and Table V.

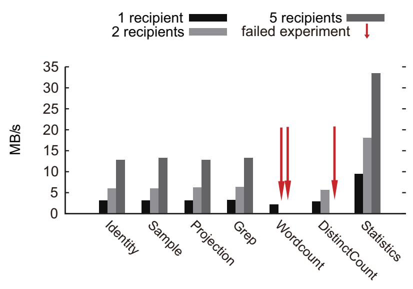
We list our observations from Fig.4 here. **[Observation 1]** Similar with performance workload suite, throughput with multiple recipients for Spark is 4-8 times that of Storm's except for Statistics, which is quite close. Besides, Spark's satire volume for more complex programs in this suite is greater than Storm's. **[Observation 2]** For both Spark and Storm, throughput grows with reception ability although with a slightly smaller multiplier. Throughput for 2 recipients is about 1.8 times as great as that for single recipient while 5 recipients' throughput is roughly 2.15 times that of 2 recipients' for both platforms on average.

Latency metric is also collected in the two platforms as Fig.5 and Table V show. Judging from the graph, latency for Spark doesn't vary much across different configurations in terms of recipients. As with Storm, slight increases are witnessed for 2 recipients configuration compared with single recipient configuration. Latency for 5 recipients shows an obvious growth from 2 recipients' configuration with a multiplier of 9 for Statistics and 5 for others. As having more recipients in the cluster means faster stream input speed, we may deduce that latency for Storm doesn't scale linearly in terms of input speed.

This suite in essence profiles the platforms in terms of faster input speed as half or all nodes are engaged in

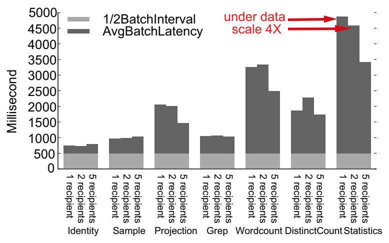


(a) Spark throughput with data scale 10X

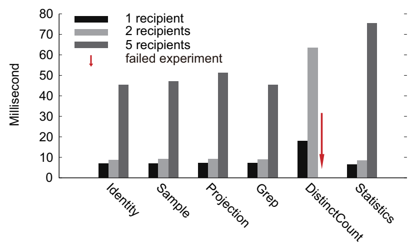


(b) Storm throughput with data scale 4X

Figure 4: Throughput of two platforms under different reception ability



(a) Spark latency with data scale 10X



(b) Storm latency with data scale 4X

Figure 5: Latency of two platforms under different reception ability

Table V: Storm Latency Under Different Reception Ability

Program	1 recipient(ms)	2 recipients(ms)	5 recipients(ms)
Wordcount	12019.25625	fail	fail

receiving data. As the throughput grows with reception ability, the suite is eliminating limitations from undersized input speed to further measure performance of the platforms. In addition, this suite leads to discover different patterns in terms of latency, which helps to profile the platforms more comprehensively.

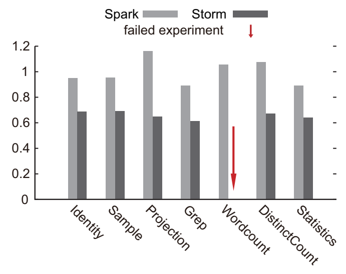
3) Fault tolerance workload suite: According to our fault tolerance workload suite, both TPF (Throughput Penalty Factor) and LPF (Latency Penalty Factor) defined in the metrics section are obtained and calculated for the two platforms.

Storm Wordcount indexes are not obtained since it meets saturate volume in this workload. As has been pointed out, TPF is usually less than 1 and the closer to 1 the better. From Fig.6(a), Spark's TPF varies between 0.9 and 1.17. Nearly half TPFs are over 1, which means that the throughput under failure increases on the contrary. It is because of slight performance variance of network or other components. We may draw the conclusion that failure doesn't impose any significant penalty to throughput in Spark. In contrast, Storm's throughput penalty is apparent as its average TPF is about 0.66, suggesting almost a drop of one third for throughput is witnessed.

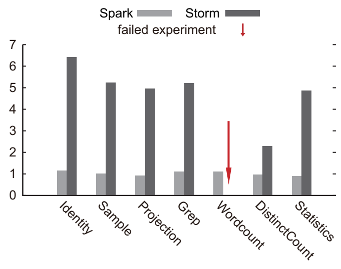
LPF, which is an index related to latency penalty, is usually larger than 1 and also the closer to 1 the better. It is evident that Spark's LPF averages to 1 from Fig.6(b), which indicates that just as the TPF shows, failure doesn't impact Spark platform much. As with Storm, it's a totally different story. For most benchmark programs whose latency under 2 recipients configuration is around 10ms, their latency has risen to around 50ms due to failure, causing average LPF of them to about 5.3. For DistinctCount, its LPF is 2.3, less than others but still is quite high.

In conclusion, our two indexes help to capture difference between the two platforms when faced with failure. For Storm this workload suite leads to penalty of failed records re-handle, lost worker reallocation and so on, as is reflected by TPF and LPF. For Spark, as most data are stored in memory of data recipients and operations are scheduled where data are located, the workload suite has little penalty on it.

4) Durability workload suite: In this suite, we simply focus on whether the frameworks keep responding. After running this suite, we find that for 10,000 records per minute, both platforms fulfill their mission and stay alive throughout, that is, their durability index is 100%. As with 1 million records per minute, Spark handles successfully while Storm keeps alive and responding but occasionally some records fail and gets reprocessed. The failure record rate is roughly

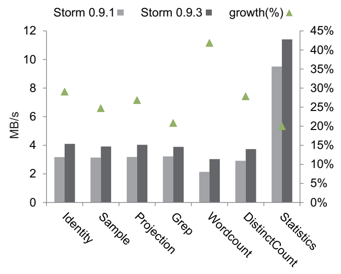


(a) TPF comparison

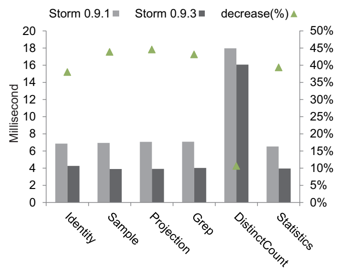


(b) LPF comparison

Figure 6: Performance penalty comparison under one node failure



(a) Throughput comparison



(b) Latency comparison

Figure 7: Performance comparison between different Storm versions

2.16%.

5) Comparisons against different versions: Storm's master version in github has evolved to 0.9.3-incubating[24] and efforts have been devoted to improving performance [25]. We apply performance workload suite with data scale 4X to evaluate this enhancement. From Fig.7(a) it's obvious that version 0.9.3 bears an increase of throughput. Average growth compared with the old version among different benchmark programs is 26 % while maximum growth may reach to over 40 %. As shown in Fig.7(b), an apparent decrease with latency is witnessed. Latency for Wordcount is far larger than others', thus we didn't include it in the graph. Its decrease of latency has reached to 66%. Average percentage of the decrease is about 40%, which is quite impressive. Our performance workload suite hence helps to demonstrate this improvement in performance.

V. Related Work

As mentioned previously, no specific benchmarks targeting at distributed stream computing frameworks have yet emerged. For standalone stream data management systems such as Aurora[8], Arasu et al. [14] presented a benchmark LinearRoad which simulates a scenario of toll system for motor vehicle expressways. Benchmark for big stream may also borrow methodology from emerging big data benchmarks. Hibench[17] is a benchmark suite for Hadoop covering 7 workloads ranging from micro benchmarks to machine learning algorithms. BigBench[18] is an end-to-end big data benchmark that covers various business cases and semi-structured as well as unstructured data in addition to borrowed data model from TPC-DS. BigDataBench[16] incorporates 6 real-world data sets and 19 workloads from scenarios such as social network and search engine.

Authors of modern stream computing systems carry out experiments on their own to evaluate their systems. We also summarize evaluation approaches adopted in assessments of these state-of-the-art stream computing systems. Yahoo S4[10] implements streaming click-through rate computation as their benchmark application and reports performance results. Apache Spark Streaming[13] leverages two programs, grep and wordcount, to assess not only performance but also scalability and fault tolerance of the framework. TimeStream[4] applies Distinct Count and Sentiment analysis of Tweets to evaluate its scalability and fault tolerance ability.

VI. Conclusion

In this paper, we presented StreamBench, a benchmark for modern distributed stream computing frameworks. We first summarized the challenges in this field and provided requirements for benchmarking these frameworks. We then defined StreamBench to meet these challenges and requirements. StreamBench proposes leveraging a message system as the stream data feed to stream computing frameworks. It also includes 7 benchmark programs covering both basic operations as well as common use cases and also four workload suites addressing not only performance but also fault recovery ability as well as durability of the frameworks. In addition, we applied StreamBench to Spark Streaming and Storm. We found that Spark tends to have larger throughput and less node failure impact compared with Storm while Storm has much less latency except with complex workloads under large data scale for which its latency may be multiple times of Spark's. Both two frameworks demonstrate durability under constant workload. We also helped to verify performance improvement of Storm's new version.

For future work, we suggest refining on data generation, program set and workload suite. As with data generation, more data emitting patterns can be applied based on research of real world stream input traces. To further expand program set, we hope to include windowed operations and analytical queries such as join of input stream and static data. Besides, new workload suites targeting at measuring scalability and load balancing are also desired. We are also planning to release it on github after some refinement.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[2] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[3] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[4] 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*. ACM, 2013, pp. 1–14.

[5] Z. Shao, "Real-time analytics at facebook," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2011, pp. http://www–conf.slac.stanford.edu.

[6] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time url spam filtering service," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 447–462.

[7] U. Cetintemel, "The aurora and medusa projects," *Data Engineering*, vol. 51, no. 3, 2003.

[8] 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," vol. 12, no. 2, pp. 120–139, 2003.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[11] "Storm: Distributed and fault-tolerant realtime computation." http://storm.incubator.apache.org/.

[12] "Samza." http://samza.incubator.apache.org/.

[13] 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 *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[14] 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*. VLDB Endowment, 2004, pp. 480–491.

[15] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2012.

[16] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, Z. Li, J. Zhan, Y. Qi, Y. He, S. Gong *et al.*, "Bigdatabench: a big data benchmark suite from web search engines," *arXiv preprint arXiv:1307.0320*, 2013.

[17] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[18] 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 international conference on Management of data*. ACM, 2013, pp. 1197–1208.

[19] "TPC-DS benchmark." http://www.tpc.org/tpcds/.

[20] "AOL Search Data." http://www.researchpipeline.com/mediawiki/index.php?title=AOL_Search_Query_Logs.

[21] CAIDA, "The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces," http://www.caida.org/data/passive/passive_trace_statistics.xml.

[22] "Kafka Wiki." https://cwiki.apache.org/confluence/display/KAFKA/Powered+By.

[23] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011.

[24] "apache/incubator-storm." https://github.com/apache/incubator-storm.

[25] S. Zhong, "Storm performance cannot be scaled up by adding more CPU cores." https://issues.apache.org/jira/secure/attachment/12641867/Storm_performance_fix.pdf.