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].

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

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

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.

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.

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

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.

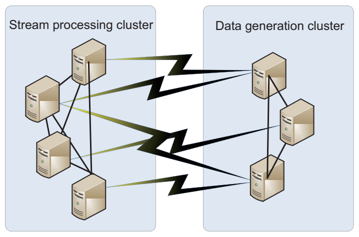
In addition, fault tolerance is one of the key concerns for distributed computing, especially for stream computing where data are not stored. Modern distributed stream computation systems handle fault tolerance with high concern[4], [13], [11], [12] and the recovery ability is worth inclusion. The fault 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.

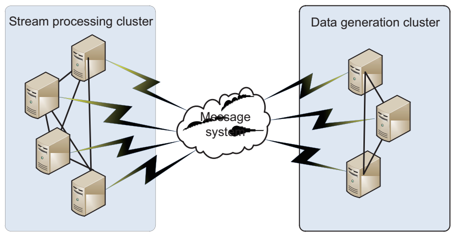
Multi-recipient performance workload suite: This suite contains seven benchmark programs too, but selects only one data scale. It defines a new descriptor of the stream computation cluster, **reception ability**, to be the proportion of nodes that receive input data out of the whole cluster, ignoring 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.

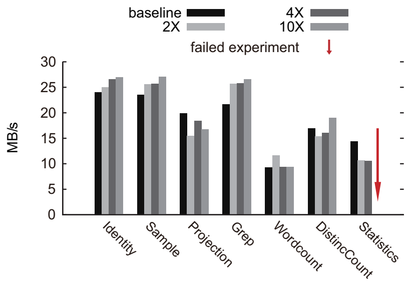
Fig.2 shows the throughput of the cluster in terms of MB/s of the two platforms while throughput in terms of records/s and per node is omitted. One thing to note is that the stream computation frameworks may encounter **saturate volume**, which is the data volume that will overwhelm the frameworks. If 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.

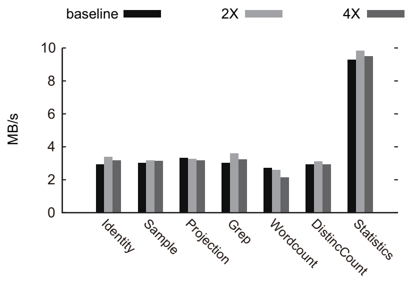
This suite helps to illustrate the throughput gap between the two platforms under current configurations. Different behaviors of Spark and Storm when running various programs further highlight the rationality to include diverse programs in the program set. Besides, according to the observations, record 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

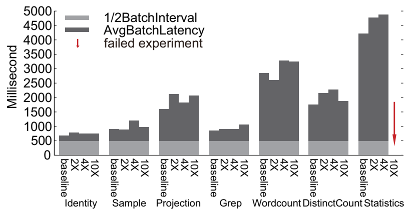


(a) Spark throughput

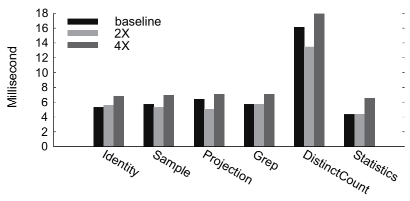


(b) Storm throughput

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.

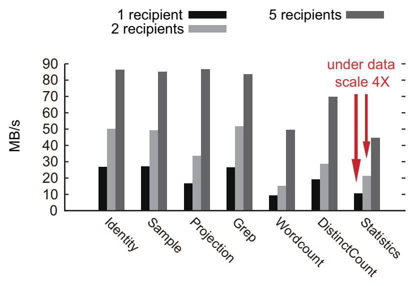
Latency metric of this workload suite reflects another trait of the stream computing platforms in terms of the life cycle of single records, which contributes to revealing differences between the two platforms from another perspective. Besides, data scales impose obvious impacts for Wordcount of Storm, 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.

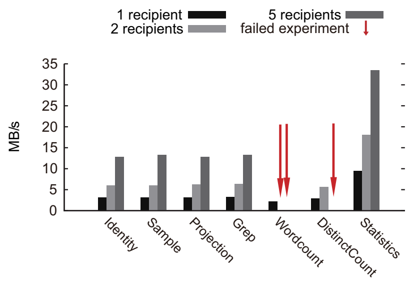
We list our observations from Fig.4 here. **[Observation 1]** Similar with performance workload suite, throughput with multiple recipients for Spark is 4-8 times that of Storm's except for Statistics, which is quite close. Besides, Spark's satire volume for more complex programs in this suite is greater 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

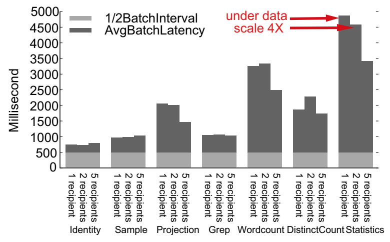


(a) Spark throughput with data scale 10X

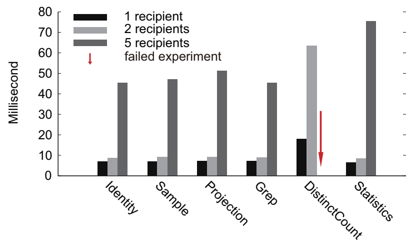


(b) Storm throughput with data scale 4X

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.

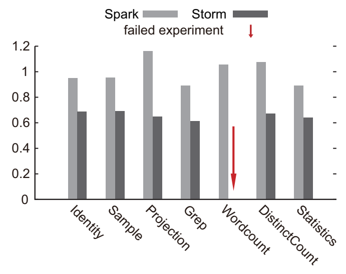
3) Fault tolerance workload suite: According to our fault tolerance workload suite, both TPF (Throughput Penalty Factor) and LPF (Latency Penalty Factor) defined in the metrics section are obtained and calculated for the two platforms.

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.

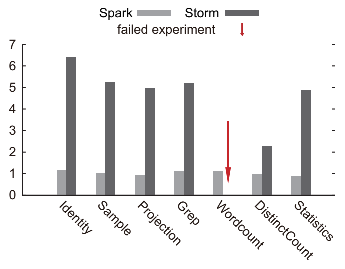
LPF, which is an index related to latency penalty, is usually larger than 1 and also the closer to 1 the better. It is evident that Spark's LPF averages to 1 from Fig.6(b), which indicates that just as the TPF shows, failure doesn't impact Spark platform much. As with Storm, it's a totally different 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

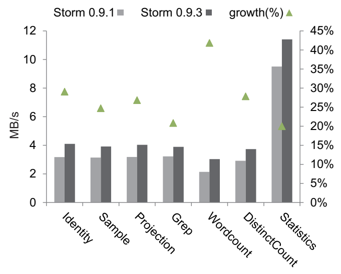


(a) TPF comparison

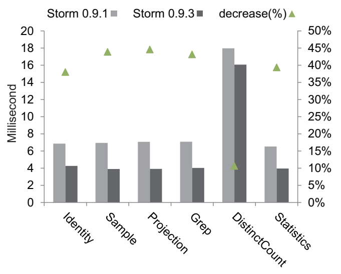


(b) LPF comparison

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.

A Performance Comparison of Open-Source Stream Processing Platforms

Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, Otto Carlos M. B. Duarte
Universidade Federal do Rio de Janeiro - GTA/COPPE/UFRJ - Rio de Janeiro, Brazil
Email: {martin, antonio, otto}@gta.ufrj.br

***Abstract*—Distributed stream processing platforms are a new class of real-time monitoring systems that analyze and extract knowledge from large continuous streams of data. These type of systems are crucial for providing high throughput and low latency required by Big Data or Internet of Things monitoring applications. This paper describes and analyzes three main open-source distributed stream-processing platforms: Storm, Flink, and Spark Streaming. We analyze the system architectures and we compare their main features. We carry out two experiments concerning threats detection on network traffic to evaluate the throughput efficiency and the resilience to node failures. Results show that the performance of native stream processing systems, Storm and Flink, is up to 15 times higher than the micro-batch processing system, Spark Streaming. However, Spark Streaming is robust to node failures and provides recovery without losses.**

I. Introduction

Sensor monitoring, network traffic analysis, cloud management [1], and security threats detection are applications that generate large amount of data to be processed in real time. These stream applications are characterized by an unbounded sequence of events or tuples that continuously arrive [2]. The advent of the Internet of things (IoT) increases the need of real-time monitoring. The estimate number of sensors networked by 2025 is around 80 billion [3]. Hence, data with this order of magnitude cannot always be processed centrally.

The main method to analyze big data in a distribute fashion is the MapReduce technique with Hadoop open-source implementation. Nevertheless, the platforms based on this technique are inappropriate to process real-time streaming applications. Applications processed by Hadoop correspond to queries or transactions performed in a stored and passive database and without real-time requirements, data elements are synchronized having an exact answer. Real-time monitoring applications require distributed stream processing that substantially differs from current conventional applications processed by distributed platforms. Monitoring normally requires the analysis of multiple external stream sources, generating alerts in abnormal condition. The real-time feature is intrinsic to stream processing applications and a big amount of alerts is normally expected. The stream data are unbounded and arrive asynchronously. Besides, the stream analysis requires historical data rather than only the latest reported and the data arrive. In cases of high entrance rates is common to filter the most important data discarding others and, therefore, approximate solutions are required, such as sampling methods. Hence, to meet these applications requirements, distributed processing models have been proposed and received attention from researchers.

Real-time distributed stream processing models can benefit traffic monitoring applications for cyber security threats detection [4]. Current intrusion detection and prevention systems are not effective, because 85% of threats take weeks to be detected and up to 123 hours for a reaction after detection to be performed [5]. New distributed real-time stream processing models for security critical applications is required and in the future with the advancement of the Internet of Things, their use will be imperative. To respond to these needs, Distributed Stream Processing Systems have been proposed to perform distributed processing with minimal latency and open-source general-purpose stream processing platforms are now available, meeting the need of processing data continuously. These open-source platforms are able to define custom stream processing applications for specific cases. These general-purpose platforms offer an Application Programming Interface (API), fault tolerance, and scalability for stream processing.

This paper describes and analyzes two native distributed real time native stream processing systems, the Apache Storm [6] and the Apache Flink [7], and one micro-batch system, the Apache Spark Streaming [8]. The architecture of each analyzed systems is discussed in depth and a conceptual comparison is presented showing the differences between these open-source platforms. Furthermore, we evaluate the data processing performance and the behavior of systems when a worker node fails. The experiments consider a real-time threat detection application developed by the authors [9]. The results are analyzed and compared with the conceptual model of each evaluated platform.

The remainder of this paper is organized as follow. In Section II we describe related work. We detail the Stream processing concept in Section III. Analyzed Platforms are presented in Section IV. Experimental results are shown in Section V. Finally, Section VI concludes the paper.

II. Related Work

Distributed real-time stream processing systems is a recent area and performance evaluations and comparisons between systems are fairly unexplored in the scientific literature. A few number of older and discontinued stream processing systems such as Aurora [10], and the project developed by Yahoo Apache S4 [11] served as base for most current systems. Google Millwheel [12], InfoSphere Streams IBM [13], among others are examples of proprietary solutions, but, in this paper, we focus on open-source systems.

Hesse and Lorenz compare the Apache Storm, Flink, Spark Streaming, and Samza platforms [14]. The comparison is restricted to description of the architecture and its main elements. Gradvohl *et. al* analyze and compare Millwheel, S4, Spark Streaming, and Storm systems, focusing on the fault tolerance aspect in processing systems [15]. Nevertheless, these two cited paper are restricted to conceptual discussions without experimental performance evaluation. Landset *et. al* perform a summary of the tools used for process big data [16], which shows the architecture of the stream processing systems. However, the major focus is in batch processing tools, which use the techniques of MapReduce. Roberto Colucci *et. al* show the practical feasibility and good performance of distributed stream processing systems for monitoring Signalling System number 7 (SS7) in a Global System for Mobile communications (GSM) machine-to-machine (M2M) application [17]. They analyze and compare the performance of two stream processing systems: the Storm and Quasit, a prototype of University of Bologna. The main result is to prove Storm practicability to process in real time a large amount of data from a mobile application.

Nabi *et. al* compare Apache Storm with IBM InfoSphere Streams platform in an e-mail message processing application [18]. The results show a better performance of InfoSphere compared to Apache Storm in relation to throughput and CPU utilization. However, IBM owns InfoSphere system and the source code is unavailable. Lu *et. al* propose a benchmark [19] creating a first step in the experimental comparison of stream processing platforms. They measure the latency and throughput of Apache Spark and Apache Storm. The paper does not provide results in relation to Apache Flink and the behavior of the systems under failure.

Dayarathna e Suzumura [20] compare the throughput, CPU and memory consumption, and network usage for the stream processing systems S, S4, and the Event Stream Processor Esper. These systems differ in their architecture. The S system follows the manager/workers model, S4 has a decentralized symmetric actor model, and finally Esper is software running on the top of Stream Processor. Although the analysis using benchmarks is interesting, almost all evaluated systems are already discontinued or not currently have significant popularity.

Unlike the above-mentioned papers, the following sections describe the architectural differences of open-source systems Apache Storm, Flink, and Spark Streaming. Moreover, we provide experimental performance results focusing on the throughput and parallelism in a threat detection application on a dataset created by the authors. They also evaluated the response and tolerance of the systems when one worker node fail. Finally, we conducted a critical overview of the main characteristics of each analyzed systems and discussed how the characteristics influence in the obtained results.

III. The Stream Processing

Data stream processing is modeled by a graph that contains data sources emitting continuously unbounded samples. The sources emit tuples or messages that are received by Processing Elements (PE). Each PE receives data on its input queues, performs computation using local state and produces an output to its output queue. The set of sources and processing nodes creates a logical network connected in a Directed Acyclic Graph (GAD). GAD is a graphical representation of a set of nodes and the processing tasks.

Stonebraker *et.al.* [2] highlight the most important requirements for distributed stream processing platforms. The data mobility identifies how it moves through the nodes. In addition, data mobility is a fundamental aspect to maintain low latency, since blocking operations, as done in batch processing platforms, such as Hadoop, decrease data mobility. Due to the large volume, data should be separated in partition to treat it in parallel. High availability and fail recovery are also critical in stream processing systems. In low latency applications, the recovery should be quick and efficient, providing processing guarantees. Thus, the stream processing platforms must provide resilience mechanisms against imperfections, such as delays, data loss, or out of order samples, which are common in data stream. Moreover, processing systems must have a highly optimized execution engine to provide real-time response for applications with high data rates. Thus, the ability to process millions of messages per second with low latency, within microsecond, is essential. To achieve a good processing performance, platforms shall minimize communication overhead between distributed processes in data transmission.

A. Methods of Data Processing

Data processing is divided in three main approaches: batch processing, micro-batch processing and stream processing. The analysis of large sets of static data, which are collected over previous time periods, is done with batch processing. However, this technique has large latency, with responses greater than 30 seconds, while several applications require real-time processing, with responses in microsecond order [21]. Despite, this technique can perform near real-time processing by doing micro-batch processing. Micro-batch treats the stream as a sequence of smaller data blocks. For small time intervals, the input is grouped into data blocks and delivered to the batch system to be processed. On the other hand, the third approach, stream processing, analyzes massive sequences of unlimited data that are continuously generated.

In contrast to batch processing, stream processing is not limited by any unnatural abstraction. Further, latency of stream processing is better than micro-batch, since messages are processed immediately after arrival. Stream processing performs better in real time, however, fault tolerance is more costly, considering that it must be performed for each processed messages. In batch and micro-batch processing, some functions, such as *join* and *split* are hard to implement, because the system handles an entire batch at time. However, unlike stream processing, fault tolerance and load balancing are much simpler, since the system sends the entire batch to a worker node, and if something goes wrong, the system can simply use another node.

B. Fault Tolerance

High availability is essential for real-time stream processing. The stream processing system should recover from a failure rapidly enough without affecting the overall performance. Hence, guaranteeing the data to be processed is a major concern in stream processing. On large distributed computing systems, various reasons lead to failure, such as node, network, and software failure. In batch processing systems, latency is acceptable and, consequently, the system does not need to recover quickly from a crash. However, in real-time systems without failures prevention, failures mean data loss.

Exactly once, *at least once*, and *at most once* are the three types of message delivery semantics. These semantics relate to the warranty that system gives to process or not a sample. When a failure occurs, the data can be forwarded to other processing element without losing information. The simplest semantic is *at most once* in which there is no error recovery, that is, either the samples are processed or lost. In *at least once* semantic, the error correction is made jointly for a group of samples, this way, if an error occurs with any of these samples, the entire group is repeated. This semantics is less costly than *exactly once*, which requires an acknowledgment for each sample processed.

IV. Analyzed Platforms

A. Apache Storm

Apache Storm [6] is a real-time stream processor, written in Java and Clojure. Stream data abstraction is called tuples, composed by the data and an identifier. In Storm, applications consists of topologies forming a directed acyclic graph composed of inputs nodes, called spouts, and processing nodes, called bolts. A topology works as a data graph. The nodes process the data as the data stream advance in the graph. A topology is analog to a MapRedude Job in Hadoop. The grouping type used defines the link between two nodes in the processing graph. The grouping type allow the designer to set how the data should flow in topology.

Storm has eight data grouping types that represent how data is sent to the next graph-processing node, and their parallel instances, which perform the same processing logic. The main grouping types are: *shuffle*, *field*, and *all grouping*. In *shuffle* grouping, the stream is randomly sent across the bolt instances. In *field* grouping, each bolt instance is responsible for all samples with the same key specified in the tuple. Finally, in *all grouping*, samples are sent to all parallel instances.

Figure 1 shows the coordination processes in a Storm cluster. The manager node, Nimbus, receives a user-defined topology. In addition, Nimbus coordinates each process considering the topology specification, i.e., coordinates spouts and bolts instantiation and their parallel instances. Each Worker node runs on a Java Virtual Machine and execute one or more tasks, also called processes. The Supervisors monitor the processes of each topology and inform the state to Nimbus using the Heartbeat protocol. *Zookeeper* is used as a coordinator between Nimbus and Supervisors. Nimbus and Supervisor are stateless, granting to *Zookeeper* all state management.

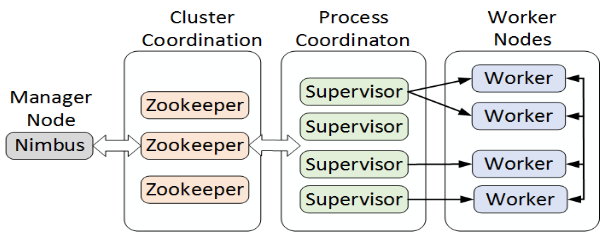


Figure 1: Nimbus receives topologies and communicates to Supervisors which coordinate process in workers, all the coordination between Nimbus and Supervisor is made by *Zookeeper* who store the cluster state.

Apache Storm uses an upstream backup and acknowledgments mechanism to ensure that tuples are re-processed after failure. For each processed tuple, an acknowledgment (ACK) is sent to the acker bolt. Every time a tuple enters the topology, the spout add an *id*, then send the *id* to the acker bolt. The acker bolt saves all *ids* and when a bolt processes the tuple, the bolt sends an ACK to the acker bolt. When tuples exit the topology, the acker bolt drops the *ids*. If a fault occurs, not all acknowledgments have been received or the ACK timeout expired. This guarantee that each tuple is either processed once, or re-processed in the case of a failure, known as delivery *at-least-once*.

B. Apache Flink

Apache Flink [7] is a hybrid processing platform, supporting both stream and batch processing. Flink core is the stream processing, making batch processing a special class of application. Analytics jobs in Flink compile into a directed graph of tasks. Apache Flink is written in Java and Scala. Figure 2 shows Flink architecture. Similar to Storm, Flink uses a master-worker model. The job manager interfaces with clients applications, with responsibilities similar to Storm master node. The job manager receives client applications, organizes the tasks and sends them to workers. In addition, the job manager maintains the state of all executions and the status of each worker. The workers states are informed through the mechanism Heartbeat, like in Storm. Task manager has a similar function as a worker in Storm. Task Managers perform tasks assigned by the job manager and exchange information with other workers when needed. Each task manager provides a number of processing slots to the cluster that are used to perform tasks in parallel.

Stream abstraction is called DataStream, which are sequences of partially ordered records. DataStreams are similar to Storm tuples. The DataStreams receive stream data from external sources such as message queues, sockets, and others. The DataStreams support multiple operators or functions, such as *map*, *filtering* and *reduction*, which are applied incremen-

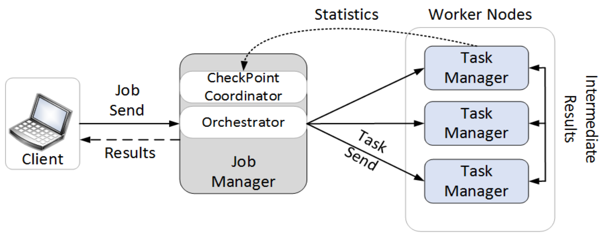


Figure 2: Architecture of Flink system. The Job manager receives jobs from clients, divides the jobs into tasks, and sends the tasks to the workers. Workers communicate statistics and results.

tally to each sample generating a new DataStreams. Each operator or function can be parallelized, running instances on different partitions of the respective stream. This method allows distributed execution on the streams.

Flink fault tolerance approach is based on snapshot over distributed checkpoints that maintain the status of jobs. Snapshots act as consistency checkpoints to which the system can return in case of failure. Barriers are injected in source elements and flow through the graph together with the samples. A barrier indicates the beginning of a checkpoint and separates records into two groups: records that belong to current snapshot and others that belongs to the next snapshot. Barriers trigger new snapshots of the state when they pass through operators. When an operator receives a barrier, it stores the status of the corresponding stream snapshot and sends the checkpoint coordinator to the job manager. In case of software, node, or network failure, Flink stops the DataStreams. The system immediately restarts operators and resets to the last successful checkpoint stored. All records processed in the restart of a stream are guaranteed not to have been part of the previous checked state, ensuring delivery of *exactly-once*.

C. Apache Spark Streaming

Spark is a project initiated by UC Berkeley and is a platform for distributed data processing, written in Java and Scala. Spark has different libraries running on the top of the Spark Engine, including Spark Streaming [8] for stream processing. The stream abstraction is called Discrete Stream (D-Stream) defined as a set of short, stateless, deterministic tasks. In Spark, streaming computation is treated as a series of deterministic batch computations on small time intervals. Similar to MapReduce, a job in Spark is defined as a parallel computation that consists of multiple tasks, and a task is a unit of work that is sent to the Task Manager. When a stream enters Spark, it divides data into micro-batches, which are the input data of the Distributed Resilient Dataset (RDD), the main class in Spark Engine, stored in memory. Then the Spark Engine executes by generating jobs to process the micro-batches.

Figure 3 shows the layout of a Spark cluster. Applications or jobs within the Spark run as independent processes in the cluster which is coordinated by the master or Driver Program, responsible for scheduling tasks and creating the `Spark Context`. The `Spark Context` connects to various types of cluster managers, such as the Spark StandAlone,

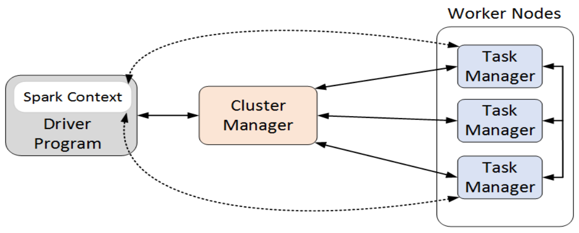


Figure 3: Cluster architecture of Spark Streaming system.

Mesos or Hadoop YARN (*Yet Another Resource Negotiator*). These cluster managers are responsible for resource allocation between applications. Once connected, Spark executes task within the task managers, which perform processing and data storage, equivalent to Storm workers, and results are then communicated to the `Spark Context`. The mechanism described in Storm, in which each worker process runs within a topology, can be applied to Spark, where applications or jobs are equivalent to topologies. A disadvantage of this concept in Spark is the messages exchange between different programs, which is only done indirectly such as writing data to a file, worsen the latency that could be around seconds in applications of several operations.

The Spark has Streaming delivery semantics *exactly-once*. The idea is to process a task on various worker nodes. During a fault, the micro-batch processing may simply be recalculated and redistributed. The state of RDDs are periodically replicated to other worker nodes, in case of node failure. Tasks are then discretized into smaller tasks that run on any node without affecting execution. Thus, the failed tasks can be launched in parallel evenly distributing the task without affecting performance. This procedure is called Parallel Recovery. The semantics of *exactly-once* reduces the overhead shown in upstream backup in which all tuples are acknowledge like in Storm. However, micro-batch processing has disadvantages. Micro-batch processing takes longer in downstream operations. The configuration of each micro-batch may take longer than conventional native stream analysis. Consequently, micro-batches are stored in the processing queue.

Table I presents a summary of features underlined in the comparison of the stream processing systems. The programming model can be classified as compositional and declarative. The compositional approach provides basic building blocks, such as Spouts and Bolts on Storm and must be connected together in order to create a topology. On the other hand, operators in the declarative model are defined as higher-order functions, that allow writing functional code with abstract types and the system will automatically create the topology.

V. Results

Fault tolerance and latency requirements are essential in real-time stream processing. This section evaluates the processing throughput and the behavior during node failure of the three presented systems: Apache Storm version 0.9.4, Apache Flink version 0.10.2 and Apache Spark Streaming version 1.6.1, with micro-batch size established in 0.5 seconds.

Table I: Overview of the comparison between Stream Processing Systems.

	Storm	Flink	Spark Streaming
Stream Abstraction	Tuple	DataStream	DStream
Build Language	Java/Closure	Java/Scala	Java/Scala
Messages Semantic	At least once	Exactly one	Exactly one
Failure Mechanism	Upstream Backup	Check-point	Parallel Recovery
API	Compositional	Declarative	Declarative
Failures Subsistem	Nimbus, Zookeeper	No	No

We perform the experiments in an environment with eight virtual machines running on a server with Intel Xeon E5-2650 processor at 2.00 GHz and 64 GB of RAM. The experiment topology configuration is one master and seven worker nodes for the three evaluated systems. We calculate the results with confidence interval of 95%. To enter data at high rates in the stream processing systems, we use a message broker that operates as a publish/subscribe service, Apache Kafka in version 0.8.2.1. In Kafka, samples or events are called messages, name that we will use from now on. Kafka abstracts message stream into topics that act as buffers or queues, adjusting different production and consumption rates.

The dataset used in the experiments is the one from an threat detection application created by the authors [9]. The dataset is replicated to assess the maximum processing throughput at which the system can process. The application tested was a threat detection system with a neural network classifier programmed in Java.

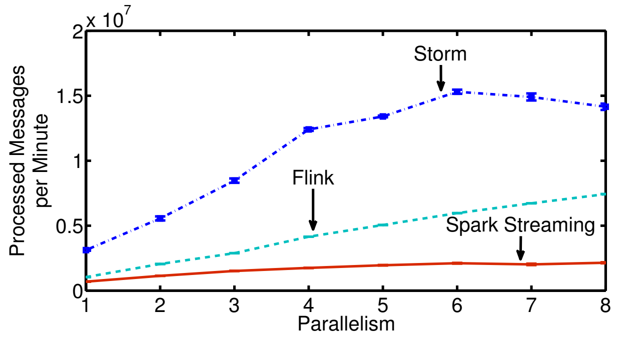
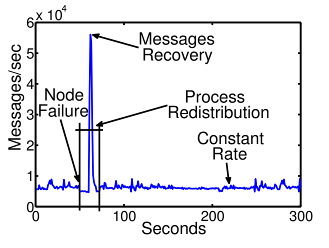
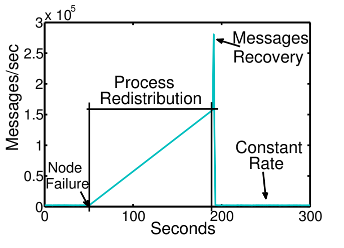


Figure 4: Throughput results of the platforms in terms of number of messages processed per minute in function of the task parallelism.

The first experiment measures the performance of the platforms in terms of processing throughput. The data set is injected into the system in its totality and replicated as many times as necessary. The experiment calculates the consumption and processing rate of each platform. It also varies the parallelism parameter, which represents the total number of cores available for the cluster to process samples in parallel. Figure 4 shows the results of the experiment. Apache Storm has a better throughput than the others do. For a single core, i.e., without parallelism, Storm already shows better performance compared to Flink and Spark Streaming in at least 50%. Flink has a completely linear growth, but with values always below the Apache Storm. The processing rate of the Apache Spark Streaming, when compared to the Storm and Flink, is much lower, this is due to the use of micro-batch, as each batch is grouped before processing. Apache Storm behavior is linear until the parallelism of four cores. Then, the processing rate grows until parallelism of six, where the system saturates. This behavior was also observed in Apache Spark Streaming with the same six-core parallelism.

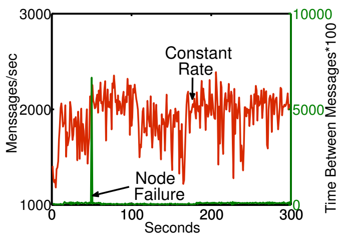


(a) Storm behavior under node failure.

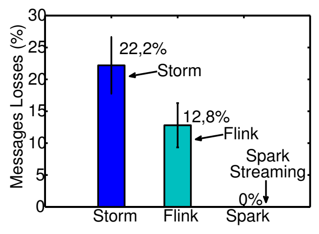


(b) Flink behavior under node failure.

Figure 5: A failure is produced at 50 seconds. a) Storm and b) Flink system behavior after detecting the failure and consisting of process redistribution and message recovery procedures.



(a) Spark behavior under node failure.



(b) Messages losses during node failure

Figure 6: a) The Spark system behavior under failure, indicating that it keeps stable and does not lose messages. b) Percentage of message losses.

The second experiment shows the system behavior when a node fails. Messages are sent at a constant rate to analyze the system behavior during the crash. The node failure is simulated by turning off a virtual machine. Figures 5a, 5b and 6a show the behavior of the three systems before and after a worker node failure at 50 seconds. Apache Storm takes some time in the redistribution processes after the fault was detected. This time is due to communication with the Zookeeper. Zookeeper has an overview of the cluster and reports the state for Nimbus in Storm, which reallocates the processes on other nodes. Soon after this redistribution, the system retrieves Kafka messages at approximately 75 seconds. Although the system can quickly recover from node failure, during the process there is a significant message loss. A similar behavior is observed in Apache Flink. After detecting the failure at approximately 50 seconds, the system redistributes the processes for active nodes. Flink does this process internally without the help of any subsystem, unlike Apache Storm that uses Zookeeper.

Figure 5b shows that time period in which Flink redistributes processes is much greater than the time spent in Apache Storm. However, message recovery is also higher, losing some messages during the process redistribution. Figure 6a shows Spark streaming behavior during a failure. When a failure occurs at approximately 50 seconds, the system behavior is basically the same as before. This is due to the use of tasks with micro-batch that are quickly distributed without affecting performance. Spark Streaming shows no message loss during fail. Thus, despite the low performance of Spark Streaming, it could be a good choice in applications where resilience and processing all messages are necessary.

Figure 6b shows the comparison of lost messages between Storm, Flink and Spark. It shows that Spark had no loss during the fault. The measure shows the percentage of lost messages by systems, calculated by the difference of messages sent by Apache Kafka and messages analyzed by the systems. Thus, Apache Flink has a smaller loss of messages during a fault with about a 12.8% compared to 22.2% in Storm. We obtain the result with 95% confidence interval.

VI. Conclusion

This paper describes and compares the three major open source distributed stream processing systems: Apache Storm, Apache Flink, and Apache Spark Streaming.The systems are similar in some characteristics, such as the fact that tasks run inside a Java Virtual Machine and the systems use master-worker model. A performance analysis of stream systems in a threat detection experiment by analyzing network traffic was carried out. The results show that Storm has the highest processing rate when the parallelism parameter, in number of processing cores, is changed, getting shorter response time, up to 15 times lower.

We also performed another experiment to show the behavior of the systems the during node failure. In this case, we show that Spark streaming, using micro-batch processing model, can recover the failure without losing any messages. Spark Streaming stores the full processing state of the micro-batches and distributes the interrupted processing homogeneously among other worker nodes. On the other hand, the stream processing native systems, Storm and Flink, lose messages despite using more complex recovery mechanisms. Apache Flink, using a checkpoint algorithm, has a lower message loss rate, about a 12.8% during the redistribution process after a failure. Storm loses 10% more, about 22.2% of messages since it uses a subsystem, *Zookeeper*, for nodes synchronization. Therefore, in order to select a platform, the user should take into account the application requirements and balance the compromise between high processing rates and fault tolerance.

Acknowledgment

This work was supported by CNPq, CAPES, and FAPERJ.

References

[1] B. Dab, I. Fajjari, N. Aitsaadi, and G. Pujolle, “VNR-GA: Elastic virtual network reconfiguration algorithm based on genetic metaheuristic,” in *IEEE GLOBECOM*, Dec 2013, pp. 2300–2306.

[2] 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.

[3] P. Clay, “A modern threat response framework,” *Network Security*, vol. 2015, no. 4, pp. 5–10, 2015.

[4] M. Andreoni Lopez, D. M. F. Mattos, and O. C. M. B. Duarte, “An elastic intrusion detection system for software networks,” *Annals of Telecommunications*, pp. 1–11, 2016.

[5] I. Ponemon and IBM, “2015 cost of data breach study: Global analysis,” www.ibm.com/security/data-breach/, may 2015, accessed: 16/04/2016.

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

[7] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, “Lightweight asynchronous snapshots for distributed dataflows,” *Computing Research Repository (CoRR)*, vol. abs/1506.08603, 2015.

[8] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *XXIV ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[9] A. Lobato, M. A. Lopez, and O. C. M. B. Duarte, “An accurate threat detection system through real-time stream processing,” Grupo de Teleinformática e Automação (GTA), Univeridade Federal do Rio de Janeiro (UFRJ), Tech. Rep. GTA-16-08, 2016.

[10] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, “Monitoring streams: A new class of data management applications,” in *28th International Conference on Very Large Data Bases*, 2002, pp. 215–226.

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

[12] 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,” *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.

[13] C. Ballard, O. Brandt, B. Devaraju, D. Farrell, K. Foster, C. Howard, P. Nicholls, A. Pasricha, R. Rea, N. Schulz *et al.*, “IBM infosphere streams,” *Accelerating Deployments with Analytic Accelerators, ser. Redbook. IBM*, 2014.

[14] G. Hesse and M. Lorenz, “Conceptual survey on data stream processing systems,” in *IEEE 21st International Conference on Parallel and Distributed Systems*, 2015, pp. 797–802.

[15] A. L. S. Gradvohl, H. Senger, L. Arantes, and P. Sens, “Comparing distributed online stream processing systems considering fault tolerance issues,” *Journal of Emerging Technologies in Web Intelligence*, vol. 6, no. 2, pp. 174–179, 2014.

[16] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, “A survey of open source tools for machine learning with big data in the hadoop ecosystem,” *Journal of Big Data*, vol. 2, no. 1, pp. 1–36, 2015.

[17] R. Coluccio, G. Ghidini, A. Reale, D. Levine, P. Bellavista, S. P. Emmons, and J. O. Smith, “Online stream processing of machine-to-machine communications traffic: A platform comparison,” in *IEEE Symposium on Computers and Communication (ISCC)*, June 2014, pp. 1–7.

[18] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, “Of streams and storms,” *IBM White Paper*, 2014.

[19] R. Lu, G. Wu, B. Xie, and J. Hu, “Stream bench: Towards benchmarking modern distributed stream computing frameworks,” in *IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 69–78.

[20] M. Dayarathna and T. Suzumura, “A performance analysis of system S, S4, and Esper via two level benchmarking,” in *Quantitative Evaluation of Systems*. Springer, 2013, pp. 225–240.

[21] M. Rychly, P. Koda, and P. Smrz, “Scheduling decisions in stream processing on heterogeneous clusters,” in *Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, Jul. 2014, pp. 614–619.

Benchmarking Modern Distributed Streaming Platforms

Shilei Qian[1], Gang Wu[1], Jie Huang[2], Tathagata Das[3],
choice127@sjtu.edu.cn, wugang@cs.sjtu.edu.cn, jie.huang@intel.com, tdas@databricks.com
Shanghai Jiao Tong University[1], Intel APAC R&D Center[2], Databricks Inc[3]

***Abstract*—The prevalence of big data technology has generated increasing demands in large-scale streaming data processing. However, for certain tasks it is still challenging to appropriately select a platform due to the diversity of choices and the complexity of configurations. This paper focuses on benchmarking some principal streaming platforms. We achieve our goals on StreamBench, a streaming benchmark tool based on which we introduce proper modifications and extensions. We then accomplish performance comparisons among different big data platforms, including Apache Spark, Apache Storm and Apache Samza. In terms of performance criteria, we consider both computational capability and fault-tolerance ability. Finally, we give a summary on some key knobs for performance tuning as well as on hardware utilization.**

***Keywords*-distributed streaming computing; benchmark; big data; spark streaming; storm**

I. Introduction

Data stream computation has a wide range of scenarios that have been witnessed for a long time. Examples of such applications include financial applications, network monitoring, sensor networks and web log mining [1], [2], [3]. Nowadays we are experiencing a more rapidly expanding data scale because of the growing coverage of sensors, the prevalence of the mobile Internet and the soaring popularity of social media such as Twitter [4] and Facebook [5].

Although distributed stream processing systems have been under research by academia for over a decade (*e.g.* Medusa projects [6]), these efforts are merely distributed extensions to the single-node system Aurora [7]. The introduction of MapReduce [8] has revolutionized modern distributed computing systems. Owing to its scalability and fault-tolerance, MapReduce has become the basis for various data stream processing frameworks. Examples are S4 from Yahoo [9], Apache Storm [10], Apache Samza [11], Apache Spark Streaming [12] and TimeStream from Microsoft. As these systems keep sprouting and maturing, the necessity to evaluate and compare them rises. However, except StreamBench [13], which just carries out some evaluations to verify the correctness of its benchmark definition, no more evident efforts have been devoted to benchmark and compare these modern distributed stream processing frameworks.

Hence, this paper aims to take a step towards benchmarking several primary streaming computing platforms. In the first place we analyze the motivation for our benchmark work, followed by the description of some preliminaries. It is so hard to choose a proper platform and make best use of it because of the diversity of choices and the complexity of configurations for each platform. A comprehensive reference on how to choose a streaming platform is still unavailable. In this work, we accomplish this goal with the benchmark tool - StreamBench [13]. Considering the popularity and novelty of streaming platforms, we choose these three as our target: Apache Spark Streaming(both direct and receiver approach), Apache Storm(including Trident) and Apache Samza.

This work is supported by National Natural Science Foundation of China(NSFC 61472241).

In the second step, we specify the benchmark analysis techniques. We introduce some modifications and extensions to accommodate StreamBench to our benchmark goals, and we also associate it with fair test configurations and reasonable parameters. For each platform, we compare the capability as well as the fault-tolerance ability, identify and tune some key knobs, and summarize the characteristics of hardware utilization.

Finally, we carry out experiments on each platform and provide further analysis. We notice that Spark direct approach and Storm Trident can saturate the network resource and guarantee larger throughput. Spark receiver approach also has higher throughput than Storm, while Storm has much shorter latency. Spark is quite fault tolerant and stable with the increase of data scale and node failure. We also find that some key knobs have a great impact on the performance of these platforms. In detail, batch size, write-ahead-logs mechanism and kryo serialization are keys to the efficiency of Spark, while Storm depends more on its ACK mechanism and the the size of pending list.

We hope this paper can provide a useful reference for users on how to properly choose a streaming platform and allocate resources for the deployment of streaming programs.

In summary, our key contributions are as follows:

1) A general description of several primary streaming platforms on capability and fault-tolerance ability.
2) Completely benchmark and analyze Apache Spark(both direct and receiver approach), Apache Storm(including Trident), and tune some key knobs which have a great influence on performance.
3) Conclude the characteristics of hardware utilization for these platforms.

The rest of this paper are organized as follows: Section II describes the motivation for our work, Section III displays some preliminaries. Analysis techniques are discussed in Section IV, followed by the experiment and analysis in Section V. And we conclude this paper in Section VI.

II. Motivation

A user may always face some problems when choosing a streaming platform. We summarize them in this section, which form the motivation of our work.

The main problem is just how to select a proper streaming platform. There are many distributed streaming computing frameworks nowadays. However, reference on how to choose a streaming platform is still unavailable. StreamBench carries out some evaluations to verify the correctness of its benchmark definition, while omitting completely benchmarking and tuning work. Except that, no more evident efforts have been devoted to benchmark on these modern distributed stream processing platforms.

It is also challenging for users to know whether the platform is reliable and fault-tolerant. Apart from the capability issue, reliability and fault-tolerance ability are major criteria on selecting platforms to process real streaming data.

The lack of knowledge on factors impacting the performance of streaming applications can be a serious problem. The complexity of the configurations for each platform makes it time-consuming to reach the best performance. For a specific streaming platform, among these large numbers of parameters, there are some key parameters influencing the performance more significantly.

The last problem is the decision on the amount of hardware resources for streaming applications, including CPU, memory, network and disk. The allocations of these resources generally vary according to platforms and application requirements. How to characterize the hardware utilization for the streaming platforms is another important motivation of our work.

III. Preliminary

Before the introduction of our benchmark workload suites and the experiments, we need to present some preliminaries in this section.

A. Target Platform

Considering the popularity and novelty of streaming platforms, we choose these three as our target: Apache Spark Streaming, Apache Storm and Apache Samza. In our benchmark work, we take Spark Streaming and Storm as our primary target, and tune some key parameters for them. Particularly, for Samza, the benchmark on it is limited to the capacity issue due to the immaturity of this platform.

1) Apache Spark Streaming: Apache Spark Streaming is an extension of the core Spark API, supports the processing of live data streams. Internally, it receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches [14].

2) Apache Storm: Apache Storm is a distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, do for realtime processing. Trident is a high-level abstraction for doing realtime computing on top of Storm. It allows you to seamlessly intermix high throughput, stateful stream processing with low latency distributed querying [15].

3) Apache Samza: Apache Samza is a distributed stream processing framework. It uses Kafka for messaging and Apache Hadoop Yarn to provide fault tolerance, processor isolation, security, and resource management. The architecture of Samza makes it simple and pluggable [11].

B. Benchmark Tool

Regarding the convenience of our benchmark work, we modify and extend the definition of StreamBench to meet our needs. Some details about benchmark definition are discussed here. We try to make our test configurations and parameters fair and reasonable, even though the benchmark suite itself is not mature enough.

1) Benchmark Definition: StreamBench suggests using a message system as a mediator for data generation and consumption. In our experiments, we use Apache Kafka [16] as the message system.

Stream data are sent to target systems on the fly. To approach real-world cases, we choose to generate streams from seed data sets gathered from two main stream computing scenarios, respectively real time web log processing [17] and network traffic monitoring [18]. The data sets cover both text and numeric data.

2) Workloads: 7 benchmark programs are used in our experiments, summarized in Table I. 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], [19]. Projection benchmark extracts a certain field of the input. Grep checks if the input contains certain strings. It is a simple yet common operation also adopted in the evaluation of Spark Stream [12]. The 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 is widely accepted as a standard micro-benchmark for big data [20]. 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.

Table I: Workload Categorization Mapping

Benchmark	Complexity	Input data seed
Identity	Low - Staless	AOL [17]
Sample	Low - Staless	AOL [17]
Projection	Low - Staless	AOL [17]
Grep	Low - Staless	AOL [17]
Wordcount	Medium - Stateful	AOL [17]
DistinctCount	Medium - Stateful	AOL [17]
Statistics	Medium - Stateful	CAIDA [18]

Statistics benchmark calculates the maximum, minimum and sum of a field from input. Except Statistics using the numeric data with the record size of 200B, the other six workloads deal with the text source with the size of 60B for each record.

3) Modifications on StreamBench: StreamBench is defined for general platforms and supports several kinds of workloads. We need to choose a proper configuration, and introduce some modifications and extensions to accommodate it to our benchmark goals.

We choose two clusters, one for Kafka and one for streaming system. There are two kinds of deployments for Kafka integration, one is the Kafka and streaming system both deployed on same nodes, the other is what we apply, just forming two clusters. If we choose the previous one, the streaming system cannot fully occupy the resources of the node. The metrics of its behavior will also be confusing due to the influence of Kafka.

For the data preparation method, we treat offline data preparation as a better way. We need to ensure that there is no overhead from disks of Kafka. In our experiment, the scale of data does not exceed the capacity of disk cache, and full network utilization can be reached. Offline data preparation means that the generation of streaming data is done at first, and then fetch and process steps are taken. This can keep the impact of message system at a low level, and facilitate our benchmark process and validation.

The validation on the results of workloads for different platforms is necessary. Just as previous statement, the offline data preparation makes the validation work feasible. If we obtain the same result for workloads running on different platforms, we can guarantee the semantics of the workload implementations on different platforms are identical, and our work is reasonable.

Since the nodes we use in our experiments are much more powerful than those in StreamBench, we increase the data scale rather than the original definition. The partition number for the input streams is customized as well. Flow control for some stream systems is added to make them accommodate to the speed of the stream processing.

IV. Analysis Technique

In our benchmark work, we need to set some work goals and define some workload suites, as well as some metrics for them. The details of analysis techniques are displayed in this section.

A. Workload Suite

1) Capability Workload Suite: The capability workload suite is designed to test throughput, latency and capacity. It incorporates all of the seven benchmark programs, demands all input data be pushed to the message system where stream computing systems obtain data as fast as they can.

2) Fault-tolerance Workload Suite: This suite uses Identity workload and considers only one node failure. It requires intentionally failing one node in about the middle of the executions to capture penalty brought by the loss of the computing resource. For feasibility, we take a straightforward approach by simply collecting the performance metrics through the whole process and comparing it with failure free executions. 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.

B. Key Knobs for Certain Platforms

The complexity of the configurations for each platform always takes users a lot of time getting the best performance. For a specific streaming platform, among these large numbers of parameters, there are some key parameters influencing the performance much more significantly but users are not aware of. We need to find these key knobs and do some tuning experiments.

C. Characteristic of Hardware

The utilization of hardware resources (*e.g.* CPU, memory, network, disk) varies significantly with different workloads and platforms. We aim to conclude a general description on the characteristics of hardware utilization for each streaming platform, based on the monitoring data of hardware during the experiments.

D. 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 platforms. Throughput metric is the data size in terms of bytes processed per second. The second metric is latency, which is the average time span from the arrival of a record till the end of processing this record.

New metrics are raised for the fault tolerance workload suite. Suppose originally N nodes are serving, as described in the workload suite definition, one node is deliberately failed approximately in the middle of execution, remaining N-1 nodes 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

Table II: Hardware Configuration Summary

Hardware	Configuration
CPU	Xeon E5-2699@2.30GHz 72 cores
Memory	128 G
Disk	12 SATA HDDs 1T
NIC	10 Gb

Table III: Software Version Summary

Software	Version
OS/Kernel	Ubuntu 14.04.2 LTS 3.16.0-30-generic x86_64
JDK	Oracle jdk1.8.0_25
Hadoop	Hadoop-2.3.0-cdh5.1.3
Storm	0.9.3
Spark	1.4.0
Samza	0.8.0
Kafka	Kafka_2.10-0.8.1

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.

To achieve the goal of characteristics, we need to monitor the usage of hardware resource during the computation.

V. Experiment and Analysis

This section briefly dips into the configurations of the cluster followed by the result and analysis of these platforms in terms of goals in previous sections. For Apache Spark Streaming, both receiver-based approach and direct approach introduced since Spark 1.3 are evaluated. And for Apache Storm, result of Trident is also carried out.

A. Experiment Hardware

6 identical nodes are used forming two clusters: 3-node Kafka cluster, running as input source and 3-node streaming system, doing stream computation. The hardware configurations of these nodes are given in Table II. The maximum ethernet link speed of these nodes can reach 10000Mb/s. Hyper threading is enabled for all tests, while hugepage is always disabled. One extra node is needed to deploy master and Apache Zookeeper providing services for Apache Kafka.

B. Cluster Configuration

All software versions can be found in Table III. For Spark, we set executor memory size to 100G, with 20 seconds batch duration. For Storm, 4 workers are set up in each node, each with 32G max memory allocation. And for Samza, it needs the Apache Hadoop YARN environment, where we use Hadoop-2.3.0-cdh5.1.3. We don't make any in-depth tuning and just make adequate configurational adjustments according to system hardware resources, trying to make full utilization of them.

C. Workload Suite Experiment

This section presents the result for the workload suites defined previously on these platforms.

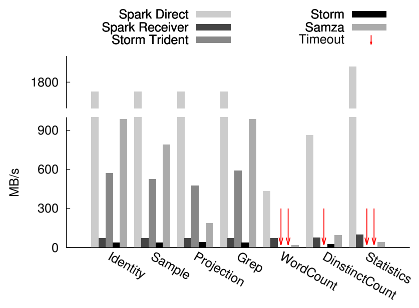
1) Capability Workload Suite: The capability workload suite is designed to test the throughput, latency and capacity.

Figure 1(a) shows the throughput for platforms. The batch interval here we choose for Spark is 20 seconds, and 500MB for the Storm Trident batch size. Input data scale is defined, namely 1 billion records for record size of 60B and 0.5 billion for 200B.

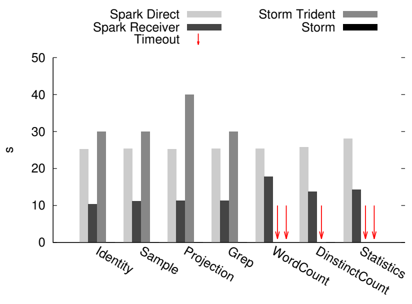
Here come our observations. Observation 1 to 4 are based on workload Identity. **[Observation 1]** To the Spark platform, the throughput for direct mode is the best. It can reach the network upper bound. **[Observation 2]** Storm Trident can also saturate the network and gain a good throughput performance due to the batch work. **[Observation 3]** The throughput of Storm is the worst in these platforms with ACK mechanism on. **[Observation 4]** For Samza, its throughput is much better than Spark receiver approach.

Following observations are about the throughput performance on different workloads for a certain platform. **[Observation 5]** The throughput of Spark direct mode varies with benchmark programs, and the maximum (Identity) throughput is close to 4 times that of the minimum (WordCount). However, the throughput for Spark receiver mode is quite similar among different workloads. The reason is that WordCount in direct mode needs more CPU resource, and cannot saturate the network, while the network utilization for receiver mode is quite similar for different workloads. **[Observation 6]** The throughput for Storm Trident also varies a lot, because different implementations of workloads ask for different allocation of resources. **[Observation 7]** For Storm, its throughput metric is almost constant to various benchmark programs.

In Figure 1(b), the latency for platforms is displayed. The latency for Storm is very short, around 100 milliseconds, so that it can hardly seen in the figure. And the latency for Samza haven't been collected. Two observations come out regarding latency. **[Observation 1]** The latency for Spark direct approach is about twice that for receiver approach, except WordCount. **[Observation 2]** The weakness of Storm Trident is its long latency. However, the latency for Storm is the shortest among these platforms. If we decrease the batch size of Spark and Storm Trident, we can gain better latency, but still no better than Storm.



(a) Throughput for platforms



(b) Latency for platforms

Figure 1: Performance overview for platforms

Table IV: Storm result of WordCount in small scale

Scale (million)	Storm Throughput(MB/s)	Storm Latency(s)	Storm Trident Throughput(MB/s)	Storm Trident Latency(s)
5	10.3	0.31	103.0	3
50	9.7	0.29	154.5	20
500	Timeout	Timeout	Timeout	Timeout

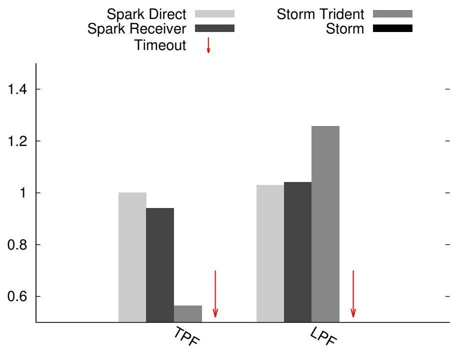


Figure 2: Fault tolerance evaluation for platforms

For the capacity, Spark and Samza can handle larger amount of data than Storm. Take the WordCount workload into acount, it is so complex that Storm as well as Storm Trident cannot handle this workload at 1 billion data scale. This is mainly because high CPU occupation increases the possibility of message failure and resending, resulting in the workload timeout. We execute it at some smaller scales. The result can been found in Table IV.

In conclusion for this capability workload suite, we find that Spark direct approach and Storm Trident can saturate the network resource and have a larger throughput. Spark receiver approach also has higher throughput than Storm. But Storm has much shorter latency. The capacity for Spark is better than Storm.

2) Fault-tolerance Workload Suite: This workload suite focuses on the fault tolerance evaluation for platforms. As you can see in Figure 2, we evaluate Spark Streaming direct mode, receiver mode, Storm and Storm Trident, both TPF (Throughput Penalty Factor) and LPF (Latency Penalty Factor) defined in the metrics section are obtained and calculated.

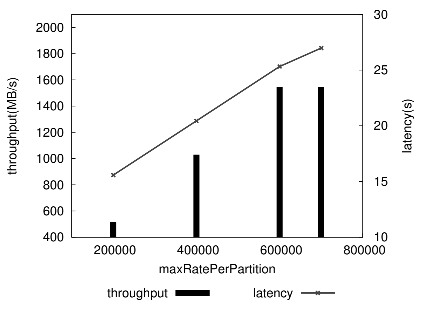
TPF is usually less than 1 and the closer to 1 the better. From Figure 2, the TPF for Spark, both for direct approach and receiver approach, is quite close to 1, which means the failure doesn't impose any significant penalty to throughput in Spark. In contrast, TPF for Storm Trident is below 0.6, suggesting over 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 is really close to 1 in both two approaches, which indicates that just TPF shows, failure doesn't impact Spark platform as much. However, the LPF for Storm Trident is quite high, which means a great impact on latency has been caused by the failure.

We choose the same record size and data scale for all fault tolerance tests, but 2 nodes seem unable to handle that amount of scale for Storm. That's why we mark it timeout for the Storm's TPF and LTP. We can infer that, for Storm and Storm Trident, due to errors occurring in some segments of topology, the related data has to be reprocessed; moreover, resources for services providing are declined, putting more loads on the existing worker nodes, so we see a big performance loss.

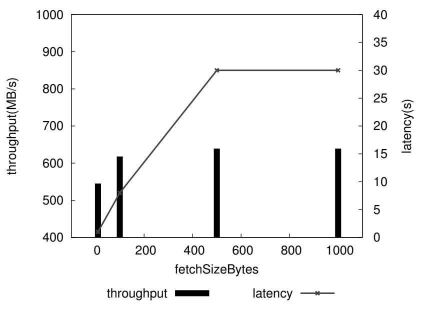
D. Key Knobs for Certain Platforms

There are some key parameters having a significant impact on the performance of each platform. The tuning result for them is present here.

The first parameter we consider is the batch size for Spark direct approach and Storm Trident. The throughput and latency for Spark direct approach at different batch size are displayed in Figure 3(a). The parameter *maxRatePartition* defines the records number per second per partition. When it changes from 200000 to 600000, the throughput becomes



(a) Spark direct approach



(b) Storm Trident

Figure 3: Batch size evaluation for Spark direct approach and Storm Trident

larger while the latency becomes longer. After 600000, the stream process comes into an unhealthy status, which means the data in a batch can not be handled in one batch interval time. Figure 3(b) shows the result for Trident. The parameter *fetchSizeBytes* defines the data scale per partition for a batch, and both throughput and latency become convergency with the increase of this parameter.

Spark Streaming has a lot of features, we tune some key parameters in our experiments, that is the batch interval, the write-ahead-logs feature and the kryo serializer. Here come our observations. **[Observation 1]** We mainly choose 20s to be the batch interval of Spark, but also experiment for 1s. The latency is obviously lower in 1 second interval. As for the throughput, in direct approach, it is quite similar in one step workload and Statistics, but worse in WordCount and DistinctCount, especially for WordCount, about ten times difference. And the result for receiver approach is approximately similar to direct approach. Since short batch interval cannot benefit much from batch work. **[Observation 2]** Write-ahead-logs mechanism is created to achieve strong fault-tolerance guarantees since Spark 1.2. When WAL option is enabled, the default 2 copies will be reduced to 1, thus greatly reduce network and disk overheads, but HDFS are replicated to 3 copies. According to our result, it doesn't effect throughput much, but increases the latency by about 12%. **[Observation 3]** Kryo serialization has always been recommended. Compared to the default java serialization, 33% throughput performance improvement has been observed, as well as 26% latency decrease, in receiver approach. While in direct approach, both throughput and latency score are steady due to the full utilization of network.

The ACK mechanism and the size of pending list for Storm are quite important to its performance and fault tolerance ability. **[Observation 1]** The ACK option is enabled to acknowledge message if it has been successfully processed. But it can be turned off to gain better performance, about 8 times increase in throughput in our experiment. **[Observation 2]** The size of pending list we mainly choose in Storm is 1000, however, when we increase it to 10000, 50% throughput increase has been observed, but the latency is about 5.6 times longer.

Table V: System resource maximal utilization for platforms

Platform	CPU	Network	Memory
Spark Streaming Direct	10%	100%	40%
Spark Streaming Receiver	10%	13%	32%
Storm Trident	60-80%	100%	60%
Storm	40%	7%	40%
Samza	60-80%	40%	20%

The number of partitions for Kafka for all previous experiments is 48, 4 partitions per broker and 4 brokers per node in Kafka configuration. We try to increase this to 162 (9 partitions per broker and 6 broker per node) to see if there is any performance improvement. We test the WordCount workload on Spark direct approach. The result shows 69% throughput improvement with 33% latency increase. This is because Spark Streaming is not fully loaded with the default concurrency number on this workload. However, the experiments on Storm and Storm Trident are still out of time, due to the large data scale beyond their handling abilities.

E. Characteristic of Hardware

System resource maximal utilization is given in Table V.

Both Spark direct approach and Storm Trident can saturate the network resource. This is the main cause of the high throughput for these two platforms. However, Storm Trident is poorly pipelined in receiving and computing. It has a notable period of network bandwidth full-load before each mini-batch starts, which is caused by data sending and receiving. Following that period, Trident starts processing every batch data and consequently has a relatively high CPU load. In contrast, Spark Streaming realizes good pipeline operations in network utilization and computing during this process, which leads to better utilization of resources. Therefore, the throughput performance of Trident is worse than Spark direct approach.

However, in Spark receiver mode, the mechanism of serializer and deserializer makes it hard to fetch the data as fast as Spark direct mode. The CPU load is just as low as in direct mode.

Storm has higher CPU utilization even in single step workloads. Besides, with the increase of data size, Storm and Storm Trident have a high rate of data resending errors especially on computation intensive applications, due to high CPU occupation. Whereas, the network utilization for storm is just at a low level.

VI. Conclusion

In this paper, we benchmark several primary streaming platforms. We first present the motivation of our work: to deal with the diversity of streaming computing platforms and the complexity of the configurations, as well as the lack of reference. We then present some preliminaries. We target on Apache Spark Streaming, Apache Storm(including Trident) and Apache Samza, and use StreamBench as our benchmark tool, with some modifications and extensions. Afterwards, some analysis techniques are defined, followed by the real experiments and analyses. For benchmark criteria, we not only consider capability, but also fault-tolerance ability. A general evaluation for different streaming platforms, some key knobs influencing performance tuning, as well as the characteristics of hardware utilization for them have been carried out.

In summary, we find that Spark direct approach and Storm Trident can saturate the network resource and have larger throughput, especially for Spark direct approach due to the well pipelined operations. Spark receiver approach also has higher throughput than Storm. But Storm has much shorter latency. Spark is quite fault tolerant and stable with the increase of data scale and node failure. Besides, some key knobs really have a great impact on the performance of these platforms.

We hope this paper can be a reference for a user who need to choose a streaming platform and deploy some streaming computation programs. For future work, complete benchmark on Apache Samza need to be carried out. And more test cases can be included, such as more kinds of hardware resources, more tuning parameters, etc.

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] 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.

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

[7] 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.

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

[9] 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.

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

[11] "Samza." http://samza.apache.org/.

[12] 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.

[13] 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*. IEEE, 2014, pp. 69–78.

[14] "Spark Streaming Programming Guide." http://spark.apache.org/docs/1.4.0/streaming-programming-guide.html.

[15] "Trident Tutorial," http://storm.apache.org/documentation/Trident-tutorial.html.

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

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

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

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

[20] 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.