Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Yangjun Wang

Stream Processing Systems Benchmark: StreamBench

Master's Thesis
Espoo, Nov 20, 2015

DRAFT! — March 28, 2016 — DRAFT!

Supervisors:	Assoc. Prof. Aristides Gionis
Advisor:	D.Sc. Gianmarco De Francisci Morales

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Yangjun Wang		
Title:			
Stream Processing Systems Benchmark: StreamBench			
Date:	Nov 20, 2015	**Pages:**	44
Major:	Foundations of Advanced Computing	**Code:**	T-110
Supervisors:	Assoc. Prof. Aristides Gionis		
Advisor:	D.Sc. Gianmarco De Francisci Morales		

Batch processing technologies(Such as MapReduce, Hive, Pig) have matured and been widely used in the industry. These systems solved the issue processing big volumes of data successfully. However, first big data need to be collected and stored in a database or file system. Then it takes time to finish batch processing analysis jobs before get any results. While there are many cases that need analysed results from unbounded sequence of data in seconds or sub-seconds. To satisfy the increasing demand of processing such stream data, several streaming processing systems are implemented and widely adopted, such as Apache Storm, Apache Spark, IBM InfoSphere Streams and Apache Flink. They all support online stream processing, high scalability, and tasks monitoring. While how to evaluate a stream processing system before choosing it to use in production development is a open question.

In this thesis, we introduce StreamBench, a benchmark framework to facilitate performance comparisons of stream processing systems. A common API component and a core set of workloads are defined. We implement the common API and run benchmarks for three widely used open source stream processing systems: Apache Storm, Spark Streaming and Flink Streaming. Therefore, a key feature of the StreamBench framework is that it is extensible – it supports easy definition of new workloads, in addition to making it easy to benchmark new stream processing systems.

Keywords:	Big Data, Stream, Benchmark, Storm, Flink, Spark
Language:	English

Acknowledgements

I want to thank Professor Aristides Gionis and my advisor Gianmarco De Francisci Morales for their good guidance.

Espoo, Nov 20, 2015

Yangjun Wang

Abbreviations and Acronyms

Abbreviations

DFS	Distribute File System
HDFS	Hadoop Distribute File System
GFS	Google File System
YCSB	Yahoo Cloud Serving Benchmark

Contents

List of Figures

Chapter 1

Introduction

Along with the rapid development of information technology, the speed of data generation increases dramatically. To process and analysis such large amount of data, the so-called Big Data, cloud computing technologies get a quick development, especially after these two papers related to MapReduce and BigTable published by Google [7, 12].

In theory, Big Data doesn't only mean "big" **v**olume. Besides volume, Big Data still have two other properties: **v**elocity and **v**ariety [14]. Velocity means the amount of data is growing at high speed. Variety refers to the various data formats. They are called three ***V***s of Big Data. When deal with Big Data, there are two types of processing models to handle different kinds of data, batch processing and stream processing. Masses of structured and semi-structured historical data (**V**olume + **V**ariety) such as the Internet Archive of Wikipedia that are usually stored in distribute file systems and processed with batch processing technologies. On the other side, stream processing is used for fast data requirements (**V**elocity + **V**ariety)[44]. Some fast data streams such as twitter stream, bank transactions and web page clicks are generated continuously in daily life.

Batch processing is generally more concerned with throughput than latency of individual components of the computation. In batch processing, data is collected and stored in file system. When the size of data reaches a threshold, batch jobs could be configured to run without manual intervention, executing against entire dataset at scale in order to produce output in the form of computational analyses and data files. Because of time consume in data collection and processing stage, depending on the size of the data being processed and the computational power of the system, output can be delayed significantly.

Stream processing is required for many practical cases which need analysed results from streaming data in a very short latency. Stock exchanges, sensor networks and user behaviour online analysis need to deal with high volume streams of real-time data. For example, a online shopping website would want give a customer accurate recommendations as soon as possible after the customer scans the website for a while. By analysing online transaction data stream, credit card fraud could be detected. Several stream processing systems are implemented and widely adopted, such as Apache Storm, Apache Spark, IBM InfoSphere Streams and Apache Flink. They all support high scalable, real-time stream processing and fault detection.

How to evaluate a real time stream processing system before choosing it to use in production development is a open question. Before these real time stream processing systems are implemented, Stonebraker et al. demonstrated the 8 requirements[42] of real-time stream processing, which gives us a standard to evaluate whether a real time stream processing system satisfies these requirements. Although some previous works[9, 15, 28, 38, 45] are done to evaluate a stream processing system or compare the performance of several systems, they check performance through system specific applications. There is no such a benchmark tool that evaluates stream processing systems with a common set of workloads. In this thesis, we introduce a benchmark framework called StreamBench to facilitate performance comparisons of stream processing systems. The extensibility property of StreamBench allows us to add more workloads or benchmark new stream processing systems.

The main topic of this thesis is stream processing systems benchmark. First, cloud computing and benchmark technology background is introduced in Chapter 2. Chapter 3 presents architecture and main features of three widely used stream processing systems: Storm, Flink and Spark Streaming. In Chapter 4, we demonstrate the design of our benchmark framework – StreamBench, including the whole architecture, test data generator and extensibility of StreamBench. Experiments and results are discussed in Chapter 5. At last, conclusions are given in Chapter 6.

Chapter 2

Background

The goal of this project is to build a benchmark framework for stream processing systems, which aim to solve issues related to **V**elocity and **V**ariety of big data[44]. In order to process continuously incoming data with a low latency, both the data and stream processing task have to be distributed. Usually the data is stored in a distributed storage system which consists of a set of data nodes. A distributed storage system could be a distributed file system, for example HDFS(§ 2.1.5), or distributed messaging system such as Kafka discussed in § 2.1.6. The stream processing task is divided into a set of sub-tasks which are distributed in a computer cluster(see § 2.1.2). The nodes in a computer cluster read data from distributed storage system and execute sub-tasks in parallel. Which achieves both data parallelism and task parallelism of parallel computing which is discussed in § 2.1.1.

In § 2.2, we present several widely accepted benchmarks of DBMS, cloud services and graph processing systems. There are many good features of the design and implementation of these benchmarks and some of them could be used in our benchmark as well. At the end, we discuss an existing benchmark of stream processing systems – `The Yahoo Streaming Benchmark`.

2.1 Cloud Computing

Cloud computing, also known as "on-demand computing", is a kind of Internet-based computing, where shared resources, data and information are provided to computers and other devices on-demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resource[32, 35]. Cloud computing and storage solutions provide users and enterprises with various capabilities to store and process their data in third-party data centers[25]. Users could use computing and storage resources as need elastically and pay according to the amount of resources used. In another way, we could say cloud computing technologies is a collection of technologies to provide elastic "pay as you go" computing. That includes scalable file system and data storage such as Amazon S3 and Dynamo, scalable processing such as MapReduce and Spark, visualization of computing resources and distributed consensus.

2.1.1 Parallel Computing

Parallel computing is a computational way in which many calculations participate and simultaneously solve a computational problem, operating on the principle that large problems could be divided into smaller ones and smaller problems could be solved at the same time. As mentioned in the beginning of this chapter, both data parallelism and task parallelism are achieved during stream processing. Besides, base on the level of parallelism there are two other types of parallel computing: bit-level parallelism and instruction-level parallelism. In the case of bit-level and instruction-level parallelism, parallelism is transparent to the programmer. Compared to serial computation, parallel computing has the following features: multiple CPUs, distributed parts of the problem, concurrent execution on each compute node. Because of these features, parallel computing could obtain better performance than serial computing.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Therefore, when the need for parallelism arises, there are two different ways to do that. The first way is "Scaling Up", in which a single powerful computer is added with more CPU cores, more memory, and more hard disks. The other way is dividing task between a large number of less powerful machines with (relatively) slow CPUs, moderate memory amounts, moderate hard disk counts, which is called "Scaling out". Compare to "Scaling up", "Scaling out" is more economically viable. Scalable cloud computing is trying to exploiting "Scaling Out" instead of "Scaling Up".

2.1.2 Computer Cluster

The "Scaling Out" strategy turns out computer cluster. A computer cluster consists of a set of computers which are connected to each other and work together so that, in many respects, they can be viewed as a single system. The components of a cluster are usually connected to each other through fast local area networks (”LAN”), with each node running its own instance of an operating system. In most circumstances, all of the nodes use the same hardware and the same operating system and are set to perform the same task, controlled and scheduled by software. The large number of less powerful machines mentioned above is a computer cluster.

One common kind of clusters is master-slave cluster which has two different types of nodes, master nodes and slave nodes. Generally, users only interact with the master node which is a specific computer managing slaves and scheduling tasks. Slave nodes are not available to users which makes the whole cluster as a single system.

As the features of computing cluster demonstrated above, it is usually used to improve performance and availability over that of a single computer. In most cases, the average computing ability of a node is less than a single computer as scheduling and communication between nodes consume resources.

2.1.3 Batch Processing and Stream Processing

According to the size of data processed per unit, processing model could be classified to two categories: batch processing and stream processing. Batch processing is very efficient in processing high **V**olume data. Where data is collected as a dataset, entered to the system, processed as a unit. The output is another data set that can be reused for computation. Depending on the size of the data being processed and the computational power of the computer cluster, the latency of a task could be measured in minutes or more. Since the processing unit of batch processing is a dataset, any modification such as incorporating new data of an existing dataset turns out a new dataset and the whole computation need start again. MapReduce and Spark are two widely used batch processing models.

In contrast, stream processing emphasizes on the **V**elocity of big data. It involves continual input and outcome of data. Each records in the data stream is processed as a unit. Therefore, data could be processed within small time period or near real time. Streaming processing gives decision makers the ability to adjust to contingencies based on events and trends developing in real-time. Beside low-latency, another key feature of stream processing is incremental computation, whenever a piece of new data arrives, attempts to save time by only recomputing those outputs which “depend on” the incorporating data without recomputing from scratch.

Except batch processing and stream processing, between them there is another processing model called mini-batch processing. Instead of processing the streaming data one record at a time, mini-batch processing model discretizes the streaming data into tiny, sub-second mini-batches. Each mini-batch is processed as a batch task. As each batch is very small, mini-batch processing obtains much better latency performance than batch processing.

	Batch	Stream
Latency	mins - hours	milliseconds - seconds
Throughput	Large	Small(relatively)
Prior **V**	Volume	Velocity

Table 2.1: Comparison of batch processing and stream process

2.1.4 MapReduce

MapReduce is a parallel programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster of commodity hardware in a reliable, fault-tolerant manner[12]. To achieve the goal, there are two primitive parallel methods (map and reduce) predefined in MapReduce programming model. A MapReduce job usually executes map tasks first to split the input data-set into independent chunks and perform map operations on each chuck in a completely parallel manner. In this step, MapReduce can take advantage of locality of data, processing it on or near the storage assets in order to reduce the distance over which it must be transmitted. The final outputs of map stage are shuffled as input of reduce tasks which performs a summary operation.

Usually, the outputs of map stage are a set of key/value pairs. Then the outputs are shuffled to reduce stage base on the key of each pair. The whole process of MapReduce could be summarized as following 3 steps:

- **Map:** Each worker node reads data from cluster with lowest transmit cost and applies the "map()" function to the local data, and writes the output to a temporary storage.
- **Shuffle:** Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
- **Reduce:**Worker nodes now process each group of output data, per key, in parallel.

One good and widely used MapReduce implementation is the Hadoop [1] MapReduce [18] which consists of a single master JobTracker and one slave

[1]http://hadoop.apache.org/

TaskTracker per cluster-node. The programming models of many stream processing systems like Storm, Flink and Spark Streaming are all inspired from MapReduce. Operators `Map` and `Reduce` are either built-in these systems or could be implemented with provided built-in APIs. For example, Flink supports `Map`, `Reduce` and some other MapReduce-inspired operations like `FlatMap`, `Filter` by default. In Storm, all these mentioned operators could be implemented with APIs of `Spout` and `Bolt`.

2.1.5 Hadoop Distribution File Systems

Hadoop Distributed File System [16] is open source clone of Google File System (GFS) [20] that is deployed on computing cluster. In Hadoop MapReduce framework, locality relies on Hadoop Distributed File System (HDFS) that can fairly divide input file into several splits across each worker in balance. In another word, MapReduce is built on the distributed file system and executes read/write operations through distributed file system. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. [16] The assumptions and goals of HDFS includes: hardware failure, high throughput, large dataset, streaming access, data load balance and simple coherency model, "Moving Computation is Cheaper than Moving Data", and portability across heterogeneous hardware and software platforms.

In general, HDFS and MapReduce always works together. In a distribute cluster, each datanode in HDFS runs a TaskTracker as a slave node in MapReduce. MapReduce retrieves data from HDFS and executes computation and finally writes results back to HDFS.

2.1.6 Kafka

Apache Kafka is a high-throughput distributed publish-subscrib messing system which was originally developed by LinkedIn. Now it is one top level project of the Apache Software Foundation. It aims to provide a unified, high-throughput, low-latency platform for handling continuous data feeds. A stream processing system subscribes from Kafka will get notified within very short time after a publisher publishes something. In StreamBench, we use Kafka provide messaging service. Before we go into architecture of Kafak, there are some basic messaging terminologies: [17]

- **Topic:** Kafka maintains feeds of messages in categories called topics.

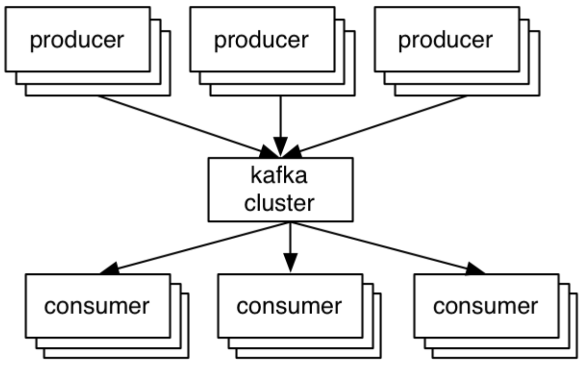

Figure 2.1: Kafka producer and consumer [17]

- **Producer:** Processes that publish messages to a Kafka topic are called producers.
- **Consumer:** Processes that subscribe to topics and process the feed of published messages are consumers.
- **Broker:** Kafka is run as a cluster comprised of one or more servers each of which is called a broker.

As Figure 2.1 shown, producers send messages over the network to the Kafka cluster which holds on to these records and hands them out to consumers. More specifically, producers publish their messages to a topic, and consumers subscribe to one or more topics. Each topic could have multiple partitions that are distributed over the servers in Kafka cluster, allowing a topic to hold more data than storage capacity of any server. Each partition is replicated across a configurable number of servers for fault tolerance. Each partition is an ordered, immutable sequence of messages that is continually appended to a log. The messages in the partitions are each assigned a sequential id number called the offset that uniquely identifies each message within the partition. Figure 2.2 shows a producer process appending to the logs for the two partitions, and a consumer reading from partitions sequentially.

At a high-level Kafka gives the following guarantees: [17]

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a message M1 is sent by the same producer as a message M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.

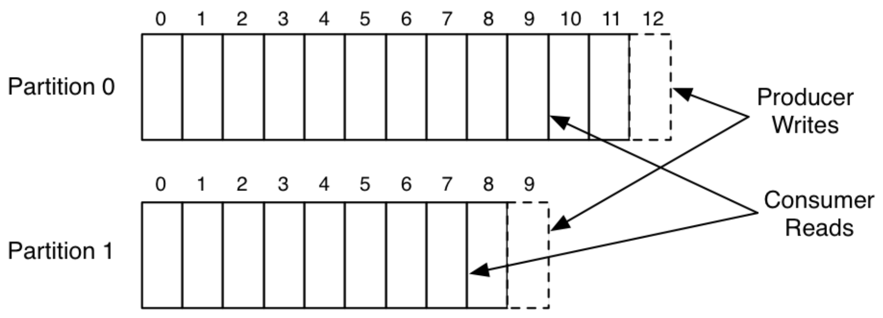

Figure 2.2: Kafka topic partitions

- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any messages committed to the log.

Another very important feature of Kafka is messages with the same key will be sent to the same partition. When a distributed application consumes data from a kafka topic in parallel, data with the same key goes to the same executor which could avoid data shuffle.

2.2 Benchmark

As system advanced and complicated, it became more difficult to compare the performance of various systems simply by looking at their specifications. For one kind of systems, there are always a sequence of performance metrics which indicate the performance of a specific system, such as average response time of a query in DBMS. In computing, a benchmark is the act of running a set of programs mimicking a particular type of workload on a system, in order to assess the relative performance metrics of a system. Many benchmarks are designed and widely used in industry to compare systems.

In this Section, we discuss benchmarks for different computer systems: DBMS, Cloud Data Service and Graph Processing System. An existing benchmark of Stream processing system developed by Yahoo is also present in § 2.2.5.

2.2.1 Traditional Database Benchmarks

Traditional database management systems were evaluated with industry standard benchmarks like TPC-C[10], TPC-H[11]. These have focused on simulating complete business computing environment where plenty of users execute business oriented ad-hoc queries that involve transactions, big table scan, join, and aggregation. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions[11]. The integrity of the data is verified during the process of the execution of the benchmark to check whether the DBMS corrupt the data. If the data is corrupted, the benchmark measurement is rejected entirely[13]. Benchmark systems for DBMS mature, with data and workloads simulating real common business use cases, they could evaluate performance of DBMS very well. Some other works were done related to specific business model.

Linkbench [5] benchmarks database systems which store “social network” data specifically. The workloads of database operations are based on Facebook’s production workload and the data is also generated in such a way that key properties of the data match the production social graph data in Facebook. LinkBench provides a realistic and challenging test for persistent storage of social and web service data.

2.2.2 Cloud Service Benchmarks

As the data size keep increasing, traditional database management systems could not handle some use cases with very big size data very well. To solve this issue, there are plenty of NoSQL database systems developed for cloud data serving. With the widespread use of such cloud services, several benchmarks are introduced to evaluate these cloud systems.

One widely used and accepted extensible cloud serving benchmark named *Yahoo! Cloud Servicing Benchmark*(YCSB) developed by Yahoo[8]. It proposes two benchmark tiers for evaluating the performance and scalability of cloud data serving systems such as Cassandra, HBase, and CouchDB. A core set of workloads are developed to evaluate different tradeoffs of cloud serving systems. Such as write/read heavy workloads to determine whether system is write optimised or read optimised. To evaluate transaction features in later NoSQL database, YCSB+T [13] extends YCSB with a specific workload for transaction called Closed Economy Workload(CEW). A validation phase is added to the workload executor to check consistency of these cloud databases.

YCSB++ [39] is another set of extensions of YCSB to benchmark other five advance features of cloud databases such as bulk insertions, server-side filtering. YCSB++ could run multiple clients on different machines that coordinated with Apache ZooKeeper, which increases test ability of benchmark framework. Pokluda and Sun [40] explore the design and implementation of two representative systems and provide benchmark results using YCSB. In addition to the performance aspect of NoSQL systems, they also benchmark availability and providing an analysis of the failover characteristics of each. Kuhlenkamp et al. [29] made some contributions to benchmarking scalability and elasticity of two popular cloud database systems HBase and Cassandra. The benchmark strategy is changing workloads and/or system capacity between workload runs, load and/or system capacity are changed.

These efforts are island solutions and not policed by any industry consortia. BigBench aims to be implemented as an industry standard big data benchmark [19]. It is an end to end benchmark identify business levers of big data analytics. Inherit from TPC-DS benchmark, BigBench implements the complete use-case of a realistic retail business. The data model of which covers three **V**s of big data system: volume, velocity and variety. The main part of the workload is the set of queries to be executed against the data model. These queries are designed along one business dimension and three technical dimensions[19].

These benchmarks aim to evaluate performance of NoSQL systems. Even though they couldn't be applied to stream processing systems directly. There are some good features of them inspiring us to implement StreamBench. For example, configurability and extensibility of YCSB are also implemented in StreamBench. The workloads in StreamBench is also business relevant which is inspired by BigBench/TPC-H.

2.2.3 Distributed Graph Benchmarks

A specific type of big data which keeps increasing in day-to-day business is graph data. The growing scale and importance of graph data has driven the development of numerous specialized graph processing systems including Google's proprietary Pregel system [31], and Apache Giraph [1], PowerGraph[21]. Compare to DBMSs, stream processing systems are more similar to graph processing ones which are very diverse with non-standard features. In the paper, Guo et al. [24] demonstrate the diversity of graph-processing platforms and challenges to benchmark graph-processing platforms. Among these challenges, some are common for general benchmark systems, such as evaluation process, dataset selection and result reporting. To address these issues of evaluating graph-processing platforms, Guo et al. implemented a benchmark suit using an empirical performance-evaluation method which includes four stages: identifying the performance aspects and metrics of interest; defining and selecting representative datasets and algorithms; implementing, configuring, and executing the tests; and analyizing the results. In order to create an industry-accepted benchmark, this method still raises some issues. In latest released papers[6, 26], the team implemented a benchmark called Graphalytics for graph-processing platforms.

2.2.4 Existing stream processing benchmarks

To help users have a better understanding of stream processing system and choose one intelligently in practical work, there are already several tests or benchmarks of stream processing systems published on the Internet. Early work by Córdova [9] focuses on analysing latency of two notable real time stream systems: Spark Streaming and Storm Trident. While it could not be a standard benchmark because it is not extensible and both Spark Streaming and Storm Trident are micro-batch processing model. Moreover, the data model and workloads are quite simple which could not reflect the real use cases in business.

IBM compares the performance of IBM InfoSphere Streams against Apache Storm with a real-world stream processing application which enables email classification to detect online spam[34]. This application is a good reflection of stream processing systems used in practical projects. The processing pipeline for the benchmark email classification system is divided into 7 stages and implemented by InfoSphere Streams and Apache Storm separately. But the workload includes too many steps(operations) makes it hard to detect the possible performance bottleneck. Another main drawback of this approach is there is only one scenario.

LinkedIn benchmarked its own real-time streaming process system Samza running four simple jobs and got excellent performance: 1.2 million messages per second on a single node [15]. Process-envelopes metric is used to measure message-processing rate that indicates how fast Samza processes the input stream messages. A monitoring tool called inGraph is used to monitor these performance metrics. When the job starts, these performance metrics are emitted to Kafka and later consumed by inGraph. In StreamBench, we use similar methods logging information related to performance metrics and analysing by a package of python scripts. Since this benchmark is designed for Samza specifically, it is not a standard benchmark for other stream processing systems.

Xinh [45] in her blog compared Storm and Spark Streaming side-by-side, including processing model, latency, fault tolerance and data guarantees.

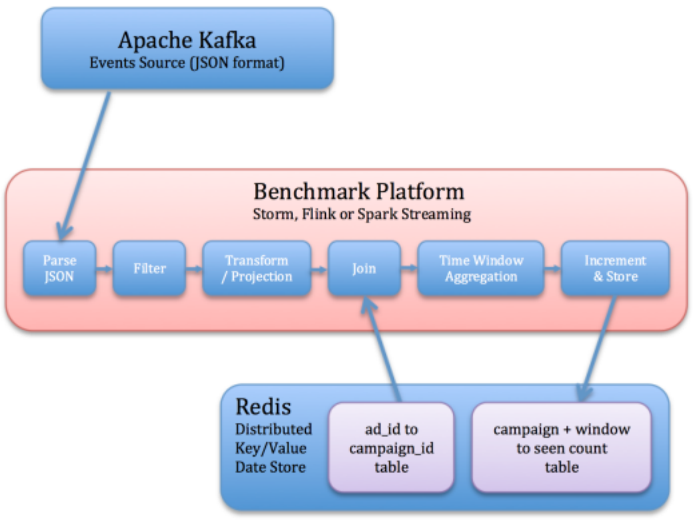
Apache Kafka
Events Source (JSON format)
Benchmark Platform
Storm, Flink or Spark Streaming
Parse JSON
Filter
Transform / Projection
Join
Time Window Aggregation
Increment & Store
Redis
Distributed Key/Value Date Store
ad_id to campaign_id table
campaign + window to seen count table

Figure 2.3: Operations flow of YSB [43]

There are some other similar works [37, 38]. But there is no workload designed and experiment result provided in these works. Recently, Yahoo Storm Team demonstrated a stream processing benchmark. Design and more features of *The Yahoo Streaming Benchmark* will be introduced in detail in the next section because it is the closest benchmark to our project.

2.2.5 The Yahoo Streaming Benchmark

The Yahoo Streaming Benchmark (YSB) is introduced to analysis what Storm is good at and where it needs to be improved compared to other stream processing systems by Yahoo storm team [43]. The benchmark is a single advertisement application to identify relevant advertisement events. There are a number of advertising campaigns, and a number of advertisements for each campaign. The application need read various JSON events from a Kafka topic, identify the relevant events, and store a windowed count of relevant events per campaign into Redis. The flow of operations could be shown as Figure 2.3.

Each event (message) in Kafka topic contains a timestamp marking the time producer created this event. Truncating this timestamp to a particular digit gives the begin-time of the time window that the event belongs in. When each window is updated in Redis, the last updated time is recored.

After each run, a utility reads windows from Redis and compares the windows' times to their last update times in Redis, yielding a latency data point. Because the last event for a window cannot have been emitted after the window closed but will be very shortly before, the difference between a window's time and its last update time minus its duration represents the time it took for the final tuple in a window to go from Kafka to Redis through the application.

$$finalEventLatency = (lastUpdatedTime - createTime) - duration$$

- **finalEventLatency:** latency of an event;
- **lastUpdatedTime:** the latest time that an event updated in Redis;
- **createTime:** the time when an event created;
- **duration:** duration of a window.

More details about how the benchmark setup and the configuration of experiment environment could be found online [2]. One shortcoming of this benchmark is one single workload could not reflect features of stream processing systems comprehensively, even the steps of benchmark flow attempt to probe common operations performed on data streams. Moreover, Redis is used to perform the join operator that could affect performance, therefore, there would be inaccuracy between the benchmark results and real performances of these stream processing systems. From experiments demonstrated in this page, Storm 0.10.0 was not able to handle throughputs above 135,000 events per second. The largest rate at which Kafka emitted data events into the Flink benchmark is varied 170,000 events/sec which doesn't reach throughput of Flink. The benchmark measures the latency of the frameworks under relatively low throughput scenarios, and establishes that both Flink and Storm can achieve sub-second latencies in these scenarios, while Spark Streaming has much higher latency.

To benchmark the real throughput of Flink, Grier [22] reran YSB with some modifications that used the features in Flink to compute the windowed aggregates directly in Flink. In the last step of YSB, window updates to Redis is implemented with custom code with a separate user thread to flush results to Redis periodically on both Storm and Flink, which doesn't take advantage of Flink?s window API for computing windows. By reimplement the workload with Flink's window API, it came up with much better throughput numbers while still maintaining sub-second latencies. In StreamBench, we design a set of common APIs which are implemented with the best solution on different stream processing systems independently.

[2]`http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`

Chapter 3

Stream Processing Platforms

There are many stream processing systems that include Apache Apex, Aurora, S4, Storm, Samza, Flink, Spark Streaming, IBM InfoSphere Streams, and Amazon Kinesis. Some systems are being improved, and new stream processing systems are being developed. The design, computational model, and architecture of these systems are different so that they own different features. Among these systems, some are not open sourced and we couldn't download and setup a stream processing cluster. For instance, Amazon Kinesis is only available in AWS. Some ones are open source systems with system design, source code public and it is possible to set up our own distribute cluster to run experiment benchmarks. In StreamBench, we choose three popular and widely used open source stream processing systems which have many similarities. These systems are Storm, Flink and Spark Streaming.

First, all these systems are MapReduce inspired, support transformations similar to `Map` and aggregations like `Reduce`. The computational models of these systems are also similar which are graphs consisted of transformations and/or aggregations. For example, in Storm a topology is a graph of stream transformations where each node is a spout or bolt. Unlike MapReduce, these systems deal with unbound stream of tuples and support integration with Kafka, a distributed messaging system discussed in § 2.1.6, in which stream data is distributed in different partitions. Moreover, these stream processing frameworks run tasks in parallel. Integrate with Kafka, these system achieve both data parallelism and task parallelism of parallel computing. Another similarity of these systems is horizontal scalability. Work with these systems, we could add more machines into a stream processing cluster to achieve better performance.

In this chapter, we introduce these three systems in detail from different perspectives: system architecture, core concepts, and key features. Several other stream processing systems are also discussed.

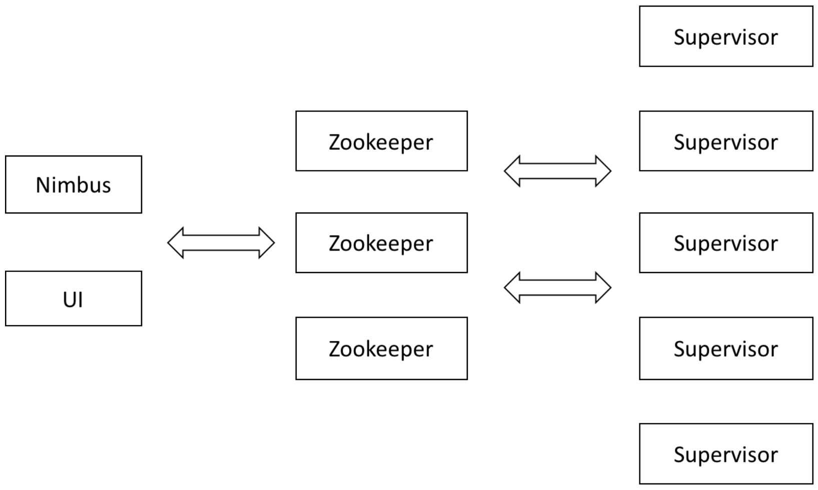

Figure 3.1: Storm cluster components

3.1 Apache Storm

Apache Storm, one of oldest distributed stream processing system which was open sourced by Twitter in 2011 and became Apache top-level project in 2014. Storm is to realtime data processing as Apache Hadoop is to batch data processing. A Hadoop job is finished after all data processing operations are done. Unlike Hadoop, Storm is an always-active service that receives and processes unbound streams of data and delivers that data instantaneously, in realtime. Storm solutions can also provide guaranteed processing of data, with the ability to replay data that was not successfully processed the first time.With its simple programming interface, Storm allows application developers to write applications that analyze streams of tuples of data; a tuple may can contain object of any type.

3.1.1 Storm Architecture

As a distributed stream processing system, a storm cluster consists of a set of nodes. Figure 3.1 shows components of a storm cluster which contains four different kinds of nodes, ”Supervisor”, ”Nimbus”, ”Zookeeper” and ”UI”.

”Nimbus” is a daemon runs on the master node which is similar to Hadoop’s ”JobTracker”. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. Each worker node runs a daemon called the ”Supervisor.” It listens for work assigned to its machine and starts and stops worker processes as dictated by

Nimbus. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all state is kept in Zookeeper or on local disk. Hence, Storm clusters are stable and fault-tolerant

"UI" is a daemon which monitors summary of cluster and running topologies on a web interface. Properties of a topology (such as id, name, status and uptime) and emitted tuples of a spout/bolt also could be found in "ui". More detail about understanding the Storm UI could be found on the page [1].

3.1.2 Computational Model

At the core of Storm's data stream processing is a computational topology, which is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams as shown in Figure 3.2.

Spouts are the sources of streams in a topology which will read tuples from external sources (e.g. Twitter API, Kafka) or from disk and emit them in the topology. Bolt receives input streams from spout or other bolt, process them and produce output streams. They encapsulate the application logic which dictates how tuples are processed, transformed,aggregated, stored, or re-emitted to other nodes in the topology for further processing. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.

A stream in topology is an unbounded sequence of tuples. Spouts read tuples from external sources continuously. Once a topology is submitted, it processes messages forever, or until it is killed. Storm will automatically reassign any failed tasks. Each node in a Storm topology executes in parallel. In your topology, you can specify how much parallelism you want for each node, and then Storm will spawn that number of threads across the cluster to do the execution.

Additionally, Storm guarantees that there will be no data loss, every tuple will be fully processed at least once. This is achieved by following steps:

1. Each tuple emitted in a spout is specified with an unique message ID;

2. Each tuple emitted in a bolt, anchor it with corresponding input tuples;

[1]`http://www.malinga.me/reading-and-understanding-the-storm-ui-storm-ui-explained/`

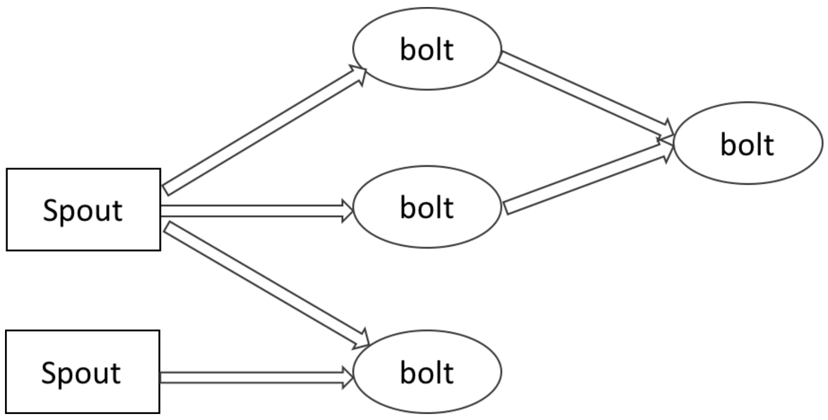

Figure 3.2: Storm topology model

3. When bolts are done processing the input tuple, ack or fail the input tuple

This mode tracks whether each spout tuple is ?fully? processed within a configured timeout. Any input tuple not fully processed within the timeout is re-emitted. This means the same tuple could be processed more than once and messages can be processed out-of-order.

3.2 Apache Flink

Apache Flink used to be known as Stratosphere which was started off as an academic open source project in Berlin's Technical University in 2010. Later, it became a part of the Apache Software Foundation incubator and was accepted as an Apache top-level project in December 2014. Apache Flink aims to be a next generation system for big data system. It is a replacement for Hadoop MapReduce that works in both batch and streaming models. Its defining feature is its ability to process streaming data in real time. In Flink, batch processing applications run efficiently as special cases of stream processing applications.

3.2.1 Flink Architecture

The architecture of Flink is a typical master-slave architecture that is quite similar with other scaleable distribute cloud systems. The system consists of a JobManager and one or more TaskManagers. The JobManager is the

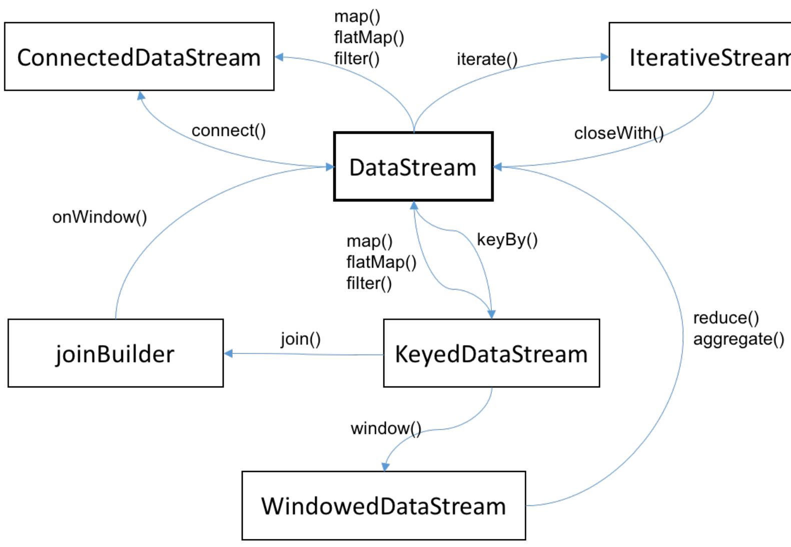

Figure 3.3: Flink computing model

coordinator of the Flink system, while the TaskManagers are the workers that execute parts of the parallel programs.

In Hadoop MapReduce, a operation wouldn't start until the previous operation is finished. In Flink records are forwarded to receiving tasks as soon as they are produced which is called pipelined data transfers. For efficiency, these records are collected in a buffer which is sent over the network once it is full or a certain time threshold is met. This threshold controls the latency of records because it specifies the maximum amount of time that a record will stay in a buffer without being sent to the next task.

Flink's runtime natively supports both domains due to pipelined data transfers between parallel tasks which includes pipelined shuffles. Batch jobs can be optionally executed using blocking data transfers. They are special cases of stream processing applications.

3.2.2 Computational Model

The computational model of Flink is graph consists of `transformaitons` and `sources`, which is similar to Storm. Edges in the graph indicate different states of `DataStream`. That is different with Storm, in which input of of a bolt is always a stream of tuples without specific state. These states includes `DataStream`, `KeyedDataStream`, `WindowedDataStream`, and `ConnectedDataStream`[2]:

- A KeyedDataStream represents a data stream where elements are evaluated as "grouped" by a specified key.
- In a WindowedDataStream, records in the stream are organized into groups by the key, and per group, they are windowed together by the windowing. A WindowedDataStream is always created from a KeyedDataStream that assigns records to groups.
- The ConnectedDataStream is a way to share state between two tuple-at-a-time operations. It can be through of as executing two `map` (or `flatMap`) in the same object.

The transformations between different states of DataStream could be shown as Figure 3.3.

Similar to Storm, Flink supports cycle in the graph model. Flink supports the iterations in native platform by defining a step function and embedding it into a special iteration operator. There are two variants of this operator: Iterate and Delta Iterate. Both operators repeatedly invoke the step function on the current iteration state until a certain termination condition is reached.

3.3 Apache Spark

Apache Spark is currently the most active open source large-scale data processing framework used in many enterprises across the world. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project. Compared to other big data and MapReduce technologies like Hadoop and Storm, Spark has several advantages.

Spark is very easy to use by providing APIs in Scala, Java, Python and R languages. Moreover, there are more than 80 high-level built-in operators. In the case of implementing a simple WordCount application, all the execution logic code could be written in one line with Spark' Scala API.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Spark achieves much better performance than Hadoop MapReduce mainly because Spark supports in-memory computing. By using RDDs which will be discussed in § 3.3.1, intermediate results are stored in memory and reused for further performing functions thereafter, as opposed to being written to hard disk.

3.3.1 Resilient Distributed Dataset(RDD)

Resilient **D**istributed **D**ataset(RDD) is a fault-tolerant abstraction of read-only collection of elements partitioned across the distributed computer nodes in memory which can be operated on in parallel. To achieve fault-tolerant property, RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs[46]. An RDD could keep all information about how it was derived from other datasets to compute its partitions from data in table storage. Therefore, RDDs are fault tolerance because they could be reconstructed from a failure with these kept information.

RDD supports two different kinds of operations, transformation and action. When a transformation operation is called on a RDD object, a new RDD returned and the original RDD remains the same. For example, map is a transformation that passes each element in RDD through a function and returns a new RDD representing the results. Some of the transformation operations are `map`, `filter`, `flatMap`, `groupByKey`, and `reduceByKey`.

An action returns a value to the driver program after running a computation on the RDD. One representative action is `reduce` that aggregates all the elements of the RDD using some function and returns the final result to the driver program. Other actions include `collect`, `count`, and `save`.

By default, each transformed RDD may be recomputed each time you run an action on it. However, by caching RDD in memory, allowing it to be reused efficiently across parallel operations. The recomputation of cached RDD is avoid and a significant amount of disk I/O could be reduced. Especially in the case of looping jobs, the performance would be improved. In Spark, users can manually specify if working sets are cached or not. The runtime engine would manage low-level caching mechanism like how to distribute cache blocks.

3.3.2 Computational Model

The execution engine of Spark supports lazy evaluation. When transformations applied to an RDD, the execution engine doesn't do any computation, but remember the transformations. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

Spark has an advanced directed acyclic graph (DAG) execution engine that implements stage-oriented scheduling. Whenever a user runs an action

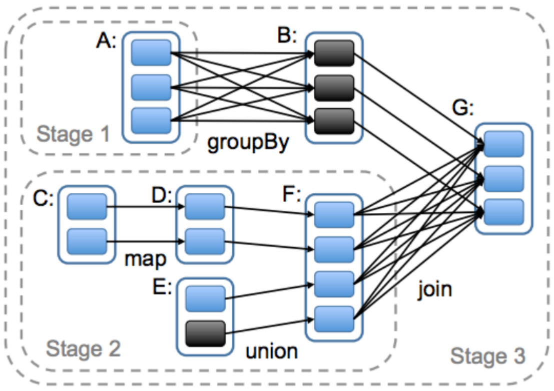

Figure 3.4: Spark job stages[46]

(e.g., count or save) on an RDD, the scheduler examines that RDD?s lineage graph to build a DAG of stages to execute, as illustrated in Figure 3.4. Each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can shortcircuit the computation of a parent RDD[46]. The execution engine executes each stage in computer nodes that perform on distributed partitions of RDD in parallel. Different with Flink's pipeline execution model, Spark does not execute multi stages simultaneously[41]. In the DAG, tasks of a stage wouldn't start until all its dependencies are done.

3.3.3 Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Spark Streaming 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. The model of Spark Streaming is different from that of Storm and Flink, which process stream records one by one. In Spark Streaming, data streams are divided according to a batch interval which could be configured in the program. Divided data stream is abstracted as discretized stream or DStream, which represents a continuous stream of data.

DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of micro RDDs.

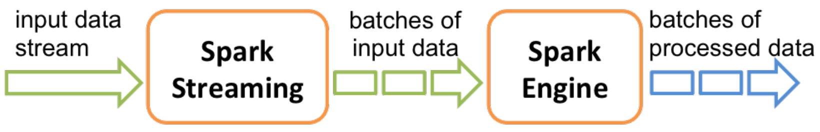

Figure 3.5: Spark Streaming Model

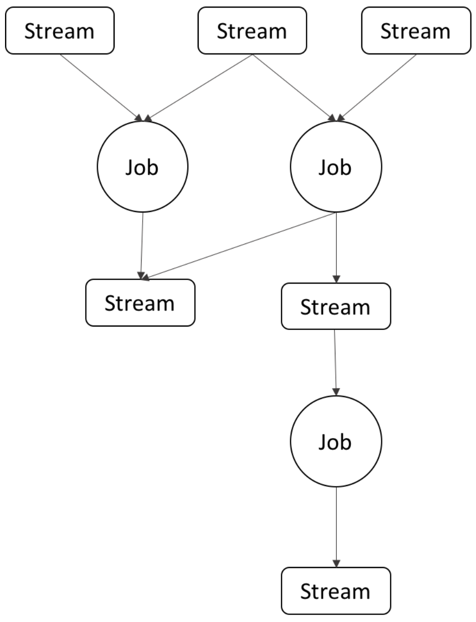

Figure 3.6: Samza DataFlow Graph

After applied transformations or actions on a DStream, a new DStream or result values would be get which can be pushed out to filesystems, databases, and live dashboards.

3.4 Other Stream Processing Systems

3.4.1 Apache Samza

Apache Samza is a top-level project of Apache Software Foundation which open sourced by LinkedIn to solve stream processing requirements. It's been in production at LinkedIn for several years and currently runs on hundreds

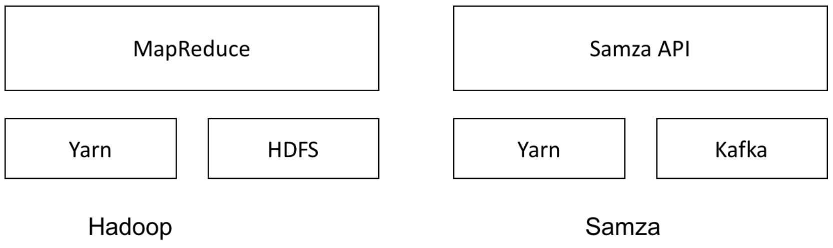

Figure 3.7: Samza and Hadoop architecture

of machines.

A Samza application is constructed out of streams and jobs. A stream is composed of immutable sequences of messages of a similar type or category. A Samza job is code that performs a logical transformation on a set of input streams to append output messages to set of output streams. In order to scale the throughput of the stream processor, streams are are are broken into partitions and jobs are broken into smaller units of execution called tasks. Each task consumes data from one or more partitions for each of the job's input streams. Multiple jobs could be composed together to create a dataflow graph, where the nodes are streams containing data, and the edges are jobs performing transformations.

Except streams and jobs, Samza uses YARN as execution layer. The architecture of Samza follows a similar pattern to Hadoop which could be shown as Figure 3.7.

3.4.2 Apache S4

S4 is another open source distributed, scalable, fault-tolerant, stream data processing platform released by Yahoo.

In S4, a stream is defined as a sequence of events of the form (**K,V**) where **K** is the key of record tuple, and V is the corresponding value. Processing Element(PE) is the basis computational unit that consume streams and applies computational logic. After takes in an event, a PE either emits one or more events which may be consumed by other PEs or publishes results[36].

S4 makes sure that two events with the same key end up being processed on the same machine. Crucial to the scalability of S4 is the idea that every instance of a processing element handles only one key. The size of an S4 cluster corresponds to the number of logical partitions.

Chapter 4

Benchmark Design

We developed a tool, called StreamBench, to execute benchmark workloads on stream processing systems. A key feature of StreamBench is extensibility, so that it could be extended not only to run new workloads but also to benchmark new stream processing systems. We have used StreamBench to measure the performance of several stream processing systems, as we report in the next chapter. StreamBench is also available under an open source license, so that others may use and extend it, and contribute new workloads and stream processing system interfaces.

In this chapter, we describe the architecture of StreamBench and introduce more detail of main components of StreamBench.

4.1 Architecture

The main component of StreamBench is a Java program for consuming data from partitioned kafka topic and runs workloads on stream processing cluster. In the program, there is a set of common APIs for stream processing which could be engined by stream processing systems. Currently we support these APIs on three stream processing systems: Storm, Flink and Spark Streaming. Several workloads are implemented by these common APIs. In StreamBench we implemented three workloads to benchmark performance of stream processing systems in different aspects. The architecture of StreamBench is shown in Figure 4.1.

Except the core Java program, the architecture also includes three more components: Cluster Deploy, Data Generator and Statistic. Section 4.2 describes how to use cluster deploy scripts to setup experiment environment. Data generators generate test data for workloads and send it to kafka cluster that is demonstrated detailedly in Section 4.4. The Statistic component dis-

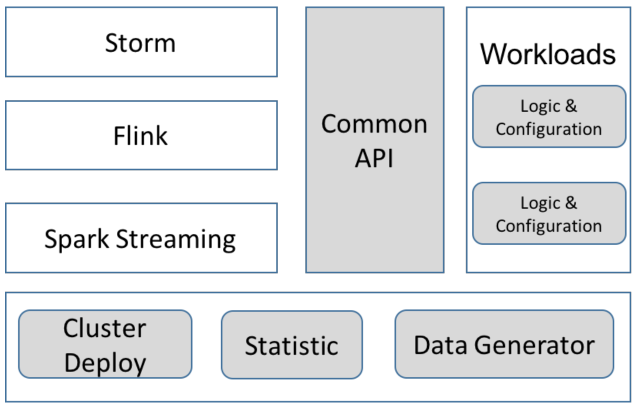

Figure 4.1: StreamBench architecture

cussed in Section 4.5 includes experiment logging and performance statistic.

4.2 Experiment Environment Setup

The operator system running on experiment nodes is Ubuntu 14.04 LTS. Benchmarked stream processing systems are Spark-1.5.1, Storm-0.10.0 and Flink-0.10.1. To enable checkpoint feature of Spark, Hadoop2.6(HDFS) is installed in compute cluster. Kafka 0.8.2.1 is running as distribute message system here.

To deploy these software in compute cluster and kafka cluster automatically, we developed a set of python script. The prerequisites of using these scripts include internet access, ssh passwordless login between nodes in cluster and cluster configuration that describes which nodes are compute node or kafka node and where is the master node. The basic logic of deploy scripts is to download softwares online and install them, then replace configure files which are contained in a Github repository. For detail information of how to use cluster deploy scripts and configure of Storm, Flink, Spark and Kafka, please check this Github repository [1].

4.3 Workloads

In StreamBench, a workload consists of a stream processing application and one or more kafka topics. The application consumes messages from kafka

[1]`https://github.com/wangyangjun/StreamBench`

cluster and executes operations or transformations on the messages. We have developed 3 workloads to evaluate different aspects of a system's performance. Each workload contains a representative operation or feature of stream processing system that can be used to evaluate systems at one particular point in the performance space. We have not attempted to exhaustively examine the entire performance space. As StreamBench is open sourced, users could also defined their own workloads either by defining a new set of workload parameters, or if necessary by implement a new workload which is discussed detailedly in section 4.6.

4.3.1 Basic Operators

With the widespread use of computer technologies, there is increasing demand of processing unbounded, continuous input streams. In most cases, only basic operations need to be performed on the data streams such as `map`, `reduce`. One good sample is stream WordCount. WordCount is a very common sample application of Hadoop MapReduce that counts the number of occurrences of each word in a given input set.[18] Similarly, many stream processing systems also support it as an sample application to count words in a given input stream. Stream WordCount is implemented with basic operations which are supported by almost all stream processing systems. It means either the system has such operations by default or the operations could be implemented with provided built-in APIs. Other basic operations include `flatMap`, `mapToPair` and `filter` which are similar to `map` and could be implemented by specializing `map` if not supported by default. In StreamBench, there are a set of corresponding basic APIs defined. The pseudocode of WordCount implemented with these basic APIs could be abstracted as Algorithm 1.

Algorithm 1 WordCount

```
1: sentenceStream.flatMap(...)
2:                .mapToPair(...)
3:                .reduceByKey(...)
4:                .updateStateByKey(...)
```

One special case of the basic APIs is `updateStateByKey`. Only in Spark Streaming there is a corresponding built-in operation. As discussed in Section 3.3, the computing model of Spark Streaming is micro-batch which is different with that of other stream processing systems. The results of operation `reduceByKey` of WordCount running in Spark Streaming is word counts of one single micro batch data set. Operation `updateStateByKey` is used to accumulate word counts in Spark Streaming. Because the model of Flink and Storm is stream processing and accumulated word counts are returned from `reduceByKey` directly. Therefore, when implementing the API `updateStateByKey` with Flink and Storm engine, nothing need to do. The goal of this workload is to evaluate the performance of stream processing systems executing basic operations.

4.3.2 Join Operator

Besides the cases in which only basic operations are performed, another typical type of stream use case is processing joins over unbounded streams. For example, in a surveillance application, we may want to correlate cell phone traffic with email traffic. Theoretically unbounded memory is required to processing join over unbounded input streams, since every record in one infinite stream must be compared with every record in the other. Obviously, this is not practical.[27] Since the memory of a machine is limited, we need restrict the number of records stored for each stream with a time window.

A window join takes two key-value pair streams of tuple, say stream *S1:* `(k, v1)` and stream *S2:* `(k, v2)`, along with window sizes for both *S1* and *S2* as input. The output is a stream of tuple `(k, v1, v2)`. Assuming a sliding window join between stream *S1* and stream *S2*, a new tuple arrival from stream *S1*, then a summary of steps to preform join is the following:

1. Scan window of stream *S2* to find any tuple which has the same key with this new tuple and propagate the result;
2. (a) Insert the new tuple into stream *S1*'s window or
   (b) invalidate target tuple in stream *S2*'s window if found;
3. Invalidate all expired tuples in stream *S1*'s window.

In step 2, there are two options. In the case of at most one tuple which has the same key with the new tuple could be found in stream *S2*, option (b) is executed. Otherwise, option (a) is performed. Every time new tuple arrives stream *S1*, window of stream *S2* need be scanned. That reduces the performance of join operator, especially when the window is big.

In the first case mentioned above, with a data structure named `cachedHashTable` there is another way to implement stream join. The tuples in the window of a stream are stored in a cached hash table. Each tuple is cached for window time and expired tuples are invalidated automatically. One of such a

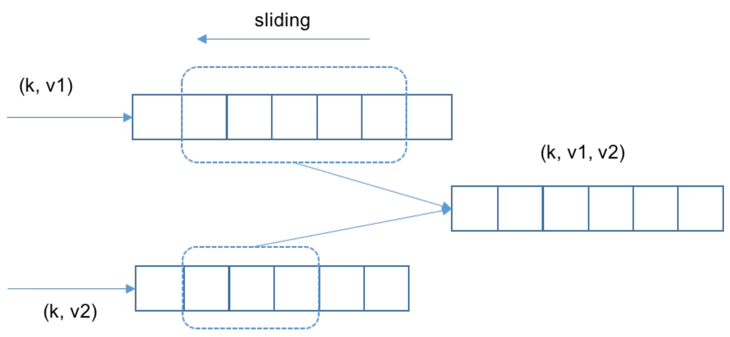

Figure 4.2: Window join scenario

`cachedHashTable` could be found in Guava.[2] Instead of scanning window of stream *S2*, we could find tuple with the same key in *S2* directly by calling `cachedHashTable.get(k)`. In theory, this implementation achieves better performance.

Since Spark Streaming doesn't process tuples in a stream one by one, the join operator in Spark Streaming has different behaviours. In each batch interval, the RDD generated by stream1 will be joined with the RDD generated by stream2. For windowed streams, as long as slide durations of two windowed streams are the same, in each slide duration, the RDDs generated by two windowed streams will be joined. Because of this, window join in Spark Streaming could only make sure that a tuple in one stream will always be joined with corresponding tuple in the other stream that arrived earlier up to a configureable window time. Otherwise, repeat joined tuples would exist in generated RDDs of joined stream. As Figure 4.3 shown, a tuple in Stream2 could be always joined with a corresponding tuple in Stream1 that arrived up to 2 seconds earlier. Since the slide duration of Stream2 is equal to its window size, no repeat joined tuple exists. On the other hand, it is possible that a tuple arrives earlier from Stream2 than the corresponding tuple in Stream1 couldn't be joined. Figure 4.4 exemplifies that there are tuples joined repeatedly when slide duration of Stream2 is not equal to its window size.

To evaluate performance of join operator in stream processing systems, we designed a workload called AdvClick which joins two streams in a online advertisement system. Every second there are a huge number of web pages opened which contain advertisement slots. A corresponding stream of shown

[2]`http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheBuilder.html`

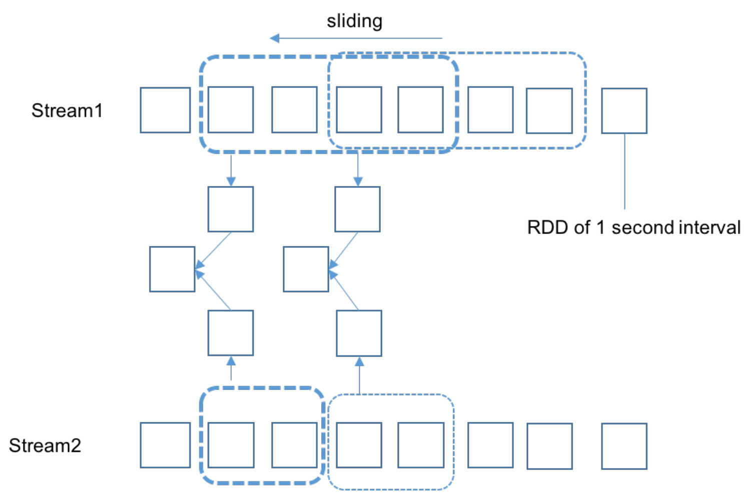

Figure 4.3: Spark Stream join without repeated tuple

advertisements is generated in the system and a record in the stream could be simply described as a tuple of `(id, shown time)`. Some of advertisements would be clicked by users and clicked advertisements is a stream which could be abstracted as a unbound tuples of `(id, clicked time)`. We assume that a record of advertisement shown always arrivers earlier than corresponding click record. Normally, if an advertisement is attractive to a user, it will be clicked in seconds or a few minutes after shown. We call such a click of an attractive advertisement valid click. To bill a customer, we need count all valid clicks regularly for advertisements of this customer. Which could be counted after joining stream `advertisement clicks` and stream `shown advertisements` with a configureable window time.

4.3.3 Iterate Operator

Iterative algorithms occur in many domains of data analysis, such as machine learning or graph analysis. Many stream data processing tasks require iterative sub-computations as well. To achieve these requirements, a data processing system should have the capacity to perform iterative processing on a real-time data stream. To achieve iterative sub-computations, low-latency interactive access to results and consistent intermediate outputs, Murray et al. introduced a computational model named timely dataflow that is based on a directed graph in which stateful vertices send and receive logically timestamped messages along directed edges. [33] The dataflow graph may contain nested cycles and the timestamps reflect this structure in order

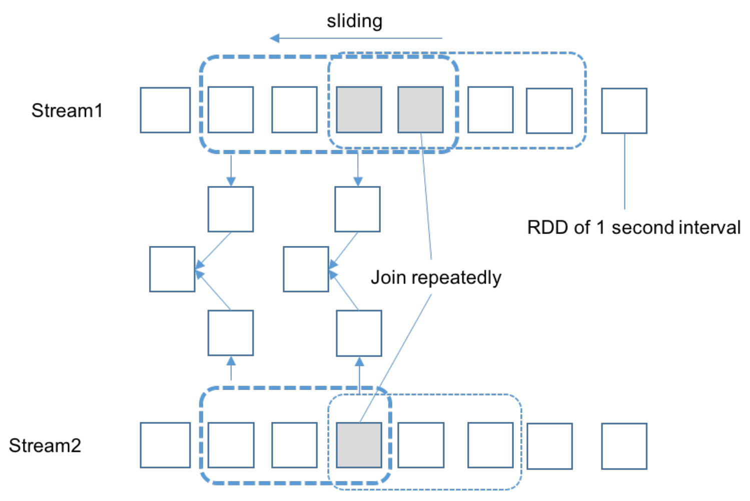

Figure 4.4: Spark Stream join with repeated tuples

to distinguish data that arise in different input epochs and loop iterations. With iterate operator, many stream processing systems already support such nested cycles in processing data flow graph. We designed a workload named StreamKMeans to evaluate iterate operator in stream processing systems.

KMeans is a clustering algorithm which aims to partition n points into k clusters in which each point belongs to the cluster with the nearest mean, serving as a prototype of the cluster.[3] Given an initial set of k means, the algorithm proceeds by alternating between two steps: [30]

Assignment step: assign each point to the cluster whose mean yields the least within-cluster sum of squares.

Update step: Calculate the new means to be the centroids of the points in the new clusters.

The algorithm has converged when the assignments no longer change. We apply k-means algorithm on a stream of points with an iterate operator to update centroids.

Compared to clustering for data set, the clustering problem for the data stream domain is difficult because of two issues that are hard to address: (1) The quality of the clusters is poor when the data evolves considerably over time. (2) A data stream clustering algorithm requires much greater functionality in discovering and exploring clusters over different portions of the stream.[4] Considering the main purpose of this workload is to evaluate iterative loop in stream data processing, we don't try to solve these issues

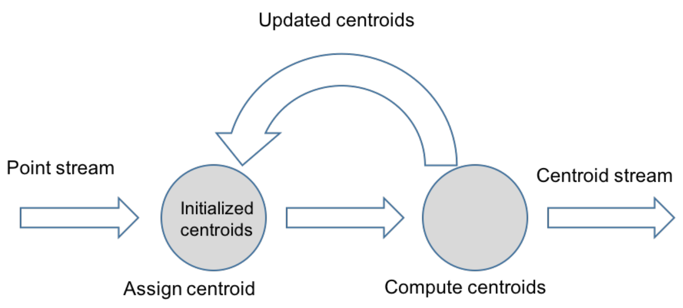

Figure 4.5: Stream k-means scenario

here. Similarly, stream k-means also has two steps: assignment and update. The difference is each point in the stream only passes the application once and the application doesn't try to buffer points. As shown in Figure 4.5, once a new centroid calculated, it will be broadcasted to assignment executors.

Spark executes data analysis pipeline using directed acyclic graph scheduler. Nested cycle doesn't exist in the data pipeline graph. Therefore, this workload will not be used to benchmark Spark Streaming. Instead, a standalone version of k-means application is used to evaluate the performance of Spark Streaming.

4.4 Data Generators

A data generator is a program that produces and sends unbound records continuously to kafka cluster which are consumed by corresponding workload. For each workload, we designed one or several data generators with some parameters configureable which define the skew in record popularity, the size of records etc. These parameters could be changed to evaluate the performance of a system executing one workload on similar data streams with different properties.

4.4.1 WordCount

Generators of workload WordCount produce unbound lists of sentences, each sentence consists of several words. The number of words in each sentence satisfies normal distribution with mean and variance configureable. Each word is a 5-digit zero-padded string of a binary integer, such as "00001". There are two generators implemented with these integers satisfy two different distribution: uniform distribution and normal distribution.

4.4.2 AdvClick

As discussed in Section 4.3.2, workload AdvClick joins stream `shown advertisements` and stream `advertisement clicks`. Each shown advertisement is a tuple consist of a universally unique identifier(**UUID**) and a timestamp. Each advertisement has a probability to be clicked. Then the data generator could be a multi-threads application with main thread producing advertisements and sub-threads producing clicks. The pseudocode of the main thread is shown as Algorithm 2. After a sub-thread starts, it sleeps for dalta time and then sends click record to corresponding kafka topic.

Algorithm 2 AdvClick data generator

```
 1: load clickProbability from configure file
 2: cachedThreadPool ← new CachedThreadPool
 3: dataGenerator ← new RandomDataGenerator
 4: producer ← new KafkaProducer
 5: while not interrupted do
 6:     advId ← new UUID
 7:     timestamp ← current timestamp
 8:     producer.send(...)
 9:     if generator.nextUniform(0,1) < clickProbability then
10:         deltaTime ← generator.nextGaussian(...)
11:         cachedPool.submit(new ClickThread(advId, daltaTime))
```

4.4.3 KMeans

Stream k-means is a one-pass clustering algorithm for stream data. In this workload, it is used to cluster a unbound stream of points. First, a set of centroids are generated and wrote to a external file. Then the generator produces points according these centroids as Algorithm 3.

4.5 Experiment Logging and Statistic

For evaluating the performance, there are two performance measurement terms used in StreamBench that are latency and throughput. Latency is the required time from a record entering the system to some results produced after some actions performed on the record. In StreamBench, messaging system and stream processing system are combined together and treated as one

Algorithm 3 KMeans data generator

```
1: load $covariances$ from configure file
2: $means \leftarrow$ original point
3: load $centroids$ from external file
4: $producer \leftarrow$ new KafkaProducer
5: $normalDistributon \leftarrow$ new NormalDistribution(means, converiances)
6: while not interrupted do
7:     $centroid \leftarrow$ pick a centroid from $centroids$ randomly
8:     $point \leftarrow centroid + normalDistributon.sample()$
9:     $producer.send(point)$
```

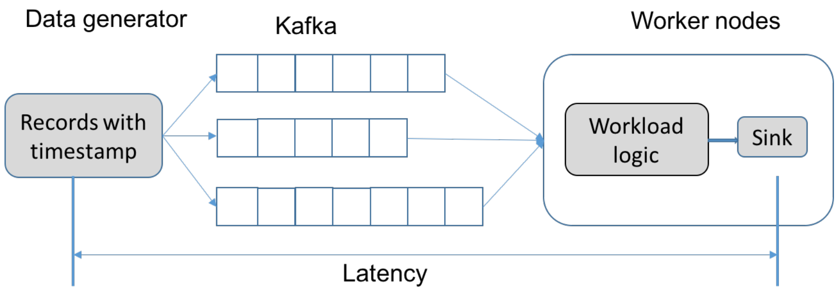

Figure 4.6: Latency

single system. The latency is computed start from when a record is generated. As discussed in Section 4.4, data is sent to kafka cluster immediately after generation. Figure 4.6 shows how latency computed in StreamBench.

Throughput is the number of actions executed or results produced per unit of time. In the WordCount workload, throughput is computed as the number of words counted per seconds in the whole compute cluster. Joined clicked stream and the number of points processed per second are the throughput of workloads AdvClick and Stream KMeans respectively.

There is an inherent tradeoff between latency and throughput: on a given hardware setup, as the amount of load increases by increase the speed of data generation, the latency of individual records increases as well since there is more contention for disk, CPU, network, and so on. Computing latency start from records generated makes it easy to measure the highest throughput, since records couldn't produced in time will stay in kafka topics that increase latency dramatically. A stream processing system with better performance will achieve low latency and high throughput with fewer servers.

4.6 Extensibility

One significant feature of StreamBench is extensibility. The component "Workloads" in Figure 4.1 contains three predefined workloads discussed in Section 4.3 that are implemented with common stream processing APIs. First, with some configuration modification of a data generator, which allows user to vary the skew in record popularity, and the size and number of records. The performances of a workload processing data streams with different properties could be different a lot. Moreover, it is easy for developers to design and implement a new workload to benchmark some specific features of stream processing systems. This approach allows for introducing more complex stream processing logic, and exploring tradeoffs of new stream processing features; but involves greater effort compared to the former approach.

Besides implementing new workloads, StreamBench also could be extended to benchmark new stream processing systems by implement a set of common stream processing APIs. A few samples of APIs could be shown as following:

- **map**(MapFunction<**T**, **R**>fun, String componentId): map each record in a stream from type **T** to type **R**
- **mapToPair**(MapPairFunction<**T, K, V**>fun, String componentId): map a item stream**<T>** to a pair stream**<K, V>**
- **reduceByKey**(ReduceFunction<**V**>fun, String componentId): called on a pair stream of (**K, V**) pairs, return a new pair stream of (**K, V**) pairs where the values for each key are aggregated using the given reduce function

These methods are quite simple, representing common data transformations. There are some other APIs like `filter()`, `flatMap()` and `join` which are also easily to implement and supported well by most stream processing systems. Despite its simplicity, this API maps well to the native APIs of many of the stream processing systems we examined.

Chapter 5

Experiment

The experiment environment consists of two clusters: compute cluster and Kafka cluster. Computer cluster consists of 8 work nodes and one master nodes. Kafka cluster has 5 brokers with one zookeeper instance running on the same machine with one Kafka broker. Each above node is a visual machine which has 4 CPU and 15 GB RAM. All the nodes are in the same local area network that provides low latency and high throughput connectivity.

5.1 Experiment Environment

5.1.1 Hardware

The experiment environment consists of two clusters: compute cluster and Kafka cluster. Computer cluster consists of 8 work nodes and one master nodes. Kafka cluster has 5 brokers with one zookeeper instance running on the same machine with one Kafka broker. Each above node is a visual machine which has 4 CPU and 15 GB RAM. All the nodes are in the same local area network that provides low latency and high throughput connectivity.

5.2 Classic Workload

5.2.1 WordCount

5.2.1.1 Offline WordCount

5.2.1.2 Short WordCount

5.2.1.3 Windowed WordCount

5.2.1.4 Online WordCount

5.2.2 Data Source

5.2.3 Results and Discussion

5.3 Multi-Streams Join Workload

5.3.1 Advertisements Click

5.3.2 Data Source

5.3.3 Results and Discussion

5.4 Iterate Workload

5.4.1 WordCount

5.4.2 Data Source

5.4.3 Results and Discussion

Chapter 6

Conclusions

Summary of experiment results

6.1 Selection in Practice

Summarize several factors which affect selection of stream processing systems in practic

6.1.1 Performance Summary

6.1.2 Issues

6.2 Future Work

Future works

Increase computation nodes, to test scaleability. Benchmark more stream processing systems and design more workloads.

6.2.1 Scale-out and Elasticity Evaluation

6.2.2 Evaluation of Other Platforms

Bibliography

[1] Giraph. URL http://giraph.apache.org/.

[2] Streams and operations on streams, Mar 2016. URL https://cwiki.apache.org/confluence/display/FLINK/Streams+and+Operations+on+Streams.

[3] K-means clustering, Mar 2016. URL https://en.wikipedia.org/wiki/K-means_clustering.

[4] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 81–92. VLDB Endowment, 2003.

[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465296. URL http://doi.acm.org/10.1145/2463676.2465296.

[6] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. 2015.

[7] F Chang, J Dean, S Ghemawat, WC Hsieh, DA Wallach, M Burrows, T Chandra, A Fikes, and R Gruber. Bigtable: A distributed structured data storage system. In *7th OSDI*, pages 305–314, 2006.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10,

pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[9] Patricio Córdova. Analysis of real time stream processing systems considering latency.

[10] Transaction Processing Performance Council. Tpc-c benchmark specification, . URL http://www.tpc.org/tpcc/.

[11] Transaction Processing Performance Council. Tpc-h benchmark specification, . URL http://www.tpc.org/tpch/.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Anamika Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. Ycsb+t: Benchmarking web-scale transactional databases. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 223–230. IEEE, 2014.

[14] L Doug. Data management: Controlling data volume, velocity, and variety, 2001.

[15] Tao Feng. Benchmarking apache samza: 1.2 million messages per second on a single node. URL https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node.

[16] Apache Software Foundation. Hdfs, . URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[17] Apache Software Foundation. Kafka, . URL http://kafka.apache.org/documentation.html.

[18] Apache Software Foundation. Mapreduce, . URL https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[19] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013.

ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2463712. URL `http://doi.acm.org/10.1145/2463676.2463712`.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[21] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[22] Jamie Grier. Extending the yahoo! streaming benchmark. URL `http://data-artisans.com/extending-the-yahoo-streaming-benchmark/`.

[23] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. pages 395–404, 2014.

[24] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. Benchmarking graph-processing platforms: a vision. pages 289–292, 2014.

[25] Mohammad Haghighat, Saman Zonouz, and Mohamed Abdel-Mottaleb. Cloudid: Trustworthy cloud-based and cross-enterprise biometric identification. *Expert Systems with Applications*, 42(21):7905–7916, 2015.

[26] Alexandru Iosup, AL Varbanescu, M Capota, T Hegeman, Y Guo, WL Ngai, and Merijn Verstraaten. Towards benchmarking iaas and paas clouds for graph analytics. 2014.

[27] Jaewoo Kang, Jeffery F Naughton, and Stratis D Viglas. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 341–352. IEEE, 2003.

[28] Robert Metzger Kostas Tzoumas, Stephan Ewen. High-throughput, low-latency, and exactly-once stream processing with apache flink. URL `http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/`.

[29] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proc. VLDB Endow.*, 7(12):1219–1230, August 2014. ISSN 2150-8097. doi: 10.14778/2732977.2732995. URL `http://dx.doi.org/10.14778/2732977.2732995`.

[30] David JC MacKay. *Information theory, inference and learning algorithms*, chapter 20. An Example Inference Task: Clustering, pages 284–292. Cambridge university press, 2003.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing - "abstract". pages 6–6, 2009. doi: 10.1145/1582716.1582723. URL `http://doi.acm.org/10.1145/1582716.1582723`.

[32] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[33] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[34] Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas. Of streams and storms. *IBM White Paper*, 2014.

[35] Paulo Neto. Demystifying cloud computing. In *Proceeding of Doctoral Symposium on Informatics Engineering*, 2011.

[36] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[37] Manoj P. Real time processing frameworks, . URL `http://www.ericsson.com/research-blog/data-knowledge/real-time-processing-frameworks/`.

[38] Manoj P. Apache-storm-vs spark-streaming, . URL `http://www.ericsson.com/research-blog/data-knowledge/apache-storm-vs-spark-streaming/`.

[39] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038925. URL `http://doi.acm.org/10.1145/2038916.2038925`.

[40] Alexander Pokluda and Wei Sun. Benchmarking failover characteristics of large-scale data storage applications: Cassandra and voldemort.

[41] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[42] Michael Stonebraker, U?ur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34 (4):42–47, 2005.

[43] Yahoo Storm Team. Yahoo streaming benchmark. URL `http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`.

[44] Kai Wähner. Real-time stream processing as game changer in a big data world with hadoop and data warehouse, 2014. URL `http://www.infoq.com/articles/stream-processing-hadoop`.

[45] Xinh. Xinh's tech blog. URL `http://xinhstechblog.blogspot.fi/2014/06/storm-vs-spark-streaming-side-by-side.html`.

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Appendix A

Source Code

.1 WordCount

.2 Advertisements Click