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

Yangjun Wang

Stream Processing Systems Benchmark: StreamBench

Master's Thesis
Espoo, May 15, 2016

DRAFT! — May 25, 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:	May 15, 2016	**Pages:**	59
Major:	Foundations of Advanced Computing	**Code:**	SCI3014
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 amount of data need to be collected and stored in a database or file system. That is very time-consuming. 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 streaming 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 stream processing systems before choosing one in production development is an 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, Flink, and Spark Streaming. 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 would like to express my gratitude to my supervisor Aristides Gionis for providing me this opportunity and introducing me to the topic. Furthermore I would like to thank my advisor Gianmarco De Francisci Morales for all his guidances and supports. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor for my master thesis.

Besides my advisors, I would like to thank all my friends in data mining group of Aalto university. Thanks for your advises and helps during my master study. It was a happy time to work with you and I have learnt a lot from you.

Last but not least, I want thank my parents for trying to support me with the best they can do all these years. A lot of thanks to all friends for their support and patience too.

Espoo, May 15, 2016

Yangjun Wang

Abbreviations and Acronyms

Abbreviations

DFS	Distribute File System
CPU	Central Processing Unit
HDFS	Hadoop Distribute File System
GFS	Google File System
LAN	Local Area Networks
YCSB	Yahoo Cloud Serving Benchmark
POSIX	The Portable Operating System Interface
DBMS	DataBase Management System
TPC	Transaction Processing Performance Council
AWS	Amazon Web Services
RDD	Resilient Distributed Dataset
DAG	Directed Acyclic Graph
YARN	Yet Another Resource Negotiator
API	Application Programming Interface
GB	Gigabyte
RAM	Random Access Memory
UUID	Universally Unique Identifier

Contents

Abbreviations and Acronyms **iv**

1 Introduction **1**

2 Background **3**

2.1 Cloud Computing 3

2.1.1 Parallel Computing 4

2.1.2 Computer Cluster 5

2.1.3 Batch Processing and Stream Processing 5

2.1.4 MapReduce 6

2.1.5 Hadoop Distribution File Systems 7

2.1.6 Kafka 7

2.2 Benchmark 9

2.2.1 Traditional Database Benchmarks 10

2.2.2 Cloud Service Benchmarks 10

2.2.3 Distributed Graph Benchmarks 11

2.2.4 Existing stream processing benchmarks 12

2.2.5 The Yahoo Streaming Benchmark 13

3 Stream Processing Platforms **16**

3.1 Apache Storm 17

3.1.1 Storm Architecture 18

3.1.2 Computational Model 19

3.2 Apache Flink 19

3.2.1 Flink Architecture 20

3.2.2 Computational Model 20

3.3 Apache Spark 21

3.3.1 Resilient Distributed Dataset(RDD) 22

3.3.2 Computational Model 23

3.3.3 Spark Streaming 24

3.4 Other Stream Processing Systems 24

3.4.1 Apache Samza 24
3.4.2 Apache S4 26

4 Benchmark Design **27**
4.1 Architecture 27
4.2 Experiment Environment Setup 29
4.3 Workloads 29
4.3.1 Basic Operators 30
4.3.2 Join Operator 31
4.3.3 Iterate Operator 34
4.4 Data Generators 35
4.4.1 WordCount 36
4.4.2 AdvClick 36
4.4.3 KMeans 37
4.5 Experiment Logging and Statistic 38
4.6 Extensibility 39

5 Experiment **41**
5.1 WordCount 41
5.1.1 Offline WordCount 42
5.1.2 Online WordCount 43
5.2 AdvClick 47
5.3 K-Means 49
5.4 Summary 52

6 Conclusion and Future Work **53**
6.1 Conclusion 53
6.2 Future Work 54

List of Figures

2.1 Kafka producer and consumer [17] 8
2.2 Kafka topic partitions 9
2.3 Operations flow of YSB [44] 13

3.1 Stream processing model 17
3.2 Storm cluster components 18
3.3 Flink computing model 21
3.4 Spark job stages[47] 23
3.5 Spark Streaming Model 24
3.6 Samza DataFlow Graph 25
3.7 Samza and Hadoop architecture 25

4.1 StreamBench architecture 28
4.2 Windowed WordCount 31
4.3 Window join scenario 32
4.4 Spark Stream join without repeated tuple 33
4.5 Spark Stream join with repeated tuples 34
4.6 Stream k-means scenario 35
4.7 Latency 39

5.1 Throughput of Offline WordCount (words/second) 42
5.2 Throughput Scale Comparison of Offline WordCount 43
5.3 Throughput of work nodes (words/s) 44
5.4 Latency of Online WordCount 45
5.5 AdvClick Performance 47
5.6 KMeans Latency of Flink and Storm 49
5.7 Convergences 50
5.8 Spark KMeans Latency 51

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 also has two other important 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 dealing 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 hand, stream processing is used for fast data requirements (**V**elocity + **V**ariety) [45]. 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 demand analysed results from streaming data in a very short latency. For example, an 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, it is possible to detect credit card fraud. Other cases like stock exchanges, sensor networks, and user behaviour online analysis also have this demand. 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.

Industry standard benchmark is a common and widely accepted way to evaluate a set of similar systems. Benchmarks enable rapid advancements in a field by having a standard reference for performance, and focus the attention of the research community on a common important task [40]. For example, TPC benchmarks such as TPC-C [10], TPC-H [11] have promoted the development of database systems. TPC-C simulates a complete computing environment where a population of users executes transactions against a database. TPC-H gives a standard evaluation of measuring the performance of highly-complex decision support databases.

How to evaluate real time stream processing systems and select one for a specific case wisely is an open question. Before these real time stream processing systems are implemented, Stonebraker et al. demonstrated the 8 requirements[43] 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, 46] 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 standard 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 not only enables us to implement new workloads to evaluate more aspects of stream processing systems, but also allows extending to 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. Chapter 5 presents and compares experiment results of three selected stream processing systems. 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 [45]. In order to process continuously incoming data in 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 demonstrated in § 2.1.5, or a 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. Therefore, stream processing 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 in the design and implementation of these benchmarks, and some of which 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

Many stream processing frameworks are run on the cloud such as Google Cloud Dataflow, Amazon Kinesis, and Microsoft Azure Stream Analytics. 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 include computing ability, scalable file system, 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 in 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, and 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 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 that 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 so that 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 output 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 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 include: 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 usually work together. In a distribute cluster, each data node 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 at providing a unified, high-throughput, low-latency platform for handling continuous data feeds.

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

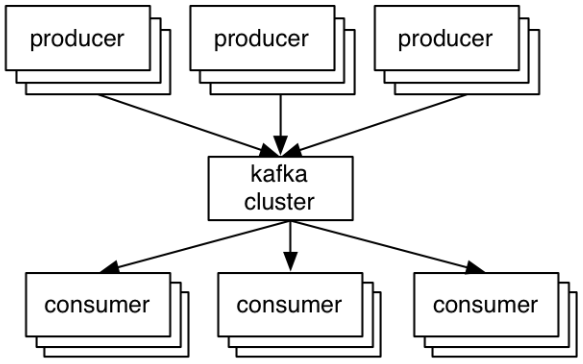


Figure 2.1: Kafka producer and consumer [17]

A stream processing system subscribing to Kafka will get notified within a very short time after a publisher published some data into a Kafka topic. In StreamBench, we use Kafka to provide messaging service. Before we go into architecture of Kafka, there are some basic messaging terminologies [17]:

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

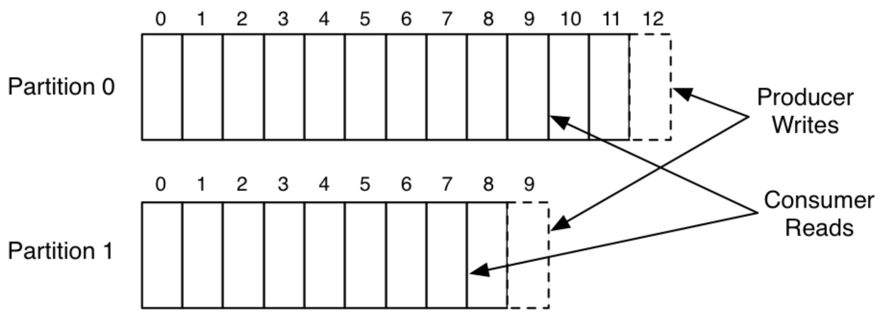


Figure 2.2: Kafka topic partitions

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.
- 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 systems become more and more complex and thus complicated, it becomes more and 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 are 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 [41] 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 provide an analysis of the failover characteristics of each. Kuhlenkamp et al. made some contributions to benchmarking scalability and elasticity of two popular cloud database systems HBase and Cassandra [29]. 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 at evaluating 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 and 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], Apache Giraph [1], and 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. demonstrate the diversity of graph-processing platforms and challenges to benchmark graph-processing platforms [24]. 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 [23]. 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. The processing model of both Spark Streaming and Storm Trident is micro-batch processing. One shortage of this work is the benchmark is not extensible. 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) that 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

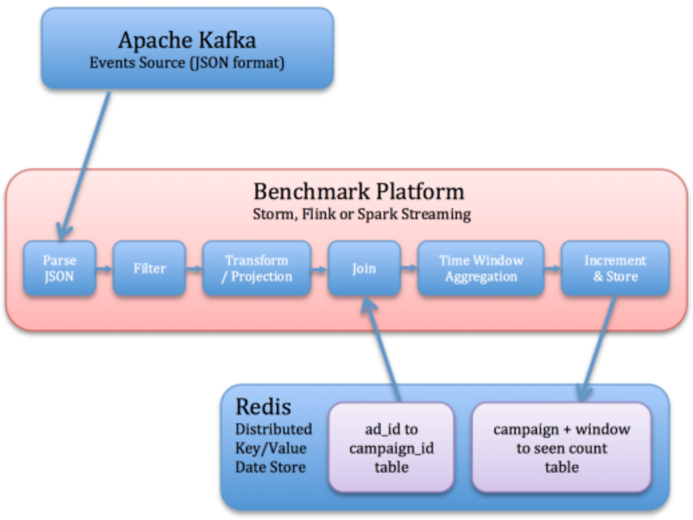

Apache Kafka
Events Source (JSON format)
Benchmark Platform
Storm, Flink or Spark Streaming
Parse JSON
Filter
Transform / Projection
Join
Time Window Aggregation
Increment & Store
Redis
Distributed Key/Value Date Store
ad_id to campaign_id table
campaign + window to seen count table

Figure 2.3: Operations flow of YSB [44]

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 in her blog compared Storm and Spark Streaming side-by-side, including processing model, latency, fault tolerance and data guarantees [46]. 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 [44]. 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

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

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 re-implementing 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.

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. More new stream processing systems are being developed. The design, computational model, and architecture of these systems are different so that they own different features and have different performance. Among these systems, some are not open sourced and it is impossible to 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. That enables us to set up our own distributed 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, they consider the stream as keyed tuples, 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. Unlike MapReduce, these systems deal with unbounded 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. Integrated with distributed message system, these systems achieve both data parallelism and task parallelism of parallel computing. The lower part of Figure 3.1 shows that 3 compute nodes in a cluster consume messages from a Kafka topic with 3 partitions and run operators `mapToPair` and `reduce` in parallel. Each record in the stream is a tuple and there are different ways to forward tuples from a operator to another one, such as key grouping, all grouping, and shuffle grouping. The computational topology of these systems is a graph shown

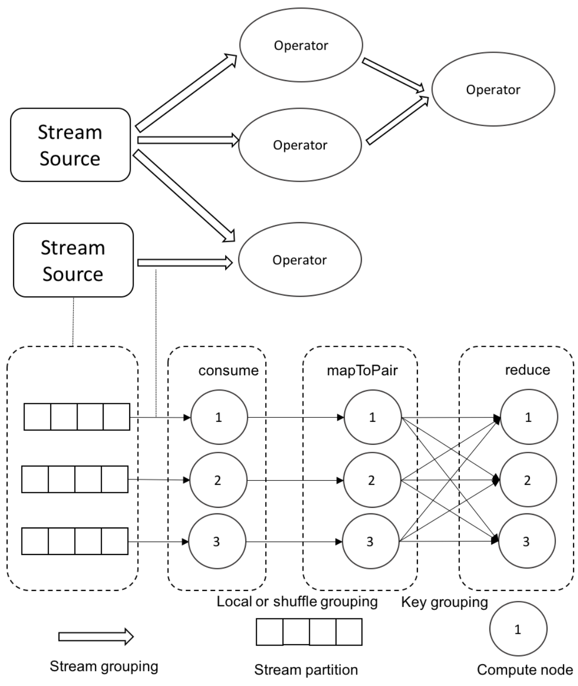


Figure 3.1: Stream processing model

as the upper part of Figure 3.1 in which nodes are stream sources and operators connected by stream grouping. 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, computational model, and key features. Several other stream processing systems are also discussed.

3.1 Apache Storm

Apache Storm, one of the oldest distributed stream processing systems 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 unbounded 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

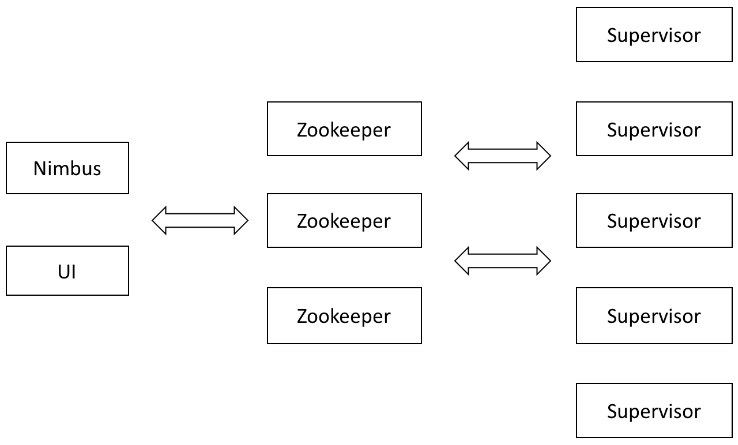


Figure 3.2: Storm cluster components

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 3.2 shows components of a storm cluster which contains four different kinds of nodes, "Supervisor", "Nimbus", "Zookeeper" and "UI".

Nimbus is a daemon running 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 or bolt also could be found in "UI". More detail about understanding the Storm UI could be found on the page [1].

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

3.1.2 Computational Model

The core of Storm data processing is a computational topology, which is a graph of stream transformations similar to the one in Figure 3.1. In the computational topology, stream source is consumed by a type of operator called spout. The operator with processing logic is called bolt. Spouts 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;
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

storm-ui-explained/

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 distributed cloud systems. The system consists of a JobManager and one or more TaskManagers. JobManager is the coordinator of the Flink system, while TaskManagers are the workers that execute parts of the parallel programs.

In Hadoop MapReduce, reduce operation wouldn't start until map 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 stream and batch processing 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 could be described as a graph consists of `transformaitons` and `sources`, which is similar to Storm. Edges in the graph indicate state changes 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.

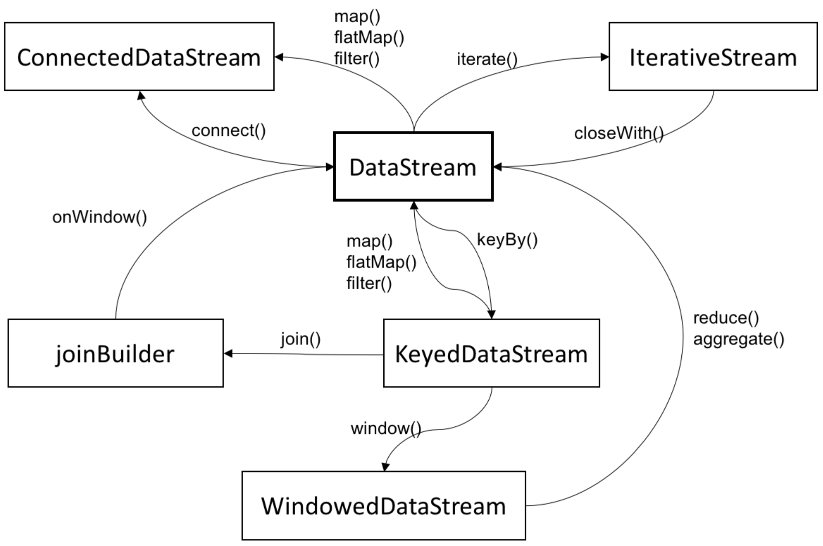


Figure 3.3: Flink computing model

- 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`) operations 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 could be kept 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 [47]. 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 is 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 avoided 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.

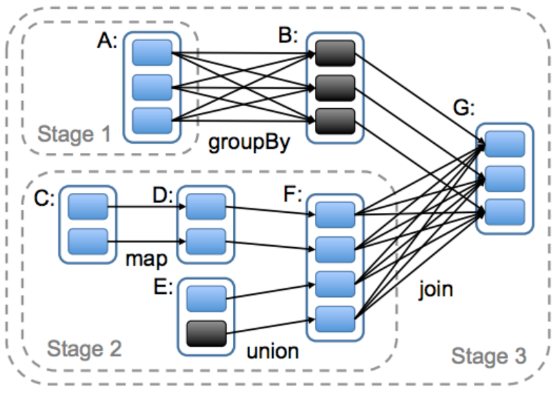


Figure 3.4: Spark job stages[47]

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, when a reduce operation is performed on a dataset created through transformation map, only the result of the reduce is returned to the driver program, rather than the larger mapped dataset.

Spark has an advanced directed acyclic graph (DAG) execution engine that implements stage-oriented scheduling. The are two main differences between the computational model of Spark Streaming and other two systems. First, the computing unit in Spark Streaming is a dataset. While in Storm and Flink, it is one single record tuple. The other difference is each node in DAG is a stage which contains as many pipelined transformations with narrow dependencies as possible. Whenever a user runs an action (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. The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can short-circuit the computation of a parent RDD [47]. 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 [42]. In the DAG, tasks of a stage wouldn't start until all its dependencies are done.

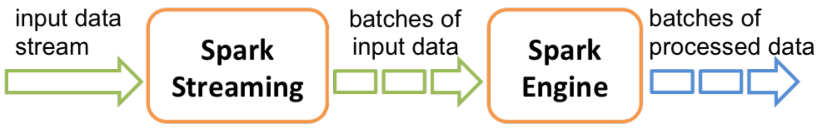


Figure 3.5: Spark Streaming Model

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 micro batches, which are then processed by the Spark engine to generate the final stream of results in batches. Each batch is processed in a Spark batch processing job. 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 configurable interval. 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. 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 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

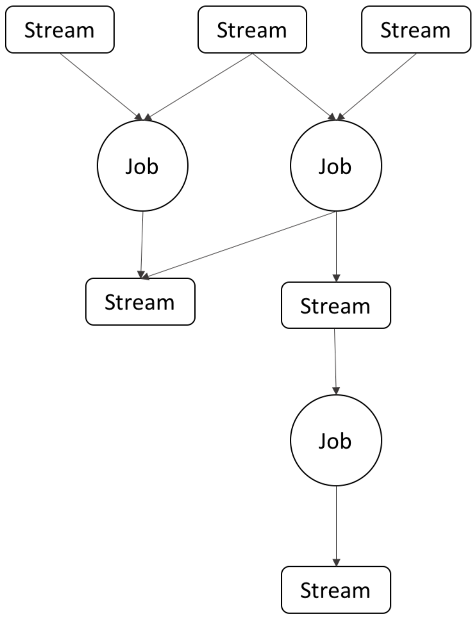


Figure 3.6: Samza DataFlow Graph

the throughput of the stream processor, streams are broken into partitions and jobs are divided 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.

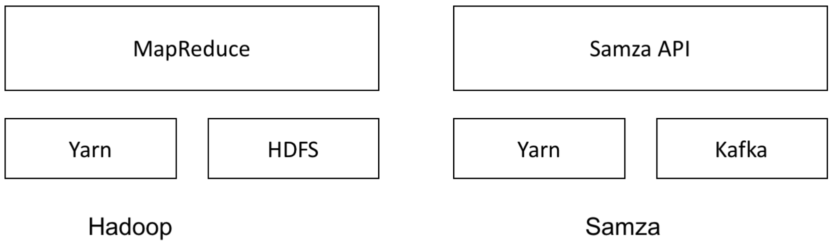


Figure 3.7: Samza and Hadoop architecture

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. 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. We have used StreamBench to measure the performance of three selected stream processing systems, and that is reported in the next chapter.

This chapter illustrates 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 executing workloads on stream processing cluster. The architecture of StreamBench is shown in Figure 4.1. The core of the architecture is StreamBench API which contains several states of stream and a set of stream processing APIs that are very similar to Flink's computational model. For example, API `mapToPair` maps a normal data stream to a keyed data stream, and API `filter` is a method with a parameter of boolean function and evaluates this boolean function for each element and retains those for which the function returns true.

StreamBench API could be engined by different stream processing systems. Currently we support these APIs on three stream processing systems: Storm, Flink and Spark Streaming. It is very convenient to implement most interfaces of StreamBench API on Flink and Spark Streaming which have similar high level APIs. But there are also some APIs that Flink

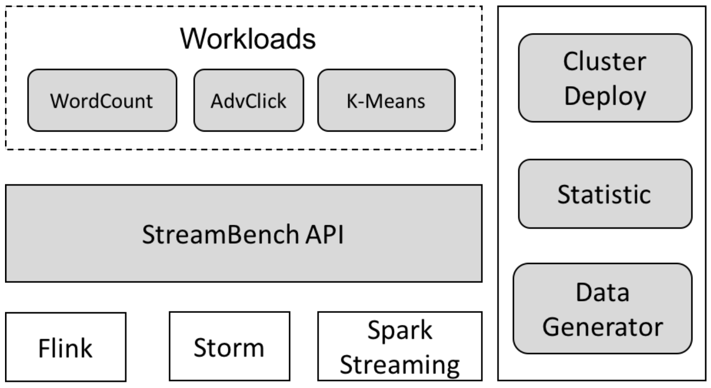


Figure 4.1: StreamBench architecture

and/or Spark Streaming doesn't support well. For example, currently, Flink `join` operator only supports two streams joining on the same size window time. Therefore, we implemented another version `join` operator discussed in § 4.3.2 with Flink's low level API. Compare to Flink and Spark Streaming, Storm is more flexible by providing two low level APIs: spout and bolt. Bolts represent the processing logic unit in Storm. One can utilize bolts to do any kind of processing such as filtering, aggregating, joining, interacting with data stores, and talking to external systems.

With these common stream processing APIs, we implemented three workloads to benchmark performance of stream processing systems in different aspects. WordCount discussed in § 4.3.1 aims to evaluate the performance of stream processing systems performing basic operators. In § 4.3.2, we demonstrated a workload named AdvClick to benchmark two keyed streams joining operation. To check the performance of iterate operator, we designed a workload to calculate k-means of a point stream. More detail of this workload could be found in § 4.3.3.

Besides the core Java program, the architecture also includes three more components: Cluster Deploy, Data Generator and Statistic. Section 4.2 illustrates 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 § 4.4. The Statistic component discussed in § 4.5 includes experiment logging and performance statistic.

4.2 Experiment Environment Setup

The experiment environment is a cloud service called cPouta which is the main production IaaS cloud at CSC – a non-profit, state-owned company administered by the Ministry of Education and Culture. In cPouta, there are several available virtual machine flavors. Each visual machine used in our experiment has 4 CPU cores, 15GB RAM, 10GB root disk and 220GB ephemeral disk. 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. The cPouta service is based on the hardware of the Taito cluster. Communication among nodes and to the storage is done by Infiniband FDR fabric, which provides low latency and high throughput connectivity. The detail information about hardware and inter connection could be found online [1].

The operating 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 [2].

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 cluster and executes operations or transformations on the messages. We have developed 3 workloads to evaluate different aspects of a stream processing system. Each workload contains a representative operation or feature of

[1] `https://research.csc.fi/taito-supercluster#1.1.2`
[2] `https://github.com/wangyangjun/StreamBench`

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 § 4.6.

4.3.1 Basic Operators

With the widespread use of computer technologies, there is an 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`, and `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. The pseudocode of WordCount implemented with StreamBench 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.

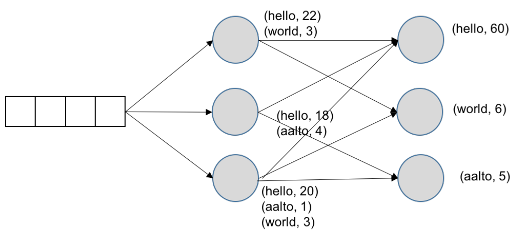


Figure 4.2: Windowed WordCount

When dealing with skewed data, the compute node which count the word with largest frequency might be the bottleneck. Inspired from MapReduce Combiner, we designed another version of WordCount with `window` operator of stream processing. Windows are typically groups of events within a certain time period. In the reduce step of Windowed WordCount, first words are shuffle grouped and applied pre-aggregation. In a certain time period, local pre-aggregation results are stored at local compute nodes. At the end of a window time, the intermedia word counts are key grouped and reduced to get the final results.

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 two input 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, say stream *S1* and stream *S2*, along with windows with the same slide size for both *S1* and *S2* as input. Each record in *S1* is a tuple of pair `(k, v1)` with `k` as the primary key. The key in Stream *S2:*`(k, v2)` is a foreign key referencing primary key in *S1*. The output of join operator is a stream of tuple `(k, v1, v2)`. This primary key join operation could be described as a SQL query illustrated in Algorithm 2. Assuming a sliding window join between stream *S1* and stream *S2*, a new tuple arrival from stream *S2*, then a summary of steps to preform join is the following:

1. Scan window of stream *S1* to find any tuple which has the same key

Algorithm 2 Join Query

```
1: SELECT $S1.k$, $S1.v1$, $S2.v2$
2: FROM $S1$
3: INNER JOIN $S2$
4: ON $S1.k = S2.k$
```

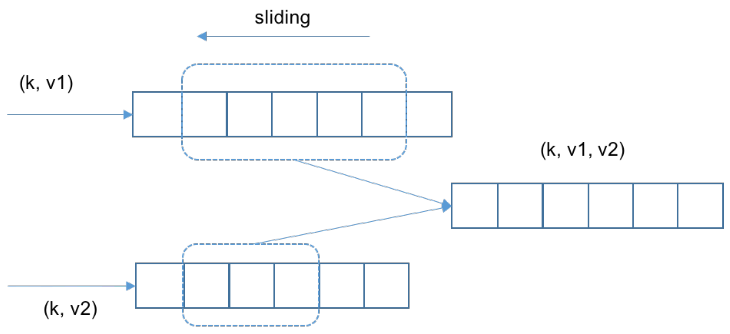


Figure 4.3: Window join scenario

with this new tuple and propagate the result;

2. (a) Invalidate target tuple in stream *S1*'s window if found ;
   (b) If not, insert the new tuple into stream *S2*'s window
3. Invalidate all expired tuples in stream *S2*'s window.

Every time new tuple arrives stream *S2*, window of stream *S1* need be scanned. That reduces the performance of join operation, especially when the window is big. 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 `cachedHashTable` could be found in Guava.[3] 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

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

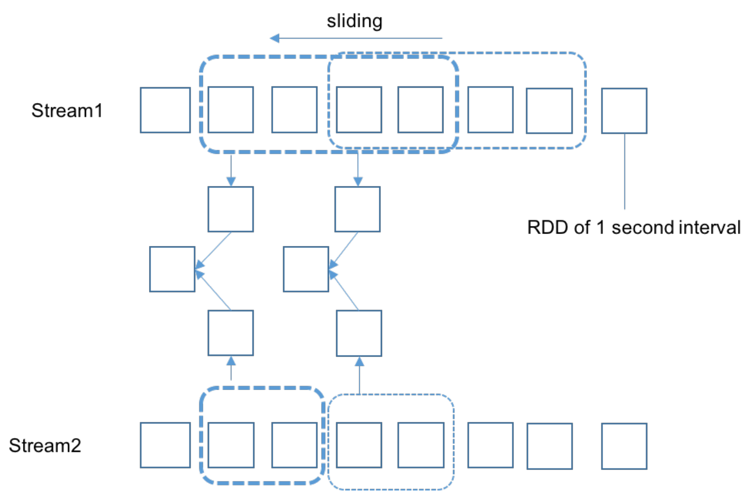


Figure 4.4: Spark Stream join without repeated tuple

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 4.4 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 4.5 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 advertisements is generated in the system. Each 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 unbounded tuples of `(id, clicked time)`. Normally, if an advertisement is attractive to a user, the user would click it in a short time after it is shown. We call such a click of an attractive advertisement valid click. To bill a customer, we need count all valid clicks regularly for

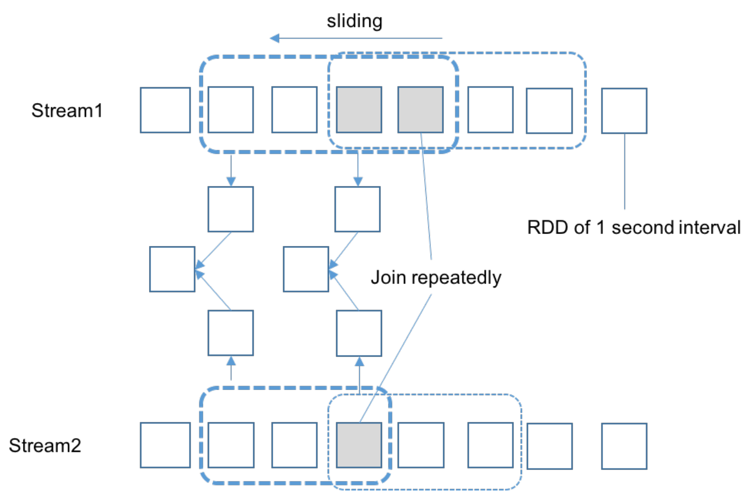


Figure 4.5: Spark Stream join with repeated tuples

advertisements of this customer. That could be counted after joining stream `advertisement clicks` and stream `shown advertisements`.

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. These require a data processing system having 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 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 stream k-means to evaluate iterate operator in stream processing systems.

K-means 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

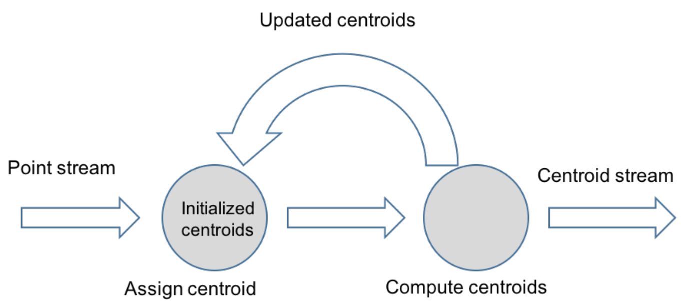


Figure 4.6: Stream k-means scenario

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 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 4.6, 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 unbounded records continuously to kafka cluster which are consumed by corresponding workload.

For each workload, we designed one or several data generators with some configureable parameters which define the skew in record popularity, the size of records, and the distribution of data etc. These parameters could be changed to evaluate the performance of a system executing one workload on similar data streams with different properties. Users of StreamBench also could implement their own data generators to produce benchmarking data. Data generators are defined in a submodule of StreamBench project named *generator*. The submodule could be compiled and packaged to get a jar file that could run on any node of our kafka cluster.

4.4.1 WordCount

A data generator of workload WordCount produces unbounded lists of sentences, each sentence consists of several words. In StreamBench, we have implemented two versions of WordCount data generator. Each word in both generators is a 5-digit zero-padded string of a binary integer, such as “00001”. The number of words in each sentence satisfies normal distribution with mean and sigma configured as `(10, 1)`. The difference between these two generators is the corresponding integers satisfy two different distributions: uniform distribution and zipfian distribution. Both uniform distribution and zipfian distribution have the same size – 10000. Exponent of zipfian distribution is configured as 1.

There are two different ways to run WordCount data generators to cooperate experiments of WordCount discussed in § 5.1. For Online WordCount, we start data generation after benchmark program and pass a parameter when starting generator to control the generation speed. For Offline model, we could either preload data to kafka cluster or make the generation speed much faster than the throughput of stream processing system executing WordCount workload.

4.4.2 AdvClick

As discussed in Section 4.3.2, workload AdvClick performs join operator on two streams: `shown advertisements` and `advertisement clicks`. Each record in `shown advertisements` is a tuple consist of a universally unique identifier(**UUID**) and a timestamp. Each advertisement has a probability to be clicked, which is set to 0.3 in our experiments. Then the data generator could be a multi-threads application with main thread producing advertisements and sub-threads generating clicks. The pseudocode of the main thread is shown as Algorithm 3. After a sub-thread starts, it sleeps for delta time and then sends click record to corresponding kafka topic. The probability of advertisements click is a configureable parameter. In our experiments, mean of click delay is set to 10 seconds.

Algorithm 3 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 unbounded stream of points. The data generator produces such a stream of points. In order to make experiment results checking easy, we use pre-defined centroids. First, a set of centers are generated and written to a external file. There is a minimum distance between every two centers so that no two clusters are overlapped together. Then the generator produces points according these centers as Algorithm 4. The distance of each point to corresponding center satisfies normal distribution with mean and variance as configurable parameters. In our experiments, we found that random initial centroids would lead to results that two groups points cluster to a centroid which is in the middle of two real centers. Which is not desired output to measure the speed of convergence. Therefore, we generate initial centroids for each cluster in the same way as points generation.

In our experiment environment, the compute cluster consists of 8 work nodes, each work node has 4 cpu cores. The parallelism of the compute cluster is 32. In order to have a better workload balance, we set the number of centers as 96. The dimension of point is configurable that enables to evaluate whether computation is a bottleneck of this workload.

Algorithm 4 KMeans data generator

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

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 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 4.7 shows how latency computed in StreamBench. In our experiments, we noticed that in the beginning of processing data, the performance of Storm cluster is bad. That leads to high latency of records in the head of a stream. Therefore, we ignored latency logs of first 1 minute in our statistic.

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 click events 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.

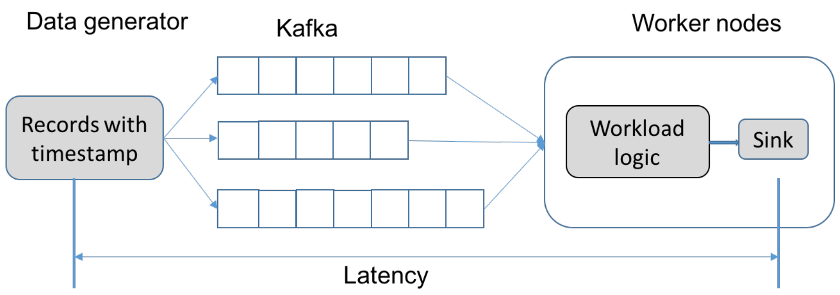


Figure 4.7: Latency

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.

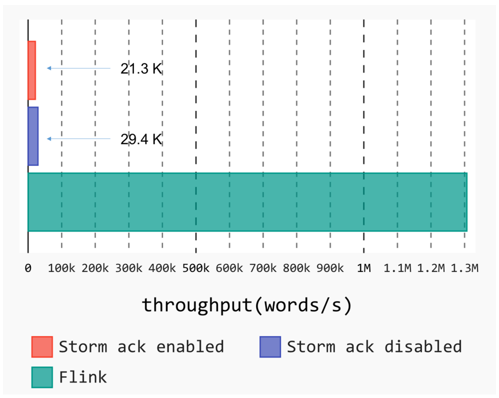
Chapter 5

Experiment

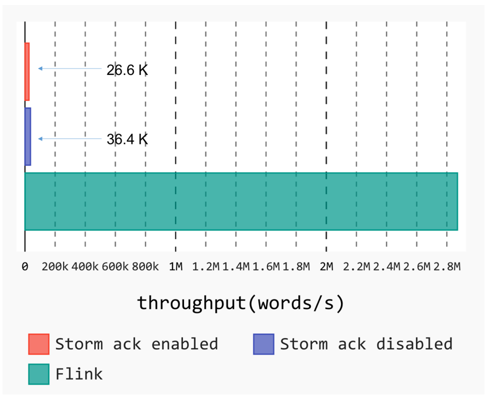
After StreamBench architecture and design of workloads are demonstrated, this chapter will draw our attention to experiments. Each experiment case is executed several times to get a stable result. In this chapter, we present experiment results of three selected stream processing systems running the workloads that are discussed in § 4.3. As illustrated in § 4.5, two performance metrics that we are concerned with latency and throughput. For each workload, we compare the experiment results of different stream processing systems with visualization of these two metrics.

5.1 WordCount

First, we examine workload WordCount which aims to evaluate performance of stream processing systems performing basic operators. In order to check different performance metrics of stream processing systems, we performed the WordCount experiments in two different models: Offline model and Online model. Offline WordCount focuses on throughput and aims to find maximum throughput of this workload performed on each system. Offline means that the workload application consumes data that already exists in Kafka cluster. On the contrary, experiments of consuming continuous coming data is called Online model, which measures latency of stream processing with different throughputs. Moreover, we also made some modification to the original workload to evaluate pre-aggregation property of Storm. As mentioned in § 4.4.1, for this workload, we designed two data generators to produce words that satisfy uniform and zipfian distributions. Comparison between experiment results of processing these two different data streams is also presented.



(a) Skewed WordCount



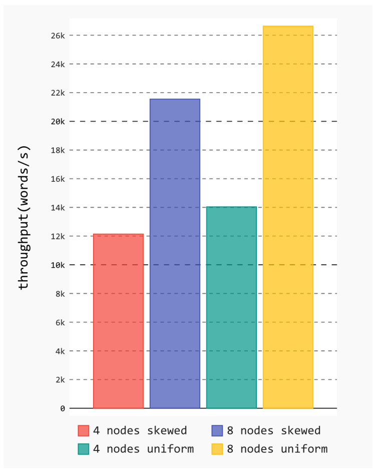
(b) Uniform WordCount

Figure 5.1: Throughput of Offline WordCount (words/second)

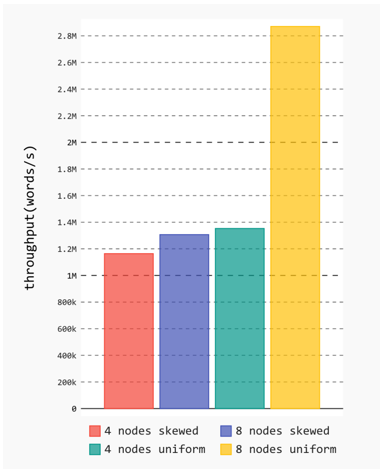
5.1.1 Offline WordCount

Since the computing model of Spark Streaming is micro-batch processing, existing data in Kafka cluster would be collected and processed as one single batch. The performance of processing one large batch with Spark Streaming is similar to a Spark batch job. There are already many works evaluating performance of Spark batch processing. Therefore, we skip experiments of Spark Streaming here. Figure 5.1 presents throughputs of Offline WordCount processing both skewed and uniform data on Storm and Flink clusters. It is obvious that the throughput of Flink is incomparably larger than Storm, tens of times higher. The throughput of Flink cluster dealing with uniform data stream is very high and reaches 2.83 million words per second, which is more than two times as large as throughput of performing skewed data. The corresponding ratio of Storm is around 1.25. The skewness of experiment data has greater influence of performance on Flink than Storm.

The difference of performance between skewed data and uniform data indicates that the bottleneck of a cluster processing skewed data would be the node dealing with the data with highest frequency. To verify this assumption, we reduce the number of computing nodes from 8 to 4, and run these experiments. The experiment results are presented as Figure 5.2. The throughput of 8-nodes cluster of both systems dealing with uniform data is nearly two times as large as that of 4-nodes cluster. It means that the scalability of both systems is good. While processing skewed data, increasing the number of work nodes in a Flink cluster doesn't bring significant performance increase. Storm cluster gets about 58% throughput improvement



(a) Storm (ack enabled)



(b) Flink

Figure 5.2: Throughput Scale Comparison of Offline WordCount

when increasing cluster from 4 nodes to 8 nodes. The result indicates that the assumption is correct in Flink, and the bottleneck of a storm cluster might be other factors.

The throughput of each work node in computing cluster is displayed in Figure 5.3. Obviously, for both Storm and Flink, workloads processing uniform data achieve better balance than corresponding workloads dealing with skewed data. Either dealing with uniform data or skewed data, Storm achieves better workload balance than Flink. The experiment results also shows that Flink cluster with 4 compute nodes has better workload balance than clusters with 8 nodes.

5.1.2 Online WordCount

Base on the experiment results of Offline WordCount, we perform experiments of Online WordCount on Storm and Flink at around half of the maximum achieved throughput of Offline WordCount respectively. In Online scenario, the stream processing application starts earlier than data generation. Which means data is processed as soon as possible after it is generated. As mentioned in § 4.5, the latency is computed as spending time from a record generated to corresponding result computed.

In Spark Streaming, depending on the nature of the streaming computation, the batch interval used may have significant impact on the data rates

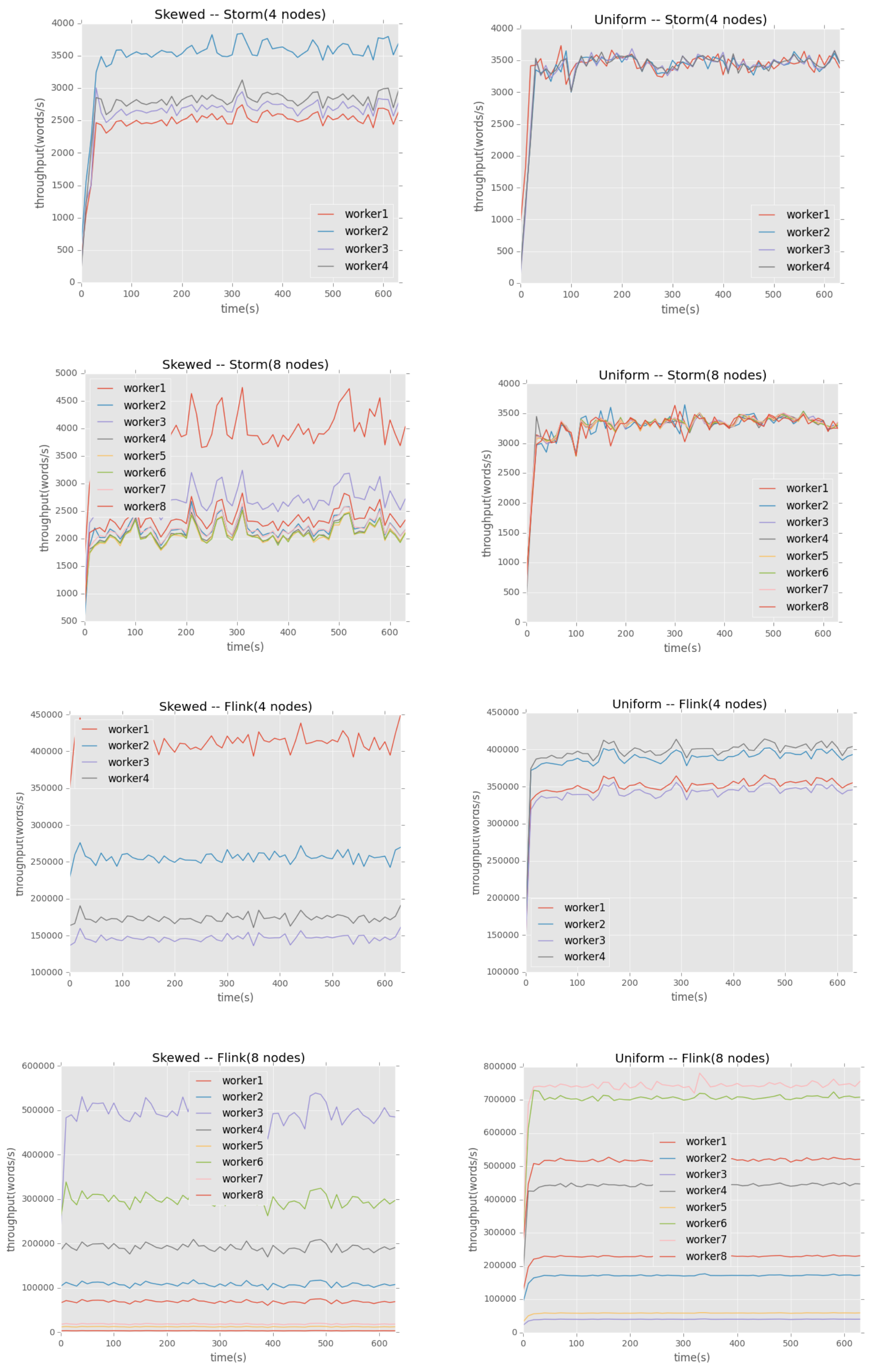
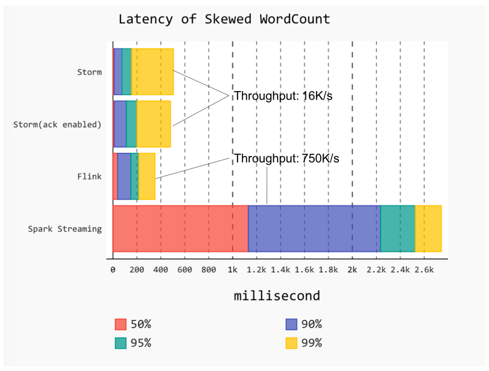
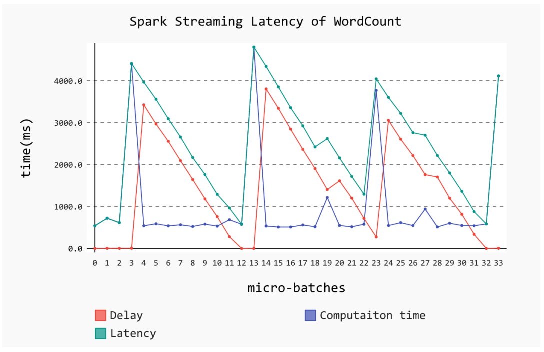


Figure 5.3: Throughput of work nodes (words/s)



(a) Records latency



(b) Micro-batches latency

Figure 5.4: Latency of Online WordCount

that can be sustained by the application on a fixed set of cluster resources[1]. Here, we perform the experiments with one second micro-batch interval and 10 seconds checkpoint interval which are the default configurations. Checkpointing is enabled because of a stateful transformation, `updateStateByKey` is used here to accumulate word counts. Checkpointing is very time consuming due to writing information to a fault- tolerant storage system. Figure 5.4(b) shows that the latency of micro-batches increasing and decreasing periodically because of checkpointing. A micro-batch is collected during one micro-batch interval, early records are buffered before the last record in the micro-batch arrives. In figure 5.4(b), buffer time of records in a micro-batch is not took in consideration. Before the computation of a micro-batch is finished, computation job of following micro-batches will not start. Therefore, the start time of computation job of a micro-batch would be delayed, this is indicated by "Delay" in the figure. The throughput of experiment corresponding to Figure 5.4(b) is 1.4M/s (million words per second) of skewed data. When the speed of data generation reaches 1.8M/s, the delay and latency increase infinitely with periodic decreasing.

Figure 5.4(a) shows the latency of Online WordCount performing skewed data. Storm with ack enabled achieves a median latency of 10 milliseconds, and a 95-th percentile latency of 201 milliseconds, meaning that 95% of all latencies were below 201 milliseconds. Flink has a higher median latency (39 milliseconds), and a similar 95-th percentile latency of 217 milliseconds. Since the records in a micro-batch are buffered up to batch interval time, the buffer time are also counted into the latency according to our latency

[1]`http://spark.apache.org/docs/1.5.1/streaming-programming-guide.html#setting-the-right-batch-interval`

computational method present in Figure 4.7. For example, median latency of Spark Streaming is equal to the sum of median latency of micro-batches and half of micro-batch interval. Obviously, the latency of Spark Streaming is much higher than that of others.

As mentioned in § 4.3.1, we designed another version of WordCount named Windowed Wordcount. Actually, Spark Streaming supports pre-aggregation by default, therefore, above Spark Streaming WordCount experiments already own this feature. Currently, Flink-0.10.1 doesn't support pre-aggregation, and parallel window can only be applied on keyed stream. It is possible to implement Windowed WordCount with Flink's low level API. But it is too time consuming and we leave it to future works. Therefore, only Storm is benchmarked with this workload.

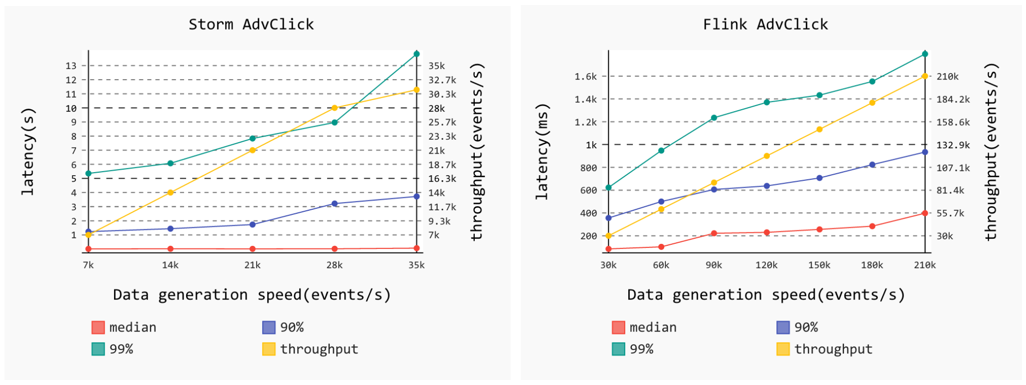
To support Windowed WordCount, we implemented a `window` operator in Storm[2] with ack disabled. Our experiments show that the window time has very limited effect on throughput. Here only experiment results of one second window workload are presented. The throughput of Windowed WordCount performing skewed data in Offline model could reach 60K/s (thousand words per second) that is more than two times as large as experiments without window. While dealing with uniform data, the throughput doesn't have any obvious improvement. Pre-aggregation on a window only helps in case of skewed data because it compacts the data thus removing the skew. Online model with a generation speed of 50K/s achieves a median latency of 1431 milliseconds, and a 99-th percentile latency of 3877 milliseconds.

Throughput of WordCount workload is summarized as Table 5.1. Obviously, Flink and Spark Streaming achieve incomparably higher throughput than Storm. The skewness of data has a dramatic effect on WordCount applications without pre-aggregation.

	No Pre-aggregation		Windowed Pre-aggregation	
	Uniform	Zipfian	Uniform	Zipfian
Storm (ack enabled)	26.6K/s	21.3K/s	∅	∅
Storm (ack disabled)	36.4K/s	29.4K/s	50K/s	60K/s
Spark Streaming	∅	∅	1.3M/s	1.4M/s
Flink	2.8M/s	1.4M/s	∅	∅

Table 5.1: WordCount Throughput

[2]`https://github.com/wangyangjun/Storm-window`



(a) Storm (b) Flink

Figure 5.5: AdvClick Performance

5.2 AdvClick

As described in § 4.4.2, click delays of clicked advertisements satisfy normal distribution and the mean is set to 10 seconds. In our experiments, we define that clicks within 20s after corresponding advertisement shown are valid clicks. In theory, overwhelming majority records in the click stream could be joined. Kafka only provides a total order over messages within a partition, not between different partitions in a topic [17]. Therefore, it is possible that click record arrives earlier than corresponding advertisement shown record. We set a window time of 5 seconds for `advertisement clicks` stream, as acking a tuple would require knowing whether it will be joined with a corresponding one from the other stream in the future.

When benchmarking Storm and Flink, first we perform experiments with low speed data generation, and then increase the speed until obvious joining failures occur when throughput is much less than generation speed of stream `advertisement clicks`. The experiment result shows that the maximum throughput of Storm cluster is around 8.4K/s (joined events per second). The corresponding generation speed of `shown advertisements` is 28K/s. As we can see in Figure 5.5(a), cluster throughput of `shown advertisements` is equal to the data generation speed when it is less than 28K/s. That means there is no join failures. Figure 5.5(a) also shows that Storm cluster has a very low median latency. But the 99-th percentile of latency is much higher and increase dramatically with the data generation speed.

Compared to Storm, Flink achieves a much better throughput. In our experiments, the throughput of Flink cluster is always equal to the generation speed of stream `shown advertisements`. But when the generation speed of stream `shown advertisements` is larger than 200K/s, the Flink AdvClick processing job is usually failed because of a bug in flink-connector-kafka [3]. This issue is fixed in the latest versions of Flink and Kafka. But Storm and Spark don't support the latest version of Kafka yet. We will upgrade all these systems in StreamBench in the next version of StreamBench. The maximum throughput of Flink we achieved in experiments is 63K/s (joined events per second), around 6 times larger than Storm. The latency of Flink performing this workload is shown as Figure 5.5(b). Even though the median latencies are a little higher than Storm, but 90-th and 99-th percentiles of Flink latency are much lower.

As discussed in § 4.3.2, Spark Streaming join operator is applied with sliding window. With the configuration of 20s/5s, the slide intervals of both windows are 5 seconds. That means a micro-batch join job is submitted to Spark Streaming cluster every 5 seconds. Because of different processing model, there is no joining failure in Spark Streaming. But high data generation speed leads to increasing delay of micro-batch jobs, because micro-batch jobs couldn't be finished in interval time. With this configuration, Spark Streaming has a very small throughput which is lower than 2K/s. Increasing micro-batch jobs submitting interval might increase the throughput, but leads to higher latency. For this workload, increasing the window lengths also because of the presence of duplicate records, as the windows overlap. Therefore, we did some experiments with larger windows. Increasing windows length of these two streams to 60s/30s, the cluster could achieve a throughput of 20K/s which is ten times larger.

	Maximum	Latency		
	Throughput	Throughout	Median	90%
Storm (ack disabled)	8.4K/s	4.2K/s	14ms	2116ms
Flink	63K/s	33K/s	230ms	637ms
Spark Streaming (20s/5s)	< 2K/s	Ø	Ø	Ø
Spark Streaming (60s/30s)	20K/s	20K/s	∼20s	∼24s

Table 5.2: Advertisement Click Performance

Table 5.2 summarizes maximum throughputs and latencies at a specific throughput of these systems. Flink achieves the largest throughput and

[3] `https://issues.apache.org/jira/browse/KAFKA-725`

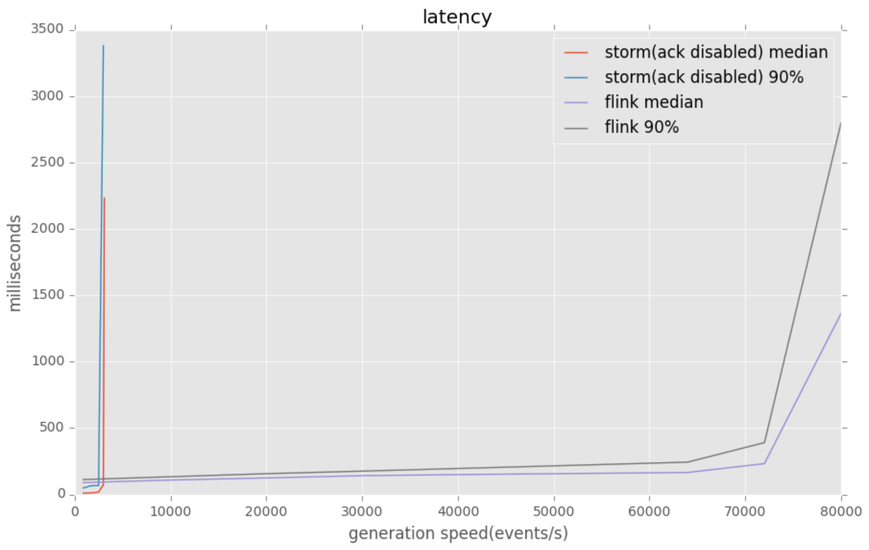
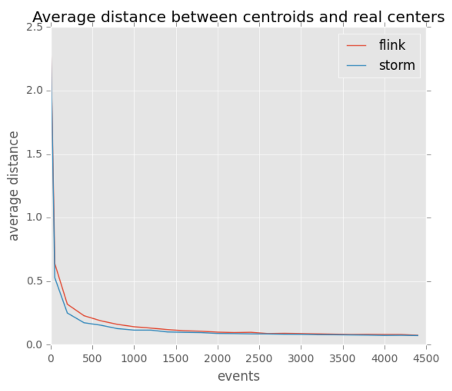


Figure 5.6: KMeans Latency of Flink and Storm

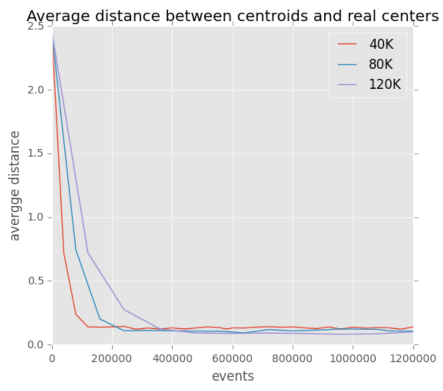
lowest 90-th percentile latency. While the median latency of Storm is 14ms, that is much lower than other systems. Latencies of Spark Streaming shown in the table is the latencies of micro-batches that doesn't include buffer time of records in a window.

5.3 K-Means

Experiment results of stream k-means processing 2-dimensional points shows that Storm cluster with at-least-once processing guarantee has a throughput of 1.7K/s. Without this guarantee, the throughput is a litter higher, around 2.7K/s. The maximum throughput of Flink cluster is much larger and reaches 78K/s. Figure 5.6 shows the latencies of Flink cluster and Storm Cluster without at-least-once guarantee. When the generation speed of point stream is low, Storm achieves very low median latency. The 90-th percentile of latency of Storm is also lower than Flink. The latency of Storm rises sharply when the generation speed is around 2.7K/s to 3K/s. Compared with Storm, latency percentiles of Flink is more compact. When the speed of data generations is 30K/s, Flink achieves a median latency of 141 milliseconds, and a 90-th percentile latency of 195 milliseconds. From this figure, it is easy to know that the throughput of Flink is significantly larger than storm.



(a) Flink and Storm



(b) Spark Streaming

Figure 5.7: Convergences

Since the throughput of Flink is tens of times higher than Storm, this workload converges much quicker on Flink cluster. In order to compare convergences of the algorithm running on Storm and Flink clusters, we calculated average distance between centroids and corresponding nearest center over the number of points processed on each compute node and visualized as Figure 5.7(a). The results indicate that the k-means algorithm performing on Storm cluster achieves a little better convergence.

Because of Spark's DAG computational model, Spark Streaming doesn't support iterate operator. Instead of forwarding updated centroids in a nested cycle, Spark Streaming implements stream k-means in a different way. It maintains a clustering model and updates the model after each micro batch processed. The update interval of clustering model is the same as micro batch interval. More detail about Spark Streaming K-Means could be found online[4]. With the default setting of one second micro-batch interval, to keep the average latency of micro-batches is less than one second, the maximum throughput of the cluster achieved is around 1M/s. The average latencies of micro-batches processing data with different generation speed are shown as Figure 5.8. The latency of a micro-batch is the time from when the batch is ready to the time that the processing job is done. The latency of each record also includes the time that the record buffered in a window. The experiment results show that the convergence of Spark Streaming K-Means is very fast. Usually, it is converged in a few micro-batches. Figure 5.7(b)

[4]`https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html`

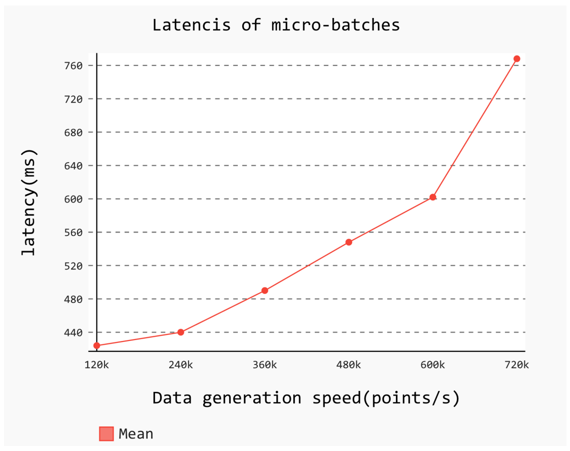


Figure 5.8: Spark KMeans Latency

shows the convergence of Spark Streaming over the number of processed events. It is obvious that stream with lower speed converges faster. But stream with higher speed achieves lower average distance between centroids and real centers after converged. In Figure 5.7, it is easy to notice that the workload running in Flink and Storm is converged after 2000 points processed, that is much lower than Spark Streaming. This is mainly because of the difference between processing models of these systems. In Flink and Storm, record in a stream is processed one by one. That means once a latest centroid is calculated, it could be updated to the k-means clustering model. While in Spark Steaming, the k-means cluster model is updated when a micro-batch job is done. The update frequency is significantly lower than Flink and Storm.

	Maximum Throughput (K/s)	Latency			
		Throughout (K/s)	Median (ms)	90% (ms)	99% (ms)
Storm (ack enabled)	1.7	0.9	13	100	410
Storm (ack disabled)	2.7	1.6	21	107	388
Flink	78	40	122	183	310
Spark Streaming	1000	480	986	1271	1837

Table 5.3: KMeans Performance

The performance of this workload is summarized in Table 5.3. It is clear that Spark Streaming achieves the best maximum throughput, and Storm achieves the lowest median latency. The percentile latencies of Flink is more compact than that of Storm. We also performed some experiments with high dimension point stream. The experiment results show that increasing point dimension has very limit effect on workload performance. The main result of increasing point dimension is leading to more computation in point distance calculation. Which indicates that computation is not a bottleneck of this workload.

5.4 Summary

The experiment results of these workloads show that both Flink and Spark Streaming achieve significantly higher throughput than Storm. But Storm usually achieves a very low median latency. Median latencies of these workloads running in Flink is much higher than Storm. But Flink achieves a similar 99-th percentile latency. In most case, the latencies of Storm and Flink is less than one second. Compared with other two workloads, all these systems get a worse performance in workload AdvClick. Because of micro-batch computational model, there is a tradeoff between throughput and latency of Spark Streaming executing this workload. Normally, the latency of Spark Streaming is much larger than Storm and Flink.

In practice, the selection of stream processing systems depends on the situation. If an application requests very low latency, but the requirement of throughput is not stringent, Storm would be the best choice. On the contrary, if throughput is the key requirement, Spark Streaming is a very good option. Flink achieves both low latency and high throughput. For most stream processing cases, Flink would be a good choice.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have presented a benchmark framework named StreamBench, which aims to provide tools for apples-to-apples comparison of different stream processing systems. One contribution of the benchmark is the definition of three core workloads and corresponding data generators, which begin to fill out the space of evaluating performance of these systems performing different operators. Another contribution of the benchmark is its extensibility property. New workloads can be easily created, including generalized workloads to examine system fundamentals, as well as more domain-specific workloads to model particular applications. As an open-source project, developers also could extend StreamBench to evaluate other stream processing systems. Currently, there is not a standard benchmark framework for stream processing systems yet. The extensibility of StreamBench allows the stream processing community extend and develop it to be a standard benchmark framework.

We have used this tool to benchmark the performance of three stream processing systems, and observed that the throughput of Flink and Spark Streaming is much better than Storm, but Storm achieves a much lower median latency in most workloads. The AdvClick workloads also indicates that Storm and Flink provide low level APIs, and it is flexible for users to implement new operators with these APIs. These results highlight the importance of a standard framework for examining system performance so that developers can select the most appropriate system in practic.

6.2 Future Work

In addition to performance comparisons, other aspects of stream processing systems are also very important factors when selecting a system to deal with stream data in practice, such as scalability, elastic speedup, and availability. For example, a stream processing application is running in a cluster, it is possible that the speed of input stream keeps increasing and becomes larger than cluster's maximum throughput. In this case, scalability is very important which indicates how much the processing ability of a cluster could increase by adding more compute nodes. In our future works, we will implement more workloads to examine these aspects of stream processing systems.

Beside design new workloads, we also could extend StreamBench to benchmark other stream processing systems. In the thesis, we only selected three stream processing systems to run the benchmarks. There are many other stream processing systems that are widely used, such as Amazon Kinesis and Apache Samza. Extending StreamBench to evaluate these systems is listed in our future work.

There are some issues in StreamBench that could be improved or fixed in the future. First, the speed of data generation can't be controlled precisely. It fluctuates around some point. The throughputs mentioned in this thesis are all approximate value. Another issue we noticed is the amount of data in Kafka affects the performance of Offline WordCount, especially for Storm. How the amount of data in Kafka and other features of Kafka cluster affect the performance of stream processing is a very interesting research topic. It might help us evaluate the integration between system processing systems and distributed message systems. As mentioned in § 5.1.2, currently, Flink doesn't support pre-aggregation and parallel window could only be applied on keyed stream. It is possible to implement Windowed WordCount with Flink's low level API. Last but not least, it is always good to upgrade stream processing systems to the latest version in StreamBench.

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] David Patterson. For better or worse, benchmarks shape a field: technical perspective. *Communications of the ACM*, 55(7):104–104, 2012.

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

[42] 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.

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

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

[45] 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`.

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

[47] 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.