

© 2018 Srujun Thanmay Gupta

BENCHMARKING MODERN DISTRIBUTED STREAM PROCESSING
SYSTEMS WITH CUSTOMIZABLE SYNTHETIC WORKLOADS

BY

SRUJUN THANMAY GUPTA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Prof. Indranil Gupta

Abstract

Real-time analysis of continuous data streams using distributed systems is an emerging class of data analytics problems that require systems with high throughput and low latency to efficiently analyze high velocity data. As stream processing applications become increasingly popular, many frameworks used to build clusters to process this data have emerged in recent years. These include frameworks like Samza, Storm, Heron, Spark Streaming, Flink, and Apex. For system administrators and developers, there is great value in understanding the capabilities and performance of their stream processing workloads, given the various frameworks running on their cluster configuration.

In this thesis, we present Finch, a new benchmarking tool that can be used to create synthetic stream processing workloads. Finch generates metrics that system administrators and developers can use to benchmark their stream processing applications. To achieve this, Finch provides a flexible and easy way to define arbitrary workloads using tunable operators. It then translates these workloads into applications that are run by the target system. To describe Finch's design, we investigate what parameters affect workload performance, and present studies on fault tolerance and system scalability. We then use Finch to understand and compare two popular stream processing frameworks, Samza and Heron.

Keywords: stream processing; analytics; benchmarking

To my family, for their love, motivation, and support.

Acknowledgments

I would like to thank Professor Indranil Gupta for his advice and ideas that guided me through writing this thesis. I would also like to thank Shadi Noghabi for mentoring me throughout the project. I learned a lot from her about technical research, design, and systems. I would not have completed this thesis without her help.

Table of Contents

Chapter 1	Introduction	1
1.1	Contributions of this thesis	3
1.2	Outline of this thesis	3
Chapter 2	Background and Motivation	4
2.1	What is stream processing?	4
2.2	Stream operations	6
2.3	Modern stream processing frameworks	8
2.4	Related work in stream processing benchmarks	13
Chapter 3	Finch: Benchmarking Modern Stream Processing Systems	15
3.1	Goals	15
3.2	Design	16
3.3	Finch Workload Interface (API)	18
Chapter 4	Finch: Architecture	19
4.1	Finch-producer	20
4.2	Synthetic Workload Description	21
4.3	Finch-core	24
4.4	Finch modules	25
4.5	Metrics	26
Chapter 5	Experiments with Finch	27
5.1	Experimental setup	27
5.2	Feature-based analysis	28
5.3	Application-based analysis	34
5.4	Summary of experimental results	36
Chapter 6	Conclusion	38
References	39

Chapter 1

Introduction

When dealing with data analytics, it is common in the industry to describe “Big Data analytics” using **Four V**’s: Volume, Variety, Velocity, and Veracity [1]. Currently, data analysis is largely accomplished using large-scale distributed batch systems like Hadoop [2] that focus on the Volume aspect. They utilize the MapReduce programming model [3, 4] where computations on the data are specified in terms of *map* and *reduce* operations. MapReduce is applicable for offline data analysis as the input dataset needs to be read and processed completely to extract meaningful information. In recent years, another set of data analytics problems have emerged in which input data (abstracted as a continuous stream of data items) needs to be processed online and dynamically. The solution for this case is called stream processing.

Stream data falls under the Velocity category, wherein the data is produced rapidly and continuously over time, possibly even at a varying and unpredictable rate. The data items could originate from multiple sources like application logs, sensor measurements, or user action loggers. In this model, data items are read and processed in real-time through a DAG (directed acyclic graph) with many types of operations that include *map*, *reduce*, *filter*, *join*, etc. This model enables a wide range of possible applications. Stream processing can be seen in live event-processing applications like localized time-based trends, for example, calculating trending discussion topics in a social network like Twitter or Facebook. Stream processing can also be used in live-processing of IoT (Internet of Things) data from sensor devices to detect anomalies or perform predictions using machine learning. These and more use cases are described in [5].

Stream processing systems, the focus of this thesis, are built to ingest this data, continuously process it using stream operations, and either respond to user queries or produce live results about the data. As the number of applications and use cases of stream data analytics grows, so do these software

platforms to build and run stream processing clusters. Classical systems include Medusa [6], STREAM [7], Aurora [8, 9] and Borealis [10]. These systems originally described the operator model that is used in streaming systems today, and how the computation can be distributed for parallel processing.

In recent years, more modern systems have been introduced that are built for larger scale and higher data throughput needs. These include frameworks like Yahoo’s S4 [11], MillWheel [12], Photon [13], Pulsar [14], Flink [15], Apex [16], Samza [17], Storm [18], Heron [19], Spark Streaming [20, 21], etc. A majority of these papers have emerged from industrial research to address the unique requirements of their business end-user applications. Each framework has particular features and incorporates choices that were originally designed to target a particular use case. However, many of these complicated details are either explained in lengthy documentations or not explained at all. This large variety in available systems has made it difficult for developers to understand the differences among systems and their trade-offs. Choosing the appropriate system with suitable configurations for specific applications and workloads is extremely complex [22]. Users of stream processing frameworks instead need analysis that details what framework would best fit the specific streaming applications they would like to run. Additionally, users need concise information about the design trade-offs in each of these frameworks, and the features and guarantees they provide. In this thesis, we present a new benchmarking tool called Finch that addresses these requirements.

The goal of this thesis is two-fold:

- (i) To create a benchmarking tool that enables users to define **generic** and **flexible** stream processing pipelines. These pipelines should execute on the multiple streaming systems listed above without much modification. The tool should be able to generate analysis data that explains the system’s design trade-offs. It should also help users select the framework that best matches their needs so that they can appropriately tune their cluster to maximize performance.
- (ii) To explain the requirements and analyze the parameters related to stream processing frameworks. We discuss features like fault tolerance, scalability, state recovery, and more. The goal is to use Finch to create

experimental studies for these features and application-based workloads that serve as examples for users to understand and benchmark their workloads.

1.1 Contributions of this thesis

In this thesis, we present the following:

- (i) The design and implementation of Finch, a new stream data processing benchmarking tool to generate and run synthetic workloads.
- (ii) An analysis of requirements of stream data processing frameworks and the parameters that affect the performance of stateless and stateful stream processing workloads.
- (iii) Feature-oriented studies that use Finch to understand specific features. In particular, we focus on fault tolerance and scalability of streaming systems.
- (iv) Application-oriented studies to compare systems via real-world applications. We use Finch to measure and compare performances of simple benchmark applications (e.g., “word count”) across two popular frameworks, Samza and Heron.

1.2 Outline of this thesis

In Chapter 2, we summarize the motivation and goals of stream-processing frameworks, and describes some existing stream-processing frameworks and their design. Chapter 3 introduces Finch and its design requirements. In Chapter 4, we provide a detailed description of Finch’s architecture and the implementation of the benchmarking tool. Chapter 5 details the studies on feature and application based workloads that use Finch to create and collect experimental data. We describe and compare the results from tests within these studies. Finally, Chapter 6 concludes the thesis and lists possible future work ideas to extend Finch’s functionality.

Chapter 2

Background and Motivation

In this chapter, we describe what stream processing is, define stream data operations, discuss the common requirements of stream processing frameworks, and review some popular modern streaming frameworks. Finally, we provide motivation for benchmarking tools and discuss related work in stream processing benchmarking.

2.1 What is stream processing?

Stream data processing has become a common medium for today’s analytics needs. In live applications like monitoring systems, the data is typically spread out over time and often emitted at very high rates such that it becomes challenging to log and store all the data points. Instead, it is more intuitive to process the data immediately and only log the partial results. The added benefit of this model is that analysis can be obtained in real time, thus, the response latency is greatly reduced when compared to batch-processing systems. With improved computational power of modern systems, it has become possible to create continuous data processing systems.

Within stream processing, data is abstracted as a stream which can be defined as an ordered sequence of data tuples. To process this data, each item is passed through a series of “operations” or functions. Operations either transform the incoming data to emit a new data tuple, merge the tuple with data from other streams or database tables, or aggregate multiple items together to derive some information from them.

Stream processing frameworks provide implementations of these various operations. Users can connect operations together to build a stream processing pipeline. These frameworks essentially abstract the live data analysis logic as a directed graph of operators. This abstraction can be visualized as in

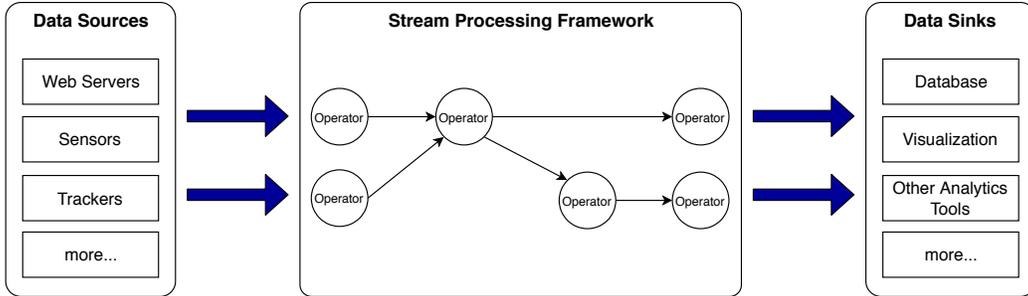


Figure 2.1: Abstraction of Stream-Processing Systems

Figure 2.1. Note that the graph does not necessarily need to be acyclic, as events can be reprocessed as required. The operators are functions that receive an incoming message, operate on it, and optionally produce any number of outgoing messages. Data can be sourced from multiple locations, combined with other sources, and finally sent to multiple sinks similarly.

One of the earliest descriptions of stream processing was given in the Aurora paper [8] in 2003. Aurora was designed as a data-flow system with a query algebra that consisted of functions like *filter*, *select*, and *aggregate*. Complex data flow queries could be defined by combining these functions. In modern stream processing frameworks, the operators available have evolved from these primitive functions that originated in Aurora.

Simple functions like the ones described above are enough to define data stream pipelines of varying complexity. On the other hand, in some applications it is necessary to maintain information across multiple messages. This information could be collected and updated based on the data items seen until the current point. In stream processing, this information is called **state**, and can be represented as a data-structure that is persistent in an operator across multiple items. An example of a stateful streaming pipeline is counting the clickthrough rate of ads, which is the ratio of a website’s visitors who click on a given ad versus the total number of viewers of the website. The clickthrough rate is measured by joining two streams on the user ID: ads viewed and ads clicked. As the web-tracker emits user interaction events, the stateful operation will need to keep track of which interaction events are describing user ad-clicks and which ones are ad-views. Another example is tracking user events within a given time window, like per-user interactions (pages or posts viewed) in the last 5 minutes. This is accomplished by aggregating the stream of interactions over a timed window. Yet another example

Table 2.1: Operator Types in Stream-Processing

Stateless	Stateful
Split	Join
Filter	Window
Map	
FlatMap	
Merge	

is detecting fraudulent credit card transactions by saving the last transaction of a credit card and comparing its attributes to those of the new incoming transaction [23].

Thus, based on the examples above, stream processing operators can be broadly categorized into **stateless** and **stateful** operators. These are listed in Table 2.1. Stateless operators only apply a function to the current message. Stateful operators are more complex and maintain some contextual data that is retrieved and/or updated when applying the function to the message. The state is persistent across multiple messages, and hence, is usually stored in-memory, or cached to another persistent store. The techniques various stream processing frameworks use to manage state are described in §2.3.

2.2 Stream operations

Given the operators in Table 2.1, we can describe each of them as below:

- **split**: takes an input data tuple and makes n copies of it, emitting either the full message or a fraction of it. This operator can be used when multiple downstream operators are reading from the same upstream operator. Example: splitting a stream of log messages to regular messages and messages with exceptions/errors.
- **filter**: takes an input data tuple and emits it, either whole or in part, only if it satisfies the given predicate. Example: in a document word frequency calculator, filter out all the stop words like ‘a,’ ‘the,’ ‘I,’ etc.
- **map**: applies a given single-input function to the data item to emit a new tuple. This can be used to modify the data tuple to extract information about it or augment it with data from external sources. Example: in a

stream of temperature sensor outputs (large JSON maps of readings), extract the relevant data points and optionally convert them between units (Celsius to Fahrenheit).

- **flat map**: this operation is similar to **map** but if the map transformation creates multiple output data tuples, it collects them into a single sequence item (like a Java `List` or `Collection`) and emits it. Example: convert a stream of sentences to a stream of the collection of words in each sentence.
- **merge**: also called **union**, this operator takes items from multiple input streams and emits them one-by-one to a single output stream. Example: when there are multiple sources that are emitting similar data, we can combine them into a single stream.
- **join**: this operation is similar to SQL Joins. It takes multiple streams as input and combines the messages that have the same key (or the same result from a key extraction function) and emits a single combined message. As with SQL Joins, it is possible to use inner, left, right, or outer joins in this operator. **join** is stateful because it needs to store the join-keys across multiple messages to combine them as needed. Example: if we have two streams, where one is a stream of shipment records and the other is a stream of customer orders, then we can calculate all the fulfilled orders by joining the two streams on the product ID.
- **window**: collects messages based on a finite-size time frame, and emits new messages based on specified triggers after running a given computation on the collected messages. Windows are usually defined as two types:
 - **sliding**: windows can overlap in time, thus a message can be collected in multiple windows. Sliding windows are defined using the window time length and the time interval after which the window slides forward. Example: a stream of active users in the last 5 minutes. In this case, the time window is always sliding and the start time is relative to current time.

Table 2.2: Stream-Processing Algorithms

Algorithm	Description
<i>Sampling</i>	Collect data items to approximate information about the data stream.
<i>Windowed Analysis</i>	Data items are grouped into windows, which can be based on time or number of messages. Example applications are timed pattern analysis, anomaly detection, etc.
<i>Clustering</i>	Collection of problems in which we need to compute k representative data items that minimize error over all the items in the data stream.
<i>Sketch/Summary</i>	Executing queries over the data stream. Examples include frequency sketches (heavy hitters) or quantile data (mean, median, percentiles, etc.) [24].
<i>Histogramming</i>	Partitioning the stream data items into buckets and running computations over the bucket items. An example applications is categorizing input stream data items into categories.

- **tumbling**: windows do not overlap in time, thus a message is processed in exactly one window. Tumbling windows are defined using just the window interval, after which the window moves forward. Example: a stream of active users for each day. In this case, the window time is fixed and user events do not overlap between days.

Using the basic operators described above, a few general purpose algorithms can be defined over stream data as discussed in further detail in [5]. Most stream data processing pipelines implement derivations of these algorithms by combining the operators we listed above. We describe some of these algorithms in Table 2.2.

2.3 Modern stream processing frameworks

Since there has been enormous growth in the various applications, algorithms, and use-cases of stream processing systems, several frameworks have emerged in recent years. To guide ourselves towards motivating a benchmarking tool

to analyze these frameworks, it is useful to first look at stream processing properties and requirements. By understanding these, we can reason about the features and trade-offs of each system and focus the benchmarking tool towards studying them.

Stonebraker et al. describe the “8 requirements of real-time stream processing” in their 2005 paper [25] and reason that these requirements are important for any system to excel at real-time stream processing. We adapt their findings to summarize common goals or requirements of modern streaming systems:

- (i) **Distributed parallel computation:** Since each operator can potentially process thousands or even millions of data items, backlogging of messages should not happen as it could hamper the availability of the system. The processing logic should be distributed across multiple smaller tasks that process a subset of the incoming data stream. Techniques to achieve load balancing and parallelism across these smaller tasks, like *distribution-aware key partitioning* and *adaptive scaling of operators*, are described in [26] and [27] respectively.
- (ii) **Scalability:** With the ever decreasing price-to-performance ratio of computer hardware, it is becoming easier to expand clusters by scaling-out with additional nodes. Concurrently, the amount of data and the number of applications are also increasing greatly. Stream processing frameworks should make efficient use of the cluster resources to deliver low processing latency and high data throughput. These requirements and applications are discussed in [28]. The trade-offs that the system delivers between these two properties greatly influences the type of applications that will be suitable for that framework.
- (iii) **Resiliency and fault-tolerance for high availability:** A consequence of the scalability requirement is that with more components in the system, the probability of individual component failure also increases. Stream processing frameworks should be resilient to component failures and recover quickly with minimal or no loss of data. Within the context of stream processing, state recovery is also important, otherwise intermediate processing results can be lost.

- (iv) **Flexible data-model integration:** Incoming data streams can be from multiple sources, each possibly of differing type and message production rate. Similarly, processed output could also be sent to multiple destinations. Both sources and sinks can be any combination of key-value/data stores like Redis [29], databases, filesystems, pub-sub systems like Kafka [30], or message queue systems like RabbitMQ [31] and ZeroMQ [32]. Since common streaming applications need to source data from these different sources, the framework should provide integration with these endpoints to acquire and deliver data.

All of these requirements are implemented at varying levels in different stream processing frameworks. It is important to be able to measure, analyze, and evaluate these features in the various frameworks. Now that we understand these base requirements, we look into some of the popular frameworks available currently and how they address these requirements.

2.3.1 Storm

Storm [18] was originally developed at Twitter to power the real-time stream data tasks for Twitter services, before becoming an open source Apache project in 2014. Streaming applications in Storm are defined using directed graphs called “topologies” that are built using *Spouts* (data sources) and *Bolts* (data operations). A data stream itself is abstracted as a stream of tuples. When data moves between a producer spout/bolt and a consumer bolt, the data tuples are partitioned by various grouping strategies (shuffle, field grouping, etc.). Storm provides two semantic guarantees for processing each tuple, “at least once” and “at most once.” *At least once* guarantees that each tuple is processed by the topology with the possibility of some extra processing in case of failures. In contrast, *at most once* guarantees that each tuple is processed once or dropped completely upon a failure. To provide for this, each tuple has a unique ID associated with it that bolts acknowledge upon processing, backflowing all the way up to the originating spout, after which the tuple is retired. Storm can use the YARN Hadoop scheduler [33] to manage instance *workers*, which in turn are controlled by a master node called the *Nimbus*. *Monitors* detect node failures and allow Storm to recover and resume processing, but there is no provision for recovery of state.

2.3.2 Heron

Since Storm had a few shortcomings, both in performance and features, Twitter developed Heron [19] in 2015. Heron is an improved system that is backwards compatible with Storm. Heron builds upon Storm’s data model and adds more features like on-demand scaling of hardware resources using containers (Linux cgroups [34]) for more flexible streaming applications, and a dynamic *backpressure* handling mechanism. In a pipeline with separate branches operating at different speeds, the slower branches can cause operators further down the stream to clog and lose incoming data. Heron instead dynamically adjusts the upstream *stream managers* to clamp down on its tuple emission rate. Heron also introduces *Streamlets* [35], a functional programming-style API to define topologies that is a more flexible data model than the spout-bolt model from Storm. State in Heron is managed similarly to Storm: it is stored in-memory and cannot be recovered after failure although there is ongoing work by the maintainers of Heron to introduce reliable state features.

2.3.3 Flink

Flink [15] presents a unified streaming dataflow architecture for both *unbounded* (stream) and *bounded* (batch) data models. Flink also uses a DAG abstraction for representing dataflows but with separated *DataSet* and *DataStream* APIs. State management is explicit within each operator by giving users the flexibility to register variables within each operator. Flink exchanges data between producers and consumers using buffers that are transmitted upon filling or timeout. As the buffer trigger is configurable, it allows Flink to be flexible with regard to the throughput-latency trade-off: large buffer size implies better throughput, smaller buffer size gives lower latency. Flink implements fault-tolerance with state recovery by using distributed consistent snapshots. It checkpoints each operator’s state at regular intervals and re-executes the checkpoints upon recovery. The mechanism is called *Asynchronous Barrier Snapshotting*. Control messages are injected into the data streams at user-defined intervals to tell the operators to save current state to a durable store (for example, HDFS [36]).

2.3.4 Samza

Samza [17] is another open-source stream processing system originally developed at LinkedIn. Samza, unlike Flink, combines the dataflow abstraction for stream and batch data by using the same API for both. It supports stateful operators with stream reprocessing mechanics, and size-independent state recovery. Samza achieves its optimizations for recovery time by using a parameter called *Host Affinity* that tries to restart failed tasks on the same machine to quickly access and rebuild state. Samza partitions the state data among each operator’s individual tasks and stores it either in-memory or on-disk. It also maintains a compacted changelog that captures updates to the state for replaying in the case of a failure. Finally, to address scalability, Samza splits data streams into partitions that are mapped onto individual operator tasks using consistent hashing. The task containers are then distributed across multiple workers for distributed parallel processing.

2.3.5 Spark Streaming

Spark Streaming [21] is built on top of the Spark batch data processing framework [37]. It uses an abstraction called *D-Streams* that exposes a functional programming API that is similar to those used in Heron and Samza. It also provides unification with batch processing interfaces. To facilitate this, D-Streams is a unique data structure that is internally batch-based and built upon Spark’s *resilient distributed dataset* (RDD) [38]. RDDs are immutable data collections that enable Spark Streaming to rebuild data upon failures by tracking the deterministic operations that led to the creation of a given RDD. Within a streaming context, Spark Streaming builds micro-batches of data into RDDs that are transmitted to downstream operators. Because the operators have to wait until enough data has arrived to build the micro-batch, Spark Streaming incurs comparatively higher latency in streaming applications. On the other hand, there are higher gains in throughput because more data is transmitted on average. This makes Spark Streaming suitable for workloads that can tolerate up to second-scale latency.

As we can see in the sections above, there are numerous frameworks with innumerable intricacies in implementation that make it challenging for users

to discern the trade-offs of each system. There is a need for tools that can benchmark these systems under various types of workloads and applications to highlight the impact of each system’s design choices. These benchmarking tools need to be created in order to aid users in understanding what parameters are relevant to their use cases. Additionally, we need to understand where each system stands with regards to the requirements that were discussed in §2.3. Before we introduce a new benchmarking tool, we first look at some previous work in benchmarking tools for stream processing frameworks and highlight their innovations and shortcomings.

2.4 Related work in stream processing benchmarks

There has been extensive research conducted on analyzing and comparing databases [39] and MapReduce systems [40, 41, 42]. Multiple tools exist that test various parts or features of these systems. Stream processing benchmarks, on the other hand, are much less researched due the relatively recent increase in popularity of these systems.

In Yahoo’s analysis on benchmarking streaming computation engines [43], they build a very specific advertisement analytics pipeline. The authors present evaluations on Flink, Spark Streaming, and Storm. The analysis focuses on the window update times and latency measurements on these systems. Although this benchmark simulates streaming transformations that encompass both stateless and stateful operations, the benchmark only represents a single type of pipeline whereas real-world applications of streaming systems can implement various dataflows that this benchmark does not capture. Additionally, the performance evaluation only focuses on window update times of each framework. As noted in §2.3, stream processing frameworks have multiple design and trade-off differences that should also be analyzed. To add to that, benchmarking tools need to also compare other metrics like data throughput, fault tolerance mechanisms, and scalability which were also not addressed in their analysis.

Lopez et al. [44] compare the features of Storm, Flink, and Spark Streaming in their paper. Their experiments were carried out on a single dataset that was produced by the authors [45] and they only analyze throughput vs. parallelism, and system behavior under failures. Stream processing systems

have many more parameters that an ideal benchmarking tool should also evaluate.

Another paper [46] produced in collaboration with Databricks, the maintainers of Spark and Spark Streaming, introduces StreamBench. This tool includes a suite of seven benchmark programs (identity, sample, projection, grep, word count, distinct count, and statistics) covering both stateless and stateful operations. These example workloads are representative of even more types of streaming applications and the presented analysis tests the throughput, latency, and fault-tolerance behaviors of three frameworks: Spark Streaming, Storm, and Samza. However, their benchmark workloads are only application-focused. A more flexible tool to stress test targeted features by simulating any type of workload will provide a better evaluation of streaming systems. Additionally, users are more concerned with the feature performance of frameworks (scalability, fault tolerance, resiliency, etc.) that this paper does not address in much detail.

A more comprehensive benchmarking tool, albeit for MapReduce, is developed in [42] that analyzes two production traces of MapReduce runs from Facebook and Yahoo to highlight the diversity in different types of MapReduce workloads. These traces were then sampled to generate synthetic workloads that were more representative of real-world MapReduce runs. Their analysis can be extended to show that one-size-fits-all benchmarks are insufficient to completely understand streaming systems, and hence, a better solution is needed.

Based on the works described above, it is apparent that current stream processing benchmarks are not extensive enough to support generic workload types that benchmark both feature and application based tests for streaming systems. These shortcomings lead us to the design for a new benchmarking tool called Finch, which we introduce in the next chapter.

Chapter 3

Finch: Benchmarking Modern Stream Processing Systems

Now that we have detailed the motivation behind designing a generic and extensible benchmarking tool for stream processing systems, we introduce Finch, our open-source and extensible benchmarking tool. Finch can run arbitrary user-defined workloads without writing any code for the target system to be benchmarked. This chapter explores the goals of Finch and its design.

3.1 Goals

Finch was created to achieve the following goals:

1. *Generate arbitrary and flexible synthetic workloads without writing code for each specific target system:*

Since most existing benchmarks only focus on one or a few specific workloads, Finch should allow users to generate arbitrary synthetic workloads that reflect real-world streaming applications. Finch should give users the flexibility to define parameters of the workload like distribution, operations, parallelism, and more to customize the workload to the test case.

Additionally, each workload should easily compile to the target system without the user needing to write any code for that system. The module implemented for that target should handle all necessary configuration and execution.

2. *Enable evaluation of both system features and application performance:*

Users should be able to create workloads to test streaming scenarios or features like failure response, scalability, etc., along with real-world application-based workloads to test the end-to-end system. Ideally, this

should be done without requiring users to manually define the actual operator graph or consuming actual data. Thus, Finch should be able to generate arbitrary data that emulates actual data streams and evaluate feature-based and application-based workloads.

3. *Highlight trade-offs in features between different target systems:*

As highlighted in §2.4, it is very challenging to compare how each stream processing framework handles the various trade-offs in state management, scalability, failure recovery, throughput, and latency. Finch should generate results from each tested workload that compare how each target system performs in each of these metrics.

3.2 Design

To address the above goals, Finch’s design includes 4 different components: input data generation, workload generation, integration with frameworks, and collection of metrics. These are described in Figure 3.1 and in the following sections.

3.2.1 Input data generation

We call Finch’s input generation module *finch-producer*. To simulate real-world workloads, there are two options that can be considered when trying to source input data for the workload:

- Real data streams can be sampled to reflect actual use cases.
- Users can define the characteristics of the data stream, like variability in message rate, size, and distribution of keys.

In Finch, we chose to implement the second option as the customizable source parameters give users flexibility to define any type of input sources. If instead we store collections of data and sample them, the input data will only conform to the characteristics of that real data stream and will not be generic enough to simulate arbitrary workloads. However, if users would like to use their own data streams, Finch supports plugging any source stream as the input.

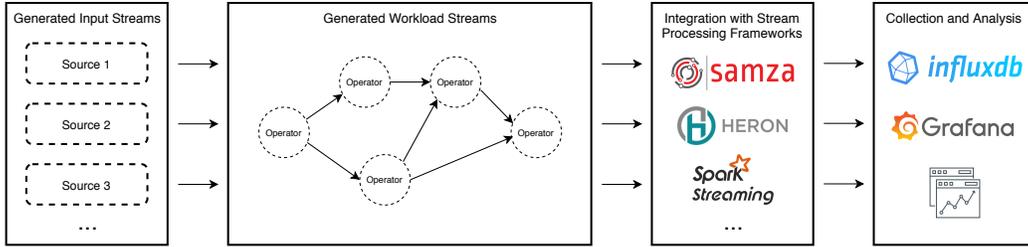


Figure 3.1: Finch Design

3.2.2 Workload generation

To allow users to easily create workloads that can execute on any stream processing framework with flexible customization options, Finch needs a simple format to describe the workload. The description should generalize to all streaming frameworks, and be able to compile to a specific framework’s implementation.

The workload is described by the sources, operators, and the sinks. These are combined to create a DAG for the application’s pipeline. Each operator is configurable by various parameters specified by the operator’s function. These configurations make Finch flexible and generic to describe arbitrary stream processing applications that can be converted to the specific implementation for each target framework.

In Finch, we chose the JSON format to describe the workload. JSON is a popular data interchange format with available parsers in multiple programming languages, hence it is a viable choice here. Data sources, operators, and data sinks are defined as JSON objects in a description file. The module that parses this information is called *finch-core*.

3.2.3 Integration with stream processing frameworks

Each stream processing framework uses different APIs and languages to define a streaming pipeline. Instead of a monolithic design, Finch is built with modularity such that each framework’s implementation can be plugged into the core Finch architecture and be customized easily without affecting the other components of Finch. These modules receive the JSON workload description and build the workload DAG within the target framework, and are responsible for executing the workload using the appropriate computation

abstractions like Hadoop/YARN, Apache Mesos [47], or container management systems like Docker or Kubernetes [48].

3.2.4 Output metrics and data collection

Once Finch executes a workload, it should be able to collect measurements from that execution into a persistent datastore. Users should be able to view and analyze the collected data in the same manner, regardless of the streaming feature or target system being tested.

Measurements from system benchmarks are typically sorted by time and so Finch modules export collected system metrics to a time-series database. Each measurement is also tagged with metadata about the originating node or container, the version of Finch and the module creating the measurement, and other related details. This allows easy querying either real-time using graphing dashboards or later on for static analysis.

3.3 Finch Workload Interface (API)

Finch combines each of the previously detailed components to expose a simple API for users to create and run workloads that work with multiple target stream processing applications. The operator API gives users the freedom to create workloads with any combination of the operators detailed in §2.2. To this end, users only need to create a JSON file listing the sources, operators, and sinks, and define their individual parameters. This makes it very easy for developers as they do not need to be familiar with the target system to be able to create a stream processing application that runs on it. Finally, we also provide definitions for typical queries on the collected metrics. These definitions are combined as “dashboards” that can be imported in visualization tools like Grafana for easy and quick analysis of performance.

With these design ideas laid out, in the next chapter we describe Finch’s architecture and the implementation of each design component.

Chapter 4

Finch: Architecture

In this chapter, we shall dive into the architecture of Finch, the implementation of each component, and how they integrate to create the overall benchmarking tool. The high-level architecture can be seen in Figure 4.1. Based on the design components described in §3.2, the five major parts of Finch are:

- *Finch-producer*: responsible for creating the input data sources in Kafka and generating data into them based on the source parameters defined in the workload description.
- *Workload description*: describes the scheme used to create customizable workloads.
- *Finch-core*: parses the workload definition and instantiates the operator DAG in the workload. It also coordinates the individual Finch modules and the metrics collection.
- *Finch modules*: implement interactions with various target stream processing frameworks (`finch-samza`, `finch-heron`, etc.) and handles execution of the workload.
- *Metrics*: collects the various performance measurements from the target system like message throughput, end-to-end latency, CPU usage, memory usage, etc. These metrics are used in the analysis of system features and application performance.

All components of Finch depend on the **workload description** that is specified by the user. First we shall describe the data source emulation in Finch (§4.1), followed by the structure of the workload description file (§4.2), and then explain each of its components in more detail (§4.3, §4.4). Finally, we will describe how the workload metrics are collected (§4.5).

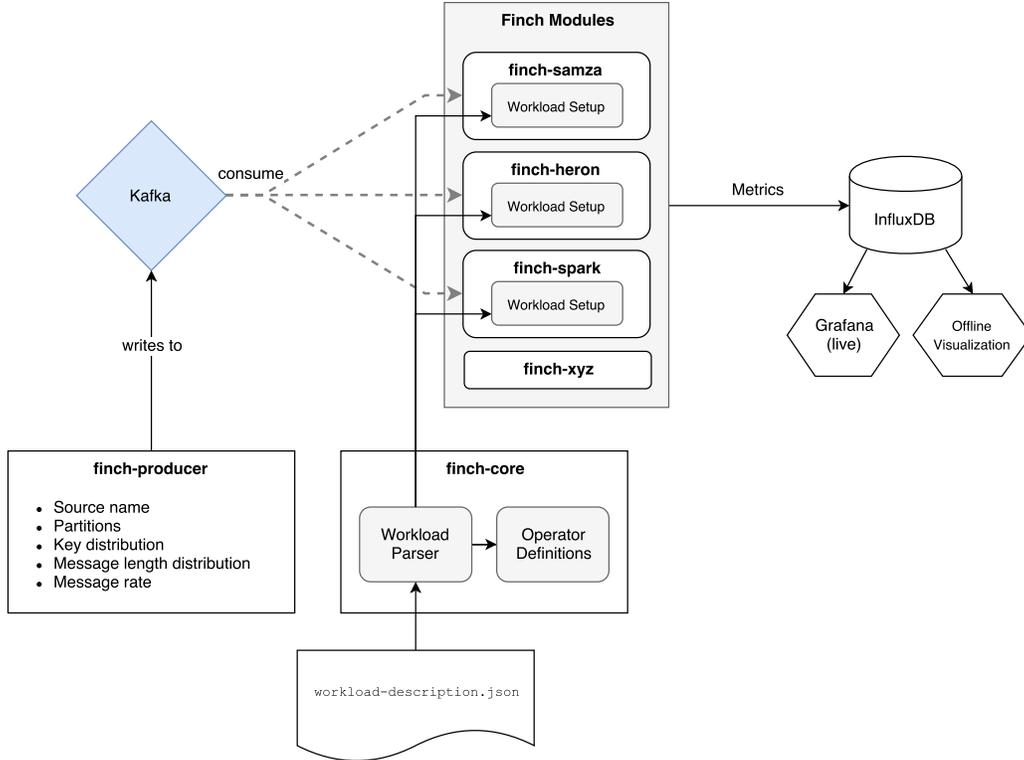


Figure 4.1: High-level Architecture of Finch

4.1 Finch-producer

Finch’s input data production component needs to simulate real-world sources of data. To emulate them, we need to first to understand the characteristics of these real data sources. Each data stream has messages that have varying **message length**, and are possibly produced at varying **rates**. Finally, since data streams in stream processing frameworks are keyed, the stream has a certain limited number of **keys** that are also created at varying rates. Each of these are customizable within Finch.

To address the above characteristics, various parameters are exposed by the workload description that `finch-producer` parses. It creates a multi-threaded message emitter, one thread per source to produce messages to. Messages are produced to the Kafka pub-sub system using the Kafka Java Client API [49]. Each thread uses the various distribution parameters to create messages at the requested rate, and publishes the message to the Kafka topic. The Kafka client is configured to wait for acknowledgements only from the leader of that topic’s partition group. Since `finch-producer`

should normally be on one of the Kafka nodes, this configuration is sufficient to ensure that created messages are not lost.

The generated messages have the format:

$$key = \textit{“keyX”} \tag{4.1}$$

$$value = \textit{“timestamp,random_string([a,z],N)”} \tag{4.2}$$

where $X \in [0, 1, 2, \dots, K - 1]$, K is the size of the keyspace, and N is the sampled length from the message length distribution defined in the workload description.

This format allows Finch to create arbitrarily sized messages that simulate any kind of source. The keyspace size defines the level of redundancy among keys: smaller keyspace means more redundant distribution and vice versa. Additionally, the key distribution defines the probability of duplicate keys (some keys will be more likely in a Zipf distribution, compared to being equally likely in Uniform). The distribution parameter can also be used to simulate skewed workloads. These skewed workloads have messages of some type appearing many more times than messages of other types. Thus, this format satisfies the various characteristics of data sources that we described earlier. The user API to define workload sources are defined in §4.2.1.

4.2 Synthetic Workload Description

Synthetic workloads that emulate real-world streaming pipelines are defined in a JSON-schema file. This file has three main sections: **sources**, **sinks**, and **transformations**. These three make up the complete directed graph of a pipeline, and users can combine these as needed to create various types of workloads from simple single-function operations to complex multi-operator stateful pipelines.

4.2.1 Sources

The **sources** definition is used to configure **finch-producer**. Sources are essentially customizable random message generators, where each message is a key-value pair. The following parameters can be set to make the generated

messages match the pattern of real source messages:

- **name**: the name used to refer to this source.
- **num_keys**: the number of keys in the keyspace of this source’s messages.
- **key_dist**: the distribution of keys used by messages produced from this source.
- **msg_dist**: the distribution of the length of the messages produced from this source.
- **rate_dist**: the distribution of number of messages produced per second from this source.

For parameters that accept distributions, a Java class that implements Apache Common Math [50] `IntegerDistribution` can be supplied. Some examples include:

- (i) Uniform: each item has equal probability of being sampled.
- (ii) Zipf: some items are more likely to be sampled than others, can be used in `key_dist` to simulate “hot” keys.

4.2.2 Sinks

`sinks` is a list of endpoints in pipeline. These become Kafka topics that the benchmark application finally writes messages to.

4.2.3 Transformations

`transformations` is a JSON map of operators and the stream they operate on. The types of transformations are based on Table 2.1. There are a few optional parameters common among all transformations:

- **cpu_load**: a floating-point number between 0.0 and 1.0 that represents the CPU load of this operator (implemented using for-loop spins).
- **mem_load**: a floating-point number between 0.0 and 1.0 that represents the operator’s load on the memory (read and write operations).

```

1  {
2    "sources": {
3      "s1": {
4        "key_dist": "ZipfDistribution",
5        "key_dist_params": {
6          "num_keys": 10,
7          "exponent": 1.2
8        },
9        "msg_dist": "UniformIntegerDistribution",
10       "msg_dist_params": {
11         "lower": 100,
12         "upper": 1000
13       },
14       "rate_dist": "UniformIntegerDistribution",
15       "rate_dist_params": {
16         "rate": 1000
17     }
18   }
19 },
20 "transformations": {
21   "t1": {
22     "operator": "filter",
23     "input": "s1",
24     "params": { "p": 0.5 }
25   },
26   "t2": {
27     "operator": "modify",
28     "input": "t1",
29     "params": { "size_ratio": 0.5 }
30   }
31 },
32 "sinks": ["t2"]
33 }

```

Listing 4.1: Sample workload description in JSON

- `disk_load`: a floating-point number between 0.0 and 1.0 that represents the operator's load on the disk (read and write operations).

Each operator that defines a transformation has a set of unique parameters. The specific operators in Finch and their parameters are defined below.

4.2.3.1 Stateless transformations

- `filter`: drop some messages from the incoming stream.
 - p : the probability of dropping a given message ($0 \leq p < 1$).
- `split`: copy input message to n output streams, with optional resize of message length.

- *n*: the number of output streams to write incoming messages to.
- *ratio*: ratio of the output message’s size to input message’s size.
- modify: change the size of individual messages or the rate of messages passing in a stream.
 - *size_ratio*: ratio of the output message’s size to input message’s size.
 - *rate_ratio*: ratio of messages emitted per input message (a value of 1 implies that the rate does not change).
- merge: take messages from the given *n* input streams and output them to a single outgoing stream.

4.2.3.2 Stateful transformations

- join: match keys of incoming messages on two streams and output a single combined message to output stream.
 - *tll*: time-to-live for received messages that have not yet joined with another message.
- tumbling window: group messages based on windows of fixed length of time that are non-overlapping and contiguous.
 - *duration*: the time length of the tumbling window.
- sliding window: group message based on fixed length intervals that can overlap.
 - *duration*: the time length of the sliding window.

A sample workload description JSON and its corresponding directed graph are shown in Listing 4.1 and Figure 4.2.

4.3 Finch-core

`finch-core` serves two main purposes: workload parsing and defining the stream operators.

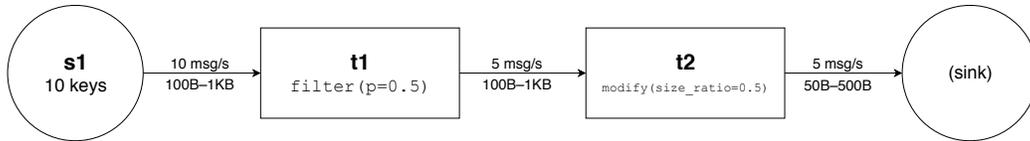


Figure 4.2: Sample Workload Graph

The parser loads the workload definition JSON and converts it to a directed graph representation. We use Google Guava’s graph package [51] to represent the pipeline. Each node indicates the stream operation, and the edges indicate the flow of the data stream. We use the package’s `Network` class since each edge needs to be a full-fledged object holding information about the stream.

Once the graph object is constructed, it is traversed source-to-sink and each operator is loaded. Depending on the actual streaming framework being used, the appropriate implementation of each operator is loaded and the framework’s internal representation of the streaming pipeline is built. For example, `finch-samza` uses a `StreamGraph` object, whereas `finch-heron` uses a `Streamlet Builder`. These are defined in further detail in §4.4.

4.4 Finch modules

For each streaming framework that Finch works with, there is a plugin that implements the workload’s different components. These plugins provide framework-specific code to translate the user-defined workload. To this end, it needs to implement the integration between the framework and the Kafka sources and sinks, and also the implementation for each of the transformations defined in §4.2.3. Currently, Finch provides modules for two streaming frameworks, Samza [17] and Heron [19], and their implementations are described in the following sections. Because of Finch’s pluggable architecture, new modules for integrating with other stream processing frameworks can be easily created.

4.4.1 Finch-samza

Finch’s generic transformations were originally adapted from Samza’s supported operators, which in turn are based on Java 8’s Stream package [52]. Java’s stream API provides a functional programming interface to perform operations on data structures, which adapts very well to streaming architectures.

`finch-samza` uses Samza’s API to implement each of the transformations, and are instantiated by the the workload parser provided by the module. Samza also uses an additional “properties” file to describe the source and sink streams that the streaming application interacts with. `finch-samza` injects these properties at run-time and hence they are not the user’s concern. Samza pipelines can be executed on a variety of clusters, but support for Hadoop YARN has existed for a long time and is well tested. `finch-samza` in turn executes Samza workloads using YARN.

4.4.2 Finch-heron

Heron’s Streamlet API is also very similar to Samza’s functional API. Finch’s Heron module builds the configuration for the workload with the requested CPU/memory resources and the number of containers, and uses the Heron workload parser to instantiate the operators. Finally, the created workload is executed using the Nomad scheduler [53]. Heron workloads can also be executed using the Apache Aurora scheduler [54] on a Mesos cluster.

4.5 Metrics

When running the workloads, each Finch module publishes run-time statistics and other metrics. Since each framework operates differently and publishes various non-uniform metrics, each Finch module also provides a service to collect these metrics and publish them to a Kafka topic in a standardized format. The data in this Kafka topic is then collected and written to a time-series database InfluxDB [55]. This makes it easier to query the performance results of any workload later, and use any tool that can interact with these databases to create visualizations.

Chapter 5

Experiments with Finch

In this chapter, we shall look at experiments that we conducted using Finch on two streaming frameworks, Apache Samza and Twitter Heron. These experiments fall under two primary study cases:

1. *Feature-based analysis*: how does the target system implement features like fault tolerance, scalability, resiliency, and state recovery?
2. *Application-based analysis*: how does the system perform when executing simulations of real application and workloads like word-count, stream grep, and statistics?

These studies can serve as examples that users of Finch can adapt from to create specific experiments to run on their stream processing cluster. It is important to note that each stream processing framework provides numerous specific tunables that can be used to extract better performance output. Finch does not focus on these specifics as they require expert, in-depth knowledge of each target system that is complex to abstract away. Users can adapt Finch and fine-tune the cluster settings to benchmark their workloads using Finch's extensible and easy-to-configure workload interface.

5.1 Experimental setup

To run the studies described in this chapter, Amazon Web Service (AWS) was used. Kafka, as the primary data source and sink, was set up on a 3-node cluster of `c5.xlarge` instances. For the YARN architecture used in Samza tests, we set up a 3-node cluster of compute-optimized `c4.xlarge` instances through AWS Elastic MapReduce, with an additional `m4.large` as the master coordinator node. For Heron tests, we used a similar 3-node cluster of `c4.xlarge` instances with the Nomad scheduler installed. Finally,

Table 5.1: Experimental Setup System Specifications

Type	vCPU	Memory	Network Bandwidth
m4.large	2	8 GB	450 Mbps
c4.xlarge	4	7.5 GB	750 Mbps
c5.xlarge	4	8 GB	2.25 Gbps

system metrics were collected on a single containerized instance of InfluxDB. Specifications of each instance type used are in Table 5.1.

5.2 Feature-based analysis

Within feature-based analysis, our objective is to understand how each system implements features like failure recovery, state resiliency, etc. Understanding these features from a design and implementation perspective is crucial to better knowing the capabilities of these frameworks. Our goal with Finch is to look at these features individually and create benchmarks to analyze and discuss their performance. Additionally, performance metrics in the stream processing context mean measuring metrics like message throughput and message processing latency.

Possible features to study, and the corresponding questions to discuss, include:

1. *Failure recovery mechanics:*

- How does the system react to the failure of individual components?
- How much is the overall system’s performance affected during the failure?
- How long does the system take to recover to normal operation?

2. *Scalability analysis:*

- How well does the system scale upon increasing the number of nodes (containers) participating in the system?
- What is the performance gain or loss when scaling the system up or down?

- How does the system scale as the workload impact increases or if there are more sources of input?

3. *State analysis:*

- How does the framework performance change when state information is stored in-memory, on-disk, or in a remote store?
- What are the performance effects of different state backup and recovery mechanisms like checkpointing and state logs?

4. *Parallelism techniques:*

- What are the performance effects of using different parallelism techniques like keyspace partitioning or operator scaling?

5. *Isolation and multi-tenancy:*

- When running multiple workloads on the same cluster, how much do the workloads impact each other?
- Does one workload cause resource starvation for the others?

6. *Handling workload dynamism:*

- How well does the stream processing system handle spikes in the workload? What is the effect on throughput and latency?
- If the load on the system changes, how well does the system adapt?

In this thesis, we focused our experiments on two of these features: **failure recovery** and **scalability**.

5.2.1 Test workloads

Three sample workloads were used in the test cases for each study (unless otherwise indicated). Each workload produced approximately 10,000 messages to Kafka per second, uniformly distributed across 10 keys (or zipfian as indicated).

1. *Stateless transformation:* a `filter` operation with 0.5 drop probability, followed by a `map` operation with `size_ratio = 2` (doubles the message size).

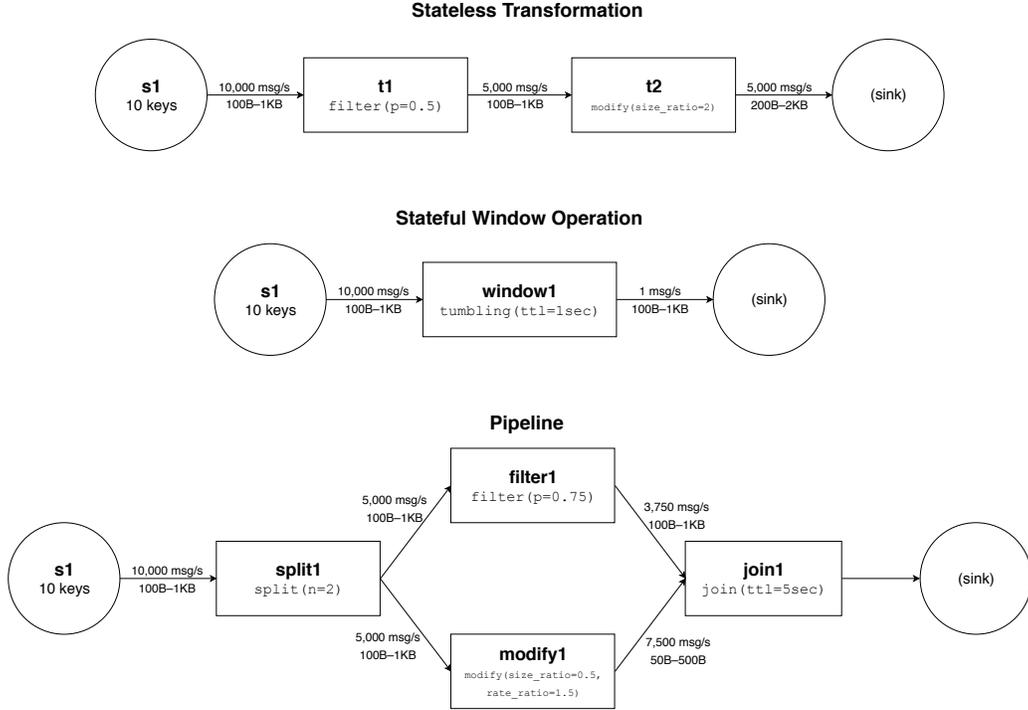


Figure 5.1: Test Workloads Used in Evaluation

2. *Stateful window*: a single tumbling window operator that emits the count of messages received in every 1 second time window.
3. *Stateful join pipeline*: a **split** operation with two output streams: the first is a **filter** operation with 0.75 drop probability, the second is a **map** operation that produces messages with half the size at $rate_ratio = 1.5$. Finally, the output of the **filter** and **map** are joined by key with a time-to-live of 5 seconds.

These three workloads are illustrated in Figure 5.1.

5.2.2 Failure recovery

In this study, we want to understand how the stream processing system reacts to the failure of individual components. We investigate the impact on the overall system’s performance during the failure and how long it takes for the system to recover to normal operation.

To understand the failure recovery response of the target system, we identified 2 metrics—throughput and recovery time. Both of these are compared

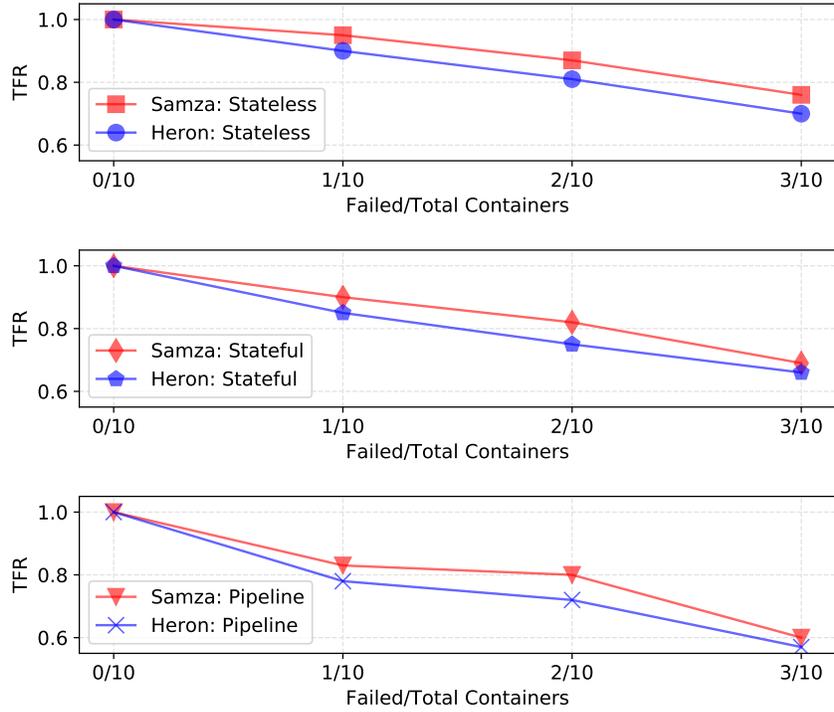


Figure 5.2: Size of Failure vs. Throughput under 3 Workloads

against the number of containers failed in the following tests.

5.2.2.1 Size of failure vs. System throughput

This test measures the impact of various failure sizes on the performance of the system, in this case the average system throughput. We plot TFR (Throughput Failure Ratio) on the y-axis which we define as:

$$TFR = \frac{T_f}{T_{nf}}$$

where T_f is messages processed per second during failure recovery and T_{nf} is average number of messages processed per second while there are no failures in the system.

Using this ratio we can compare the impact of failures on the throughput of the system. We conducted tests for the 3 different types of workloads described in §5.2.1 to understand how impact differs when using stateful vs.

stateless operators. Each workload was distributed across 10 containers.

The results for Samza and Heron are shown in Figure 5.2. We can observe that the impact on throughput is higher with larger failures, and it is also affected by the size of the workload. The simple Stateless and Stateful workloads have higher *TFR* indicating that the failure had lower impact on throughput than it did in the Pipeline workload. Heron shows slightly lower throughput ratios when compared to Samza because Heron employs the **backpressure** mechanism that was described in §2.3.2. Upon failure, the upstream operators are slowed down so as to not flood the network unnecessarily until recovery is complete. The backpressure mechanism prevents loss of messages when using the “at least once” delivery mechanism. Samza, on the other hand, uses parallel operators and replays the messages to them when the failed containers have recovered.

5.2.2.2 Size of failure vs. System recovery time

This test measures how long each system took to recover from various sizes of failures, i.e., the number of containers that failed. We measure the time taken for each failed container to recover and resume with the same throughput as before the failure. Since some target systems implement state recovery, they are expected to be impacted more heavily by failures, perhaps by prolonging the recovery time. The recovery time itself is measured by the time delta between when the failure was triggered and when the system resumes with the same throughput as before. We also factor into this time delta any indication from the system that state rebuild is complete, if applicable.

The results for Samza and Heron are shown in Figure 5.3. Each workload was run with 10 containers and between 1 and 3 containers were failed, chosen at random. We can see that recovery time does not increase much as size of the failures is increased. This is because each container recovers independent of the other failures. Additionally, because the stateful and pipeline workloads have to also rebuild the previous state, they take longer to recover than the stateless workload. Samza implements a comprehensive state recovery mechanism that rebuilds each container’s operator state from a backup log to ensure no loss of information. Heron, on the other hand, does not currently implement a state recovery mechanism, so state information is lost upon container failure. Thus, Heron gains on recovery time as its simpler

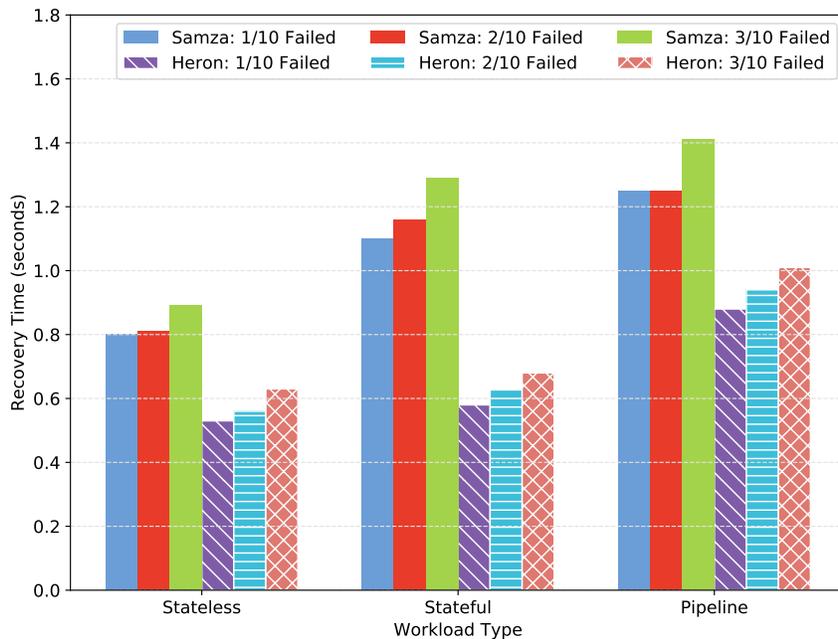


Figure 5.3: Size of Failure vs. System Recovery Time under 3 Workloads

recovery protocol is comparatively quicker and it works by only reversing upstream backpressure changes. Samza’s better state recovery management incurs minor time costs that result in slightly higher recovery times.

5.2.3 Scalability

Scalability tests measure how much performance changes when more containers are added to the system. In the following tests we want to investigate how the throughput and latency are impacted in each system. The latency metric here is measuring the end-to-end time delta from when each message was produced to when it reaches the sink after being processed completely.

5.2.3.1 Number of containers vs. Maximum throughput & Average processing latency

In this test, we measure the maximum throughput that the system achieves and the average processing latency at that throughput as we increase the number of nodes (containers) in the system. We use a heavier Pipeline work-

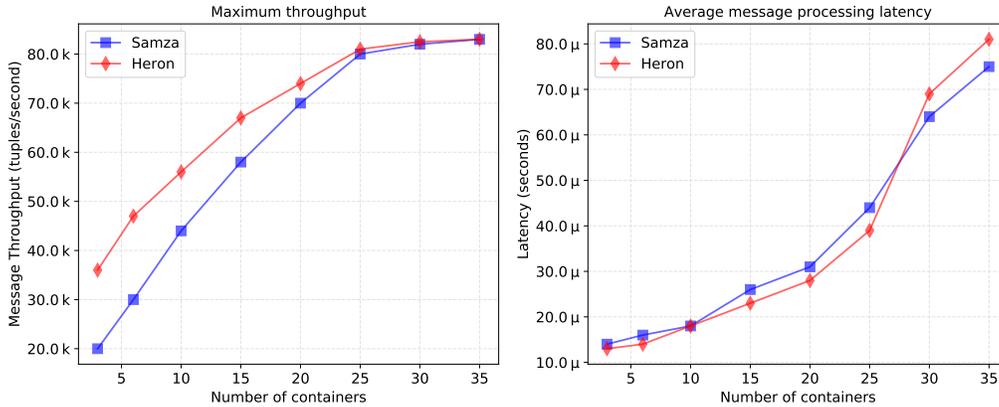


Figure 5.4: Number of containers vs. Maximum throughput & Average Processing Latency

load with 100 keys to test the scalability limits. The results for Samza and Heron are shown in Figure 5.4. As we increase the number of containers, the overall throughput of the system increases because of the increased parallelism in processing of messages. Average message processing latency also increases linearly until the network cap is hit, at which point there is a greater increase in latency.

Based on the results we see that Samza achieves slightly lower throughput than Heron. These tests were conducted on the base configurations of either systems, and so this difference can be accounted for by adjusting the specific system configuration. Both systems are capped by the network bandwidth when there are a large number containers participating. Latency is approximately the same for both systems, and increases as more containers are added because there are more messages propagating in the network.

5.3 Application-based analysis

In application-based analysis, we describe how we can use Finch to create synthetic workloads that simulate any real-world workload. Using Finch’s operator descriptions, we can build data flows that closely match specific streaming applications like grep, word-count, ad click-through rate, etc. In our application based analysis, we show how we create a workload that simulates the popular *word count* application, and investigate its performance in Samza and Heron.

5.3.1 Word-count workload description

The word-count application is used to get the frequency of each word in a document or corpus. In a naïve solution, the program maintains a hash map in memory that maps a given word to its frequency. The program iterates through the document word-by-word and increment the frequency in the hash map for each word encountered.

To convert word-count into a stream-based abstraction, we will define the source stream and the operators to get the same output. The source incrementally emits successive lines of the document per message. The order of operations will be as below:

1. Split the incoming sentence into a list of words, and emit each word separately
2. Upon receiving a word, update local state by incrementing the frequency of the received word. The local state maintained here is similar to the frequency hash map.
3. Once the source stream is exhausted, the local state of each operator is combined to create the global word-count of the document.

This methodology depends on the fact that each time the word X is emitted from the second step, when the next word X' that satisfies $X' = X$ is emitted, X' is sent to the same partition of the next operator that X was sent to. This can be achieved by consistent hashing of the emitted word. If this functionality is unavailable, an explicit “group by” operation can be used.

5.3.2 Word-count analysis

We analyze the simulated word-count workload on Samza and Heron, and measure the average processing latency of each message at various sentence emission rates. The results are shown in Figure 5.5. From this graph, we see that Samza and Heron perform very similarly in terms of word processing latency. As we increase the rate at which new sentences are emitted from the source, the latency also increases. This is because each operator buffers messages that it has received but not yet processed. Increasing the rate at

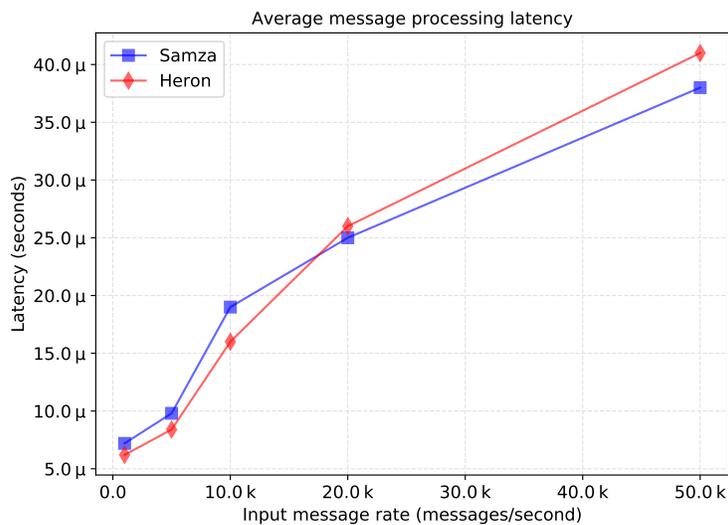


Figure 5.5: Average Processing Latency for Word-count

which messages come in mean that each message on average is in the queue for longer.

5.4 Summary of experimental results

From the analysis we have presented, we can see that both Heron and Samza achieve comparable performances in various types of feature and application workloads. The major differences are seen in how failures are handled.

Heron uses backpressure to slow down or stop upstream bolts until the failed operators have recovered. This results in a much lower throughput during operator failure in comparison to Samza, which parallelizes the operators using key-based distribution of messages. On the other hand, once the failed containers have recovered, Samza triggers the state recovery mechanism to rebuild each failed operator’s state from the changelog. This ensures that minimum information is lost due to the failure. The trade-off that Samza takes here is that the recovery time upon failures is slightly increased.

Finally, we conclude by restating that each of these workload performance metrics can be tuned by adjusting the framework’s individual tunables or parameters. These specifics require more knowledge about the system being

tested but this is out of the scope of what Finch can abstract. Finch targets users who would like first-hand empirical analysis of stream processing frameworks and hence focuses on exposing the base performance of each system. Experienced users can build upon this and run Finch on their clusters to test what system parameters to tune or, in general, run benchmarking workloads against their clusters using a host of target frameworks.

Chapter 6

Conclusion

We have presented Finch, a new benchmarking tool for stream processing systems that is flexible and can be used to benchmark both features and applications of various frameworks. We described the various characteristics of stream processing frameworks and compared their designs and trade-offs. Then, using Finch, we showed how these differences and trade-offs translate to system performance, both in terms of features like fault-tolerance and scalability, and in terms of applications that simulate real-world pipelines.

As future work, the first target would be to integrate Finch with other stream processing frameworks like Spark Streaming, Flink, Storm, etc. With these, the breadth of comparison that can be obtained using Finch would significantly increase. Another future target could be to generalize the metrics collected from each Finch module. Each stream processing framework has very specific performance metrics that are emitted, and similarly very specific methods for reporting them. It is thus challenging to analyze metrics from different systems in a homogeneous fashion without doing additional work to combine them. Finch's usability will considerably increase if these issues are also addressed. Finally, we can add more real-world workloads and analyze the other features discussed in §5.2 like state, parallelism, multi-tenancy, and dynamism. This would showcase Finch's ability to benchmark multiple different components of stream processing frameworks.

Finch GitHub Repository:

<https://github.com/srujun/finch-benchmark>

References

- [1] “4 V’s of Big Data.” [Online]. Available: <http://www.ibmbigdatahub.com/infographic/four-vs-big-data>
- [2] “Apache Hadoop.” [Online]. Available: <https://hadoop.apache.org/>
- [3] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [4] J. Dean and S. Ghemawat, “MapReduce: a flexible data processing tool,” *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.
- [5] A. Kejariwal, S. Kulkarni, and K. Ramasamy, “Real time analytics: algorithms and systems,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 2040–2041, 2015.
- [6] S. Z. Sbz, S. Zdonik, M. Stonebraker, M. Cherniack, U. C. Etintemel, M. Balazinska, and H. Balakrishnan, “The Aurora and Medusa projects,” in *IEEE Data Engineering Bulletin*. Citeseer, 2003.
- [7] The STREAM Group, “STREAM: The Stanford stream data manager,” Stanford InfoLab, Technical Report 2003-21, 2003. [Online]. Available: <http://ilpubs.stanford.edu:8090/583/>
- [8] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: A new model and architecture for data stream management,” *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.
- [9] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin et al., “Aurora: A data stream management system,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 666–666.
- [10] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina et al., “The design of the Borealis stream processing engine,” in *CIDR*, vol. 5, no. 2005, 2005, pp. 277–289.

- [11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.
- [12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “MillWheel: Fault-tolerant stream processing at internet scale,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.
- [13] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, “Photon: Fault-tolerant and scalable joining of continuous data streams,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 577–588.
- [14] “Apache Pulsar.” [Online]. Available: <https://pulsar.incubator.apache.org/>
- [15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache Flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
- [16] “Apache Apex.” [Online]. Available: <https://apex.apache.org/>
- [17] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringham, I. Gupta, and R. H. Campbell, “Samza: stateful scalable stream processing at LinkedIn,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.
- [18] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., “Storm @ Twitter,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 147–156.
- [19] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, “Twitter Heron: Stream processing at scale,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.
- [20] “Spark Streaming.” [Online]. Available: <https://spark.apache.org/streaming/>
- [21] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters.” *HotCloud*, vol. 12, pp. 10–10, 2012.

- [22] R. Lu, G. Wu, B. Xie, and J. Hu, “Stream bench: Towards benchmarking modern distributed stream computing frameworks,” in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014, pp. 69–78.
- [23] “What is Stream Processing?” [Online]. Available: <https://data-artisans.com/what-is-stream-processing>
- [24] G. Cormode, “Sketch techniques for approximate query processing,” *Foundations and Trends in Databases*. NOW Publishers, 2011.
- [25] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.
- [26] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, “Parallel stream processing against workload skewness and variance,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 15–26.
- [27] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, “Elastic scaling of data parallel operators in stream processing,” in *IEEE International Symposium on Parallel & Distributed Processing, 2009*. IEEE, 2009, pp. 1–12.
- [28] W. Wingerath, “Scalable stream processing: A survey of Storm, Samza, Spark and Flink,” Sep 2016. [Online]. Available: <https://medium.baqend.com/real-time-stream-processors-a-survey-and-decision-guidance-6d248f692056>
- [29] “Redis.” [Online]. Available: <https://redis.io/>
- [30] J. Kreps, N. Narkhede, J. Rao et al., “Kafka: A distributed messaging system for log processing,” in *Proceedings of the NetDB*, 2011, pp. 1–7.
- [31] “RabbitMQ.” [Online]. Available: <https://www.rabbitmq.com/>
- [32] “ZeroMQ.” [Online]. Available: <https://www.zeromq.org/>
- [33] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth et al., “Apache Hadoop YARN: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
- [34] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, “Hierarchical multiprocessor cpu reservations for the linux kernel,” in *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-time Applications (OSPERT 2009), Dublin, Ireland, 2009*, pp. 15–22.

- [35] “The Heron streamlet API.” [Online]. Available: <https://apache.github.io/incubator-heron/docs/concepts/streamlet-api/>
- [36] D. Borthakur et al., “HDFS architecture guide,” *Hadoop Apache Project*, vol. 53, 2008.
- [37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
- [38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.
- [39] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154.
- [40] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM, 2009, pp. 165–178.
- [41] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai, “Hibench: A representative and comprehensive Hadoop benchmark suite,” in *Proc. ICDE Workshops*, 2010.
- [42] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, “The case for evaluating MapReduce performance using workload suites,” in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 390–399.
- [43] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng et al., “Benchmarking streaming computation engines: Storm, Flink and Spark Streaming,” in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1789–1792.
- [44] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, “A performance comparison of open-source stream processing platforms,” in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.
- [45] A. Lobato, M. A. Lopez, and O. Duarte, “An accurate threat detection system through real-time stream processing,” *Grupo de Teleinformática e Automação (GTA), Universidade Federal do Rio de Janeiro (UFRJ), Tech. Rep. GTA-16-08*, 2016.

- [46] S. Qian, G. Wu, J. Huang, and T. Das, “Benchmarking modern distributed streaming platforms,” in *Industrial Technology (ICIT), 2016 IEEE International Conference on*. IEEE, 2016, pp. 592–598.
- [47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.
- [48] D. Bernstein, “Containers and cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [49] “Apache Kafka API.” [Online]. Available: <https://kafka.apache.org/documentation/#api>
- [50] “Apache Commons Math.” [Online]. Available: <https://commons.apache.org/proper/commons-math/>
- [51] “Graphs, explained.” [Online]. Available: <https://github.com/google/guava/wiki/GraphsExplained>
- [52] “Package java.util.stream.” [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>
- [53] “HashiCorp Nomad.” [Online]. Available: <https://www.nomadproject.io/>
- [54] “Apache Aurora.” [Online]. Available: <https://aurora.apache.org/>
- [55] “InfluxDB — the time series database in the TICK Stack.” [Online]. Available: <https://www.influxdata.com/time-series-platform/influxdb/>