Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka

M. Haseeb Javed
The Ohio State University
Columbus, Ohio
javed.19@osu.edu

Xiaoyi Lu
The Ohio State University
Columbus, Ohio
lu.932@osu.edu

Dhabaleswar K. (DK) Panda
The Ohio State University
Columbus, Ohio
panda.2@osu.edu

ABSTRACT

In recent years there has been a surge in applications focusing on streaming data to generate insights in real-time. Both academia, as well as industry, have tried to address this use case by developing a variety of Stream Processing Engines (SPEs) with a diverse feature set. On the other hand, Big Data applications have started to make use of High-Performance Computing (HPC) which possess superior memory, I/O, and networking resources compared to typical Big Data clusters. Recent studies evaluating the performance of SPEs have focused on commodity clusters. However, exhaustive studies need to be performed to profile individual stages of a stream processing pipeline and how best to optimize each of these stages to best leverage the resources provided by HPC clusters. To address this issue, we profile the performance of a big data streaming pipeline using Apache Flink as the SPE and Apache Kafka as the intermediate message queue. We break the streaming pipeline into two distinct phases and evaluate percentile latencies for two different networks, namely 40GbE and InfiniBand EDR (100Gbps), to determine if a typical streaming application is network intensive enough to benefit from a faster interconnect. Moreover, we explore whether the volume of input data stream has any effect on the latency characteristics of the streaming pipeline, and if so how does it compare for different stages in the streaming pipeline and different network interconnects. Our experiments show an increase of over 10x in 98 percentile latency when input stream volume is increased from 128MB/s to 256MB/s. Moreover, we find the intermediate stages of the stream pipeline to be a significant contributor to the overall latency of the system.

KEYWORDS

Stream Processing; Big Data; Real-Time Processing; Message Queue; Profiling; HPC Clusters

1 INTRODUCTION

Since the last decade, there has been an exponential increase in the volume of data and the rate at which it is produced caused by popular social media networks, Internet of Things (IoT) applications etc. Even the very definition of the so-called term "Big Data" has started to lose its significance, as the amount of data that was considered to be "big" is now the same amount of data which small IT and data warehouse companies crunch every day. However, more so than ever is there a need to process, analyze, and generate insights from data in real time i.e. before it gets obsolete. Businesses are focusing more and more to set up infrastructures that would help them gain valuable insights about their customers in real-time so that they can satisfy the changing customer needs as soon as possible and thus gain a competitive advantage over their rivals. Common examples of businesses which are heavily dependent on analyzing data in real-time include online advertisements, stock markets etc. To address this problem of real-time data processing, several systems, both proprietary and open source, have been proposed by academia as well as the industry. A cursory analysis of the streaming domain indicates that there is a plethora of SPEs available with a large variety of features. Therefore, it is not hard to imagine how complicated the simple task of selecting an appropriate SPE for a simple use-case would be for a novice user. Most businesses that deal with data processing already have some notion of a data processing pipeline already set up, which in most cases has been Hadoop [35] powered batch processing. When these businesses expand to the domain of real-time data processing, their main goal is to adopt a framework that would make the maximum use of the existing batch processing infrastructure they already have had set up. However, integrating an SPE that performs real-time data processing with an already established batch processing pipeline thus executing them in a synchronized fashion is no easy task. This task is further complicated by the various intermediate layers involved to set up a stream processing pipeline which usually involves some message queue such as Apache Kafka [22], Flume [15], and DistributedLog [13] which acts as an intermediate buffer between the stream source and the SPE.

Moreover, there has been a dearth of evaluative studies comparing different SPEs on various performance metrics of throughput and latency. In particular, no work has been done to evaluate potential performance gains of running a full-blown stream processing pipeline on an HPC cluster. The low latency network and storage commonplace in HPC clusters could be utilized to greatly improve the performance of a stream processing pipeline. However, this domain is open for exploration and calls for exhaustive studies to be done to arrive at definitive conclusions.

From the above discussion, we can generalize that in order to best leverage HPC clusters for big data stream processing, the following issues need to be addressed:

- What are the various bottlenecks in a typical stream processing pipeline?

BDCAT'17, December 5–8, 2017, Austin, Texas, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5549-0/17/12…$15.00
https://doi.org/10.1145/3148055.3148068

- What high performance features provided by HPC clusters can be used to remove these bottlenecks?

Stream Processing Engines has been evaluated by the academia previously. However, performance evaluation of such frameworks on High-Performance Interconnects such as InfiniBand, has not been systematically carried out. We summarize existing studies in this area in Table 1.

	Pipeline Breakdown	Role of Middleware	InfiniBand
[11]	×	×	×
[32]	×	×	×
[20]	×	×	×
This paper	√	√	√

Table 1: Comparison with existing studies

The study [11] evaluates Storm, Spark Streaming, and Flink in terms of 99th percentile latency and throughput comparison. The study [32] compares Spark Streaming, Storm, and Samza on standardized micro-benchmarks such as grep, projection etc. The study [20] performs an exhaustive performance analysis of streaming as well as batch frameworks. However, none of these studies makes an attempt to study which phase of the streaming pipeline is the most performance intensive, what is the contribution of Message Oriented Middleware to this pipeline and what is the behavior of such systems when run on High-Performance Networks. In this paper, we present an in-depth study of all these aspects of a stream processing stack and also provide some other interesting insights.

To begin exploring these challenges, we first give an overview of the prevalent stream processing infrastructure. Next, we outline the various intermediate stages the messages in the pipeline go through from when they are first emitted by the stream source to when they are eventually processed by the SPE.

Furthermore, we evaluate the end-to-end latency characteristics of a streaming pipeline constructed with Apache Kafka and Apache Flink. We run tests with varying stream throughput, on two common interconnects available in HPC clusters i.e. 40Gbps Ethernet and EDR (100Gbps) InfiniBand.

Lastly, we evaluate the individual contributions of different stages in a streaming pipeline to its overall latency. We also introduce a volume parameter for input data stream and evaluate how changes to this parameter affect the latency characteristics of the streaming pipeline. All of these benchmarks are run over both TCP/IP over Ethernet and IPoIB [12] to evaluate if a faster interconnect could benefit the overall stream processing pipeline and the particular characteristics of a stream processing pipeline which make these differences more pronounced.

We want to deconstruct the Stream Processing pipeline to better understand its characteristics and in doing so bridge the gap between Big Data Streaming and HPC for them both to benefit from their respective expertise. We believe this paper will help people in both industry and academia to develop new tools to help Big Data Streaming extract the most use out of the "High-Performance" features provided by HPC clusters.

To summarize, this paper makes the following key contributions:

- Deconstruct the streaming pipeline in two distinct stages to evaluate the effect on overall performance due to potential bottlenecks in them.
- Evaluate the performance of streaming pipeline in response to varying input stream volume.
- Orchestrate these tests on networks with varying bandwidth to determine if faster interconnects provided by HPC clusters can benefit Big Data Streaming applications.

The paper is organized as follows. Section 2 covers background knowledge. Section 3 reviews the state-of-the-art in the domain of SPEs. Section 4 exhibits and presents performance characterization results. Related work is summarized in Section 5. Finally, concluding remarks and future direction for continued work appears in Section 6.

2 BACKGROUND

In this section, we provide some background knowledge of SPEs, their history, and how they have evolved into their current state. We also provide a brief overview of HPC so that later we can motivate leveraging the advanced features provided by these clusters to improve a generic stream processing pipeline.

2.1 Data Stream

Data Stream is a continuous pipeline of unbounded data, the termination point of which is not predefined. Usually, the stream consists of multiple key-value pair records. The data stream can come in at different rates and the message size of the stream may also vary, depending on the application. The popularity of social media, Internet of Things and other industries heavily reliant on real-time processing, have made streaming data even more important. Other examples of industries heavily reliant on stream data processing include fraud detection, high-frequency trading, network monitoring, intelligence and surveillance etc.

2.2 Stream Processing Engines (SPEs)

The concept of processing and generating insights from data that is "in motion" is not new. Stream processing systems, in one form or the other, have been present in academia as well as industry for quite a while. From their early development to the present state, SPEs can be broadly categorized in three generations.

The first generation of such systems mainly consisted of database systems and rule engines. These systems allowed users to define actions that would be performed when certain conditions were met. These conditions would generally be triggered by new data being fed into the system. The actions that were subsequently performed could modify the internal state of the system or in some cases trigger other actions. Postgres [36] is a well known example of systems belonging to this generation.

Second generation of such systems were specifically designed to process data streams and hence had more intuitive primitives for performing operations on streams compared to the earlier generations. STREAM [5] implemented streaming semantics on SQL data. Aurora [2] used a Directed Acyclic Graph (DAG) to represent streaming computation but was limited to a single node whereas Borealis [1] implemented it is a distributed cluster-based system.

Third generation SPEs were the first to process a distributed stream source. Apache Storm [37] implemented a DAG of both the stream source and the streaming computation to run it in a distributed manner. Apache Spark [39] used a micro-batch technique using which it first collects the input stream in very small batches before processing it. Apache Flink [7] is based on the DataFlow [4] architecture for processing data whereby both bounded and unbounded streams of data are processed in the same manner by the underlying core. Other frameworks from this generation include S4 [31], MillWheel [3], Amazon Kinesis [23], and IBM Inphoshere [6]

2.3 Message Oriented Middleware (MOMs)

SPEs, like any other data processing framework, can have faults. This is not a major concern in the case where data that is being processed resides in persistent storage. If a crash occurs, all other factors held constant, the data can be read from the disk and be processed again. However, due to the inherent ephemeral nature of data streams, the durability of data becomes a major issue. If the SPE consumes a message from the stream source but crashes before completely processing it, that message is all but lost. This is because most, if not all, stream sources have very difficult semantics for replaying data.

In order to address this issue, the concept of Message Oriented Middleware is introduced. MOMs in their most general form, are publish-subscribe systems which basically act as an intermediate layer between the stream source, and the SPE. The stream source publishes messages to the MOM and an SPE can subscribe and retrieve messages from it. With the presence of MOM in the streaming pipeline, the stream source and SPE become decoupled. Each can read/write data at their own respective rates. Moreover, as the messaging queue itself is distributed and fault tolerant, it can not only retain data stream of high throughput but also replay messages in case a fault occurs at the SPE.

Apache Kafka [22] is one such system commonly used in big data streaming pipelines. Apache Kafka stores a stream of messages in queues called topics. Each message in Apache Kafka consists of a key-value pair combined with a unique timestamp. Both the producers and consumers of data declare the topics they want to interact with. Unique timestamps associated with each message are used by the consumer, an SPE in our case, to replay messages in case the message needs to be reprocessed. Apache Kafka can act as receiver and provider of data for a vide variety of systems. Due to this heterogeneity of systems interacting with Apache Kafka, particularly for consumers, Apache Kafka adopts a pull model whereby consumers of data have to "pull" messages from Apache Kafka servers, called brokers. This allows for greater flexibility on consumer side to process data at whatever rate and semantics it deems suitable. Figure 1 depicts how Apache Kafka and Apache Flink, as examples of the messaging middleware and processing engine, combine together to form a big data streaming pipeline.

Other examples of MOMs include Flume [15] and DistributedLog [13].

2.4 High-Performance Computing Clusters

High-Performance Computing has typically utilized superior hardware to support and grow the field. Consequently, HPC clusters usually consist of hardware that has far superior performance characteristics than their counterparts in typical Bid Data clusters. Other

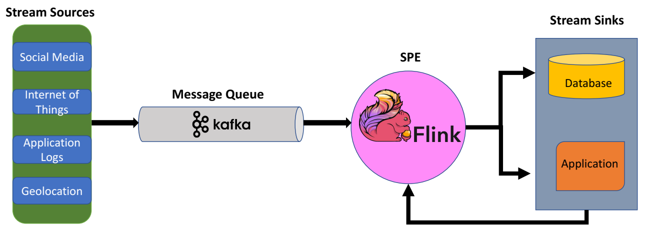


Figure 1: High level view of a streaming data processing pipeline

than faster processing and storage, one important aspect that defines HPC clusters is low latency network and interconnects. Technologies such as InfiniBand [19] have improved network I/O latency and bandwidth significantly compared to their Ethernet counterpart. Moreover, socket based applications can also be run with InfiniBand hardware using the IP over InfiniBand (IPoIB) protocol [12] without any modifications.

High-Performance Message Passing Interface (MPI) libraries such as MVAPICH [30] have utilized these hardware-based features to greatly improve the performance of user facing applications. Recently, however, RDMA aware versions of Hadoop [26], Spark [27], Memcached [21] have shown that Big Data technologies can also exhibit vast performance improvements by utilizing the features provided by the these fast networks. We feel that stream processing is another domain which can heavily benefit from these features, as minimizing latency is one of the primary aims of an SPE and faster interconnects provided in HPC clusters can help achieve this goal.

3 OVERVIEW OF STATE OF THE ART

In order to better understand the landscape of state of the art for Stream Processing Engines, we carry out an exhaustive survey. We study the following frameworks to determine how each framework chooses to address the main challenges involved in Big Data stream processing. Through this survey, we are better able to identify what characteristics are essential for a distributed stream process engine and what are the different ways that popular systems implement them.

3.1 General Purpose SPEs

Storm [37] is one of the first frameworks to address the problem of real-time distributed big data processing. Each store cluster consists of Nimbus node(s) and Supervisor nodes(s). Each job submitted to Storm is described as a Topology, representing stream sources called Spouts and transformations which are applied to the stream, called Bolts. User submits a topology to Nimbus which then allocates a subset of the topology to each Supervisor node in the cluster. Storm achieves fault tolerance using Zookeeper. Both Nimbus and Supervisors in a Storm topology are stateless, so in order to coordinate between them and to handle failures Zookeeper [18] is used. Every node in the cluster writes its state to Zookeeper. In the case of a node failure, it can then restart and obtain its state from Zookeeper to resume its role in the topology. Storm only offers at-least once message delivery semantics, which implies that messages in the topology may be delivered more than once and therefore it is up to the application to handle duplicates.

Spark Streaming [39] is a stream processing layer on top of the core Spark data processing engine. The basic unit of processing in Spark Streaming is a Discretized Stream (D-Stream) which is an abstraction for a collection of RDDs [38] to be processed together. Unlike Storm however, Spark follows a micro-batch policy for stream processing whereby streaming data coming into the systems is first collected in small batches called D-Streams before being processed by the system. This micro-batch policy inherently adds significant latency to the data pipeline as even though messages arrive in the system in real-time, they are not processed until the batch duration is reached. However, Spark Streaming offers exactly-one delivery semantics and therefore application does not have to worry about duplicates. Also having the same underlying core means that streaming logic can easily be incorporated into other Spark modules such as batch, graph, or machine learning.

Apache Flink [7] is a distributed data processing engine particularly tuned for cyclic workflows by performing iterative transformations on collections. It provides a uniform architecture for processing both batch and real-time data by treating them both as streams. Apache Flink has an underlying layer which provides optimizations for various join, shuffle and partition operations, which result in even faster data processing.

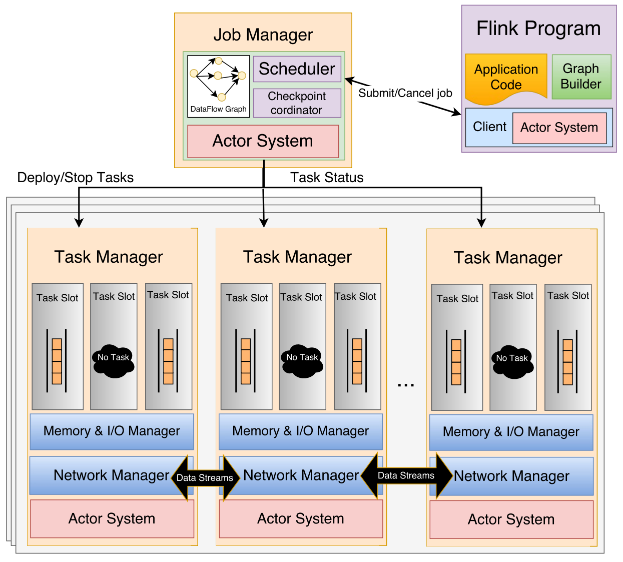


Figure 2: Architectural Overview of Apache Flink

Figure 2 describes the internal components of Flink and how they interact with each other. A program written in Flink is submitted to the Job Manager which acts as master of the entire Flink cluster. The Job Manager decomposes the job into smaller components and schedules them on various Task Managers in the cluster. These Task Managers are responsible for executing the task and sending the results back to the Job Manager while also communicating with other Task Managers in the process.

Flink, unlike Spark, also supports iterations in data pipeline natively meaning that data produced from an iteration can just be fed back to the pipeline for subsequent iterations. This feature makes Apache Flink highly suitable for iterative and machine learning workloads. For fault taulerance, Apache Flink utilizes a specialized program state checkpointing system [8] which, according to its creators, is inspired from the Chandy-Lamport algorithm [10].

Samza [24] is another distributed stream processing engine built on top of Apache Kafka. Samza consumes stream from a Kafka producer, processes the messages in an event loop and produces desired results. It then pushes them to Kafka consumer which then provides them to the client. Samza maintains the intermediate states of the data stream by saving the processing results at each step in RocksDB [33] and a replicated copy of it in Kafka. Streams from more than one input can be joined for processing using window join, tabular join etc. Data can be made available in log compacted stream, which can act as a local replica with low latency. However, being tightly coupled with Apache Kafka leads to some drawbacks. In the case where a failed Samza instance restarts, it will consume the message with the largest offset. In this scenario, the processing done on the message after the log increment and before the crash is lost and hence must be done again. Also, Samza uses one message at-a-time programming model which is not ideal for automatic optimizations.

Heron [25] is a streaming framework used in production at Twitter and is heavily inspired by Storm. Heron jobs, called Topologies, also operate on a Directed Acyclic Graph (DAG) model consisting of Spouts and Bolts. It offers full compatibility with Storm while claiming to be 2-5 times more efficient. Heron executes Spouts and Bolts in isolation for the purpose of debuggability. It applies strict restrictions on resource consumption such that resources used by a topology should not exceed beyond the amount allocated during execution. If there is an attempt to consume more resources, the container will be throttled, leading to a slowdown of the topology. When a topology slows down due to slow containers, a back pressure mechanism is used to allow the topology to readjust its speed accordingly to minimize data loss. A client writes topologies using Heron API which are sent to a scheduler. The scheduler provides the required number of resources and spawns containers on different nodes. Master container handles the topology and sends its location based on a host-port pair to an ephemeral Zookeeper instance that handles the coordination among other containers. All slave containers run stream managers, metric managers, and instances of Spouts/Bolts. Stream manager controls the flow of data between the entire topology. A Heron instance on these containers runs the processing logic of Spouts or Bolts. The gateway thread in the instances communicates with the stream manager to send and receive messages and passes them to the task-execution thread. The task-execution thread then applies the processing logic.

Trill [9] is another framework developed at Microsoft which has similarities with both Storm and Spark Streaming. Trill delegates the trade-off between throughput and latency as a decision to be made by the user. For every application, the user has to specify a latency threshold and Trill will collect input stream in mini batches of a size that allows it to operate at the requirements specified by the user. This is a similar approach adopted by other micro-batch based streaming systems such as Spark. However, Trill is unique in the sense that batching in Trill is not temporal which allows for identical results irrespective of the size of the batch or velocity of the input stream. Trill achieves these requirements using a hybrid system architecture which exposes a latency-throughput tradeoff. Similar to Storm, Trill runs jobs as a DAG of streaming operators and within each operation, it uses columnar data organization to make processing more efficient.

3.2 Domain-Specific SPEs

Connected Streaming Analytics (CSA) [34] is a system developed by Cisco specifically designed for streaming data generated by IoT applications. IoT applications produce large amounts of heterogeneous data, the sources of which are quite dispersed geographically. So traditional SPEs do not suffice for this use case as just bringing the data to computation nodes itself would incur a huge latency penalty. CSA tries to address this issue by bringing the computation nearer to the data. Under CSA, an intelligence layer is implemented inside network edges i.e routers and switches across the network thus data is processed with minimal delay. This architecture allows for the data processing to scale with the size of the network. CSA also provides a Continuous Query language to provide support for SQL-like queries on unbounded streams of data.

FUGU [16] is a system specifically designed with a focus on elasticity, meaning the framework should vary the number of nodes in the system in response to a sudden spike in data processing needs by allocating more resources. It also monitors the workload so as to reduce the number of unused resources in the system. It implements a scaling policy which guarantees to provide latency restrictions specified by the user. Internally, it also models data stream as DAG of stream sources and transformations to be applied on them. It has a centralized management system which is responsible for allocating just the optimum amount of resources the system while also attempting to minimize the number of latency peaks.

3.3 Lambda Architecture

Lambda architecture [29] uses both batch and stream processing in tandem to address the constraints of latency, throughput, and fault tolerance in a single system. The architecture consists of three layers; a batch processing layer is used to process large amounts of stored historical data. It allows for accurate results to be produced as there are no stringent latency constraints. It allows for errors that were generated in other layers to be rectified and the output to be persisted in a database for long-term storage. Batch layer produces views corresponding to query functions which are then used by the serving layer to give a holistic picture of the entire dataset. Apache Hadoop [35] is the standard framework used in this layer.

The Speed layer is used to perform real-time computations on data coming into system. This layer may compromise on the accuracy of data in order to provide results with minimum latency. Speed layer will provide insights based on the data that was consumed by the system but not yet been processed by the Batch layer. However, the Batch layer would eventually override the results of Speed layer as the long-term correctness guarantees become important. SPEs such as Storm, Spark, and Flink are typically used in this layer.

Serving layer takes in views generated by both Speed and Batch layer and runs ad-hoc queries on them to provide users with requested results. It indexes the results produced by the other two layers to efficiently respond to any new query that the user might generate. ElephantDB [14] is a commonly used tool for this layer.

3.4 Key Observations

Through our analysis of the current state-of-the-art of stream processing we realized that although high throughput, scalability, fault tolerance are important factors while designing stream processing engines, low latency happens to be the single most desirable quality for them. Considering that these systems operate on real-time data, it is only natural that such stringent requirements of latency have been established.

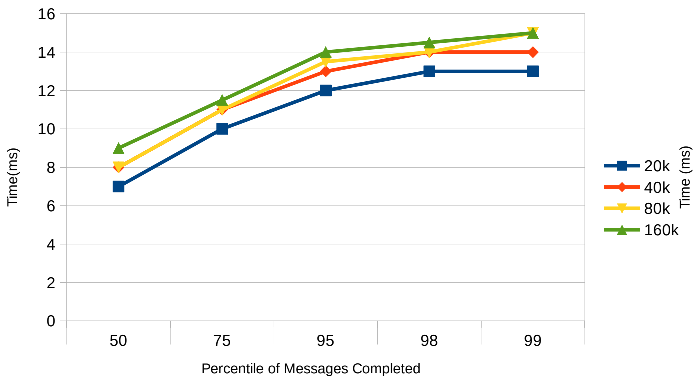
We use this observation as the bedrock of the characterization we perform. We construct a simplistic streaming pipeline and attempt to dissect the contributions of individual stages to the end-to-end latency of the entire pipeline. Moreover, we evaluate the effects on the performance of the pipeline in response to changes in the characteristics of the input stream while also by varying the underlying network the pipeline executes on. Through these experiments, we finally conclude on how tools provided by HPC clusters can be best leveraged to benefit an arbitrary streaming pipeline.

4 PERFORMANCE EVALUATION

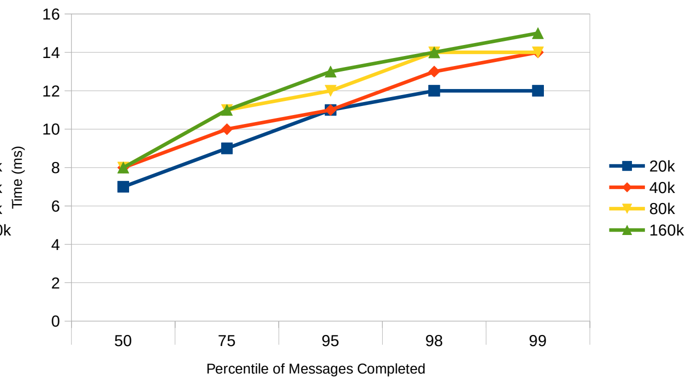
There are 4Vs which can be used to generalize every form of Big Data Processing, namely Velocity, Volume, Variety, and Veracity. However, to analyze stream processing from a High-Performance perspective, Velocity and Volume are of particular importance.

Some studies exist in the literature which evaluate the performance of SPEs in terms of latency with respect to Velocity [11] [32]. However, we take a different approach where we break down the streaming pipeline into two major stages to determine which part of the pipeline is network intensive and which part is compute intensive. Then we show how percentile latencies for message processing vary in response to changes in not only input message rate i.e varying velocity but also input data size per unit time i.e volume.

Any streaming pipeline can be broken down into three distinct phases. In the first phase, a stream source would periodically write records into a message queue. The rate at which data is input to the messaging queue is usually not a parameter that can be controlled by the user, therefore a message queue should not only be able to store large amounts of data, but also ensure reliable delivery of the data to an SPE. During the second phase, depending on the architecture of the message queue, it will either emit data to the SPE, in which case it would be following a Push Model. Otherwise, if it obeys the Pull Model, then an SPE will fetch records from the message queue, relative to the last message offset it processed. This phase is network intensive as records have to be transferred from the message queue all the way to the SPE in order for them to be processed. Lastly, in the third phase, the SPE itself would perform operations on the data stream to either modify it or use it to generate a new data stream. Operations performed in this phase range from relatively compute intensive actions such as filter to outright network intensive actions such as shuffle. The contribution of this phase to the overall latency of the streaming pipeline is quite variable and is a function of the number of operations performed and individual characteristics of these operations.

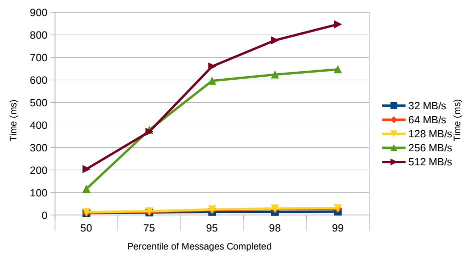


(a) 40GbE

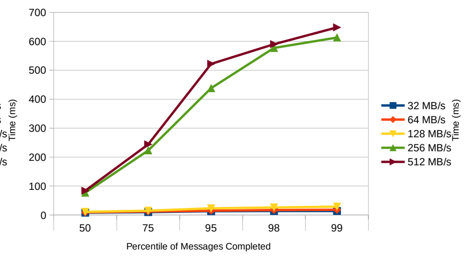


(b) IB EDR

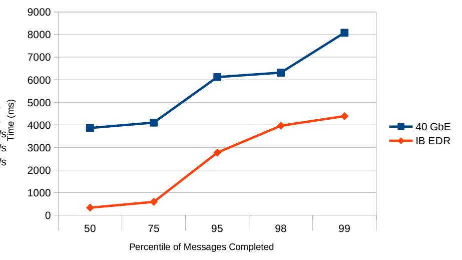
Figure 3: Performance of Apache Flink for Rebalance microbenchmark with varying input data stream velocity



(a) 40GbE



(b) IB EDR



(c) 40GbE vs IB EDR at 1024MB/s

Figure 4: Performance of Apache Flink for Rebalance Microbenchmark with varying input data stream volume

The eventual destination of data in a streaming pipeline varies from application to application. It could be stored in a distributed file system for long-term record keeping, be fed into Hadoop to be combined with old batch processed data or even fed back to the streaming pipeline to reinforce a learning system.

In this paper, we try to break down the streaming pipeline into a Read Phase and Processing Phase to study the contribution of each of these stages to the end-to-end latency of the pipeline. For this evaluation, we use the streaming module of Intel HiBench [17] benchmark suite. HiBench, originally designed for MapReduce frameworks, also provides a few streaming based microbenchmarks. We use Apache Flink as SPE and Apache Kafka as intermediate message queue for our evaluation. Our testbed consists of five physical nodes on the in-house OSU RI2 cluster. The configuration details of this cluster is described in Table 2.

Hardware	Configuration
CPU	Xeon E5-2680v4 2.4GHz 14 cores x2
Memory	512 GB
Disk	2 TB HDD
NIC	40 GbE & EDR IB (100Gb)

Table 2: Cluster Hardware Configuration

Configurations of each software and their instances are summarized in Table 3. Each process of these frameworks runs on a different node.

Software	Version	Instances
OS	CentOS 7	10
Apache Flink	1.0.3	1 JobManager 4 TaskManager
Apache Kafka	2.10-0.8.2	4 Brokers
Zookeeper	3.4.8	2

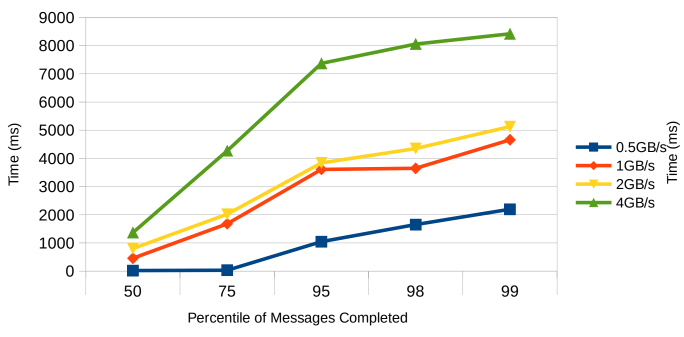
Table 3: Software Configuration

The benchmarks provided by HiBench are end-to-end benchmarks designed to measure the overall latency of processing a record in a streaming pipeline. More specifically:

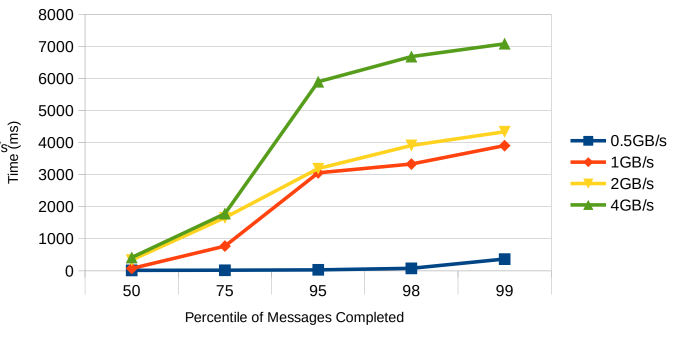
$$T_{total} = T_{read} + T_{processing}$$

where T_{total} is the total latency of the streaming pipeline, T_{read} is the time spent obtaining data from the intermediate message queue, which in our case would be read latency from Kafka to Flink and $T_{processing}$ is the time taken by the framework to process the stream.

In the first set of experiments, we evaluate the T_{read} of Rebalance operation in Apache Flink for varying input throughput rate,

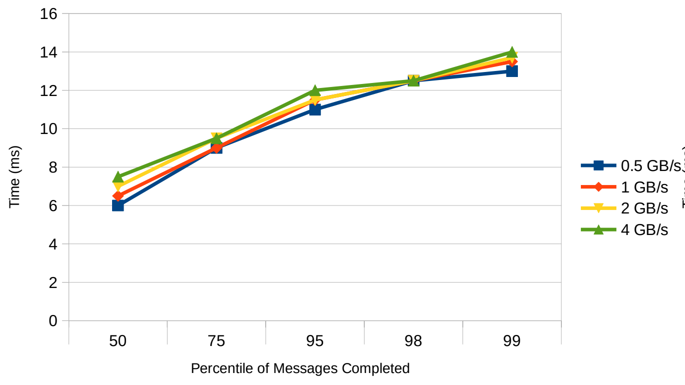


(a) 40GbE

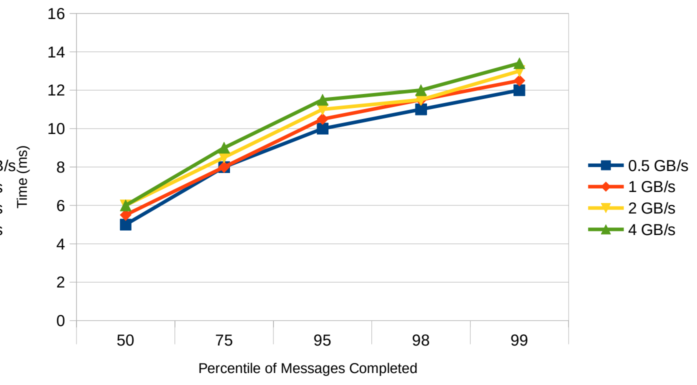


(b) IB EDR

Figure 5: Performance of Message Fetch from Kafka to Flink with varying input data stream volume



(a) 40GbE



(b) IB EDR

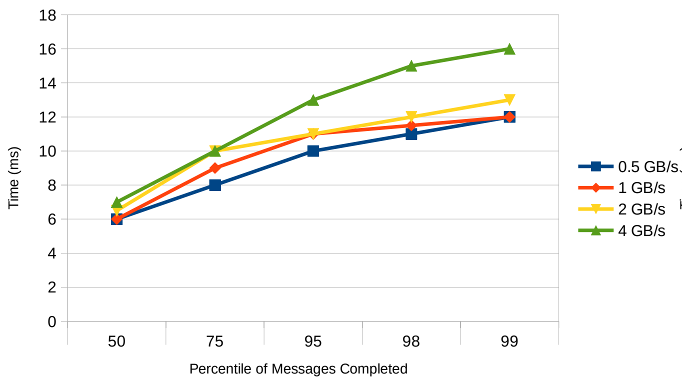
Figure 6: Processing Latency of Apache Flink for Wordcount Microbenchmark with varying input data stream volume

the rate of which is determined via records per second input to the Apache Kafka stream. We run this benchmark on both 40Gbps Ethernet and the faster InfiniBand EDR 100Gbps network over IPoIB protocol to evaluate if a faster interconnect influences latency of the pipeline. The results of these experiments are summarized in Figure 3(a) and 3(b). The increase in rate of records emitted by the source to Apache Kafka has a minimal effect on the end-to-end latency of the streaming pipeline. Moreover, the trend is similar even in the case of a faster interconnect, suggesting that the streaming pipeline is not network intensive thus far. These results are similar to the ones derived by [11]. However, their results evaluated only end-to-end latency over a unique Ethernet network.

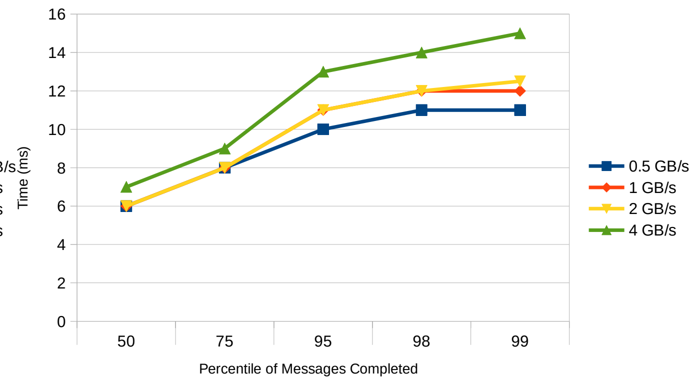
In the next phase, modify our setup to better mimic the volume of real life streaming applications. Holding the throughput rate constant at 160,000 records per second, we increase the volume of the input stream by increasing the message size from 200B to 6,400B, which translates to the volume of input data stream increased from 64 MB/s to 1024 MB/s. We ran the same Rebalance operation to measure the end-to-end latency in this modified environment on both 40Gb Ethernet and InfiniBand EDR network. The results of these experiments are illustrated in Figure 4(a) and 4(b). As opposed to increasing number of records per second produced by input source stream, varying throughput in terms of increasing the volume of data emitted by the stream source has a significant impact on end-to-end latency of the streaming pipeline. For 40Gb Ethernet, the latency curve rises steadily in response to the increase in the volume of the data stream from 32MB/s to 128MB/s. However, there is over 10x increase in 98 percentile latency when the volume of the data stream is increased from 128MB/s to 256MB/s and 512MB/s. A similar pattern can be observed in the case of InfiniBand EDR, the faster network seems to be playing its part in reducing the end-to-end latency of the pipeline and for the same volume, latencies for 40Gbps Ethernet are significantly larger compared to EDR InfiniBand.

To confirm these findings, we compared the percentile latencies for 40Gbps Ethernet and EDR InfiniBand for the volume of 1GB/s. Figure 4(c) summarizes this experiment. There is almost a 2x increase in percentile latency for 40Gbps Ethernet compared to InfiniBand EDR.

But the real question is, what is the contribution of T_{read} and $T_{processing}$ individually to the end-to-end latency T_{total} of the streaming pipeline. In order to answer this question, we modify HiBench to develop a profiling suit to measure T_{read} i.e latency of message fetch operation from Kafka to Flink. Each message



(a) 40GbE



(b) IB EDR

Figure 7: Processing Latency of Apache Flink for Rebalance Microbenchmark with varying input data stream volume

published to Kafka is timestamped with T_{event}. We then log the time $T_{arrival}$ at which each record is first encountered inside Flink. From this we have:

$$T_{read} = T_{arrival} - T_{event}$$

Similarly, we log the time T_{exit} at which each message is last operated on inside Flink. From this we conclude:

$$T_{processing} = T_{exit} - T_{arrival}$$

We evaluate the effect of varying data stream volume on T_{read} for both 40Gbps Ethernet and InfiniBand EDR and the results are summarized in Figure 5(a) and 5(b). Through these configurations, we best try to mimic typical high-volume event streams such as real-time log aggregation. Increasing the volume of input stream has a significant effect on time T_{read} taken to fetch messages from the Kafka broker to Flink. Moreover, doing so takes significantly less time for EDR InfiniBand compared 40Gbps Ethernet, indicating that that faster interconnect significantly improves the latency of this operation.

Next, we evaluate $T_{processing}$ using two workloads of different characteristics. The first workload is Wordcount which basically calculates the total number of occurrences of each word in a data stream. This workload, although quite simple in its complexity, involves computation as well as data being exchanged between processing nodes over the network. Through this workload, we aim to characterize how the performance of workloads that involve both network data transfer and computation on data changes in response to varying volume of the input stream. We carry out this experiment on both 40Gbps Ethernet and EDR InfiniBand and the results are shown in Figure 6(a) and 6(b) respectively. For both 40Gbps Ethernet and EDR InfiniBand, the volume of data stream seems to have no effect on the latency of the processing stage. This behavior is quite unlike what we saw for Fetch operation from Kafka to Flink.

For the second experiment, we select a network-intensive workload to characterize how applications with similar characteristics would respond to high volume data streams as input. To represent such a workload, we chose Rebalance operation which arbitrarily redirects messages in the data stream to nodes in the cluster. Figure 7(a) and 7(b) contains the results of this experiment. It turns out that even for a network-intensive workload, the volume of data stream does not have a significant impact on the processing time, which holds true regardless of the bandwidth of the underlying network.

From these experiments, we see that for simple data stream pipelines, the read latency for reading messages from the message queue to the SPE plays a major role in the overall end-to-end latency of the streaming pipeline, and that read latency increases significantly in response to increase in the volume of the input stream. This is not the case for the processing latency for computation-intensive workloads. Our experiments show that even with the high volume of the data stream, the processing latency does not become the major contributor to the overall latency of the pipeline, regardless of the underlying network. Therefore, just speeding up the transfer of messages from the messaging middleware to the SPE can significantly enhance the performance of the entire streaming pipeline.

5 RELATED WORK

Different research studies focusing on stream processing and processing engines have been performed. These studies have generally focused on evaluating the effect on end-to-end latency of the streaming pipeline in response to varying input throughput, in terms of increasing input message rate. Chintapalli et al. [11] evaluate Storm, Spark Streaming, and Flink using an application designed to mimic a real life stream processing scenario. They present their results in terms of 99th percentile latency and throughput comparison. Qian et al. [32] compare Spark Streaming, Storm, and Samza on standardized micro-benchmarks such as grep, projection etc. However, both of these studies base their results on end-to-end latency of the streaming pipeline. Moreover, these studies are limited to low-speed networks, such as 1GigE. This paper evaluated the performance on cutting-edge networking technologies such as 40GigE and InfiniBand EDR, which show the potential of using high-speed interconnects for Big Data streaming processing. Inoubli et al. [20] perfrom an exhaustive perfromance analysis of streaming as well as batch frameworks. They evaluate Spark, Storm, Flink, and Hadoop on the metrics of performance, scalability, and resource utilization. Marcu et al. [28] attempt to correlate different parameter settings and execution plans with resource usage of Spark and Flink to find that none of them outperforms the other for all use cases. The performance comparison in these two studies focuses mainly on framework's internal processing rather than the overall pipeline while also excluding the role of messaging middleware from the picture.

6 CONCLUSION AND FUTURE WORK

Latency is one of the most important characteristics of any of the so called "real time" data processing pipelines. In this paper, by decomposing a streaming pipeline we identify the fetch stage to be a significant contributor to the end-to-end latency of the streaming pipeline. We also present the volume of input stream as a key characteristic of determining the overall performance of the streaming pipeline.

The findings of this paper suggests that, for High-Performance Interconnects, optimizations to improve the performance of middleware such as Apache Kafka can significantly improve the performance of the entire infrastructure. This is a direction that we would like to explore further by providing native support for RDMA in the various components of the pipeline, particularly Kafka so that the benefits provided by InfiniBand in an HPC cluster may best be leveraged for such applications.

As this paper has been limited to Apache Kafka for its intermediate message queue, we would like to evaluate other such systems such as Flume [15] with a different data access model (push as opposed to pull) compared to Apache Kafka. We plan to study the implications of these results on complex real life streaming scenarios as well as with ultra-high volume data streams [40] to further strengthen our findings.

7 ACKNOWLEDGEMENTS

This research is supported in part by National Science Foundation grants CNS-1419123, IIS-1447804, and CNS-1513120.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The Design of the Borealis Stream Processing Engine.. In *Cidr*, Vol. 5. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases* 12, 2 (2003), 120–139.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. Stream: The stanford data stream management system. In *Data Stream Management*. Springer, 317–336.

[6] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 1093–1104.

[7] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* (2015), 28.

[8] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR* abs/1506.08603 (2015). http://arxiv.org/abs/1506.08603

[9] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[10] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[11] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1789–1792.

[12] Jerry Chu and Vivek Kashyap. 2006. *Transmission of IP over InfiniBand (IPoIB)*. Technical Report.

[13] DistributedLog. 2015. (2015). http://distributedlog.incubator.apache.org

[14] ElephantDB. 2011. (2011). https://github.com/nathanmarz/elephantdb

[15] Apache Flume. 2016. Welcome to apache flume. (2016).

[16] Thomas Heinze, Yuanzhen Ji, Lars Roediger, Valerio Pappalardo, Andreas Meister, Zbigniew Jerzak, and Christof Fetzer. 2015. FUGU: Elastic Data Stream Processing with Latency Constraints. *Data Engineering* (2015), 73.

[17] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. 2010. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*.

[18] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. 9.

[19] InfiniBand Trade Association. 2017. (2017). http://www.infinibandta.org

[20] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, and Alexander Jung. 2016. Big Data Frameworks: A Comparative Study. *CoRR* abs/1610.09962 (2016). http://arxiv.org/abs/1610.09962

[21] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. 2011. Memcached design on high performance rdma capable interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 743–752.

[22] Apache Kafka. 2014. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of* 5, 1 (2014).

[23] Amazon Kinesis. 2006. (2006). Retrieved October 2, 2017 from https://aws.amazon.com/kinesis

[24] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix philosophy of distributed data. *Bulletin of the IEEE CS Technical Committee on Data Engineering* (2015).

[25] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.

[26] Xiaoyi Lu, Nusrat S Islam, Md Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K Panda. 2013. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 641–650.

[27] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K Panda. 2014. Accelerating spark with RDMA for big data processing: Early experiences. In *High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on*. IEEE, 9–16.

[28] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S Pérez-Hernández. 2016. Spark versus flink: Understanding performance in big data analytics frameworks. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 433–442.

[29] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

[30] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. 2017. (2017). http://mvapich.cse.ohio-state.edu

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

[32] Shilei Qian, Gang Wu, Jie Huang, and Tathagata Das. 2016. Benchmarking modern distributed streaming platforms. In *Industrial Technology (ICIT), 2016 IEEE International Conference on*. IEEE, 592–598.

[33] RocksDB. 2012. (2012). https://rocksdb.org/

[34] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *Data Engineering* (2015), 39.

[35] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 1–10.

[36] Michael Stonebraker and Lawrence A Rowe. 1986. *The design of Postgres*. Vol. 15. ACM.

[37] Apache Storm. 2014. Storm, distributed and fault-tolerant realtime computation. (2014). http://storm.apache.org

[38] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. 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, 2–2.

[39] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 423–438.

[40] Erik Zeitler and Tore Risch. 2006. Processing high-volume stream queries on a supercomputer. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE, x147–x147.