Streaming Data Ingestion

Anshu Maheshwari, Nomchin Banga, Shreya Inamdar

Abstract

Data ingestion is the process of importing data from outside into the system, pre-processing the imported data, and storing it internally. The focus of this project was to understand the data ingestion process and perform a comparative analysis of the existing data ingestion systems. We aimed to analyze the performance of these engines with real-world workloads and identify potential bottlenecks.

1 Introduction

Data ingestion is a primary step of using data in data-intensive applications. There is an inherent diversity in data format, data ingestion rate and preprocessing requirements of different data ingestion systems. For the scope of this study, we decided to explore two widely used data ingestion systems - Apache NiFi[2] and Apache Flink[3]. We are limiting our scope to text data sets with high velocity.

As a first step, we started with understanding the data ingestion process and the architecture of Apache NiFi and Apache Flink. We then identified three real world datasets that provide varying rates of streaming data, namely, Twitter, World Wide Live and International Traffic[6]. In this project, we focused on analyzing NiFi and Flink with these datasets for different configurations to identify performance bottlenecks and software limitations. This report summarizes our experiments, observations, and potential solutions to the identified bottlenecks.

2 Motivation

The past two decades have seen an explosion of data due to proliferation of number of Internet users. The availability of huge amounts of data presented the opportunity to build data-driven applications which analyze existing data to provide an intelligent experience. Therefore, it is necessary to have an understanding of the architecture and performance bottlenecks of data ingestion systems. With the availability of numerous commercial data ingestion systems, it is also imperative to understand the advantages and trade-offs of different design choices. This report aims to provide this understanding bolstered with experimental results of Apache NiFi and Apache Flink.

3 Background

3.1 Apache NiFi

A real-time integrated data logistics and simple event processing platform, Apache NiFi automates the movement of data between disparate data sources and systems, making data ingestion fast, easy and secure[2]. Some key design choices of NiFi are:

- NiFi has zero-master architecture.
- Every node runs the same processors with user configurable parallelism.
- Certain processors can be made to run on a single node, namely the primary, which is dynamically selected by the cluster. The primary node runs the regular workflow as well. If primary fails, a new primary is elected by NiFi, thus making the system fault tolerant.
- The primary fetches the data to avoid duplication and acts as a load balancer for distributing it among the nodes in the cluster.
- Cluster management is handled by Zookeeper.

3.2 Apache Flink

Apache Flink is a distributed streaming dataflow engine which executes arbitrary dataflow programs in a data-parallel and pipelined manner[3]. Some key design choices of Flink are:

- Flink follows master-slave architecture where the user explicitly configures one node as master.
- The job to be executed is submitted to the master node, known as job manager, which distributes the tasks among the slaves, known as task managers.
- Every task manager has pre-configured task slots which specifies the number of threads that can be dedicated to a job.
- Flink requires manual intervention to handle failures.

4 Evaluation

Table 1 summarizes few key differences between NiFi and Flink based on our observations.

4.1 Dataset

We ran our experiments with three datasets with increasing streaming rates - *Twitter* (1k requests/min), *World Wide Live* (10k requests/min) and *International Traffic* (40k requests/min). Different rates allowed us to stress test NiFi and Flink as well as evaluate scalability of these ingestion engines when load increases. To ingest these data sources, we used a third-party service Satori[3] which provides multiple real-time streaming data sources free of cost. Also, it allows the data ingestion application to be built agnostic to the data source.

4.2 Environment

These experiments were carried out on CloudLab machines located in Utah. Each machine had x86 64-bit Intel microprocessor with Ubuntu 14.04, 16 GB RAM and 16-core CPUs. These machines were shared be-

Table 1: Comparison between NiFi and Flink

Feature	Apache NiFi	Apache Flink
Fault Tolerance	Resilient to node failure	Requires manual intervention
Avoid data duplication	Primary node fetches data	Random task manager node fetches data
Avoid input bottleneck	Uses List/Fetch design model	Uses third party services like Kafka
Effective nodes	All N nodes run the workflow	N-1 task managers run the workflow
Granularity of Concurrency	Processor level	Job level
Cluster Configuration	Suited for homogeneous clusters	Can leverage heterogeneous cluster

tween multiple users. We ran each experiment for ∼16 hours. A major difference between NiFi and Flink is the granularity they offer to control the concurrency of individual elements of a workflow. While NiFi allows the user to control the concurrency at processor level, Flink only allows the user to set the concurrency of the whole task. Due to this reason, the experiments performed for Flink depict the statistics across the system while those for NiFi show us a more complete picture of the internal machinery of the system.

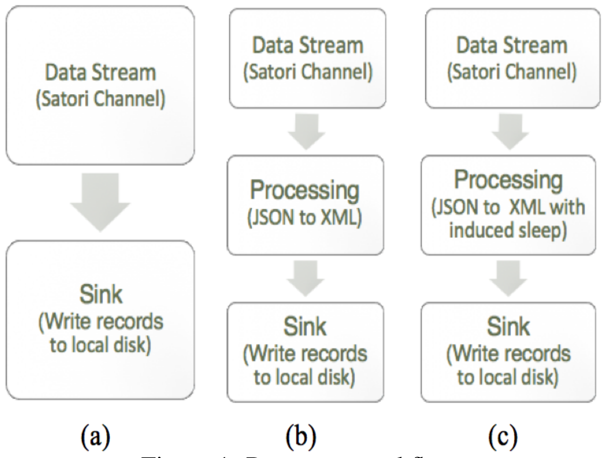


Figure 1: Processor workflow

Fig 1 shows the three processing workflows deployed on the clusters. We listened to the data from Satori (custom source) on a single node in both the workflows to avoid duplication. While the first workflow was a simple read-from-source, write-to-sink flow; the other two workflows introduced heavier processing by converting input stream from JSON to XML and inducing sleep. The first and second workflows, irrespective of configuration, posed the source node as the bottleneck. The second workflow had more processing but with no queuing of data in between the stages. The third workflow presented the opportunity to tune different components to behave as bottlenecks. Rest of this paper contains our results for the third workflow based on experiments run with different configuration.

4.3 Rate of data ingestion

To observe the effect of time on the rate of data ingestion, we ran the experiments on different days. Based on our findings, the rate of data ingestion varies across hours of the day and across different days. Table 2 summarizes the variation in rate of tweets across days. For Traffic, we find the average rate to be lower on the weekend.

Table 2: Twitter Data Rate across days

Day	Tweets/min
Weekday	∼1000
Weekend	∼1167
Thanksgiving	∼1400

Fig 2 plots the rate of data ingestion for Twitter across hours of a weekday in US. We observed that the rate of input stream is higher during early morning and evening and drops slightly during the day. We also see a spike during the afternoon hours, which we believe would be the lunch time in most professions.

4.4 Scalability

We performed experiments with 3 and 9 node clusters for both NiFi and Flink using Traffic data. This section lists our results with respect to latency, throughput and idle time of nodes.

4.4.1 Latency

Fig 3a plots the latency of the system when number of task manager nodes are increased from 2 to 8 while maintaining the concurrency of the source as 1. We find that average latency of the system decreases as number of processing nodes increase because the load is distributed across 8 nodes. Increasing the concurrency of source to 8 on 8-node cluster also decreases latency compared to 2-node cluster as the incoming network bandwidth becomes the bottleneck. Therefore, rate of ingestion does not scale linearly with cluster size.

4.4.2 Throughput

Fig 3b summarizes the changes in throughput of the system as the system scales horizontally. We observe that increasing the number of nodes, with constant concurrency, does not increase throughput as rate of data ingestion remains the same. Throughput increases if the concurrency of the whole system is increased, eventually reaching a maximum. This is expected as incoming network bandwidth is throttled after a point.

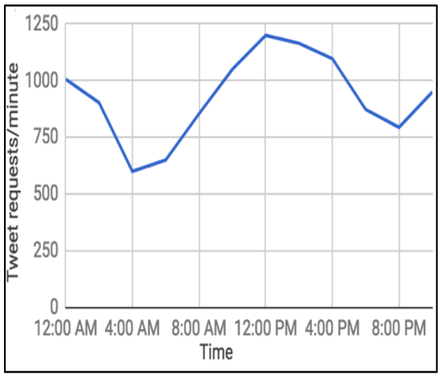
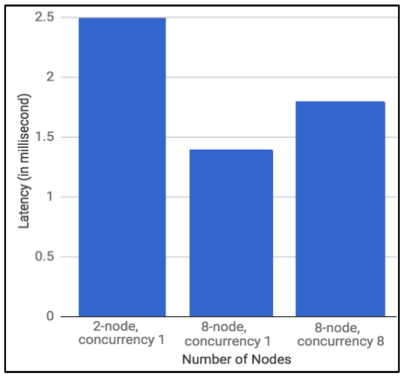
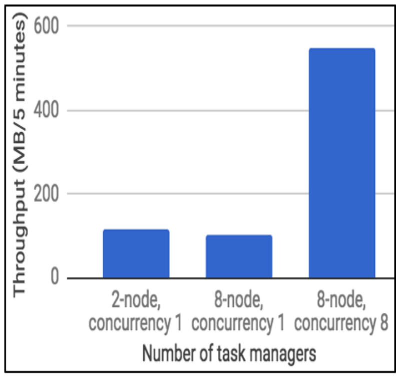


Figure 2: Variance in Data Ingestion



(a) Latency in 2-node vs 8-node



(b) Throughput in 2-node vs 8-node

Figure 3: Scalability

4.4.3 CPU Load

We captured the CPU load of individual nodes as we increased the number of task manager nodes from 2 to 8 while maintain the concurrency of source as 1. We observed that CPU load decreased by 34.8% as number of nodes increased and data got load balanced between 8 nodes instead of 2.

4.5 NiFi-specific experiments

4.5.1 Throughput

The throughput of the system is rate limited by the component with the least concurrency in the system. As can be seen in Fig 4a, throughput increased by 200% when concurrency of all components is increased to 4. Also, throughput drops further when concurrency of processing element is lower than its contemporaries. Hence we can conclude that the processing is the slowest element in the system.

4.5.2 Queue size vs Rate of ingestion

Fig 4b shows that the queue sizes increase as rate of data ingestion increases. Keeping the resource allocation per node constant and increasing rate of ingestion results in larger queues. As the latency of processing overshadows the latency of sink, processor queue is bigger than sink.

4.5.3 Low latency vs High throughput

NiFi allows the user to choose between low latency or high throughput. With low latency, data flows in a continuous stream across the system. Whereas with high throughput, each component output records in batch. Fig 4c shows high throughput increases the latency of the system. On the other hand, configuring low latency results in a lower throughput as plotted in Fig 4d.

4.5.4 Latency/Throughput trade-off vs Queuing

When system is run in high throughput mode (Fig 4e), we observe that the average source-to-network queue size increases as source releases data in batches. However, processor processes the records sequentially, thus getting sufficient time between successive batches to consume the queue. Sink also processes data in chunks, thus increasing the average size of processor-to-sink queue.

4.5.5 Concurrency vs Queuing

There are three queues in the system - source to cluster network, network buffer to processor, and processor to sink. For the same concurrency across elements, processor queue dominates the other queues as processing is the slowest element. In Fig 4f, when concurrency of source is higher than that of other components, we see a 100% increase in the source output queue arising due to back-pressure from the slow processing. However with 1:4:4 concurrency ratio, queue size reduces considerably as rate of processing is 4 time of rate of ingestion.

4.6 Flink-specific experiments

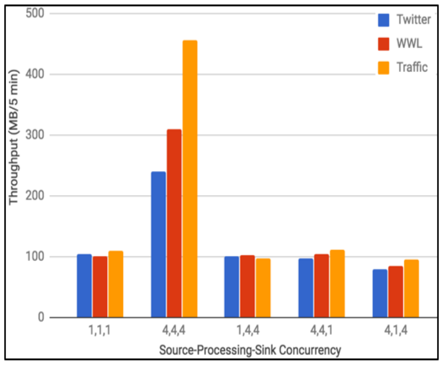
A Flink program consists of multiple tasks (transformations, sources and sinks). A task is split into several parallel instances for execution and each parallel instance processes a subset of the task's input data.

4.6.1 Ingestion rate vs Task concurrency

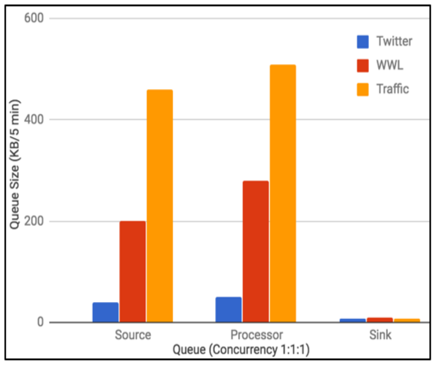
Flink schedules exactly one instance to ingest data. The rate of ingestion has little variance when the number of task slots per task manager are set to 1, 4 and 8. However, the rate decreases sharply as the concurrency of job is set to ≥ 16 (Fig 5a). This happens because we use 16-core machines which are shared resources, and when number of threads become greater than the number of cores, thread responsible for ingesting data suffers due to context switches.

4.6.2 Ingestion rate vs Network configuration

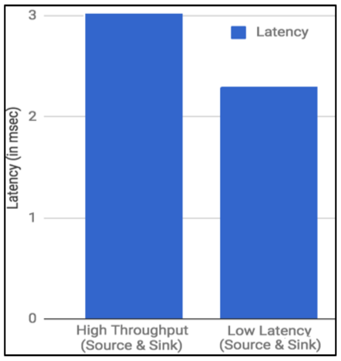
Flink, unlike NiFi, allows us to configure network properties of each node. We experimented with changes



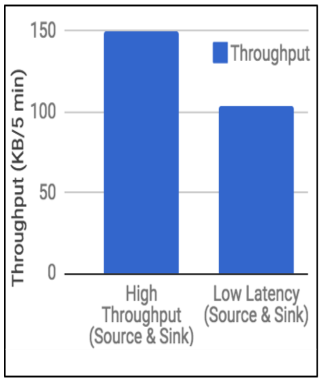
(a) Throughput vs. Concurrency



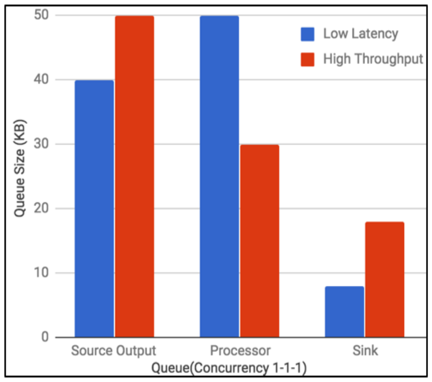
(b) Queue Size vs. Data Ingestion



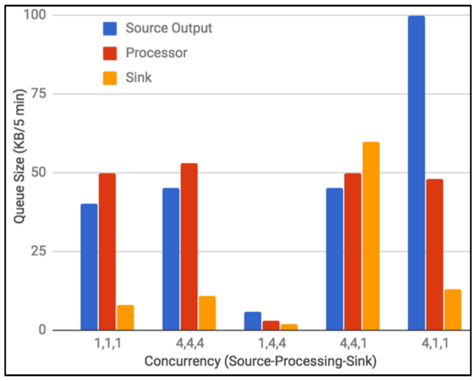
(c) Latency vs. latency/throughput trade-off



(d) Throughput vs. latency/ throughput trade-off



(e) Queue Size vs. latency/ throughput tradeoff



(f) Effect of component concurrency on queue size

Figure 4: NiFi Experiments

to sendreceive buffer size and number of Netty client threads.

taskmanager.net.sendreceive.buffer.size
The default value of buffer size is same as network buffer size of host (4096 in our case). As we increased the buffer size from 2048 to 6144, rate of data ingestion increased (Fig 5b). This implies that network bandwidth limits the amount of data ingested.

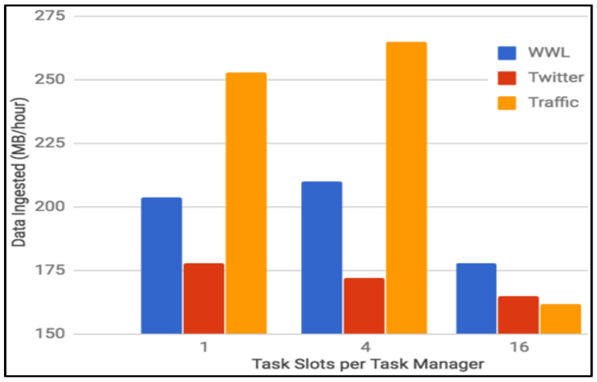
taskmanager.netty.client.threads
We expected that increasing Netty client threads will increase the ingestion rate. However, we did not observe any significant variation (Fig 5c). Possible reasons of this are - Netty threads have dedicated CPU resources, and 2 Netty threads are sufficient to handle available network bandwidth.

4.6.3 Latency

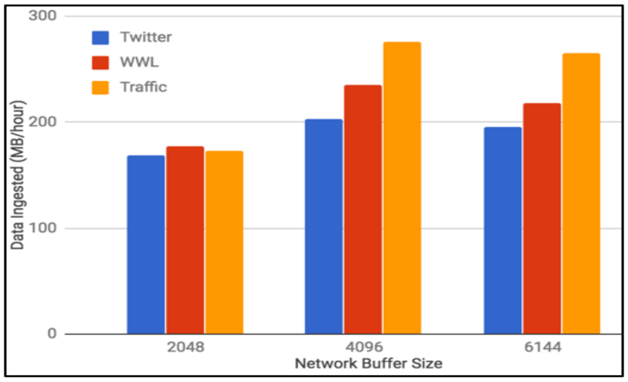
We measure the change in system latency with increase in number of task slots per task manager (Fig 5d). We observe that latency of the system remains constant till the number of threads are lesser than number of CPU cores due to lesser context switches. As we increase the concurrency of the job to 16 (equal to the number of CPU cores), mean and 99^{th} percentile latency of the system increase by 100%. This shows that increased context switching inversely affects the latency of the system.

5 Research Observations

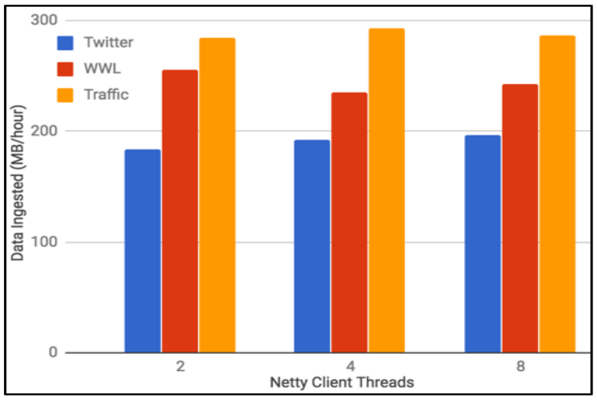
- *Latency is more important for stream ingestion systems.* Most data-driven applications ingest high velocity data. In this scenario, a single slow processor can delay the entire pipeline.
  Solution: Break one heavy processor into multiple processors for pipelining and increase in parallelism.
- *Data needs to be ingested from a single source if duplicate data is not permissible.* However, this is limited by the network bandwidth of the source node. This source also becomes highly loaded and becomes single point for failure for the system. *Solution:* Store streamed data into files and fetch without duplication or use third party services like Kafka to function as message queue.
- *Rate of data ingestion varies depending on time of the day and across days.* For maximize resource utilization and energy efficiency, data ingestion systems need to ensure dynamic resource allocation.
- *Fault tolerance is an important aspect of data ingestion systems deployed in data centers.* As clusters are



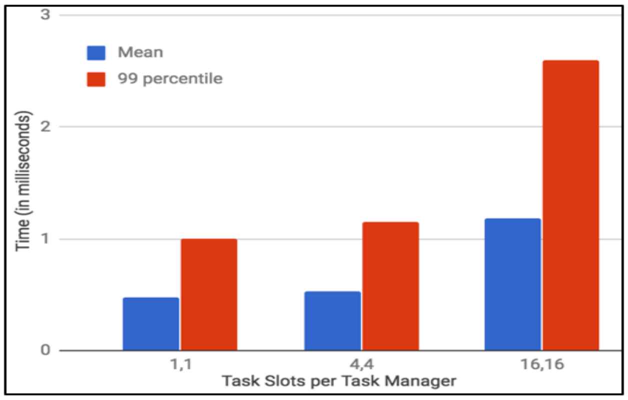
(a) Data Ingested vs. Task Concurrency



(b) Data Ingested vs. Network Buffer Size



(c) Data Ingested vs. Netty Threads



(d) Latency vs. Number of Task Slots

Figure 5: Flink Experiments

dynamic, failure of nodes is expected. When dealing with high rate of data ingestion, unavailability of one node may overload other nodes in the cluster. This further increases chances of failure due to limitations of memory and network bandwidth.

- *NiFi is more robust than Flink.* Our experiments show that NiFi was able to scale better as rate of data stream increased. Many times, nodes running Flink would get overloaded and fail whereas NiFi was able to dynamically load balance within the cluster and restart the failed nodes without manual intervention.

6 Challenges

- Though both NiFi and Flink have in-built support to analyze Twitter streams, Twitter sources are rate-limited to 1 request/15 min. This was not an acceptable rate for our experiments. To circumvent this, we had to find a third party service which provides data at reasonable streaming rates.
- Using Satori meant there were no generalized source APIs to listen to data streaming channels. We developed our own customized scripts to subscribe to Satori channels and ingest data.
- Apache Flink does not support recording the metrics beyond 5 minutes. However, recording the progress and state of experiments for 16 hours was a crucial part of our experiments. To gather the needed historical data, we integrated Flink with Graphite reporter.
- Lack of resources/documentations for Apache Flink was a major hurdle in this project. We actively reached out to Flink's user and developer communities to resolve our queries.
- Cluster setup could not be fully automated due to specific changes in cluster configurations. Therefore, we had to start from scratch whenever the CloudLab cluster expired (∼15 days).

References

[1] Everything Flows: The value of stream processing and streaming integration, 451 Research, 2016.
[2] Apache NiFi: https://hortonworks.com/apache/nifi/
[3] Apache Flink: https://en.wikipedia.org/wiki/Apache_Flink
[4] Rabl, Tilmann et. al. (2016). Apache Flink in current research. Information Technology.
[5] Jacobs, K.M.J. (2016). "Apache Flink: Distributed Stream Data Processing"
[6] Satori: https://www.satori.com/