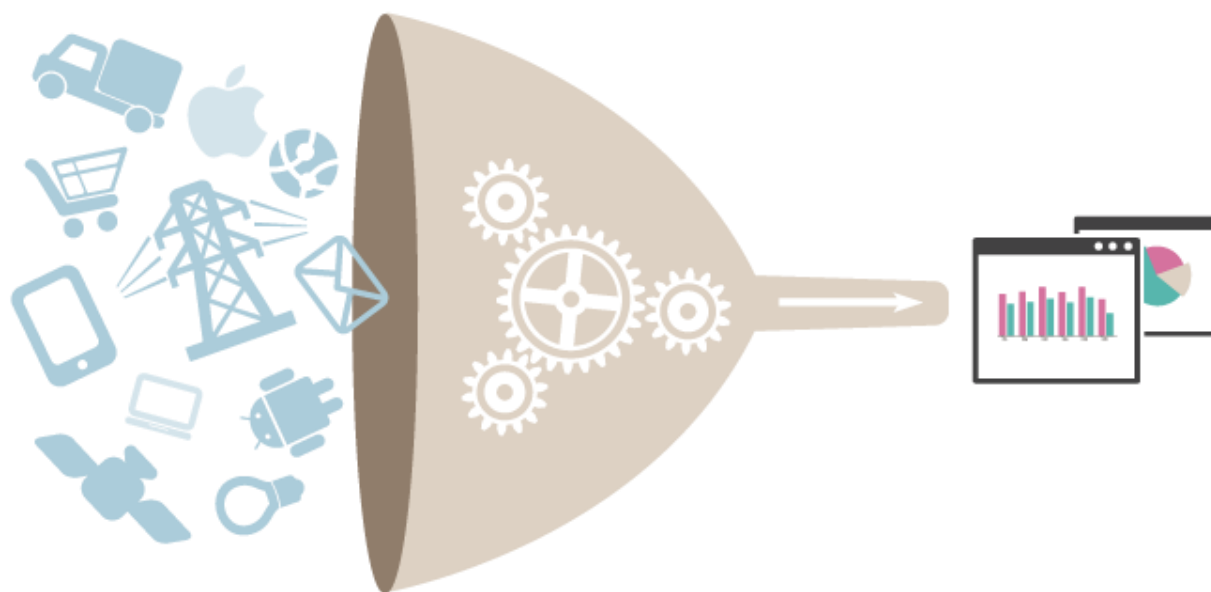


Data Ingestion

An exploration of
Apache Flume, Amazon Kinesis, Apache Kafka & Apache Samza



Report Prepared by

Anirudha Deepak Bedre

Avikal Chhetri

Table of Contents

What is a Stream?	5
Stream processing.....	6
Apache Flume	7
Architecture.....	8
Data flow model.....	8
Complex flows.....	8
Reliability.....	9
Recoverability	9
Network streams	9
Setting multi-agent flow	9
Consolidation	10
Multiplexing the flow.....	10
Spark Streaming + Flume Integration Guide	11
Amazon Kinesis	15
Important Concepts	16
The Kinesis Processing Model.....	16
Scaling and Sharding	17
Kinesis Pricing.....	19
Kinesis From the Console.....	19
Amazon Kinesis Key Concepts.....	21
Amazon Kinesis High-level Architecture	21
Amazon Kinesis Terminology	22
Amazon Kinesis Streams	22
Data Records	22
Producers	23
Apache Kafka	25
Use Cases	30
Apache Samza	33
Concepts.....	33
Streams.....	33
Jobs	34

Partitions.....	34
Tasks.....	35
Dataflow Graphs	36
Containers	37
Architecture.....	37
Kafka.....	38
YARN	39
YARN Architecture	39
Samza and YARN	39
Samza	40
Kafka Performance.....	41
Comparison.....	51
Conclusion.....	51
References.....	51

INTRODUCTION:

In this report we will explore following data ingestion tools in detail:

- Flume
- Kinesis
- Kafka
- Samza

We begin by looking at some of the basic concepts involved in the world of data ingestion.

What is Data ingestion?

Data ingestion is the process of obtaining, importing and processing data for later use or storage in a database.

What is High-Velocity Data?

Computer systems are creating ever more data at increasing speeds, and there are a growing number of consumers of that data—both operations and analytics. Hadoop-style batch processing has awakened engineers to the value of big data, but they increasing demand access to the data earlier. In essence people not only want all of the data, they want it as soon as possible; this is driving the trend toward high-velocity data. High-velocity—or fast data—can mean millions of rows of data per second, we are talking about massive volume. One of the use cases for high-velocity data is real-time analytics.

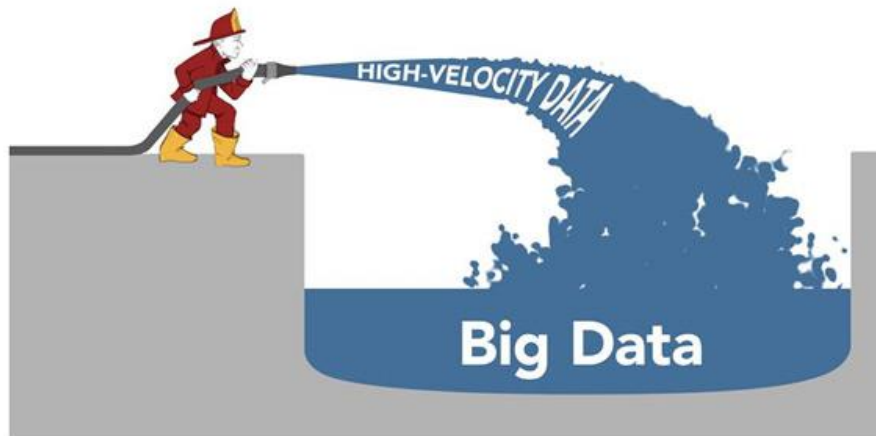
Comparing High-Velocity Data & Big Data

High-Velocity Data

- Real-Time
- Performance & Volume Challenges
- Use Cases: Operations & Analytics

Big Data

- Batch Process
- Volume Challenge
- Use Case: Analytics



Messaging systems

Messaging systems are a popular way of implementing near-realtime asynchronous computation. Messages can be added to a message queue (ActiveMQ, RabbitMQ), pub-sub system (Kestrel, Kafka), or log aggregation system (Flume, Kinesis) when something happens.

Downstream consumers read messages from these systems, and process them or take actions based on the message contents.

Suppose you have a website, and every time someone loads a page, you send a “user viewed page” event to a messaging system. You might then have consumers which do any of the following:

- Store the message in Hadoop for future analysis
- Count page views and update a dashboard
- Trigger an alert if a page view fails
- Send an email notification to another user
- Join the page view event with the user’s profile, and send the message back to the messaging system

A messaging system lets you decouple all of this work from the actual web page serving.

What is a Stream?

A stream is a sequence of data elements made available over time. A stream can be thought of as items on a conveyor belt being processed one at a time rather than in large batches

Stream processing

Stream processing enables organizations to detect insights (risks and opportunities) in high velocity data which can only be detected and acted on at a moment's notice. High velocity flows of data from real-time sources such as market data, Internet of Things, mobile, sensors, clickstream, and even transactions, remain largely un-navigated.

A messaging system is a fairly low-level piece of infrastructure—it stores messages and waits for consumers to consume them. When you start writing code that produces or consumes messages, you quickly find that there are a lot of tricky problems that have to be solved in the processing layer.

Consider a counting example, where you count page views on a website and update a dashboard. What happens when the machine that your consumer is running on fails, and your current counter values are lost? How do you recover? Where should the processor be run when it restarts? What if the underlying messaging system sends you the same message twice, or loses a message? (Unless you are careful, your counts will be incorrect.) What if you want to count page views grouped by the page URL? How do you distribute the computation across multiple machines if it's too much for a single machine to handle?

Stream processing is a higher level of abstraction on top of messaging systems, and it's meant to address precisely this category of problems.

Stream processing enables organizations to:

- Analyze and act up upon rapidly changing data in real time
- Enhance existing models with new insights
- Capture, analyze and act on insight before opportunities are lost, forever
- Move from batching process to real-time analytics and decisions

The top use cases for stream processing are:

- **Internet of Things (IoT)** – Optimize availability, performance, capacity and resource utilization.
- **Real-time sentiment analysis of social media** – Effectively respond to improve the client experience.
- **Enhanced security intelligence** – Predict, prevent and act on security threats and real-time fraud detection. Increase situational awareness.
- **Next best action** – Act on up-to-the-second observations, while the event/transaction is still happening.



Apache Flume

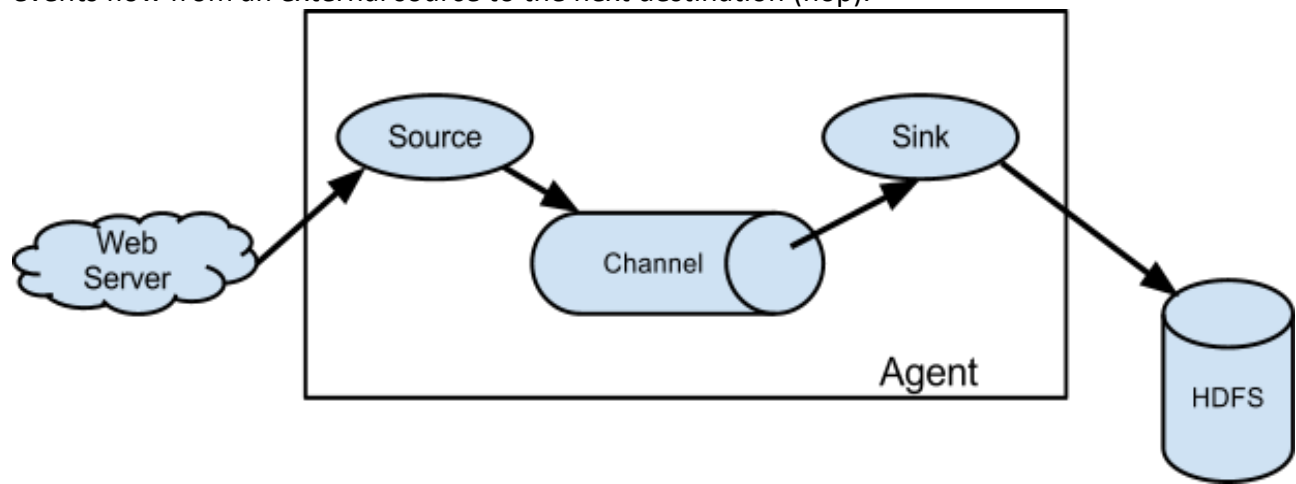
Apache Flume is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store.

The use of Apache Flume is not only restricted to log data aggregation. Since data sources are customizable, Flume can be used to transport massive quantities of event data including but not limited to network traffic data, social-media-generated data, email messages and pretty much any data source possible.

Architecture

Data flow model

A Flume event is defined as a unit of data flow having a byte payload and an optional set of string attributes. A Flume agent is a (JVM) process that hosts the components through which events flow from an external source to the next destination (hop).



A Flume source consumes events delivered to it by an external source like a web server. The external source sends events to Flume in a format that is recognized by the target Flume source. For example, an Avro Flume source can be used to receive Avro events from Avro clients or other Flume agents in the flow that send events from an Avro sink. A similar flow can be defined using a Thrift Flume Source to receive events from a Thrift Sink or a Flume Thrift Rpc Client or Thrift clients written in any language generated from the Flume thrift protocol. When a Flume source receives an event, it stores it into one or more channels. The channel is a passive store that keeps the event until it's consumed by a Flume sink. The file channel is one example – it is backed by the local filesystem. The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow. The source and sink within the given agent run asynchronously with the events staged in the channel.

Complex flows

Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination. It also allows fan-in and fan-out flows, contextual routing and backup routes (fail-over) for failed hops.

Reliability

The events are staged in a channel on each agent. The events are then delivered to the next agent or terminal repository (like HDFS) in the flow. The events are removed from a channel only after they are stored in the channel of next agent or in the terminal repository. This is how the single-hop message delivery semantics in Flume provide end-to-end reliability of the flow.

Flume uses a transactional approach to guarantee the reliable delivery of the events. The sources and sinks encapsulate in a transaction the storage/retrieval, respectively, of the events placed in or provided by a transaction provided by the channel. This ensures that the set of events are reliably passed from point to point in the flow. In the case of a multi-hop flow, the sink from the previous hop and the source from the next hop both have their transactions running to ensure that the data is safely stored in the channel of the next hop.

Recoverability

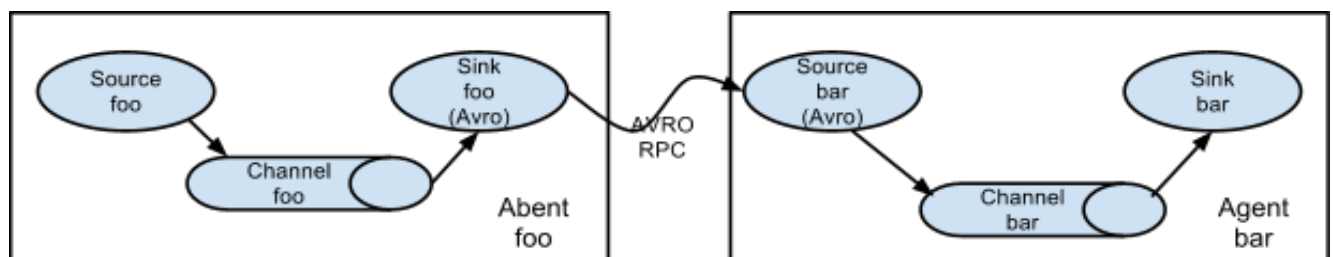
The events are staged in the channel, which manages recovery from failure. Flume supports a durable file channel which is backed by the local file system. There's also a memory channel which simply stores the events in an in-memory queue, which is faster but any events still left in the memory channel when an agent process dies can't be recovered.

Network streams

Flume supports the following mechanisms to read data from popular log stream types, such as:

- 1 Avro
- 2 Thrift
- 3 Syslog
- 4 Netcat

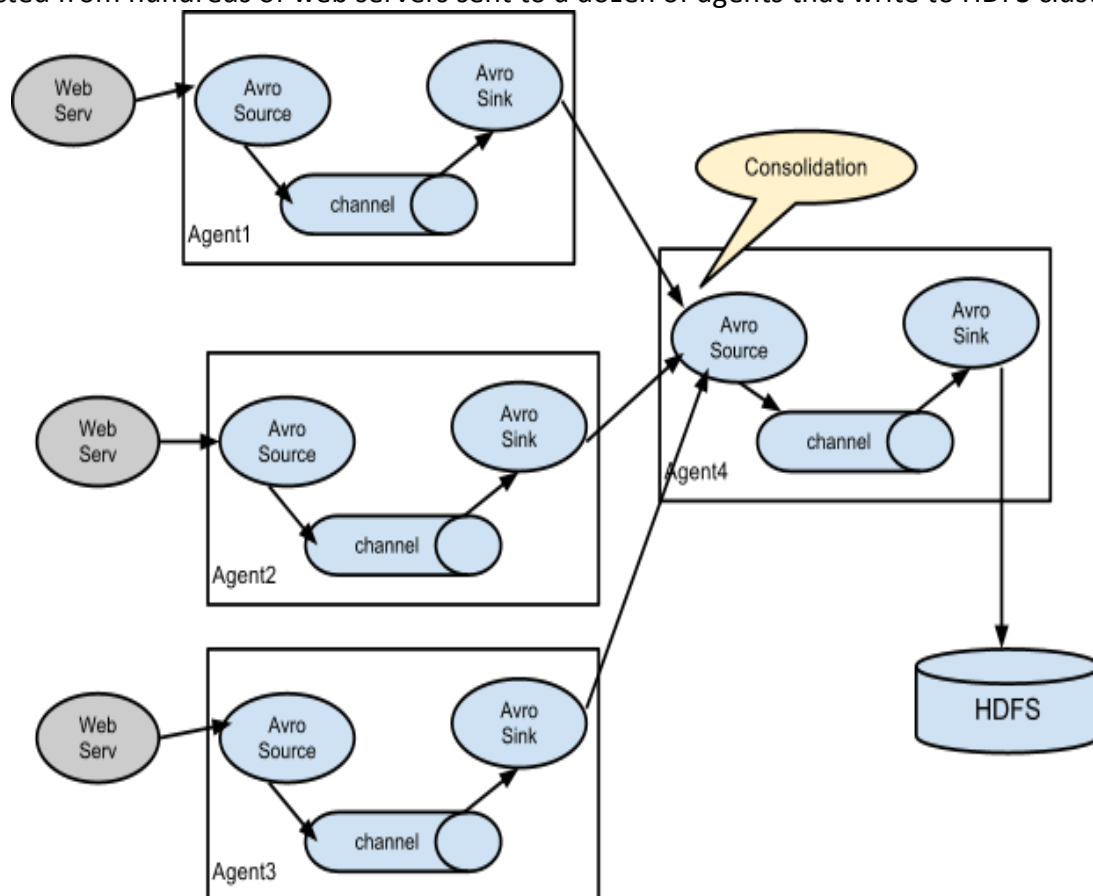
Setting multi-agent flow



In order to flow the data across multiple agents or hops, the sink of the previous agent and source of the current hop need to be avro type with the sink pointing to the hostname (or IP address) and port of the source.

Consolidation

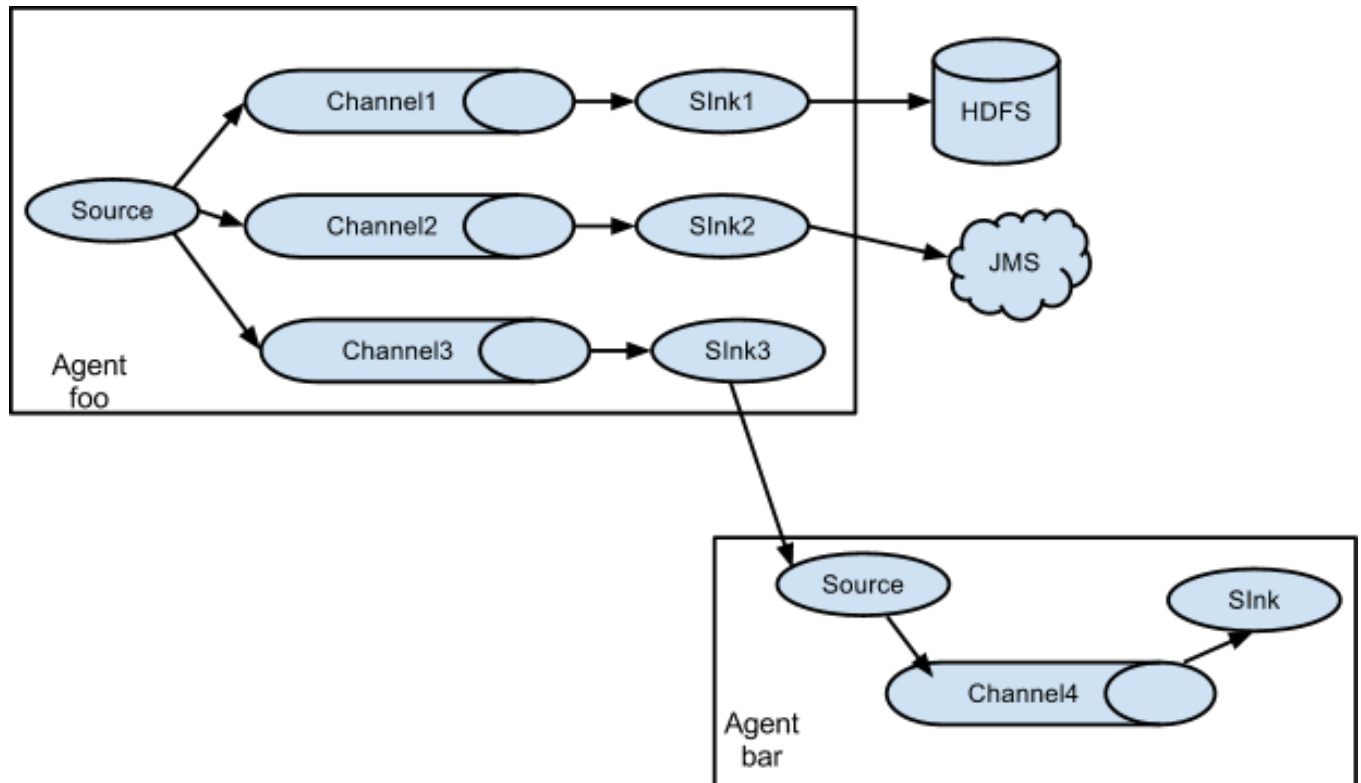
A very common scenario in log collection is a large number of log producing clients sending data to a few consumer agents that are attached to the storage subsystem. For example, logs collected from hundreds of web servers sent to a dozen of agents that write to HDFS cluster.



This can be achieved in Flume by configuring a number of first tier agents with an avro sink, all pointing to an avro source of single agent (Again you could use the thrift sources/sinks/clients in such a scenario). This source on the second tier agent consolidates the received events into a single channel which is consumed by a sink to its final destination.

Multiplexing the flow

Flume supports multiplexing the event flow to one or more destinations. This is achieved by defining a flow multiplexer that can replicate or selectively route an event to one or more channels.



The above example shows a source from agent “foo” fanning out the flow to three different channels. This fan out can be replicating or multiplexing. In case of replicating flow, each event is sent to all three channels. For the multiplexing case, an event is delivered to a subset of available channels when an event’s attribute matches a preconfigured value. For example, if an event attribute called “txnType” is set to “customer”, then it should go to channel1 and channel3, if it’s “vendor” then it should go to channel2, otherwise channel3. The mapping can be set in the agent’s configuration file.

Spark Streaming + Flume Integration Guide

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. Here we explain how to configure Flume and Spark Streaming to receive data from Flume. There are two approaches to this. Flume is not yet available in the Python API.

Approach 1: Flume-style Push-based Approach

Flume is designed to push data between Flume agents. In this approach, Spark Streaming essentially sets up a receiver that acts an Avro agent for Flume, to which Flume can push the data. Here are the configuration steps.

General Requirements

Choose a machine in your cluster such that

- When your Flume + Spark Streaming application is launched, one of the Spark workers must run on that machine.
- Flume can be configured to push data to a port on that machine.

Due to the push model, the streaming application needs to be up, with the receiver scheduled and listening on the chosen port, for Flume to be able push data.

Configuring Flume

Configure Flume agent to send data to an Avro sink by having the following in the configuration file.

```
agent.sinks = avroSink
agent.sinks.avroSink.type = avro
agent.sinks.avroSink.channel = memoryChannel
agent.sinks.avroSink.hostname = <chosen machine's hostname>
agent.sinks.avroSink.port = <chosen port on the machine>
```

Configuring Spark Streaming Application

- 1 **Linking:** In your SBT/Maven project definition, link your streaming application against the following artifact (see Linking section in the main programming guide for further information).

```
groupId = org.apache.spark
artifactId = spark-streaming-flume_2.10
version = 1.4.0
```

- 2 **Programming:** In the streaming application code, import FlumeUtils and create input DStream as follows.

```
import org.apache.spark.streaming.flume._
```

```
val flumeStream = FlumeUtils.createStream(streamingContext, [chosen machine's hostname], [chosen port])
```

Note that the hostname should be the same as the one used by the resource manager in the cluster (Mesos, YARN or Spark Standalone), so that resource allocation can match the names and launch the receiver in the right machine.

- 3 **Deploying:** Package spark-streaming-flume_2.10 and its dependencies (except spark-core_2.10 and spark-streaming_2.10 which are provided by spark-submit) into the application JAR. Then use spark-submit to launch your application (see Deploying section in the main programming guide).

Approach 2: Pull-based Approach using a Custom Sink

Instead of Flume pushing data directly to Spark Streaming, this approach runs a custom Flume sink that allows the following.

- Flume pushes data into the sink, and the data stays buffered.
- Spark Streaming uses a reliable Flume receiver and transactions to pull data from the sink. Transactions succeed only after data is received and replicated by Spark Streaming.

This ensures stronger reliability and fault-tolerance guarantees than the previous approach. However, this requires configuring Flume to run a custom sink. Here are the configuration steps.

General Requirements

Choose a machine that will run the custom sink in a Flume agent. The rest of the Flume pipeline is configured to send data to that agent. Machines in the Spark cluster should have access to the chosen machine running the custom sink.

Configuring Flume

Configuring Flume on the chosen machine requires the following two steps.

- 1 **Sink JARs:** Add the following JARs to Flume's classpath (see Flume's documentation to see how) in the machine designated to run the custom sink .
 - (i) Custom sink JAR: Download the JAR corresponding to the following artifact.

```
groupId = org.apache.spark  
artifactId = spark-streaming-flume-sink_2.10  
version = 1.4.0
```

- (ii) Scala library JAR: Download the Scala library JAR for Scala 2.10.4. It can be found with the following artifact detail.

```
groupId = org.scala-lang
```

```
artifactId = scala-library  
version = 2.10.4
```

- 2 Configuration file:** On that machine, configure Flume agent to send data to an Avro sink by having the following in the configuration file.

```
agent.sinks = spark  
agent.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink  
agent.sinks.spark.hostname = <hostname of the local machine>  
agent.sinks.spark.port = <port to listen on for connection from Spark>  
agent.sinks.spark.channel = memoryChannel
```

Also make sure that the upstream Flume pipeline is configured to send the data to the Flume agent running this sink.

Configuring Spark Streaming Application

- 1 Linking:** In your SBT/Maven project definition, link your streaming application against the spark-streaming-flume_2.10 (see [Linking section](#) in the main programming guide).
- 2 Programming:** In the streaming application code, import FlumeUtils and create input DStream as follows.

```
import org.apache.spark.streaming.flume._  
  
val flumeStream = FlumeUtils.createPollingStream(streamingContext, [sink machine hostname], [sink port])
```

Note that each input DStream can be configured to receive data from multiple sinks.

- 3 Deploying:** Package spark-streaming-flume_2.10 and its dependencies (except spark-core_2.10 and spark-streaming_2.10 which are provided by spark-submit) into the application JAR. Then use spark-submit to launch your application



Amazon Kinesis

Amazon Kinesis is a managed service designed to handle real-time streaming of big data. It can accept any amount of data, from any number of sources, scaling up and down as needed.

You can use Kinesis in any situation that calls for large-scale, real-time data ingestion and processing. Logs for servers and other IT infrastructure, social media or market data feeds, web clickstream data, and the like are all great candidates for processing with Kinesis.

A screenshot of the Amazon Kinesis console. At the top, there's a navigation bar with 'Services' and 'Edit' dropdowns. The main heading is 'Welcome to Amazon Kinesis'. Below it, a paragraph describes the service: 'Amazon Kinesis is a managed service that scales elastically for real-time processing of streaming big data. Amazon Kinesis takes in large streams of data records that can then be consumed in real time by multiple data-processing applications running on Amazon Elastic Compute Cloud (Amazon EC2) instances.' A blue button labeled 'Create Stream' is prominently displayed. Below this, a section titled 'Using Amazon Kinesis' contains three columns. The first column, 'Create a data stream', features a bar chart icon with a plus sign and text explaining that data streams consist of shards and provides a 'Learn More' link. The second column, 'Write data to the stream', features a gear icon with a mouse cursor and text explaining that data from many producers can be written to the stream for scalable and reliable intake, also with a 'Learn More' link. The third column, 'Consume data from the stream', features a monitor icon with a line graph and text explaining that multiple data-processing applications can consume data from the stream to produce real-time insights, also with a 'Learn More' link.

Imagine a situation where fresh data arrives in a continuous stream, 24 hours a day, 7 days a week. You need to capture the data, process it, and turn it into actionable conclusions as soon as possible, ideally within a matter of seconds. Perhaps the data rate or the compute power required for the analytics varies by an order of magnitude over time. Traditional batch processing techniques are not going to do the job.

Important Concepts

Your application can create any number of Kinesis streams to reliably capture, store and transport data. Streams have no intrinsic capacity or rate limits. All incoming data is replicated across multiple AWS Availability Zones for high availability. Each stream can have multiple writers and multiple readers.

When you create a stream you specify the desired capacity in terms of shards. Each shard has the ability to handle 1000 write transactions (up to 1 megabyte per second — we call this the ingress rate) and up to 5 read transactions (up to 2 megabytes per second — the egress rate). You can scale a stream up or down at any time by adding or removing shards without affecting processing throughput or incurring any downtime, with new capacity ready to use within seconds. Pricing (which I will cover in depth in just a bit) is based on the number of shards in existence and the number of writes that you perform.

The Kinesis client library is an important component of your application. It handles the details of load balancing, coordination, and error handling. The client library will take care of the heavy lifting, allowing your application to focus on processing the data as it becomes available. Applications read and write data records to streams. Records can be up to 50 Kilobytes in length and are comprised of a partition key and a data blob, both of which are treated as immutable sequences of bytes. The record's partition determines which shard will handle the data blob; the data blob itself is not inspected or altered in any way. A sequence number is assigned to each record as part of the ingestion process. Records are automatically discarded after 24 hours.

The Kinesis Processing Model

The “producer side” of your application code will use the **PutRecord** function to store data in a stream, passing in the stream name, the partition key, and the data blob. The partition key is hashed using an MD5 hashing function and the resulting 128-bit value will be used to select one of the shards in the stream.

The “consumer” side of your application code reads through data in a shard sequentially. There are two steps to start reading data. First, your application uses `GetShardIterator` to specify the

position in the shard from which you want to start reading data. `GetShardIterator` gives you the following options for where to start reading the stream:

- **AT_SEQUENCE_NUMBER** to start at given sequence number.
- **AFTER_SEQUENCE_NUMBER** to start after a given sequence number.
- **TRIM_HORIZON** to start with the oldest stored record.
- **LATEST** to start with new records as they arrive.

Next, your application uses **GetNextRecords** to retrieve up to 2 megabytes of data per second using the shard iterator. The easiest way to use `GetNextRecords` is to create a loop that calls `GetNextRecords` repeatedly to get any available data in the shard. These interfaces are, however, best thought of as a low-level interfaces; we expect most applications to take advantage of the higher-level functions provided by the Kinesis client library.

The client library will take care of a myriad of details for you including fail-over, recovery, and load balancing. You simply provide an implementation of the **IRecordProcessor** interface and the client library will “push” new records to you as they become available. This is the easiest way to get started using Kinesis.

After processing the record, your consumer code can pass it along to another Kinesis stream, write it to an Amazon S3 bucket, a Redshift data warehouse, or a DynamoDB table, or simply discard it.

Scaling and Sharding

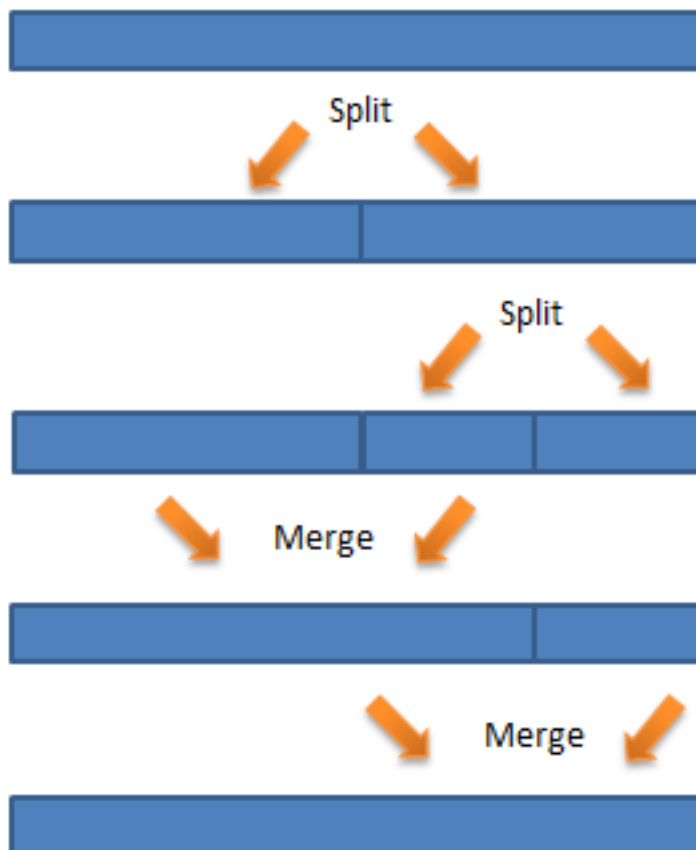
You are responsible for two different aspects of scalability – processing and sharding. You need to make sure that you have enough processing power to keep up with the flow of records. You also need to manage the number of shards.

Let’s start with the processing aspect of scalability. The easiest way to handle this responsibility is to implement your Kinesis application with the Kinesis client library and to host it on an Amazon EC2 instance within an Auto Scaling group. By setting the minimum size of the group to 1 instance, you can recover from instance failure. Set the maximum size of the group to a sufficiently high level to ensure plenty of headroom for scaling activities. If your processing is CPU-bound, you will want to scale up and down based on the CloudWatch CPU Utilization metric. On the other hand, if your processing is relatively lightweight, you may find that scaling based on Network Traffic In is more effective.

Ok, now on to sharding. You should create the stream with enough shards to accommodate the expected data rate. You can then add or delete shards as the rate changes. The APIs for these operations are **SplitShard** and **MergeShards**, respectively. In order to use these operations effectively you need to know a little bit more about how partition keys work.

As I have already mentioned, your partition keys are run through an MD5 hashing function to produce a 128-bit number, which can be in the range of 0 to 2127-1. Each stream breaks this interval into one or more contiguous ranges, each of which is assigned to a particular shard. Let's start with the simplest case, a stream with a single shard. In this case, the entire interval maps to a single shard. Now, things start to heat up and you begin to approach the data handling limit of a single shard. It is time to scale up! If you are confident that the MD5 hash of your partition keys results in values that are evenly distributed across the 128-bit interval, then you can simply split the first shard in the middle. It will be responsible for handling values from 0 to 2126-1, and the new shard will be responsible for values from 2126 to 2127-1. Reality is never quite that perfect, and it is possible that the MD5 hash of your partition keys isn't evenly distributed. In this case, splitting the partition down the middle would be a sub-optimal decision. Instead, you (in the form of your sharding code) would like to make a more intelligent decision, one that takes the actual key distribution into account. To do this properly, you will need to track the long-term distribution of hashes with respect to the partitions, and to split the shards accordingly.

You can reduce your operational costs by merging shards when traffic declines. You can merge adjacent shards; again, an intelligent decision will maintain good performance and low cost. Here's a diagram of one possible sequence of splits and merges over time:



Kinesis Pricing

Kinesis pricing is simple: you pay for PUTs and for each shard of throughput capacity. Lets assume that you have built a game for mobile devices and you want to track player performance, top scores, and other metrics associated with your game in real-time so that you can update top score dashboards and more.

Lets also assume that each mobile device will send a 2 kilobyte message every 5 seconds and that at peak youll have 10,000 devices simultaneously sending messages to Kinesis. You can scale up and down the size of your stream, but for simplicity lets assume its a constant rate of data.

Use this data to calculate how many Shards of capacity youll need to ingest the incoming data. The Kinesis console helps you estimate using a wizard, but lets do the math here. 10,000 (PUTs per second) * 2 kilobytes (per PUT) = 20 megabytes per second. You will need 20 Shards to process this stream of data.

Kinesis uses simple pay as you go pricing. You pay \$0.028 per 1,000,000 PUT operations and you pay \$0.015 per shard per hour. For one hour of collecting game data youd pay \$0.30 for the shards and about \$1.01 for the 36 million PUT calls, or \$1.31.

Kinesis From the Console

You can create and manage Kinesis streams using the Kinesis APIs, the AWS CLI, and the AWS Management Console. Here's a brief tour of the console support.

Click on the Create Stream button to get started. You need only enter a stream name and the number of shards to get started:

The screenshot shows the 'Create Stream' wizard in the AWS Management Console. At the top, there's a navigation bar with 'Services' and 'Edit' dropdowns. Below that, the title 'Amazon Kinesis Create Stream' is displayed. A descriptive paragraph explains that a stream is composed of multiple shards, each providing a fixed unit of capacity, and that the total capacity is the sum of the capacities of its shards. It also mentions that each shard corresponds to 1 MB/s of write capacity and 2 MB/s of read capacity, and provides links to the 'Amazon Kinesis Developer Guide' and 'Amazon Kinesis Pricing Page'. The form has two main input fields: 'Stream Name*' and 'Number of Shards*'. The 'Stream Name*' field has a tooltip that says 'The Stream Name identifies the stream and is used to access the data written to the stream'. The 'Number of Shards*' field has a tooltip that says 'Use the shard calculator to estimate the number of shards needed for the stream' and 'You can change the number of shards in the stream without re-creating the stream'. There is a checkbox labeled 'Help me decide how many shards I need'. Below the input fields, a section titled 'Values calculated based on the number of shards entered above:' shows a table with 'Read:' and 'Write:' columns. The table has two rows: 'Total Stream Capacity: - MB/s' and 'Max Transactions/second: -'. At the bottom, there is a note '* Required information' and two buttons: 'Cancel' and 'Create'.

Services Edit

Amazon Kinesis Create Stream

A stream is composed of multiple shards, each of which provides a fixed unit of capacity. The total capacity of the stream is the sum of the capacities of its shards. Each shard corresponds to 1 MB/s of write capacity and 2 MB/s of read capacity. See the [Amazon Kinesis Developer Guide](#) for more information on estimating number of shards needed for your stream. Note that the cost of the stream is also a function of the number of shards. To learn more about the stream, see the [Amazon Kinesis Pricing Page](#).

Stream Name* The Stream Name identifies the stream and is used to access the data written to the stream

☐ Help me decide how many shards I need Use the shard calculator to estimate the number of shards needed for the stream

Number of Shards* You can change the number of shards in the stream without re-creating the stream

Values calculated based on the number of shards entered above:

	Read:	Write:
Total Stream Capacity:	- MB/s	- MB/s
Max Transactions/second:	-	-

* Required information

Cancel Create

The console includes a calculator to help you estimate the number of shards you need:

The number of shards your stream needs depends on the volume of data written and read from the stream. Enter values below to estimate the number of shards for the stream.

Volume of Data Written

Average Item Size (KB): Uses integers between 1-50

Maximum Items Written/Second:

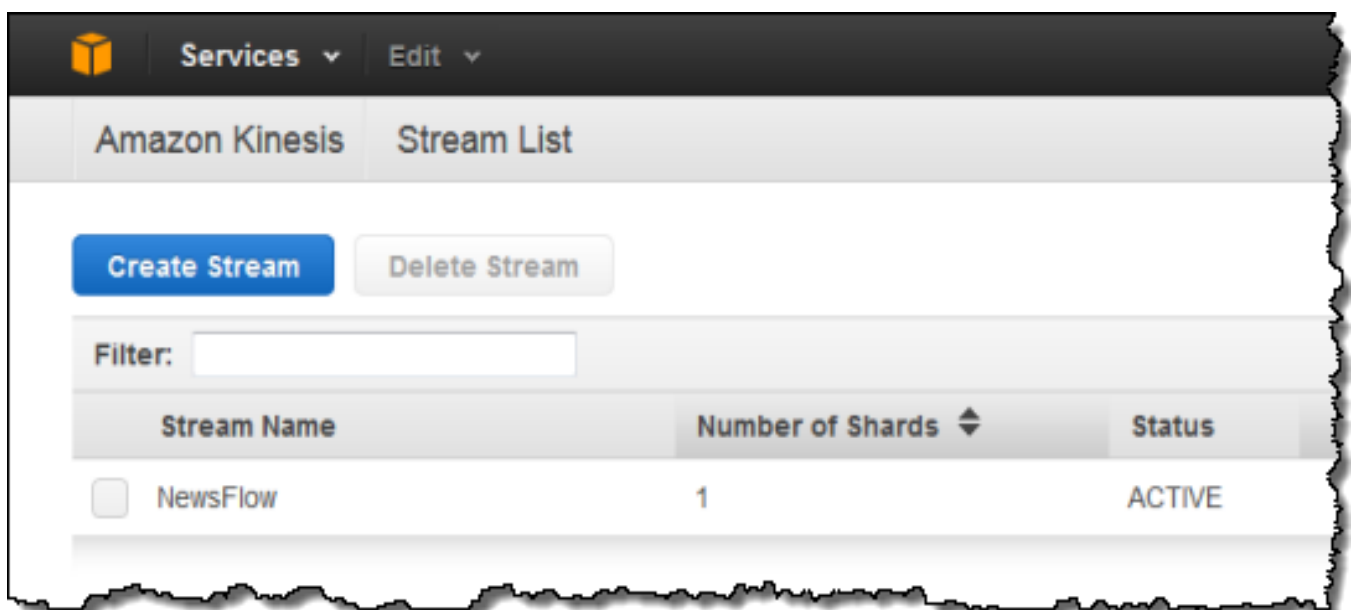
Volume of Data Read

Number of consumer applications:

Estimated Shards: Enter values above to estimate

The default shard limit for an account is 2. To raise the limit, see the [Developer Guide](#)

You can see all of your streams at a glance:



Stream Name	Number of Shards	Status
<input type="checkbox"/> NewsFlow	1	ACTIVE

And you can view the CloudWatch metrics for each stream:

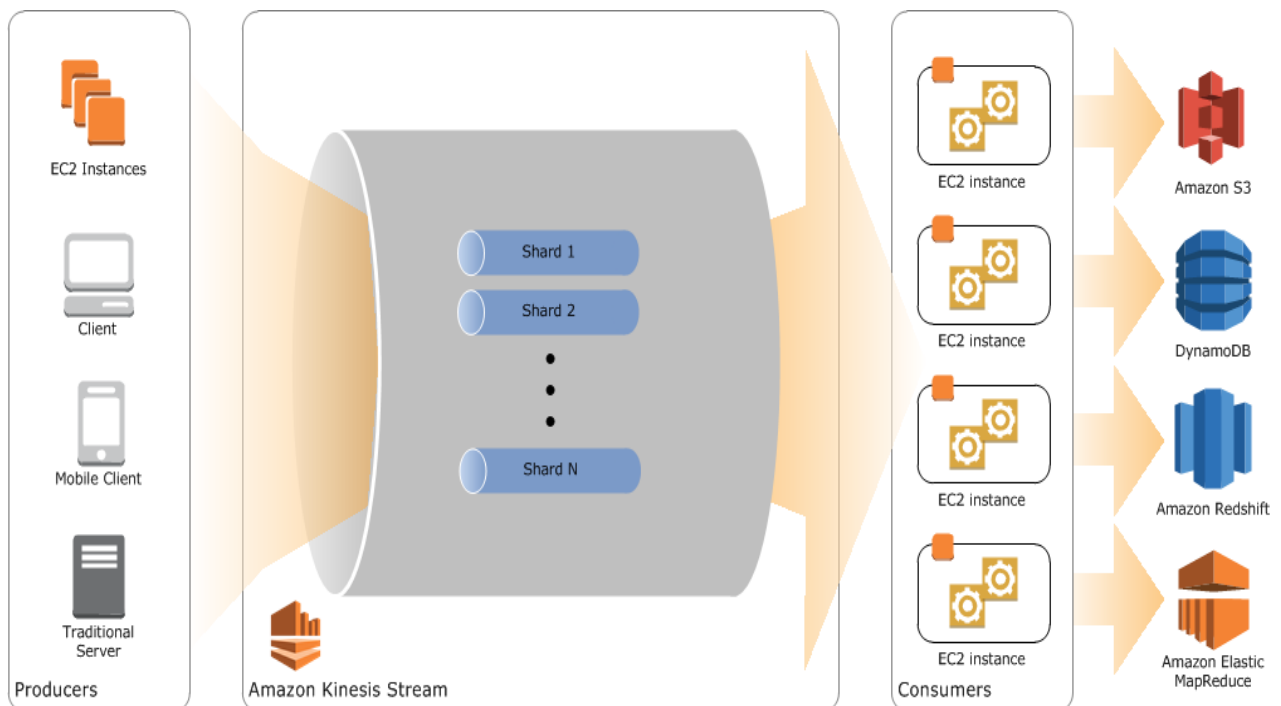


Amazon Kinesis Key Concepts

As you get started with Amazon Kinesis, you'll benefit from understanding its architecture and terminology.

Amazon Kinesis High-level Architecture

The following diagram illustrates the high-level architecture of Amazon Kinesis. The producers continually push data to Amazon Kinesis and the consumers process the data in real time. Consumers can store their results using an AWS service such as Amazon DynamoDB, Amazon Redshift, or Amazon S3.



Amazon Kinesis Terminology

Amazon Kinesis Streams

An Amazon Kinesis stream is an ordered sequence of data records. Each record in the stream has a sequence number that is assigned by Amazon Kinesis. The data records in the stream are distributed into shards.

Data Records

A data record is the unit of data stored in an Amazon Kinesis stream. Data records are composed of a sequence number, partition key, and data blob, which is an immutable sequence of bytes. Amazon Kinesis does not inspect, interpret, or change the data in the blob in any way. A data blob can be up to 1 MB.

Producers

Producers put records into Amazon Kinesis streams. For example, a web server sending log data to a stream is a producer.

Consumers

Consumers get records from Amazon Kinesis streams and process them. These consumers are known as Amazon Kinesis Applications.

Amazon Kinesis Applications

An Amazon Kinesis application is a consumer of an Amazon Kinesis stream that commonly runs on a fleet of EC2 instances.

You can develop an Amazon Kinesis application using the Amazon Kinesis Client Library or using the Amazon Kinesis API.

The output of an Amazon Kinesis application may be input for another Amazon Kinesis stream, enabling you to create complex topologies that process data in real time. An application can also send data to a variety of other AWS services. There can be multiple applications for one stream, and each application can consume data from the stream independently and concurrently.

Shards

A shard is a uniquely identified group of data records in an Amazon Kinesis stream. A stream is composed of multiple shards, each of which provides a fixed unit of capacity. Each shard can support up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second and up to 1,000 records per second for writes, up to a maximum total data write rate of 1 MB per second (including partition keys). The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards.

If your data rate increases, then you just add more shards to increase the size of your stream. Similarly, you can remove shards if the data rate decreases.

Partition Keys

A partition key is used to group data by shard within a stream. Amazon Kinesis segregates the data records belonging to a stream into multiple shards, using the partition key associated with each data record to determine which shard a given data record belongs to. Partition keys are Unicode strings with a maximum length limit of 256 bytes. An MD5 hash function is used to map partition keys to 128-bit integer values and to map associated data records to shards. A partition key is specified by the applications putting the data into a stream.

Sequence Numbers

Each data record has a unique sequence number. The sequence number is assigned by Amazon Kinesis after you write to the stream with `client.putRecords` or `client.putRecord`. Sequence numbers for the same partition key generally increase over time; the longer the time period between write requests, the larger the sequence numbers become.

Note

Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

Amazon Kinesis Client Library

The Amazon Kinesis Client Library is compiled into your application to enable fault-tolerant consumption of data from the Amazon Kinesis stream. The Amazon Kinesis Client Library ensures that for every shard there is a record processor running and processing that shard. The library also simplifies reading data from the Amazon Kinesis stream. The Amazon Kinesis Client Library uses an Amazon DynamoDB table to store control data. It creates one table per application that is processing data.

Application Name

The name of an Amazon Kinesis application identifies the application. Each of your applications must have a unique name that is scoped to the AWS account and region used by the application. This name is used as a name for the control table in Amazon DynamoDB and the namespace for Amazon CloudWatch metrics.



Apache Kafka

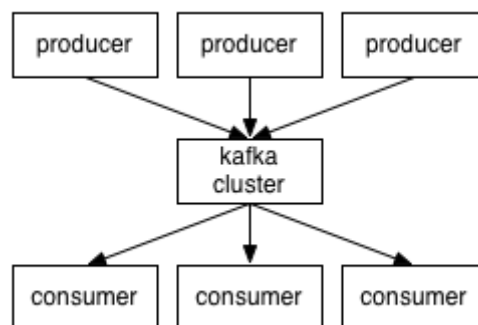
Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

What does all that mean?

First let's review some basic messaging terminology:

- Kafka maintains feeds of messages in categories called topics.
- We'll call processes that publish messages to a Kafka topic producers.
- We'll call processes that subscribe to topics and process the feed of published messages consumers..
- Kafka is run as a cluster comprised of one or more servers each of which is called a broker.

So, at a high level, producers send messages over the network to the Kafka cluster which in turn serves them up to consumers like this:

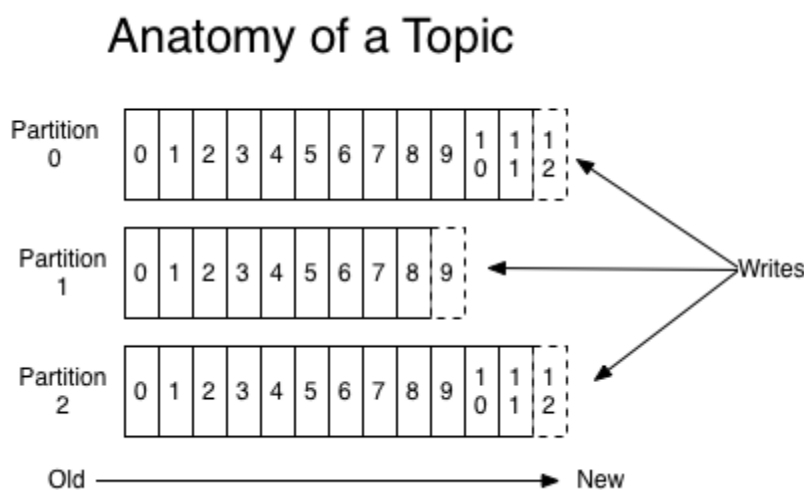


Communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. We provide a Java client for Kafka, but clients are available in many languages.

Topics and Logs

Let's first dive into the high-level abstraction Kafka provides—the topic.

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Each partition is an ordered, immutable sequence of messages that is continually appended to—a commit log. The messages in the partitions are each assigned a sequential id number called the offset that uniquely identifies each message within the partition.

The Kafka cluster retains all published messages—whether or not they have been consumed—for a configurable period of time. For example if the log retention is set to two days, then for the two days after a message is published it is available for consumption, after which it will be discarded to free up space. Kafka's performance is effectively constant with respect to data size so retaining lots of data is not a problem.

In fact the only metadata retained on a per-consumer basis is the position of the consumer in the log, called the "offset". This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads messages, but in fact the position is controlled by the

consumer and it can consume messages in any order it likes. For example a consumer can reset to an older offset to reprocess.

This combination of features means that Kafka consumers are very cheap—they can come and go without much impact on the cluster or on other consumers. For example, you can use our command line tools to "tail" the contents of any topic without changing what is consumed by any existing consumers.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism—more on that in a bit.

Distribution

The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.

Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which message to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the message). More on the use of partitioning in a second.

Consumers

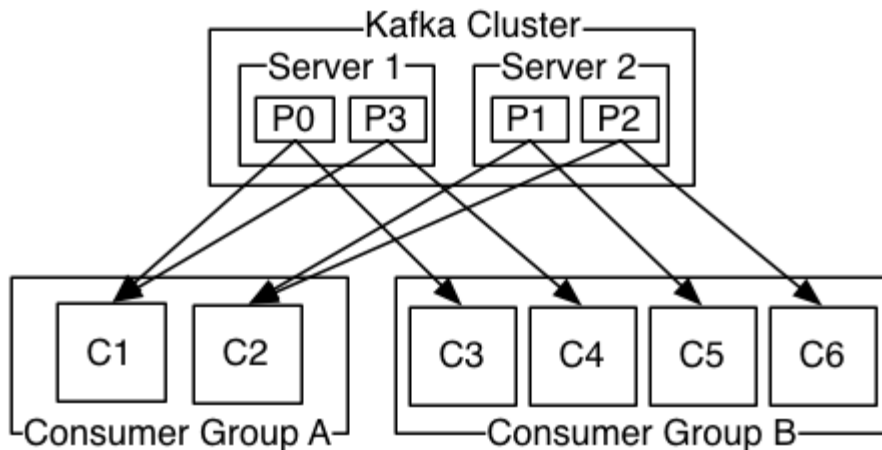
Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers. Kafka offers a single consumer abstraction that generalizes both of these—the consumer group.

Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.

If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers.

If all the consumer instances have different consumer groups, then this works like publish-subscribe and all messages are broadcast to all consumers.

More commonly, however, we have found that topics have a small number of consumer groups, one for each "logical subscriber". Each group is composed of many consumer instances for scalability and fault tolerance. This is nothing more than publish-subscribe semantics where the subscriber is cluster of consumers instead of a single process.



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups.

Consumer group A has two consumer instances and group B has four.

Kafka has stronger ordering guarantees than a traditional messaging system, too.

A traditional queue retains messages in-order on the server, and if multiple consumers consume from the queue then the server hands out messages in the order they are stored. However, although the server hands out messages in order, the messages are delivered asynchronously to consumers, so they may arrive out of order on different consumers. This effectively means the ordering of the messages is lost in the presence of parallel consumption. Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is no parallelism in processing.

Kafka does it better. By having a notion of parallelism—the partition—within the topics, Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer instances than partitions.

Kafka only provides a total order over messages within a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over messages this can be achieved with a topic that has only one partition, though this will mean only one consumer process.

Guarantees

At a high-level Kafka gives the following guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a message M1 is sent by the same producer as a message M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any messages committed to the log.

More details on these guarantees are given in the design section of the documentation.

Use Cases

Here is a description of a few of the popular use cases for Apache Kafka.

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

In this domain Kafka is comparable to traditional messaging systems such as ActiveMQ or RabbitMQ.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view.

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

Many users end up doing stage-wise processing of data where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption. For example a processing flow for article recommendation might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage

might attempt to match this content to users. This creates a graph of real-time data flow out of the individual topics. Storm and Samza are popular frameworks for implementing these kinds of transformations.

Event Sourcing

Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log compaction feature in Kafka helps support this usage



Apache Samza

Apache Samza is a distributed stream processing framework. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management.

It has the following features:

- **Simple API:** Unlike most low-level messaging system APIs, Samza provides a very simple callback-based “process message” API comparable to MapReduce.
- **Managed state:** Samza manages snapshotting and restoration of a stream processor’s state. When the processor is restarted, Samza restores its state to a consistent snapshot. Samza is built to handle large amounts of state (many gigabytes per partition).
- **Fault tolerance:** Whenever a machine in the cluster fails, Samza works with YARN to transparently migrate your tasks to another machine.
- **Durability:** Samza uses Kafka to guarantee that messages are processed in the order they were written to a partition, and that no messages are ever lost.
- **Scalability:** Samza is partitioned and distributed at every level. Kafka provides ordered, partitioned, replayable, fault-tolerant streams. YARN provides a distributed environment for Samza containers to run in.
- **Pluggable:** Though Samza works out of the box with Kafka and YARN, Samza provides a pluggable API that lets you run Samza with other messaging systems and execution environments.
- **Processor isolation:** Samza works with Apache YARN, which supports Hadoop’s security model, and resource isolation through Linux CGroups.

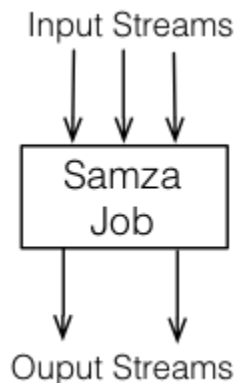
Concepts

Streams

Samza processes *streams*. A stream is composed of immutable *messages* of a similar type or category. For example, a stream could be all the clicks on a website, or all the updates to a

particular database table, or all the logs produced by a service, or any other type of event data. Messages can be appended to a stream or read from a stream. A stream can have any number of *consumers*, and reading from a stream doesn't delete the message (so each message is effectively broadcast to all consumers). Messages can optionally have an associated key which is used for partitioning, which we'll talk about in a second.

Samza supports pluggable *systems* that implement the stream abstraction: in Kafka a stream is a topic, in a database we might read a stream by consuming updates from a table, in Hadoop we might tail a directory of files in HDFS.



Jobs

A Samza job is code that performs a logical transformation on a set of input streams to append output messages to set of output streams.

If scalability were not a concern, streams and jobs would be all we need. However, in order to scale the throughput of the stream processor, we chop streams and jobs up into smaller units of parallelism: partitions and tasks.

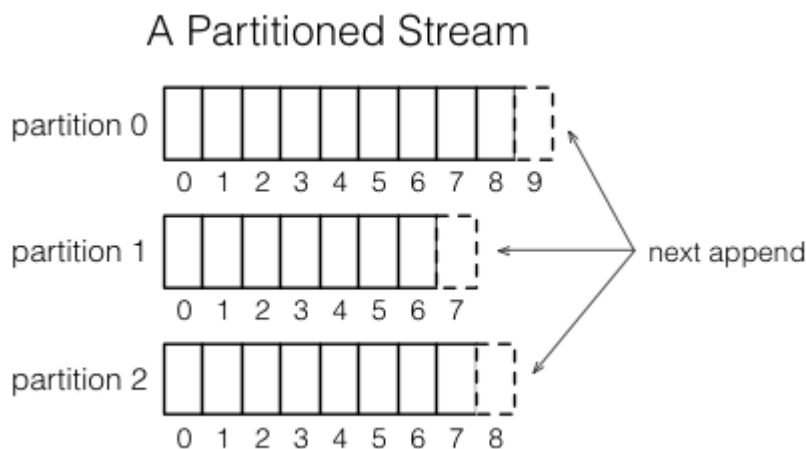
Partitions

Each stream is broken into one or more partitions. Each partition in the stream is a totally ordered sequence of messages.

Each message in this sequence has an identifier called the *offset*, which is unique per partition. The offset can be a sequential integer, byte offset, or string depending on the underlying system implementation.

When a message is appended to a stream, it is appended to only one of the stream's partitions. The assignment of the message to its partition is done with a key chosen by the writer. For

example, if the user ID is used as the key, that ensures that all messages related to a particular user end up in the same partition.



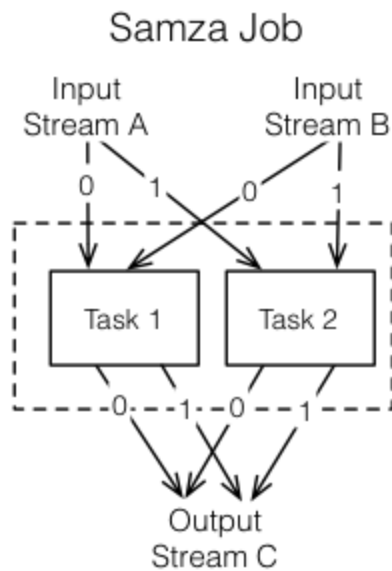
Tasks

A job is scaled by breaking it into multiple *tasks*. The *task* is the unit of parallelism of the job, just as the partition is to the stream. Each task consumes data from one partition for each of the job's input streams.

A task processes messages from each of its input partitions sequentially, in the order of message offset. There is no defined ordering across partitions. This allows each task to operate independently. The YARN scheduler assigns each task to a machine, so the job as a whole can be distributed across many machines.

The number of tasks in a job is determined by the number of input partitions (there cannot be more tasks than input partitions, or there would be some tasks with no input). However, you can change the computational resources assigned to the job (the amount of memory, number of CPU cores, etc.) to satisfy the job's needs. See notes on *containers* below.

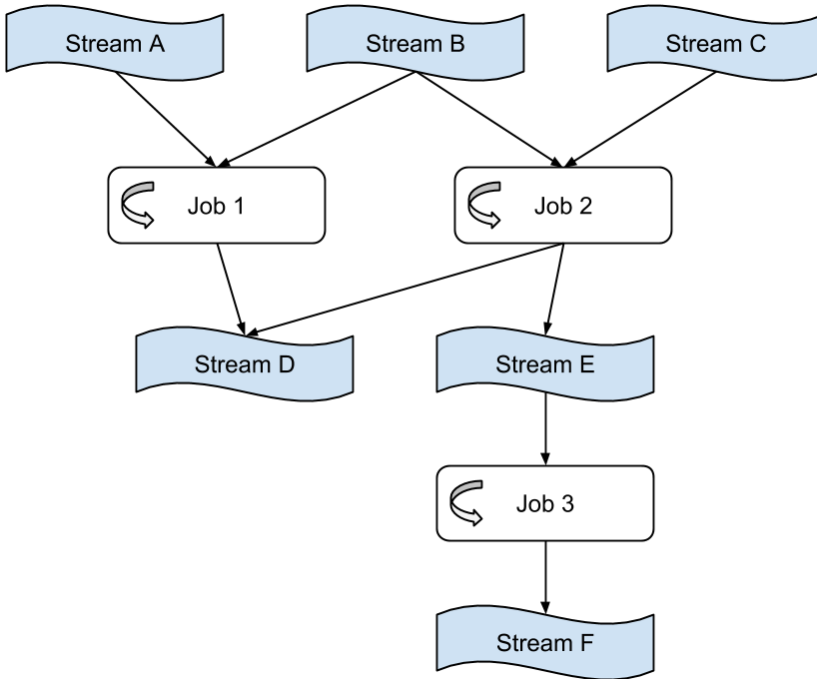
The assignment of partitions to tasks never changes: if a task is on a machine that fails, the task is restarted elsewhere, still consuming the same stream partitions.



Dataflow Graphs

We can compose multiple jobs to create a dataflow graph, where the nodes are streams containing data, and the edges are jobs performing transformations. This composition is done purely through the streams the jobs take as input and output. The jobs are otherwise totally decoupled: they need not be implemented in the same code base, and adding, removing, or restarting a downstream job will not impact an upstream job.

These graphs are often acyclic—that is, data usually doesn't flow from a job, through other jobs, back to itself. However, it is possible to create cyclic graphs if you need to.



Containers

Partitions and tasks are both *logical* units of parallelism—they don't correspond to any particular assignment of computational resources (CPU, memory, disk space, etc). Containers are the unit of physical parallelism, and a container is essentially a Unix process (or Linux cgroup). Each container runs one or more tasks. The number of tasks is determined automatically from the number of partitions in the input and is fixed, but the number of containers (and the CPU and memory resources associated with them) is specified by the user at run time and can be changed at any time.

Architecture

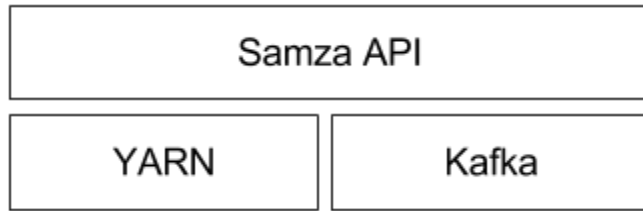
Samza is made up of three layers:

1. A streaming layer.
2. An execution layer.
3. A processing layer.

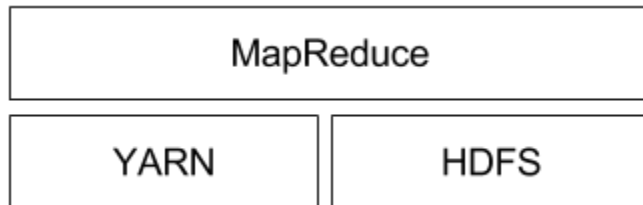
Samza provides out of the box support for all three layers.

1. **Streaming:** [Kafka](#)
2. **Execution:** [YARN](#)
3. **Processing:** [Samza API](#)

These three pieces fit together to form Samza:



This architecture follows a similar pattern to Hadoop (which also uses YARN as execution layer, HDFS for storage, and MapReduce as processing API):



Before going in-depth on each of these three layers, it should be noted that Samza's support is not limited to Kafka and YARN. Both Samza's execution and streaming layer are pluggable, and allow developers to implement alternatives if they prefer.

Kafka

Kafka is a distributed pub/sub and message queueing system that provides at-least once messaging guarantees (i.e. the system guarantees that no messages are lost, but in certain fault scenarios, a consumer might receive the same message more than once), and highly available partitions (i.e. a stream's partitions continue to be available even if a machine goes down).

In Kafka, each stream is called a *topic*. Each topic is partitioned and replicated across multiple machines called *brokers*. When a *producer* sends a message to a topic, it provides a key, which is used to determine which partition the message should be sent to. The Kafka brokers receive and store the messages that the producer sends. Kafka *consumers* can then read from a topic by subscribing to messages on all partitions of a topic.

Kafka has some interesting properties:

- All messages with the same key are guaranteed to be in the same topic partition. This means that if you wish to read all messages for a specific user ID, you only have to read the messages from the partition that contains the user ID, not the whole topic (assuming the user ID is used as key).

- A topic partition is a sequence of messages in order of arrival, so you can reference any message in the partition using a monotonically increasing *offset* (like an index into an array). This means that the broker doesn't need to keep track of which messages have been seen by a particular consumer — the consumer can keep track itself by storing the offset of the last message it has processed. It then knows that every message with a lower offset than the current offset has already been processed; every message with a higher offset has not yet been processed.

For more details on Kafka, see Kafka's [documentation](#) pages.

YARN

YARN (Yet Another Resource Negotiator) is Hadoop's next-generation cluster scheduler. It allows you to allocate a number of *containers* (processes) in a cluster of machines, and execute arbitrary commands on them.

When an application interacts with YARN, it looks something like this:

1. **Application:** I want to run command X on two machines with 512MB memory.
2. **YARN:** Cool, where's your code?
3. **Application:** `http://path.to.host/jobs/download/my.tgz`
4. **YARN:** I'm running your job on node-1.grid and node-2.grid.

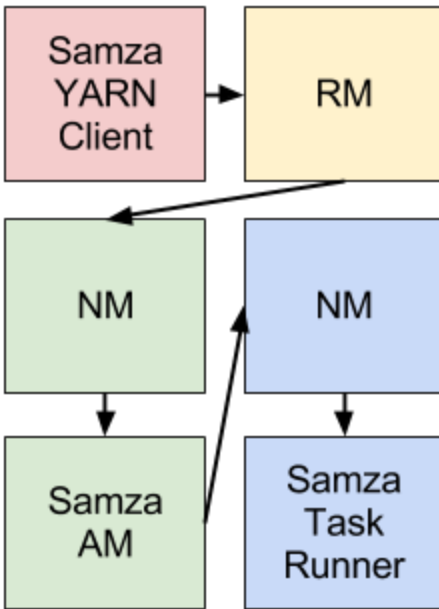
Samza uses YARN to manage deployment, fault tolerance, logging, resource isolation, security, and locality. A brief overview of YARN is below; see this page from Hortonworks for a much better overview.

YARN Architecture

YARN has three important pieces: a *ResourceManager*, a *NodeManager*, and an *ApplicationMaster*. In a YARN grid, every machine runs a NodeManager, which is responsible for launching processes on that machine. A ResourceManager talks to all of the NodeManagers to tell them what to run. Applications, in turn, talk to the ResourceManager when they wish to run something on the cluster. The third piece, the ApplicationMaster, is actually application-specific code that runs in the YARN cluster. It's responsible for managing the application's workload, asking for containers (usually UNIX processes), and handling notifications when one of its containers fails.

Samza and YARN

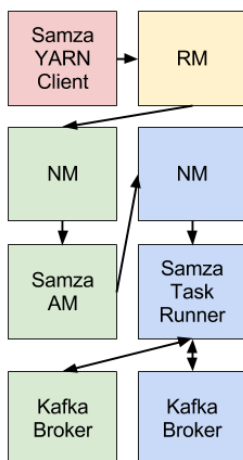
Samza provides a YARN ApplicationMaster and a YARN job runner out of the box. The integration between Samza and YARN is outlined in the following diagram (different colors indicate different host machines):



The Samza client talks to the YARN RM when it wants to start a new Samza job. The YARN RM talks to a YARN NM to allocate space on the cluster for Samza’s ApplicationMaster. Once the NM allocates space, it starts the Samza AM. After the Samza AM starts, it asks the YARN RM for one or more YARN containers to run SamzaContainers. Again, the RM works with NMs to allocate space for the containers. Once the space has been allocated, the NMs start the Samza containers.

Samza

Samza uses YARN and Kafka to provide a framework for stage-wise stream processing and partitioning. Everything, put together, looks like this (different colors indicate different host machines):



The Samza client uses YARN to run a Samza job: YARN starts and supervises one or more SamzaContainers, and your processing code (using the StreamTask API) runs inside those containers. The input and output for the Samza StreamTasks come from Kafka brokers that are (usually) co-located on the same machines as the YARN NMs.

Kafka Performance

To actually make this work, though, this "universal log" has to be a cheap abstraction. If you want to use a system as a central data hub it has to be fast, predictable, and easy to scale so you can dump all your data onto it. My experience has been that systems that are fragile or expensive inevitably develop a wall of protective process to prevent people from using them; a system that scales easily often ends up as a key architectural building block just because using it is the easiest way to get things built.

The benchmarks of Cassandra show that it does a million writes per second on three hundred machines on [EC2](#) and [Google Compute Engine](#). I'm not sure why, maybe it is a [Dr. Evil](#) thing, but doing [a million](#) of anything per second is fun.

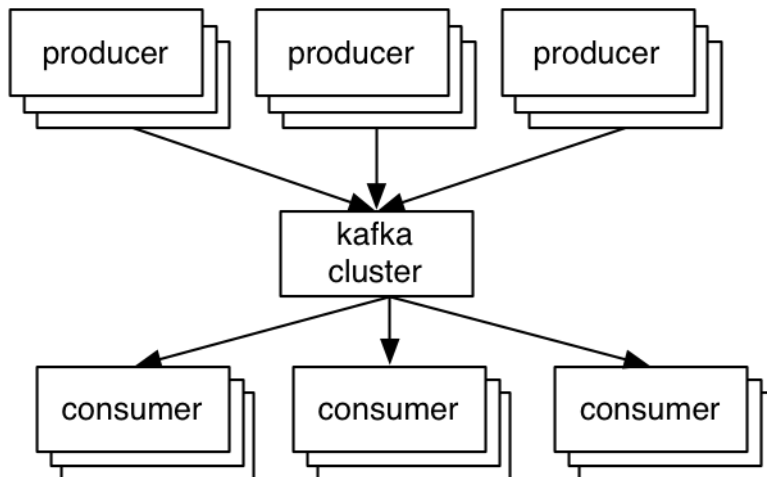
In any case, one of the nice things about a Kafka log is that, as we'll see, it is cheap. A million writes per second isn't a particularly big thing. This is because a log is a much simpler thing than a database or key-value store. Indeed our production clusters take tens of millions of reads and writes per second all day long and they do so on pretty modest hardware.

But let's do some benchmarking and take a look.

Kafka in 30 seconds

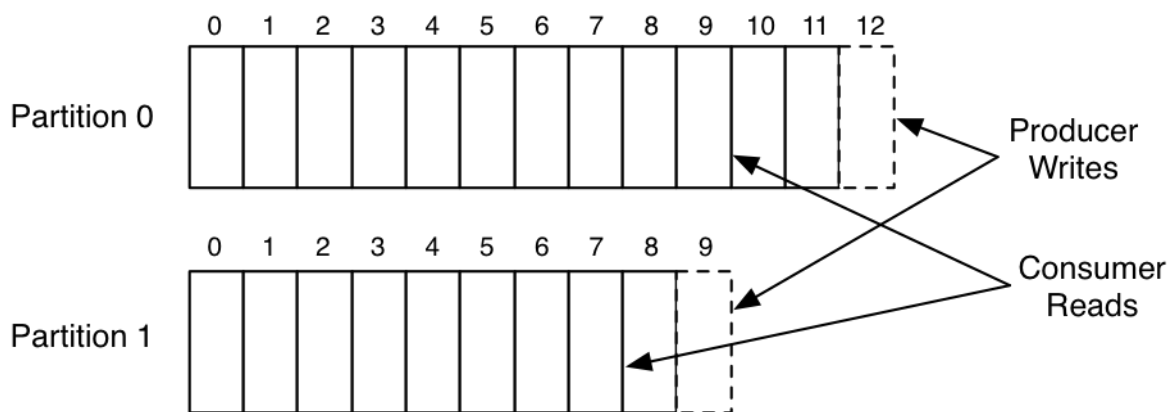
To help understand the benchmark, let me give a quick review of what Kafka is and a few details about how it works. Kafka is a distributed messaging system originally built at LinkedIn and now part of the [Apache Software Foundation](#) and [used](#) by a [variety of companies](#).

The general setup is quite simple. Producers send records to the cluster which holds on to these records and hands them out to consumers:



The key abstraction in Kafka is the topic. Producers publish their records to a topic, and consumers subscribe to one or more topics. A Kafka topic is just a sharded write-ahead log. Producers append records to these logs and consumers subscribe to changes. Each record is a key/value pair. The key is used for assigning the record to a log partition (unless the publisher specifies the partition directly).

Here is a simple example of a single producer and consumer reading and writing from a two-partition topic.



This picture shows a producer process appending to the logs for the two partitions, and a consumer reading from the same logs. Each record in the log has an associated entry number that we call the offset. This offset is used by the consumer to describe its position in each of the logs.

These partitions are spread across a cluster of machines, allowing a topic to hold more data than can fit on any one machine.

Note that unlike most messaging systems the log is always persistent. Messages are immediately written to the filesystem when they are received. Messages are not deleted when they are read but retained with some configurable SLA (say a few days or a week). This allows

usage in situations where the consumer of data may need to reload data. It also makes it possible to support space-efficient publish-subscribe as there is a single shared log no matter how many consumers; in traditional messaging systems there is usually a queue per consumer, so adding a consumer doubles your data size. This makes Kafka a good fit for things outside the bounds of normal messaging systems such as acting as a pipeline for offline data systems such as Hadoop. These offline systems may load only at intervals as part of a periodic ETL cycle, or may go down for several hours for maintenance, during which time Kafka is able to buffer even TBs of unconsumed data if needed.

Kafka also replicates its logs over multiple servers for fault-tolerance. One important architectural aspect of our [replication implementation](#), in contrast to other messaging systems, is that replication is not an exotic bolt-on that requires complex configuration, only to be used in very specialized cases. Instead replication is assumed to be the default: we treat un-replicated data as a special case where the replication factor happens to be one. Producers get an acknowledgement back when they publish a message containing the record's offset. The first record published to a partition is given the offset 0, the second record 1, and so on in an ever-increasing sequence. Consumers consume data from a position specified by an offset, and they save their position in a log by committing periodically: saving this offset in case that consumer instance crashes and another instance needs to resume from it's position. Okay, hopefully that all made sense (if not, you can read a more complete introduction to Kafka [here](#)).

This Benchmark

This test is against trunk, as I made some improvements to the performance tests for this benchmark. But nothing too substantial has changed since the last full release, so you should see similar results with [0.8.1](#). I am also using our newly re-written [Java producer](#), which offers much improved throughput over the previous producer client.

I've followed the basic template of this very nice [RabbitMQ benchmark](#), but I covered scenarios and options that were more relevant to Kafka.

One quick philosophical note on this benchmark. For benchmarks that are going to be publicly reported, I like to follow a style I call "lazy benchmarking". When you work on a system, you generally have the know-how to tune it to perfection for any particular use case. This leads to a kind of benchmarking where you heavily tune your configuration to your benchmark or worse have a different tuning for each scenario you test. I think the real test of a system is not how it performs when perfectly tuned, but rather how it performs "off the shelf". This is particularly true for systems that run in a multi-tenant setup with dozens or hundreds of use cases where tuning for each use case would be not only impractical but impossible. As a result, I have pretty much stuck with default settings, both for the server and the clients. I will point out areas where I suspect the result could be improved with a little tuning, but I have tried to resist the temptation to do any fiddling myself to improve the results.

I have posted [my exact configurations and commands](#), so it should be possible to replicate results on your own gear if you are interested.

The Setup

For these tests, I had six machines each has the following specs

- Intel Xeon 2.5 GHz processor with six cores
- Six 7200 RPM SATA drives
- 32GB of RAM
- 1Gb Ethernet

The Kafka cluster is set up on three of the machines. The six drives are directly mounted with no RAID (JBOD style). The remaining three machines I use for Zookeeper and for generating load. A three machine cluster isn't very big, but since we will only be testing up to a replication factor of three, it is all we need. As should be obvious, we can always add more partitions and spread data onto more machines to scale our cluster horizontally.

This hardware is actually not LinkedIn's normal Kafka hardware. Our Kafka machines are more closely tuned to running Kafka, but are less in the spirit of "off-the-shelf" I was aiming for with these tests. Instead, I borrowed these from one of our Hadoop clusters, which runs on probably the cheapest gear of any of our persistent systems. Hadoop usage patterns are pretty similar to Kafka's, so this is a reasonable thing to do.

Okay, without further ado, the results!

Producer Throughput

These tests will stress the throughput of the producer. No consumers are run during these tests, so all messages are persisted but not read (we'll test cases with both producer and consumer in a bit). Since we have recently rewritten our producer, I am testing this new code.

Single producer thread, no replication

821,557 records/sec
(78.3 MB/sec)

For this first test I create a topic with six partitions and no replication. Then I produce 50 million small (100 byte) records as quickly as possible from a single thread.

The reason for focusing on small records in these tests is that it is the harder case for a messaging system (generally). It is easy to get good throughput in MB/sec if the messages are large, but much harder to get good throughput when the messages are small, as the overhead of processing each message dominates.

Throughout this benchmark, when I am reporting MB/sec, I am reporting just the value size of the record times the request per second, none of the other overhead of the request is included. So the actually network usage is higher than what is reported. For example with a 100 byte message we would also transmit about 22 bytes of overhead per message (for an optional key, size delimiting, a message CRC, the record offset, and attributes flag), as well as some overhead

for the request (including the topic, partition, required acknowledgements, etc). This makes it a little harder to see where we hit the limits of the NIC, but this seems a little more reasonable then including our own overhead bytes in throughput numbers. So, in the above result, we are likely saturating the 1 gigabit NIC on the client machine.

One immediate observation is that the raw numbers here are much higher than people expect, especially for a persistent storage system. If you are used to random-access data systems, like a database or key-value store, you will generally expect maximum throughput around 5,000 to 50,000 queries-per-second, as this is close to the speed that a good RPC layer can do remote requests. We exceed this due to two key design principles:

- 1 We work hard to ensure we do linear disk I/O. The six cheap disks these servers have gives an aggregate throughput of 822 MB/sec of linear disk I/O. This is actually well beyond what we can make use of with only a 1 gigabit network card. Many messaging systems treat persistence as an expensive add-on that decimates performance and should be used only sparingly, but this is because they are not able to do linear I/O.
- 2 At each stage we work on batching together small bits of data into larger network and disk I/O operations. For example, in the new producer we use a "group commit"-like mechanism to ensure that any record sends initiated while another I/O is in progress get grouped together. For more on understanding the importance of batching, check out this presentation by David Patterson on why ["Latency Lags Bandwidth"](#).

Single producer thread, 3x asynchronous replication

786,980 records/sec
(75.1 MB/sec)

This test is exactly the same as the previous one except that now each partition has three replicas (so the total data written to network or disk is three times higher). Each server is doing both writes from the producer for the partitions for which it is a master, as well as fetching and writing data for the partitions for which it is a follower.

Replication in this test is asynchronous. That is, the server acknowledges the write as soon as it has written it to its local log without waiting for the other replicas to also acknowledge it. This means, if the master were to crash, it would likely lose the last few messages that had been written but not yet replicated. This makes the message acknowledgement latency a little better at the cost of some risk in the case of server failure.

The key take away I would like people to have from this is that replication can be fast. The total cluster write capacity is, of course, 3x less with 3x replication (since each write is done three times), but the throughput is still quite good per client. High performance replication comes in large part from the efficiency of our consumer (the replicas are really nothing more than a specialized consumer) which I will discuss in the consumer section.

Single producer thread, 3x synchronous replication

421,823 records/sec
(40.2 MB/sec)

This test is the same as above except that now the master for a partition waits for acknowledgement from the full set of in-sync replicas before acknowledging back to the producer. In this mode, we guarantee that messages will not be lost as long as one in-sync replica remains.

Synchronous replication in Kafka is not fundamentally very different from asynchronous replication. The leader for a partition always tracks the progress of the follower replicas to monitor their liveness, and we never give out messages to consumers until they are fully acknowledged by replicas. With synchronous replication we just wait to respond to the producer request until the followers have replicated it.

This additional latency does seem to affect our throughput. Since the code path on the server is very similar, we could probably ameliorate this impact by tuning the batching to be a bit more aggressive and allowing the client to buffer more outstanding requests. However, in spirit of avoiding special case tuning, I have avoided this.

Three producers, 3x async replication

2,024,032 records/sec
(193.0 MB/sec)

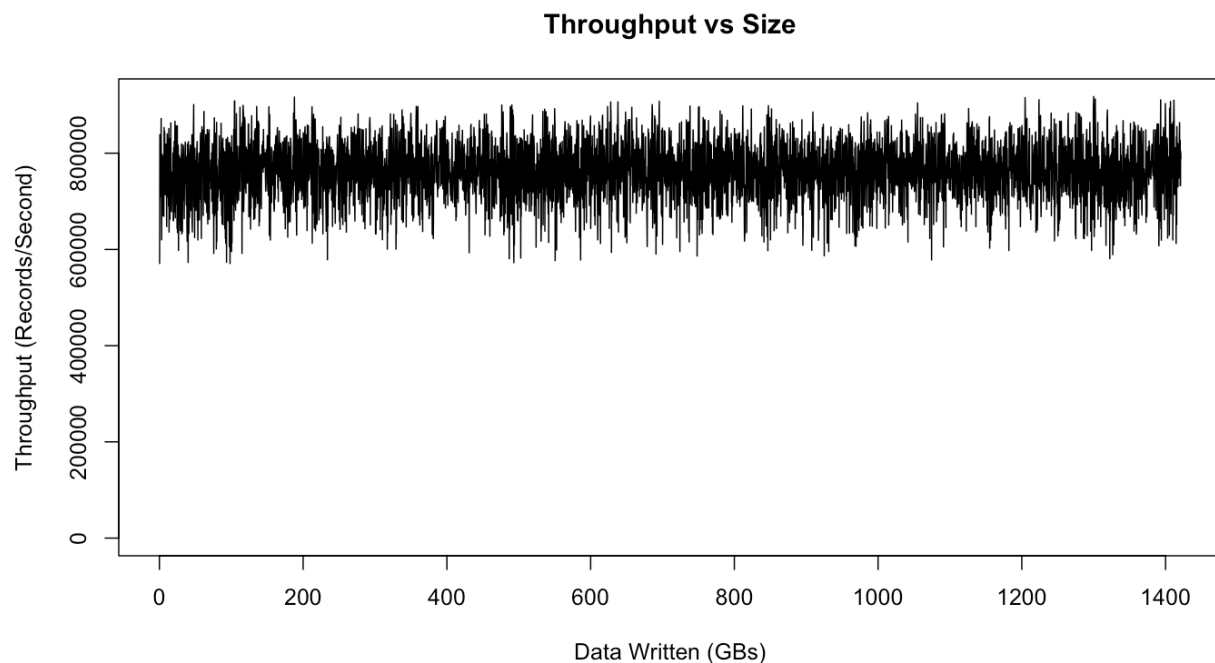
Our single producer process is clearly not stressing our three node cluster. To add a little more load, I'll now repeat the previous async replication test, but now use three producer load generators running on three different machines (running more processes on the same machine won't help as we are saturating the NIC). Then we can look at the aggregate throughput across these three producers to get a better feel for the cluster's aggregate capacity.

Producer Throughput Versus Stored Data

One of the hidden dangers of many messaging systems is that they work well only as long as the data they retain fits in memory. Their throughput falls by an order of magnitude (or more) when data backs up and isn't consumed (and hence needs to be stored on disk). This means things may be running fine as long as your consumers keep up and the queue is empty, but as soon as they lag, the whole messaging layer backs up with unconsumed data. The backup causes data to go to disk which in turns causes performance to drop to a rate that means messaging system can no longer keep up with incoming data and either backs up or falls over. This is pretty terrible, as in many cases the whole purpose of the queue was to handle such a case gracefully.

Since Kafka always persists messages the performance is $O(1)$ with respect to unconsumed data volume.

To test this experimentally, let's run our throughput test over an extended period of time and graph the results as the stored dataset grows:



This graph actually does show some variance in performance, but no impact due to data size: we perform just as well after writing a TB of data, as we do for the first few hundred MBs. The variance seems to be due to Linux's I/O management facilities that batch data and then flush it periodically. This is something we have tuned for a little better on our production Kafka setup.

Consumer Throughput

Okay now let's turn our attention to consumer throughput.

Note that the replication factor will not effect the outcome of this test as the consumer only reads from one replica regardless of the replication factor. Likewise, the acknowledgement level of the producer also doesn't matter as the consumer only ever reads fully acknowledged messages, (even if the producer doesn't wait for full acknowledgement). This is to ensure that any message the consumer sees will always be present after a leadership handoff (if the current leader fails).

Single Consumer

940,521 records/sec
(89.7 MB/sec)

For the first test, we will consume 50 million messages in a single thread from our 6 partition 3x replicated topic.

Kafka's consumer is very efficient. It works by fetching chunks of log directly from the filesystem. It uses the [sendfile API](#) to transfer this directly through the operating system without the overhead of copying this data through the application. This test actually starts at the beginning of the log, so it is doing real read I/O. In a production setting, though, the consumer reads almost exclusively out of the OS pagecache, since it is reading data that was just written by some producer (so it is still cached). In fact, if you run I/O stat on a production server you actually see that there are no physical reads at all even though a great deal of data is being consumed.

Making consumers cheap is important for what we want Kafka to do. For one thing, the replicas are themselves consumers, so making the consumer cheap makes replication cheap. In addition, this makes handling out data an inexpensive operation, and hence not something we need to tightly control for scalability reasons.

Three Consumers

2,615,968 records/sec
(249.5 MB/sec)

Let's repeat the same test, but run three parallel consumer processes, each on a different machine, and all consuming the same topic.

As expected, we see near linear scaling (not surprising because consumption in our model is so simple).

Producer and Consumer

795,064 records/sec
(75.8 MB/sec)

The above tests covered just the producer and the consumer running in isolation. Now let's do the natural thing and run them together. Actually, we have technically already been doing this, since our replication works by having the servers themselves act as consumers.

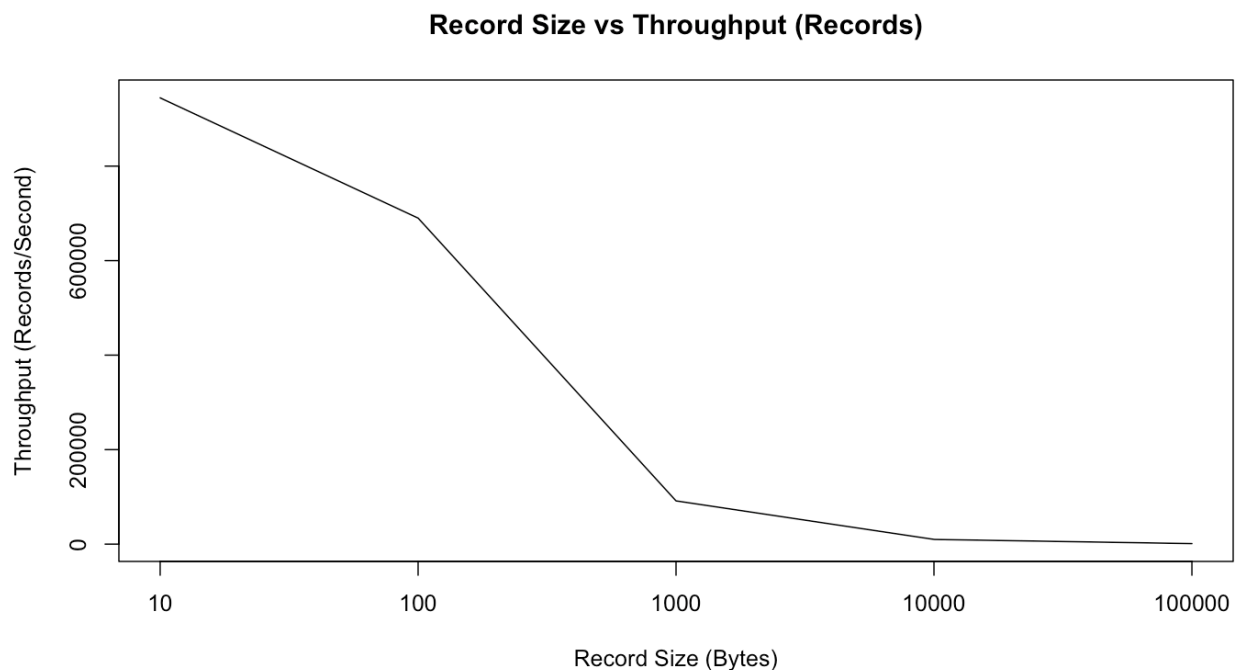
All the same, let's run the test. For this test we'll run one producer and one consumer on a six partition 3x replicated topic that begins empty. The producer is again using async replication. The throughput reported is the consumer throughput (which is, obviously, an upper bound on

the producer throughput).

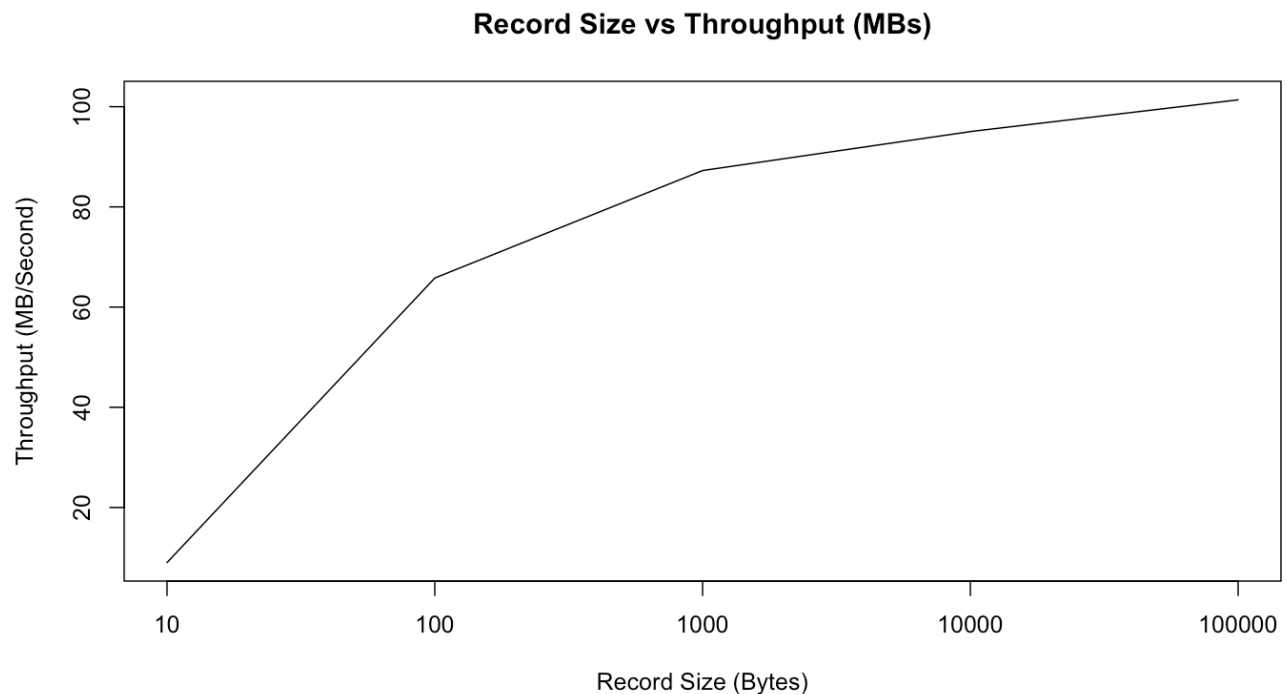
As we would expect, the results we get are basically the same as we saw in the producer only case—the consumer is fairly cheap.

Effect of Message Size

I have mostly shown performance on small 100 byte messages. Smaller messages are the harder problem for a messaging system as they magnify the overhead of the bookkeeping the system does. We can show this by just graphing throughput in both records/second and MB/second as we vary the record size.



So, as we would expect, this graph shows that the raw count of records we can send per second decreases as the records get bigger. But if we look at MB/second, we see that the total byte throughput of real user data increases as messages get bigger:



We can see that with the 10 byte messages we are actually CPU bound by just acquiring the lock and enqueueing the message for sending—we are not able to actually max out the network. However, starting with 100 bytes, we are actually seeing network saturation (though the MB/sec continues to increase as our fixed-size bookkeeping bytes become an increasingly small percentage of the total bytes sent).

End-to-end Latency

2 ms (median)
3 ms (99th percentile)
14 ms (99.9th percentile)

We have talked a lot about throughput, but what is the latency of message delivery? That is, how long does it take a message we send to be delivered to the consumer? For this test, we will create producer and consumer and repeatedly time how long it takes for a producer to send a message to the kafka cluster and then be received by our consumer.

Note that, Kafka only gives out messages to consumers when they are acknowledged by the full in-sync set of replicas. So this test will give the same results regardless of whether we use sync or async replication, as that setting only affects the acknowledgement to the producer.

Comparison between the tools

Apache FLUME	AWS KINESIS	Apache Samza
Processing large amounts of log centric data	Processes website clickstreams, social media feeds, IT logs, and location-tracking events	Distributed stream processing framework. Kafka – messaging YARN - fault tolerance, security
Variation in transaction speeds for multi hop when sink (read) takes longer than source (write)	Each shard can support up to 1 MB/second read , up to 2 MB/second write operation	Producer pushes 50MB/sec to the system Consumer can read 100MB/sec

Conclusion

The data ingestion systems *Apache Flume*, *Amazon Kinesis*, *Apache Kafka* and *Apache Samza* were explored, analyzed and compared.

References

- * <https://aws.amazon.com/blogs/aws/amazon-kinesis-real-time-processing-of-streamed-data/>
- * <http://docs.aws.amazon.com/kinesis/latest/dev/key-concepts.html>
- * <https://spark.apache.org/docs/latest/streaming-kinesis-integration.html>
- * <https://flume.apache.org/FlumeUserGuide.html>
- * <https://spark.apache.org/docs/latest/streaming-flume-integration.html>
- * <https://flume.apache.org/releases/content/1.2.0/FlumeUserGuide.pdf>
- * <https://samza.apache.org/learn/documentation/0.9/introduction/background.html>
- * <http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>
- * <http://kafka.apache.org/07/performance.html>