IoT Big Data Processing

Apache Kafka, Samza, Storm

Albert Bifet
albert.bifet@telecom-paristech.fr

October 18, 2016

Stream Engine Motivation

Digital Universe

EMC Digital Universe with Research & Analysis by IDC

The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things

April 2014

Digital Universe

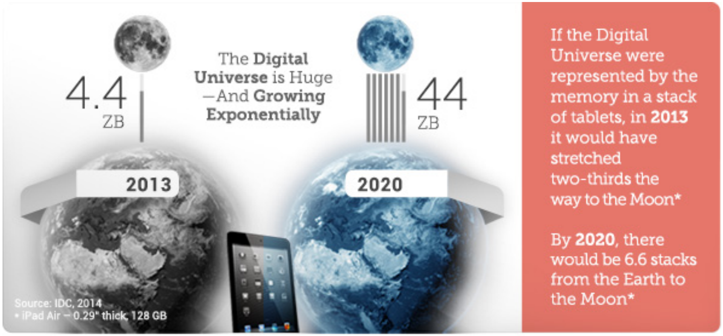

4.4 ZB
The Digital Universe is Huge —And Growing Exponentially
44 ZB
2013
2020
If the Digital Universe were represented by the memory in a stack of tablets, in 2013 it would have stretched two-thirds the way to the Moon*
By 2020, there would be 6.6 stacks from the Earth to the Moon*
Source: IDC, 2014
• iPad Air – 0.29" thick, 128 GB

Figure: EMC Digital Universe, 2014

Digital Universe

Memory unit	Size	Binary size
kilobyte (kB/KB)	10^3	2^{10}
megabyte (MB)	10^6	2^{20}
gigabyte (GB)	10^9	2^{30}
terabyte (TB)	10^{12}	2^{40}
petabyte (PB)	10^{15}	2^{50}
exabyte (EB)	10^{18}	2^{60}
zettabyte (ZB)	10^{21}	2^{70}
yottabyte (YB)	10^{24}	2^{80}

Digital Universe

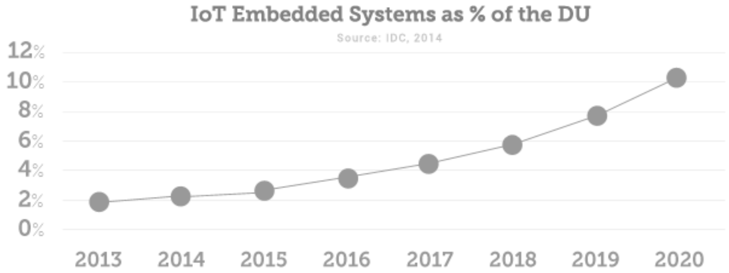


Figure: EMC Digital Universe, 2014

Digital Universe

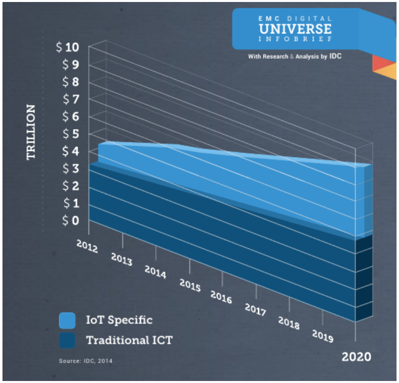


Figure: EMC Digital Universe, 2014

Big Data 6V’s

- Volume
- Variety
- Velocity
- Value
- Variability
- Veracity

Hadoop

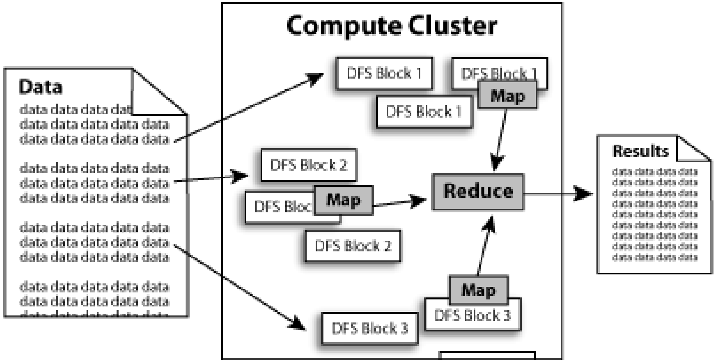


Figure: Hadoop architecture deals with datasets, not data streams

Requirements

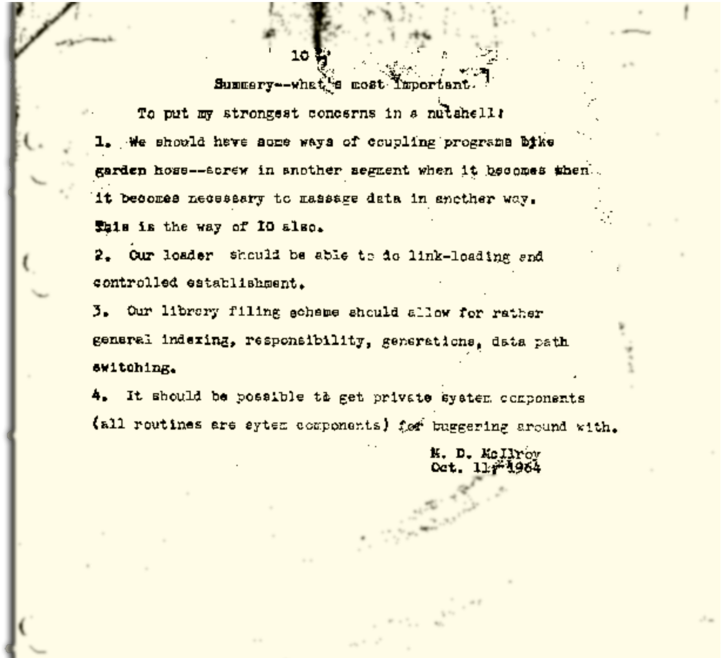
- We should have some ways of coupling programs like garden hose–screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.
- Our loader should be able to do link-loading and controlled establishment.
- Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.
- It should be possible to get private system components (all routines are system components) for buggering around with.

Requirements

- We should have some ways of coupling programs like garden hose–screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.
- Our loader should be able to do link-loading and controlled establishment.
- Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.
- It should be possible to get private system components (all routines are system components) for buggering around with.

M. D. McIlroy 1968

Unix Pipelines



10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

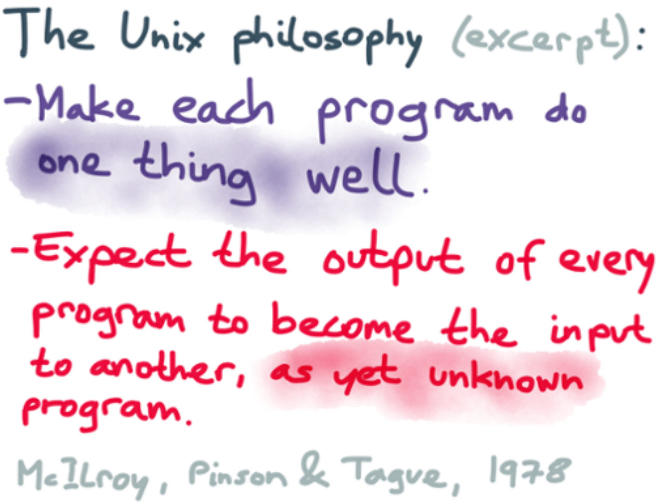
4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Unix pipelines

```
cat file.txt | tr -s '[[:punct:][:space:]]' | sort | uniq -c | sort -rn | head -n 5

cat file.txt
  |   tr -s '[[:punct:][:space:]]'
  |   sort
  |   uniq -c
  |   sort -rn
  |   head -n 5
```

Unix Pipelines



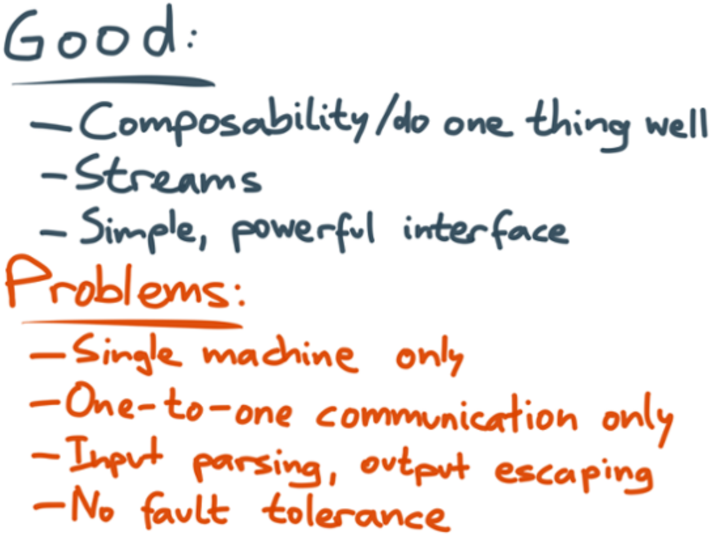
The Unix philosophy (excerpt):

- Make each program do one thing well.
- Expect the output of every program to become the input to another, as yet unknown program.

McIlroy, Pinson & Tague, 1978

Figure: Apache Kafka, Samza, and the Unix Philosophy of Distributed Data: *Martin Kleppmann, Confluent*

Unix Pipelines



Good:

- Composability/do one thing well
- Streams
- Simple, powerful interface

Problems:

- Single machine only
- One-to-one communication only
- Input parsing, output escaping
- No fault tolerance

Figure: Apache Kafka, Samza, and the Unix Philosophy of Distributed Data: *Martin Kleppmann, Confluent*

Unix Pipelines

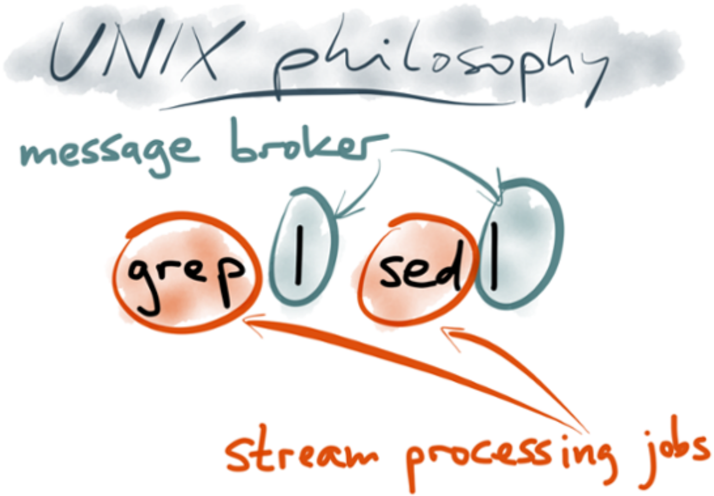
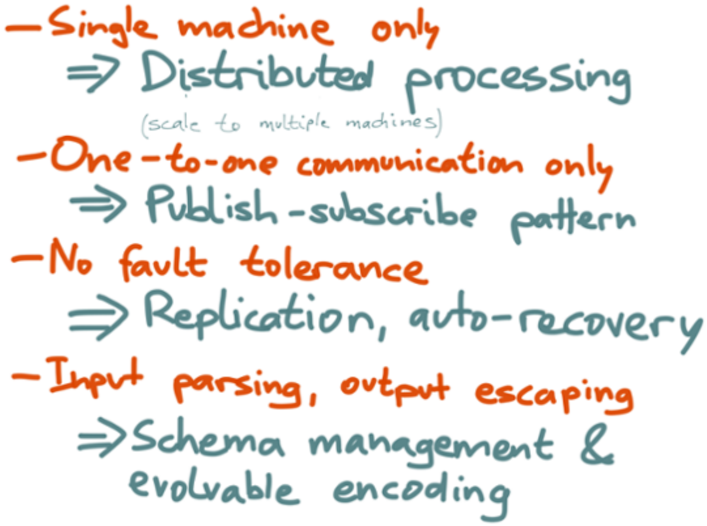


Figure: Apache Kafka, Samza, and the Unix Philosophy of Distributed Data: *Martin Kleppmann, Confluent*

Unix Pipelines



- Single machine only
  ⇒ Distributed processing
  (scale to multiple machines)
- One-to-one communication only
  ⇒ Publish-subscribe pattern
- No fault tolerance
  ⇒ Replication, auto-recovery
- Input parsing, output escaping
  ⇒ Schema management & evolvable encoding

Figure: Apache Kafka, Samza, and the Unix Philosophy of Distributed Data: *Martin Kleppmann, Confluent*

Real Time Processing

Jay Kreps, LinkedIn

The Log: What every software engineer should know about real-time data's unifying abstraction

`https://engineering.linkedin.com/distributed-systems/`

`log-what-every-software-engineer-should-know-about-real-time-datas-unifying`

The Log

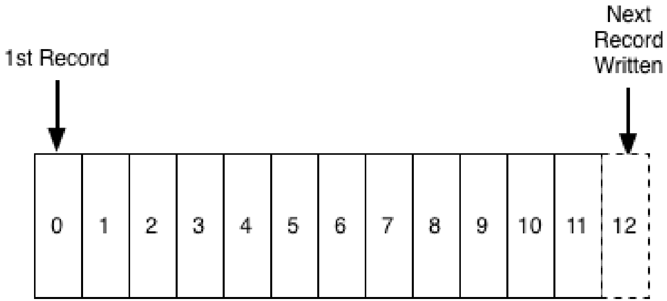


Figure: Jay Kreps, LinkedIn

The Log

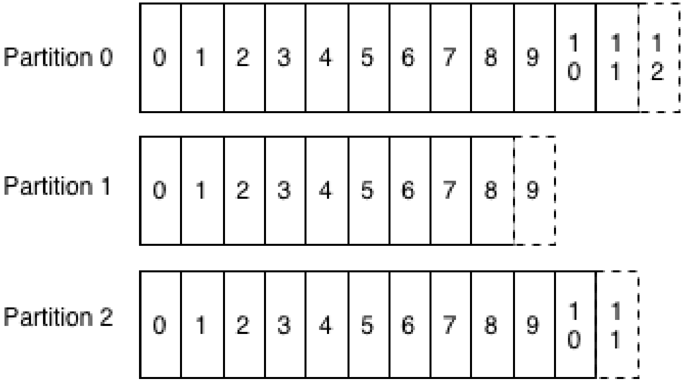


Figure: Jay Kreps, LinkedIn

The Log

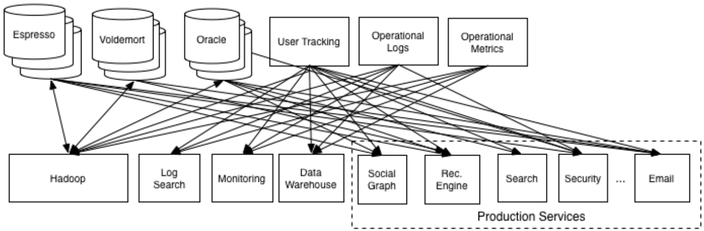


Figure: Jay Kreps, LinkedIn

The Log

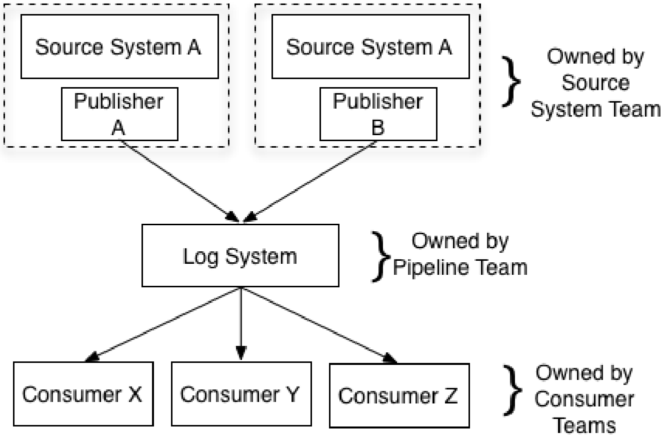
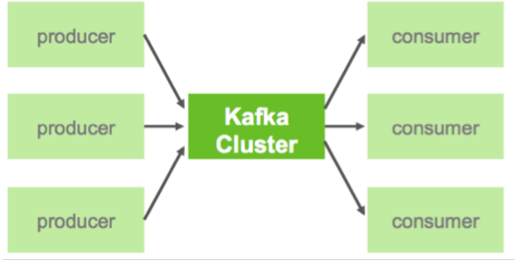


Figure: Jay Kreps, LinkedIn

Apache Kafka

Apache Kafka from LinkedIn



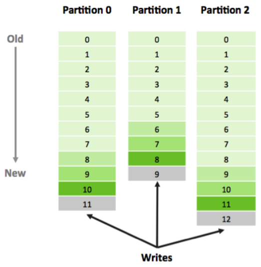
Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system.

Apache Kafka from LinkedIn

Components of Apache Kafka

- **topics:** categories that Kafka uses to maintains feeds of messages
- **producers:** processes that publish messages to a Kafka topic
- **consumers:** processes that subscribe to topics and process the feed of published messages
- **broker:** server that is part of the cluster that runs Kafka

Apache Kafka from LinkedIn



- The Kafka cluster maintains a partitioned log.
- 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.

Apache Kafka from LinkedIn

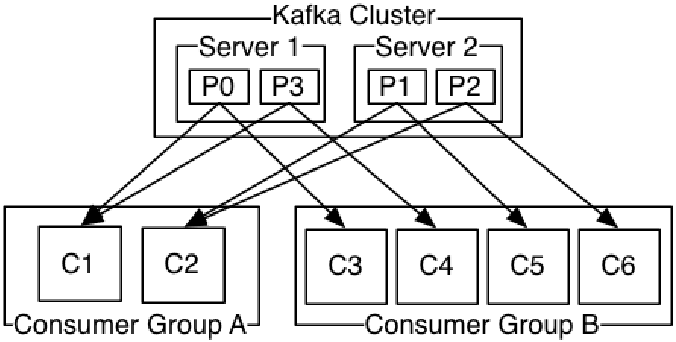


Figure: A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups.

Apache Kafka from LinkedIn

Guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, Kafka tolerates up to N-1 server failures without losing any messages committed to the log.

Kafka API

```
class kafka.javaapi.consumer.SimpleConsumer {
  /**
   *  Fetch a set of messages from a topic.
   *
   *  @param request specifies the topic name, topic partition, starting byte offset, maximum bytes to be fetche
   *  @return a set of fetched messages
   */
  public FetchResponse fetch(kafka.javaapi.FetchRequest request);

  /**
   *  Fetch metadata for a sequence of topics.
   *
   *  @param request specifies the versionId, clientId, sequence of topics.
   *  @return metadata for each topic in the request.
   */
  public kafka.javaapi.TopicMetadataResponse send(kafka.javaapi.TopicMetadataRequest request);

  /**
   *  Get a list of valid offsets (up to maxSize) before the given time.
   *
   *  @param request a [[kafka.javaapi.OffsetRequest]] object.
   *  @return a [[kafka.javaapi.OffsetResponse]] object.
   */
  public kafak.javaapi.OffsetResponse getOffsetsBefore(OffsetRequest request);

  /**
   * Close the SimpleConsumer.
   */
  public void close();
}
```

Apache Samza

Samza

Samza is a stream processing framework with the following features:

- **Simple API**: it 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.
- **Fault tolerance**: Whenever a machine 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.

Samza

samza

Samza is a stream processing framework with the following features:

- **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**: Samza provides a pluggable API that lets you run Samza with other messaging systems and execution environments.
- **Processor isolation**: Samza works with Apache YARN

Apache Samza from LinkedIn

Storm and Samza are fairly similar. Both systems provide:

1. a partitioned stream model,
2. a distributed execution environment,
3. an API for stream processing,
4. fault tolerance,
5. Kafka integration

Samza

Samza components:

- **Streams**: A stream is composed of immutable messages of a similar type or category
- **Jobs**: code that performs a logical transformation on a set of input streams to append output messages to set of output streams

Samza parallel Components:

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

Samza

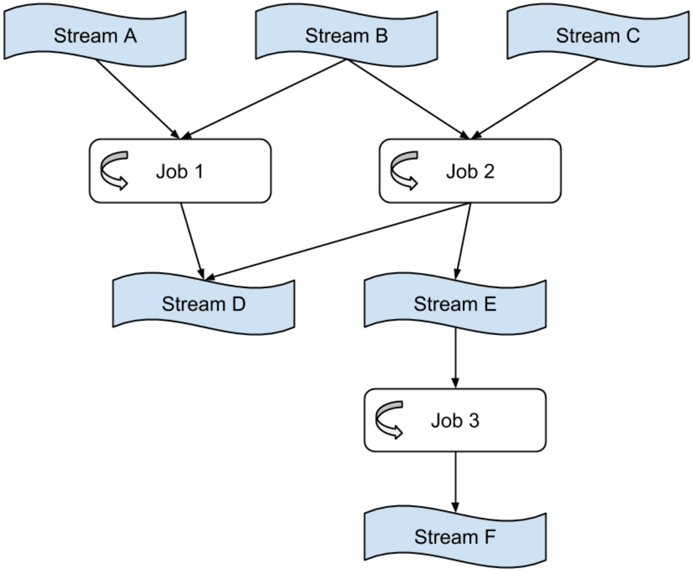
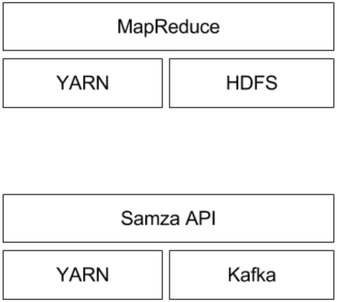


Figure: Dataflow Graphs

Samza and Yarn



Samza

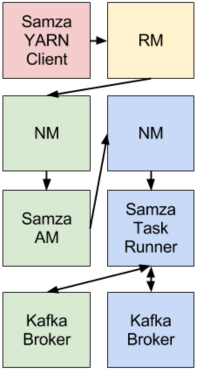


Figure: Samza, Yarn and Kafka integration

Samza API

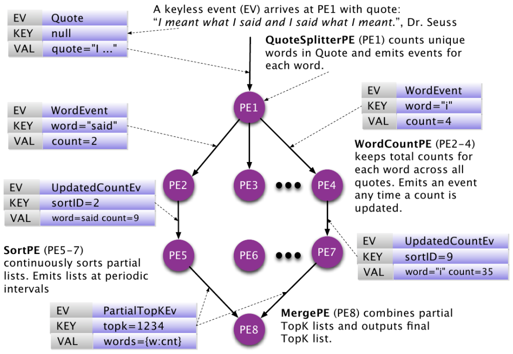
```
package com.example.samza;

public class MyTaskClass implements StreamTask {

  public void process(IncomingMessageEnvelope envelope,
                      MessageCollector collector,
                      TaskCoordinator coordinator) {
    // process message
  }
}
```

Samza API

```
# Job
job.factory.class=org.apache.samza.job.local.ThreadJobFactory
job.name=hello-world

# Task
task.class=samza.task.example.StreamTask
task.inputs=example-system.example-stream

# Serializers
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory
serializers.registry.string.class=org.apache.samza.serializers.StringSerdeFactory

# Systems
systems.example-system.samza.factory=samza.stream.example.ExampleConsumerFactory
systems.example-system.samza.key.serde=string
systems.example-system.samza.msg.serde=json
```

Apache Storm

Apache S4 from Yahoo

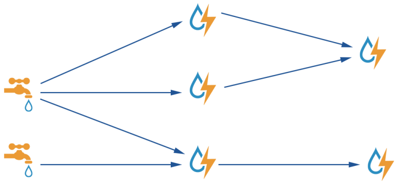


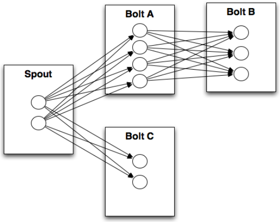
EV Quote
KEY null
VAL quote="I ..."
A keyless event (EV) arrives at PE1 with quote:
"*I meant what I said and I said what I meant.*", Dr. Seuss
QuoteSplitterPE (PE1) counts unique words in Quote and emits events for each word.
EV WordEvent
KEY word="i"
VAL count=4
EV WordEvent
KEY word="said"
VAL count=2
PE1
PE2
PE3
PE4
WordCountPE (PE2-4) keeps total counts for each word across all quotes. Emits an event any time a count is updated.
EV UpdatedCountEv
KEY sortID=2
VAL word=said count=9
PE5
PE6
PE7
EV UpdatedCountEv
KEY sortID=9
VAL word="i" count=35
SortPE (PE5-7) continuously sorts partial lists. Emits lists at periodic intervals
EV PartialTopKEv
KEY topk=1234
VAL words={w:cnt}
PE8
MergePE (PE8) combines partial TopK lists and outputs final TopK list.

PE ID	PE Name	Key Tuple
PE1	QuoteSplitterPE	null
PE2	WordCountPE	word="said"
PE4	WordCountPE	word="i"
PE5	SortPE	sortID=2
PE7	SortPE	sortID=9
PE8	MergePE	topK=1234

Not longer an active project.

Apache Storm





Stream, Spout, Bolt, Topology

Storm

Storm cluster nodes:

- **Nimbus node** (master node, similar to the Hadoop JobTracker):
  - Uploads computations for execution
  - Distributes code across the cluster
  - Launches workers across the cluster
  - Monitors computation and reallocates workers as needed
- **ZooKeeper nodes**: coordinates the Storm cluster
- **Supervisor nodes**: communicates with Nimbus through Zookeeper, starts and stops workers according to signals from Nimbus

Storm

Storm Abstractions:

- **Tuples**: an ordered list of elements.
- **Streams**: an unbounded sequence of tuples.
- **Spouts**: sources of streams in a computation
- **Bolts**: process input streams and produce output streams. They can: run functions; filter, aggregate, or join data; or talk to databases.
- **Topologies**: the overall calculation, represented visually as a network of spouts and bolts

Storm

APACHE STORM™
Distributed • Resilient • Real-time

Main Storm Groupings:

- **Shuffle grouping**: Tuples are randomly distributed but each bolt is guaranteed to get an equal number of tuples.
- **Fields grouping**: The stream is partitioned by the fields specified in the grouping.
- **Partial Key grouping**: The stream is partitioned by the fields specified in the grouping, but are load balanced between two downstream bolts.
- **All grouping**: The stream is replicated across all the bolt’s tasks.
- **Global grouping**: The entire stream goes to the task with the lowest id.

Storm API

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);

builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

Config conf = new Config();
StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
```

Storm API

```java
public static class SplitSentence extends ShellBolt implements IRichBolt {

    public SplitSentence() {
      super("python", "splitsentence.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }
```

Storm API

```
public static class WordCount extends BaseBasicBolt {
  Map<String, Integer> counts = new HashMap<String, Integer>();

  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.get(word);
    if (count == null)
      count = 0;
    count++;
    counts.put(word, count);
    collector.emit(new Values(word, count));
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word", "count"));
  }
}
```

Apache Storm

APACHE STORM™

Distributed • Resilient • Real-time

Storm characteristics for real-time data processing workloads:

1. Fast
2. Scalable
3. Fault-tolerant
4. Reliable
5. Easy to operate

Twitter Heron

Twitter Heron

Heron includes these features:

1. Off the shelf scheduler
2. Handling spikes and congestion
3. Easy debugging
4. Compatibility with Storm
5. Scalability and latency

Twitter Heron

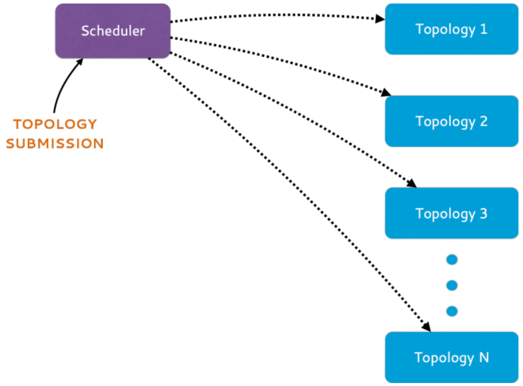


Figure: Heron Architecture

Twitter Heron

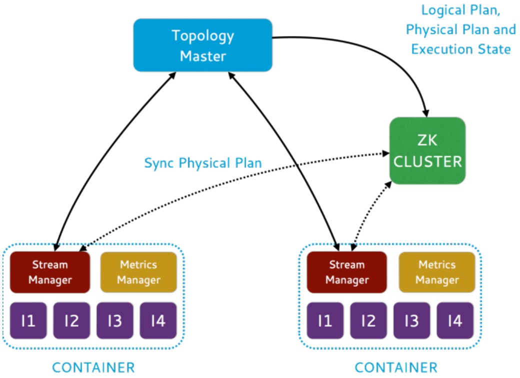


Figure: Topology Architecture

Twitter Heron

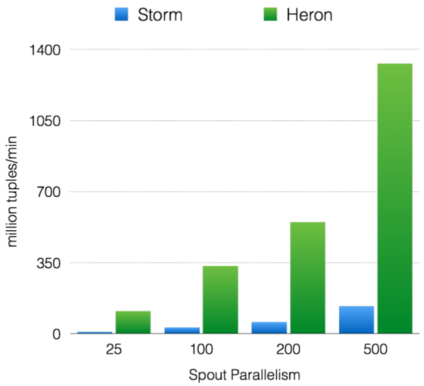


Figure: Throughput with acks enabled

Twitter Heron

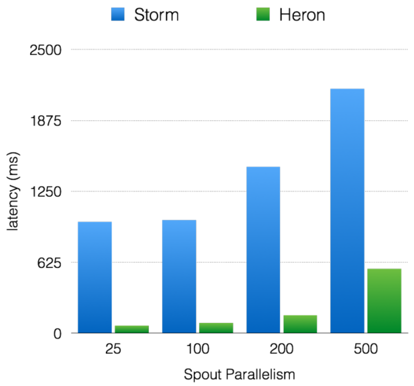


Figure: Latency with acks enabled

Twitter Heron

Twitter Heron Highlights:

1. Able to re-use the code written using Storm
2. Efficient in terms of resource usage
3. 3x reduction in hardware
4. Not open-source