Architectures for Distributed Mining of Big Data

Albert Bifet (@abifet)

MAESTRA Summer School, 6 September 2016

albert.bifet@telecom-paristech.fr

Big Data

BIG DATA are data sets so large or complex that traditional data processing applications can not deal with.

BIG DATA is an OPEN SOURCE Software Revolution.

Big Data

BIG DATA are data sets so large or complex that traditional data processing applications can not deal with.

BIG DATA is an OPEN SOURCE Software Revolution.

Big Data 6V's

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

Controversy of Big Data

- All data is BIG now
- Hype to sell Hadoop based systems
- Ethical concerns about accessibility
- Limited access to Big Data creates new digital divides
- Statistical Significance:
  - When the number of variables grow, the number of fake correlations also grow Leinweber: S&P 500 stock index correlated with butter production in Bangladesh

Batch and Streaming Engines

Batch only

Streaming only

Hybrid

Figure: Batch, streaming and hybrid data processing engines.

Motivation MapReduce

How Many Servers Does Google Have?

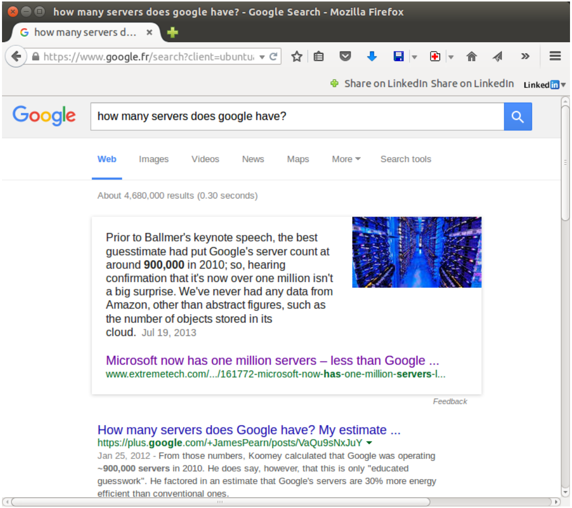


Figure: Asking Google

Typical Big Data Challenges

- How do we break up a large problem into smaller tasks that can be executed in parallel?
- How do we assign tasks to workers distributed across a potentially large number of machines?
- How do we ensure that the workers get the data they need?
- How do we coordinate synchronization among the different workers?
- How do we share partial results from one worker that is needed by another?
- How do we accomplish all of the above in the face of software errors and hardware faults?

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner.

Jeff Dean

MapReduce, BigTable, Spanner

MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
OSDI'04: Sixth Symposium on Operating System Design and Implementation

Jeff Dean Facts

Google Culture Facts

"When Jeff Dean designs software, he first codes the binary and then writes the source as documentation."
"Jeff Dean compiles and runs his code before submitting, but only to check for compiler and CPU bugs."

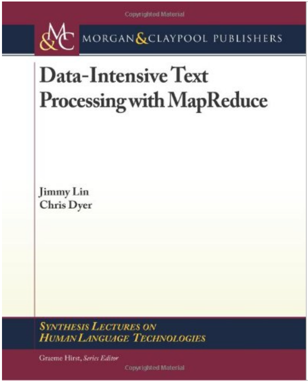
Jeff Dean Facts

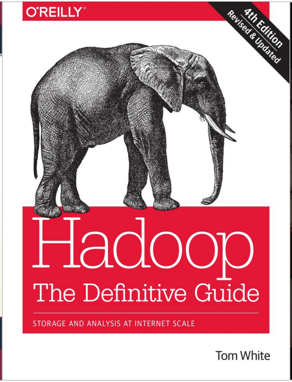
Google Culture Facts

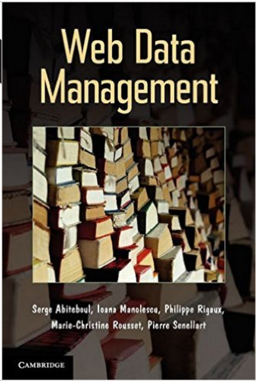
“The rate at which Jeff Dean produces code jumped by a factor of 40 in late 2000 when he upgraded his keyboard to USB2.0.”
”The speed of light in a vacuum used to be about 35 mph. Then Jeff Dean spent a weekend optimizing physics.”

MapReduce

References


Copyrighted Material
MORGAN & CLAYPOOL PUBLISHERS
Data-Intensive Text Processing with MapReduce
Jimmy Lin
Chris Dyer
SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES
Graeme Hirst, Series Editor
Copyrighted Material


O'REILLY
4th Edition
Revised & Updated
Hadoop
The Definitive Guide
STORAGE AND ANALYSIS AT INTERNET SCALE
Tom White


Web Data Management
Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart
CAMBRIDGE

Numbers Everyone Should Know (Jeff Dean)

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA to Netherlands to CA	150,000,000 ns

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each –MAP–
- Shuffle and sort intermediate results
- Aggregate intermediate results –REDUCE–
- Generate final output

Functional Programming

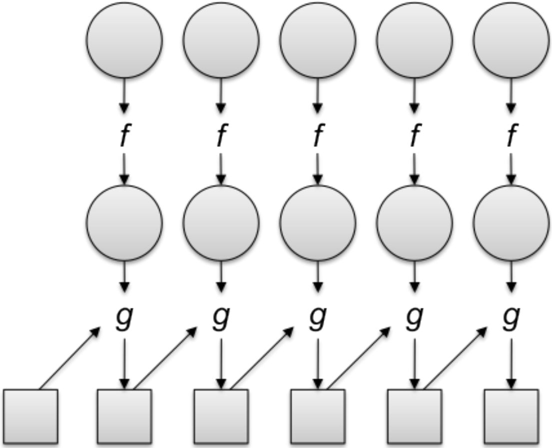


Figure: Map as a transformation function and Fold as an aggregation function

Map and Reduce functions

- In MapReduce, the programmer defines the program logic as two functions:
  - map: $(k_1, v_1) \rightarrow list[(k_2, v_2)]$
    - Map transforms the input into key-value pairs to process
  - reduce: $(k_2, list[v_2]) \rightarrow list[(k_3, v_3)]$
    - Reduce aggregates the list of values for each key
- The MapReduce environment takes in charge distribution aspects.
- A complex program can be decomposed as a succession of Map and Reduce tasks

Simplified view of MapReduce

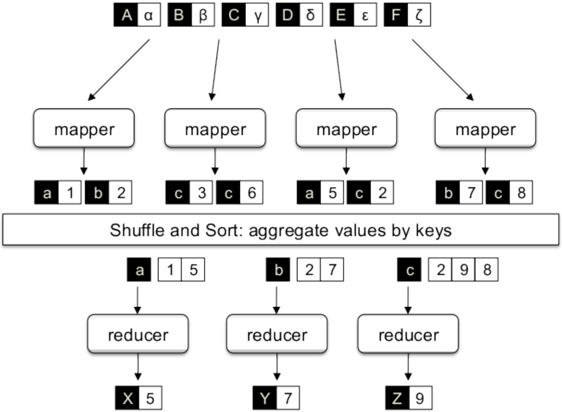


Figure: Two-stage processing structure

An Example Application: Word Count

Input Data

```
foo.txt:  Sweet, this is the foo file
bar.txt:  This is the bar file
```

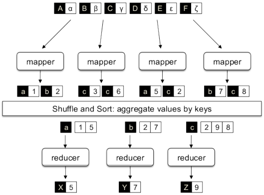
Output Data

```
sweet 1
this 2
is 2
the 2
foo 1
bar 1
file 2
```

WordCount Example

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
        end for
    end method
end class
```

```
class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        end for
        Emit(term t, count sum)
    end method
end class
```



Simple MapReduce Variations

No Reducers

Simple MapReduce Variations

No Reducers

Each mapper output is directly written to a file disk

Simple MapReduce Variations

No Reducers

Each mapper output is directly written to a file disk

No Mappers

Simple MapReduce Variations

No Reducers

Each mapper output is directly written to a file disk

No Mappers

Not possible!

Identity Function Mappers

Sorting and regrouping the input data

Simple MapReduce Variations

No Reducers

Each mapper output is directly written to a file disk

No Mappers

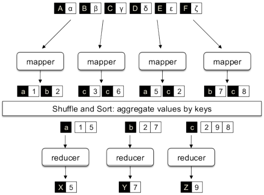
Not possible!

Identity Function Mappers

Sorting and regrouping the input data

Identity Function Reducers

Sorting and regrouping the data from mappers



MapReduce Framework

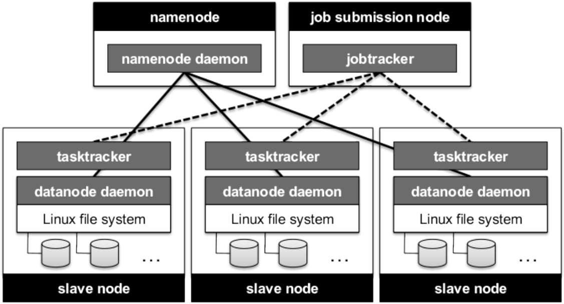


Figure: Runtime Framework

MapReduce Framework

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed filesystem

Fault Tolerance

The Master periodically checks the availability and reachability of the tasktrackers (heartbeats) and whether map or reduce jobs make any progress

- if a mapper fails, its task is reassigned to another tasktracker
- if a reducer fails, its task is reassigned to another tasktracker; this usually require restarting mapper tasks as well (to produce intermediate groups)
- if the jobtracker fails, the whole job should be re-initiated

Speculative execution: schedule redundant copies of the remaining tasks across several nodes

Complete MapReduce Framework

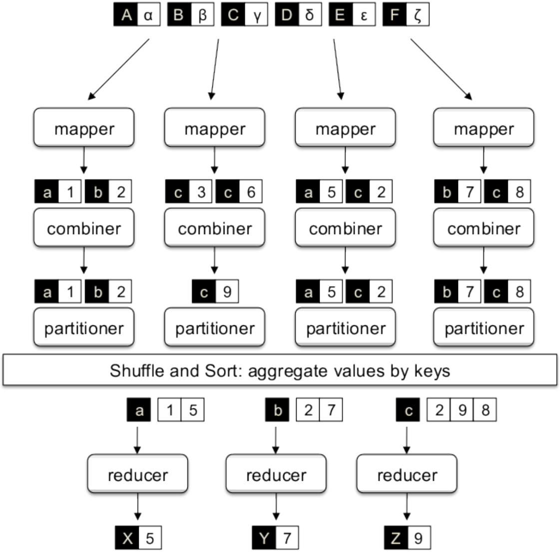


Figure: Partitioners and Combiners

Partitioners and Combiners

Partitioners

Divide up the intermediate key space and assign intermediate key-value pairs to reducers: "simple hash of the key"

partition: (k, number of partitions) $\rightarrow$ *partition for k*

Combiners

Optimization in MapReduce that allow for local aggregation before the shuffle and sort phase: "mini-reducers"

combine: $(k_2, list[v_2]) \rightarrow list[(k_3, v_3)]$

Run in memory, and their goal is to reduce network traffic.

MapReduce Algorithms

Simple MapReduce Algorithms

Distributed Grep

- Grep: reports matching lines on input files
  - Split all files across the nodes
  - Map: emits a line if it matches the specified pattern
  - Reduce: identity function

Count of URL Access Frequency

- Processing logs of web access
  - Map: outputs `<URL,1>`
  - Reduce: Adds together and outputs `<URL, Total Count>`

Simple MapReduce Algorithms

Reverse Web-Link Graph

- Computes `source` list of web pages linked to `target` URLs
  - Map: outputs `<target,source>`
  - Reduce: Concatenates together and outputs `<target, list(source)>`

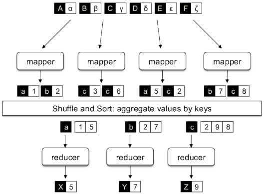
Inverted Index

- Build an inverted index
  - Map: emits a sequence of `<word, docID>`
  - Reduce: outputs `<word, list(docID)>`

WordCount Example Revisited

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
        end for
    end method
end class
```

```
class Reducer
    method Reduce(term t, counts [c1, c2, . . .])
        sum ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            sum ← sum + c
        end for
        Emit(term t, count sum)
    end method
end class
```



WordCount Example Revisited

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
        end for
    end method
end class
```

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1        ▷ Tally counts for entire document
        end for
        for all term t ∈ H do
            Emit(term t, count H{t})
        end for
    end method
end class
```

WordCount Example Revisited

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    end method
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1            ▷ Tally counts across documents
        end for
    end method
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
        end for
    end method
end class
```

Word count mapper using the“in-mapper combining”.

Average Computing Example

Example

Given a large number of key-values pairs, where

- keys are strings
- values are integers

find all average of values by key

Example

- Input: <‘‘a’’,1>, <‘‘b’’,2>, <‘‘c’’,10>, <‘‘b’’,4>, <‘‘a’’,7>
- Output: <‘‘a’’,4>, <‘‘b’’,3>, <‘‘c’’,10>

Average Computing Example

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
    end method
end class
```

```
class Reducer
    method Reduce(string t, integers [r_1, r_2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r_1, r_2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        end for
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
    end method
end class
```

Average Computing Example

Example

Given a large number of key-values pairs, where

- keys are strings
- values are integers

find all average of values by key

Average computing is not associative

- average(1,2,3,4,5) $\neq$ average(average(1,2), average(3,4,5))
- 3 $\neq$ average(1.5, 4) = 2.75

Monoidify!

Monoids as a Design Principle for Efficient MapReduce Algorithms (Jimmy Lin)

Given a set S, an operator $\oplus$ and an identity element e, for all a, b,c in S:

- Closure: $a \oplus b$ is also in S.
- Associativity: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- Identity: $e \oplus a = a \oplus e = e$

Average Computing Example

```
class Mapper
    method Initialize
        S ← new AssociativeArray
        C ← new AssociativeArray
    end method
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    end method
    method Close
        for all term t ∈ S do
            Emit(term t, pair (S{t}, C{t}))
        end for
    end method
end class
```

MapReduce Big Data Processing

A given application may have:

- A chain of map functions
  - (input processing, filtering, extraction. . .)
- A sequence of several map-reduce jobs
- No reduce task when everything can be expressed in the map (zero reducers, or the identity reducer function)

Prefer:

- Simple map and reduce functions
- Mapper tasks processing large data chunks (at least the size of distributed filesystem blocks)

Apache Flink Motivation

Apache Flink Motivation

1. Real time computation: streaming computation
2. Fast, as there is not need to write to disk
3. Easy to write code

Real time computation: streaming computation

MapReduce Limitations

Example

How compute in real time (latency less than 1 second):

1. frequent items as Twitter hashtags
2. predictions
3. sentiment analysis

Easy to Write Code

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                            .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

Easy to Write Code

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                        .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

DataStream API (streaming):

```
val lines: DataStream[String] = env.fromSocketStream(...)

lines.flatMap {line => line.split(" ")
                        .map(word => Word(word,1))}
     .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
     .groupBy("word").sum("frequency")
     .print()
```

What is Apache Flink?

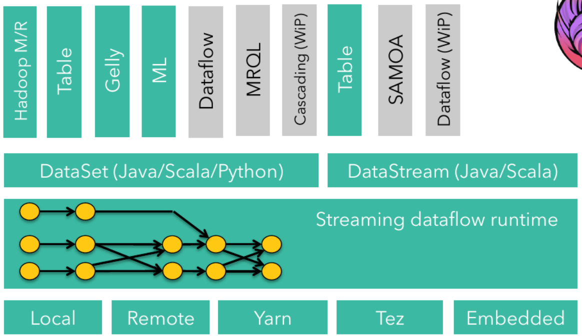


Figure: Apache Flink Overview

Batch and Streaming Engines

Batch only

Streaming only

Hybrid

Figure: Batch, streaming and hybrid data processing engines.

Batch Comparison

API	low-level	high-level	high-level
Data Transfer	batch	batch	pipelined & batch
Memory Management	disk-based	JVM-managed	Active managed
Iterations	file system cached	in-memory cached	streamed
Fault tolerance	task level	task level	job level
Good at	massive scale out	data exploration	heavy backend & iterative jobs
Libraries	many external	built-in & external	evolving built-in & external

Figure: Comparison between Hadoop, Spark And Flink.

Streaming Comparison

Streaming	"true"	mini batches	"true"
API	low-level	high-level	high-level
Fault tolerance	tuple-level ACKs	RDD-based (lineage)	coarse checkpointing
State	not built-in	external	internal
Exactly once	at least once	exactly once	exactly once
Windowing	not built-in	restricted	flexible
Latency	low	medium	low
Throughput	medium	high	high

Figure: Comparison between Storm, Spark And Flink.

Spark Motivation

Apache Spark

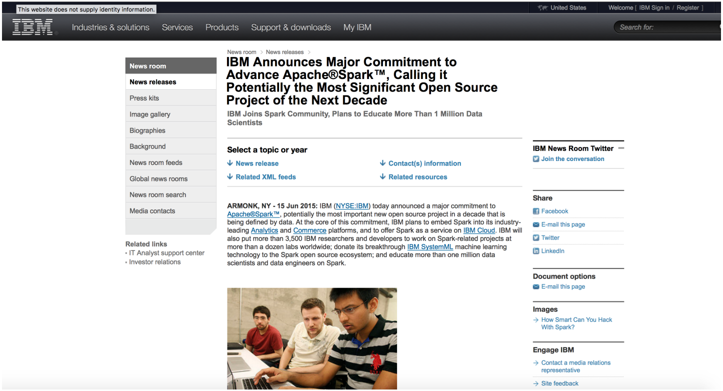


Figure: IBM and Apache Spark

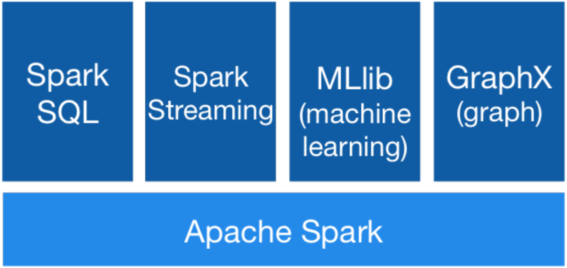
What is Apache Spark

Apache Spark is a fast and general engine for large-scale data processing.

- **Speed**: Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- **Ease of Use**: Write applications quickly in Java, Scala, Python, R.
- **Generality**: Combine SQL, streaming, and complex analytics.
- **Runs Everywhere**: Spark runs on Hadoop, Mesos, standalone, or in the cloud.

`http://spark.apache.org/`

Spark Ecosystem



Spark API

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

```
 val f = sc.textFile(hdfs://...")

val wc = f.flatMap(l => l.split(" "))
          .map(word => (word, 1))
          .reduceByKey(_ + _)
```

Word count in Spark's Scala API

Apache Spark

Apache Spark Project

- Spark started as a research project at UC Berkeley
  - Matei Zaharia created Spark during his PhD
  - Ion Stoica was his advisor
- DataBricks is the Spark start-up, that has raised $46 million

Resilient Distributed Datasets (RDDs)

- An RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- RDDs are created :
  - parallelizing an existing collection in your driver program, or
  - referencing a dataset in an external storage system

Spark API: Parallel Collections

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Spark's Python API

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Spark's Scala API

```
 List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Spark's Java API

Spark API: External Datasets

```
>>> distFile = sc.textFile("data.txt")
```

Spark's Python API

```
scala> val distFile = sc.textFile("data.txt")
distFile: RDD[String] = MappedRDD@1d4cee08
```

Spark's Scala API

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

Spark's Java API

Spark API: RDD Operations

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Spark's Python API

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Spark's Scala API

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

Spark's Java API

Apache Spark Streaming

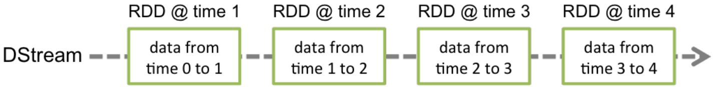
Spark Streaming

Spark Streaming is an extension of Spark that allows processing data stream using micro-batches of data.

Discretized Streams (DStreams)

Spark Streaming

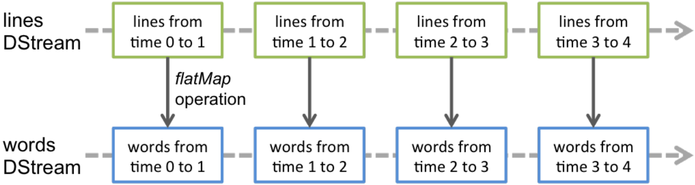
- Discretized Stream or DStream represents a continuous stream of data,
  - either the input data stream received from source, or
  - the processed data stream generated by transforming the input stream.
- Internally, a DStream is represented by a continuous series of RDDs



Discretized Streams (DStreams)

Spark Streaming

- Any operation applied on a DStream translates to operations on the underlying RDDs.



Spark Streaming

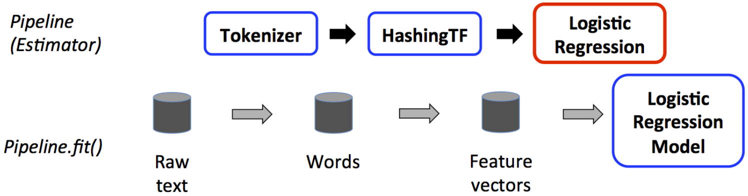
```scala
val conf = new SparkConf().setMaster("local[2]").setAppName("WCount")
val ssc = new StreamingContext(conf, Seconds(1))

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the con
wordCounts.print()

ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

Spark SQL and DataFrames

- Spark SQL is a Spark module for structured data processing.
- It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine.
- A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database .

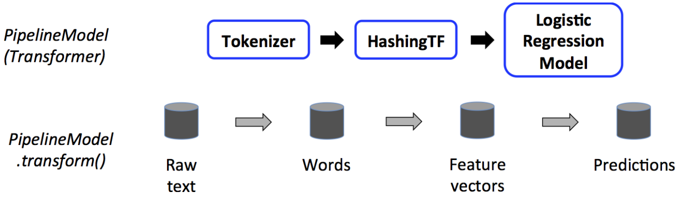
Spark Machine Learning Libraries

- MLLib contains the original API built on top of RDDs.
- spark.ml provides higher-level API built on top of DataFrames for constructing ML pipelines.



Spark Machine Learning Libraries

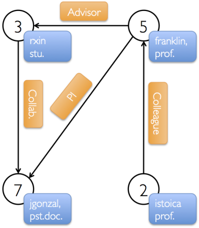
- MLLib contains the original API built on top of RDDs.
- spark.ml provides higher-level API built on top of DataFrames for constructing ML pipelines.



Spark GraphX

- GraphX optimizes the representation of vertex and edge types when they are primitive data types
- The property graph is a directed multigraph with user defined objects attached to each vertex and edge.

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Spark GraphX

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

Apache Spark Summary

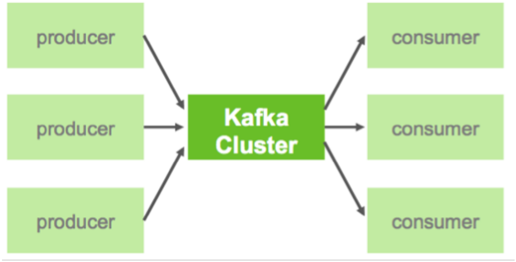
Apache Spark is a fast and general engine for large-scale data processing.

- **Speed**: Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- **Ease of Use**: Write applications quickly in Java, Scala, Python, R.
- **Generality**: Combine SQL, streaming, and complex analytics.
- **Runs Everywhere**: Spark runs on Hadoop, Mesos, standalone, or in the cloud.

`http://spark.apache.org/`

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

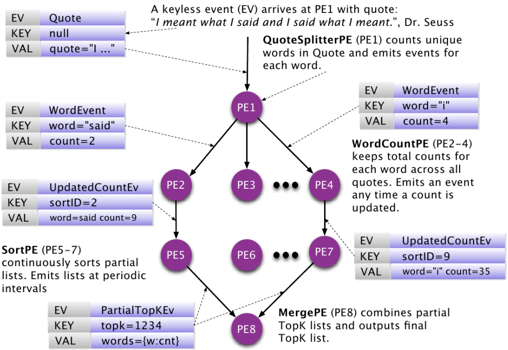
- **topics:** categories that Kafka uses to maintains feeds of messages
- **producers:** processes that publish messages to a Kafka topic
- **consumers:** processes that subscribe to topics and process the feed of published messages
- **broker:** server that is part of the cluster that runs Kafka

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 Storm

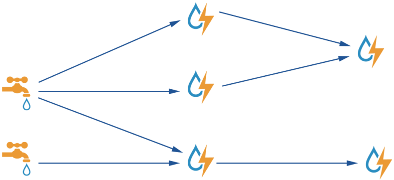
Apache S4 from Yahoo

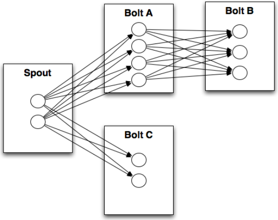


PE ID	PE Name	Key Tuple
PE1	QuoteSplitterPE	null
PE2	WordCountPE	word="said"
PE4	WordCountPE	word="i"
PE5	SortPE	sortID=2
PE7	SortPE	sortID=9
PE8	MergePE	topK=1234

Not longer an active project.

Apache Storm





Stream, Spout, Bolt, Topology

Storm

Storm 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

Google Cloud DataFlow

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner.

Google June 2014

What is using Google right now?

Google June 2014

What is using Google right now?

"We don't really use MapReduce anymore," The company stopped using the system "years ago."

Google June 2014

What is using Google right now?

“We don’t really use MapReduce anymore,” The company stopped using the system “years ago.”

“Cloud Dataflow is the result of over a decade of experience in analytics,” “It will run faster and scale better than pretty much any other system out there.”

Google Cloud Data Flow

The processing model of Google Cloud Dataflow is based upon technology from

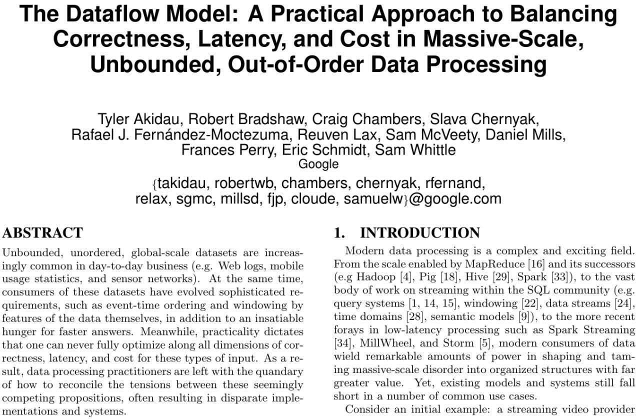
- **FlumeJava**(2010): Java library that makes it easy to develop, test, and run efficient data parallel pipelines.
- **MillWheel**(2013): framework for building low-latency data-processing applications

Google Cloud Data Flow

Cloud Dataflow consists of :

- A set of SDKs that you use to define data processing jobs:
  - **PCollection:** specialized collection class to represent pipeline data.
  - **PTransforms:** powerful data transforms, generic frameworks that apply functions across an entire data set
  - **I/O APIs:** pipeline read and write data to and from a variety of formats and storage technologies.
- A Google Cloud Platform managed service:
  - Google Compute Engine VMs, to provide job workers.
  - Google Cloud Storage, for reading and writing data.
  - Google BigQuery, for reading and writing data.

Google Cloud Data Flow Paper



The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google
{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, millsd, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider

Figure: VLDB 2015

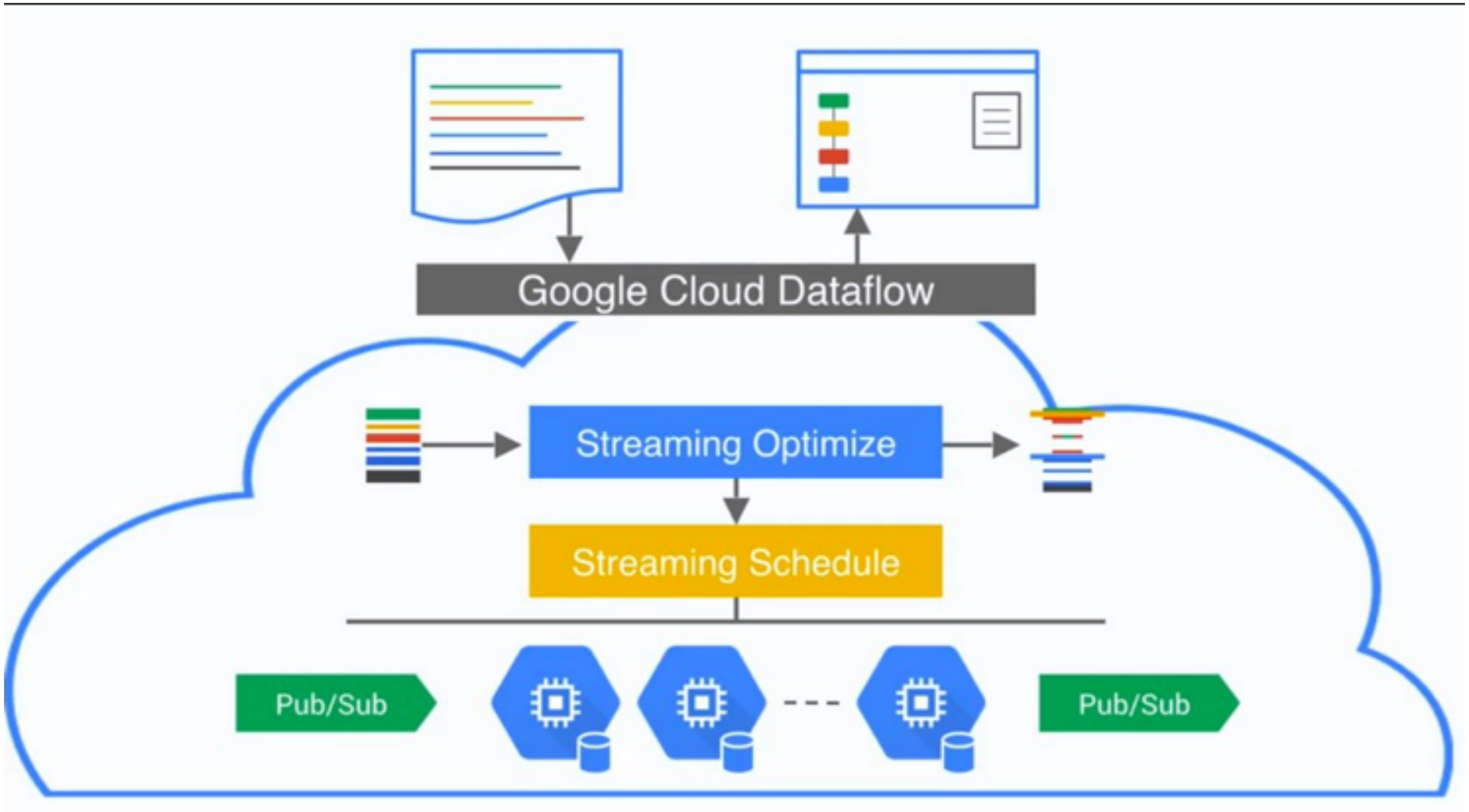
Google Cloud Data Flow

4. CONCLUSIONS

The future of data processing is unbounded data. Though bounded data will always have an important and useful place, it is semantically subsumed by its unbounded counterpart. Furthermore, the proliferation of unbounded data sets across modern business is staggering. At the same time, consumers of processed data grow savvier by the day, demanding powerful constructs like event-time ordering and unaligned windows. The models and systems that exist today serve as an excellent foundation on which to build the data processing tools of tomorrow, but we firmly believe that a shift in overall mindset is necessary to enable those tools to comprehensively address the needs of consumers of unbounded data.

Figure: Conclusions of the VLDB 2015 paper

Apache Beam



Google Cloud Dataflow
Streaming Optimize
Streaming Schedule
Pub/Sub
Pub/Sub

Apache Beam

- Apache Beam code can run in:
  - Apache Flink
  - Apache Spark
  - Google Cloud Dataflow
- Google Cloud Dataflow replaced MapReduce:
  - It is based on FlumeJava and MillWheel, a stream engine as Storm, Samza
  - It writes and reads to Google Pub/Sub, a service similar to Kafka

Architectures

Lambda Architecture

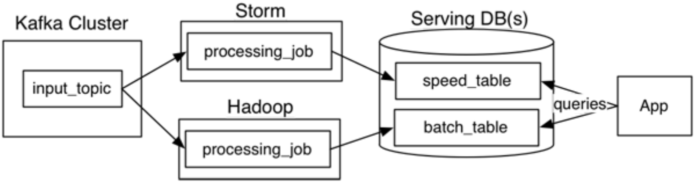


Figure: Nathan Marz

Kappa Architecture

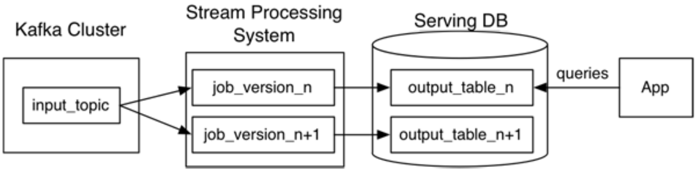
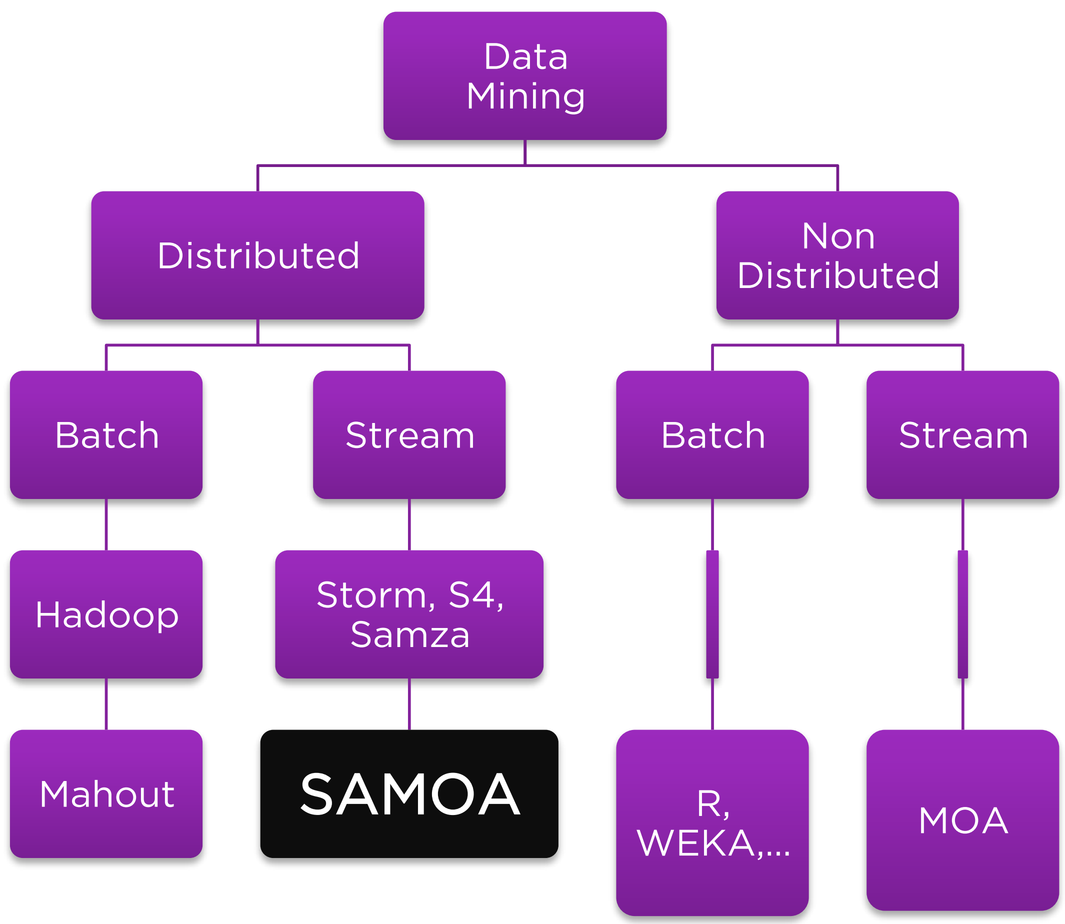
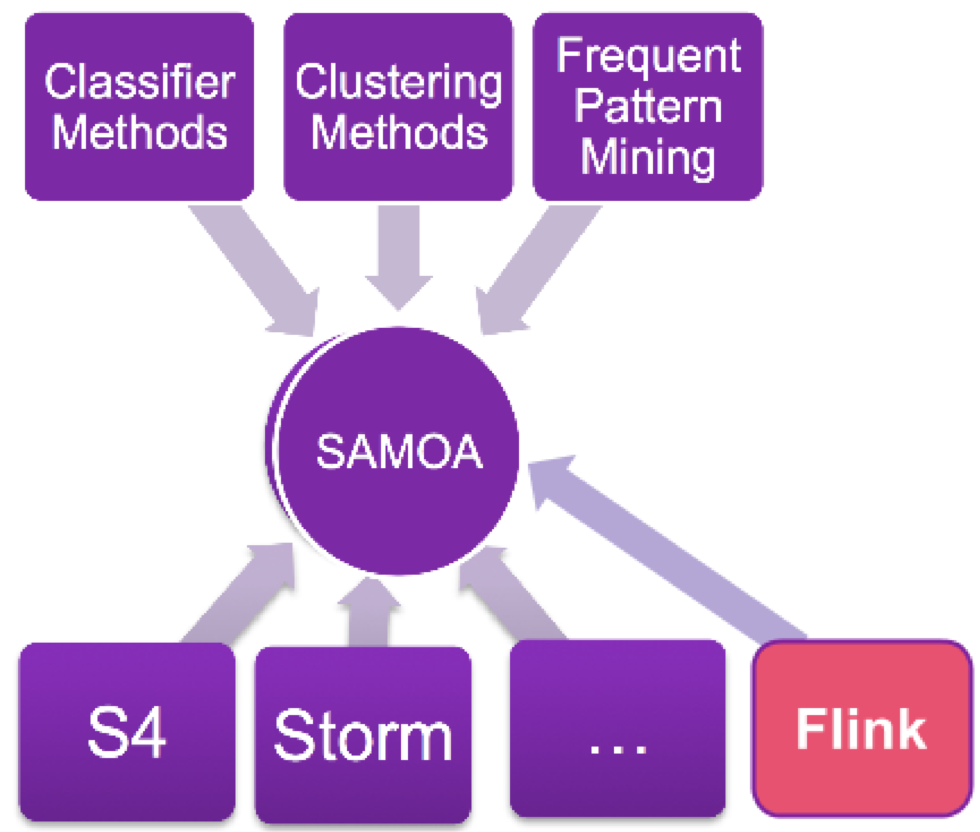


Figure: Questioning the Lambda Architecture by Jay Kreps http://samoa-project.net

SAMOA

G. De Francisci Morales, A. Bifet: “SAMOA: Scalable Advanced Massive Online Analysis”. JMLR (2014)





Thanks!

@abifet