Stream processing everywhere—what to use?

Jim Scott – Director, Enterprise Strategy & Architecture

@kingmesal #StrataHadoop

The Landscape

- Stream Processing is Fundamentally Simple
  - Inputs -> Outputs
  - But it is WAY more complicated than this…
- Optimization can be complicated
- This space is very confused
  - Performance of different options is dependent upon source
- Lots of misinformation
  - e.g. performance comparisons that are not apples-to-apples

Semantics

- There are three general categories of *delivery patterns*:
  - *At-most-once*: messages may be lost. This is usually the least desirable outcome.
  - *At-least-once*: messages may be redelivered (no loss, but duplicates). This is good enough for many use cases.
  - *Exactly-once*: each message is delivered once and only once (no loss, no duplicates). This is a desirable feature although difficult to guarantee in all cases.

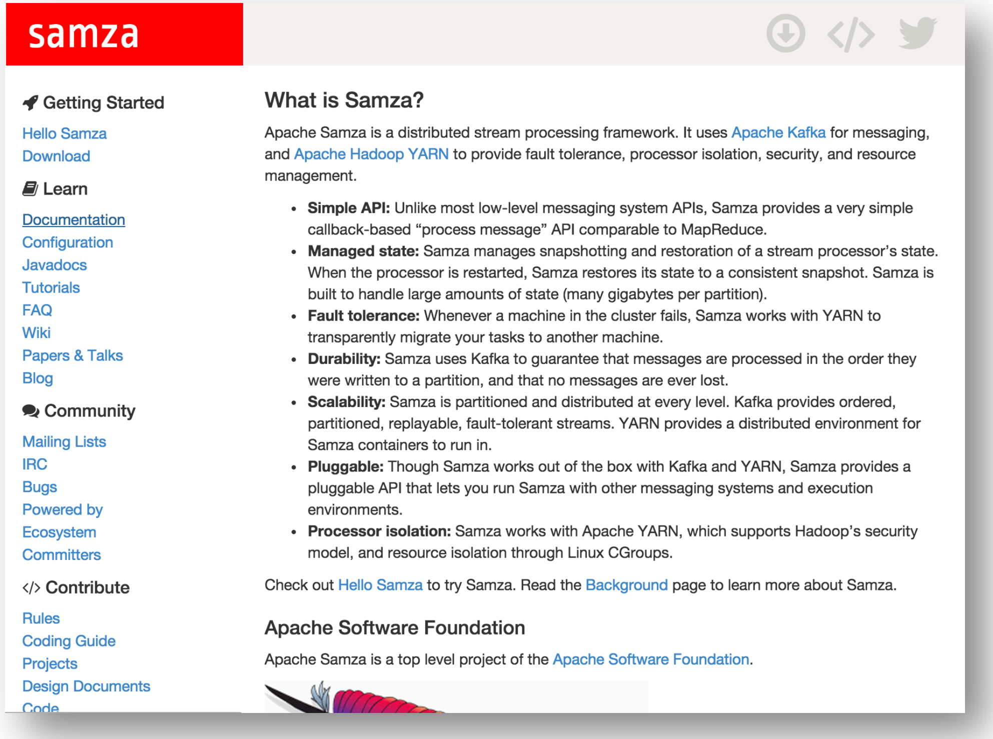
Today's Options – Apache Style

- Samza
- Storm
- Spark Streaming

Apache Samza

Apache Samza

- Originally developed in LinkedIn (Chris Riccomini/Jay Kreps), now ASF top-level project
- Distributed stream processing framework (YARN/Kafka)



samza

Getting Started

Hello Samza
Download

Learn

Documentation
Configuration
Javadocs
Tutorials
FAQ
Wiki
Papers & Talks
Blog

Community

Mailing Lists
IRC
Bugs
Powered by
Ecosystem
Committers

</> Contribute

Rules
Coding Guide
Projects
Design Documents
Code

What is 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.

- **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.

Check out Hello Samza to try Samza. Read the Background page to learn more about Samza.

Apache Software Foundation

Apache Samza is a top level project of the Apache Software Foundation.

samza

http://samza.apache.org/

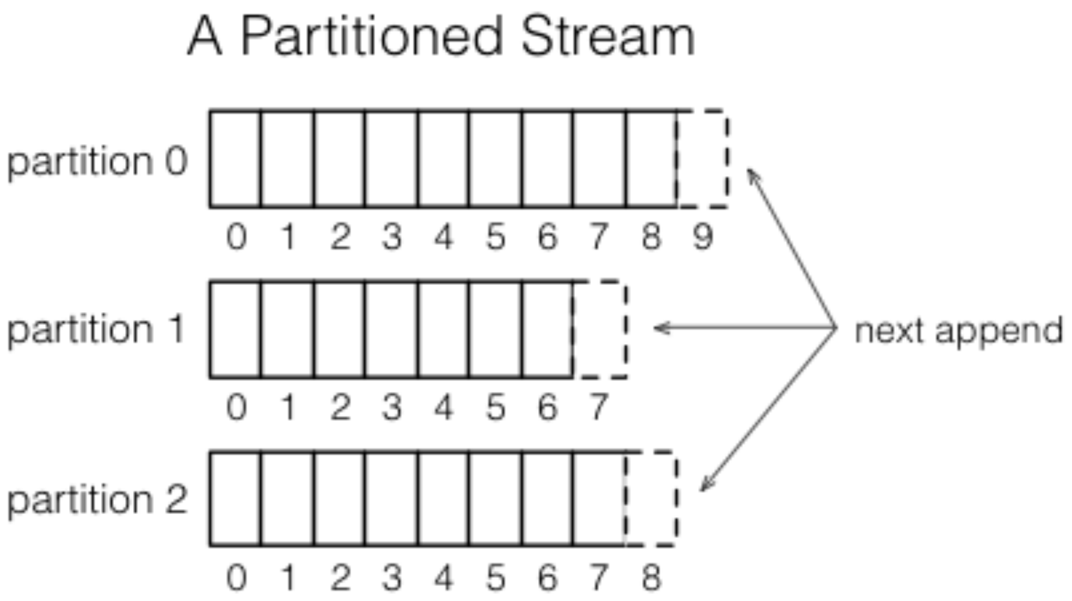
Concepts

- Streams & Partitions
- Jobs & Tasks
- Dataflow Graphs
- Containers

samza.apache.org/learn/documentation/latest/introduction/concepts.html

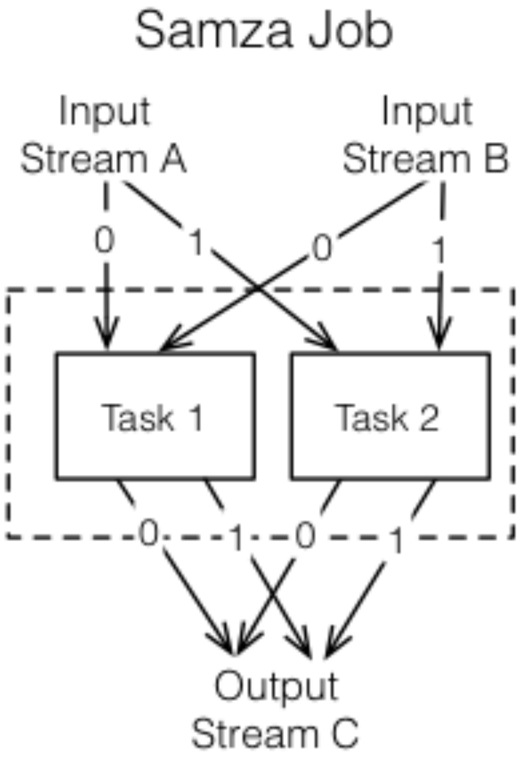
Streams & Partitions

- Stream: immutable messages
- Each stream comprises one or more partitions
- Partition: totally ordered sequence of messages



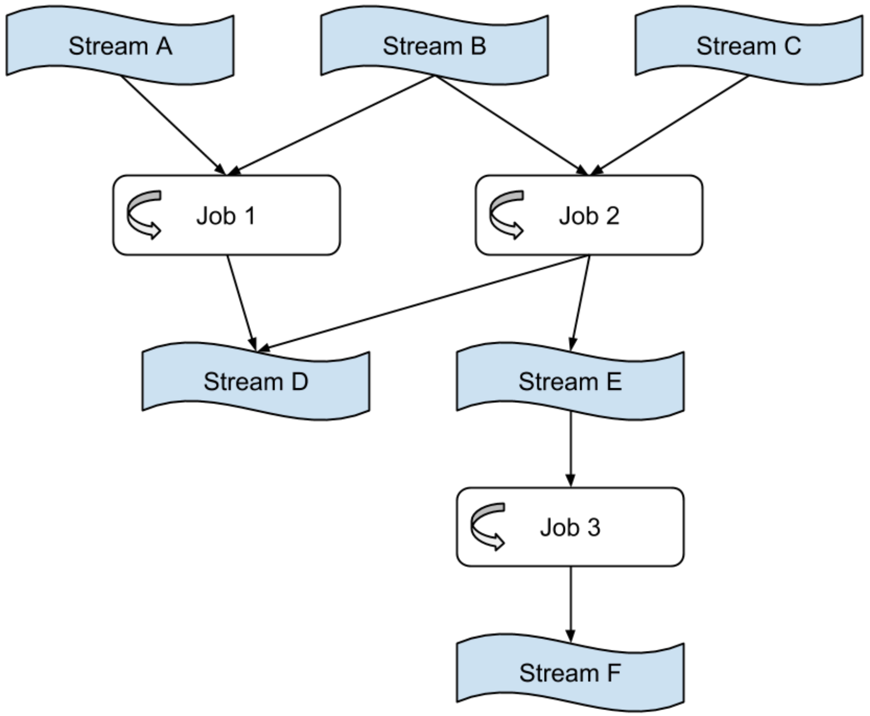
Jobs & Tasks

- Job: logical unit of stream processing, a collection of tasks
- Task: unit of parallelism
  - in the context of a job, each task consumes data from one partition
  - processes messages from each of its input partitions sequentially



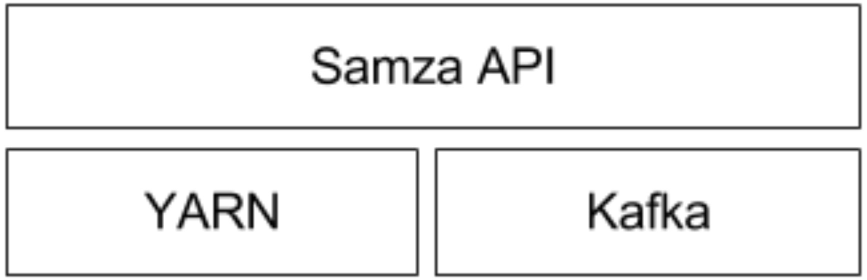
Dataflow Graphs

- Dataflow graph: logical, directed graph of jobs
- Jobs in DG are decoupled
  - can be developed independently
  - don’t impact up/downstream jobs
- DG can contain cycles



Samza Architecture

- Processing layer → Samza API
- Pluggable execution layer (default: YARN)
- Pluggable streaming layer (default: Kafka)



Samza API

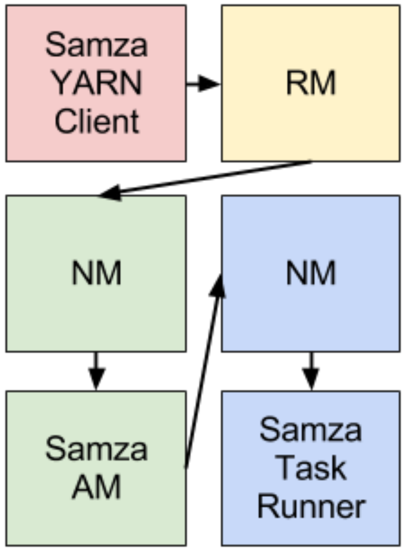
YARN

Kafka

samza.apache.org/learn/documentation/latest/introduction/architecture.html

Samza Execution: Containers

- Partitions and tasks are both logical units of parallelism
- Containers are the unit of physical parallelism, essentially a Unix process (or Linux cgroup)



Samza Resources

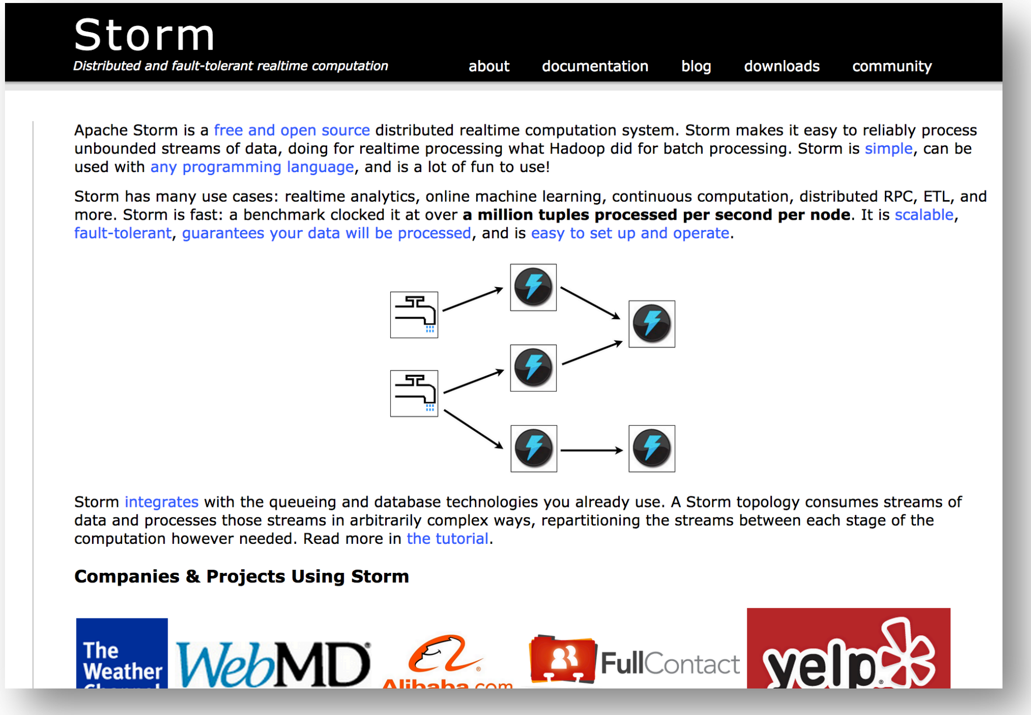
- http://www.jfokus.se/jfokus15/preso/ApacheSamza.pdf
- http://www.berlinbuzzwords.de/session/samza-linkedin-taking-stream-processing-next-level
- http://www.infoq.com/articles/linkedin-samza

Kudos to Chris Riccomini and Martin Kleppmann
for their invaluable support concerning Samza!

Apache Storm

Apache Storm

- Originally developed by Nathan Marz at Backtype/Twitter, now ASF top-level project
- Distributed, fault-tolerant stream-processing platform



Storm

Distributed and fault-tolerant realtime computation

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over **a million tuples processed per second per node**. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Read more in the tutorial.

Companies & Projects Using Storm

STORM

http://storm.apache.org/

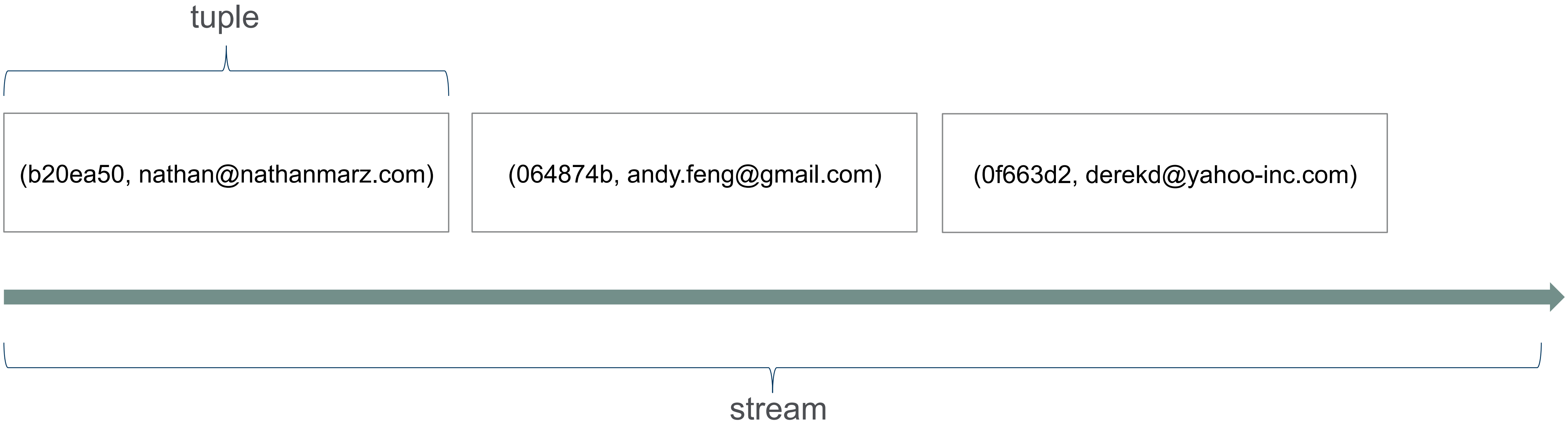
Concepts

- Tuples and Streams
- Spouts, Bolts, Topologies
- Tasks and Workers
- Stream Grouping

storm.apache.org/documentation/Tutorial.html

Tuples and Streams

- Tuple: ordered list of elements
- Stream: unbounded sequence of tuples



tuple

(b20ea50, nathan@nathanmarz.com)

(064874b, andy.feng@gmail.com)

(0f663d2, derekd@yahoo-inc.com)

stream

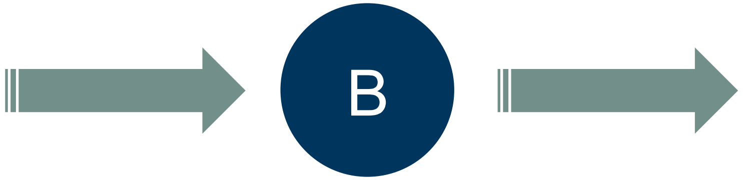
Spouts

- The sources of streams

- Can talk with
  - Queues (Kafka, Kestrel, etc.)
  - Web logs
  - API calls
  - Filesystem (MapR-FS / HDFS)
  - Etc.

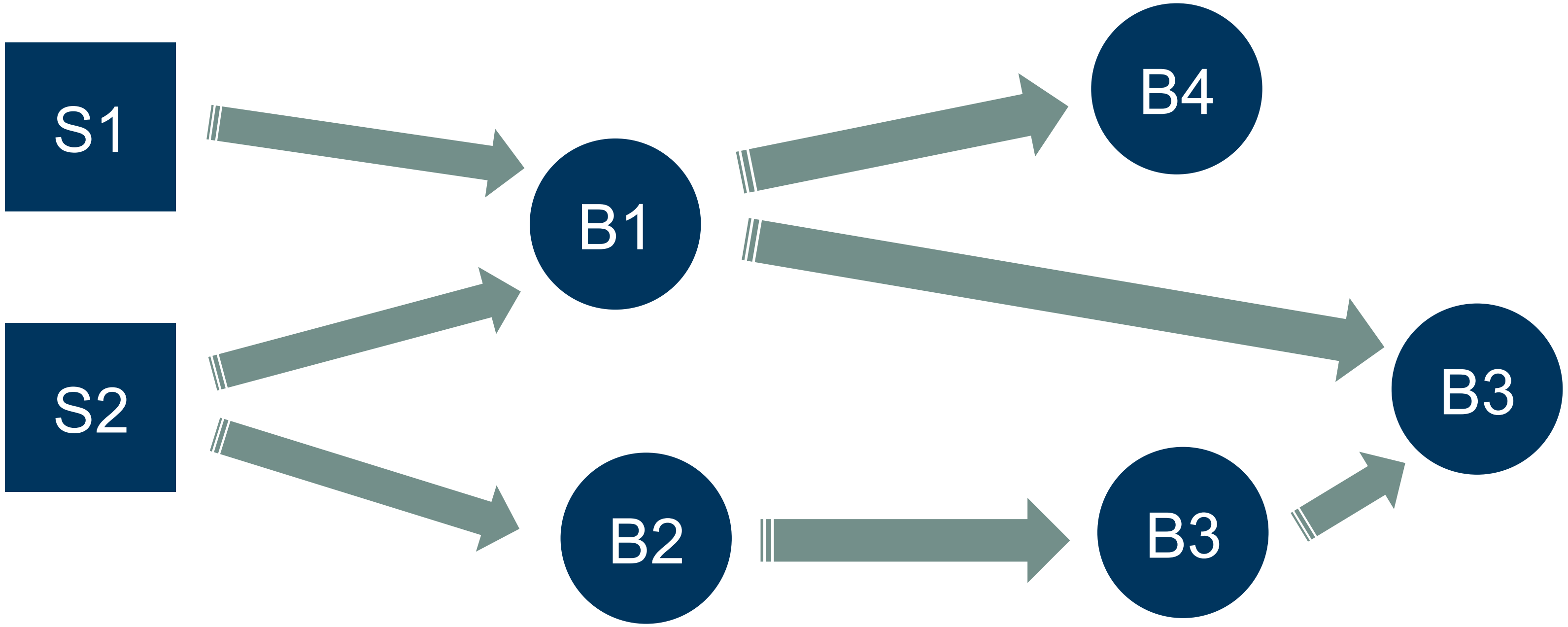
Bolts

- Process tuples and create new streams
- Implement business logic via …
  - Transform
  - Filter
  - Aggregate
  - Join
  - Access datastores & DBs
  - Access APIs (e.g., geo location look-up)



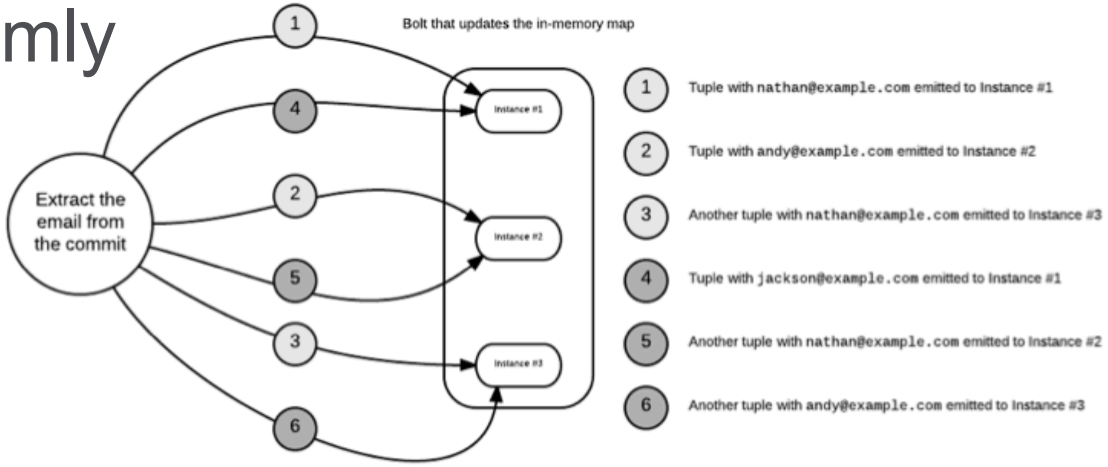
Topologies

- Directed graph of spouts and bolts



Stream Grouping

- Shuffle grouping: tuples are randomly distributed across all of the tasks running the bolt


Bolt that updates the in-memory map
Extract the email from the commit
1
4
2
5
3
6
Instance #1
Instance #2
Instance #3

1 Tuple with nathan@example.com emitted to Instance #1

2 Tuple with andy@example.com emitted to Instance #2

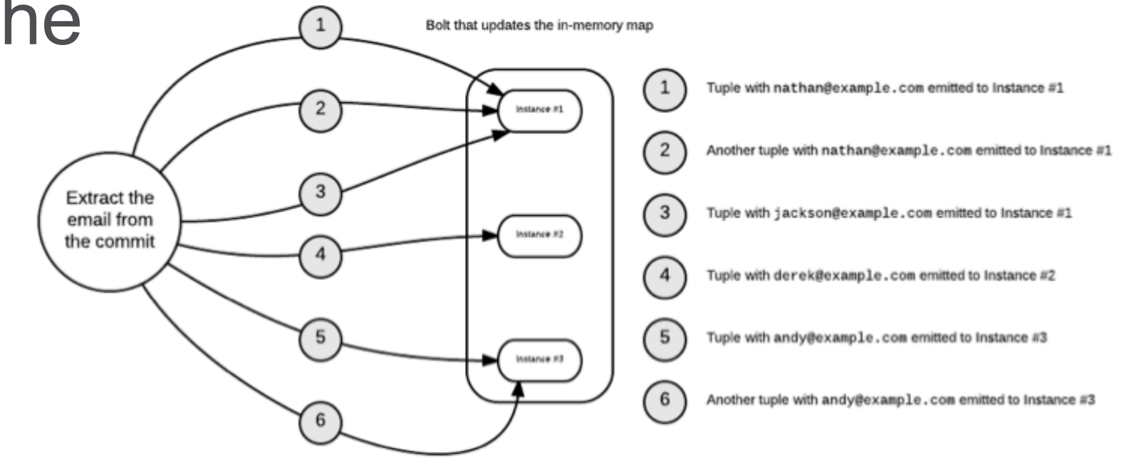
3 Another tuple with nathan@example.com emitted to Instance #3

4 Tuple with jackson@example.com emitted to Instance #1

5 Another tuple with nathan@example.com emitted to Instance #2

6 Another tuple with andy@example.com emitted to Instance #3

- Fields grouping: groups tuples by specific name field and routes to the same task


Bolt that updates the in-memory map
Extract the email from the commit
1
2
3
4
5
6
Instance #1
Instance #2
Instance #3

1 Tuple with nathan@example.com emitted to Instance #1

2 Another tuple with nathan@example.com emitted to Instance #1

3 Tuple with jackson@example.com emitted to Instance #1

4 Tuple with derek@example.com emitted to Instance #2

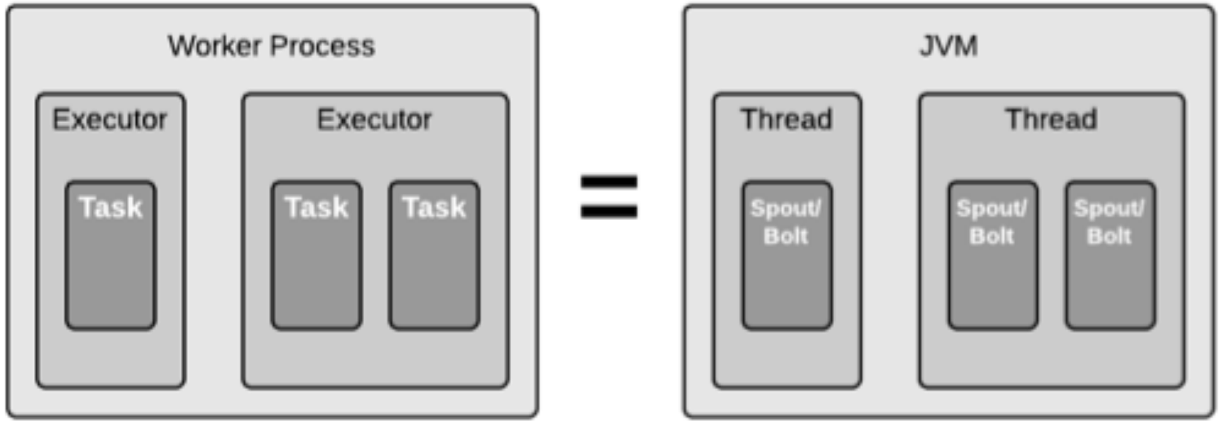
5 Tuple with andy@example.com emitted to Instance #3

6 Another tuple with andy@example.com emitted to Instance #3

Tasks and Workers

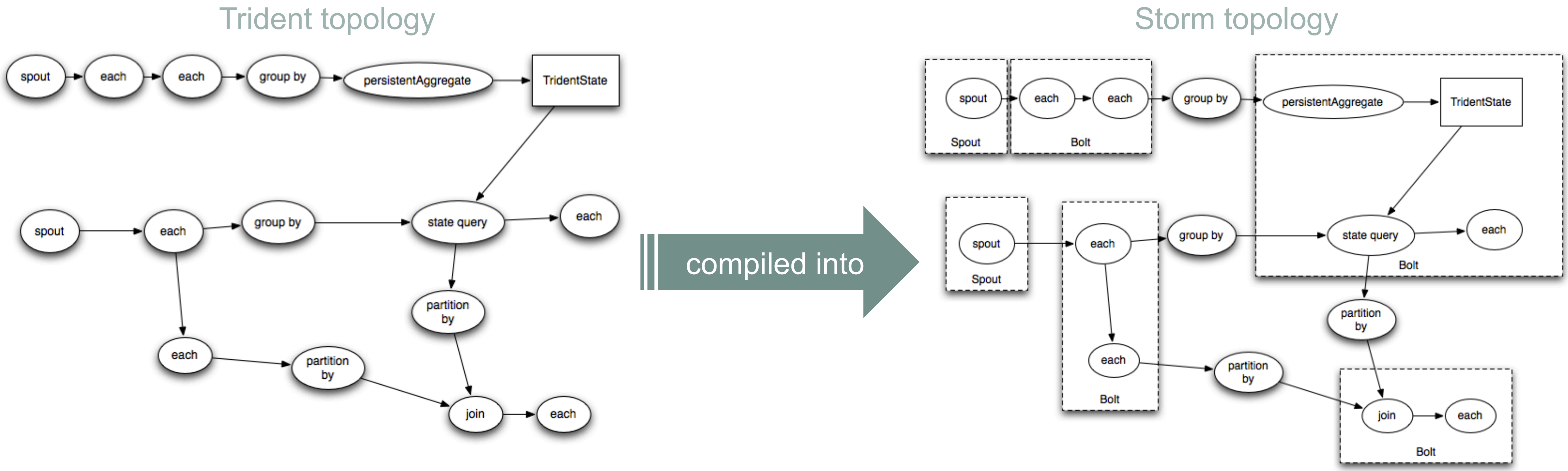
- Task: each spout/bolt executes as many threads of execution across the cluster

- Worker: a physical JVM that executes a subset of all the tasks for the topology



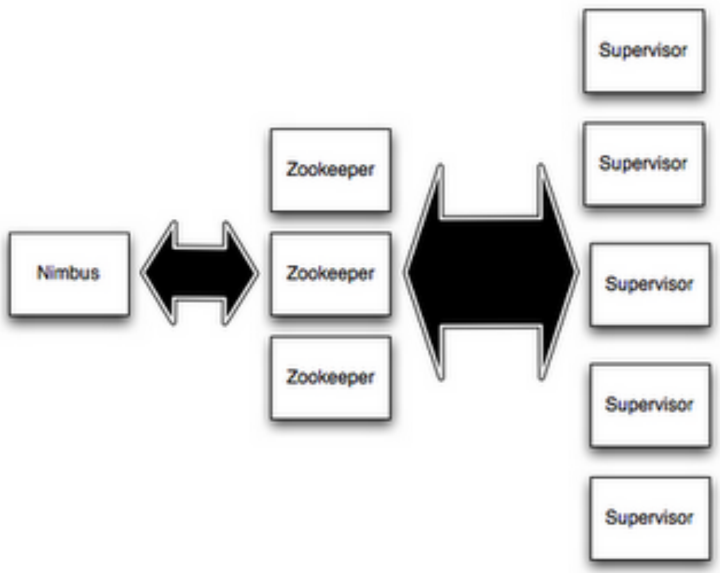
Trident—the ‘Cascading’ of Storm

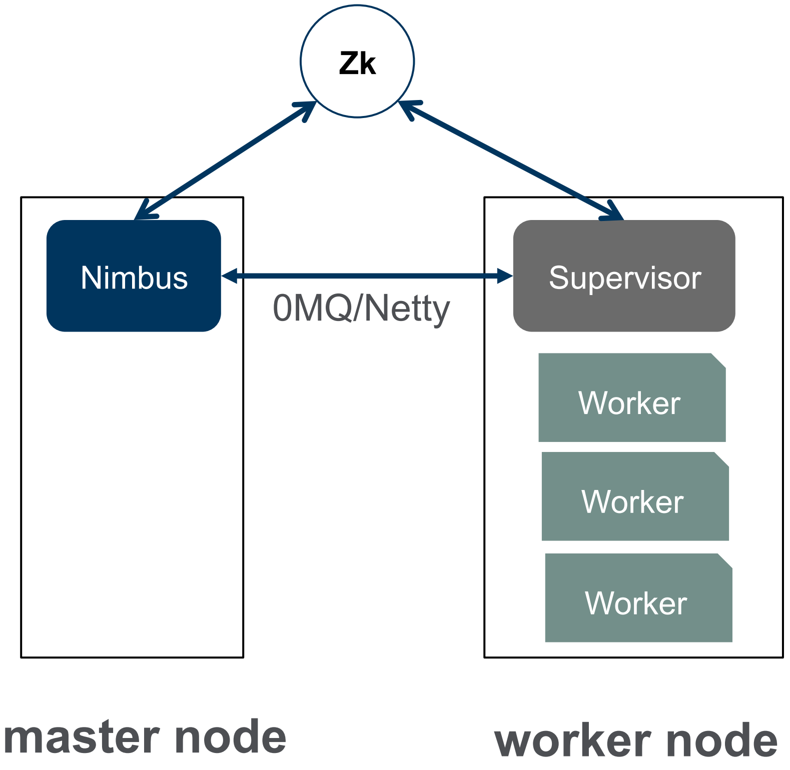
- High-level abstraction processing library on top of Storm
- Rich API with joins, aggregations, grouping, etc.
- Provides stateful, exactly-once processing primitives



storm.apache.org/documentation/Trident-tutorial.html

Execution


Nimbus
Zookeeper
Zookeeper
Zookeeper
Supervisor
Supervisor
Supervisor
Supervisor
Supervisor

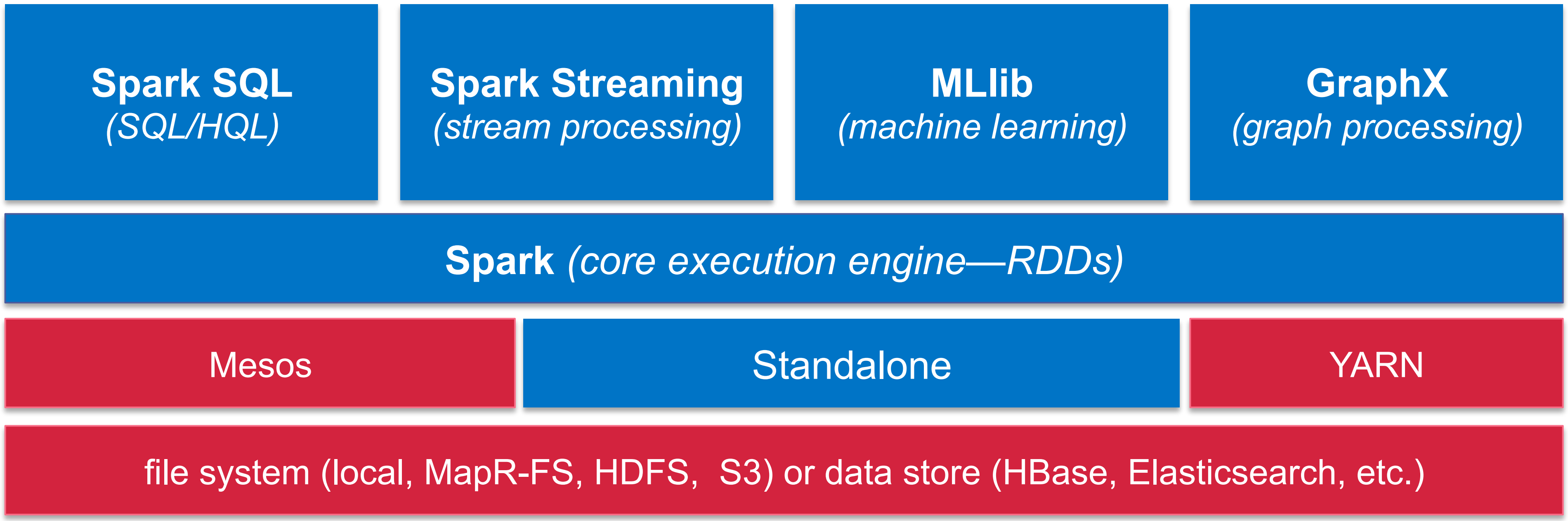

Zk
Nimbus
0MQ/Netty
Supervisor
Worker
Worker
Worker
master node
worker node

Storm Resources

- https://www.udacity.com/course/ud381
- http://www.manning.com/sallen/
- https://github.com/tdunning/storm-counts

Apache Spark

Apache Spark



Spark SQL
(SQL/HQL)

Spark Streaming
(stream processing)

MLlib
(machine learning)

GraphX
(graph processing)

Spark *(core execution engine—RDDs)*

Mesos

Standalone

YARN

file system (local, MapR-FS, HDFS, S3) or data store (HBase, Elasticsearch, etc.)

Continued innovation bringing new functionality, such as:

- **Tachyon** (Shared RDDs, off-heap solution)
- **BlinkDB** (approximate queries)
- **SparkR** (R wrapper for Spark)

http://spark.apache.org/

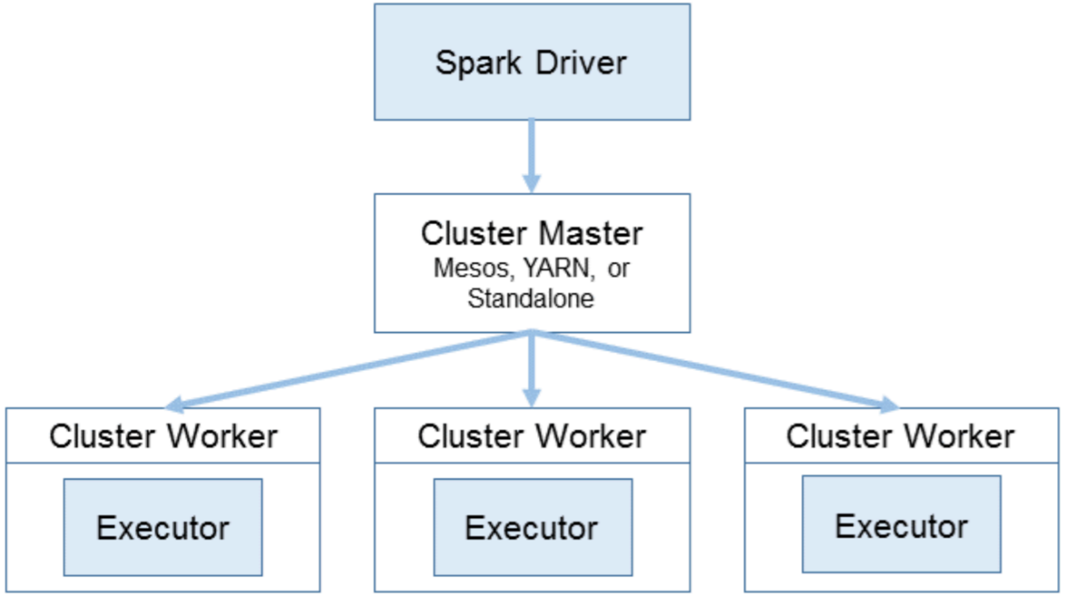
Sweet spot …

- Iterative Algorithms
  - machine learning
  - graph processing beyond DAG
- Interactive Data Mining
- Streaming Applications

Interfacing to permanent storage

- Local Files
  - file:///opt/httpd/logs/access_log
- Object Stores (e.g. Amazon S3)
- MapR-FS, HDFS
  - text files, sequence files, any other Hadoop `InputFormat`
- Key-Value datastores (e.g. Apache HBase)
- Elasticsearch

Cluster Managers



Standalone

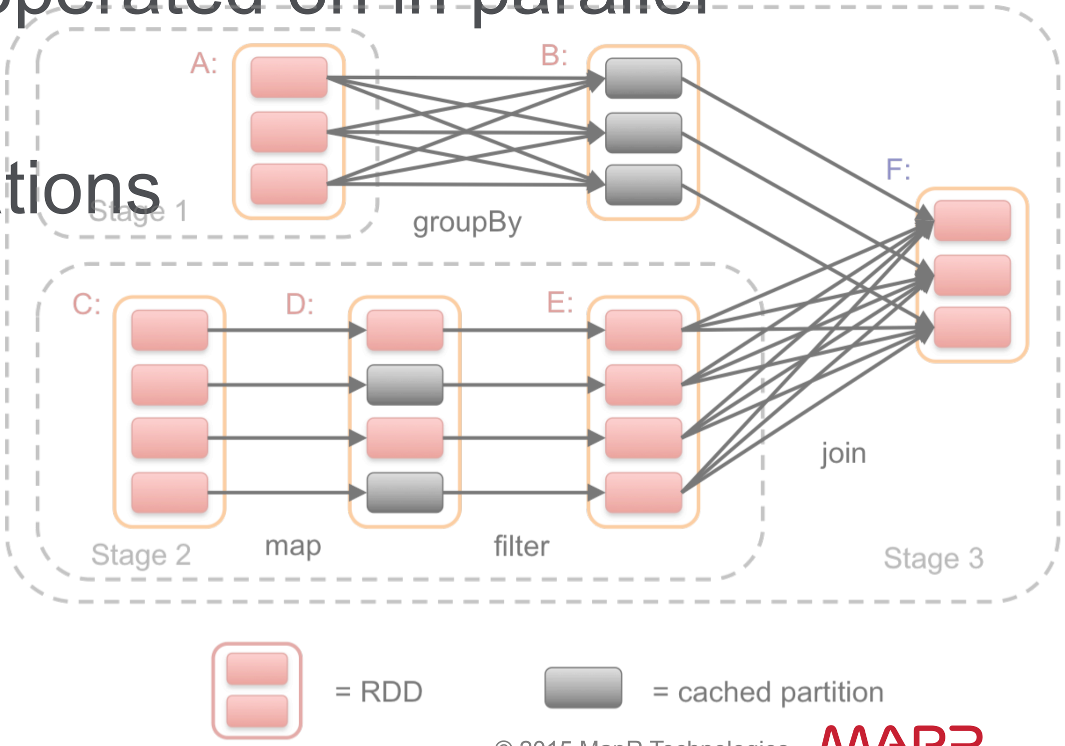
amazon web services™ EC2

YARN

MESOS

Resilient Distributed Datasets (RDD)

- RDD: core abstraction of Spark execution engine
- Collections of elements that can be operated on in parallel
- Persistent in memory between operations



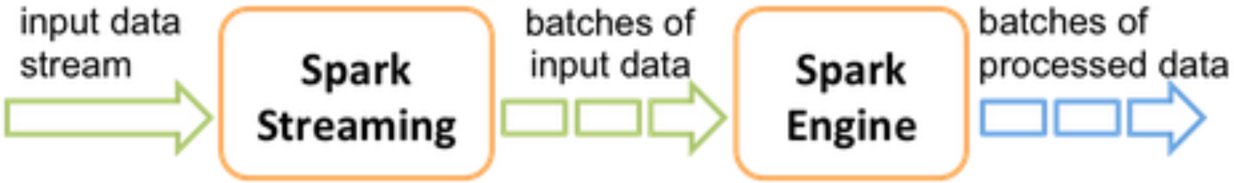
www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

RDD Operations

- Lazy evaluation is key to Spark
- Transformations
  - Creation of a new dataset from an existing:
    map, filter, distinct, union, sample, groupByKey, join, etc.
- Actions
  - Return a value after running a computation:
    collect, count, first, takeSample, foreach, etc.

Spark Streaming

- High-level language operators for streaming data
- Fault-tolerant semantics
- Support for merging streaming data with historical data

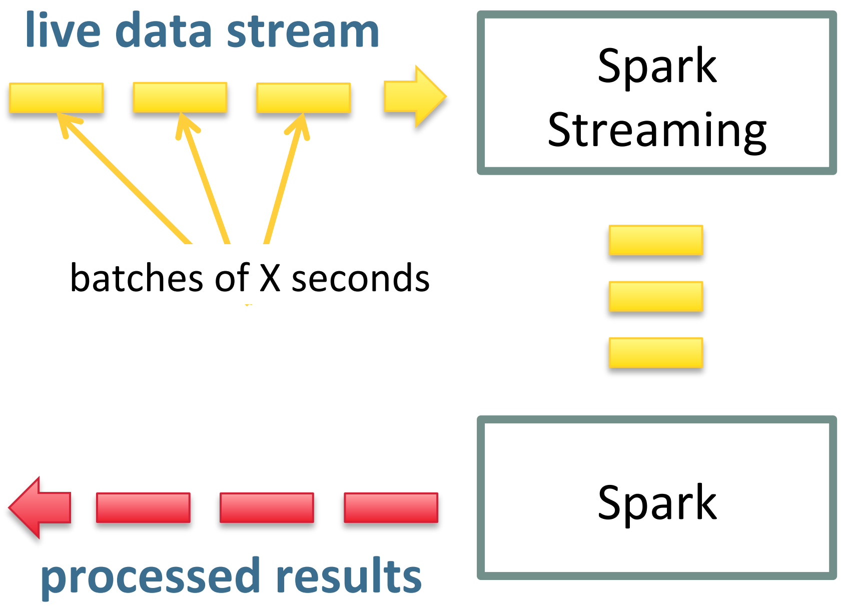


spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Streaming

Run a streaming computation as a series of small, deterministic batch jobs.

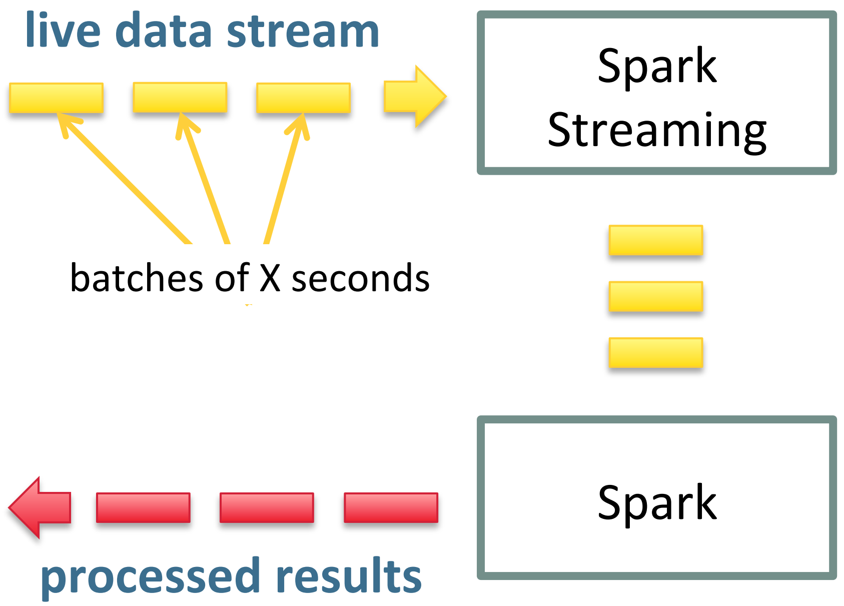
- Chop up live stream into batches of X seconds (DStream)
- Spark treats each batch of data as RDDs and processes them using RDD ops
- Finally, processed results of the RDD operations are returned in batches



Spark Streaming

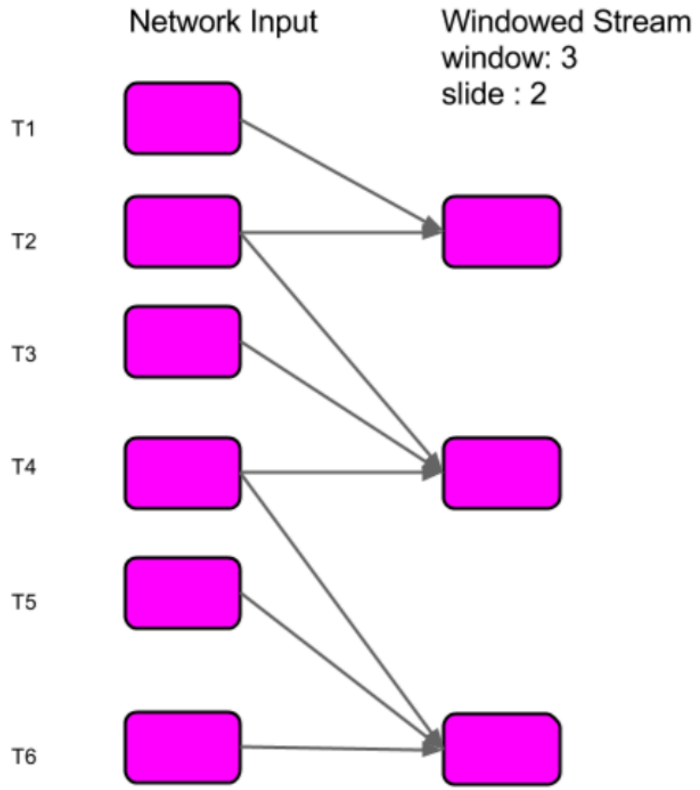
Run a streaming computation as a series of small, deterministic batch jobs.

- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



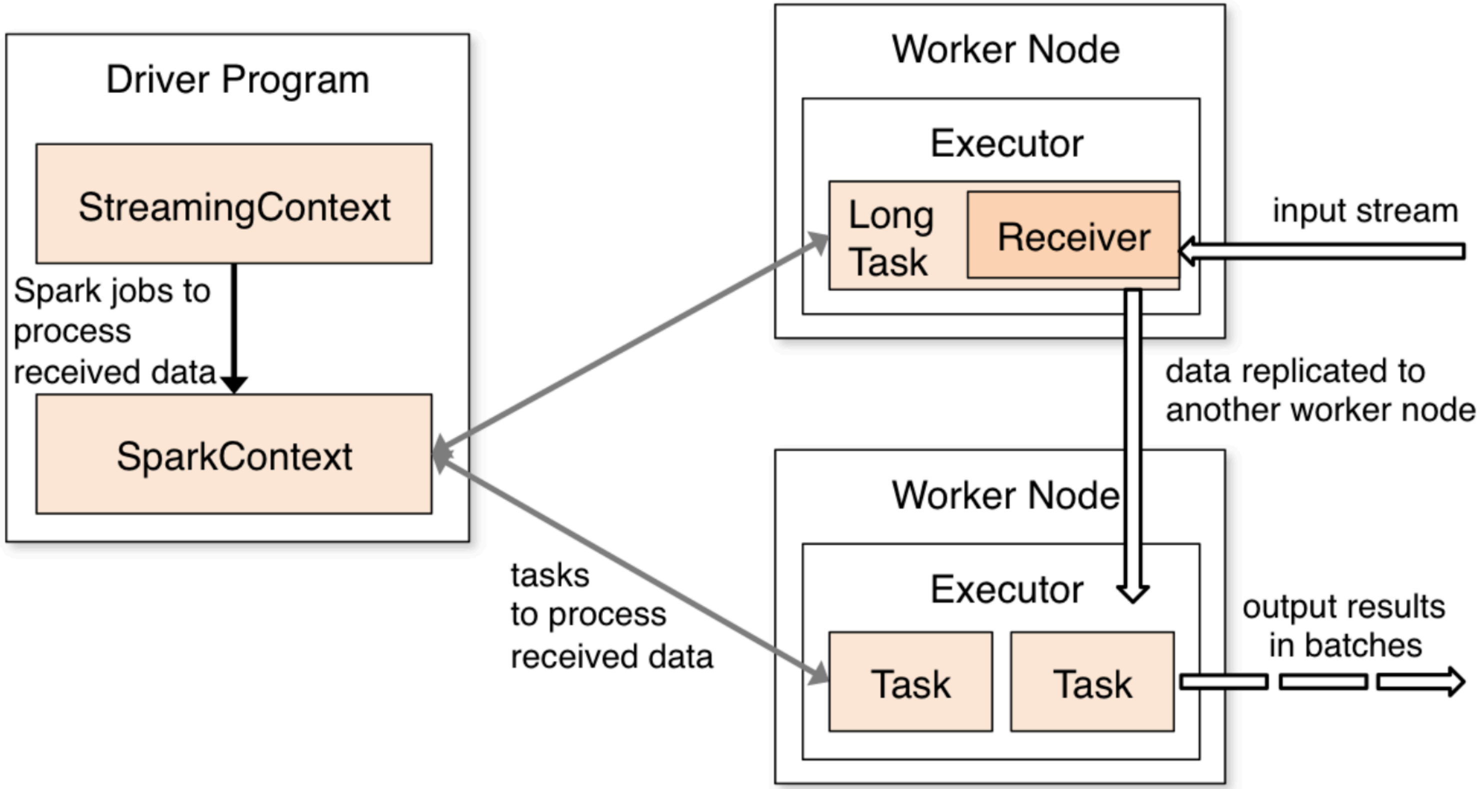
Spark Streaming: Transformations

- Stateless transformations
- Stateful transformations
  - checkpointing
  - windowed transformations
    - window duration
    - sliding duration



spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Spark Streaming: Execution


Driver Program
StreamingContext
Spark jobs to process received data
SparkContext
Worker Node
Executor
Long Task
Receiver
input stream
data replicated to another worker node
Worker Node
Executor
Task
Task
tasks to process received data
output results in batches

Spark Resources

- http://shop.oreilly.com/product/0636920028512.do
- http://databricks.com/spark-training-resources
- http://oreilly.com/go/sparkcert
- http://spark-stack.org
- https://www.mapr.com/blog/getting-started-spark-mapr-sandbox

Comparison

Samza vs Storm vs Spark

	Samza	Storm	Spark Streaming
Stream Source	Consumers	Spouts	Receivers
Stream Primitive	Message	Tuple	DStream
Stream Computation	Tasks	Bolts	Transformations

Samza vs Storm vs Spark

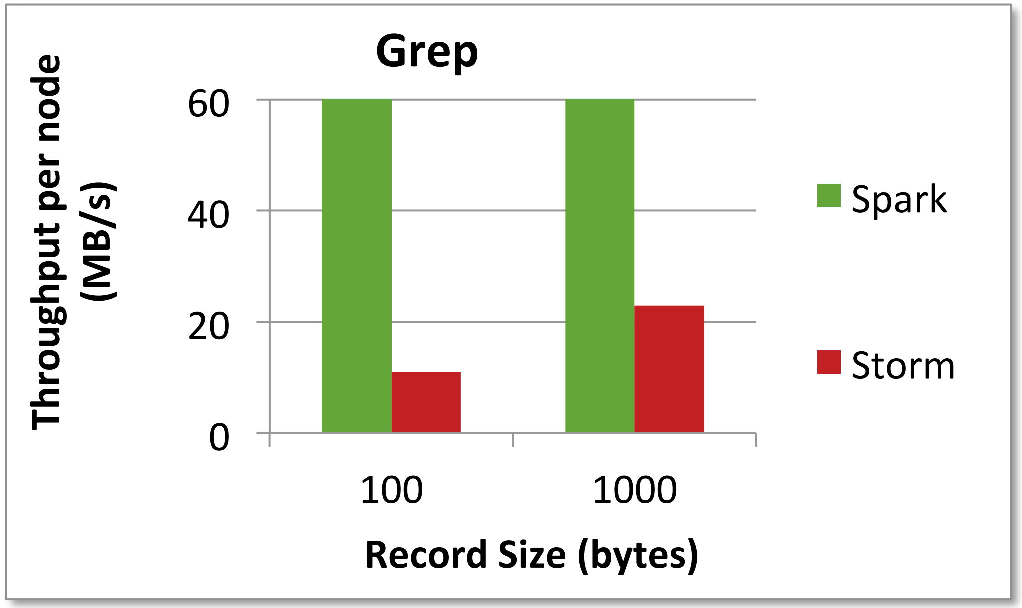
	Samza	Storm	Spark Streaming
processing model	one record at a time	one record at a time	micro-batch
latency	milliseconds	milliseconds	seconds
throughput	100k+ records per node per second	10k+ records per node per second	100k+ records per node per second
processing guarantees	at-least-once delivery; support for exactly-once planned	at-least-once / exactly once (with Trident)	exactly once
stateful operations	yes	no / yes (with Trident)	yes
language support	+	+++	++
community size/ committer & user base	+	+++	++
special	agile, state management, Kappa-native	distributed RPC	unified processing (batch, SQL, etc.)

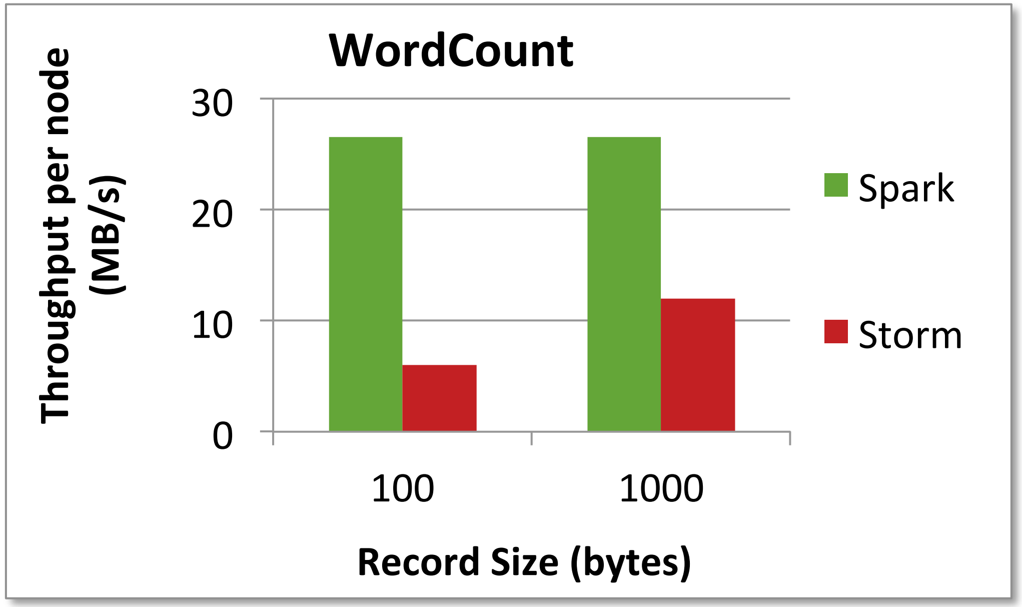
When to use what?

use case	Samza	Storm	Spark Streaming
filtering	✓	✓	✓
counting (incl. aggregations)		✓	✓
joins	✓		
distributed RPC		✓	
re-processing (aka Kappa architecture)	✓		✓
materialized view maintenance (cache invalidation)	✓		✓

Spark vs Storm: Throughput

- Spark Streaming: **670k** records/sec/node
- Storm: **115k** records/sec/node
- Commercial systems: **100-500k** records/sec/node





Comparison Resources

- http://www.zdatainc.com/2014/09/apache-storm-apache-spark/
- http://www.slideshare.net/ChicagoHUG/yahoo-compares-storm-and-spark
- http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming
- http://xinhstechblog.blogspot.ie/2014/06/storm-vs-spark-streaming-side-by-side.html
- http://samza.apache.org/learn/documentation/0.8/comparisons/storm.html
- http://samza.apache.org/learn/documentation/0.8/comparisons/spark-streaming.html
- https://www.mapr.com/blog/spark-streaming-vs-storm-trident-whiteboard-walkthrough

Q&A

Engage with us!

@MapR

maprtech

MapR Technologies

+MaprTechnologies

jscott@mapr.com

maprtech