Hadoop2 – Spark – Storm Training

Venkat Krishnan
Technology Evangelist
Corporate Trainer
[Hadoop. Spark. Storm and HDInsights]
www.evenkat.com
venkatkrishnan@vsnl.net
http://tech.groups.yahoo.com/group/venkatjava
http://www.linkedin.com/in/venkitakrishnan

Real-Time Versus Batch Processing

Real-time and batch processing are very different.

Factors		Real-Time	Batch
Data	Age	Real-time – usually less than 15 minutes old	Historical – usually more than 15 minutes old
	Location	Primarily in memory – moved to disk after processing	Primarily on disk – moved to memory for processing
Processing	Speed	Sub-second to few seconds	Few seconds to hours
	Frequency	Always running	Sporadic to periodic
Clients	Who	Automated systems only	Human & automated systems
	Type	Primarily operational applications	Primarily analytical applications

Consider These Scenarios

What if you are a financial services company and you need to analyze transactions in real time to prevent fraud?

What if you are a telecom company and you need to analyze network traffic in real time to allocate cell towers dynamically?

What if you need to monitor application logs in real time to respond to application anomalies as they happen?

What if you are a trucking company and you need to analyze real-time data to modify drive routes to save time and fuel costs?

Apache Storm can help in these types of scenarios.

Real-Time Streaming Data

The previous scenarios all had one thing in common:

- The availability of continuous streams of real-time data

Apache Storm is a distributed computation system for processing continuous streams of real-time data.

- Storm augments the batch processing capabilities provided by MapReduce

Storm is commonly used for:

- Stream processing
- Continuous computation
- Distributed remote procedure calls (DRPC)

What is Spark Streaming?

- Extends Spark for doing large scale stream processing
  - ➢Scales to hundreds of nodes and achieves second-scale latencies
- Efficient and fault-tolerant stateful stream processing
  - ➢Simple batch-like API for implementing complex algorithms

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions
- Distributed stream processing framework is required to
  - Scale to large clusters (hundreds of machines)
  - Achieve low latency (few seconds)

Integration with Batch Processing

- Many environments require processing same data in live streaming as well as batch post processing

- Existing framework cannot do both
  - Either do stream processing of 100s of MB/s with low latency
  - Or do batch processing of TBs / PBs of data with high latency

- Extremely painful to maintain two different stacks
  - Different programming models
  - Double the implementation effort
  - Double the number of bugs

Spark Streaming Overview

- Extension of Spark Core
  - Library built on top of Spark Core
  - Reuses a lot of the same APIs
- Utilizes a micro-batch architecture
  - Process batches of data, as small as 1s latency
- Process batches of data called Dstreams
  - Discretized Streams of data
  - Share very little physically with RDD's, but are used similar to RDD's

Spark Streaming Context

- Main entry point for streaming application

- Created from a Spark Context and an interval time

```
ssc = StreamingContext(sparkContext, batchDuration)
```

- Creating a streaming context with 2 working threads, named `StreamingApp`, and a batch duration of 1 second

```
>>>from pyspark.streaming import StreamingContext
>>>sc = SparkContext("local[2]","StreamingApp")
>>>ssc = StreamingContext(sc, 1)
```

Recievers

- Data comes from a receiver running on an Executor

- The receiver breaks the data into batches of the duration specified in the `StreamingContext`
  - ➢Converts the data from continuous to discrete

- Basic `socketTextStream`

```
>>>inputDS = ssc.socketTextStream("localhost", 2222)
```

Working with Dstreams

- Dstreams heavily reuse the APIs from Spark Core
  - ➢Some special APIs for stateful transformations

- Example of doing a Spark Streaming WordCount

```
>>>wcDS = inputDS.flatMap(lambda line: line.split(" \
").map(lambda word: (word,1)).reduceByKey(lambda a,b: a+b)
>>>wcDS.pprint()
```

Input Sources

- Input sources are defined as receivers

- Every input Dstream is defined by a receiver, which is configured to a source

```
>>>inputDS = ssc.socketTextStream(host, port)
```

- As of Spark 1.4.1 only a few receivers are supported by python
  - Kafka ##Refer to the documentation for setting this up
  - `textFileStream(someDir)`
  - `textSocketStream(host, port)`

Stateless Transformations (1 of 3)

- Data does not depend on any data in a previous batch is processed using stateless transformation
- Stateless transformations are just like Core transformations
- Examples of transformations that are stateless
  - `map`
  - `flatMap`
  - `filter`
  - `reduceByKey`
  - `repartition`
- These transformations are applied to the current Dstream, and not across previous Dstreams

Stateless Transformation (2 of 3)

```
>>>inputDS = ssc.socketTextStream("localhost", 2222)

>>>wcDS = inputDS.flatMap(lambda line: line.split(" \
").map(lambda word: (word,1)).reduceByKey(lambda a,b: a+b)

>>>wcDS.print()

>>>###This will perform a word count on the Dstream
```

Stateless Transformations (3 of 3)

- Can combine data from multiple receivers using `join`, `union`, etc
  - This works if the receivers have the same batch duration
  - Again, this only is applied for the current point in time
- Dstreams are not RDDs, but conceptually similar
- Dstreams can be converted to RDDs using the `transform()` API
  - A common use of the `transform()` API is to reuse code written for batch processing
  - Not all RDD operations are exposed as Dstream operations, using `transform` is an acceptable work around
  - `>>> rdd.transform(lambda rdd: somefunction(rdd))`

Combining Batch and Streaming Processing

- Using the `transform` API, Spark applications combine data streams with static datasets

```
dataset = sc.textFile("somefile.txt")

inputDS.transform(lambda Dstream:
Dstream.join(dataset).map(…))
```

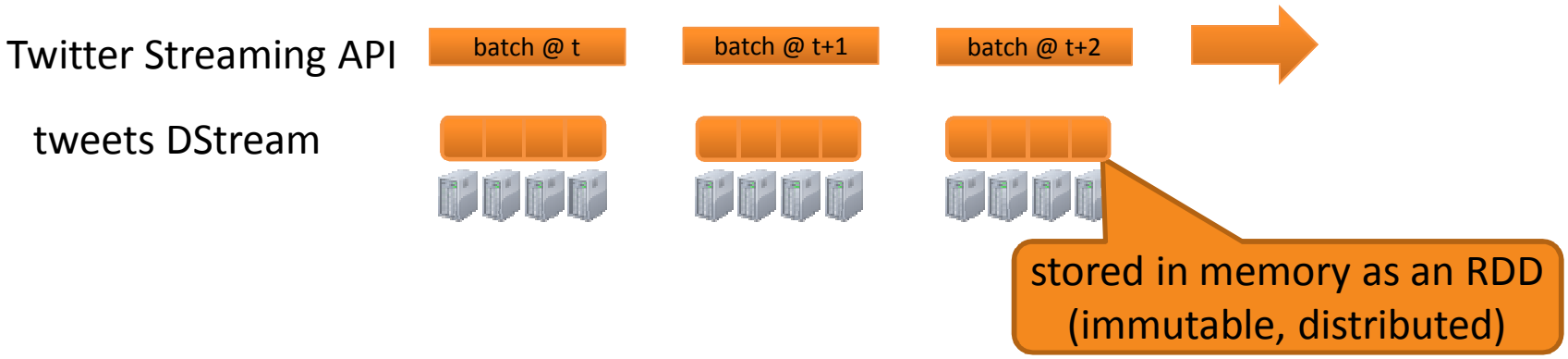
Output Operations

- Output operations allow DStream's data to be pushed out external systems
- Output operations trigger the actual execution of all the DStream transformations (similar to actions for RDDs)
- Examples of Output operations:
  - `pprint()`
  - `saveAsTextFile`
  - `foreachRDD`

Example – Get Hashtags from Twitter

```
tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

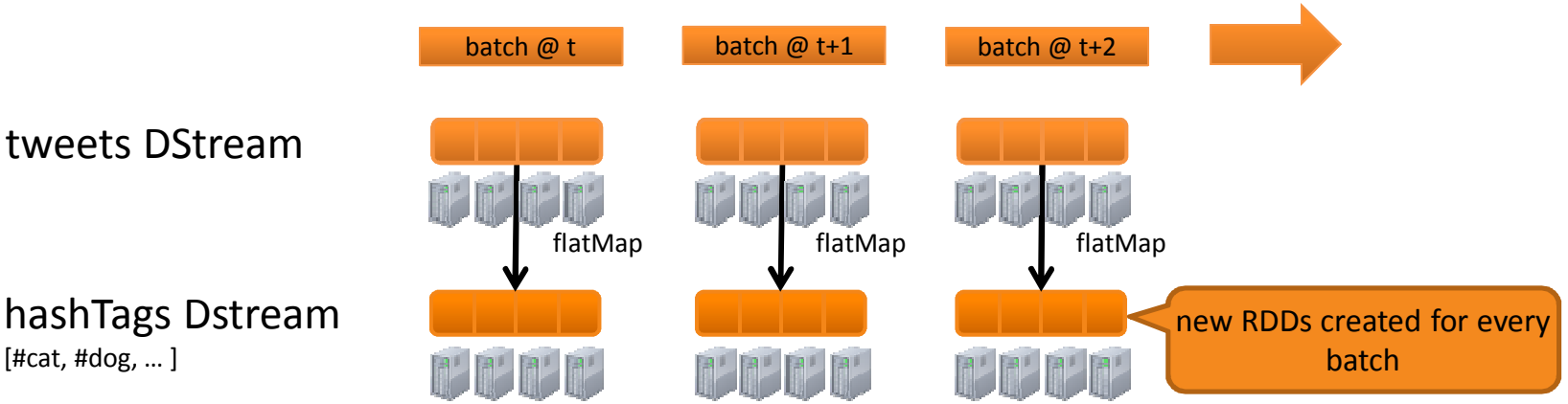


Example – Get Hashtags from Twitter

```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
```

new DStream

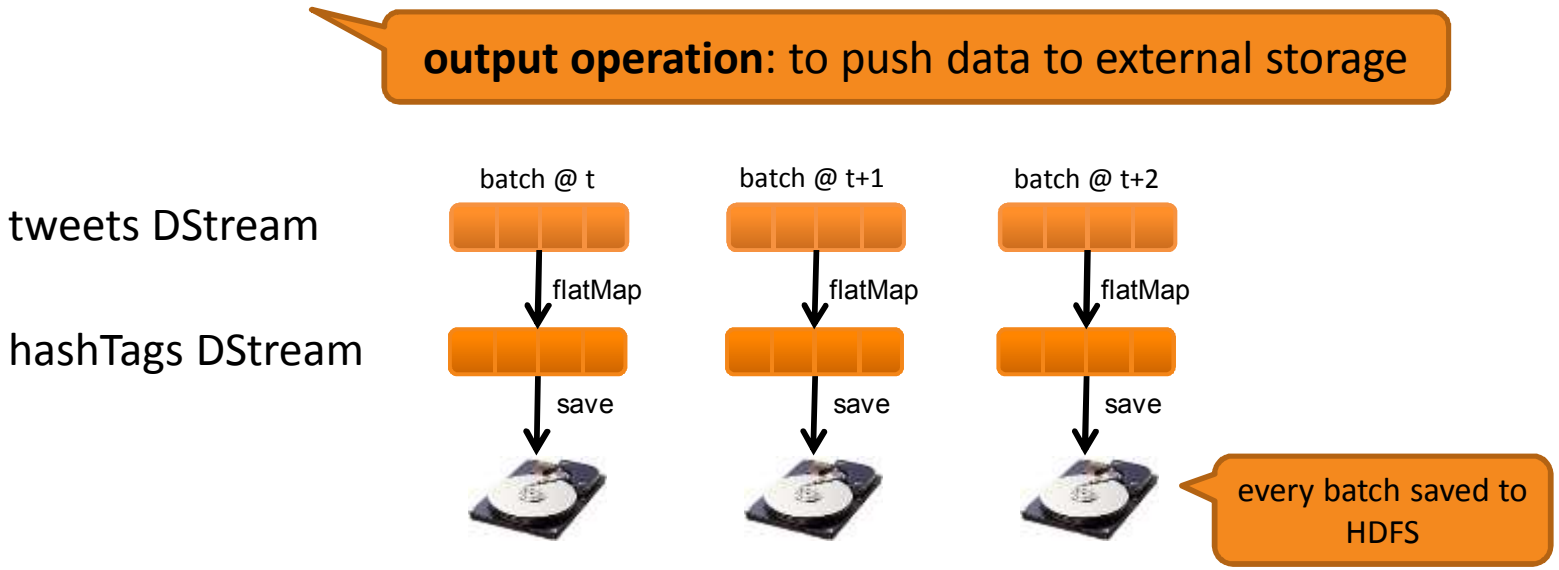
transformation: modify data in one DStream to create another DStream



Example – Get Hashtags from Twitter

```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.saveAsTextFile("hdfs://...")
```

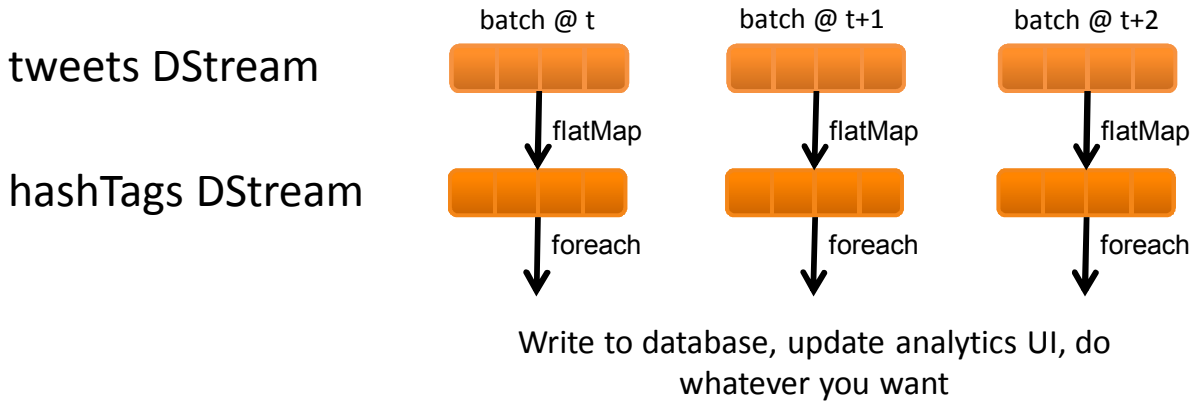
output operation: to push data to external storage



Example – Get Hashtags from Twitter

```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Combinations of Batch and Streaming Computations

Intermix RDD and DStream operations

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets.transform(lambda tweetsRDD: ( \
                        tweetsRDD.join(spamHDFSFile).filter(...)
      ))
```

Unifying Batch and Stream Processing Models

Spark Batch program on Twitter log file using RDDs:

```
tweets = sc.textFile("hdfs://...")
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.saveAsTextFile("hdfs://...")
```

Spark Streaming program on Twitter stream using Dstreams:

```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.saveAsTextFile("hdfs://...")
```

Storm Training

Venkat Krishnan
Technology Evangelist

1 – Real-Time Data Processing

Real-time versus batch data processing

Hortonworks

Learning Objectives

When you complete this lesson you should be able to:

- Identify whether Storm performs batch or real-time processing
- Recognize the differences between batch and real-time processing
- List reasons why companies deploy Storm
- Describe Storm use cases

Consider These Scenarios

What if you are a financial services company and you need to analyze transactions in real time to prevent fraud?

What if you are a telecom company and you need to analyze network traffic in real time to allocate cell towers dynamically?

What if you need to monitor application logs in real time to respond to application anomalies as they happen?

What if you are a trucking company and you need to analyze real-time data to modify drive routes to save time and fuel costs?

Apache Storm can help in these types of scenarios.

Real-Time Streaming Data

The previous scenarios all had one thing in common:

- The availability of continuous streams of real-time data

Apache Storm is a distributed computation system for processing continuous streams of real-time data.

- Storm augments the batch processing capabilities provided by MapReduce

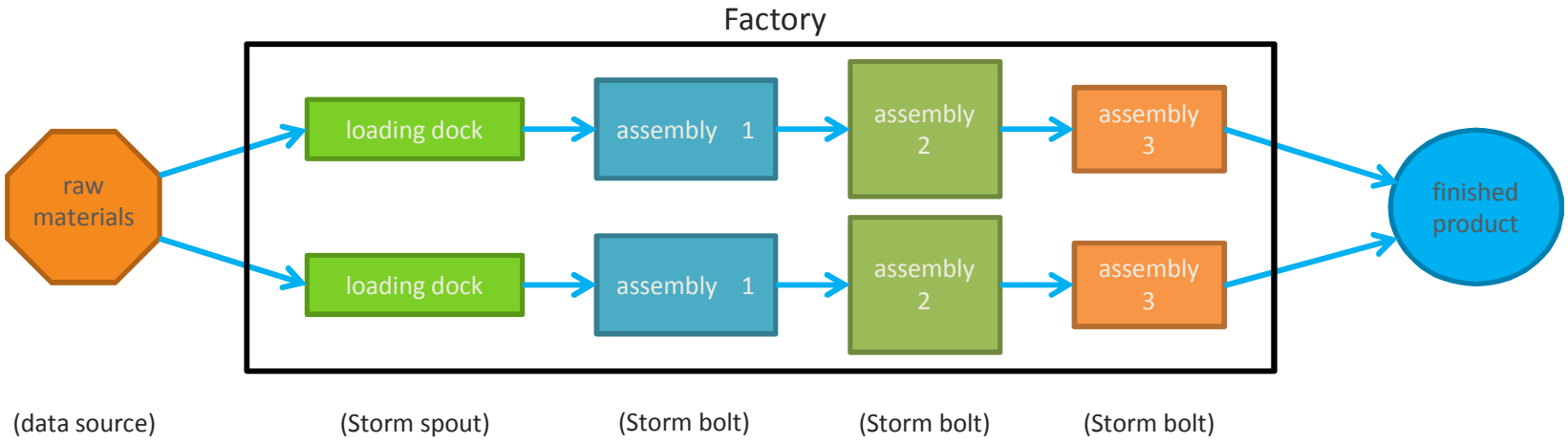
Storm is commonly used for:

- Stream processing
- Continuous computation
- Distributed remote procedure calls (DRPC)

The Assembly Line Model

Storm processes real-time data using an assembly line model similar to the automotive industry.

- Complex tasks are accomplished step by step by a series of workers performing different operations
- There are identical, parallel assembly lines to increase throughput
- In Storm, the assembly line is not always a line; there are branches and even directed acyclic graphs



Real-Time Versus Batch Processing

Real-time and batch processing are very different.

Factors		Real-Time	Batch
Data	Age	Real-time – usually less than 15 minutes old	Historical – usually more than 15 minutes old
	Location	Primarily in memory – moved to disk after processing	Primarily on disk – moved to memory for processing
Processing	Speed	Sub-second to few seconds	Few seconds to hours
	Frequency	Always running	Sporadic to periodic
Clients	Who	Automated systems only	Human & automated systems
	Type	Primarily operational applications	Primarily analytical applications

Knowledge Check

Match each question with its correct answer.

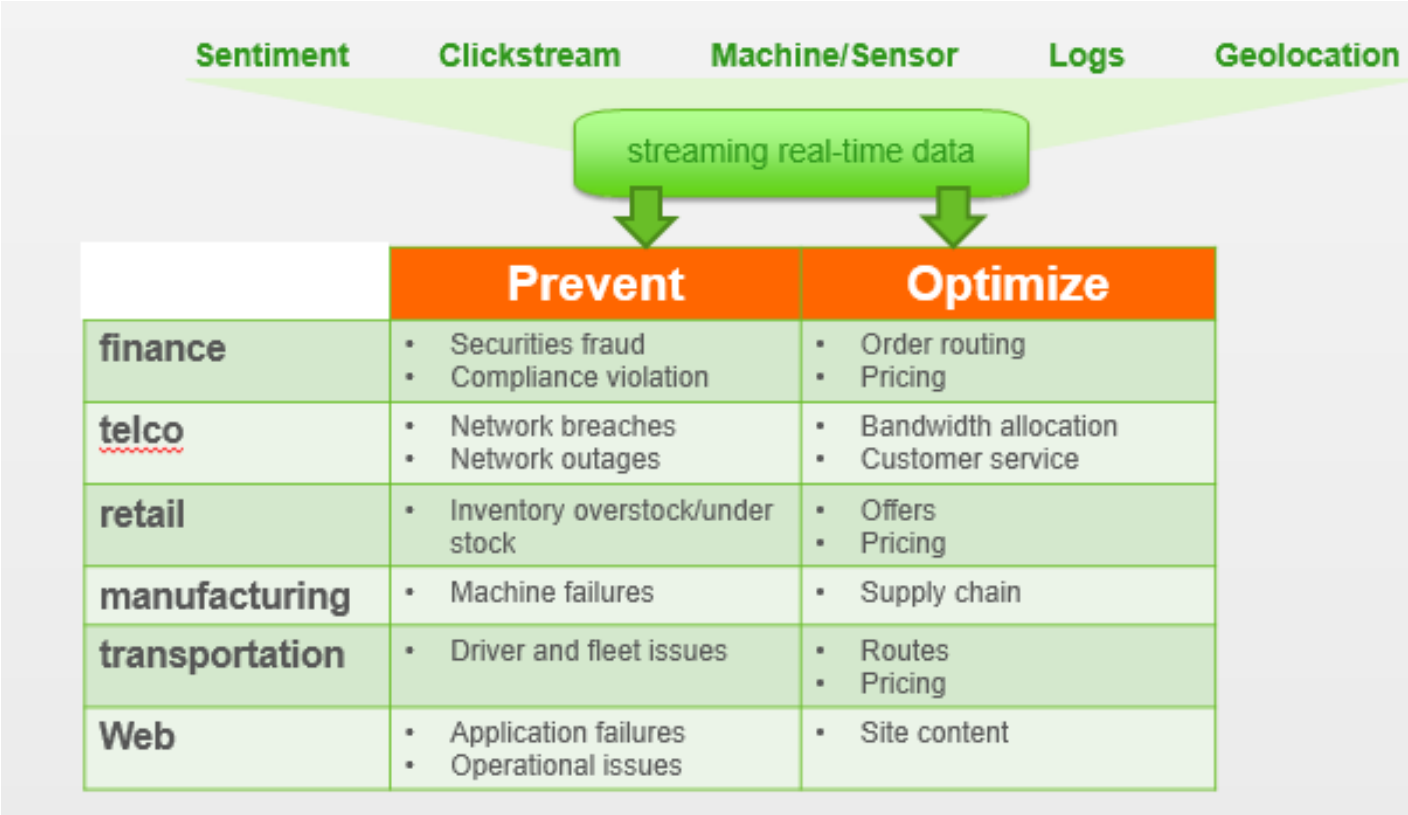
1. What is the typical age of real-time data?
2. How often is real-time data processed?
3. What is the typical age of batch data?
4. What is typically the client of a real-time processing system?
5. How often is batch data processed?
6. What is typically the client of a batch-processing system?
7. What type of application commonly processes batch data?
8. What type of application commonly processes real-time data?

a. an automated system
b. typically older than 15 minutes
c. historical analysis application
d. typically less than 15 minutes old
e. processed continually
f. processed sporadically
g. a human or an automated system
h. an operational dashboard application

Why Enterprises Choose Storm

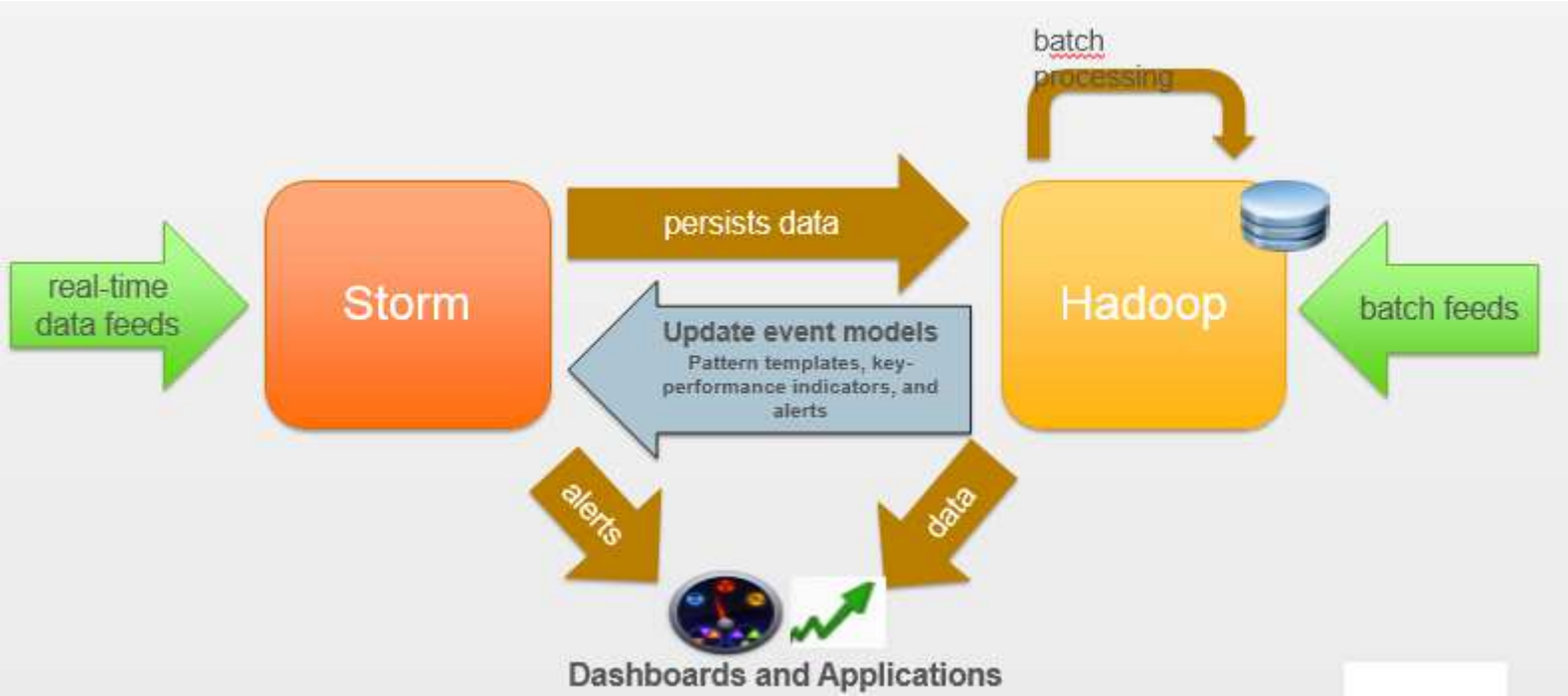
Highly scalable	• Horizontally scalable like Hadoop
Fast	• For example, a 10 node cluster can process 1M 100 byte messages per second per node
Fault tolerant	• Highly redundant services and operation with automated failover capabilities
Guarantees processing	• Supports at-least-once and exactly-once processing semantics
Language agnostic	• Data-processing logic can be written in multiple languages
Apache project	• Brand, governance, and a large active community

Storm Use Cases – Prevent and Optimize



	Prevent	Optimize
finance	• Securities fraud • Compliance violation	• Order routing • Pricing
telco	• Network breaches • Network outages	• Bandwidth allocation • Customer service
retail	• Inventory overstock/under stock	• Offers • Pricing
manufacturing	• Machine failures	• Supply chain
transportation	• Driver and fleet issues	• Routes • Pricing
Web	• Application failures • Operational issues	• Site content

Integrating Real-Time Processing Workflows



A Storm Topology

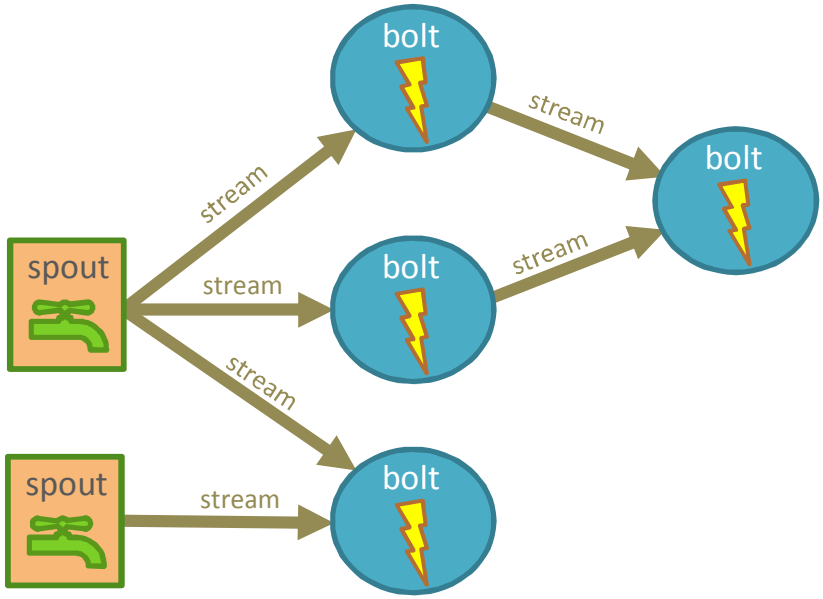
Storm data processing occurs in a topology.

A topology consists of spout and bolt components.

Spouts and bolts run on the systems in a Storm cluster.

Multiple topologies can co-exist to process different data sets in different ways.

This lesson provides information about topology components.



Storm topology

Tuples

The tuple is the fundamental data unit in Storm.

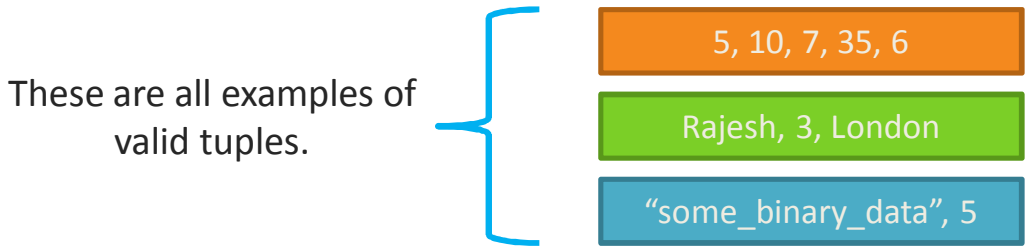
- A tuple is a unit of work to process
- A Storm topology processes tuples

A tuple is an ordered list of values.

- The values can be of any type

In Storm, each field in a tuple must assigned a field name.

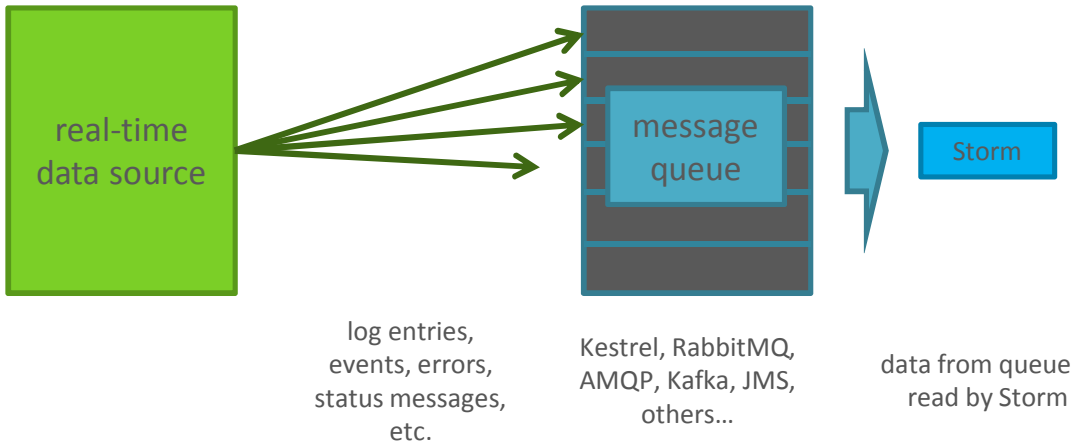
- For example, the fields in a 5-tuple might be assigned the names *name*, *user-id*, *age*, *salary*, and *currency*



Message Queues

Message queues are often the source of the data processed by Storm.

Storm integrates with many types of message queues.



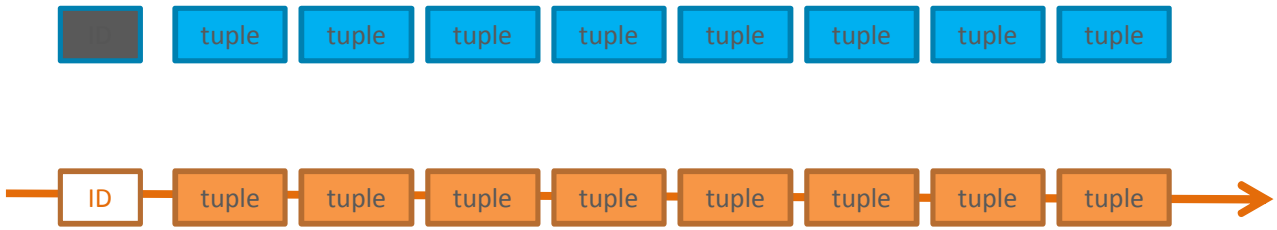
Streams

The stream is one of the core abstractions in Storm.

A stream is an unbounded sequence of tuples.

Every stream is assigned a stream ID when it is created.

- The default stream ID is *default*
- For more information about assigning stream IDs, see https://storm.apache.org/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html



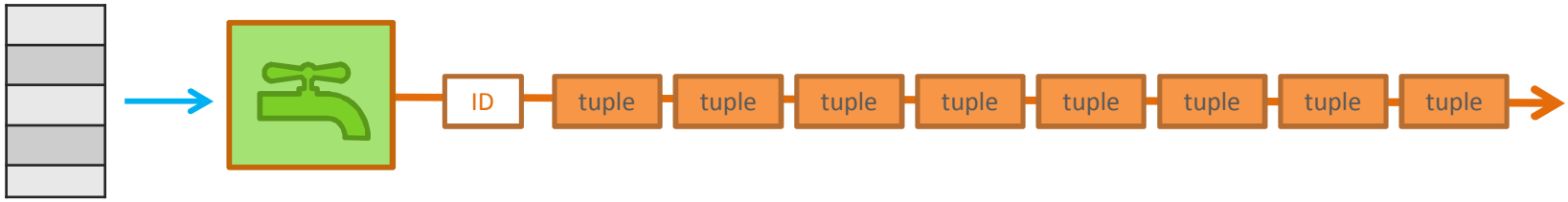
Spouts

A spout is a source of streams in a topology. Spouts:

- Act as an adapter between external data source and Storm
- Read data from an external source (commonly a message queue)
- Emit one or more streams of spout tuples into a topology
  - Each stream requires a unique stream ID

Spouts can be reliable or unreliable.

- A reliable spout replays a tuple that failed to process
- An unreliable spout does not replay a tuple that failed to be processed



Bolts

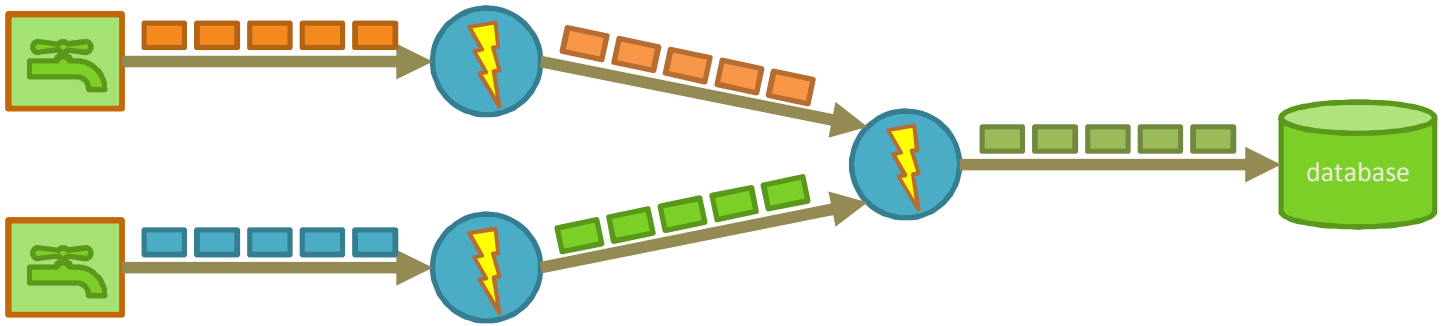
A bolt implements the data-processing logic.

- A bolt processes each tuple in a stream and emits a new stream of tuples

A bolt can run a function or filter, aggregate, or join tuples.

A bolt can also send tuples to other message queues, databases, HDFS, and more.

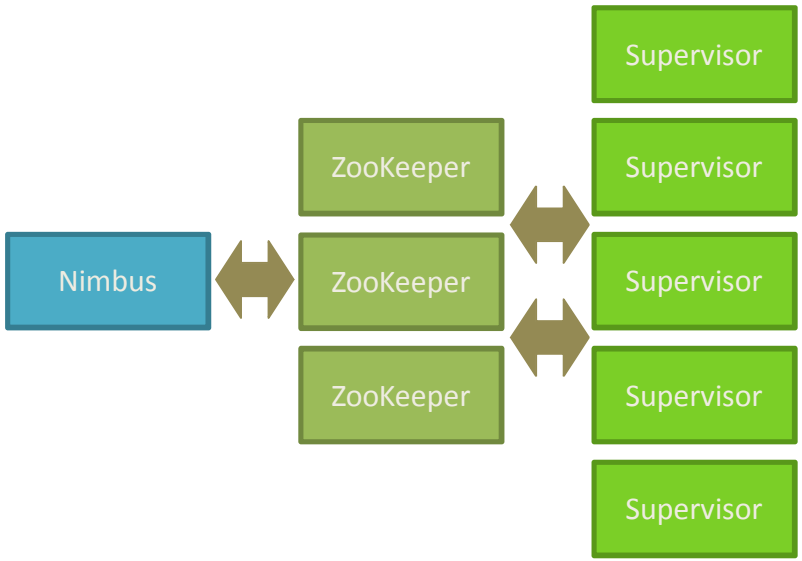
Complex transformation and analysis is possible by connecting multiple bolts together.



Storm Architecture

Storm is implemented as a cluster of machines.

- Nimbus – master node daemon
  - Similar function to YARN ResourceManager
  - Distributes program code around cluster
  - Assigns tasks
  - Handles failures
  - Responds to topology administration requests
- Supervisor – slave node daemons
  - Similar function to YARN NodeManager
  - Runs bolts and spouts as tasks
  - Commonly runs on Hadoop slave machines
- ZooKeeper
  - Cluster coordination
  - Stores cluster metrics



Hadoop MapReduce and Storm Topologies Compared

A Storm cluster is superficially similar to a Hadoop cluster.

Storm and Hadoop provide a highly parallel processing cluster to reliably process massive amounts of data.

Both Storm and Hadoop clusters can share the same machines.

Each is implemented using different daemons and libraries.

Hadoop Cluster and MapReduce	Storm Cluster and Topologies
Scalable	Scalable
Guarantees no data loss	Can guarantee no data loss
Batch processing	Real-time processing
Jobs run to completion	Topologies run until manually stopped
Stateful nodes	Stateless nodes

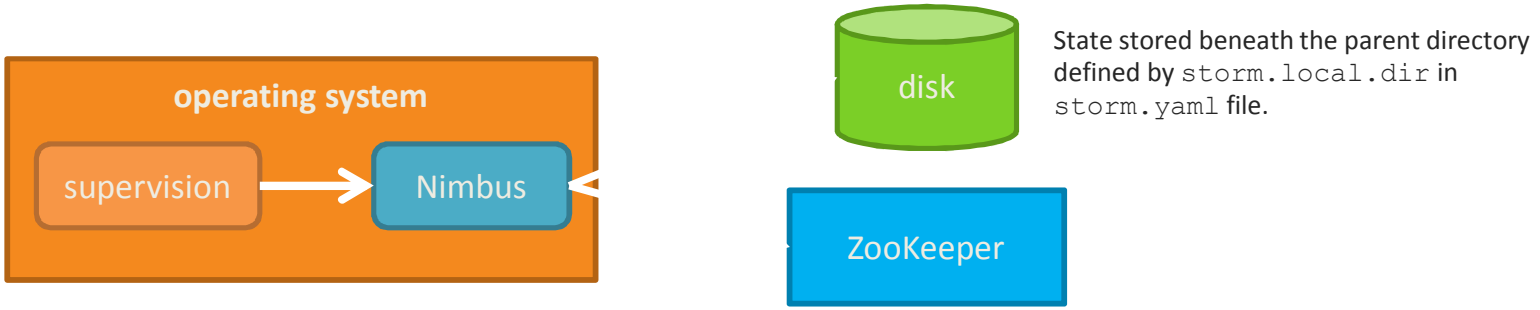
Nimbus

Nimbus is implemented as a Thrift daemon.

- `ps -ef | grep nimbus`
- It is fail-fast and stateless
- State is maintained on local disk and in ZooKeeper

The Nimbus daemon should be run under supervision.

Nimbus is a single point for configuration changes but not failure.



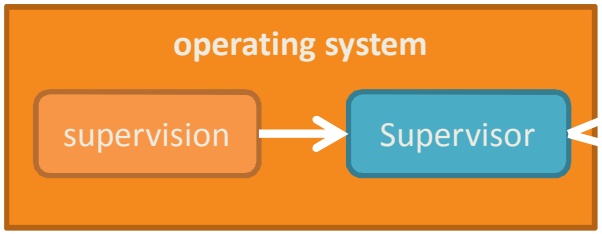
Supervisor

Supervisor is implemented as a Thrift daemon.

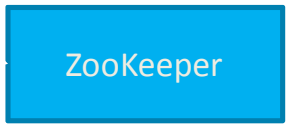
- `ps -ef | grep supervisor`
- It is fail-fast and stateless
- State is maintained on local disk and in ZooKeeper

The Supervisor daemon should be run under supervision.

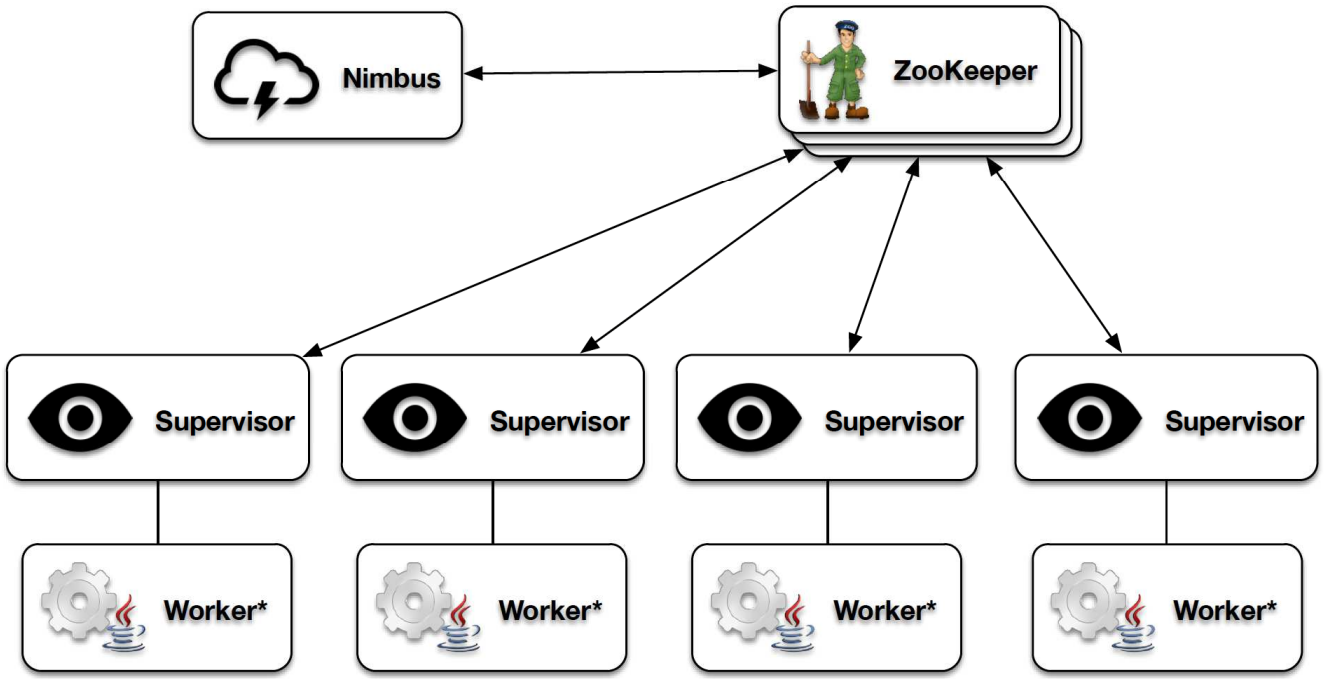
Supervisor failures do not affect running topologies.



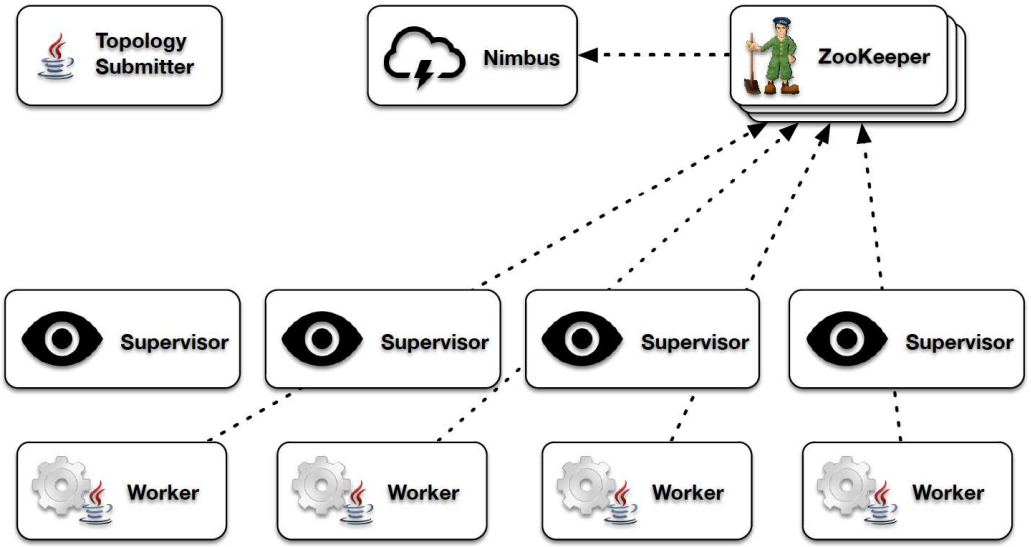
State stored beneath the parent directory defined by `storm.local.dir` in `storm.yaml` file.



Storm Cluster View

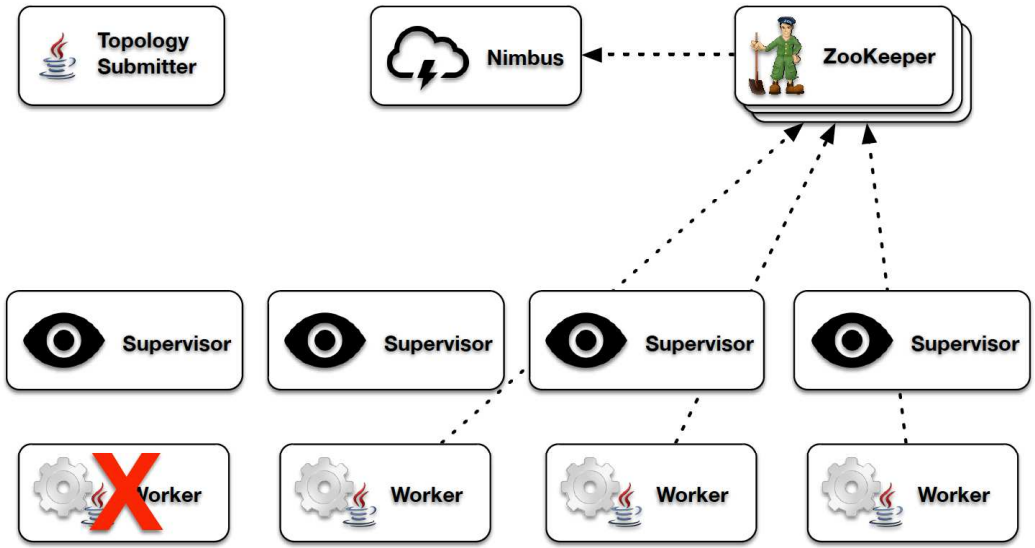


Fault Tolerance



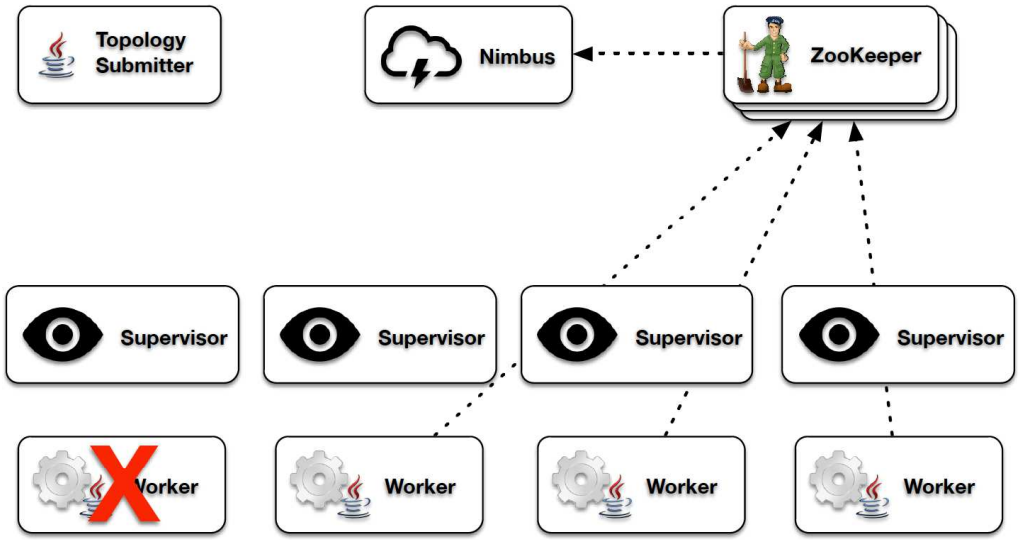
Workers heartbeat back to Supervisors and Nimbus via ZooKeeper, as well as locally.

Fault Tolerance



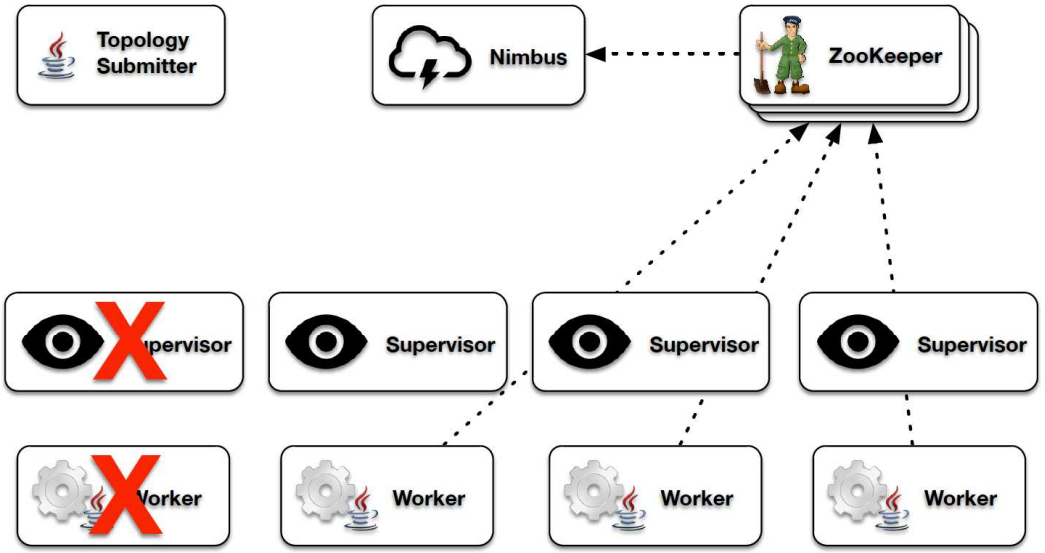
If a worker dies (fails to heartbeat), the Supervisor will restart it

Fault Tolerance



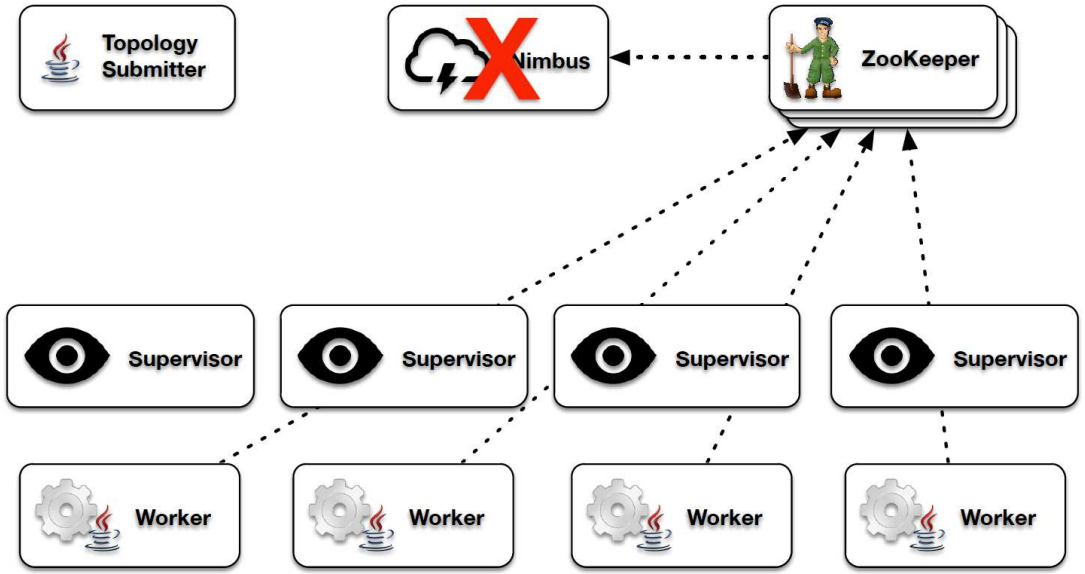
If a worker dies repeatedly, Nimbus will reassign the work to other nodes in the cluster.

Fault Tolerance



If a supervisor node dies, Nimbus will reassign the work to other nodes.

Fault Tolerance



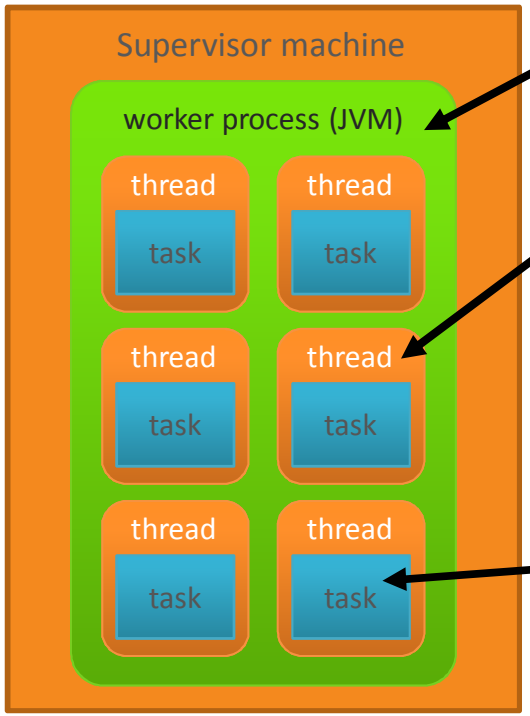
If Nimbus dies, topologies will continue to function normally, but won't be able to perform reassignments.

Worker Processes, Executors, and Tasks

Each Supervisor machine uses three entities to run a subset of a topology.

- Worker process
- Executor
- Task

Adding more machines with more of these entities can increase Storm processing scalability.



Each Supervisor machine can run one or more worker processes. Each worker process is a Java virtual machine.

Each worker process runs one or more threads, called executors.

- Executors run tasks
- One task per executor, by default
- If an executor runs more than one task, all tasks must be the same component type (spout or bolt)

A task performs the spout or bolt data processing. A spout or bolt can run in parallel across many tasks.

Storm Configuration

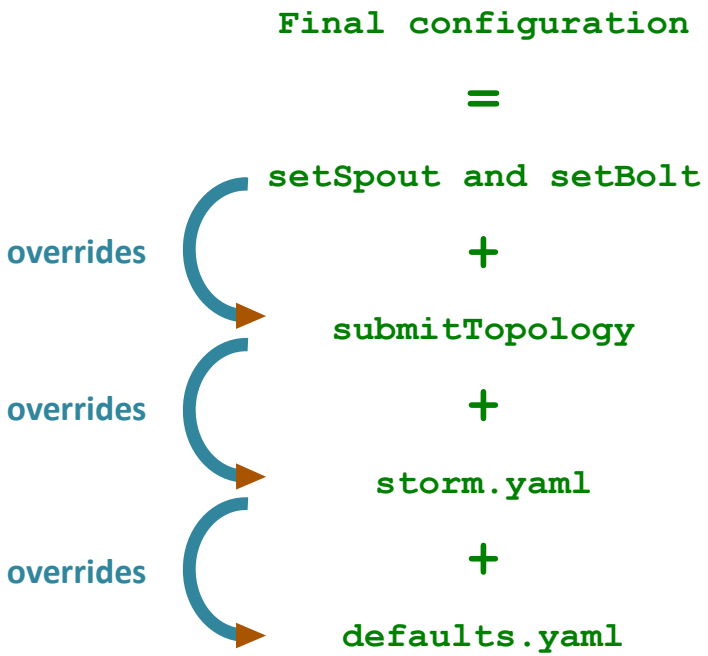
Storm has many configuration parameters.

- Most of the default settings can be used “as is”

Nimbus, Supervisors, topologies, spouts, and bolts are configurable.

- Spouts, bolts, and topologies can be individually configured

The final configuration used by a Storm component is derived by evaluating multiple configuration locations.



The `defaults.yaml` File

There are multiple references to the `defaults.yaml` file in the Storm documentation.

- There is no `defaults.yaml` file in the current HDP distribution
- Starting with Storm version 0.8.2, the file is no longer included in the standard Storm download

Default configuration settings in `defaults.yaml` are compiled into the Storm codebase.

Descriptions of the configuration parameters can be found at:

https://storm.apache.org/apidocs/backtype/storm/Config.html

The `storm.yaml` File

Default configuration settings are modified in the per-installation `storm.yaml` file.

In HDP 2.2, the default location is `/etc/storm/conf/storm.yaml`.

An Ambari installation makes all the mandatory updates to this file.

```
[root@sandbox conf]# more /usr/lib/storm/conf/storm.yaml
topology.enable.message.timeouts: true
topology.tuple.serializer: 'backtype.storm.serialization.types.ListDelegateSerializer'
topology.workers: 1
drpc.worker.threads: 64
storm.zookeeper.servers: ['sandbox.hortonworks.com']
transactional.zookeeper.root: '/transactional'
topology.executor.send.buffer.size: 1024
drpc.childopts: '-Xmx200m'
nimbus.thrift.port: 6627
nimbus.cleanup.inbox.freq.secs: 600
storm.zookeeper.retry.intervalceiling.millis: 30000
storm.local.dir: '/hadoop/storm'
storm.messaging.netty.min_wait_ms: 100
topology.worker.childopts: null
storm.messaging.netty.max_retries: 30
nimbus.task.timeout.secs: 30
nimbus.thrift.max_buffer_size: 1048576
topology.trident.batch.emit.interval.millis: 500
topology.debug: false
topology.sleep.spout.wait.strategy.time.ms: 1
topology.receiver.buffer.size: 8
--More--(18%)
```

An example `storm.yaml` file

Mandatory `storm.yaml` Changes

If you do not use Ambari to install Storm, there are a few post-installation configuration changes that are mandatory to get a working cluster.

`storm.zookeeper.servers:"<IP_address>" - "<IP_address>" - "<IP_address>"`

- Set of IP addresses used by Nimbus and the Supervisors to reach the ZooKeeper servers

`storm.local.dir:"/hadoop/storm"`

- Local disk directory used by Nimbus and Supervisors to store a small amount of state information
- Create the directories and change the ownership to "storm" and set 755 permissions

`nimbus.host:"<Nimbus_IP_address>"`

- Used by the Supervisors to download topology JAR files and configurations

`supervisors.slots.ports:6700 - 6701 - 6702 - 6703`

- List of ports that the worker processes on the Supervisors will use to receive messages
- The number of ports listed determines the maximum number of per-Supervisor worker processes

Per-Topology Configuration Settings

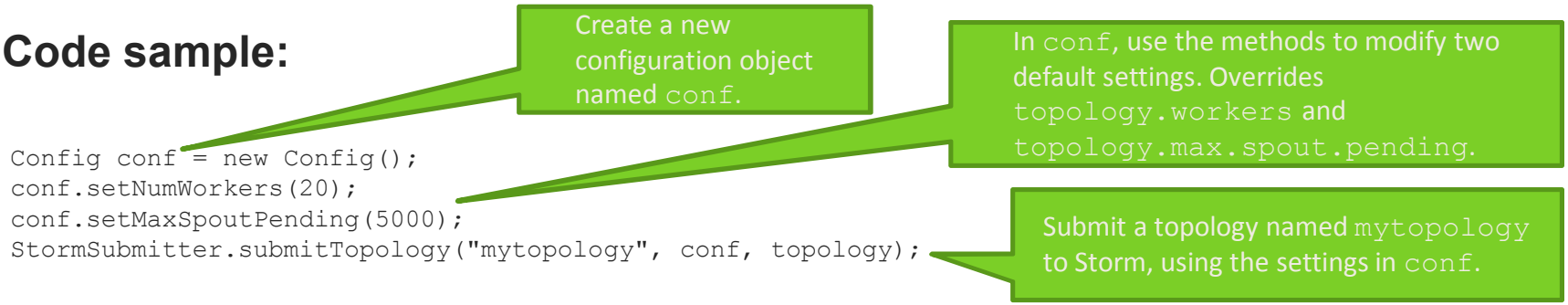
Default topology settings are configured by the `topology.*` settings in the `storm.yaml` file.

- For example, `topology.debug: false`

These settings can be overridden on a per-topology basis when submitting a topology using the `submitTopology` method in the `StormSubmitter` class.

- Only for those configuration settings prefixed by `topology`

Code sample:



Per-Spout and Per-Bolt Configuration Settings

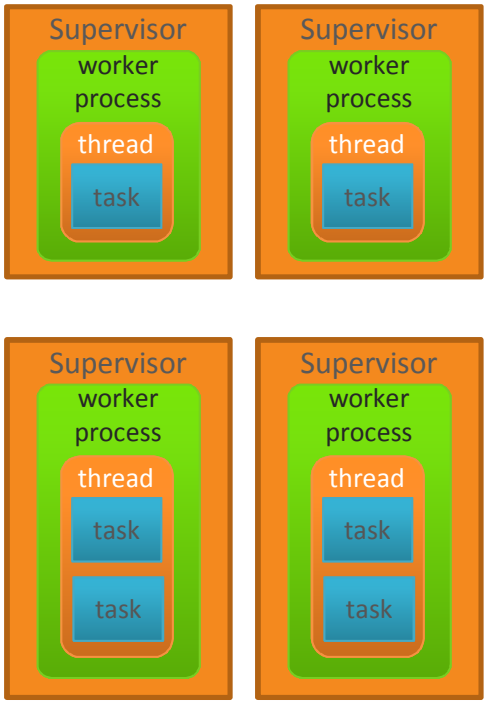
Spouts and bolts can be individually configured using the `setSpout` and `setBolt` methods in the `TopologyBuilder` class.

Code examples:

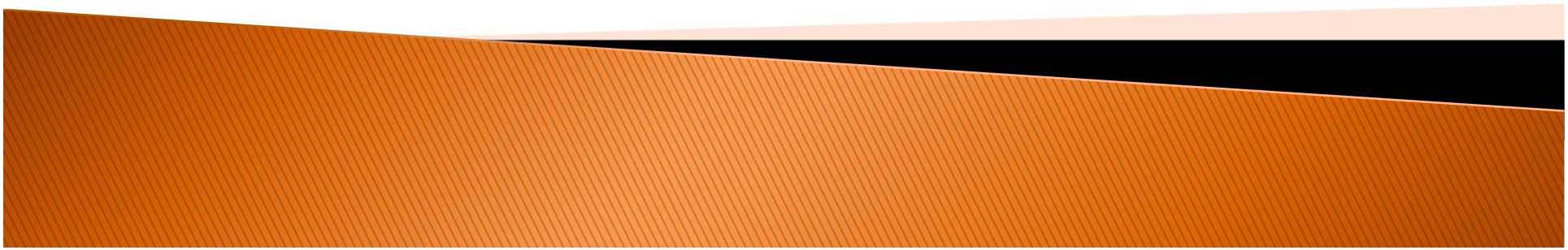
```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("blue-spout", new BlueSpout(), 2);
builder.setBolt("green-bolt", new GreenBolt(), 2)
    .setNumTasks(4) .shuffleGrouping("blue-spout");
```

Create a new spout named `blue-spout`, using the class `BlueSpout`, and modify the default configuration so that the spout uses only two executors (threads) and tasks.

Create a new bolt named `green-bolt`, using the class `GreenBolt`, and modify the default configuration so that the spout uses only two executors (threads) but four tasks.



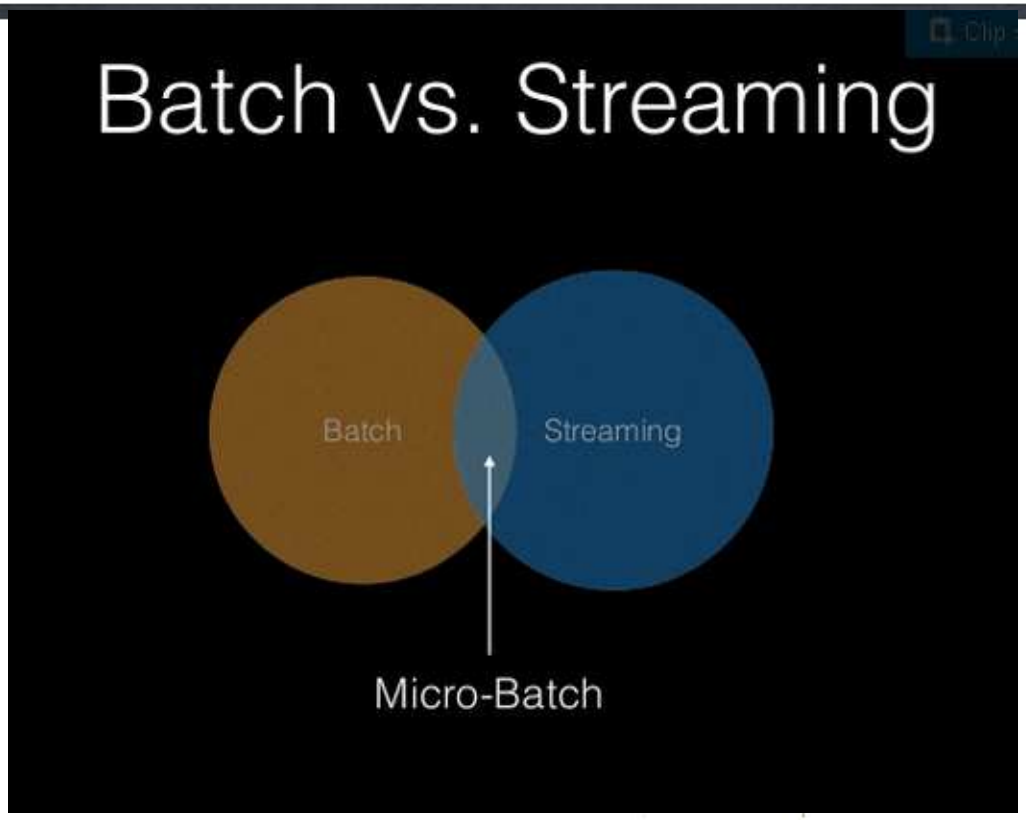
Apache Storm and Spark Streaming Compared



Streaming and batch processing are fundamentally different.

Batch vs. Streaming

- ***Storm*** is a stream processing framework that also does micro-batching (Trident).
- ***Spark*** is a batch processing framework that also does micro-batching (Spark Streaming).


Batch vs. Streaming
Batch
Streaming
Micro-Batch

Apache Storm: Two Streaming APIs

Core Storm (Spouts and Bolts)

- One at a Time
- Lower Latency
- Operates on Tuple Streams

Trident (Streams and Operations)

- Micro-Batch
- Higher Throughput
- Operates on Streams of Tuple Batches and Partitions

Language Options

Core Storm	Storm Trident	Spark Streaming
• Java • Clojure • Scala • Python • Ruby • others*	• Java • Clojure • Scala	• Java • Scala • Python

*Storm's Multi-Lang feature allows the use of virtually any programming language.

Reliability Models

	Core Storm	Storm Trident	Spark Streaming
At Most Once	Yes	Yes	No
At Least Once	Yes	Yes	No*
Exactly Once	No	Yes	Yes*

*In some node failure scenarios, Spark Streaming falls back to at-least-once processing or data loss.

Programing Model

	Core Storm	Storm Trident	Spark Streaming
Stream Primitive	Tuple	Tuple, Tuple Batch, Partition	DStream
Stream Source	Spouts	Spouts, Trident Spouts	HDFS, Network
Computation/ Transformation	Bolts	Filters, Functions, Aggregations, Joins	Transformation, Window Operations
Stateful Operations	No (roll your own)	Yes	Yes
Output/ Persistence	Bolts	State, MapState	foreachRDD

Support

	Apache Storm	Spark	Spark Streaming
Hadoop Distro	Hortonworks, MapR	Cloudera, MapR, Hortonworks (preview)	Hortonworks, Cloudera, MapR
Resource Management	YARN, Mesos	YARN, Mesos	YARN*, Mesos
Provisioning/ Monitoring	Apache Ambari	Cloudera Manager	?

*With issues: http://spark-summit.org/wp-content/uploads/2014/07/Productionizing-a-247-Spark-Streaming-Service-on-YARN-Ooyala.pdf

Worker Failure: Spark Streaming

"So if a worker node fails, then the system can recompute the lost from the the left over copy of the input data. However, if the worker node where a network receiver was running fails, ***then a tiny bit of data may be lost***, that is, the data received by the system but not yet replicated to other node(s)."

Only HDFS-backed data sources are fully fault tolerant.

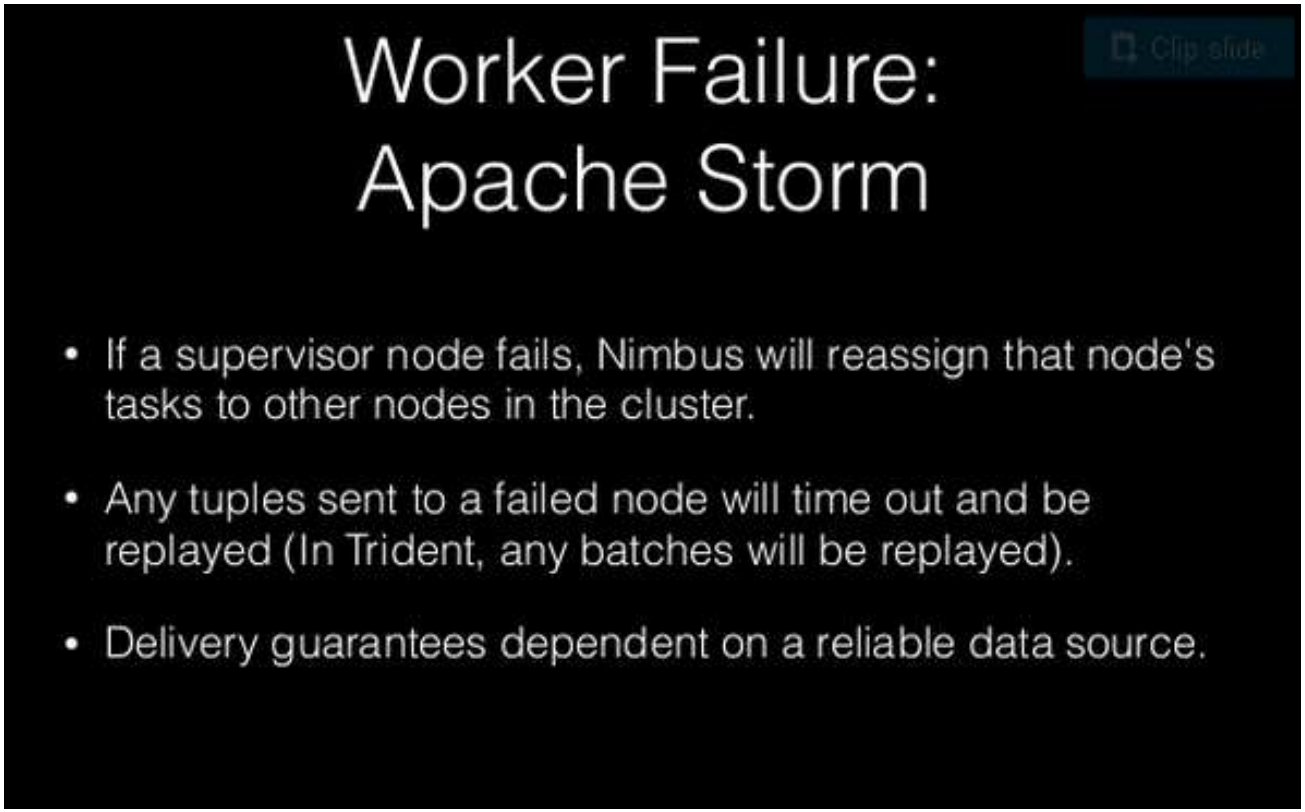
https://spark.apache.org/docs/latest/streaming-programming-guide.html#fault-tolerance-properties

Worker Failure: Spark Streaming

Solution?: Write Ahead Logs (SPARK-3129)

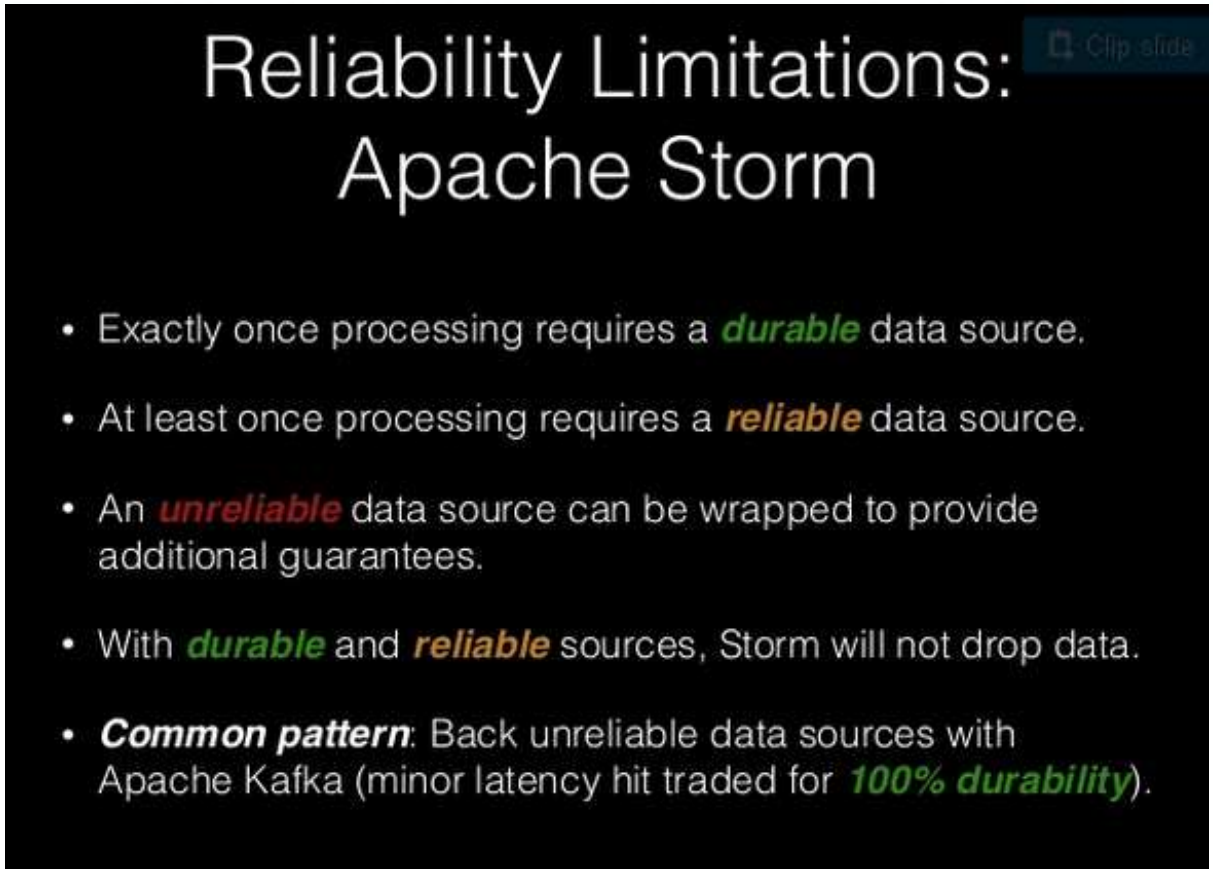
- Enabling WAL requires DFS (HDFS, S3) — no such requirement with Storm
- Incurs a performance penalty that adds to overall latency
- ***Full fault tolerance still requires a data source that can replay data (e.g. Kafka)***
- Architectural band aid?

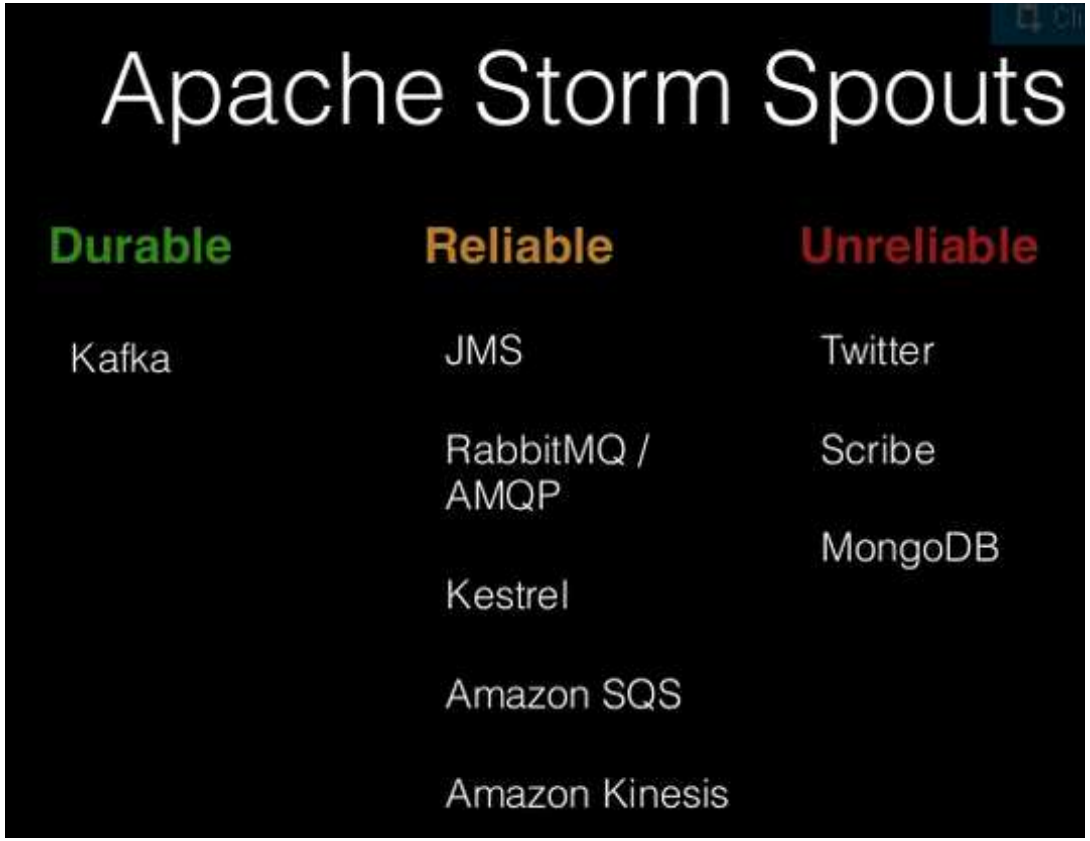
https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html


Worker Failure:
Apache Storm
• If a supervisor node fails, Nimbus will reassign that node's tasks to other nodes in the cluster.
• Any tuples sent to a failed node will time out and be replayed (In Trident, any batches will be replayed).
• Delivery guarantees dependent on a reliable data source.

Data Source Reliability

- A data source is considered ***unreliable*** if there is no means to replay a previously-received message.
- A data source is considered ***reliable*** if it can somehow replay a message if processing fails at any point.
- A data source is considered ***durable*** if it can replay any message or set of messages given the necessary selection criteria.


Reliability Limitations:
Apache Storm
Exactly once processing requires a durable data source.
At least once processing requires a reliable data source.
An unreliable data source can be wrapped to provide additional guarantees.
With durable and reliable sources, Storm will not drop data.
Common pattern: Back unreliable data sources with Apache Kafka (minor latency hit traded for 100% durability).


Apache Storm Spouts
Durable
Kafka
Reliable
JMS
RabbitMQ / AMQP
Kestrel
Amazon SQS
Amazon Kinesis
Unreliable
Twitter
Scribe
MongoDB

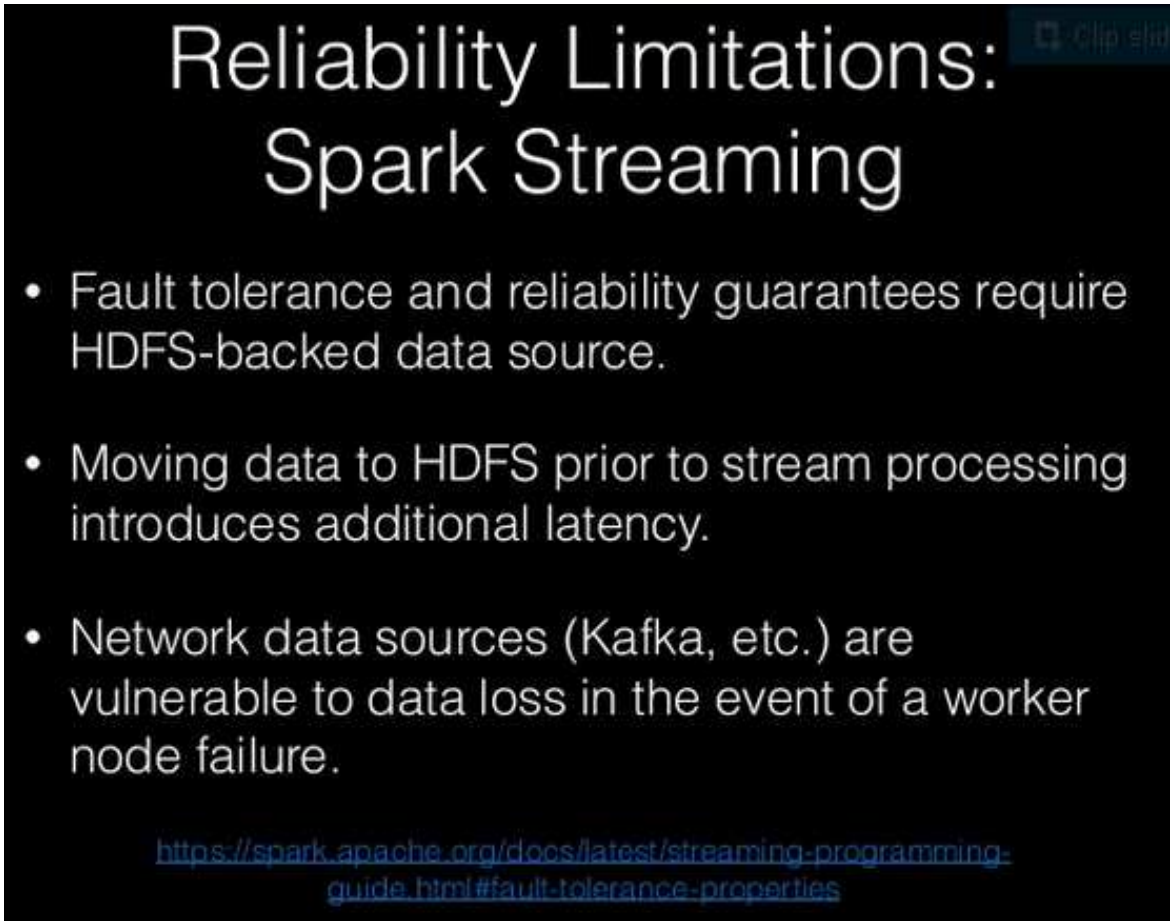
Apache Storm + Kafka

Apache Kafka is an ideal source for Storm topologies. It provides everything necessary for:

- At most once processing
- At least once processing
- Exactly once processing

Apache Storm includes Kafka spout implementations for all levels of reliability.

Kafka Supports a wide variety of languages and integration points for both producers and consumers.


Reliability Limitations:
Spark Streaming
• Fault tolerance and reliability guarantees require HDFS-backed data source.
• Moving data to HDFS prior to stream processing introduces additional latency.
• Network data sources (Kafka, etc.) are vulnerable to data loss in the event of a worker node failure.
https://spark.apache.org/docs/latest/streaming-programming-guide.html#fault-tolerance-properties

Performance

"The main reason cited by Tathagata for Spark's performance gain over Storm is the aggregation of small records that occurs through the mechanics of RDDs."

In other words: Micro-Batching

http://www.cs.duke.edu/~kmoses/cps516/dstream.html

Performance

Storm capped at 10k msgs/sec/node?
Spark Streaming 40x faster than Storm?

System	Performance
Storm (Twitter)	10,000 records/s/node
Spark Streaming	400,000 records/s/node
Apache S4	7,000 records/s/node
Other Commericial Systems	100,000 records/s/node

Others may disagree…

http://www.cs.duke.edu/~kmoses/cps516/dstream.html