WALLENBERG AI, AUTONOMOUS SYSTEMS AND SOFTWARE PROGRAM

CLOUD COMPUTING

Distributed Processing: Apache Spark and Apache Storm

PAUL TOWNEND
ASSOCIATE PROFESSOR, UMEÅ

CHALMERS UNIVERSITY OF TECHNOLOGY
KTH VETENSKAP OCH KONST
LINKÖPINGS UNIVERSITET
LUNDS UNIVERSITET
UMEÅ UNIVERSITET

Apache Spark
(in-memory processing)

WHAT IS SPARK?

Apache Spark is a general-purpose data processing engine.

Faster **batch** processing

Apps requiring **interactive** query processing

Processing of **streaming** data

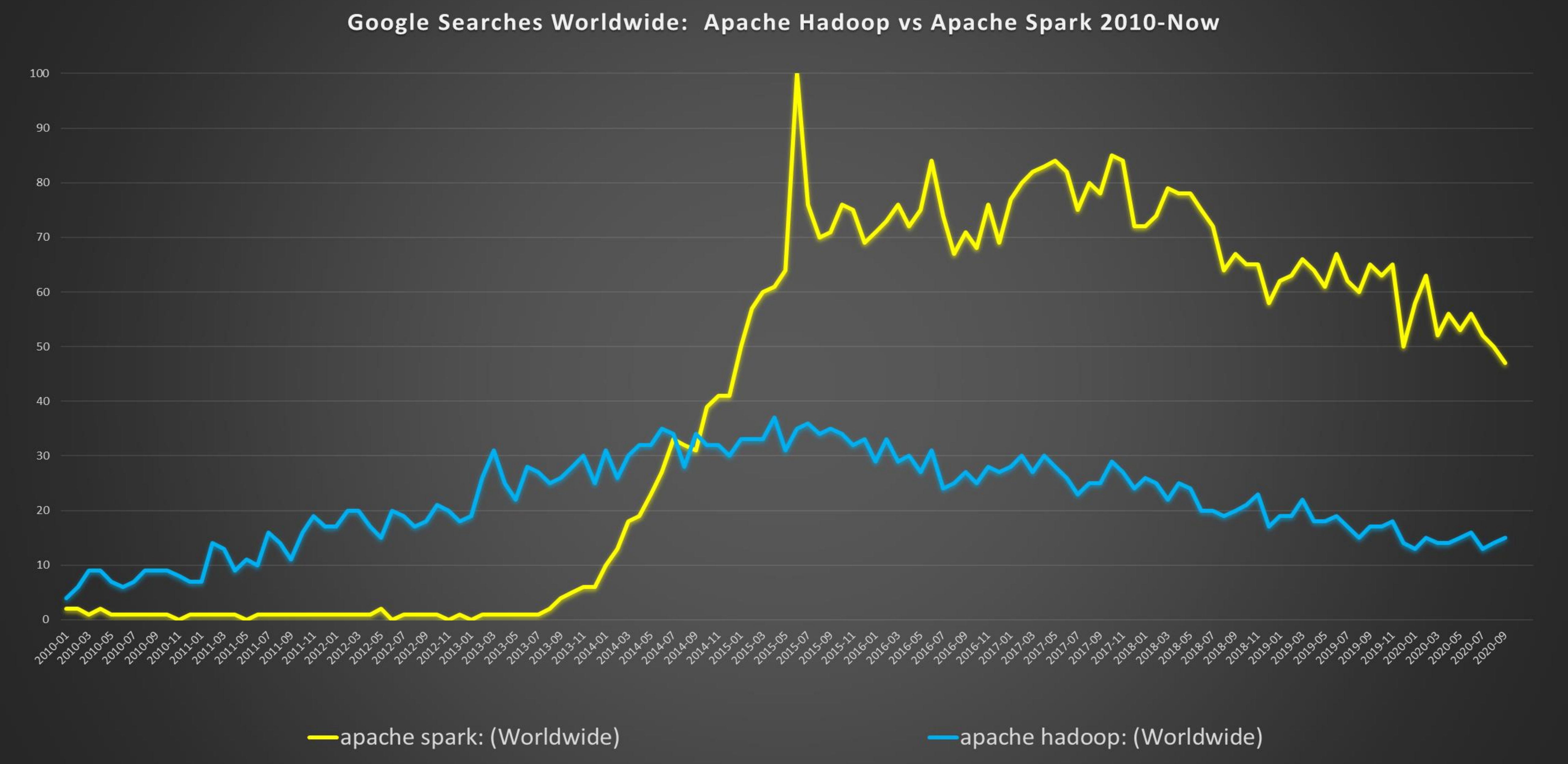
Systems that require **iterative** algorithms

Features of Spark:

In Memory computation engine

Almost 10x faster than Hadoop MapReduce using computations with Disk IO

Almost 100x faster than Hadoop MapReduce with in-memory computations



Google Searches Worldwide: Apache Hadoop vs Apache Spark 2010-Now
100
90
80
70
60
50
40
30
20
10
0
apache spark: (Worldwide)
apache hadoop: (Worldwide)

SPARK ARCHITECTURE

Apache Spark doesn't provide any storage (like HDFS) or Resource Management capabilities.

It is just a unified framework for processing large amount of data near to real time.

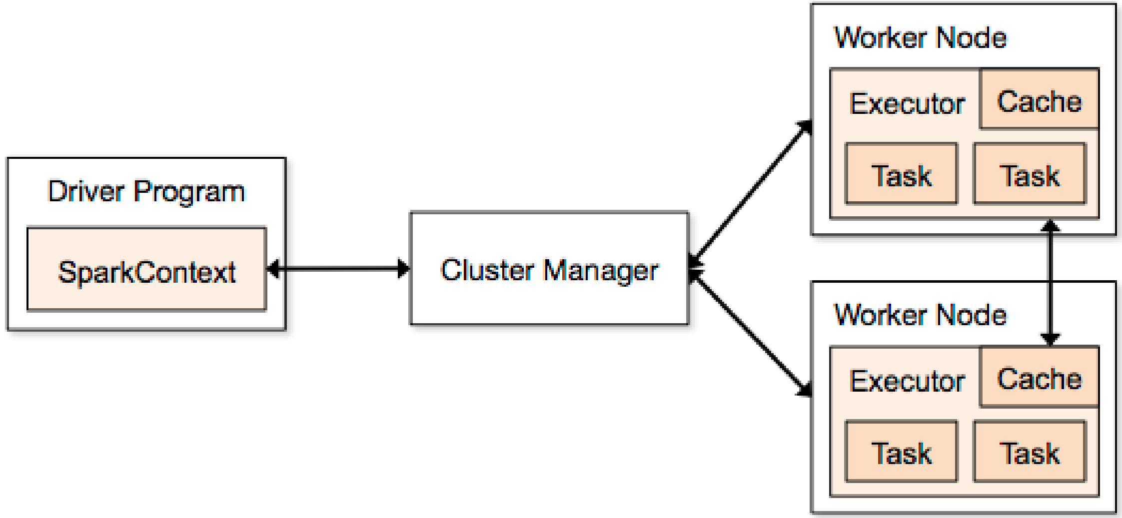
Ecosystems Layer	Contains libraries operating on top of the Spark Core.
Core Layer	The generalized layer of the framework. It defines all basic functions. All other functionalities and extensions are built on top of this.
Resource Management layer	Manages own resources in standalone mode (single node cluster setup). For distributed cluster mode, can be integrated with resource management modules like **YARN**

SPARK ARCHITECTURE

Spark applications run as independent sets of processes on a cluster, coordinated by the **SparkContext** object (aka Driver Program)

SparkContext sends application code (JAR or Python files) and tasks to run to the Executors.

Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors, and storage usage

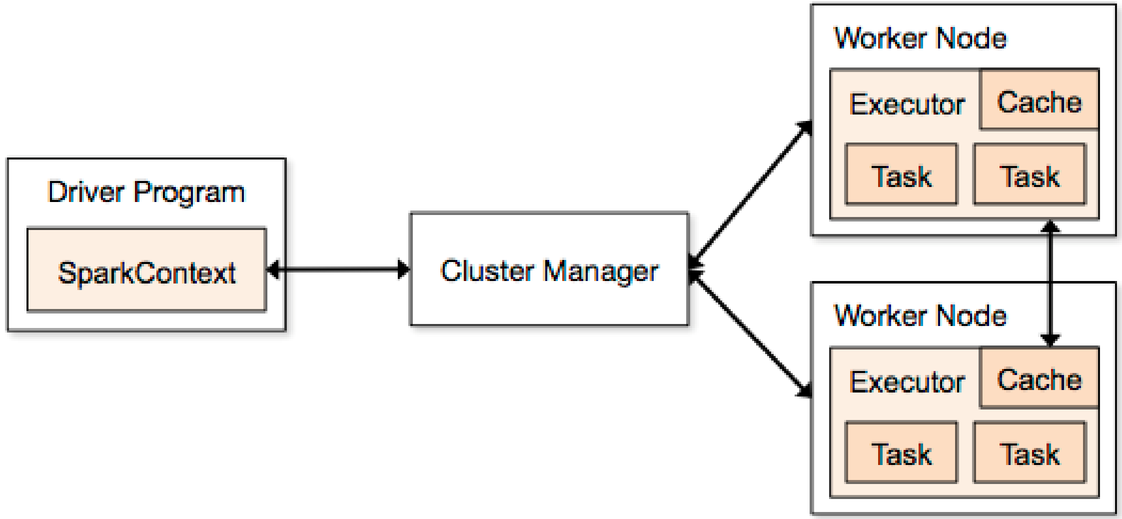


SPARK ARCHITECTURE (2)

Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

This isolates applications from each other - each driver schedules its own tasks, and tasks from different applications run in different JVMs.

However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing to external storage



SPARK LANGUAGE SUPPORT

Spark provides high-level APIs in Java, Scala, and Python

It provides an optimised engine that supports: general execution graphs, high-level tools for structured data processing, etc.

Spark API

Scala

```scala
val spark = new SparkContext()

val lines    = spark.textFile(“hdfs://docs/”)   // RDD[String]
val nonEmpty = lines.filter(l => l.nonEmpty())  // RDD[String]

val count = nonEmpty.count
```

Java 8

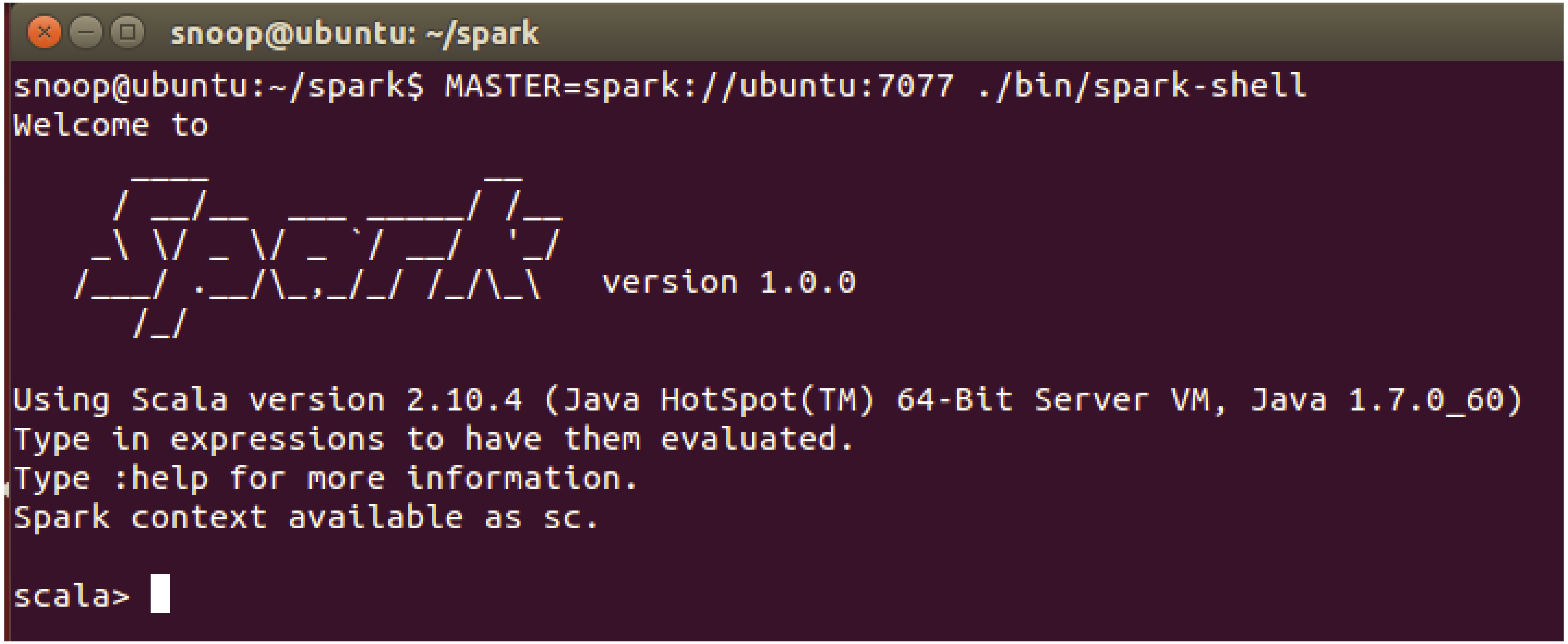
```java
SparkContext spark = new SparkContext();

JavaRDD<String> lines    = spark.textFile(“hdfs://docs/”)
JavaRDD<String> nonEmpty = lines.filter(l -> l.length() > 0);

long count = nonEmpty.count();
```

Python

```python
spark = SparkContext()

lines = spark.textFile(“hdfs://docs/”)
nonEmpty = lines.filter(lambda line: len(line) > 0)

count = nonEmpty.count()
```

SPARK INTERACTIVE SHELL

Another important aspect is the interactive shell (REPL). Using REPL, you can test the outcome of each line of code without first needing to code and execute the entire job.



SPARK ARCHITECTURE (4)

Resource Management

- Standalone
- YARN
- Kubernetes

Spark Ecosystem

- Spark SQL
- Spark Streaming
- BlinkDB
- Spark ML
- GraphX
- Tachyon

Spark Core

- Spark DataFrame API
- Java
- Scala
- Python
- R
- Spark Core

SPARK CORE

The **Spark Core** is the heart of spark. It deals with:

memory management and fault recovery

scheduling, distributing and monitoring jobs on a cluster

interacting with storage systems

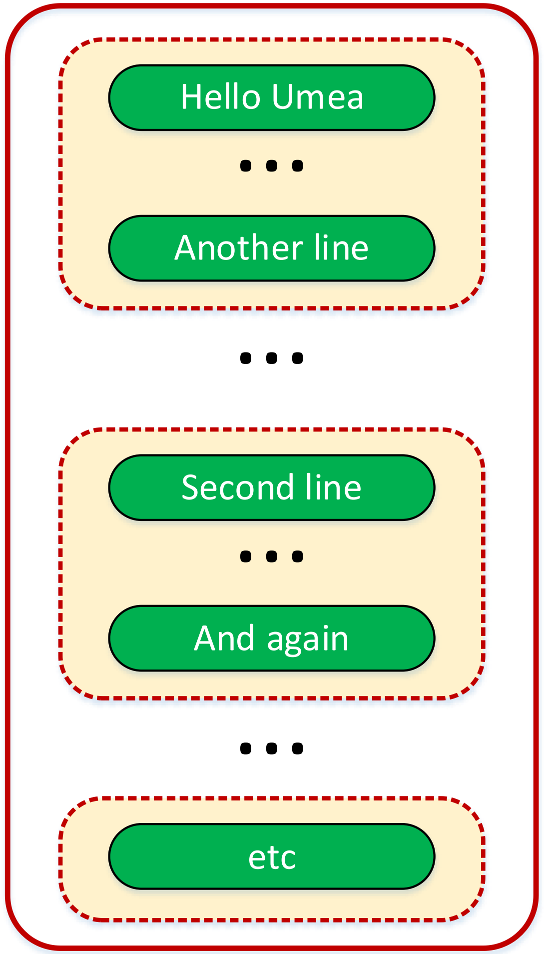
It also implements the key concept of **Resilient Distributed Databases** (RDDs)

An **immutable fault tolerant distributed collections of objects** that can be operated on in parallel.

An **RDD** can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program

An RDD is a representation of a dataset that is distributed throughout the cluster.

RESILIENT DISTRIBUTED DATABASES (RDD)



Immutable Collection of Objects

Partitioned and **Distributed**

Stored **in Memory**

Partitions **Recomputed on Failure**

Transformations

- **map**(func)
- **flatMap**(func)
- **filter**(func)
- **groupByKey**()
- **reduceByKey**(func)
- **mapValues**(func)
- ...

Actions

- **take**(N)
- **count**()
- **collect**()
- **reduce**(func)
- **takeOrdered**(N)
- **top**(N)
- ...

RESILIENT DISTRIBUTED DATABASES (2)

RDDs can be stored in memory (RAM) or on disk. **Most major performance gains come from holding them in memory**.

Current frameworks like MapReduce provide many abstractions for accessing a cluster's computational resources, **but lack abstractions for leveraging distributed memory**.

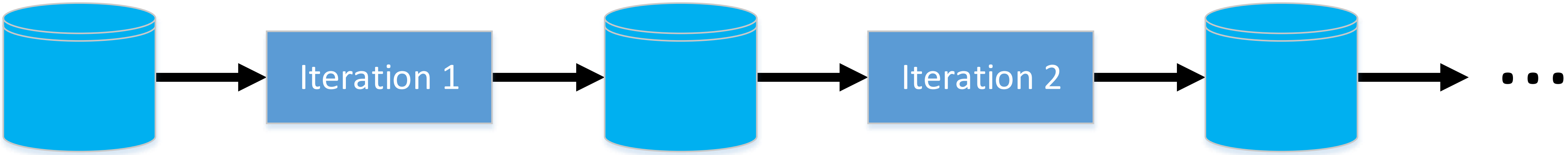
This is an important advantage of Spark: data reuse is common in many iterative M/L algorithms, such as **K-means clustering**.

Another example is when a user runs multiple ad-hoc queries on the same subset of data.

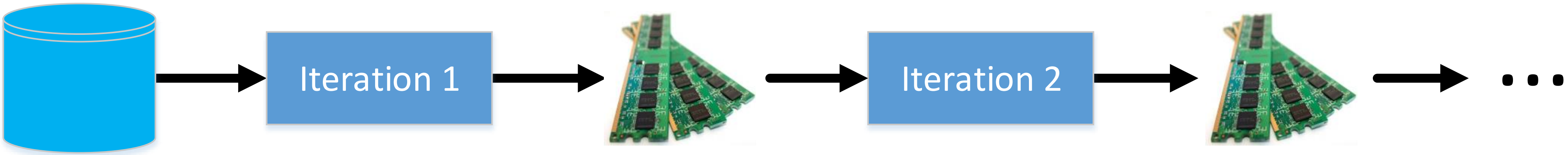
In Hadoop (and other frameworks) the only way to reuse data between different jobs is to write it to an external storage system, such as HDFS.

With in-memory RDDs, data can be processed faster. **The size of data that can be stored in distributed memory is limited only by cluster size.**

RESILIENT DISTRIBUTED DATABASES (3)

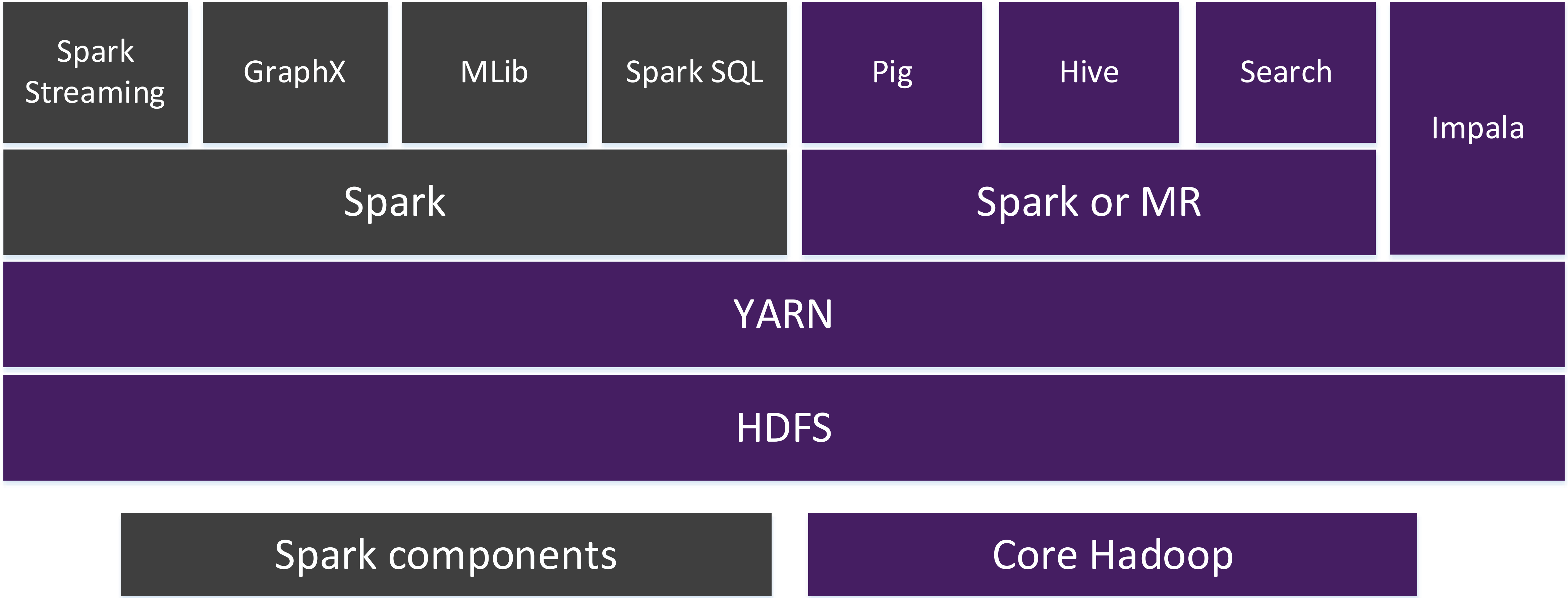


Existing solutions (MapReduce, Storm, etc.) – Slow, needs high I/O



RDD – Fast, in-memory

SPARK ON HADOOP

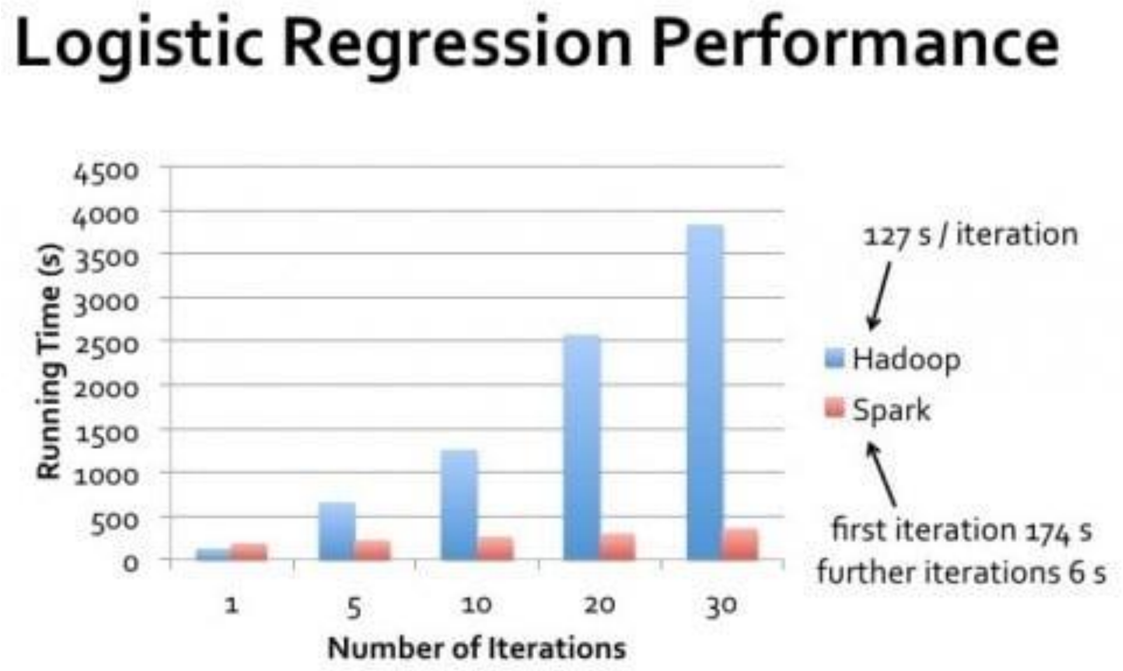


SPARK BENCHMARKS

Daytona Gray Sort 100TB Benchmark

	Data Size	Time	Nodes	Cores
Hadoop MR	**102.5 TB**	72 min	2100	50400 physical
Apache Spark	**100 TB**	23 min	206	6592 virtualised

(3x faster using 10x fewer machines)



SPARK: KEY POINTS

Handles batch, interactive, and real-time jobs in a single framework

Has native integration with Java, Python, and Scala

Allows for programming at a higher level of abstraction

Map/Reduce is just one of its supported constructs.

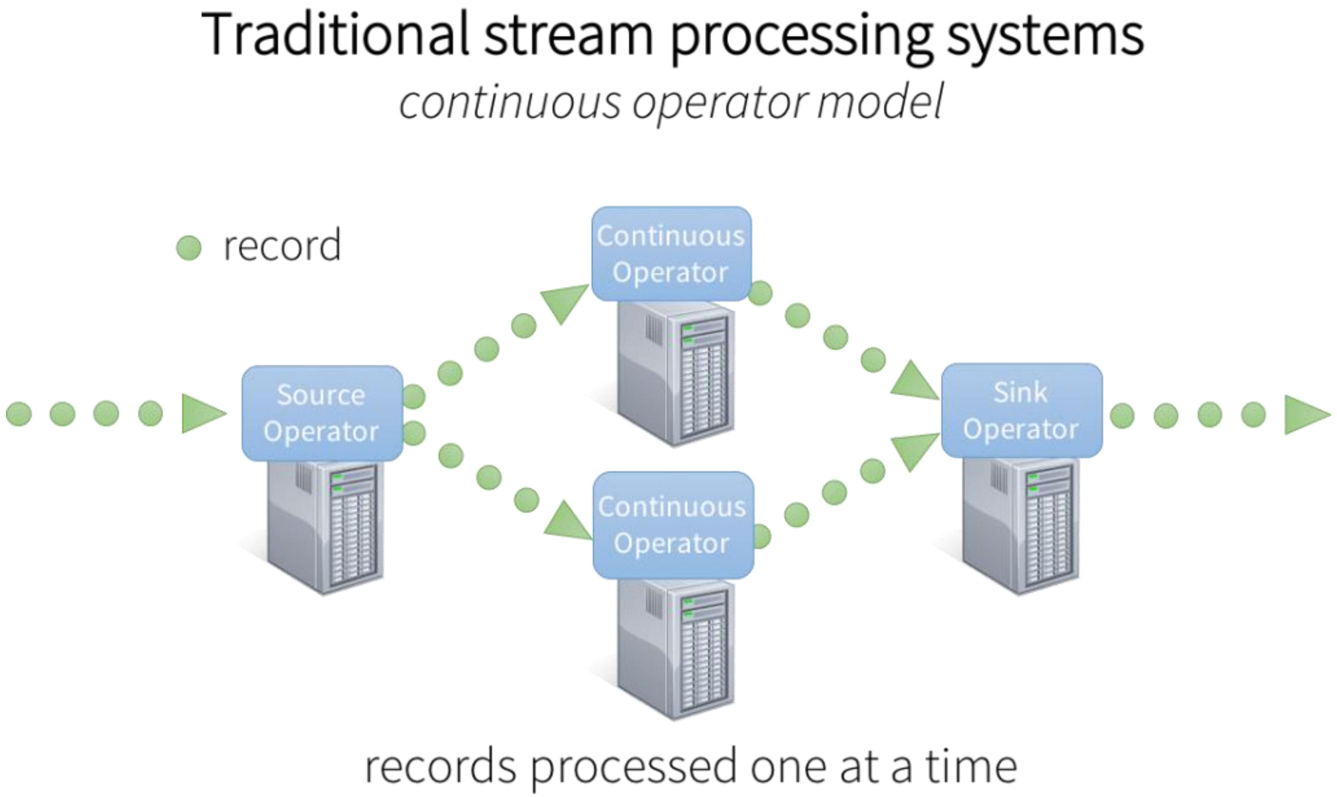
Performs in-memory operations for big performance gains

Apache Storm (continuous operator model)

CONTINUOUS OPERATOR MODELS

One of the major drawbacks of MapReduce is that it is designed for batch-processing jobs.

It is often inappropriate for dealing with **high velocity** data that requires reliable real-time processing capabilities.



APACHE STORM

A **distributed real-time computation system** for processing large volumes of high-velocity data

It is extremely fast, and can process over **one million records per second per node.**

Storm aims to make it easy to **reliably** process **unbounded** streams of data

STORM STREAMS

The core abstraction in Storm is the **Stream**.

A stream is data in the form of an **unbounded sequence of tuples**.

Storm provides **primitives** for transforming a stream into a new stream in a distributed and reliable way

Streams are defined with a schema that names the fields in the tuple. Every stream has an ID.

The two basic Storm primitives are **Spouts** and **Bolts**.

Spout	A spout is a **source of streams**. It may read tuples from a queue, or connect to an API (like Twitter), etc. Can be **reliable** or **unreliable** (i.e. can replay a tuple or not)
Bolt	A **bolt consumes any number of input streams**, does some processing, and possibly emits new streams.
	Complex transformations may require multiple bolts to create.
	Bolts can do anything, including **run functions**, **filter tuples**, **streaming aggregations**, **streaming joins**, **talk to databases**, etc.

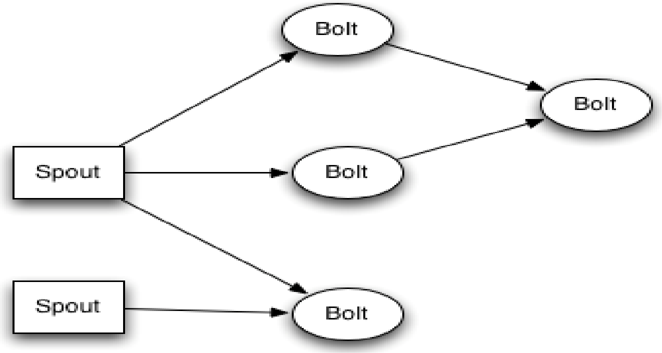
STORM TOPOLOGIES

A network of spouts and bolts are packaged into a **topology**.
This is the top level abstraction that is submitted to Storm clusters for execution.

A topology is a **graph of stream transformations** where each node is a **spout** or **bolt**

Edges in the graph indicate which **bolts** subscribe to which **streams**.

When a **spout** or **bolt** emits a **tuple** to a **stream**, it sends it to every **bolt** that subscribes.

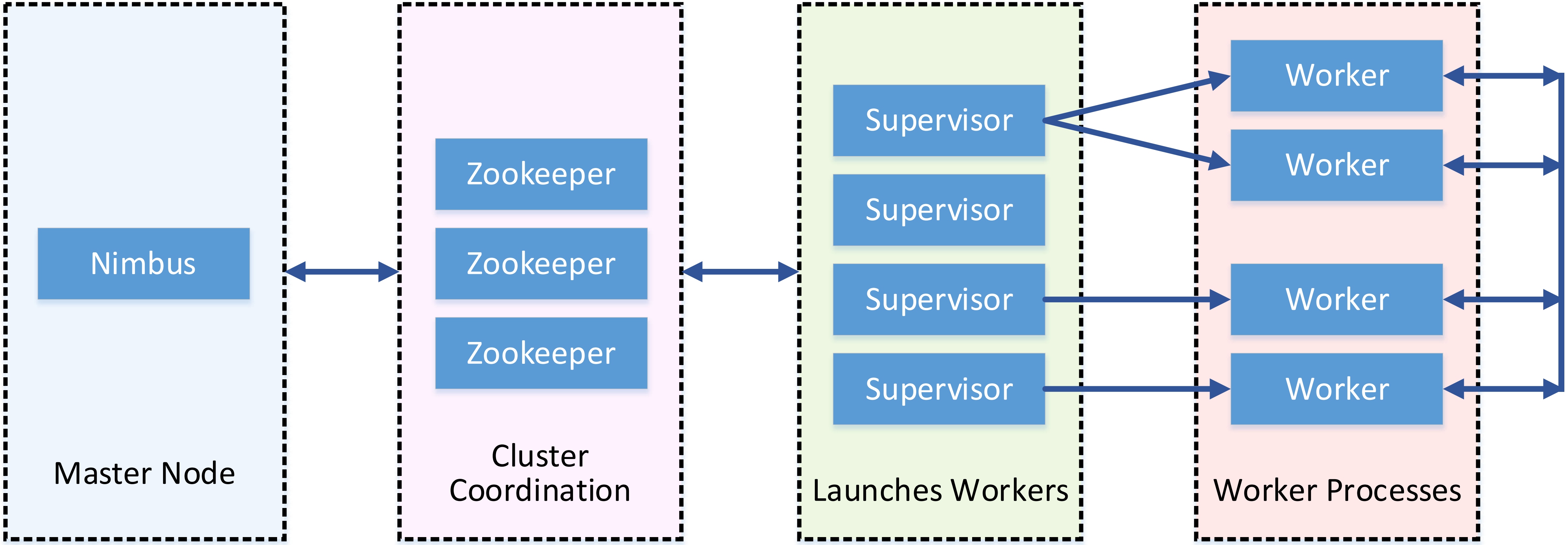


On MapReduce we run **jobs** - on Storm we run **Topologies**

A Storm topology processes messages indefinitely.

Each node in a topology executes in parallel. You specify the parallelism you want for each node, and Storm will spawn that number of threads across the cluster to do the execution

STORM ARCHITECTURE



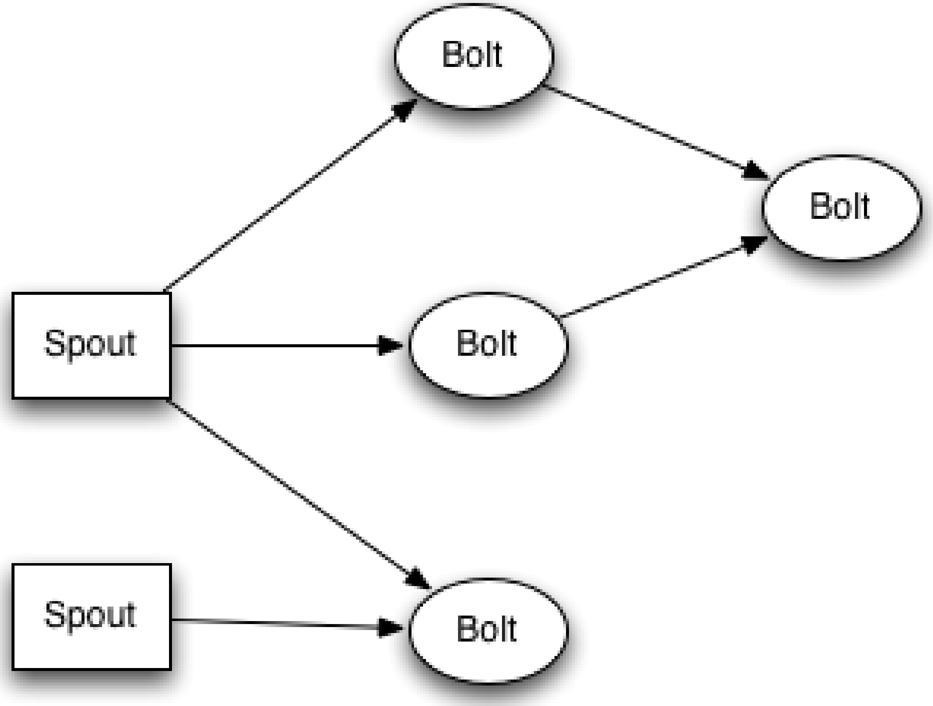
All coordination between Nimbus and the Supervisors is done through an **Apache Zookeeper cluster.**

STORM DATA MODEL

Storm uses tuples as its data model.
A tuple is a named list of values (like Apache Pig, etc.)

Storm supports all **primitive types**, **Strings**, and **byte arrays** as tuple field values. To use an object of another type in a tuple, a **serialiser** needs to be implemented.

Every node in a topology must declare the output fields for the tuples it emits.



SIMPLE SPOUT EXAMPLE

```
public class NumberSpout implements IRichSpout
{
   private SpoutOutputCollector collector;
   private TopologyContext context;

   @Override
   public void open(Map cfg, TopologyContext con, SpoutOutputCollector coll)
   {
          this.context = con;
          this.collector = coll;
   }

   @Override
   public void nextTuple()
   {
      String theOutput = “” + new Random().nextInt(50);
      this.collector.emit(new Values(theOutput));
   }

   @Override
   public void declareOutputFields(OutputFieldsDeclarer dec)
   {
          dec.declare(new Fields(“TheNumber"));
   }
}
```

So what does this Spout do?

Called at the start of the stream. Sets context and configuration information.

Called repeatedly. Can output a tuple or do nothing.

Declares the tuple the spout will emit

SIMPLE BOLT EXAMPLE

What does this Bolt do?

```
public static class ExclamationBolt extends BaseRichBolt
{
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector)
    {
         _collector = collector;
    }

    @Override
    public void execute(Tuple tuple)
    {
      _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
      _collector.ack(tuple);
    }

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

Allows bolt to emit tuples at any time.

Initialise component within worker

Bolt execution

Declares output fields

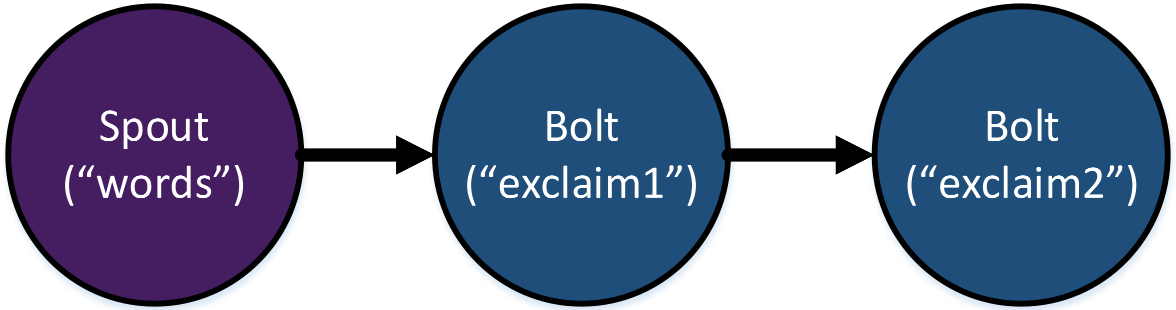
SIMPLE TOPOLOGY

```java
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("words", new TestWordSpout(), 10);

builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");

builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```

User-specified ID

Object containing processing logic

Amount of parallelism (optional)



10 parallel threads

3 parallel threads

2 parallel threads

Assume ExclamationBolt() appends “!!!” to a tuple

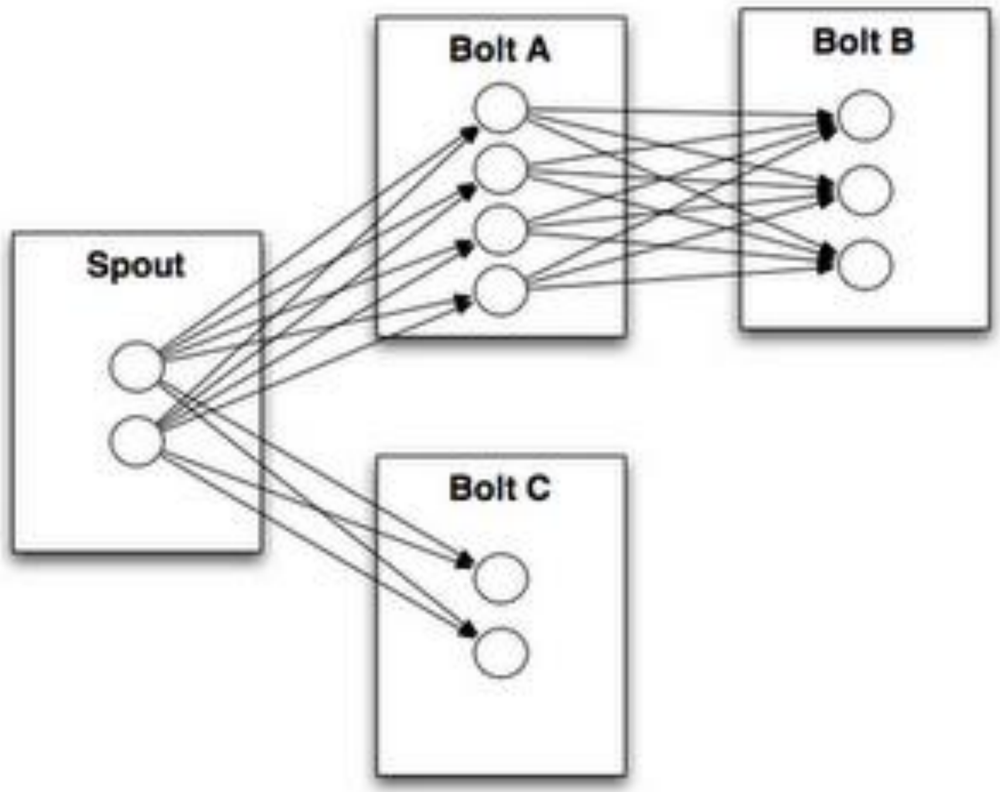
What is the output of this topology if the Spout stream is [“Hello”] and [“World”]

Hello!!!!!!
World!!!!!!

STREAM GROUPINGS

A stream grouping tells a topology how to send tuples between two components

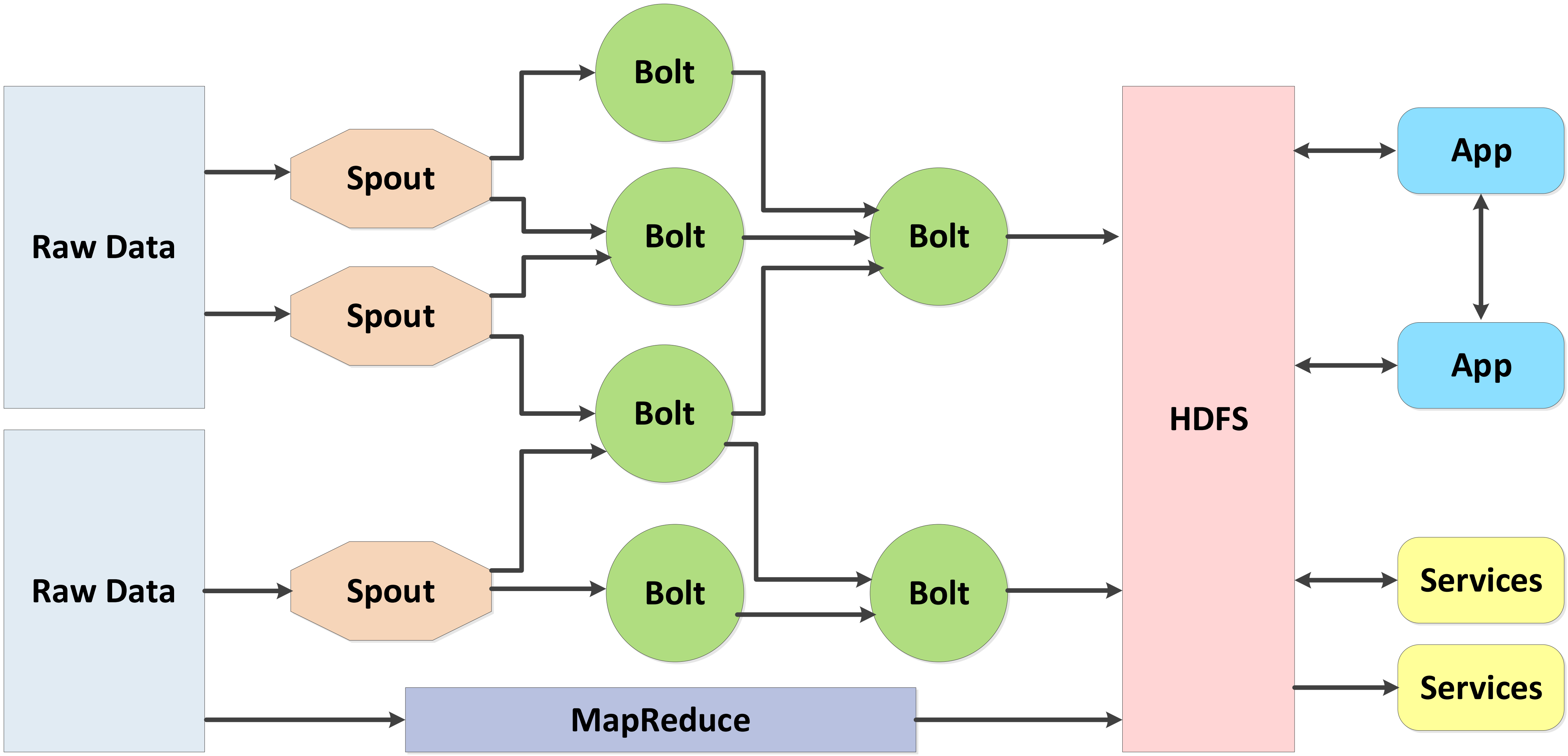
When a task for Bolt A emits a tuple to Bolt B, which task should it send the tuple to?



STREAM GROUPINGS (2)

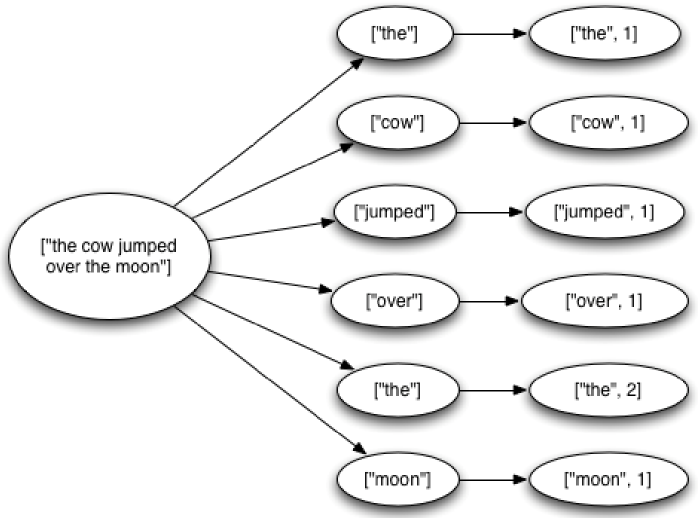
Shuffle groupings	Send tuples to a random tasks.
Fields groupings	Group a stream by a subset of its fields, allowing equal values for that subset of fields to go to the same task.
Partial Key groupings	Also groups by fields, but load balance between two bolts
All groupings.	Replicate the stream to all of the Bolt's tasks.
Global groupings	Send the entire stream to a single Bolt task with the lowest ID
Direct groupings	Allow producer of tuple to decide which task receives the tuple.

STORM IN HADOOP



STREAM RELIABILITY

Storm guarantees that every spout tuple will be fully processed by the topology.



Storm tracks the tree of tuples triggered by every spout, and determines when it has been **fully processed**.

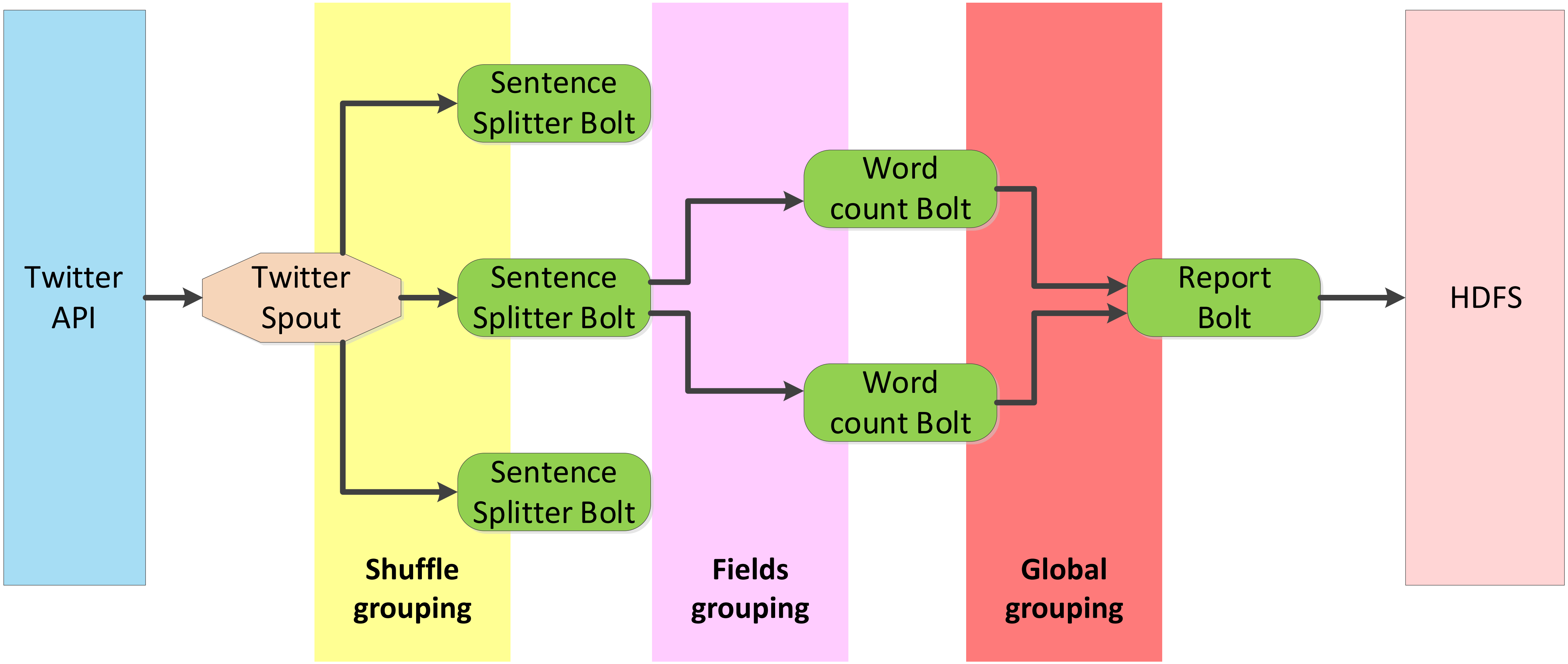
Every topology has a **message timeout** associated with it.

The timeout can be configured on a **topology-specific basis**.

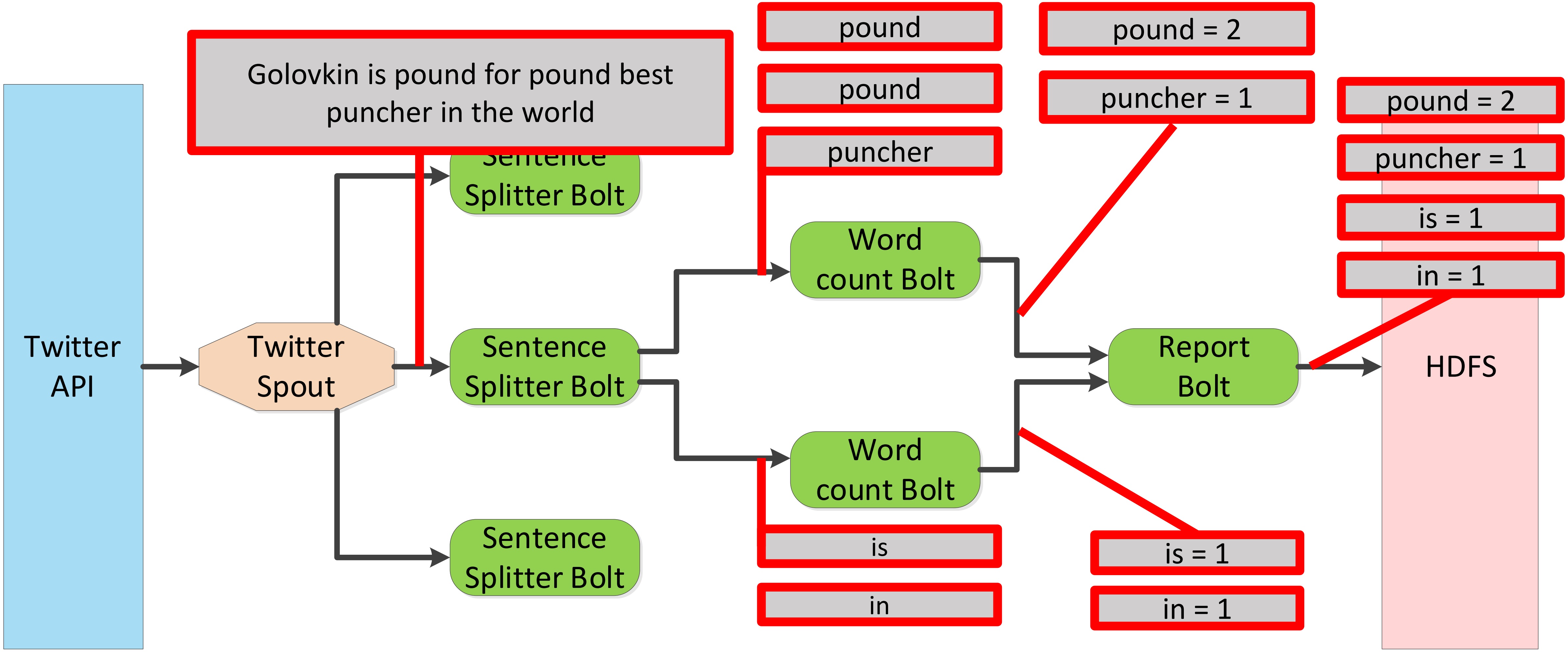
If a spout tuple is not completed within this timeout, Storm fails tuple and replays it later.

Bolts use **emit** method to inform they produced a new tuple, and **ack** method to declare they have finished

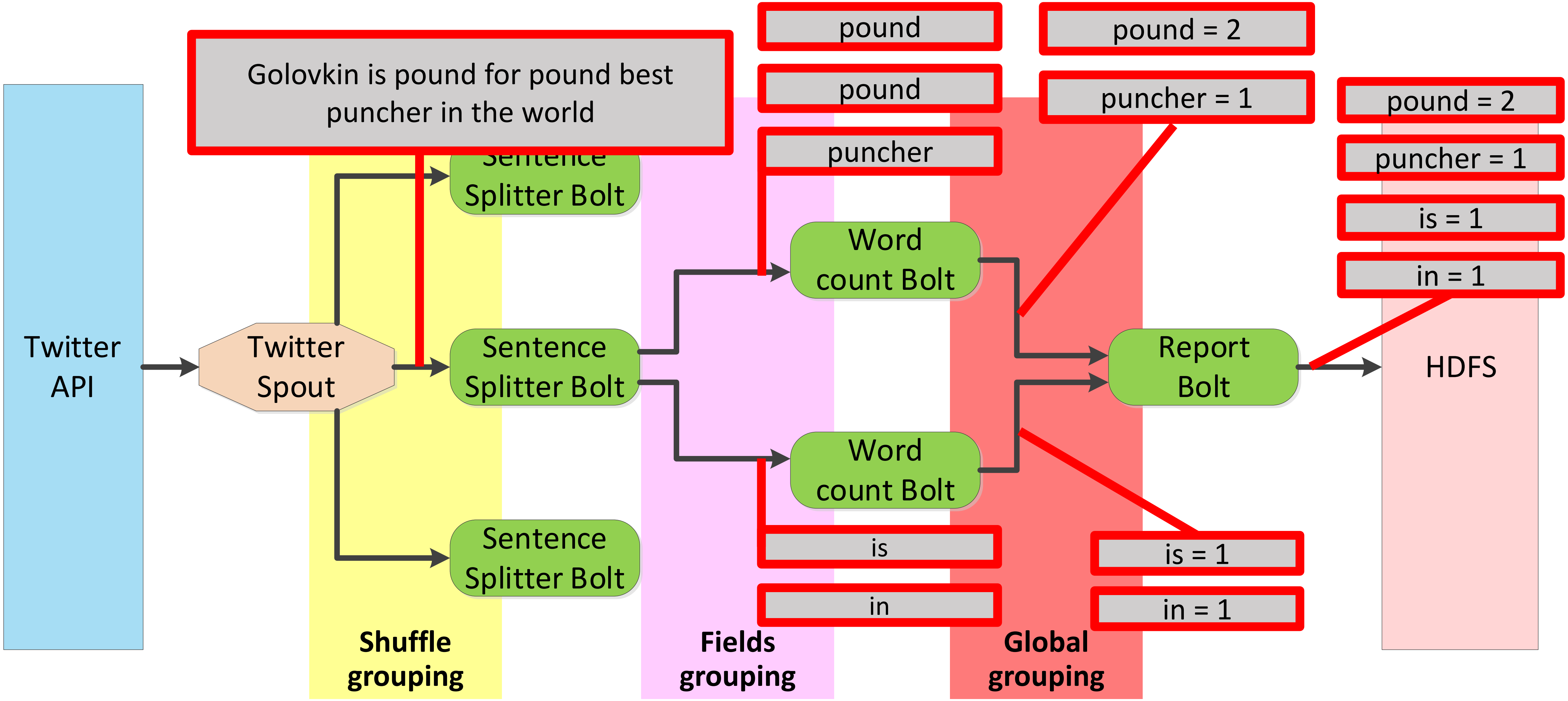
WORD COUNT EXAMPLE



WORD COUNT EXAMPLE (2)



WORD COUNT EXAMPLE (2)



WHO USES STORM?

STORM

Twitter

Yahoo

Spotify

Alibaba

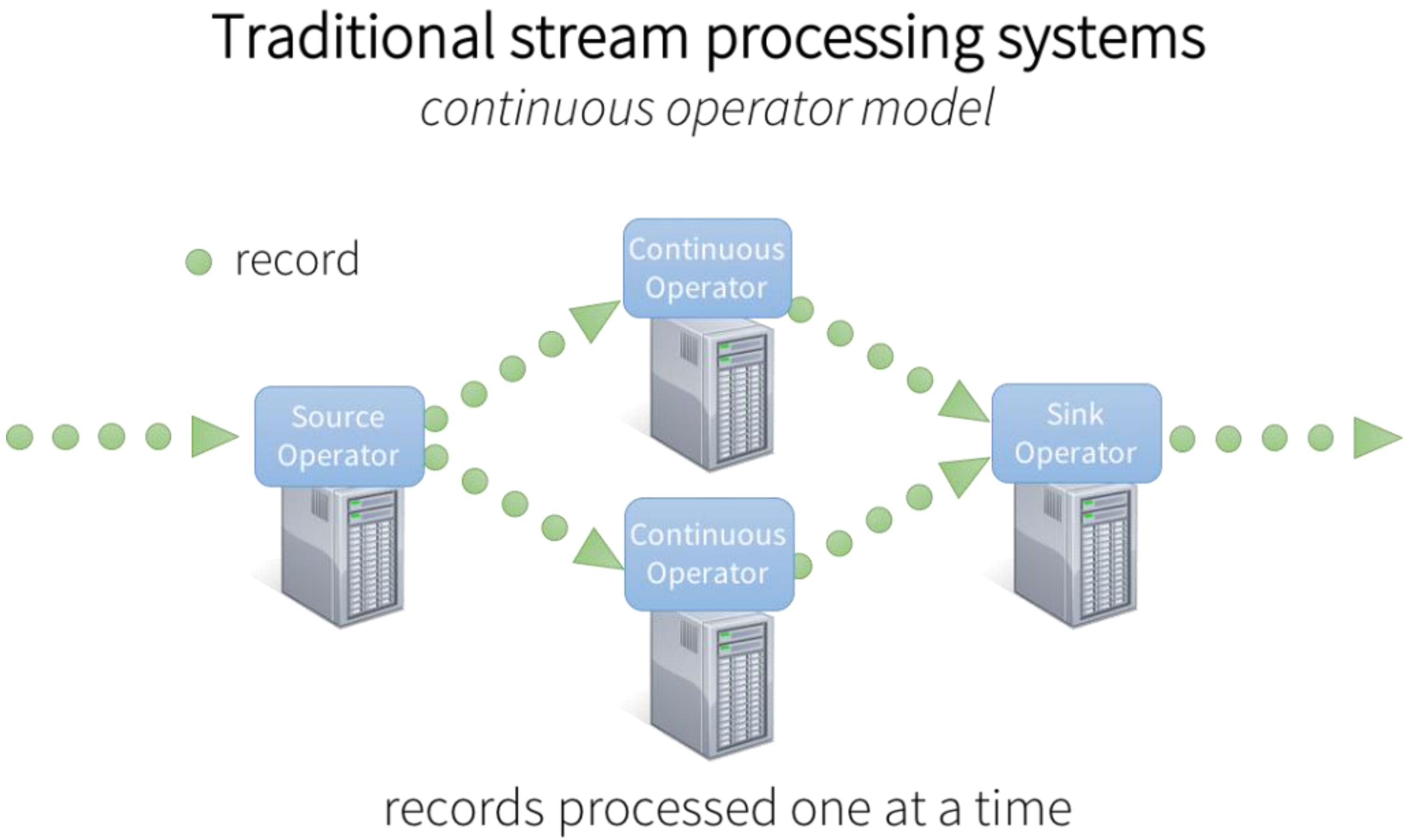
Cisco

Flickr

etc…

Issues with COMs?

ISSUES WITH THE CONTINUOUS OPERATOR MODEL (1)



Elegant solution, but as systems grow and the complexity of big data analytics increases, the model starts to struggle.

ISSUES WITH THE CONTINUOUS OPERATOR MODEL (2)

Failures and straggling tasks	With greater scale, there is a higher likelihood of a cluster node failing or unpredictably slowing down (i.e. stragglers).
	The system must be able to automatically recover from failures and stragglers to provide results in real time.
	Static allocation of continuous operators to worker nodes makes it hard for traditional systems to recover quickly from faults and stragglers.
Load Balancing	Uneven allocation of the processing load between the workers can cause bottlenecks in a continuous operator system.
	More likely to occur in large clusters and dynamically varying workloads.
	The system needs to be able to dynamically adapt the resource allocation based on the workload.

ISSUES WITH THE CONTINUOUS OPERATOR MODEL (3)

Unification of streaming, batch and interactive workloads	In many use cases, it is attractive to query streaming data interactively, or to combine it with static datasets (e.g. pre-computed models).
	This is hard in continuous operator systems as they are not designed to the dynamically introduce new operators for ad-hoc queries.
	This requires a single engine that can combine batch, streaming and interactive queries.
Advanced analytics (SQL, ML, etc)	Complex workloads require continuously learning and updating data models, or even querying “latest” view of streaming data with SQL.
	Having a common abstraction across these analytic tasks makes the developer’s job much easier

SPARK STREAMING

To address these issues, Spark Streaming uses an architecture called **discretized streams** that directly leverages the libraries and fault-tolerance of the Spark engine.

Instead of reading a single data record at a time, Spark Streaming **receivers** (**Spouts** in Apache **Storm** parlance) discretizes the streaming of data into tiny, sub-second micro-batches.

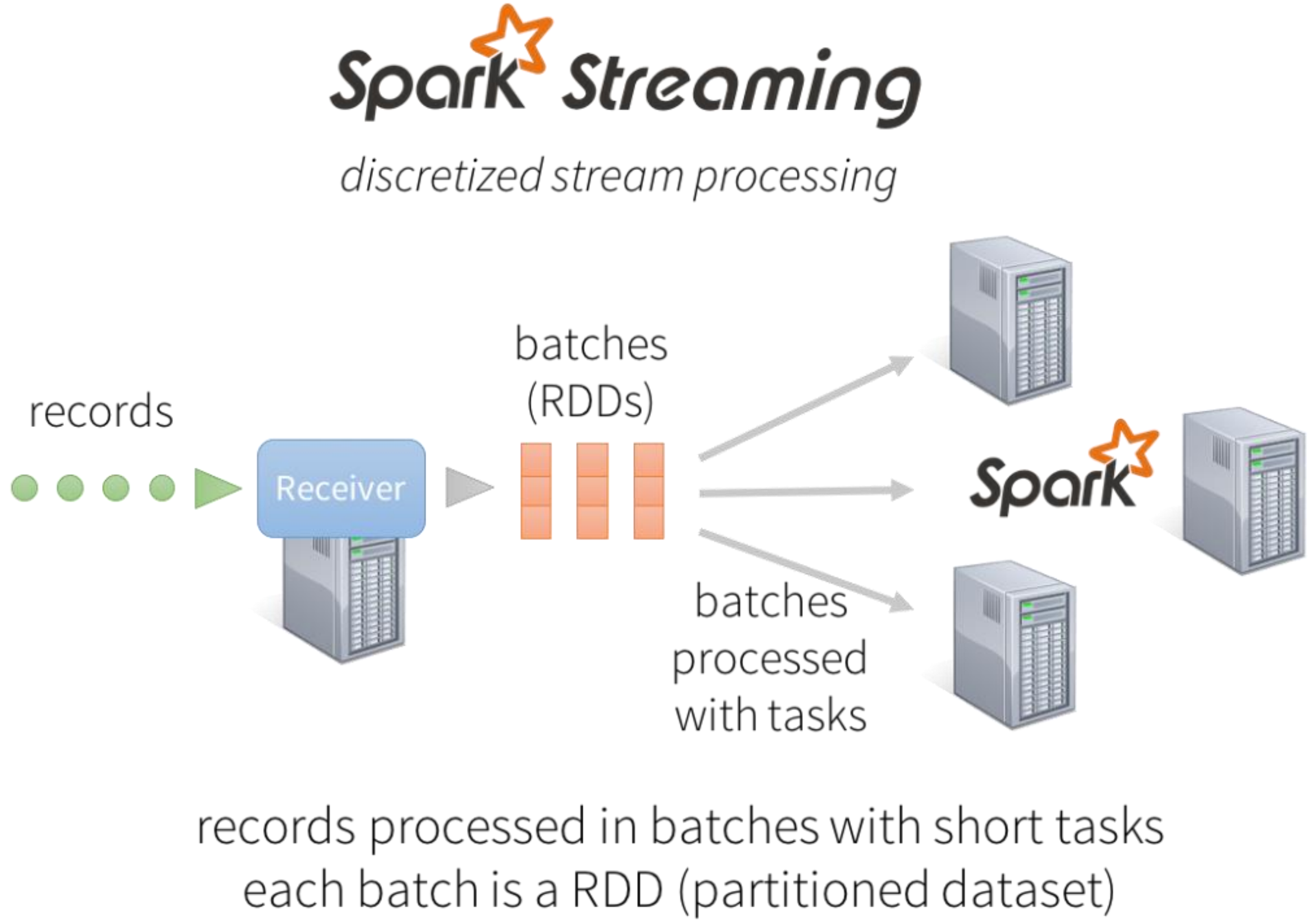
(**i.e. receivers accept data in parallel and buffer it in the memory of the Spark worker nodes**)

The Spark engine runs short tasks (tens of milliseconds) to process batches and output results to other systems

Unlike the COM, Spark tasks assigned dynamically to workers based on **locality of data and available resources**

This is to enable better **load balancing** and faster **fault recovery**.

DISCRETIZED STREAMS



Each of these batches of data is an **RDD**.

This allows the streaming data to be processed using any Spark code or library.

FINAL THOUGHTS

There is a lot to cover when talking about distributed processing!

MapReduce is very effective for specific applications. Typically batch processed.

Hadoop uses disk and is fairly slow but manages MapReduce well

Apache Spark performs computation in-memory so is much faster… but at what cost?

Continuous Operator Models (like Apache Storm) are a good way to handle high-velocity data

But Spark Streaming might be more advanced