IEMS5709 Spring 2016
Advanced Topics in Information Processing:
Big Data Processing Systems and Information Processing

Stream Processing and Apache Storm

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

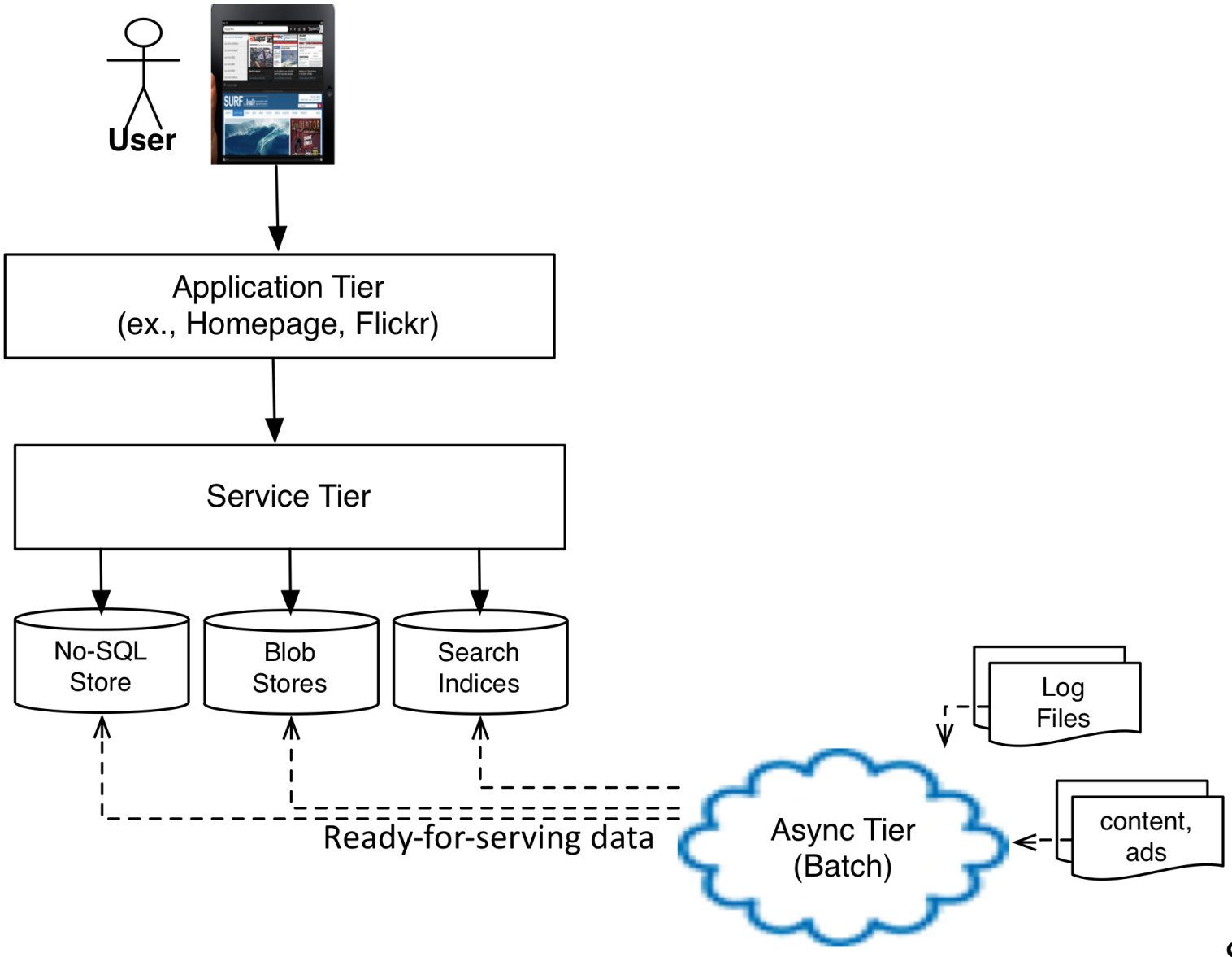
Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - Nathan Marz, “Storm – Distributed and Fault-tolerant real-time computation,” 2011, http://cloud.berkeley.edu/data/storm-berkeley.pdf
  - Krishna Gade of Twitter, “storm - Stream Processing @twitter,” June 2013.
  - Michael G. Noll of Verisign, “Apache Storm 0.9 basic training,” July, 2014, http://www.slideshare.net/miguno/apache-storm-09-basic-training-verisign
  - Guido Schmutz of Trivadis, “Apache Storm vs. Spark Streaming – Two Stream Processing Platforms compared,” DBTA Workshop on Stream Processing, Berne, Dec 2014.
  - Bobby Evans of Yahoo!, “From Gust to Tempest: Scaling Storm,” talk at Hadoop Summit 2015.
  - Sean T. Allen, Matthew Jankowski, Peter Pathirana ,Storm Applied, Published by Manning, 2015.
  - Rahul Jain, “Real time Analytics with Apache Kafka and Spark,” Big Data Hyderabad Meetup, Oct 2014
  - Roy Campell, “Paxos and ZooKeeper,” Lecture notes of CS498 Cloud Computing, UIUC course, Spring 2014.

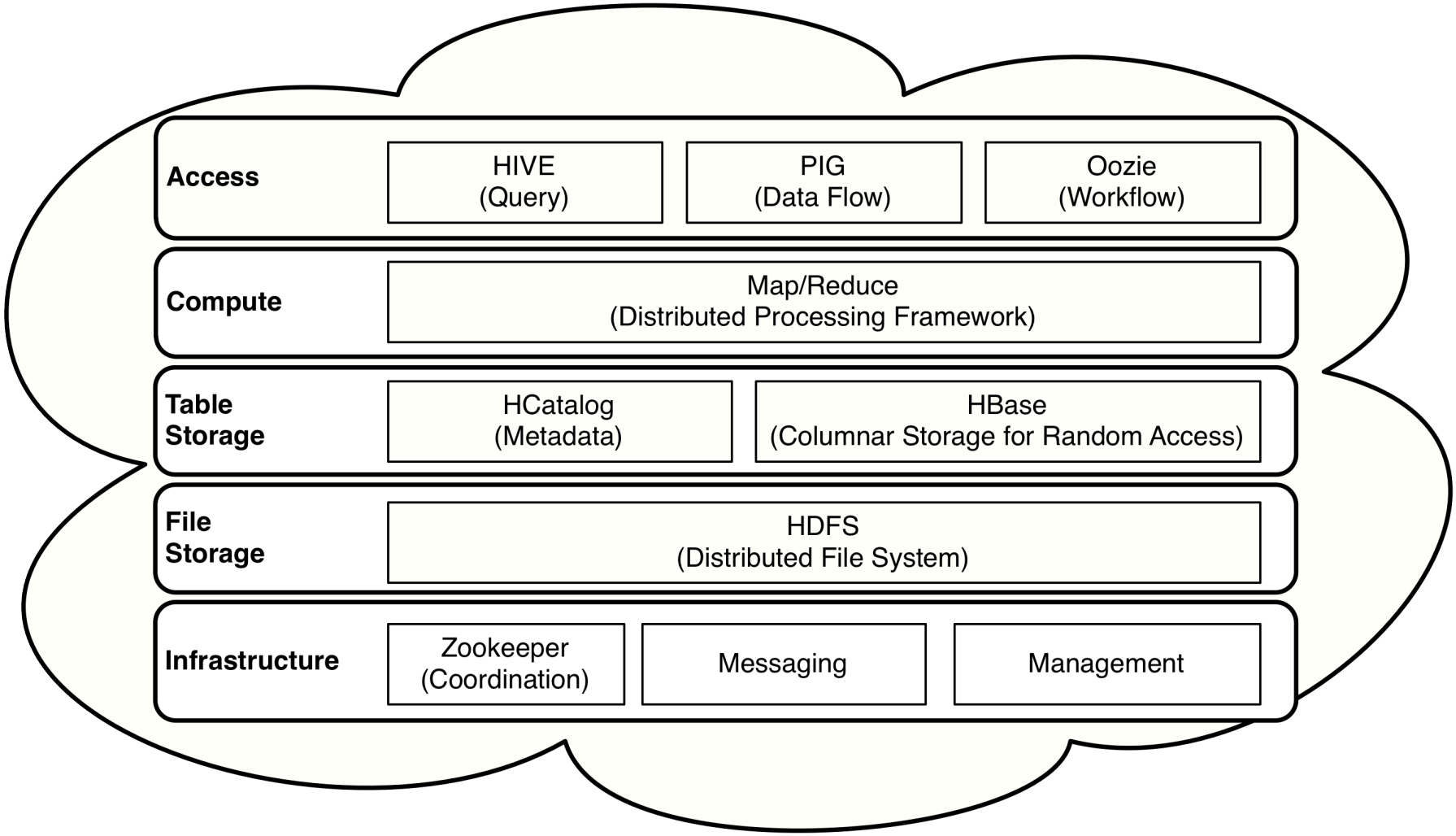
Example Use Case: Data Driven Personalization

http://visualize.yahoo.com/core

System Architecture for Data Driven Personalization based on Offline, Asynchronous (e.g. Daily/Weekly) Log Processing/ Data Analytics



@ Async Tier (before 2010)



Pros and Cons of Async Processing via Hadoop

Strength

- Batch processing
  - simple programming model
- Massively scalable
  - 1000's node cluster w/ commodity hardware
- High throughout
  - Move computation to the data nodes
- Highly available
  - Built-in failover

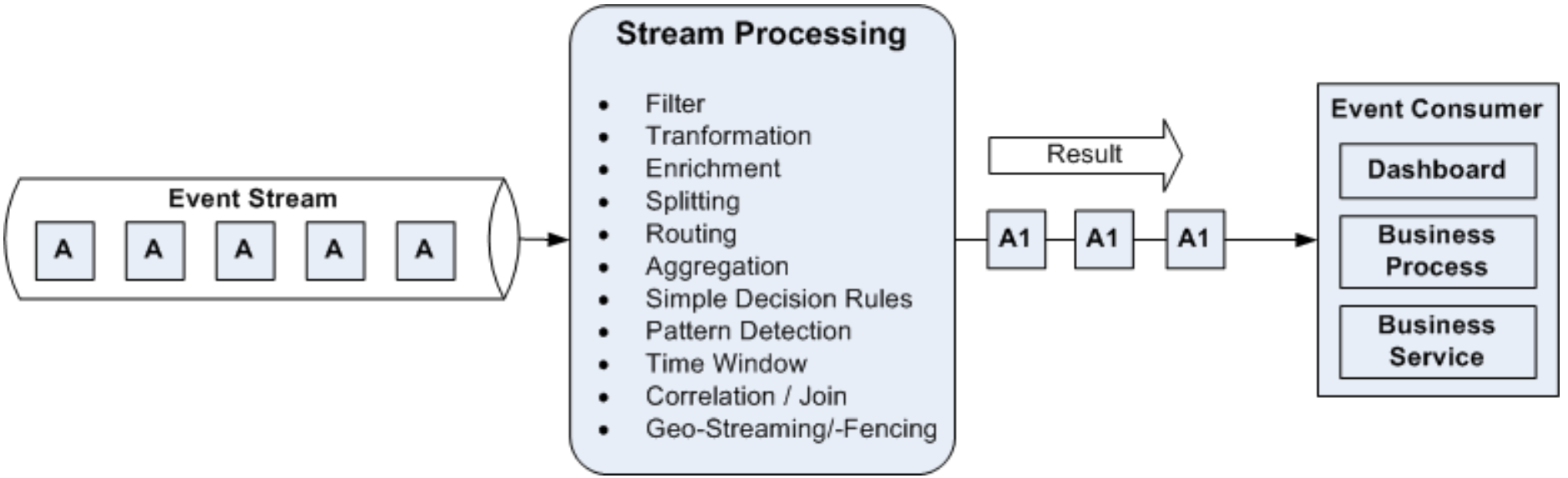
Weakness

- High Latency
  - Minutes or even hours
  - Poor support for Interactive Analysis
  - Inability to Rapidly Respond to Special/ Unexpected Events

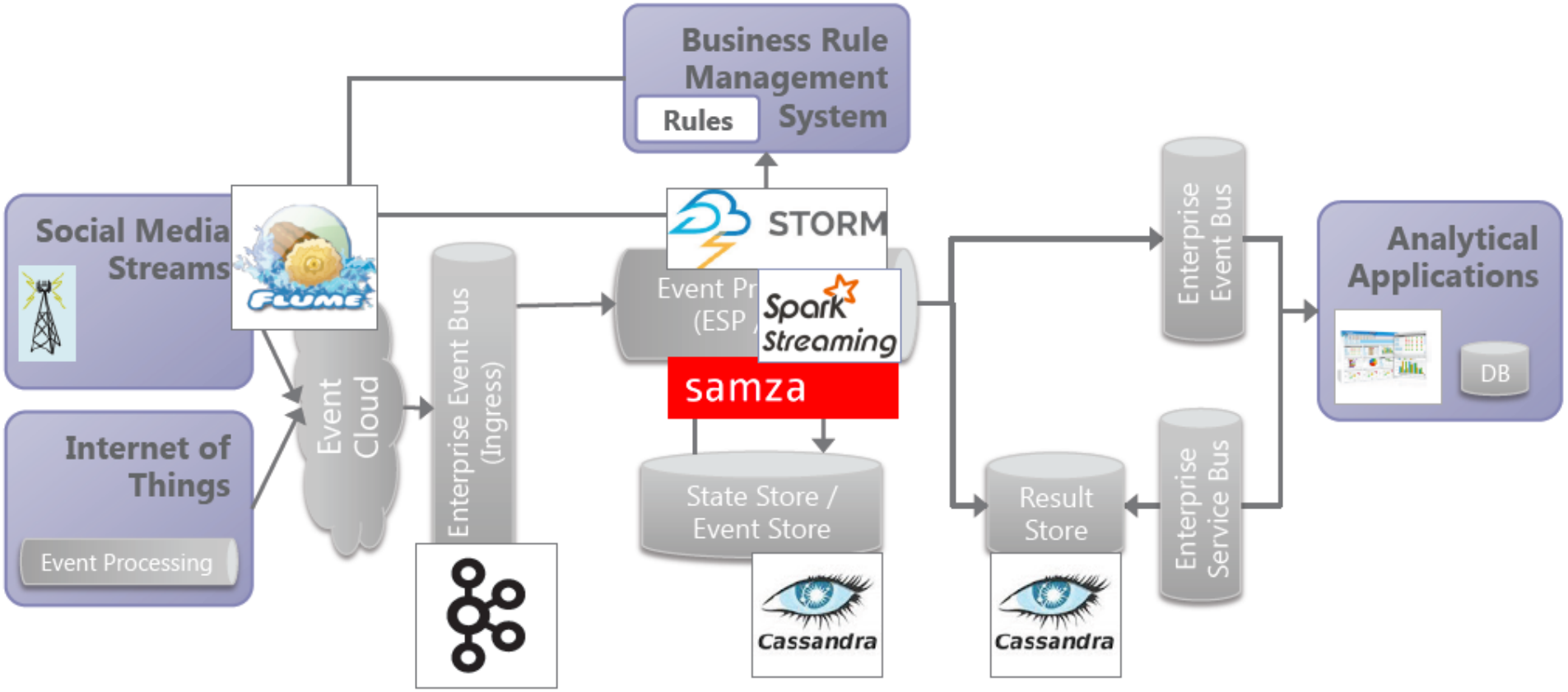
If a company can react to data more quickly, it can make more

What is Stream Processing ?

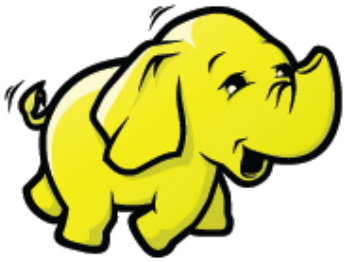
- Infrastructure for continuous (non-stopped, never-ending) data processing
- Computational model can be as general as MapReduce but with the ability to produce results under low-latency constraint
- Input Data collected continuously is naturally processed continuously
- Also known as Event Processing or Complex Event Processing (CEP)



Architectural Pattern #1: A Standalone Event Stream Processing System



Why Stream Processing ?



RPC

Stream Processing

Response latency

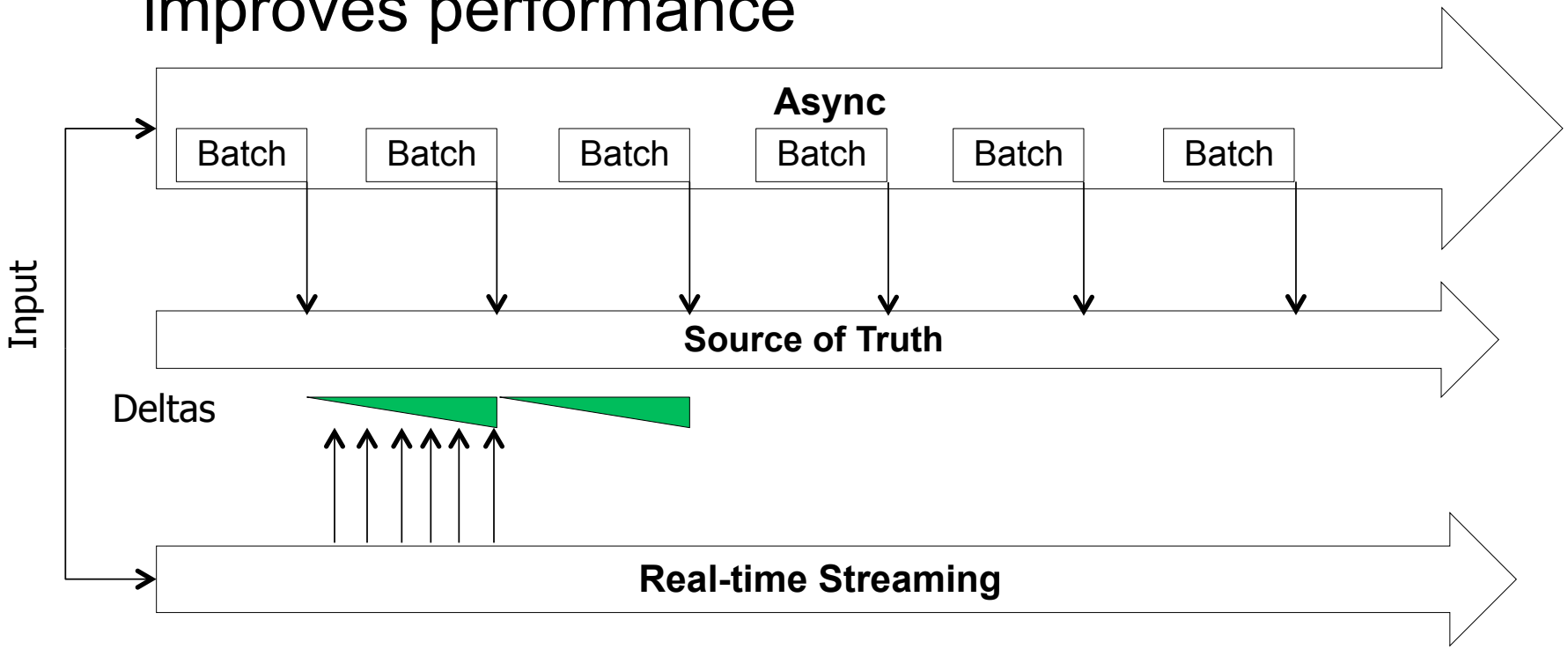
Milliseconds to minutes

Synchronous

Later. Possibly much later.

The Two-Pronged Approach

- http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
- The interesting take-away: Fast Real-Time path with Batch Backup reduces complexity and improves performance

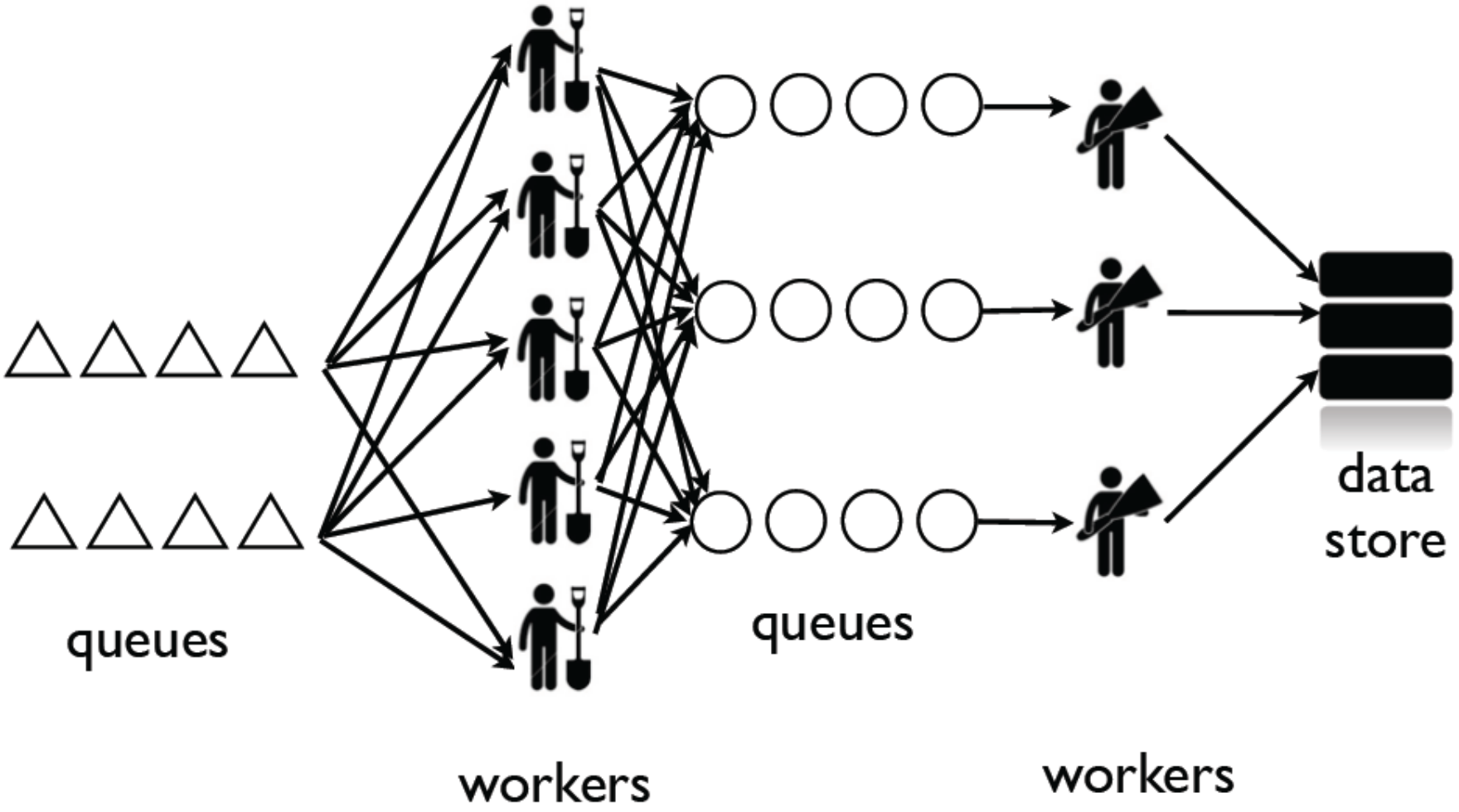


Stream Processing with Apache Storm

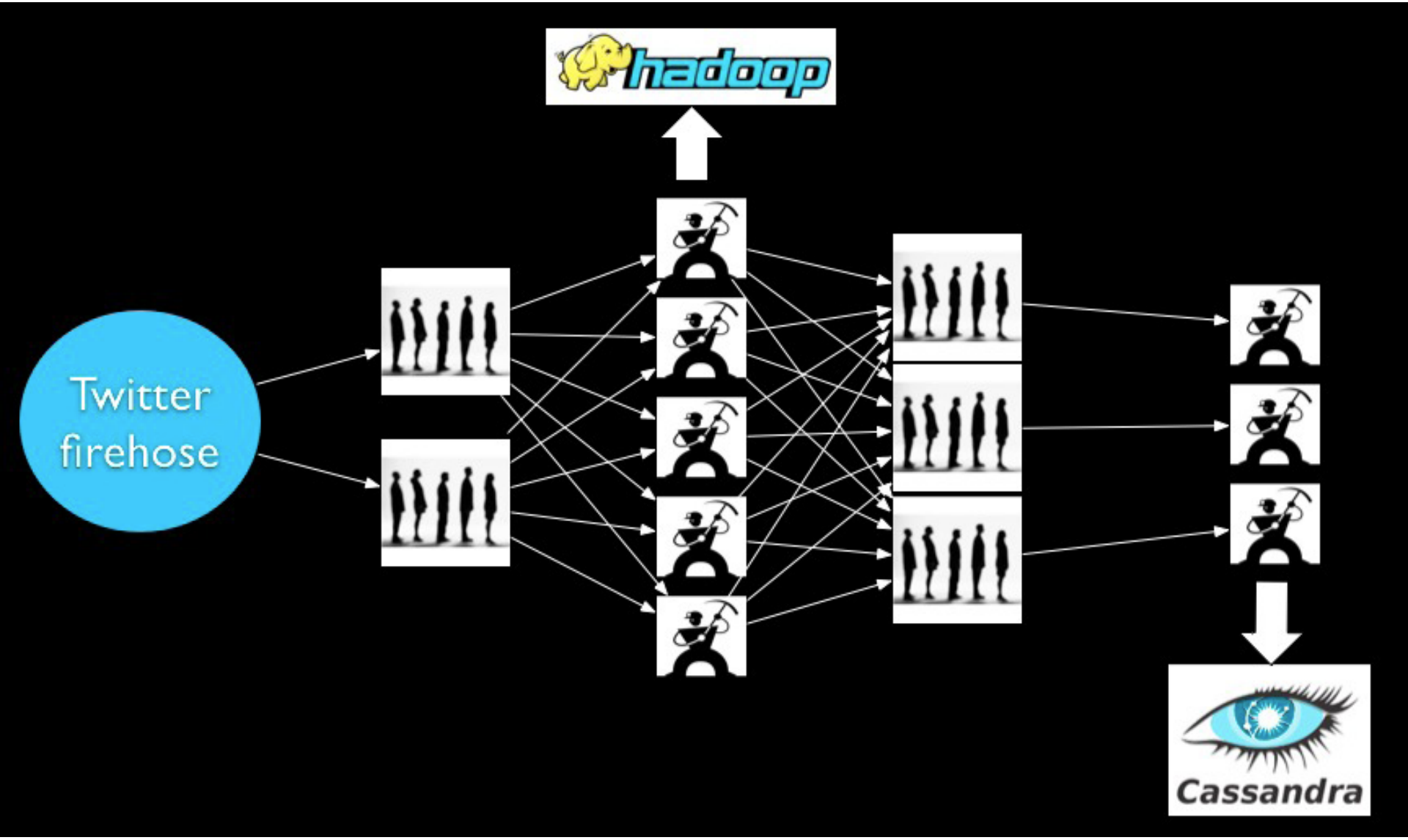
Agenda

- Motivation for Stream Processing
- Apache Storm
  - Stream-based Programming Models and Examples
  - Different Flavors of Processing Guarantees
  - Additional Computational Models with Storm
    - Distributed Remote Procedure Call (DRPC)
    - Transactional Processing with Trident (over Storm)
  - Storm System Architecture
  - A Digression to ZooKeeper
  - Operational Guidelines for Storm
  - Adoption Statistics and Real-time Use Cases
  - Future Extensions (Researchie)

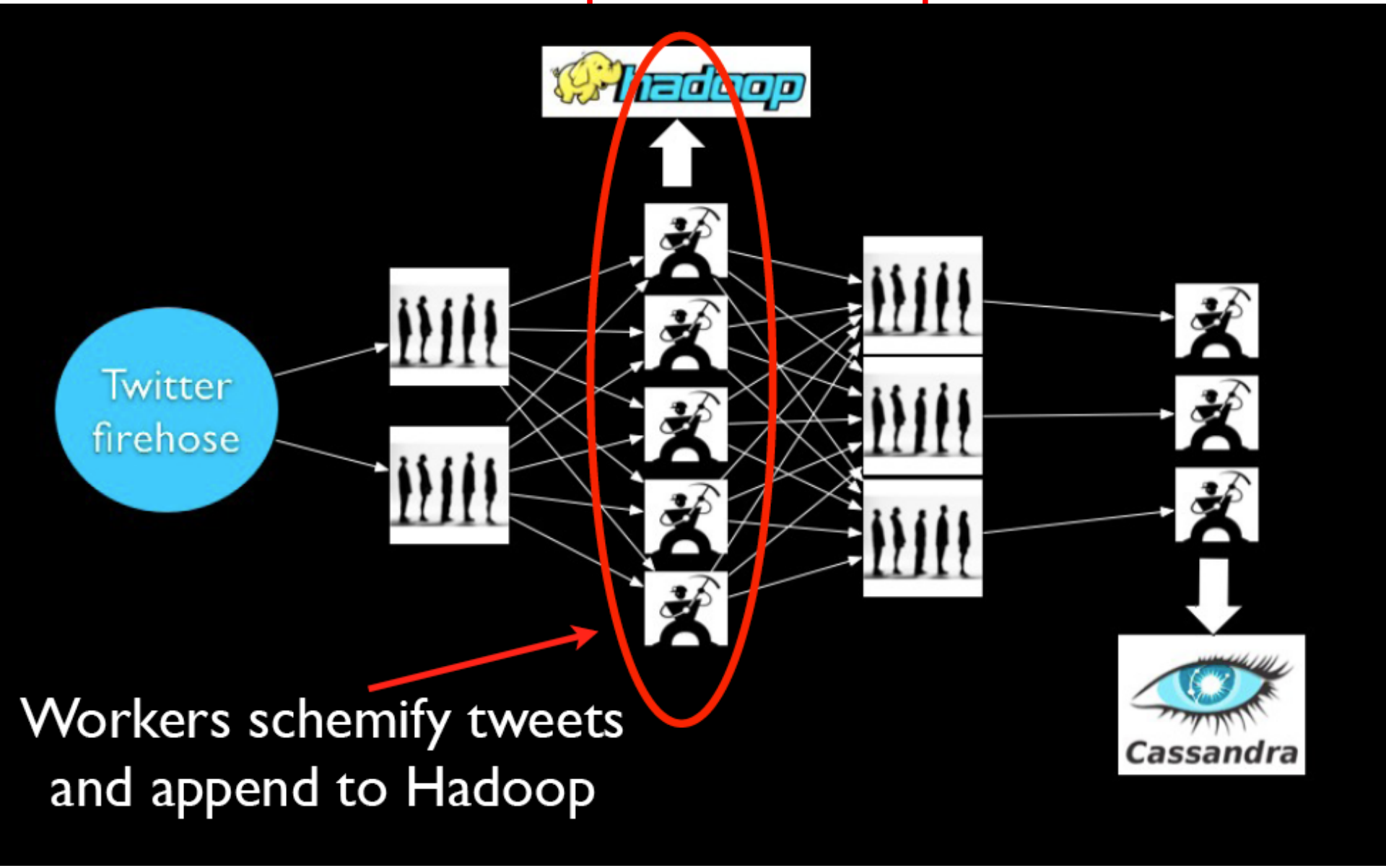
Traditional Workflow under the Queues-Workers Model for Event Processing



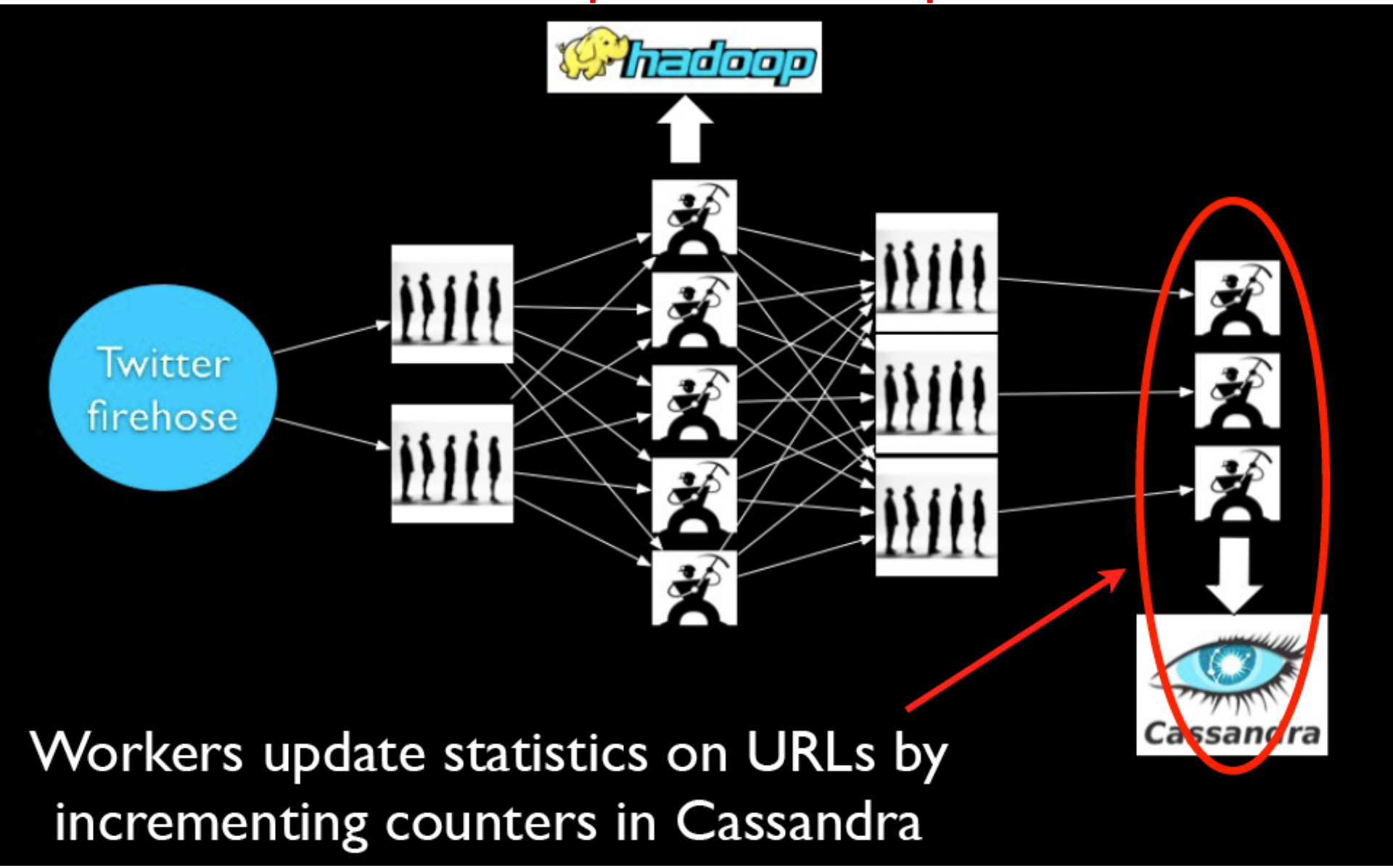
A Simplified Example



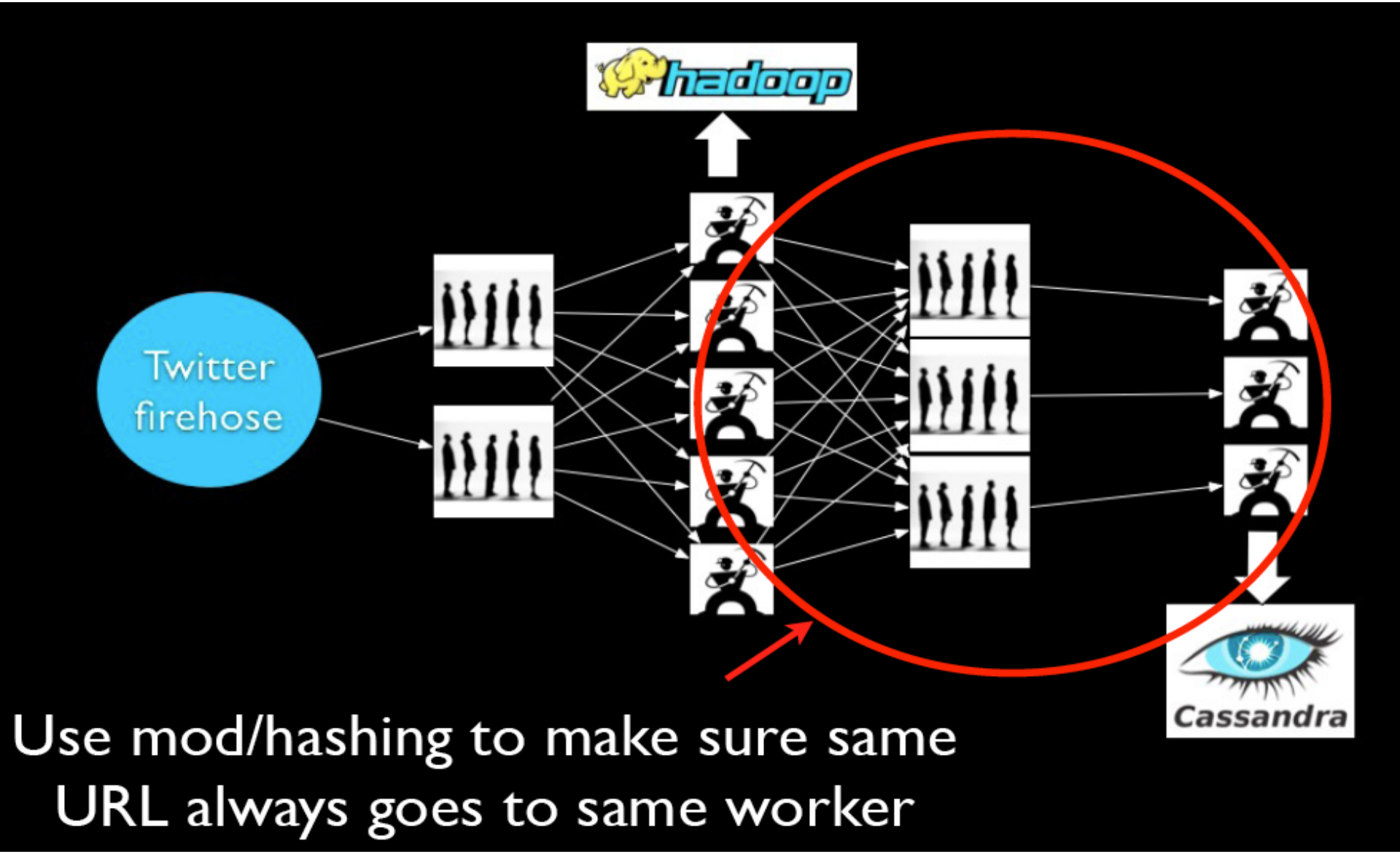
A Simplified Example



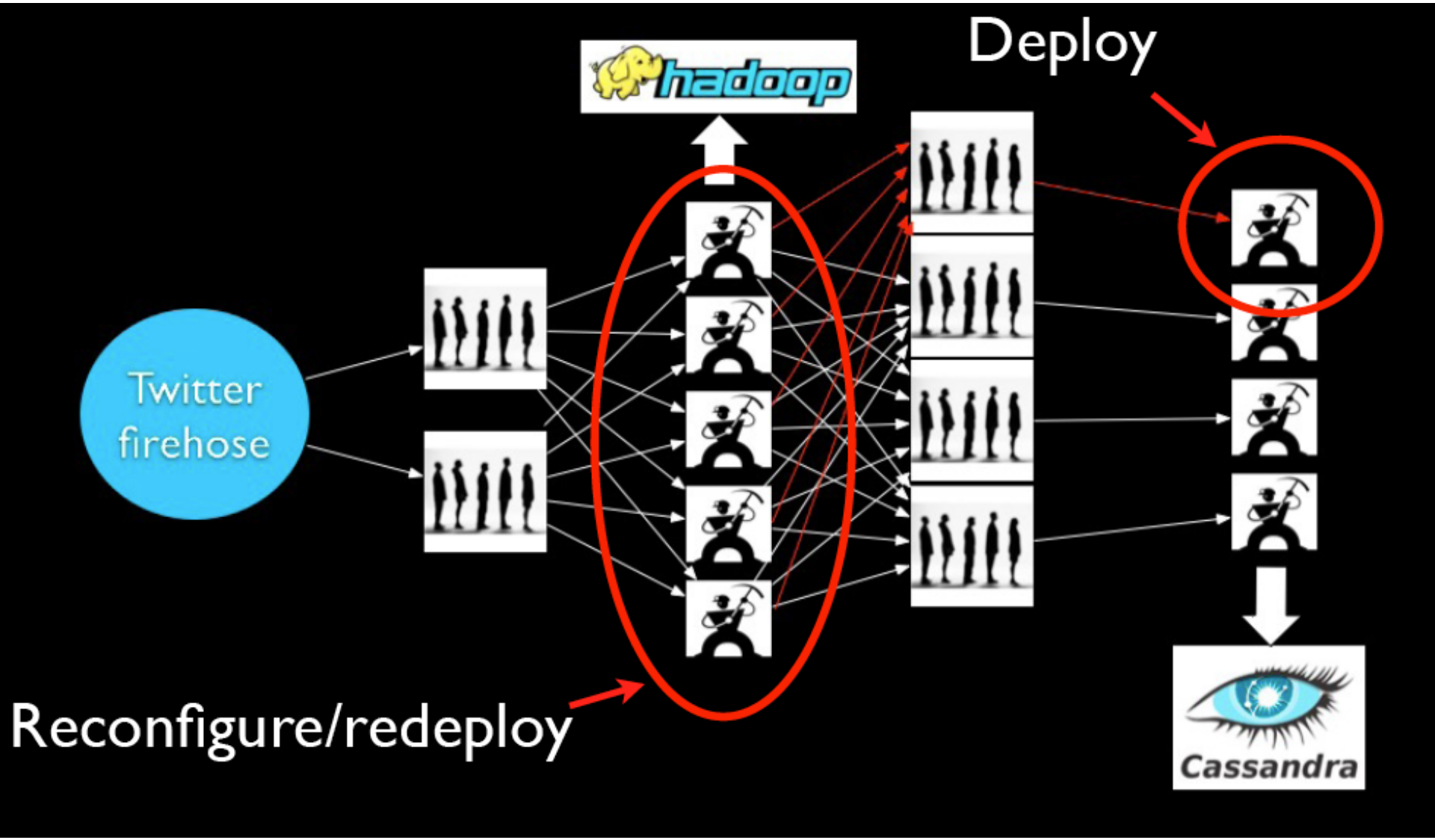
A Simplified Example



A Simplified Example



The procedure to scale-up the system



Problems of Traditional Workflow model

- Scaling is Painful as it involves Queue-partitioning and deployment of additional Workers (processes/nodes)
- Operational overhead due to Worker failures and Queue-Backups
- Coding is Tedious
- No guarantees on whether incoming Data is being processed

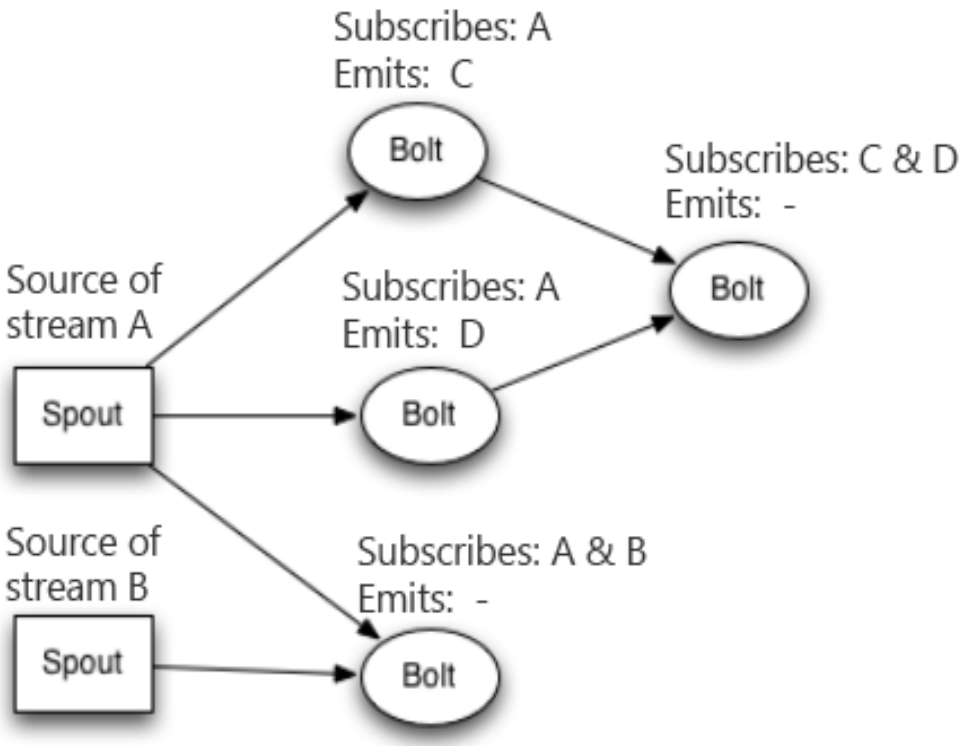
A Solution: Apache Storm

- Focus on the support of Real-Time Streaming jobs
- To simplify dealing with queues (for tasks) and many workers (for load balancing/ parallelization)
- Higher level abstraction than message passing
  - No intermediate message brokers!
- Guaranteed data processing (at least once)
- Horizontal scalability
- Fault-tolerance
- Complementary to Hadoop:
  - The “Hadoop” of real time streaming jobs
  - The “Summingbird” system by Twitter can actually compile a single programming script into a Storm and Hadoop version separately
- Built by Nathan Marz et al at Backtype, acquired and hardened by Twitter in 2011 ; Open-sourced under Apache license since 2013
- Written in Clojure (a dialect of LISP): a Functional Programming Language which generates bytecodes for JVM ;
- Let users (programmers) program in Java and Clojure
- At Twitter, Storm had been decommissioned as of summer 2015. Storm has been replaced by Heron which uses the SAME programming abstraction and is 100% API-compatible with Storm

Key Concepts in Storm

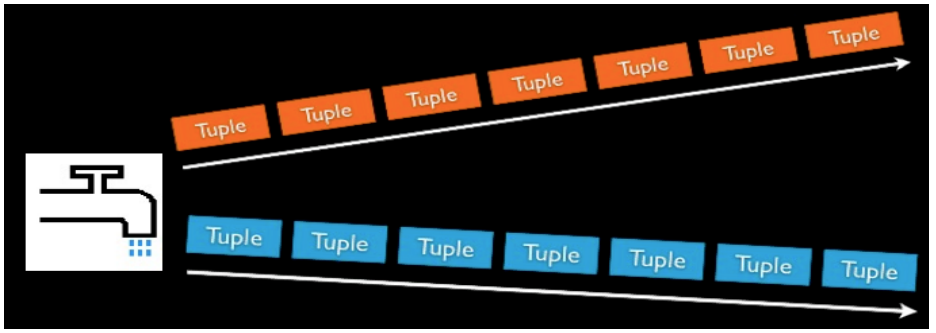
- Stream

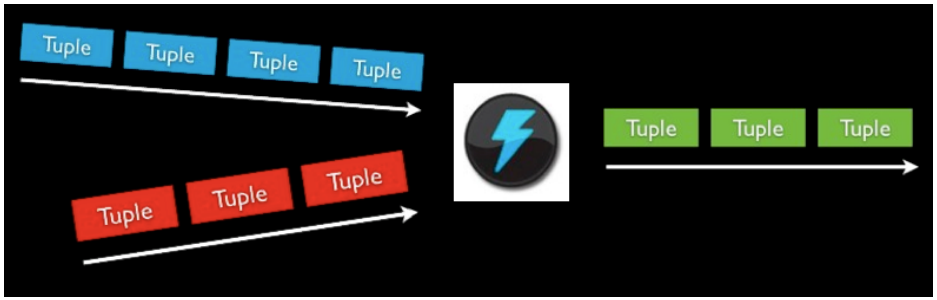
  - An Unbounded sequence of Tuples
  - Core Abstraction in Storm
  - Defined with a Schema that names the fields in the Tuple
  - Value must be serializable
  - Every Stream has an ID
- Topologies
  - A Directed Acyclic Graph (DAG) where each node is either a Data source (Spout) or a Processing node (Bolt)
  - An Edge indicates which Bolt subscribes to which Stream

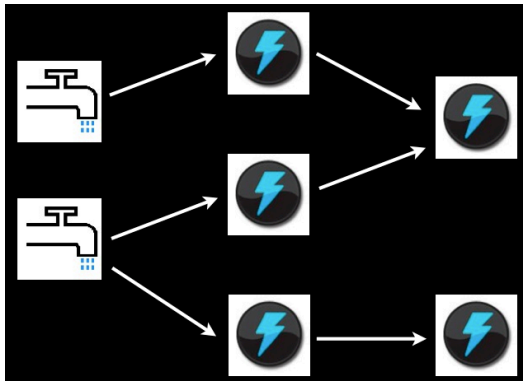


Key Concepts in Storm (cont'd)

- Spout
  - Source of data stream (tuples), e.g.
    - Read from the Twitter streaming API (tuples = tweets)
    - Read from a http server log (tuples = http requests)
    - Read from a Kafka queue (tuples = events)
- Bolt
  - Processes 1+ input stream(s) and produces 1+ new stream(s)
    - e.g. Calculate, Functions, Filters, Aggregation, Joins, talk to database
  - Complex operations may require more than 1 Bolt
- A sample Storm Topology
  - Compiled to be executed on many machines similar to a MapReduce job in Hadoop

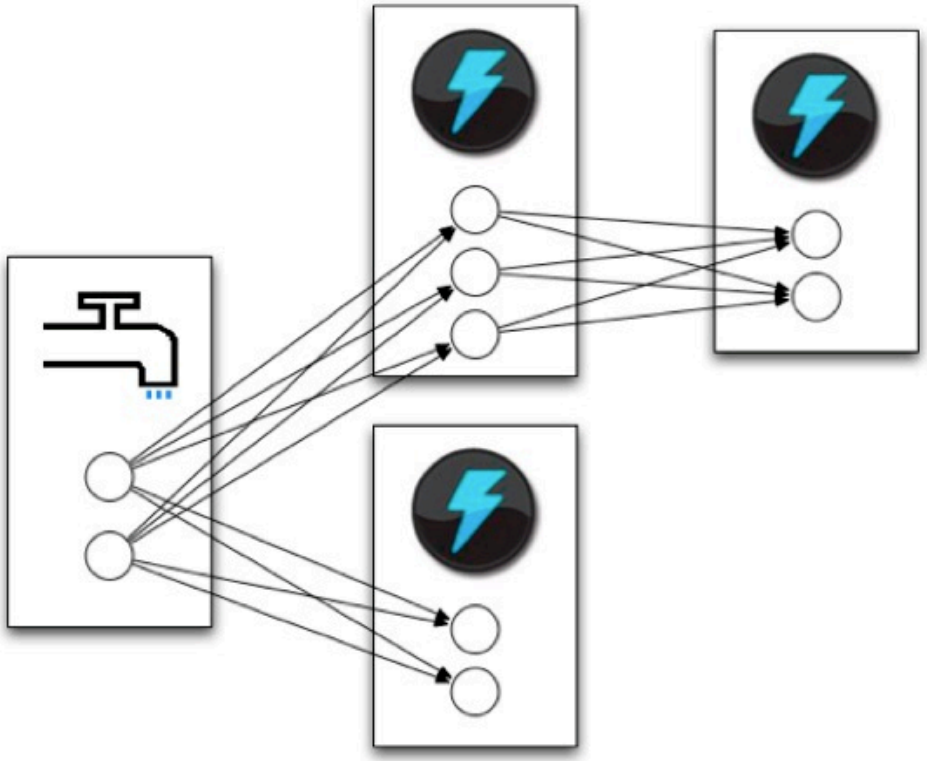


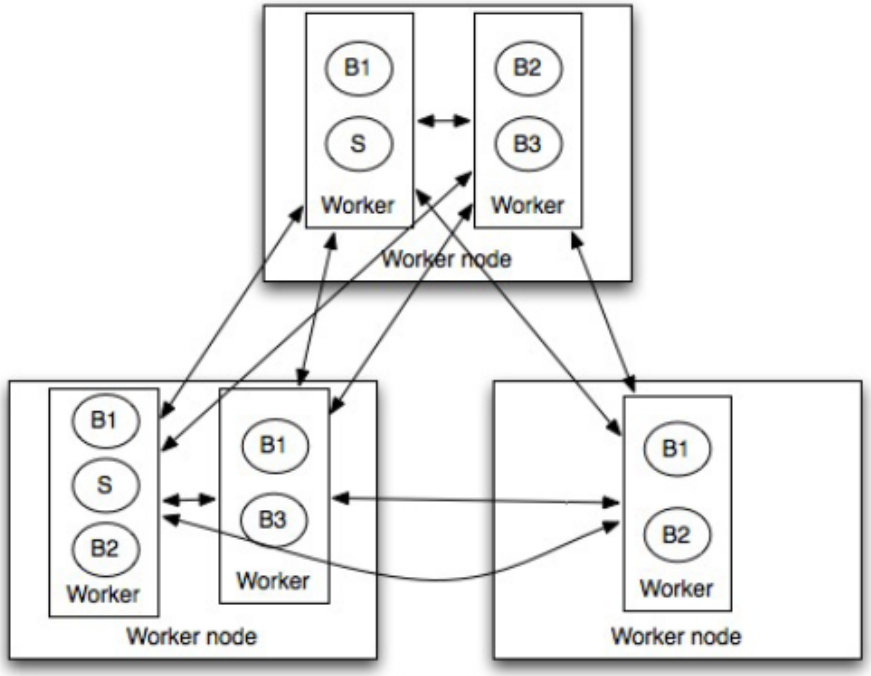




Storm Tasks

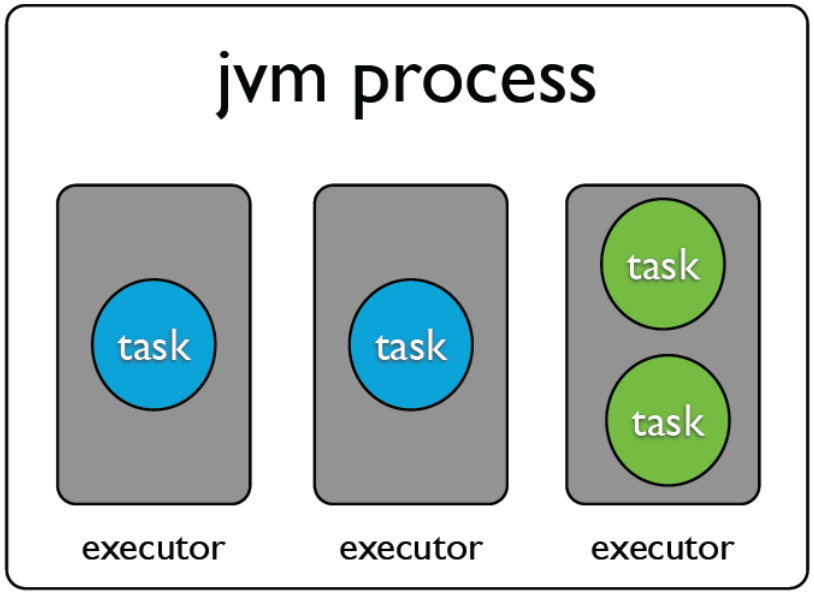
- Each Spout or Bolt is executed as one or more Tasks (instances) across the cluster



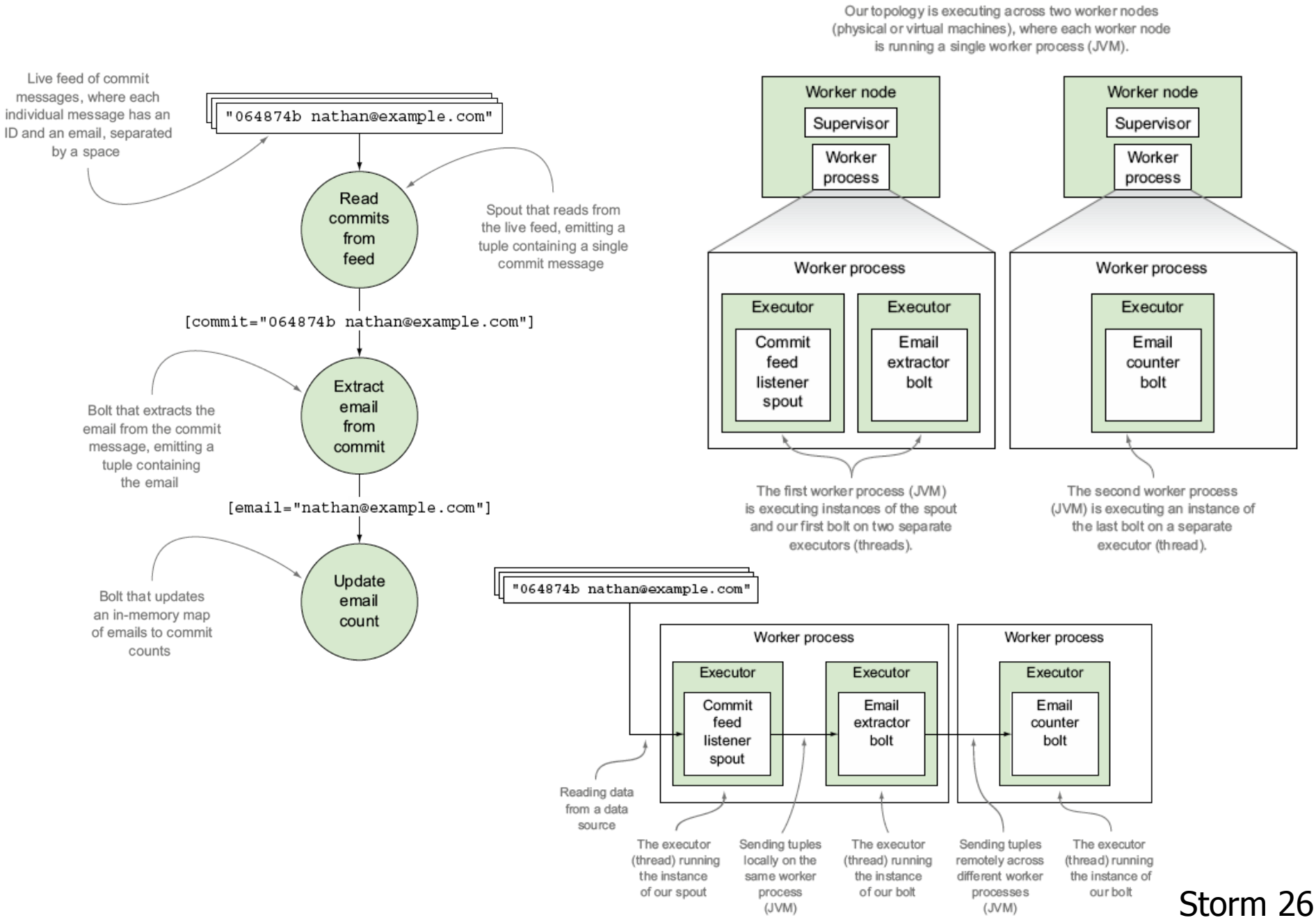


Key Concepts in Storm (cont'd)

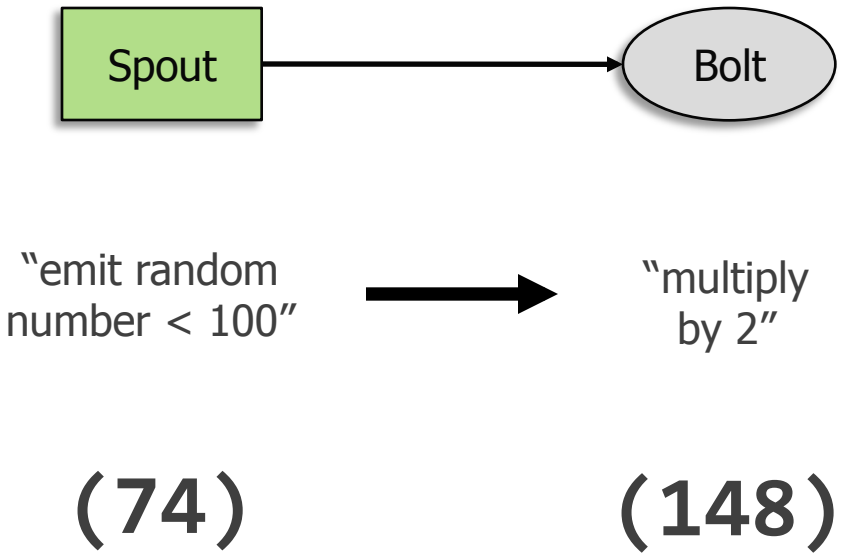
- Worker (JVM) Process
  - Executes subset of a Topology ;
  - May run 1 or more threads (Executors) for one or more components
  - One Thread per Executor
- Task
  - The actual data processing instance executing by the thread
  - It is possible for multiple tasks to share one thread (Why ? to facilitate dynamic scaling)



An Example on deploying a Topology across a Cluster



A trivial “Hello, Storm” topology



Code

Spout

```
public void nextTuple() {
    final Random rand = new Random(); // normally this should be an instance field
    int nextRandomNumber = rand.nextInt(100);
    collector.emit(new Values(nextRandomNumber)); // auto-boxing
}
```

Bolt

```
Override
public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
    this.collector = collector;
}

@Override
public void execute(Tuple tuple) {
    Integer inputNumber = tuple.getInteger(0);
    collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
    collector.ack(tuple);
}

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

Code

Topology config – for running *on your local laptop*

```
1  Config conf = new Config();
2  conf.setNumWorkers(1);
3
4  topologyBuilder.setSpout("my-spout", new MySpout(), 2);
5
6  topologyBuilder.setBolt("my-bolt", new MyBolt(), 2)
7                 .shuffleGrouping("my-spout");
8
9  StormSubmitter.submitTopology(
10         "mytopology",
11         conf,
12         topologyBuilder.createTopology()
13     );
```

Code

Topology config – for running *on a production Storm cluster*

```
1  Config conf = new Config();
2  conf.setNumWorkers(200);
3
4  topologyBuilder.setSpout("my-spout", new MySpout(), 100);
5
6  topologyBuilder.setBolt("my-bolt", new MyBolt(), 200)
7                 .shuffleGrouping("my-spout");
8
9  StormSubmitter.submitTopology(
10         "mytopology",
11         conf,
12         topologyBuilder.createTopology()
13     );
```

Creating a spout

- Very often, it suffices to use an existing spout (Kafka spout, Redis spout, etc).
- But you usually needs to implement your own **bolts** to realize your specific computation.

Creating a Bolt

- Storm is polyglot – but we focus on Java.
- Two main options for JVM users:
  - Implement the IRichBolt or IBasicBolt interfaces
  - Extend the BaseRichBolt or BaseBasicBolt abstract classes
- `BaseBasicBolt`
  - Auto-acks the incoming tuple at the end of its `execute()` method.
  - With the right type of Spout (reliable one), "at-least-once" processing guarantee for each tuple is already supported automatically (and implicitly).
  - These bolts are typically simple functions or filters.
- `BaseRichBolt`
  - Allow one to specify complex tuple-anchoring/ack mechanism explicitly.
  - Need to use this type of bolt if one wants "at-most-once", i.e. no guarantee in tuple-processing ;
  - You must – and are able to – manually `ack()` an incoming tuple.
  - Can be used to delay acking a tuple, e.g. for algorithms that need to work across multiple incoming tuples.

Extending `BaseRichBolt`

- Let's re-use our previous example bolt.

```
1  Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

Extending `BaseRichBolt`

- **`execute()`** is the heart of the bolt.
- This is where you will focus most of your attention when implementing your bolt or when trying to understand somebody else's bolt.

```
1  Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

Extending `BaseRichBolt`

- **`prepare()`** acts as a “second constructor” for the bolt’s class.
- Because of Storm’s distributed execution model and serialization, prepare() is often needed to fully initialize the bolt on the target JVM.

```
1  Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

Extending `BaseRichBolt`

- **`declareOutputFields()`** tells downstream bolts about this bolt's output format. What you declare must match what you actually emit().
  - You will use this information in downstream bolts to "extract" the data from the emitted tuples.
  - If your bolt only performs side effects (e.g. talk to a DB) but does not emit an actual tuple, override this method with an empty {} method.

```
1  Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

Common spout/bolt gotchas

- `NotSerializableException` at run-time of your topology
  - Typically you will run into this because your bolt has fields (instance or class members) that are not serializable. This recursively applies to each field.
  - The root cause is Storm's distributed execution model and serialization: Storm code will be shipped – first serialized and then deserialized – to a different machine/JVM, and then executed. (see docs for details)
  - How to fix?
    - Solution 1: Make the culprit class serializable, if possible.
    - Solution 2: Register a custom Kryo serializer for the class.
    - Solution 3a (Java): Make the field `transient`. If needed, initialize it in `prepare()`.
    - Solution 3b (Scala): Make the field `@transient lazy val` (Scala). If needed, turn it into a `var` and initialize it in in `prepare()`.
      - For example, the `var/prepare()` approach may be needed if you use the factory pattern to create a specific type of a collaborator within a bolt. Factories come in handy to make the code testable. See AvroKafkaSinkBolt in kafka-storm-starter for such a case.

Common spout/bolt gotchas

- Tick tuples are configured *per-component*, i.e. per bolt
  - Idiomatic approach to trigger periodic activities in your bolts: “Every 10s do XYZ.”
  - Don't configure them per-topology as this will throw a `RuntimeException`.

```
@Override
public Map<String, Object> getComponentConfiguration() {
  Config config = new Config();
  config.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 10);
  return conf;
}
```

- Tick tuples are not 100% guaranteed to arrive *in time*
  - They are sent to a bolt just like any other tuples, and will enter the same queues and buffers. Congestion, for example, may cause tick tuples to arrive too late.
  - Across different bolts, tick tuples are not guaranteed to arrive at the same time, even if the bolts are configured to use the same tick tuple frequency.
  - Currently, tick tuples for the same bolt will arrive at the same time at the bolt's various task instances. However, this property is not guaranteed for the future.

http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/

Common spout/bolt gotchas

- When using tick tuples, forgetting to handle them "in a special way"
  - Trying to run your normal business logic on tick tuples – e.g. extracting a certain data field – will usually only work for normal tuples but fail for a tick tuple.

```
@Override
public void execute(Tuple tuple) {
  if (isTickTuple(tuple)) {
    // now you can trigger e.g. a periodic activity
  }
  else {
    // do something with the normal tuple
  }
}

private static boolean isTickTuple(Tuple tuple) {
  return tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID)
    && tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID);
}
```

- When using tick tuples, forgetting to ack() them
  - Tick tuples must be acked like any other tuple.

http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/

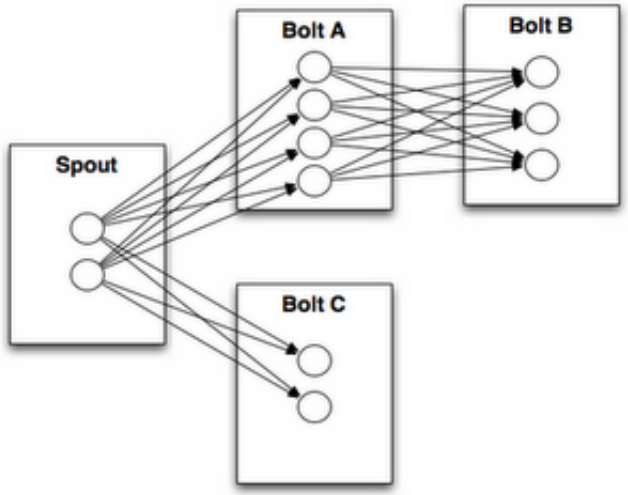
Common spout/bolt gotchas

- `Outputcollector#emit()` can only be called from the "original" thread that runs a bolt
  - You can start additional threads in your bolt, but only the bolt's own thread may call `emit()` on the collector to write output tuples. If you try to emit tuples from any of the other threads, Storm will throw a `NullPointerException`.
    - If you need the additional-threads pattern, use e.g. a thread-safe queue to communicate between the threads and to collect [pun intended] the output tuples across threads.
  - This limitation is only relevant for output tuples, i.e. output that you want to send within the Storm framework to downstream consumer bolts.
  - If you want to write data to (say) Kafka instead – think of this as a side effect of your bolt – then you don't need the `emit()` anyways and can thus write the side-effect output in any way you want, and from any thread.

Creating a topology

- When creating a topology you're essentially defining the DAG – that is, which spouts and bolts to use, and how they interconnect.
  - `TopologyBuilder#setSpout()` and `TopologyBuilder#setBolt()`
  - Groupings between spouts and bolts, e.g. `shuffleGrouping()`

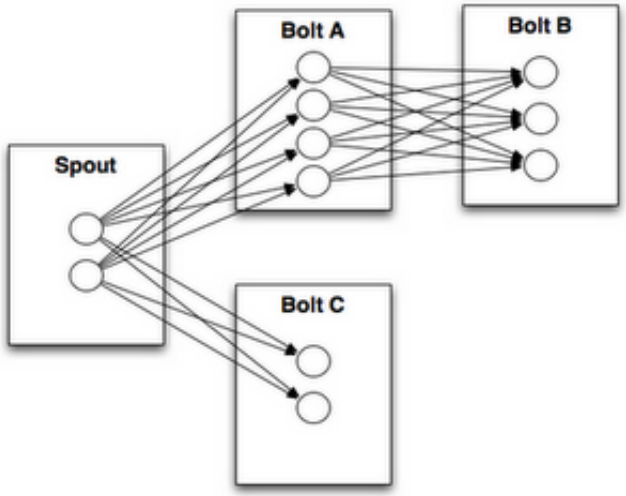
```
1  Config conf = new Config();
2  conf.setNumWorkers(200);
3
4  TopologyBuilder builder = new TopologyBuilder();
5  builder.setSpout("my-spout", new MySpout(), 100);
6  builder.setBolt("my-bolt", new MyBolt(), 200)
7         .shuffleGrouping("my-spout");
8  StormTopology topology = builder.createTopology();
```



Creating a topology

- You must specify the initial *parallelism* of the topology.
  - Crucial for Performance &Scaling but no rule of thumb.
  - You must understand concepts such as workers/executors/tasks.
- Only some aspects of parallelism can be changed later, i.e. at run-time.
  - You can change the #executors (threads).
  - You cannot change #tasks, which remains static during the topology's lifetime.

```
1  Config conf = new Config();
2  conf.setNumWorkers(200);
3
4  TopologyBuilder builder = new TopologyBuilder();
5  builder.setSpout("my-spout", new MySpout(), 100);
6  builder.setBolt("my-bolt", new MyBolt(), 200)
7         .shuffleGrouping("my-spout");
8  StormTopology topology = builder.createTopology();
```



Creating a topology

- You submit a topology either to a “local” cluster or to a real cluster.
  - `LocalCluster#submitTopology`
  - `StormSubmitter#submitTopology()` and `#submitTopologyWithProgressBar()`
  - In your code you may want to use both approaches, e.g. to facilitate local testing.

- Notes
  - A `StormTopology` is a static, serializable Thrift data structure. It contains instructions that tell Storm how to deploy and run the topology in a cluster.
  - The `StormTopology` object will be *serialized*, including *all* the components in the topology's DAG.
  - Only when the topology is *deployed* (and serialized in the process) and *initialized* (i.e. `prepare()` and other life cycle methods are called on components such as bolts) does it perform any actual message processing.

Running a topology

- To run a topology you must first package your code into a “fat jar”.
  - You must include all your code’s dependencies but:
  - Exclude the Storm dependency itself, as the Storm cluster will provide this.
    - Sbt: `"org.apache.storm" % "storm-core" % "0.9.2-incubating" % "provided"`
    - Maven: `<scope>provided</scope>`
    - Gradle with gradle-fatjar-plugin: `compile '...', { ext { fatJarExclude = true } }`
  - Note: You may need to tweak your build script so that your local tests *do include* the Storm dependency. See e.g. `assembly.sbt` in kafka-storm-starter for an example.

- A topology is run via the storm jar command.

```
$ storm jar all-my-code.jar com.miguno.MyTopology arg1 arg2
```

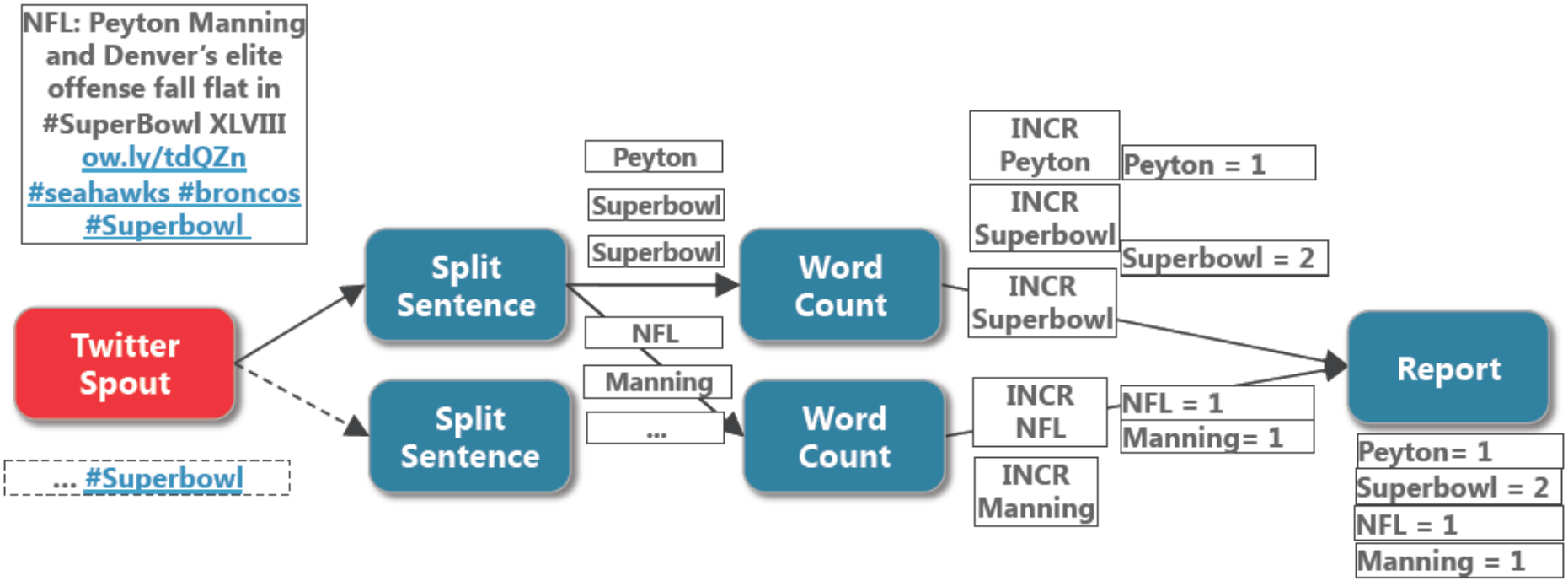
  - Will connects to Nimbus, upload your jar, and run the topology.
  - Use any machine that can run "storm jar" and talk to Nimbus' Thrift port.
  - You can pass additional JVM options via `$STORM_JAR_JVM_OPTS`.

Alright, my topology runs – now what?

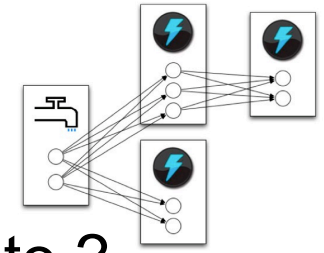
- The topology will run forever or until you kill it.
- Check the status of your topology
  - Storm UI (default: `8080/tcp`)
  - Storm CLI, e.g. `storm [list | kill | rebalance | deactivate | ...]`
  - Storm REST API

- FYI:
  - Storm will guarantee that no data (tuple) is lost, even if machines go down and messages are dropped (as long as you don't disable this feature).
  - But if you store states using your own variables in the bolts/ spouts, the state information would be lost when the bolts/spouts die/ crashes
  - Storm will automatically restart failed tasks, and even re-assign tasks to different machines if e.g. a machine dies.

Another Example: Word Counting with Storm



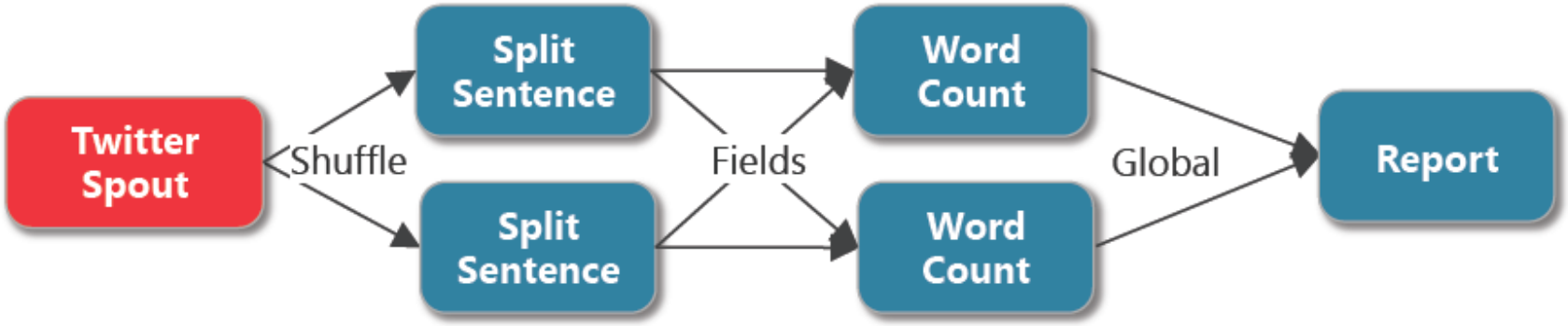
How/ Where to route a tuple?



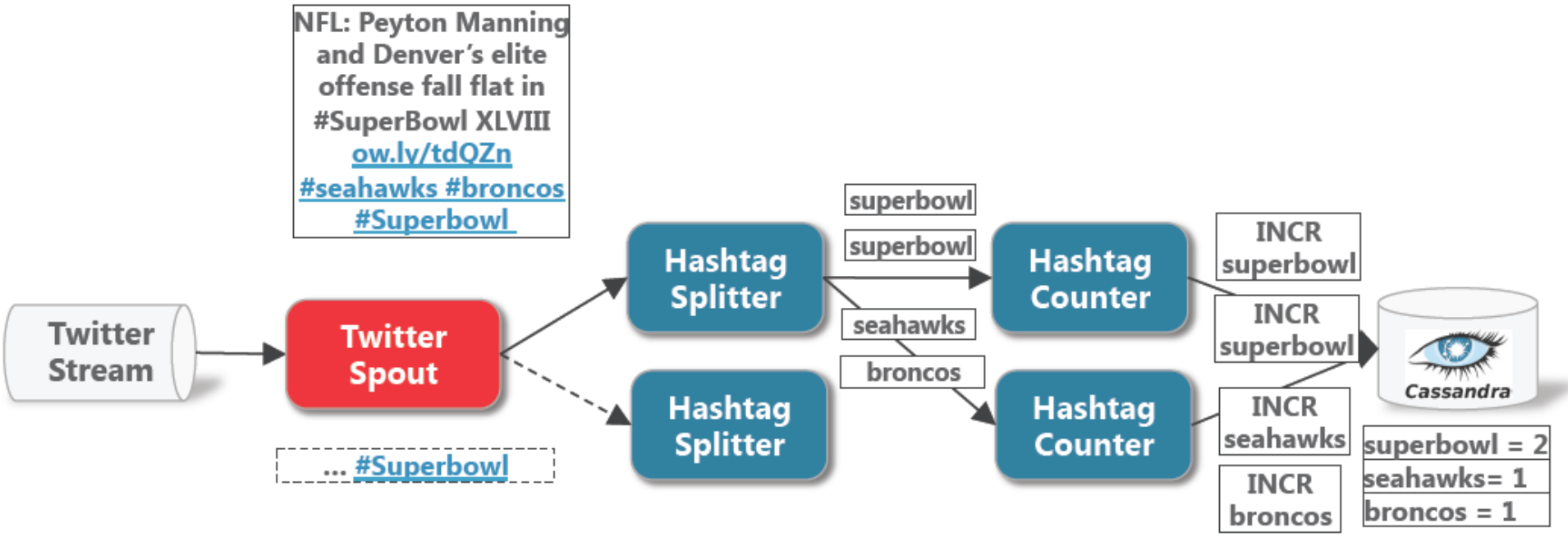
When a tuple is emitted, which task should it be routed to ?
Ans: It is User-Programmable

Shuffle grouping	is random grouping
Fields grouping	is grouped by value, such that equal value results in equal task
All grouping	replicates to all tasks
Global grouping	makes all tuples go to one task
None grouping	makes bolt run in the same thread as bolt/spout it subscribes to
Direct grouping	producer (task that emits) controls which consumer will receive
Local or Shuffle grouping	similar to the shuffle grouping but will shuffle tuples among bolt tasks running in the same worker process, if any. Falls back to shuffle grouping behavior.

- e.g. For the previous Superbowl Tweet Analysis Example:



Using a NoSQL database for storing the results (Keeping state with the counter-type columns)



Here, Cassandra serves as a persistent datastore so that the accumulated counter statistics can survive the crashing of some of the topology's components e.g. one or more of the Hashtag Counter bolt(s)

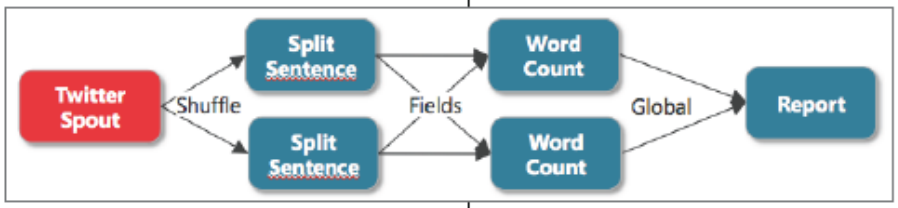
WordCount in Storm (part of the code)

```
TopologyBuilder builder = new TopologyBiulder();
builder.setSpout("spout", new KestrelSpout("kestrel.twitter.com",
                 22133, "sentence_queue", new StringScheme()),5);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12)
                        .fieldGrouping("split", new Fields("word"));

//=============================================================
public static class SplitSentence extends ShellBolt implements IRichBolt {
  //Code to split a sentence
}

public static class WordCount implements IBasicB
//Code to count words, have to override the exec
  public void execute(Tuple tuple, BasicOutputCo
   //…
   }
}
//=============================================================
StormSubmitter.submitTopology("word-count", builder.createTopology());
```

Parallelism Degree (Number of threads for a spout or Bolt)

Actual Code to Create the Topology of the Example

```
public class WordCountTopology {

    private static final String SENTENCE_SPOUT_ID = "sentence-spout";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String TOPOLOGY_NAME = "word-count-topology";

    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout(SENTENCE_SPOUT_ID, spout, 2);
        builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2).setNumTasks(4).shuffleGrouping(SENTENCE_SPOUT_ID);
        builder.setBolt(COUNT_BOLT_ID, countBolt, 4).fieldsGrouping(SPLIT_BOLT_ID,new Fields("word"));
        builder.setBolt(REPORT_BOLT_ID, reportBolt).globalGrouping(COUNT_BOLT_ID);

        Config config = new Config();
        config.setNumWorkers(2);

        if (args.length == 0) {
            LocalCluster cluster = new LocalCluster();

            cluster.submitTopology(TOPOLOGY_NAME, config, builder.createTopology());
            waitForSeconds(10);
            cluster.killTopology(TOPOLOGY_NAME);
            cluster.shutdown();
        } else {
            StormSubmitter.submitTopology(args[0], config, builder.createTopology());
        }
    }
}
```



Storm supports 3 different flavors of Message/Tuple Processing Guarantee

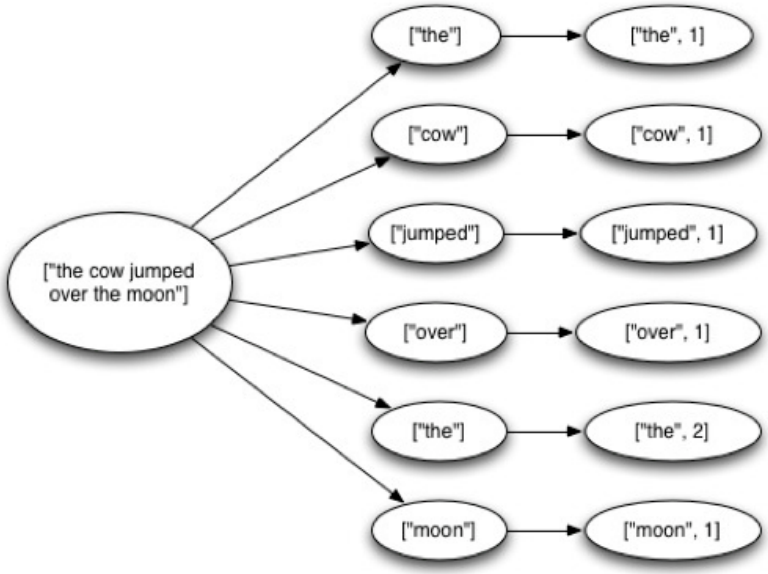
1. No Guarantee at all (like S4 of Yahoo)

2. At Least Once -- i.e. it is possible for some tuple(s) to be repeatedly processed by the topology more than once

3. Exactly Once (like Transaction)
   - but this has feature has been deprecated
   - Now, one should use Trident (is built one the top of Storm) to support Transaction-oriented processing

At Least Once

- Tuple Tree
- A spout tuple is not fully processed until all tuples in the tree have been completed
- If the tuple tree is not completed within a specified timeout, the spout tuple is replayed
- Uses acker tasks to keep track of tuple progress
- Reliability API for the user:




```
public void execute(Tuple tuple) {
    String sentence = tuple.getString(0);
    for(String word: sentence.split(" ")) {
        _collector.emit(tuple, new Values(word));
    }
    _collector.ack(tuple);
}
```

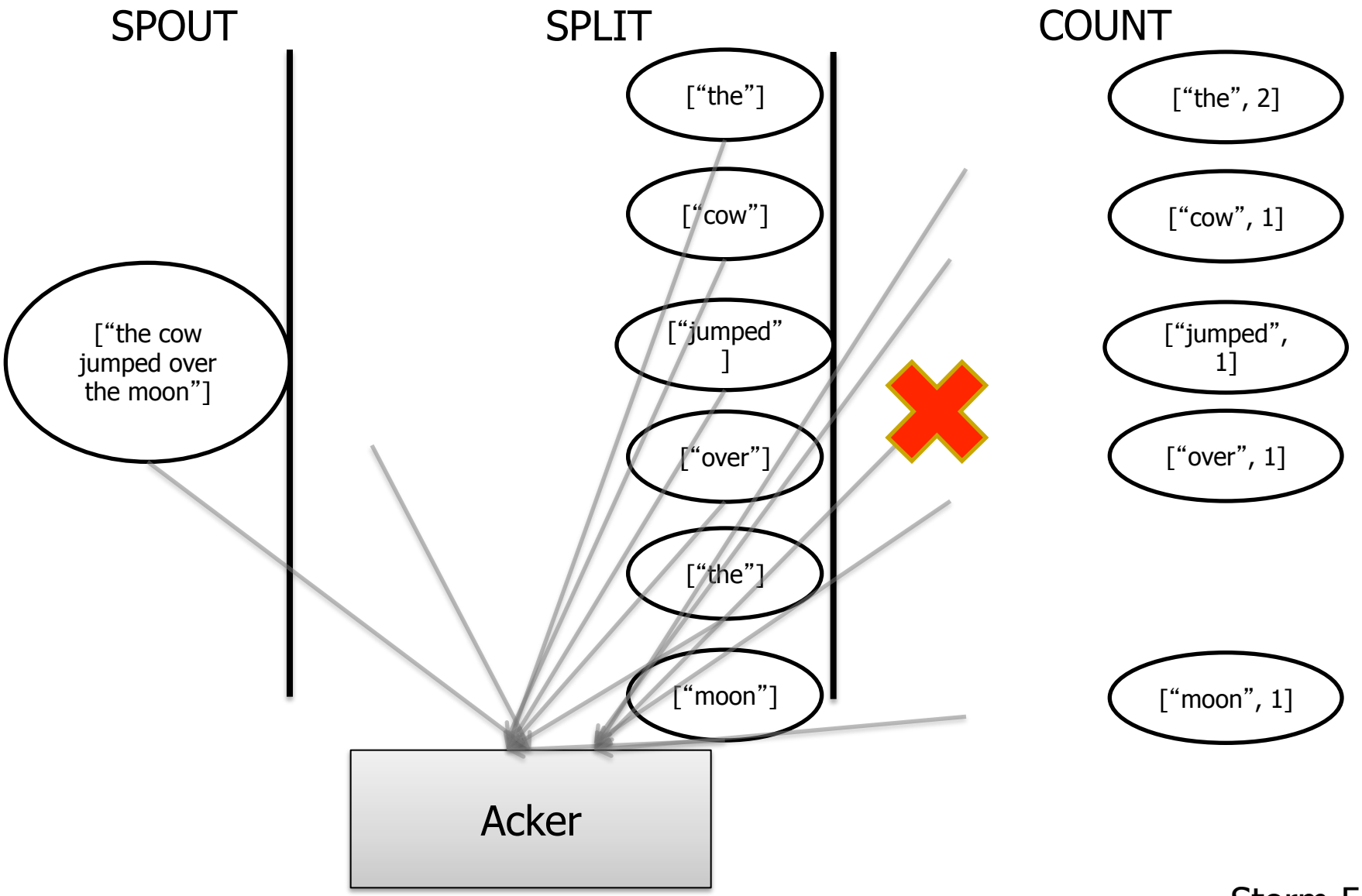
"Anchoring" creates a new edge in the tuple tree

Marks a single node in the tree as complete

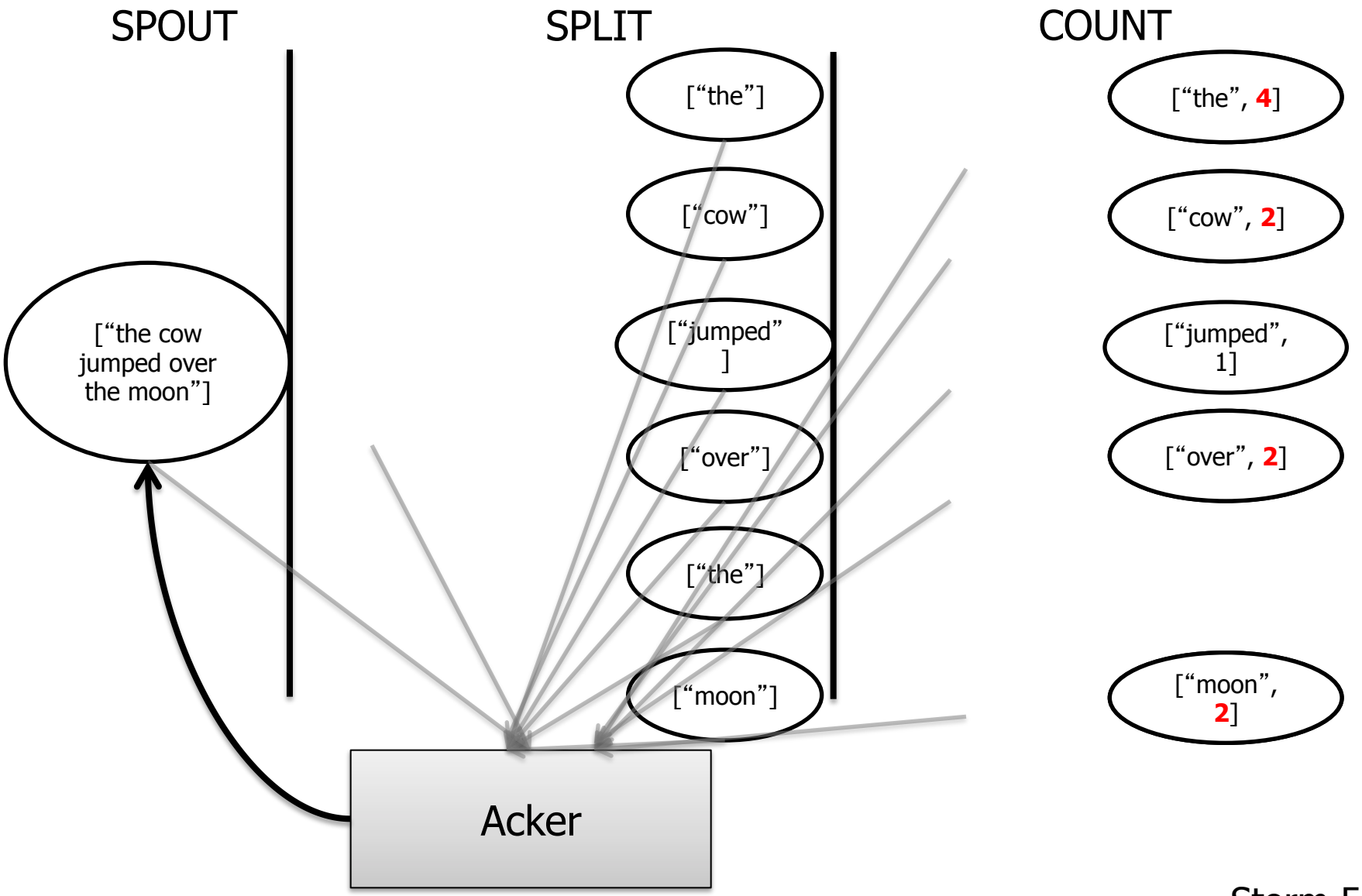
At Least Once

- What happens if there is a failure?
  - You can double process events.
  - This is not so critical if you have something like Hadoop to back you up and correct the issue later.
  - Or if you are looking at statistical trends and replay does not happen that often.
- This requires you to have a spout that supports replay. Not all messaging infrastructure does.

At Least Once



At Least Once



Exactly Once - Transactional Topologies (Deprecated in Storm ; Use Trident instead)

- Transactional Topologies provide a strong ordering of processing.
- A small batch of tuples are processed at a time.
- Each batch completely succeeds or completely fails.
- Each batch is “committed” in order.
- Partial processing is pipelined
- Requires the spout to be able to replay a batch.

Additional Computation models with Storm

- Storm DRPC
  - Parallelize the computation of really intense functions on the fly.
  - Input is a stream of function arguments, and output is a stream of the results for each of those function calls.

- Storm Trident
  - High-level abstraction on top of Storm, which intermixes high throughput and stateful stream processing with low latency distributed querying.
    - Joins, aggregations, grouping, functions, filters.
    - Adds primitives for doing stateful, incremental processing on top of any database or persistence store.
  - Has consistent, exactly-once semantics.
  - Processes a stream as small batches of messages
    - (cf. Spark Streaming)

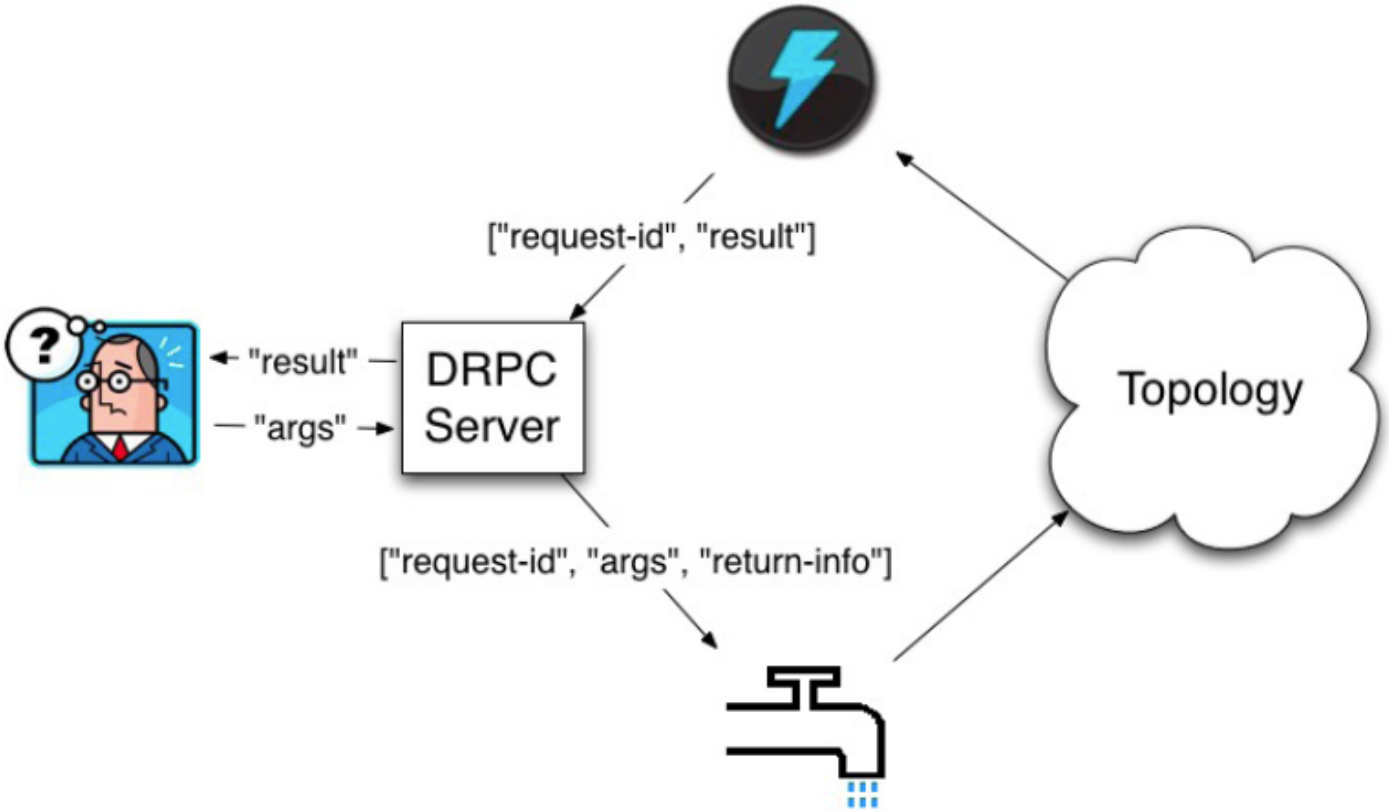
Distributed Remote Procedure Call (DRPC)

DRPC

- Distributed Remote Procedure Call
- Turn a RPC call into a tuple sent from a spout
- Take a result from that and send it back to the user.

DRPC

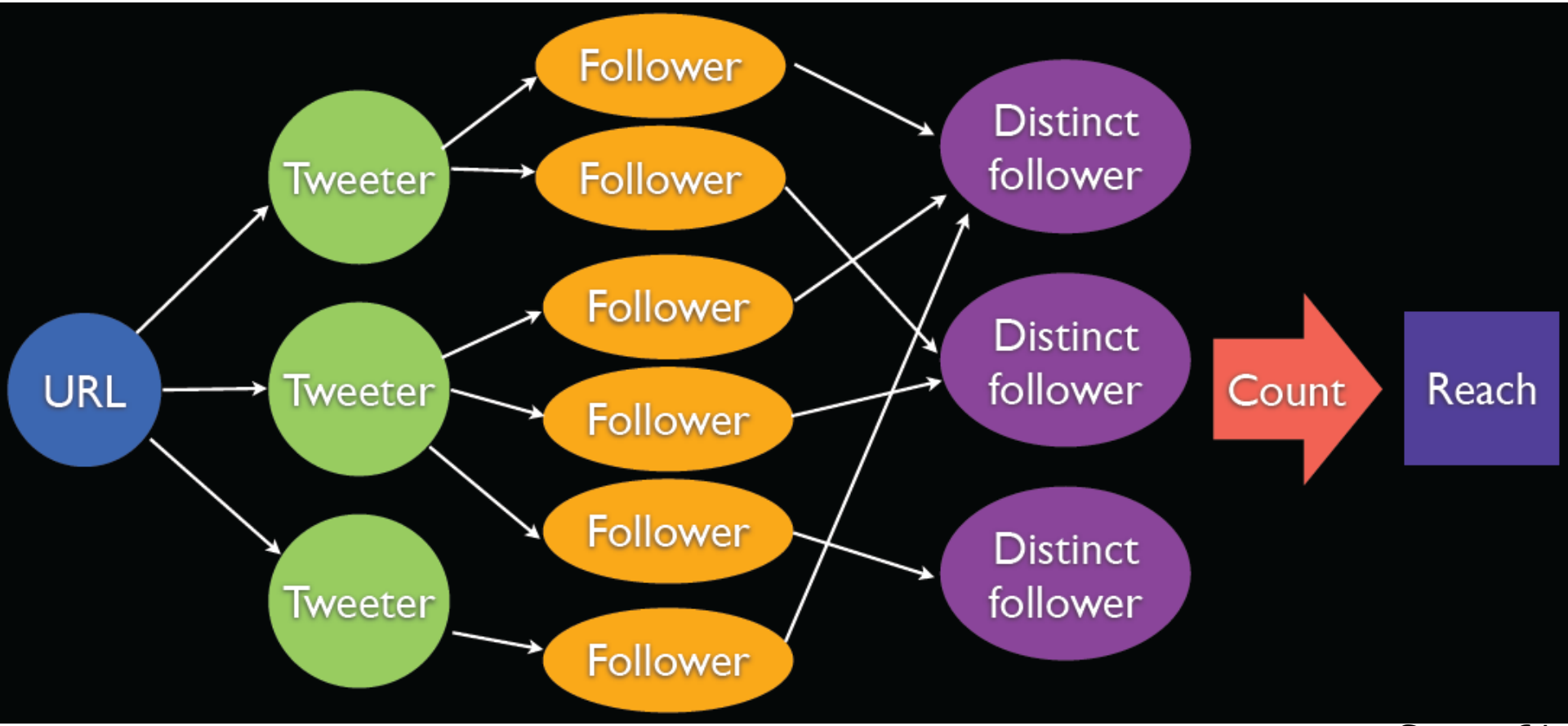
- Distributed Remote Procedure Call
- Turn a RPC call into a tuple sent from a spout
- Take a result from that and send it back to the user.



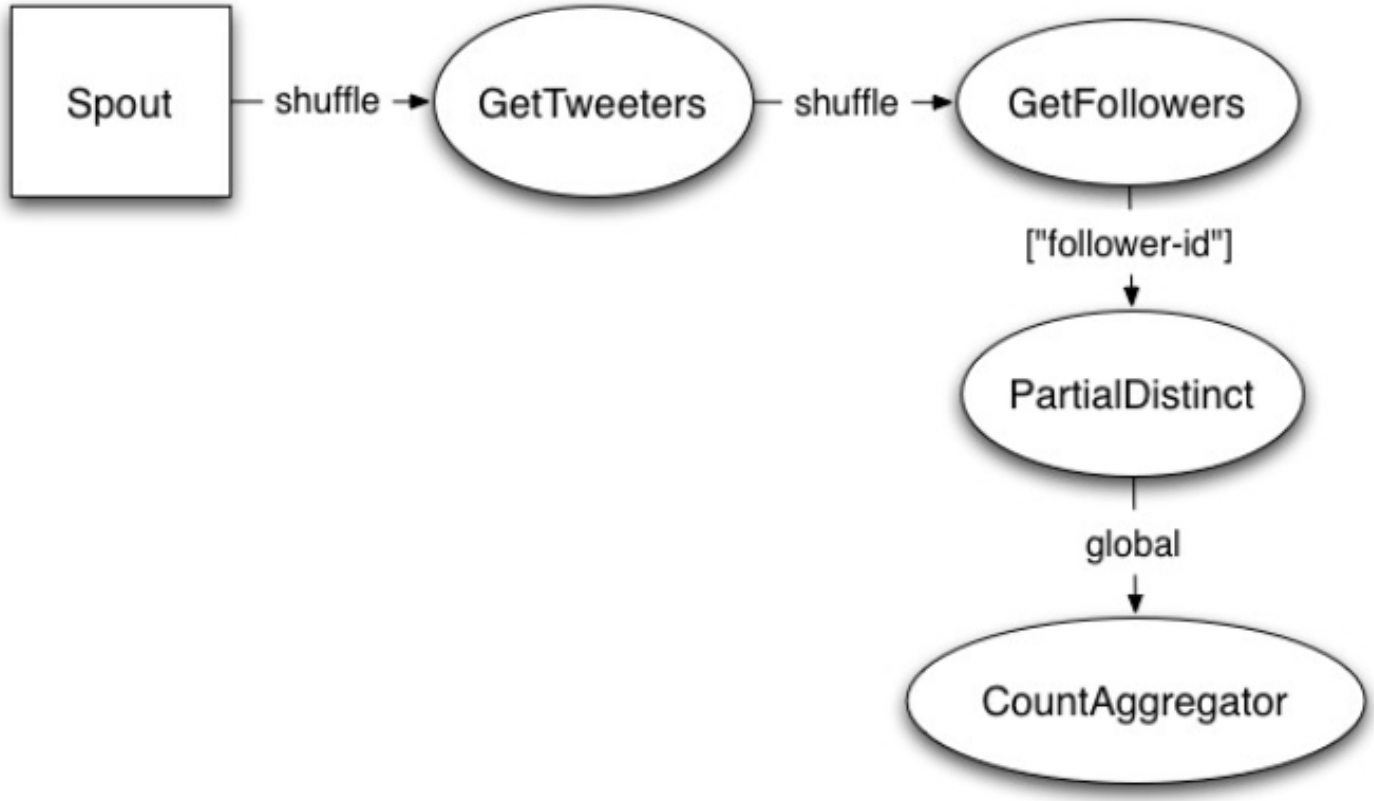
An Example using DRPC

- Computing “Reach” of a URL on the fly
  - “Reach” is the no. of UNIQUE people exposed to a given URL on Twitter

How to compute “Reach” :



A Storm Topology to Compute Reach




```
LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("reach");
builder.addBolt(new GetTweeters(), 3);
builder.addBolt(new GetFollowers(), 12)
        .shuffleGrouping();
builder.addBolt(new PartialUniquer(), 6)
        .fieldsGrouping(new Fields("id", "follower"));
builder.addBolt(new CountAggregator(), 2)
        .fieldsGrouping(new Fields("id"));
```

A Storm Topology to Compute Reach (cont'd)

```
public static class PartialUniquer implements IRichBolt, FinishedCallback {
    OutputCollector _collector;
    Map<Object, Set<String>> _sets = new HashMap<Object, Set<String>>();

    public void execute(Tuple tuple) {
        Object id = tuple.getValue(0);
        Set<String> curr = _sets.get(id);
        if(curr==null) {
            curr = new HashSet<String>();
            _sets.put(id, curr);
        }
        curr.add(tuple.getString(1));
        _collector.ack(tuple);
    }

    @Override
    public void finishedId(Object id) {
        Set<String> curr = _sets.remove(id);
        int count = 0;
        if(curr!=null) count = curr.size();
        _collector.emit(new Values(id, count));
    }
```

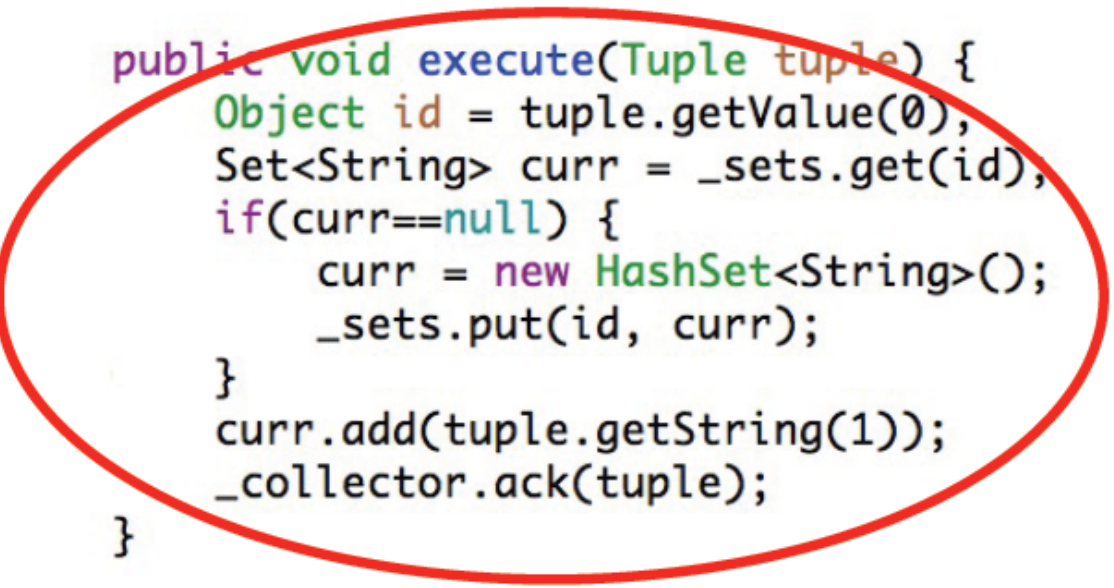
A Storm Topology to Compute Reach (cont'd)

```
public static class PartialUniquer implements IRichBolt, FinishedCallback {
    OutputCollector _collector;
    Map<Object, Set<String>> _sets = new HashMap<Object, Set<String>>();

    public void execute(Tuple tuple) {
        Object id = tuple.getValue(0);
        Set<String> curr = _sets.get(id);
        if(curr==null) {
            curr = new HashSet<String>();
            _sets.put(id, curr);
        }
        curr.add(tuple.getString(1));
        _collector.ack(tuple);
    }

    @Override
    public void finishedId(Object id) {
        Set<String> curr = _sets.remove(id);
        int count = 0;
        if(curr!=null) count = curr.size();
        _collector.emit(new Values(id, count));
    }
```

Keep set of followers for each request id in memory

A Storm Topology to Compute Reach (cont'd)

```
public static class PartialUniquer implements IRichBolt, FinishedCallback {
    OutputCollector _collector;
    Map<Object, Set<String>> _sets = new HashMap<Object, Set<String>>();

    public void execute(Tuple tuple) {
        Object id = tuple.getValue(0);
        Set<String> curr = _sets.get(id);
        if(curr==null) {
            curr = new HashSet<String>();
            _sets.put(id, curr);
        }
        curr.add(tuple.getString(1));
        _collector.ack(tuple);
    }

    @Override
    public void finishedId(Object id) {
        Set<String> curr = _sets.remove(id);
        int count = 0;
        if(curr!=null) count = curr.size();
        _collector.emit(new Values(id, count));
    }
```



Update followers set when receive a new follower

A Storm Topology to Compute Reach (cont'd)

```
public static class PartialUniquer implements IRichBolt, FinishedCallback {
    OutputCollector _collector;
    Map<Object, Set<String>> _sets = new HashMap<Object, Set<String>>();

    public void execute(Tuple tuple) {
        Object id = tuple.getValue(0);
        Set<String> curr = _sets.get(id);
        if(curr==null) {
            curr = new HashSet<String>();
            _sets.put(id, curr);
        }
        curr.add(tuple.getString(1));
        _collector.ack(tuple);
    }

    @Override
    public void finishedId(Object id) {
        Set<String> curr = _sets.remove(id);
        int count = 0;
        if(curr!=null) count = curr.size();
        _collector.emit(new Values(id, count));
    }
```

Emit partial count after receiving all followers for a request id

Trident

But What About State

- For most cases, state storage in Storm is left up to you (the programmer).
- If your Bolt goes down after accumulating 3 weeks of aggregated data that you have not stored any where -- too bad.

Enter Trident

- Trident is a high-level abstraction for doing Real-Time computing on the top of Storm
  - Similar to high-level batch processing tools like Pig or Cascading
- Provides Exactly-Once semantics like transactional topologies.
- In Trident, state is a first class citizen, but the exact implementation of state is up to you.
  - There are many pre-built connectors to various NoSQL stores like HBase
- Provides a high level API (similar to cascading for Hadoop)

Trident Example

```
TridentTopology topology =
        new TridentTopology();
TridentState wordCounts =
     topology.newStream("spo
        .each(new Fields("senter
new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new
MemoryMapState.Factory(), new Count(), new
Fields("count"))
        .parallelismHint(6);
```

Aggregates values and stores them.

Trident Example

```
public class Split extends BaseFunction {

   public void execute(TridentTuple tuple,
TridentCollector collector) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
   }
}
```

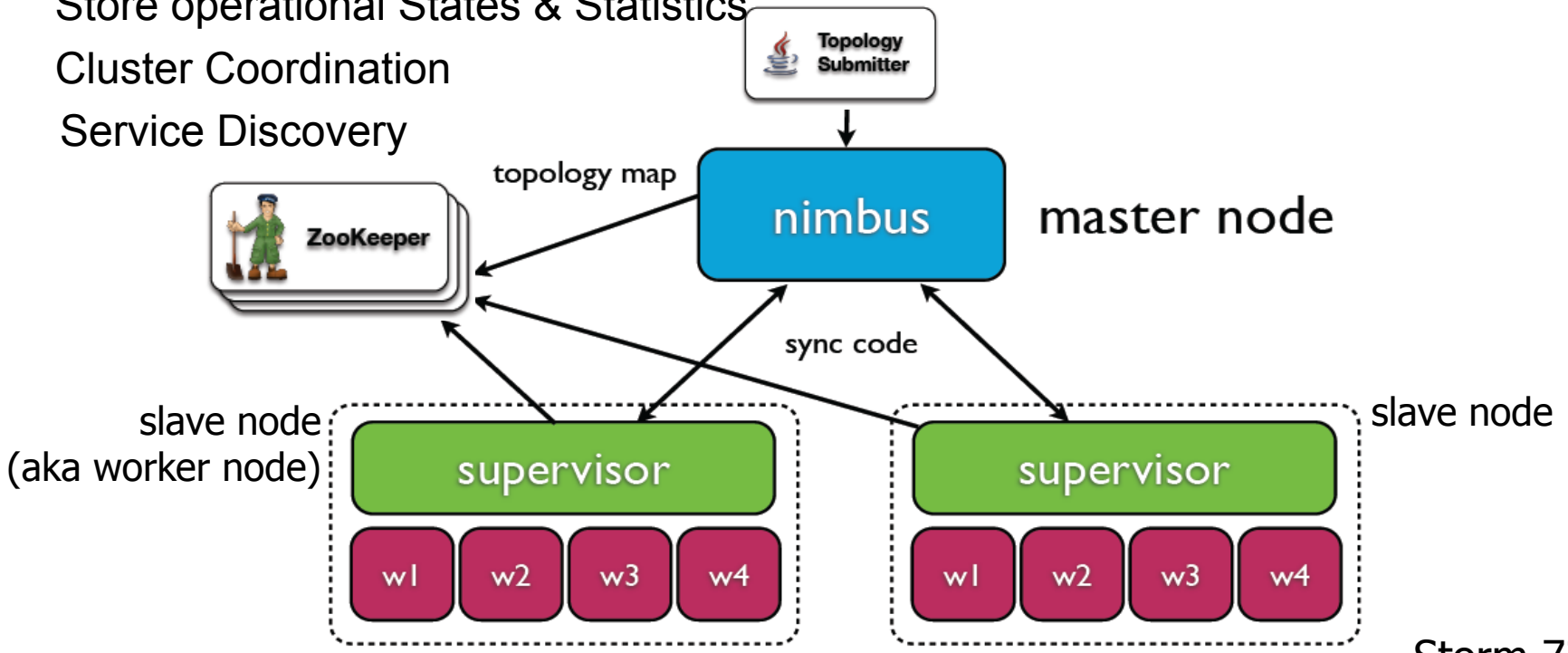
No Acking Required

A Summary on What Storm does

- Distribute Code and Configuration
- Robust Process Management
- Monitors Topologies and Assign Failed Tasks
- Provide Reliability by Tracking Tuple Tree
- Routing and Partitioning of Streams
- Serialization
- Fine-grained Performance Statistics/ Status of Topologies

Storm System Architecture

- Nimbus
  - Master node (like the Job Tracker in Hadoop ver1.0)
  - Manage Topologies ; Distribute Code ; Assign Tasks ; Monitor Failure
- Supervisor
  - Runs on Slave nodes (aka Worker nodes) ; listen to assignment and then launch & manage Worker (JVM) processes
  - Coordinate with Zookeeper for Fault-Tolerance/ Synchronization, etc
- Zookeeper
  - Store operational States & Statistics
  - Cluster Coordination
  - Service Discovery

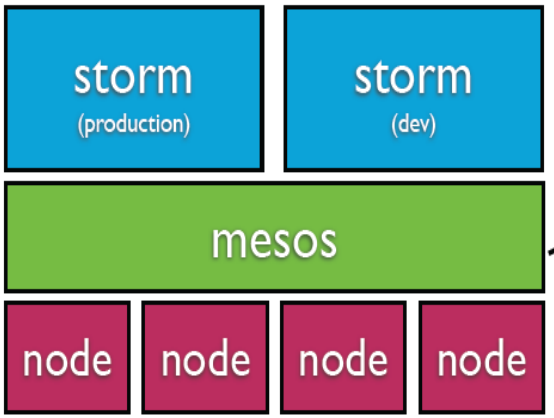


Comparison of Architecture: Hadoop v1 (MapReduce) vs. Storm

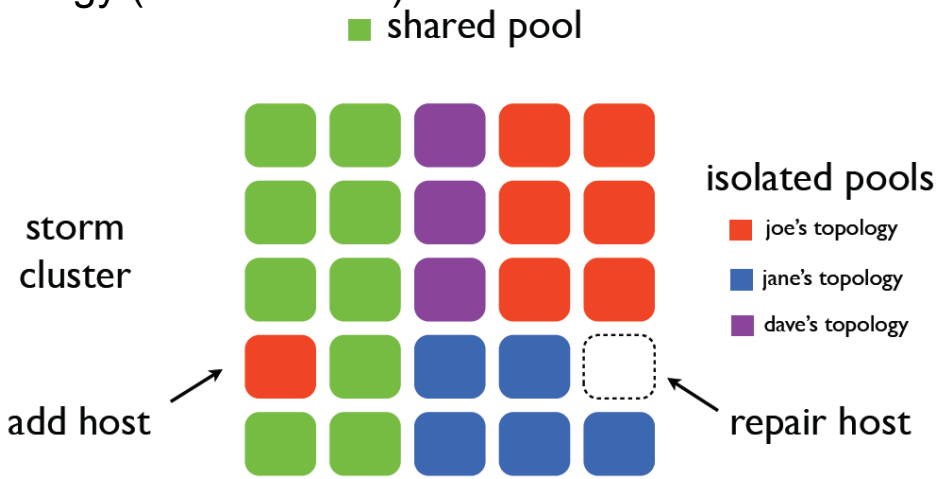
Hadoop v1	Storm	Functions in Storm
JobTracker	**Nimbus (only 1)**	▪ distributes code around cluster ▪ assigns tasks to machines/supervisors ▪ failure monitoring ▪ is fail-fast and stateless (you can "kill -9" it)
TaskTracker	**Supervisor (many)**	▪ listens for work assigned to its machine ▪ starts and stops worker processes as necessary based on Nimbus ▪ is fail-fast and stateless (you can "kill -9" it) ▪ shuts down worker processes with "kill -9", too
MR job	**Topology**	▪ processes messages forever (or until you kill it) ▪ a running topology consists of many worker processes spread across many machines

Different ways to run Storm over a Cluster

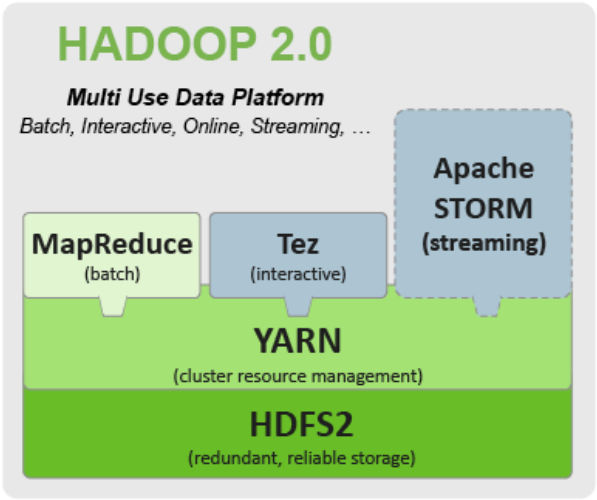
- Twitter runs multiple instances of Storm over Mesos
  - Multiple Topologies can be run on the same host (Shared Pool) or
  - Dedicated Set of hosts to run a single topology (Isolated Pool)

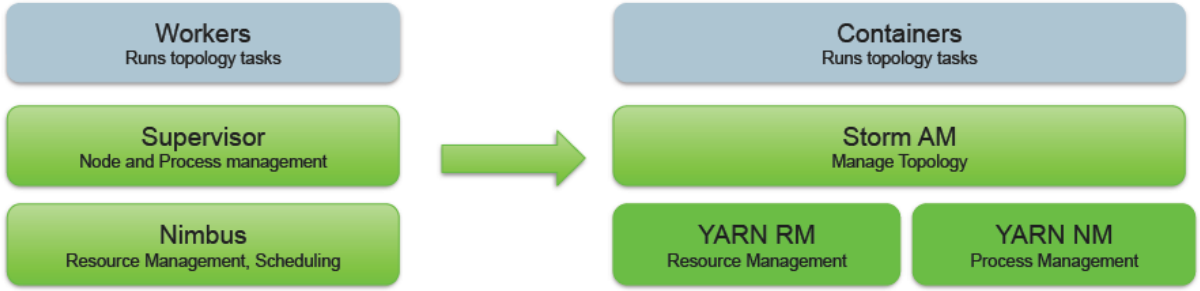


provides efficient resource isolation and sharing across distributed frameworks such as storm.



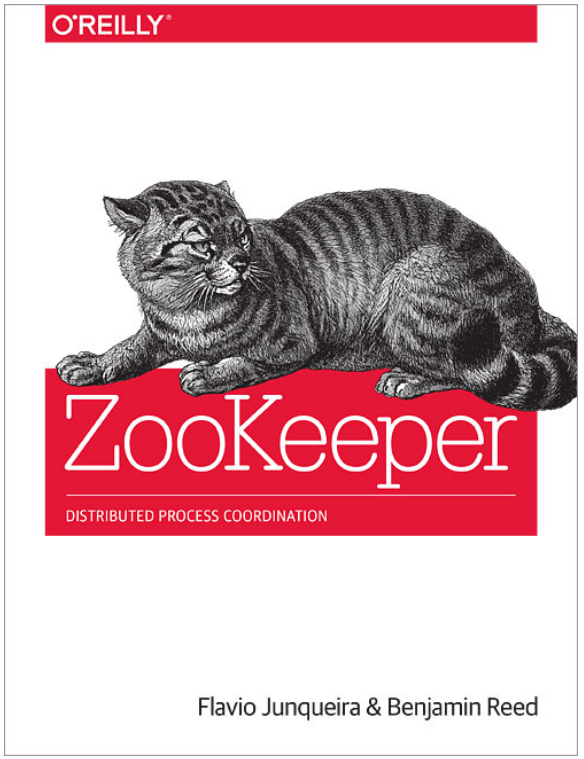
- Storm can also be run as an application (framework) over YARN:





Mapping Storm's architecture to YARN's resource management model

A Digression:
An Introduction to ZooKeeper (ZK)

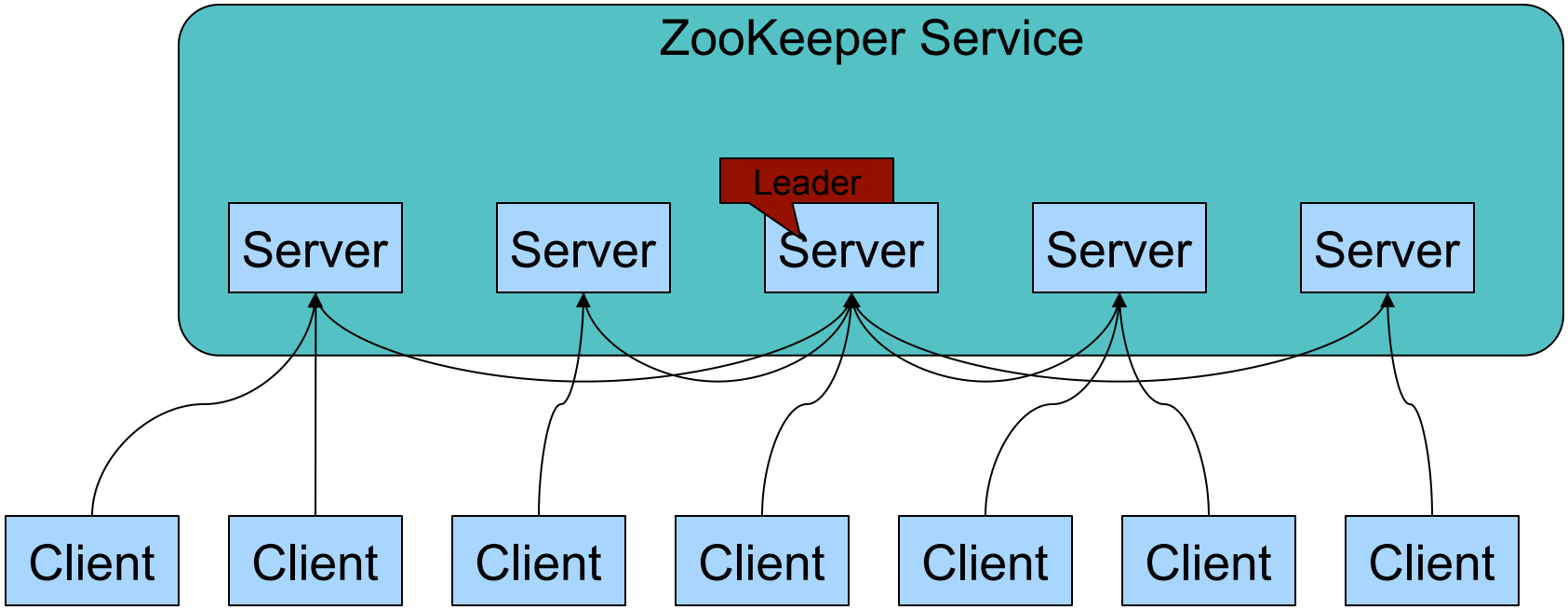


ZooKeeper – Distributed Process Coordination, by Flavio Junqueira, Benjamin Reed, Published by O'Reilly, Nov. 2013.

What is ZooKeeper?

- A highly available, scalable service to support Distributed configuration, Consensus, Group Membership,Leader Election, Naming, and Coordination
- Difficult to implement these kinds of services reliably
  - brittle in the presence of change
  - difficult to manage
  - different implementations lead to management complexity when the applications are deployed
- Originally developed by Yahoo! ; subsequently under Apache license: http://zookeeper.apache.org/

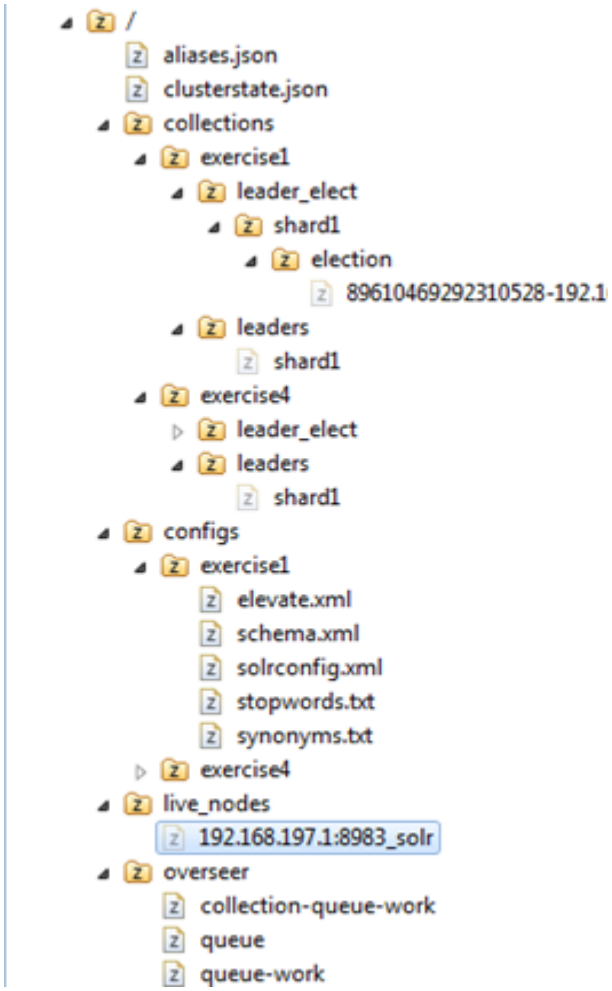
ZooKeeper Service



- All servers store a copy of the data, logs, snapshots on disk and use an in memory database
- A leader is elected at startup
- Followers service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change

The Data Model of ZooKeeper (in form of a Special distributed filesystem tracked by ALL servers in a ZooKeeper service)

- Hierarchical Namespace
- Each node is called “znode”
- UNIX-like notation for path
- Each znode has data(stores data in byte[] array) and can have children
- Znode
  - Key Object for keeping/realizing the Consistent Distributed State info
  - Maintains “Stat” structure with version of data changes , ACL changes and timestamp
  - Version number increases with each change



Zookeeper Service

- Watch Mechanism
  - Get notification
  - One time triggers
- Other properties of Znode
  - Znode doesn't not design for data storage, instead it store meta-data or configuration
  - Can store information like timestamp version
- Session
  - A connection to server from client is a session
  - Timeout mechanism

Client API

- Create(path, data, flags)
- Delete(path, version)
- Exist(path, watch)
- getData(path, watch)
- setData(path, data, version)
- getChildren(path, watch)
- Sync(path)
- Two version synchronous and asynchronous

ZooKeeper Properties

- File API (for the Special distributed filesystem) without partial reads/writes
  - Simple **wait free** data objects organized hierarchically as in file systems.
- Per Client guarantee of FIFO execution of requests
- Linearizability for all requests that change the Zookeeper state
- Built using the Zookeeper Atomic Broadcast (ZAB) algorithm, a totally ordered broadcast protocol (inspired by, but different from the Paxos algorithm) http://research.yahoo.com/files/ladis08.pdf
- 2F+1 servers can tolerate f crash failures

Any Guarantees?

1. Clients will never detect old data.
2. Clients will get notified of a change to data they are watching within a bounded period of time.
3. All requests from a client will be processed in order.
4. All results received by a client will be consistent with results received by all other clients.

Protocol Guarantees

1) **Sequential Consistency** - Updates from a client will be applied in the order that they were sent.

2) **Atomicity** - Updates either succeed or fail. No partial results.

3) **Single System Image** - A client will see the same view of the service regardless of the server that it connects to.

4) **Reliability** - Once an update has been applied, it will persist from that time forward until a client overwrites the update.

5) **Timeliness** - The clients view of the system is guaranteed to be up-to-date within a certain bound. Either system changes will be seen by a client within this bound, or the client will detect a service outage.

Examples of ZooKeeper primitives

- Configuration Management
  - For dynamic configuration propose
  - Simplest way is to make up a znode c for saving configuration.
  - Other processes set the watch flag on c
  - The notification just indicate there is a update without telling how many time updates occurs

Examples of ZooKeeper primitives

- Rendezvous
  - Configuration of the system may not be sure at the beginning
  - Create a znode r for this problem
  - When master starts, it will fill the configuration in r
  - Workers watch node r
  - Set to ephemeral node

Examples of ZooKeeper primitives

- Group Membership
  - Create a znode g
  - Each process create a znode under g in ephemeral mode
  - Watch g for group information

Examples of ZooKeeper primitives

- Simple Lock
  - Create a znode l for locking
  - If one gets to create l he gets the lock
  - Others who fail to create watch l
  - Problems: herd effect

Examples of ZooKeeper primitives

- Simple Lock without herd effect

Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```

Examples of ZooKeeper primitives

- Read/Write Lock

Write Lock

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock

```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

Examples of ZooKeeper primitives

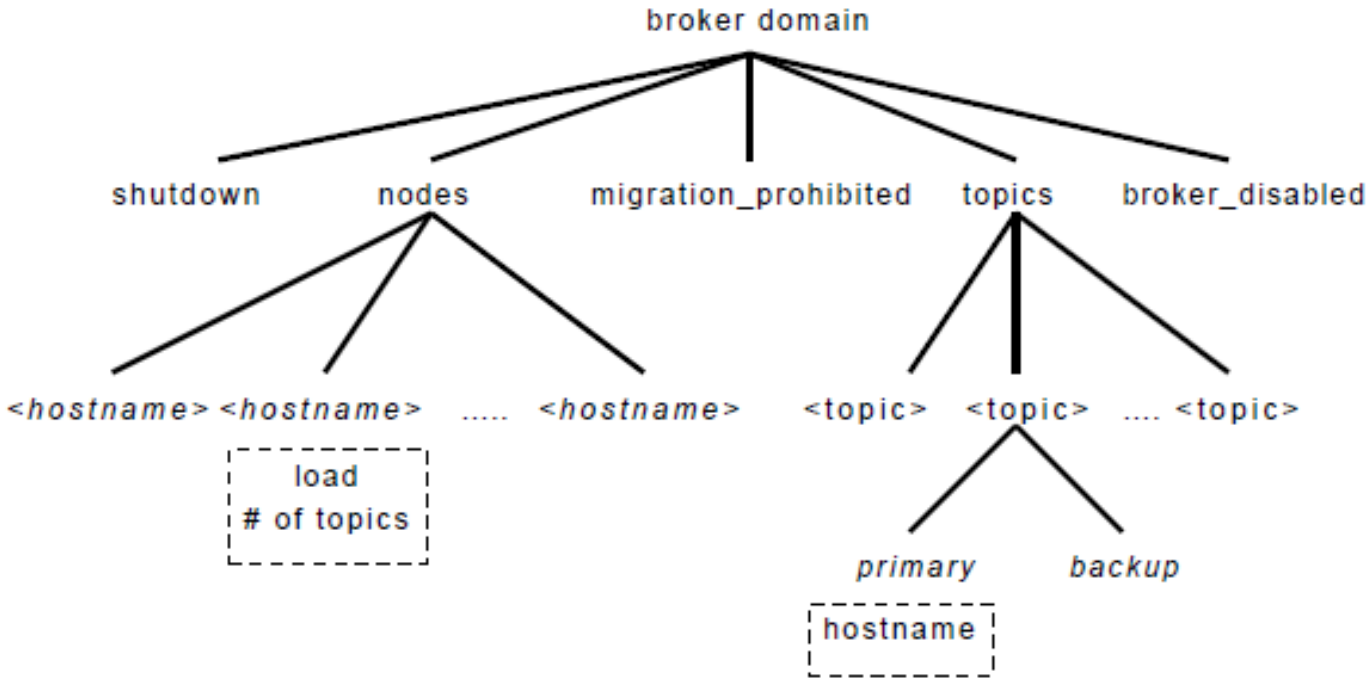
- Double Barrier
  - To synchronize the beginning and the end of compuation
  - Create a znode b, and every process needs to register on it, by adding a znode under b
  - Set a threshold that start the process

ZooKeeper Application Example

- Fetching Service
  - Using ZooKeeper for recovering from failure of masters
  - Configuration metadata and leader election

ZooKeeper Application Example

- Yahoo Message Broker
  - A distributed publish-subscribe system

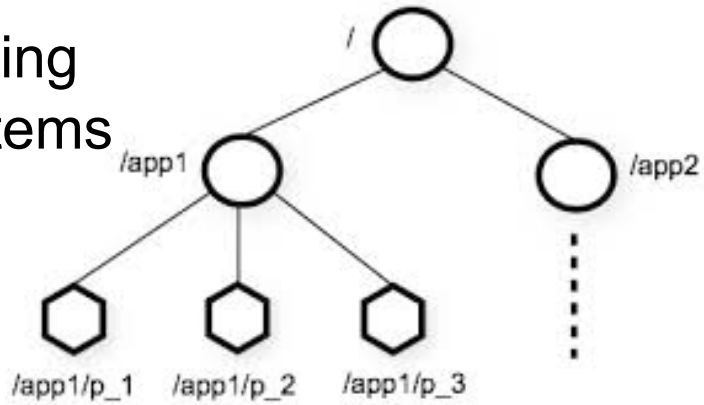


Additional ZooKeeper Use cases

- Configuration Management
  - Cluster member nodes Bootstrapping configuration from a central source
- Distributed Cluster Management
  - Node Join/Leave
  - Node Status in real time
- Naming Service – e.g. DNS
- Distributed Synchronization – locks, barriers
- Leader election
- Centralized and Highly reliable Registry

Summary of ZooKeeper (end of Digression)

- An Open source, High Performance coordination service for distributed applications
- Centralized service for
  - Configuration Management
  - Locks and Synchronization for providing coordination between distributed systems
  - Naming service (Registry)
  - Group Membership
- Features
  - Hierarchical namespace
  - Provide Watcher on a znode
  - Allow to form a cluster of nodes
- Support a "large" volume of request for data retrieval and update



Source : http://zookeeper.apache.org

Storm architecture: ZooKeeper

- Storm requires **ZooKeeper**
  - 0.9.2+ uses ZK 3.4.5
  - Storm typically puts less load on ZK than Kafka (but ZK is still a bottleneck), but caution: often you have many more Storm nodes than Kafka nodes
- ZooKeeper
  - NOT used for message passing, which is done via Netty in 0.9
  - Used for coordination purposes, and to store state and statistics
    - Register + discover Supervisors, detect failed nodes, …
      - Example: To add a new Supervisor node, just start it.
    - This allows Storm's components to be stateless. "kill -9" away!
      - Example: Supervisors/Nimbus can be restarted without affecting running topologies.
  - Used for heartbeats
    - Workers heartbeat the status of child executor threads to Nimbus via ZK.
    - Supervisor processes heartbeat their own status to Nimbus via ZK.
  - Store recent task errors (deleted on topology shutdown)

Storm architecture: fault tolerance

- What happens when **Nimbus** dies (master node)?
  - If Nimbus is run under process supervision as recommended (e.g. via supervisord), it will restart like nothing happened.
  - While Nimbus is down:
    - Existing topologies will continue to run, but you cannot submit new topologies.
    - Running worker processes will not be affected. Also, Supervisors will restart their (local) workers if needed. However, failed tasks will not be reassigned to other machines, as this is the responsibility of Nimbus.
- What happens when a **Supervisor** dies (slave node)?
  - If Supervisor run under process supervision as recommended (e.g. via supervisord), will restart like nothing happened.
  - Running worker processes will not be affected.
- What happens when a **worker process** dies?
  - It's parent Supervisor will restart it. If it continuously fails on startup and is unable to heartbeat to Nimbus, Nimbus will reassign the worker to another machine.

Storm hardware specs

- ZooKeeper
  - Preferably use dedicated machines because ZK is a bottleneck for Storm
    - 1 ZK instance per machine
    - Using VMs may work in some situations. Keep in mind other VMs or processes running on the shared host machine may impact ZK performance, particularly if they cause I/O load. (source)
  - I/O is a bottleneck for ZooKeeper
    - Put ZK storage on its own disk device
    - SSD's dramatically improve performance
    - Normally, ZK will sync to disk on every write, and that causes two seeks (1x for the data, 1x for the data log). This may add up significantly when all the workers are heartbeating to ZK. (source)
    - Monitor I/O load on the ZK nodes
  - Preferably run ZK ensembles with nodes >= 3 in production environments so that you can tolerate the failure of 1 ZK server (incl. e.g. maintenance)

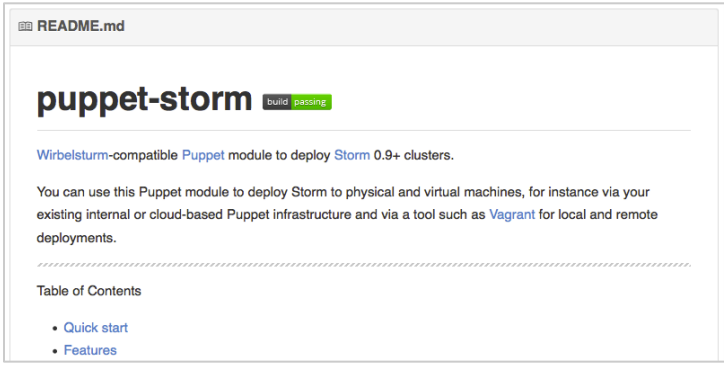
Storm hardware specs

- Nimbus aka master node
  - Comparatively little load on Nimbus, so a medium-sized machine suffices
  - EC2 example: m1.xlarge @ $0.27/hour
  - Check monitoring stats to see if the machine can keep up

Storm hardware specs

- Storm Slave nodes (aka Worker nodes)
  - Exact specs depend on anticipated usage – e.g. CPU heavy, I/O heavy, …
    - CPU heavy: e.g. machine learning
    - CPU light: e.g. rolling windows, pre-aggregation (here: get more RAM)
  - CPU cores
    - More is usually better – the more you have the more threads you can support (i.e. parallelism). And Storm potentially uses a *lot* of threads.
  - Memory
    - Highly specific to actual use case
    - Considerations: #workers (= JVMs) per node? Are you caching and/or holding in-memory state?
  - Network: 1GigE
    - Use bonded NICs or 10GigE if needed
  - EC2 examples: c1.xlarge @ $0.36/hour, c3.2xlarges @ $0.42/hour

Deploying Storm


README.md

puppet-storm build passing

Wirbelsturm-compatible Puppet module to deploy Storm 0.9+ clusters.

You can use this Puppet module to deploy Storm to physical and virtual machines, for instance via your existing internal or cloud-based Puppet infrastructure and via a tool such as Vagrant for local and remote deployments.

Table of Contents

- Quick start
- Features

- Puppet module
  - https://github.com/miguno/puppet-storm
  - Hiera-compatible, rspec tests, Travis CI setup (e.g. to test against multiple versions of Puppet and Ruby, Puppet style checker/lint, etc.)
- RPM packaging script for RHEL 6
  - https://github.com/miguno/wirbelsturm-rpm-storm
  - Digitally signed by yum@michael-noll.com
  - RPM is built on a Wirbelsturm-managed build server
- Consider Wirbelsturm for 1-click off-the-shelf cluster setups.

Deploying Storm

- An example for a Storm slave node

```
---
classes:
  - storm::logviewer
  - storm::supervisor
  - supervisor

## Custom Storm settings
storm::zookeeper_servers:
  - 'zookeeper1'
storm::logviewer_childopts:  '-Xmx128m -Djava.net.preferIPv4Stack=true'
storm::nimbus_host: 'nimbus1'
storm::supervisor_childopts: '-Xmx256m  -Djava.net.preferIPv4Stack=true'
storm::worker_childopts:     '-Xmx1024m -Djava.net.preferIPv4Stack=true'
storm::supervisor_slots_ports:
  - 6700
  - 6701
  - 6702
  - 6703
```

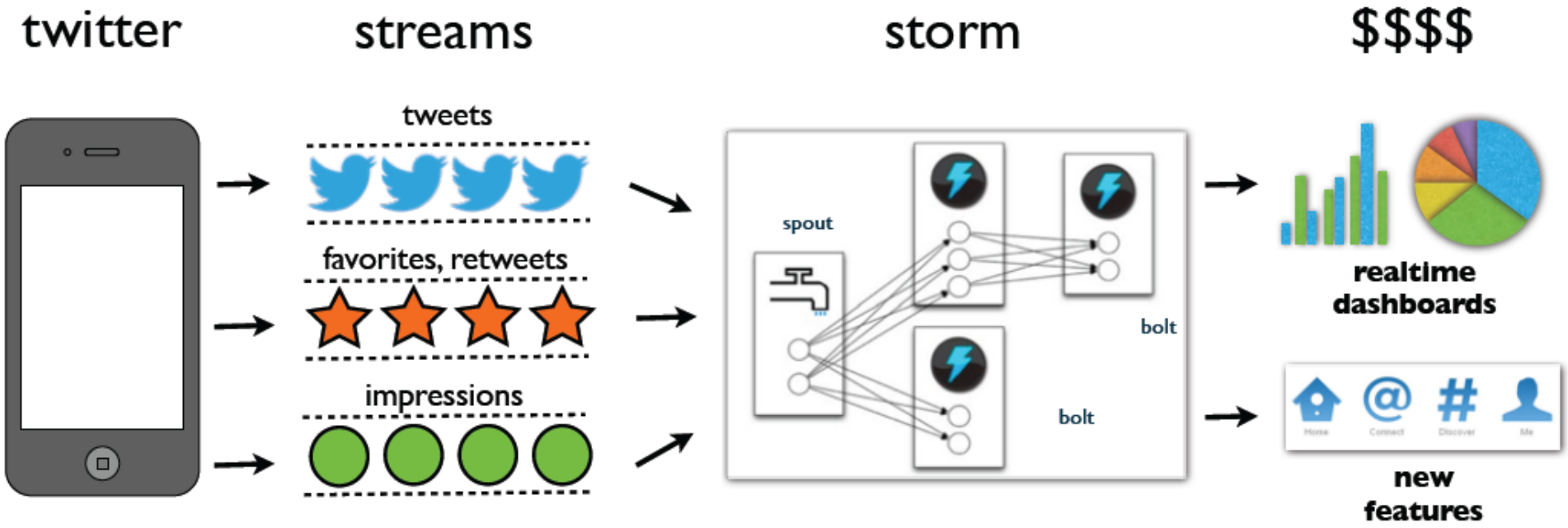
Operating Storm

- Typical operations tasks include:
  - Monitoring topologies for Performance and Scalability (P&S) : “Don’t let our pipes blow up!”
    - Tackling P&S in Storm is a joint Ops-Dev effort.
  - Adding or removing slave nodes, i.e. nodes that run Supervisors
  - Apps management: new topologies, swapping topologies, …
- See Ops-related references at the end of this part

Storm security

- Original design was not created with security in mind.
- Security features are now being added, e.g. from Yahoo!'s fork.
- State of security in Storm 0.9.x:
  - No authentication, no authorization.
  - No encryption of data in transit, i.e. between workers.
  - No access restrictions on data stored in ZooKeeper.
  - Arbitrary user code can be run on nodes if Nimbus' Thrift port is not locked down.
  - This list goes on.
- Further details plus recommendations on hardening Storm:
  - https://github.com/apache/incubator-storm/blob/master/SECURITY.md

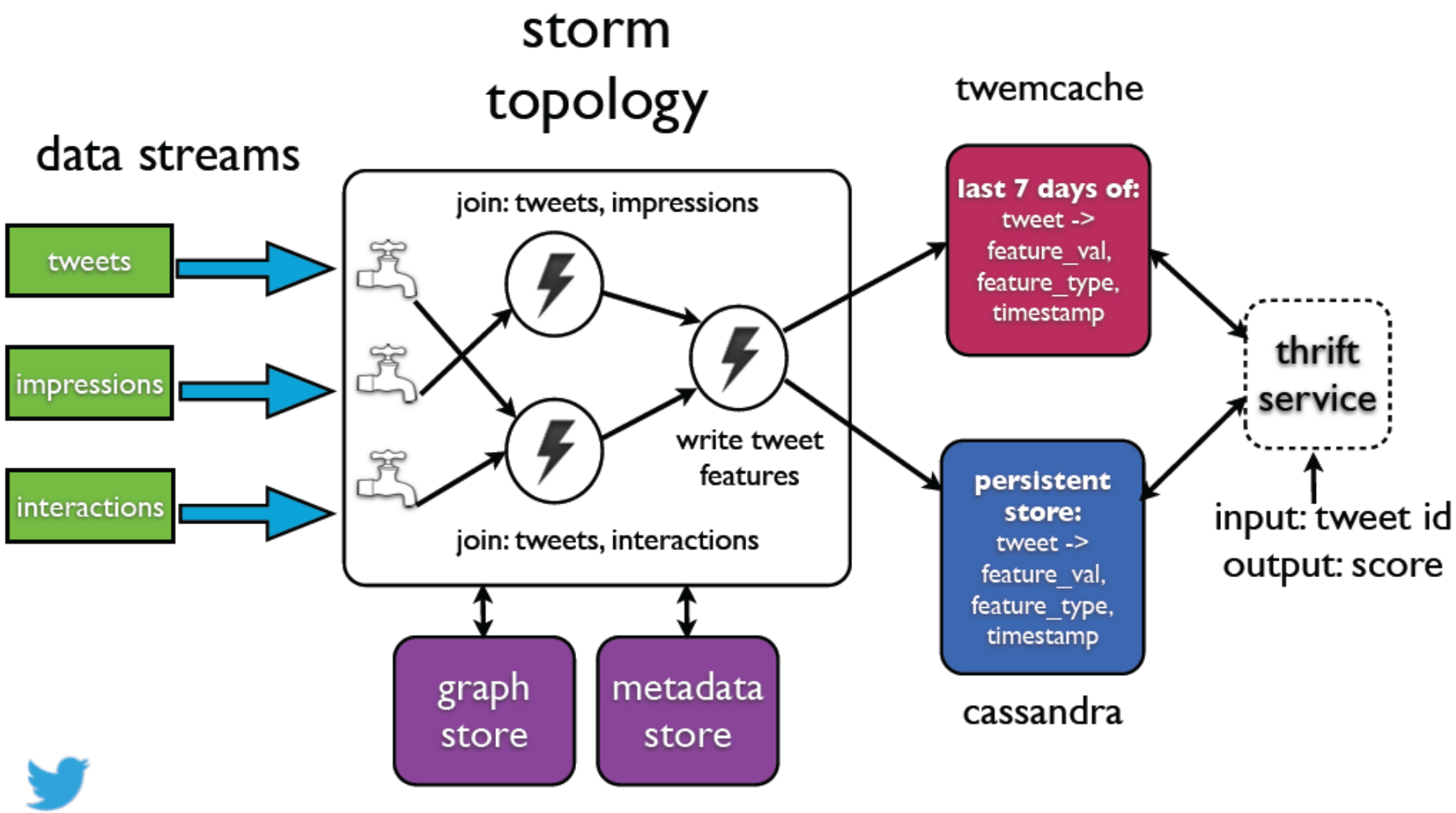
Stream Processing Applications at Twitter



Use Cases at Twitter

- Discovery of Emerging Topics and Stories
- Online Learning of Tweet Features for Search result Ranking
- Real-time Analytics for Ads
- Internal Log processing

Tweet Scoring Pipeline

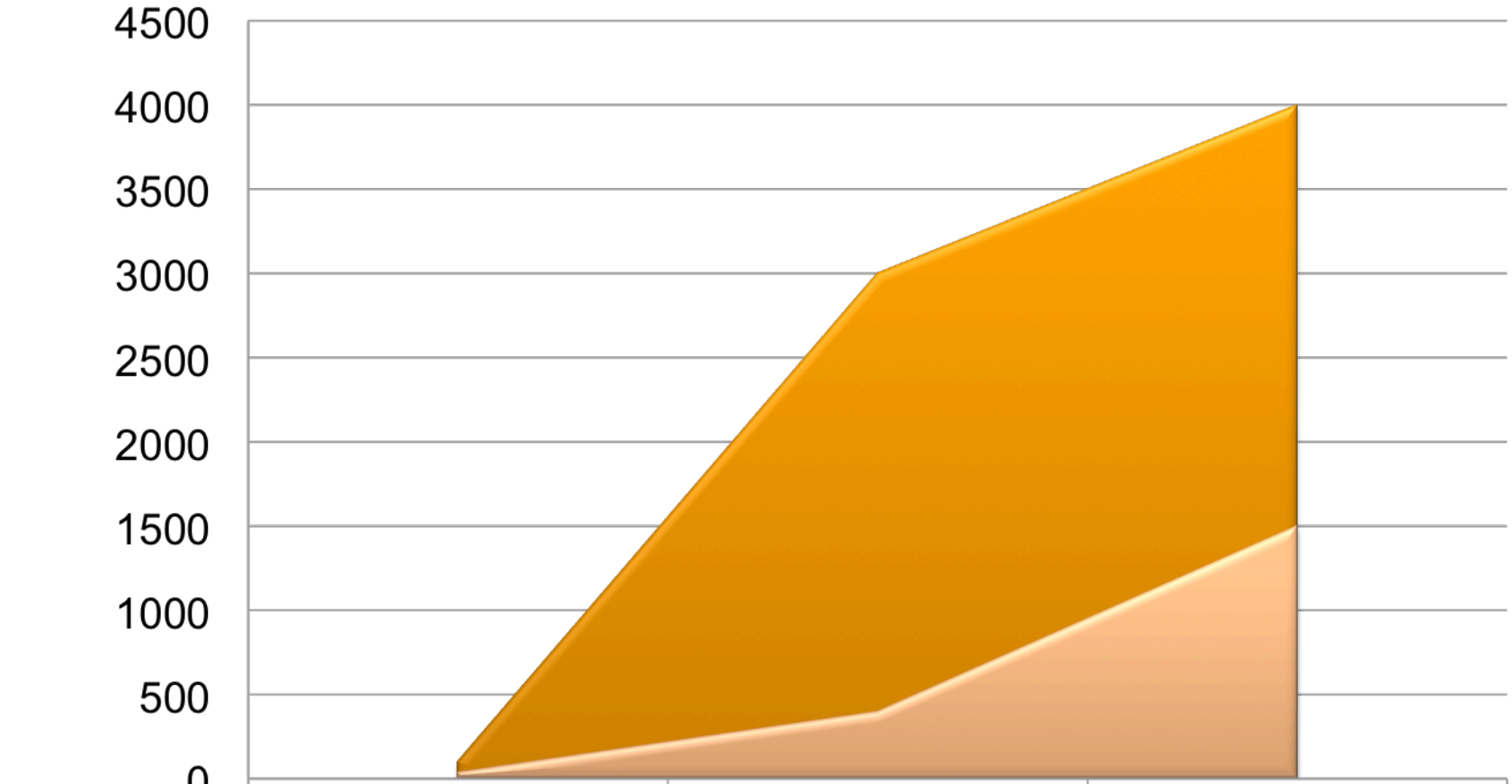


Storm adoption and use cases

https://github.com/nathanmarz/storm/wiki/Powered-By

- **Twitter:** personalization, search, revenue optimization, …
  - 200 nodes, 30 topos, 50B msg/day, avg latency <50ms, Jun 2013
- **Yahoo:** user events, content feeds, and application logs
  - 320 nodes (YARN), 130k msg/s, June 2013
- **Spotify**: recommendation, ads, monitoring, …
  - v0.8.0, 22 nodes, 15+ topos, 200k msg/s, Mar 2014
- Alibaba, Cisco, Flickr, PARC, WeatherChannel, …
  - Netflix is looking at Storm and Samza, too.

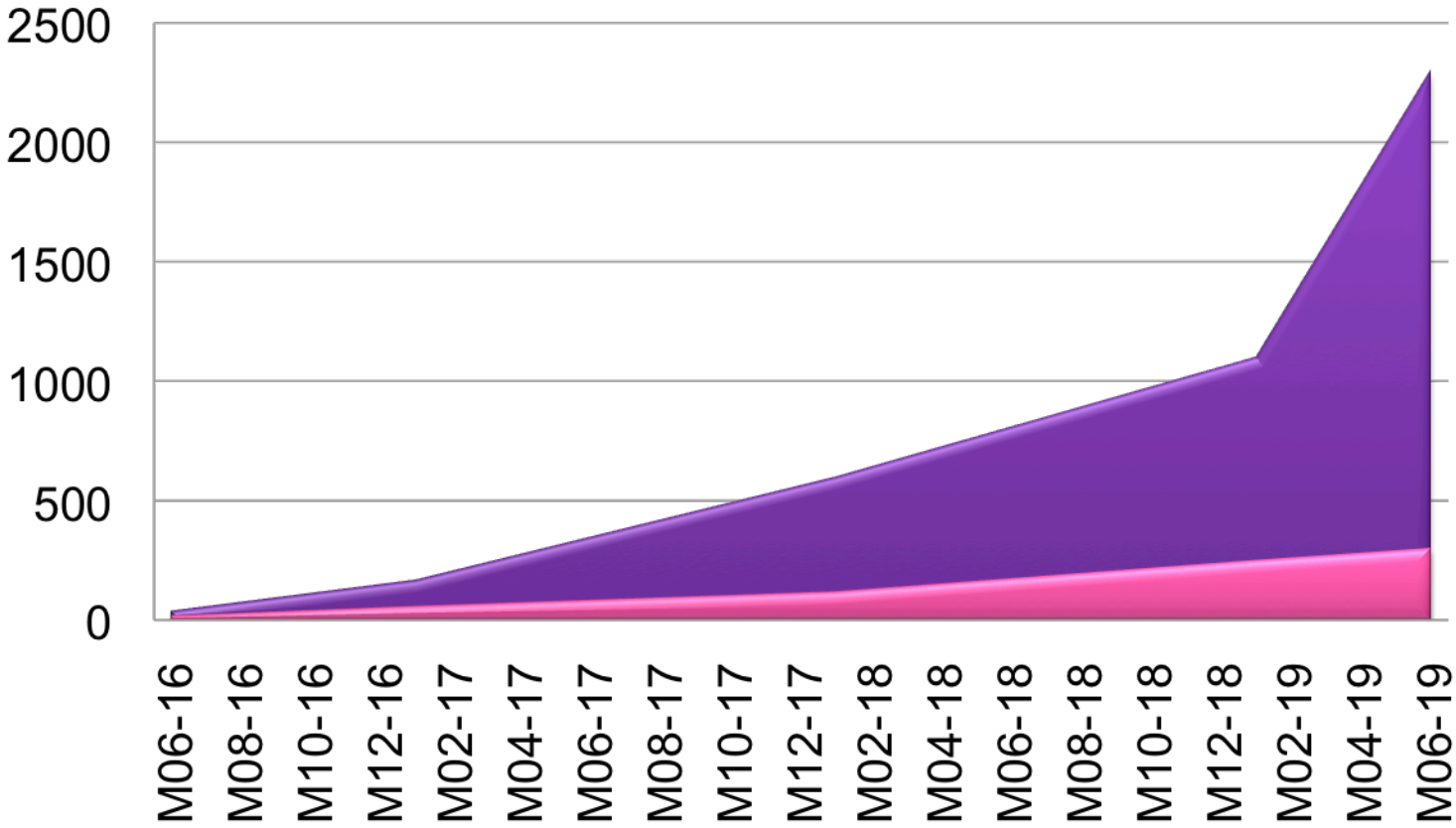
Largest Topology Growth at Yahoo



	2013	2014	2015
Executors	100	3000	4000
Workers	40	400	1500

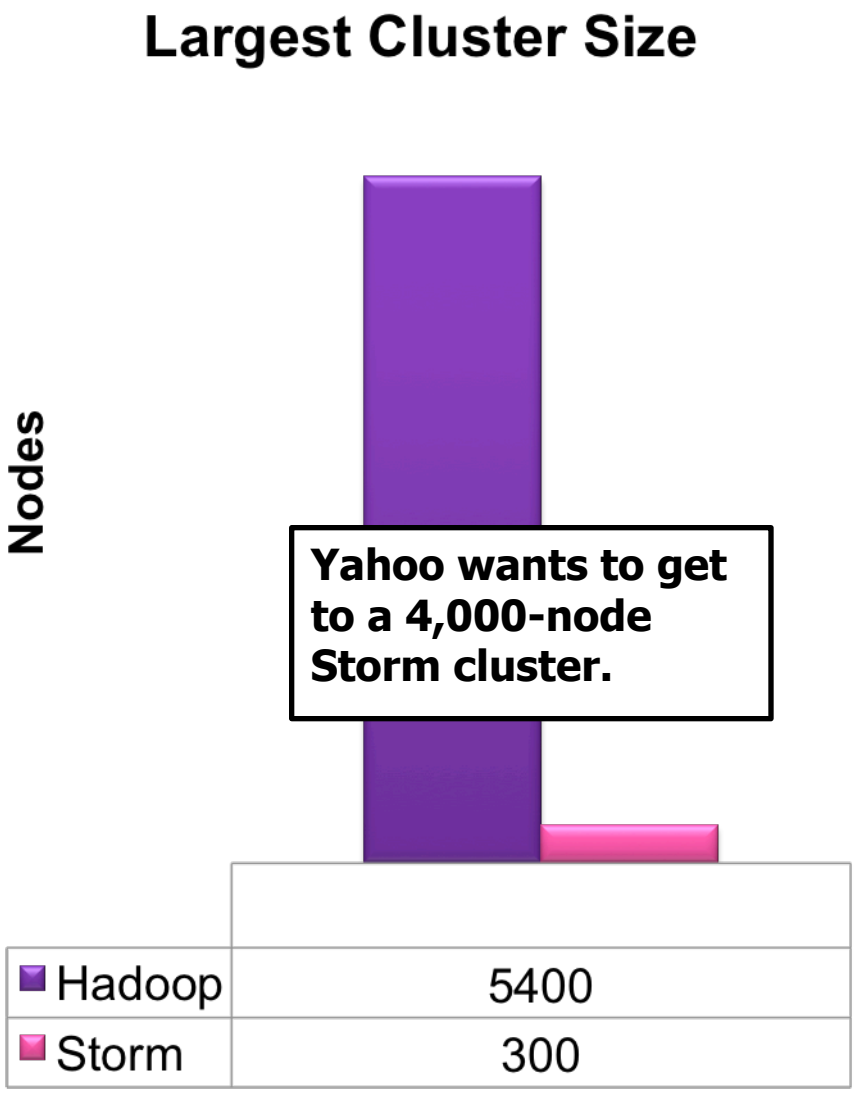
background: https://www.flickr.com/photos/68942208@N02/16242761551

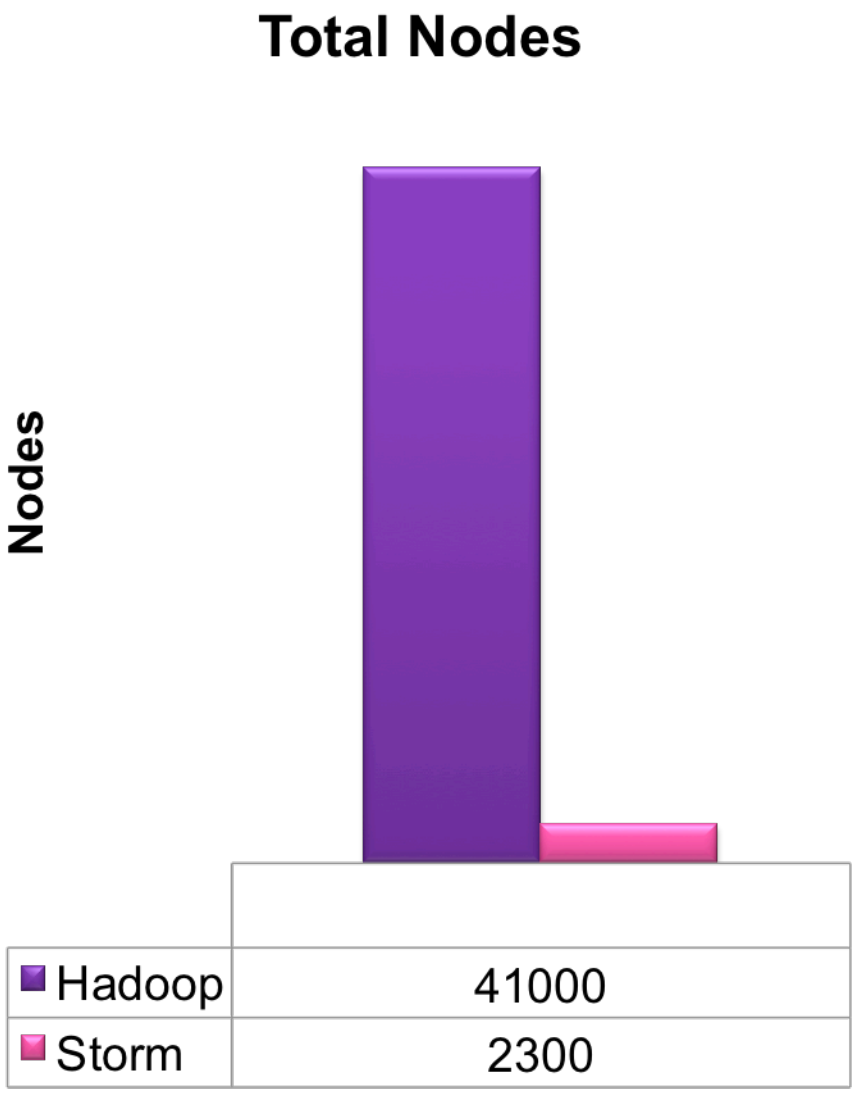
Cluster Growth at Yahoo



	M06-16	M01-17	M01-18	M01-19	M06-19
Total Nodes	40	170	600	1100	2300
Largest Cluster	20	60	120	250	300

But still have a ways to go before comparable to Yahoo's own Hadoop deployment





Initial Deployment of Storm within Yahoo !

- Mid 2011:
  - Storm is released as open source
- Early 2012:
  - Yahoo evaluation begins
  - https://github.com/yahoo/storm-perf-test
- Mid 2012:
  - Purpose built clusters 10+ nodes
- Early 2013:
  - 60-node cluster, largest topology 40 workers, 100 executors
  - ZooKeeper config -Djute.maxbuffer=4194304
- May 2013:
  - Netty messaging layer
  - http://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty
- Oct 2013:
  - ZooKeeper heartbeat timeout checks

More recent developments of Apache Storm

- Late 2013:
  - Storm enters Apache Incubator
- Early 2014 in Yahoo! :
  - 250-node cluster, largest topology 400 workers, 3,000 executors
- June 2014:
  - STORM-376 – Compress ZooKeeper data
  - STORM-375 – Check for changes before reading data from ZooKeeper
- Sep 2014:
  - Storm becomes an Apache Top Level Project
- Early 2015:
  - STORM-632 Better grouping for data skew
  - STORM-634 Thrift serialization for ZooKeeper data.
  - Yahoo deployed a 300-node cluster (Tested 400 nodes, 1,200 theoretical maximum)
  - Largest topology 1,500 workers, 4,000 executors
- June 2015:
  - Twitter announced the decommissioning of Storm ; replaced by Heron which adopts the same abstraction and 100% API-compatible with Storm:
    - http://blog.acolyer.org/2015/06/15/twitter-heron-stream-processing-at-scale/
    - Refer to the Heron paper in ACM SIGMOD 2015 for its technical details

Scalability Bottlenecks of Storm

State Storage (ZooKeeper):

- Limited to disk write speed (80MB/sec typically)
- Scheduling
  O(num_execs * resched_rate)
- Supervisor
  O(num_supervisors * hb_rate)
- Topology Metrics (worst case)
  O(num_execs * num_comps * num_streams * hb_rate)

On one 240-node Yahoo Storm cluster, ZK writes 16 MB/sec, about 99.2% of that is worker heartbeats

Theoretical Limit:
80 MB/sec / 16 MB/sec * 240 nodes = 1,200 nodes

Solution: Pacemaker

An alternative Heartbeat server

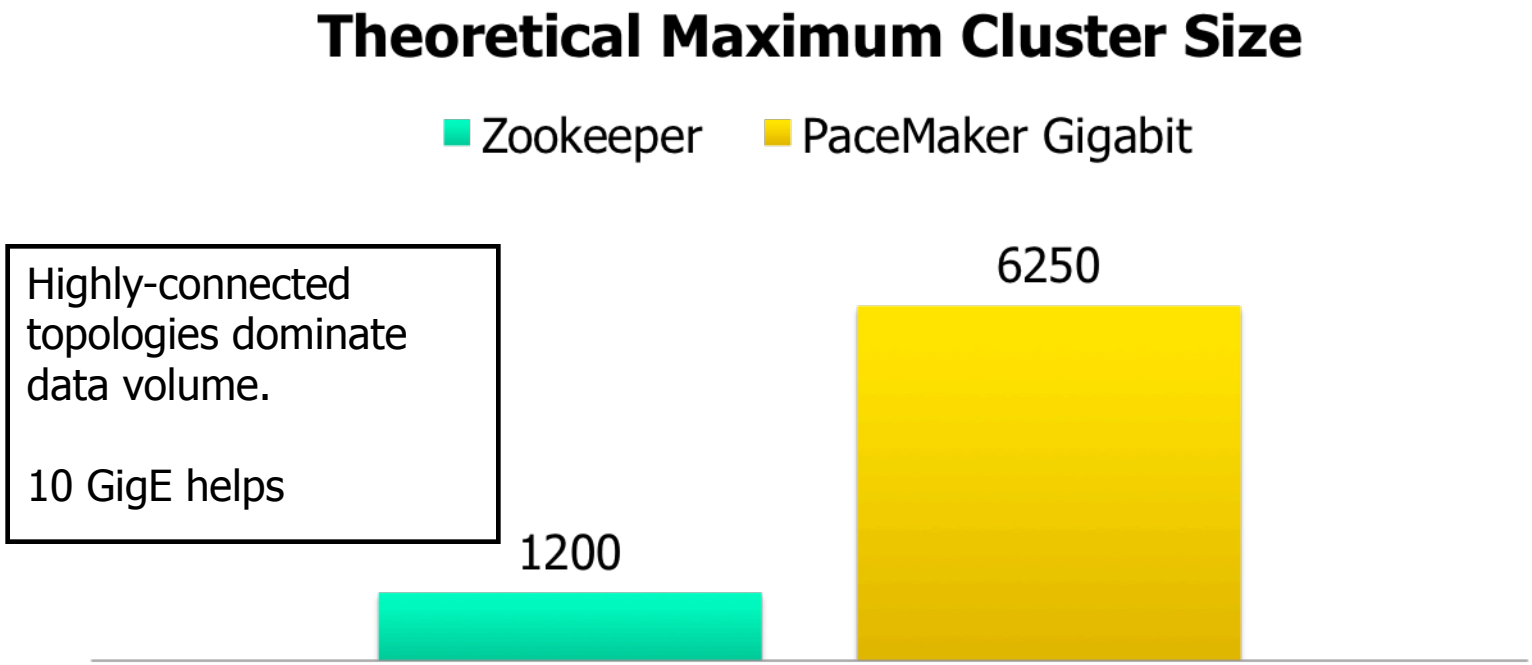
Simple Secure In-Memory Store for Worker Heartbeats.

- Removes Disk Limitation
- Writes Scale Linearly

(but nimbus still needs to read it all, ideally in 10 sec or less)

240 node cluster's complete HB state is 48MB, Gigabit is about 125 MB/s

10 s / (48 MB / 125 MB/s) * 240 nodes = 6,250 nodes



Scalability Bottlenecks of Storm (cont'd)

All raw data serialized, transferred to UI, de-serialized and aggregated per page load

Our largest topology uses about 400 MB in memory

Aggregate stats for UI/REST in Nimbus

- 10+ min page load to 7 seconds

DDOS on Nimbus for jar download

Distributed Cache/Blob Store (STORM-411)

- Pluggable backend with HDFS support

Scalability Bottlenecks of Storm (cont'd)

Storm round-robin scheduling

- R-1/R % of traffic will be off rack where R is the number of racks
- N-1/N % of traffic will be off node where N is the number of nodes
- Does not know when resources are full (i.e. network)

Possible Solution: Resource & Network Topography Aware Scheduling

One slow node slows the entire topology.

Load Aware Routing (STORM-162)
Intelligent network aware routing

Other Competing Stream Processing Systems…

- Heron (Twitter)
  - Same User Programming model and API, differ mostly in the under-the-hood system realization/ implementation, e.g.
    - Written in C++ instead of Closure
    - Better separation and scheduling of tasks, executors in JVM(s) for different components of the same/ different topologies to facilitate debugging
    - Backpressure-based congestion control of dataflow within a topology
- Google Cloud Dataflow
  - Open Source API, BUT NOT implementation
  - Scalability needed to be tested/ stressed
  - Great stream processing concepts
- Spark Streaming (BDAS of Berkeley/Databricks)
  - Micro-batch processing instead of true real-time streaming
- Apache Apex (DataTorrent), Flink, Samza (LinkedIn), Amazon's Kinesis, etc

Open Research Issues of Storm

Scheduling (Especially on Large Clusters)

- Currently round robin (We should be able to do better)
- Take into account resource utilization (Network)
- Locality with collocated services (Is there any advantage to running storm on the same nodes as HBase and/or Kafka?)
- What about better scheduling for Storm on Yarn?
- When is it worth it to kill a worker because a better location is available? (automatic rebalance)
  - Slow node detection (12 ms)

How can one grow/shrink a topology dynamically

- How to handle the different shuffles? Do we kill everything and start over or is there a better way?
- When should we grow or shrink?
- What do we do if there are no free resources and we need to grow?
- Or even more difficult can we upgrade a topology in place without killing processes?

Resource Isolation/Utilization

- Isolation is handled currently by creating a mini-cluster (whole nodes) for a single topology (so utilization suffers).
- Can we get better utilization without letting isolation suffer? cgroups/Docker. What about predictability on heavily used vs. lightly used clusters? (12ms again)
- What about if I collocate this batch on Hadoop?

Higher level APIs

- Streaming SQL (Spark streaming has it already)
- Pig on Storm (This is happening)

Statistical Analytics Systems

Sketches: Imagine compressing several terabytes of data into a few MB, and being able to do complex computational processing on them with a known maximum error.

http://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/

Online Machine Learning

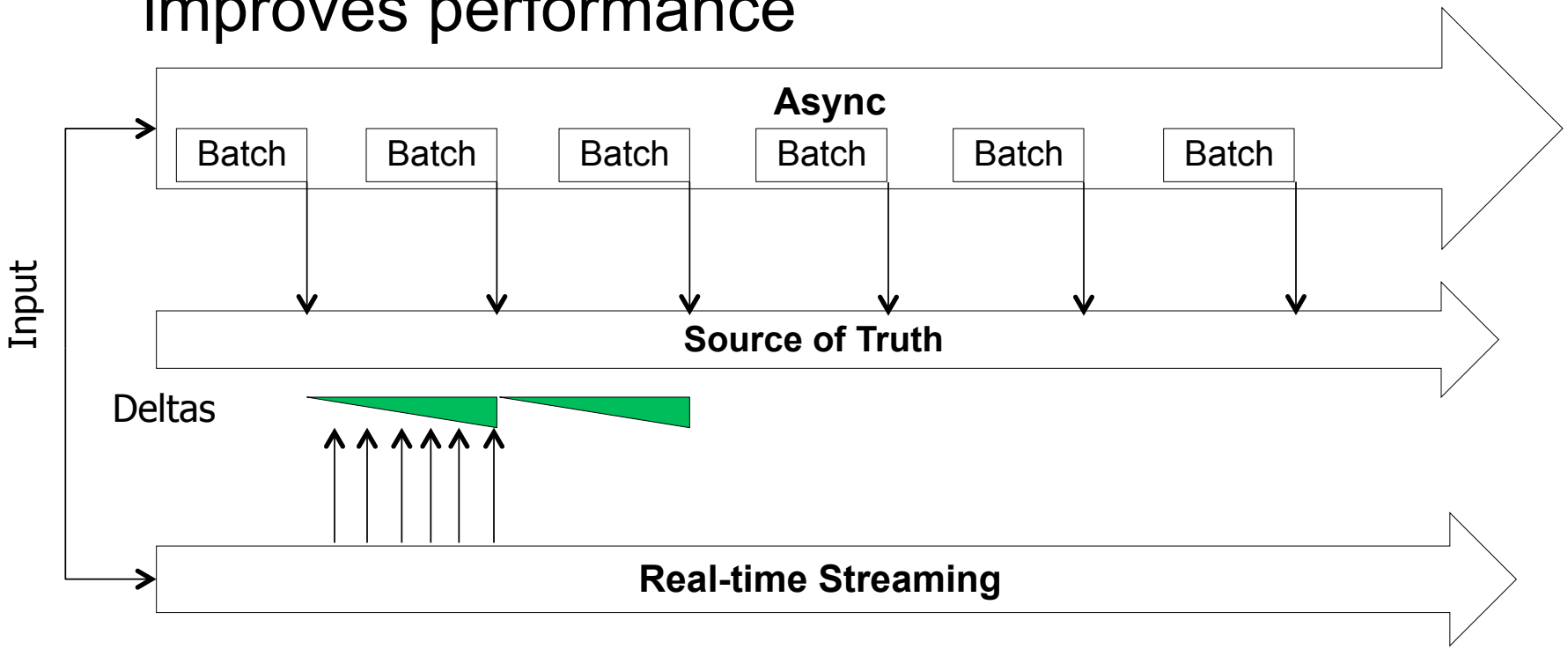
Update a ML model in near real time, and score results using it at the same time.

http://www.bigdata-cookbook.com/post/50334182923/machine-learning-over-storm

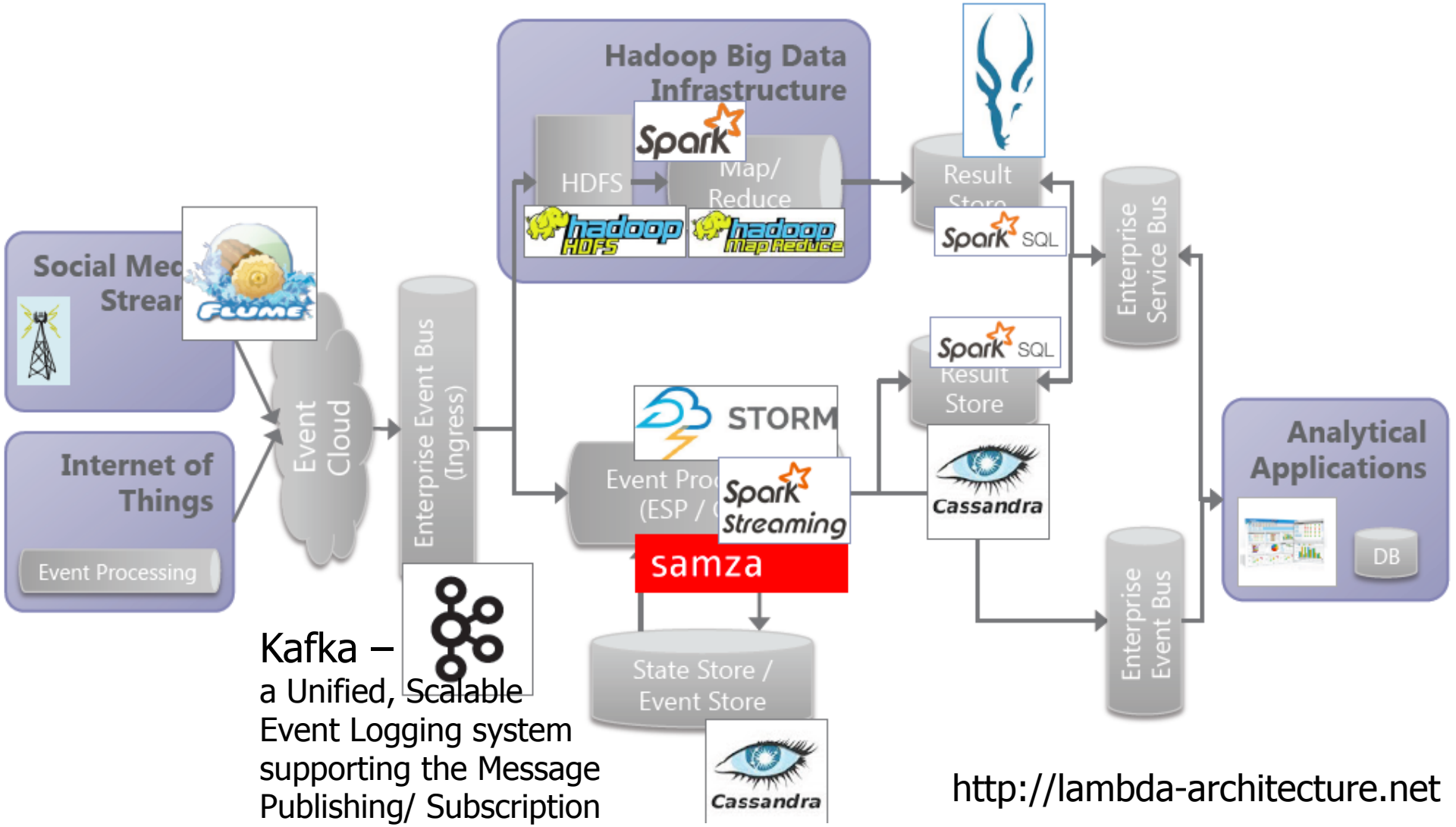
Architectural Patterns to support BOTH Real-Time and Batched Big Data Processing

Recall: The Two-Pronged Approach

- http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
- The interesting take-away: Fast Real-Time path with Batch Backup reduces complexity and improves performance



Architectural Pattern #2: Event Stream Processing as part of the Lambda Architecture (proposed by Nathan Marz)



Kafka –
a Unified, Scalable Event Logging system supporting the Message Publishing/ Subscription model

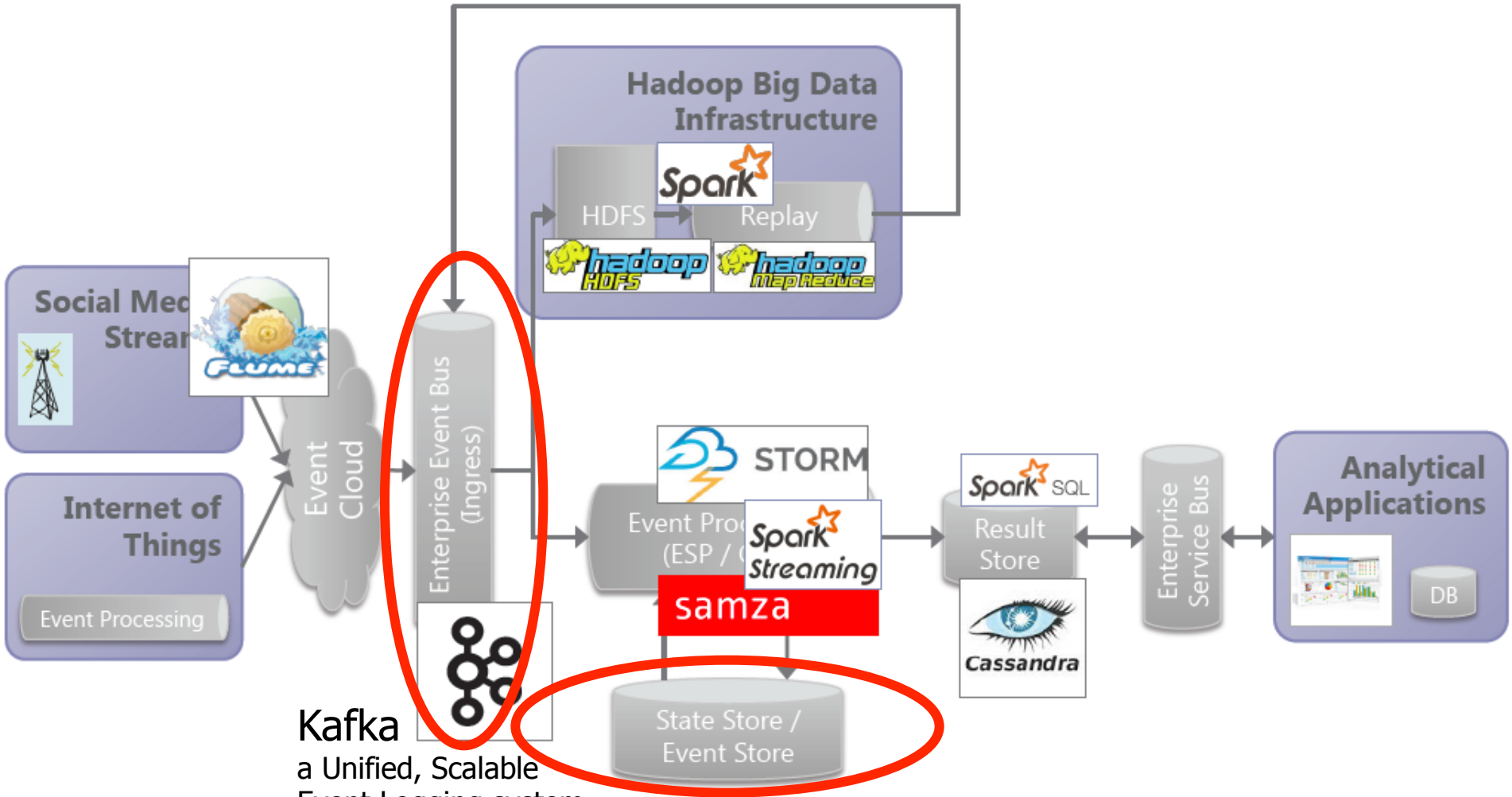
http://lambda-architecture.net

Summing Bird

- https://github.com/twitter/summingbird

Write the same code (script) and then compile to be run on Storm as well as one Hadoop.

Architectural Pattern #3: Event Stream Processing as part of the Kappa Architecture (from LinkedIn)



Kafka
a Unified, Scalable Event Logging system supporting the message Publishing/ Subscription model

http://milinda.pathirage.org/kappa-architecture.com/

Additional References

- A few Storm books are already available, e.g.
  - Storm Applied by S.T. Allen et al, published by Manning, 2015
- Storm documentation
  - http://storm.incubator.apache.org/documentation/Home.html
- storm-kafka
  - https://github.com/apache/incubator-storm/tree/master/external/storm-kafka
- Mailing lists
  - http://storm.incubator.apache.org/community.html
- Code examples
  - https://github.com/apache/incubator-storm/tree/master/examples/storm-starter
  - https://github.com/miguno/kafka-storm-starter/
- Related work aka tools that are similar to Storm – try them, too!
  - Spark Streaming
    - See comparison Apache Storm vs. Apache Spark Streaming, by P. Taylor Goetz (Storm committer)