

IEMS 5730/ IERG4330

Spring 2024

Stream Processing  
Apache Storm and Kafka

Prof. Wing C. Lau

Department of Information Engineering

wclau@ie.cuhk.edu.hk

# Acknowledgements

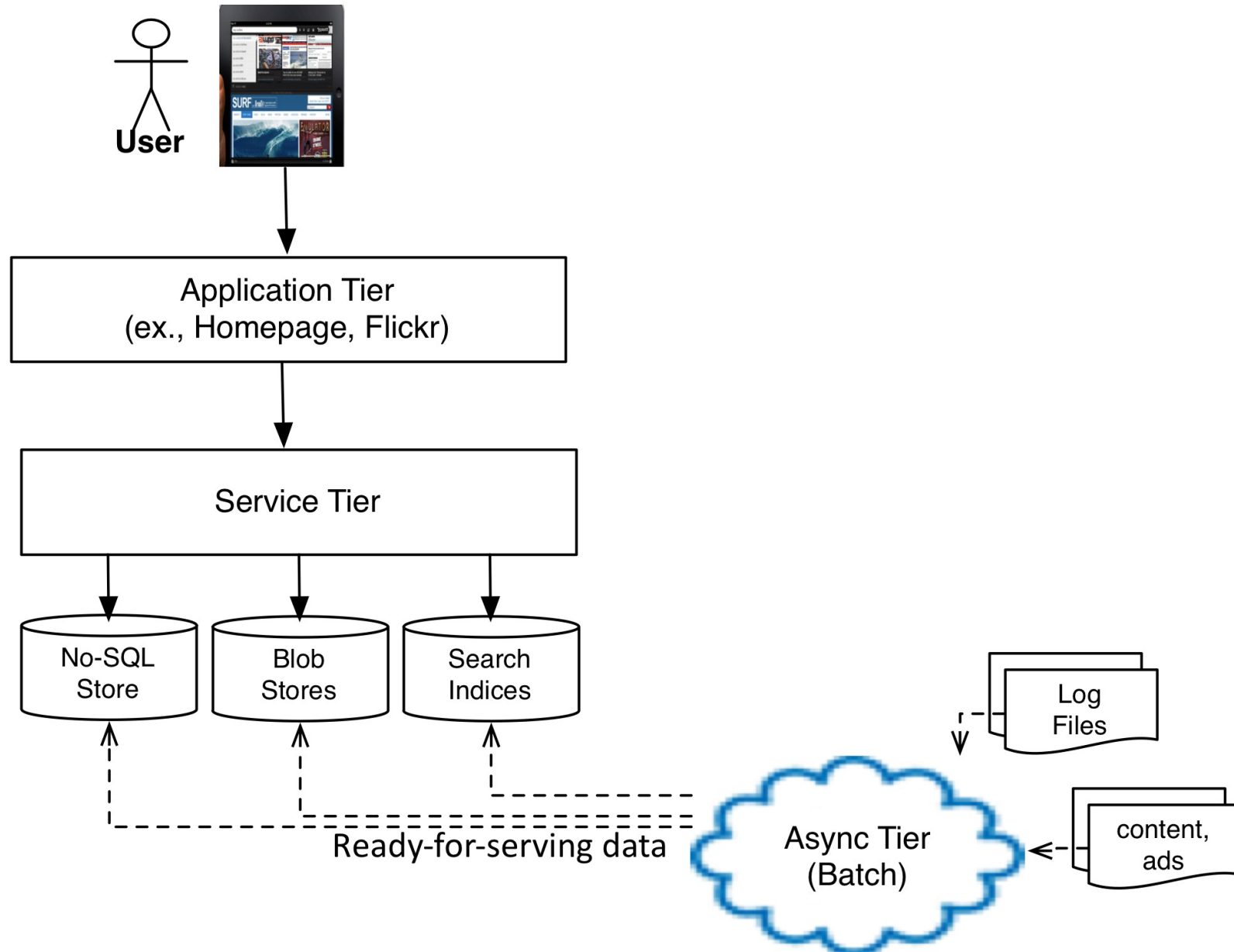
- The slides used in this chapter are adapted from the following sources:
  - Nathan Marz, “Storm – Distributed and Fault-tolerant real-time computation,” 2011, <http://cloud.berkeley.edu/data/storm-berkeley.pdf>
  - Krishna Gade of Twitter, “storm - Stream Processing @twitter,” June 2013.
  - Michael 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
- All copyrights belong to the original authors of the materials.

# Example Use Case: Data Driven Personalization

<http://visualize.yahoo.com/core>

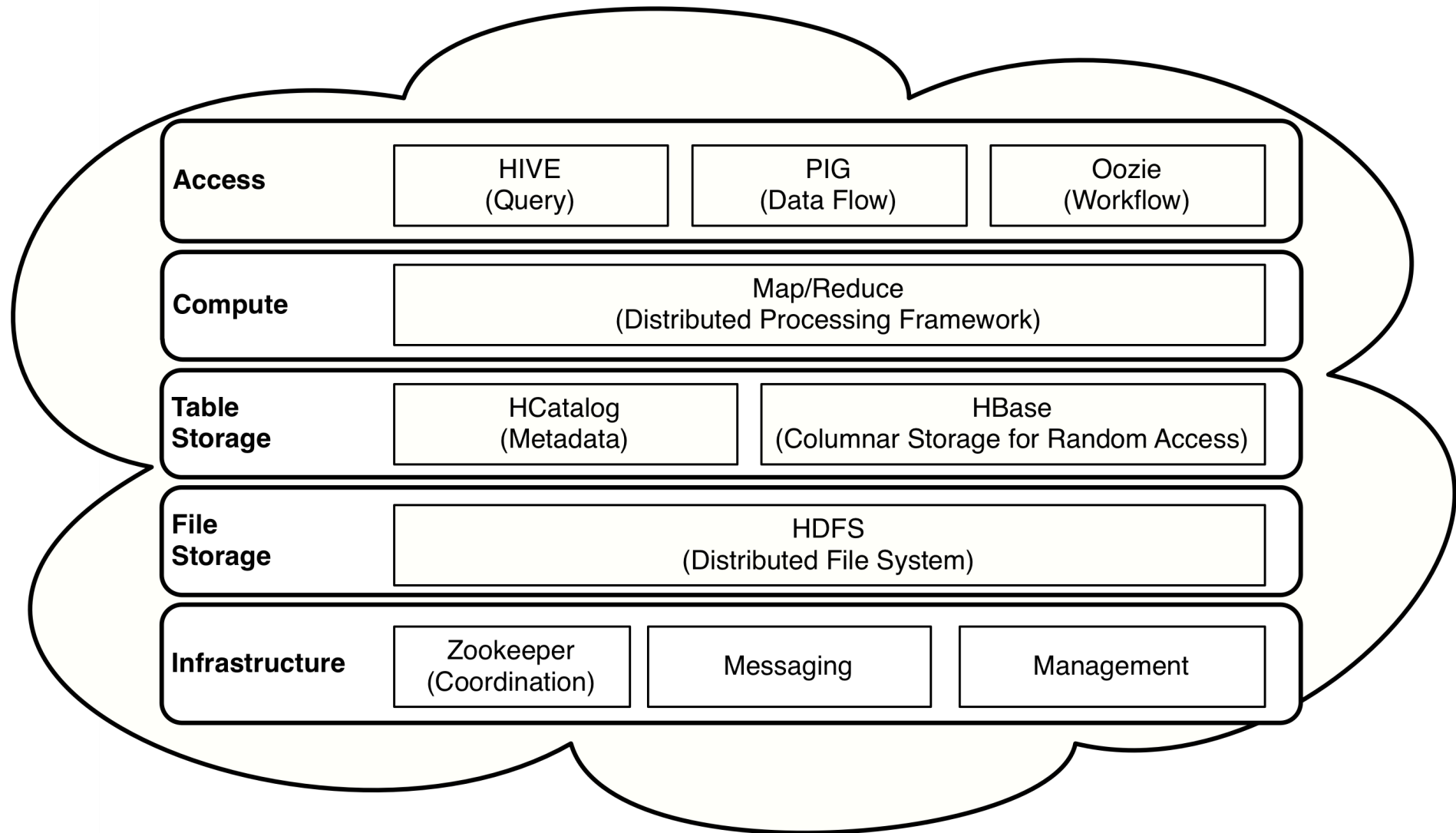


# System Architecture for Data Driven Personalization based on Offline, Asynchronous (e.g. Daily/Weekly) Log Processing/ Data Analytics





## @ 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 throughput
  - Move computation to the data nodes
- Highly available
  - Built-in failover

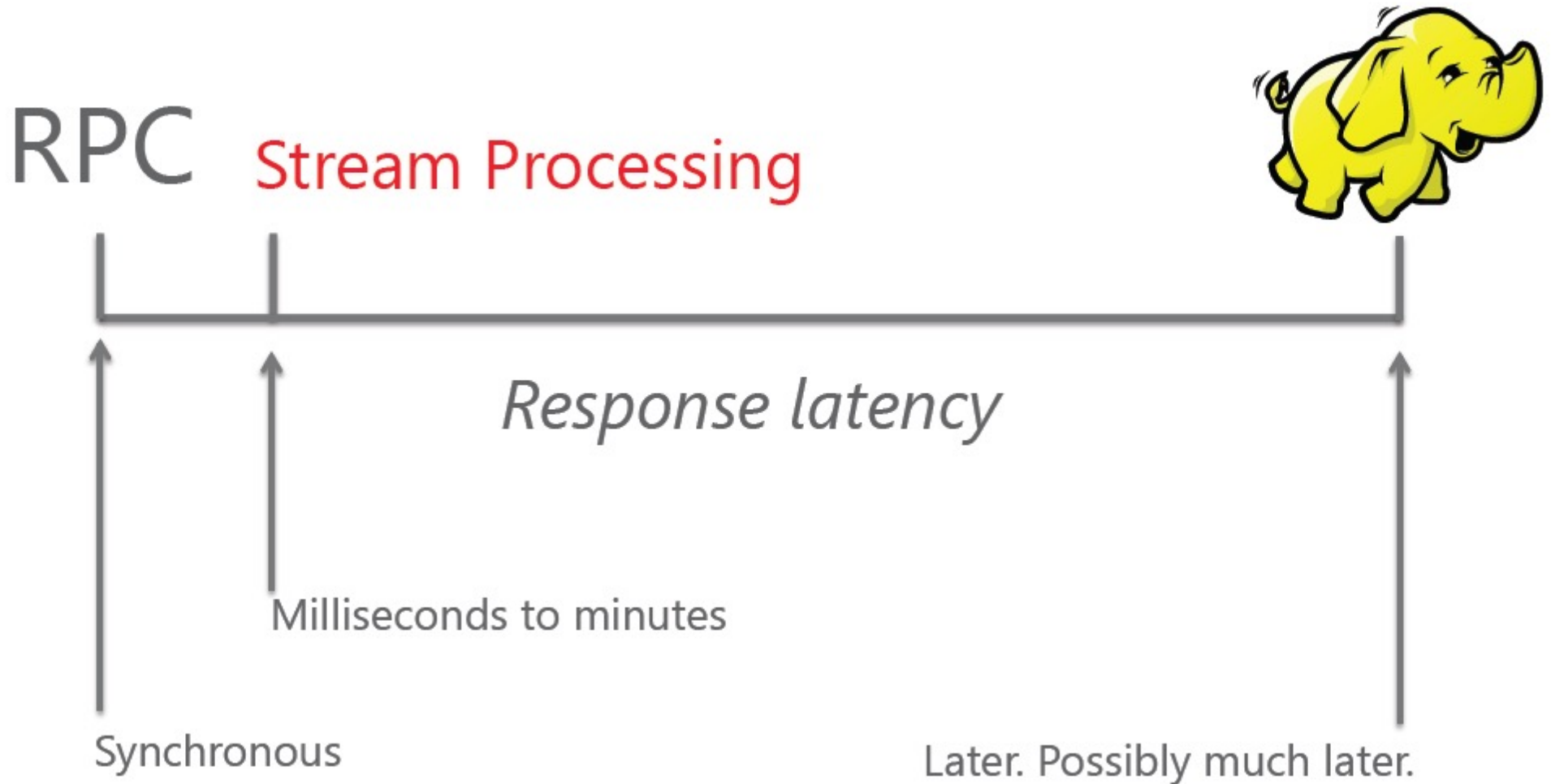
## Weakness

- 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



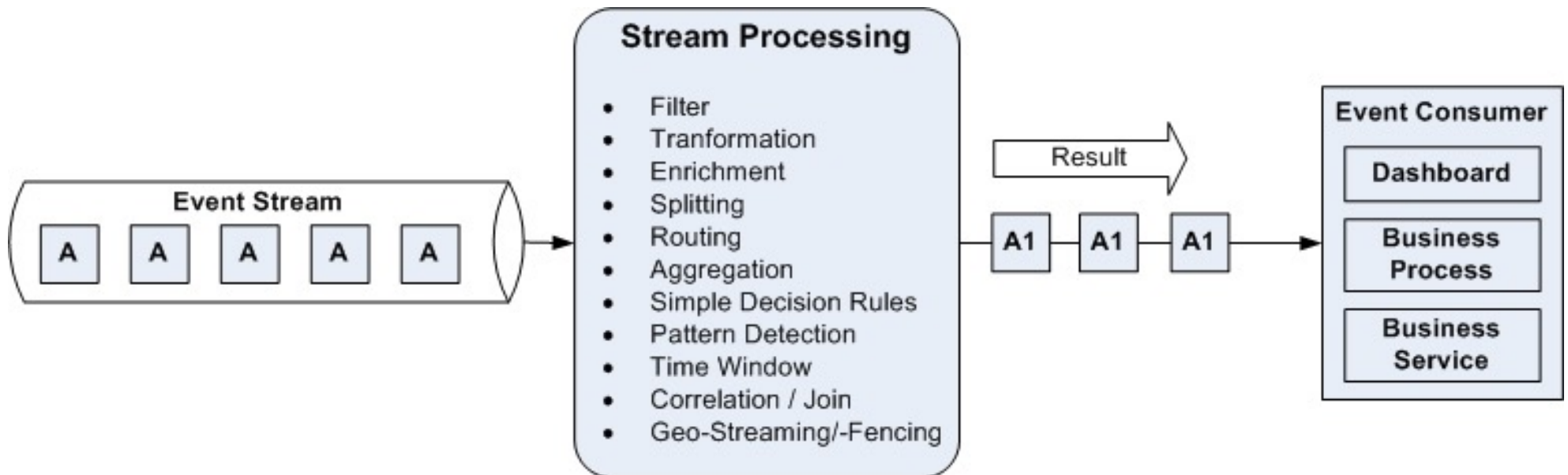
# Why Stream Processing ?



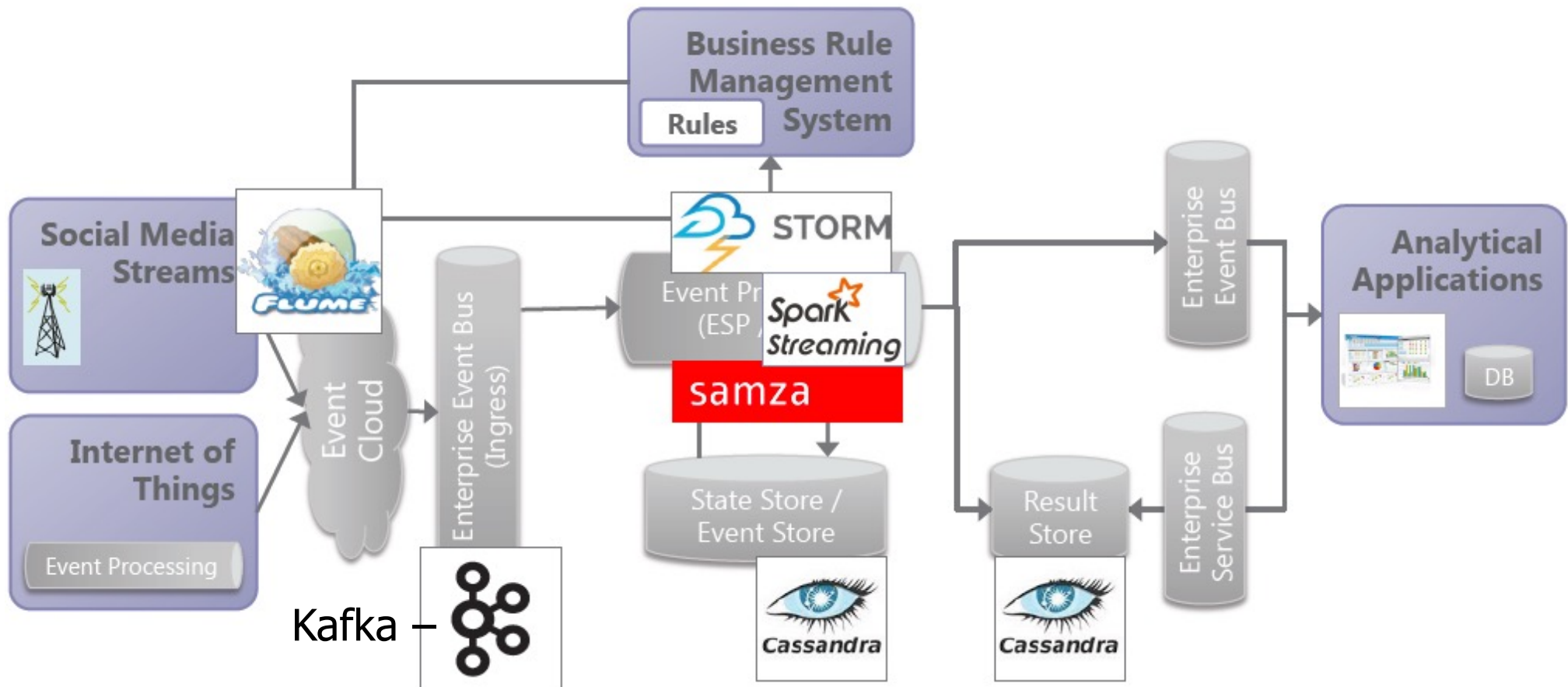


# What is Stream Processing ?

- 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:

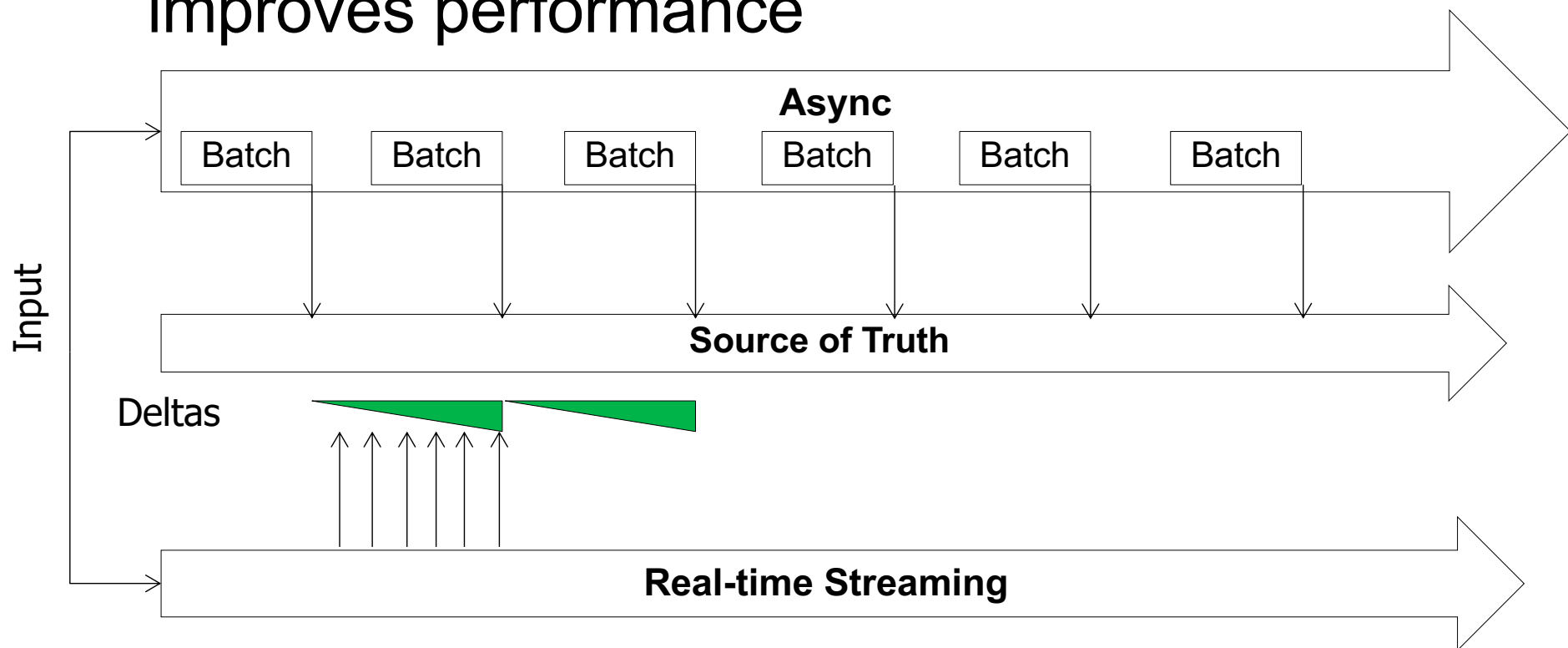


Kafka —  
a Unified, Scalable  
Event Logging system supporting  
the Message Publishing/  
Subscription model

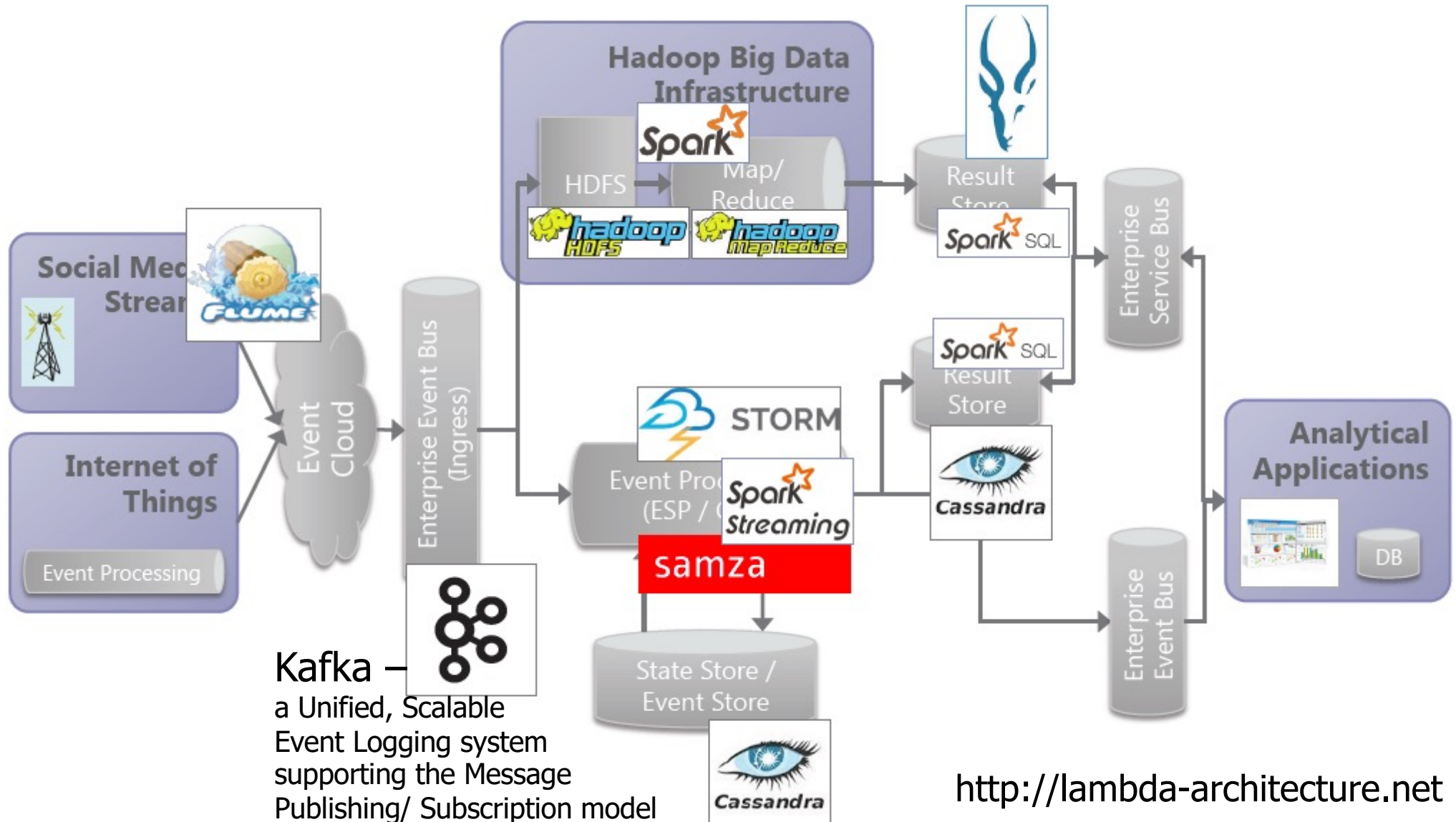
Architectural Patterns  
to support  
BOTH Real-Time and Batched  
Big Data Processing

# 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)



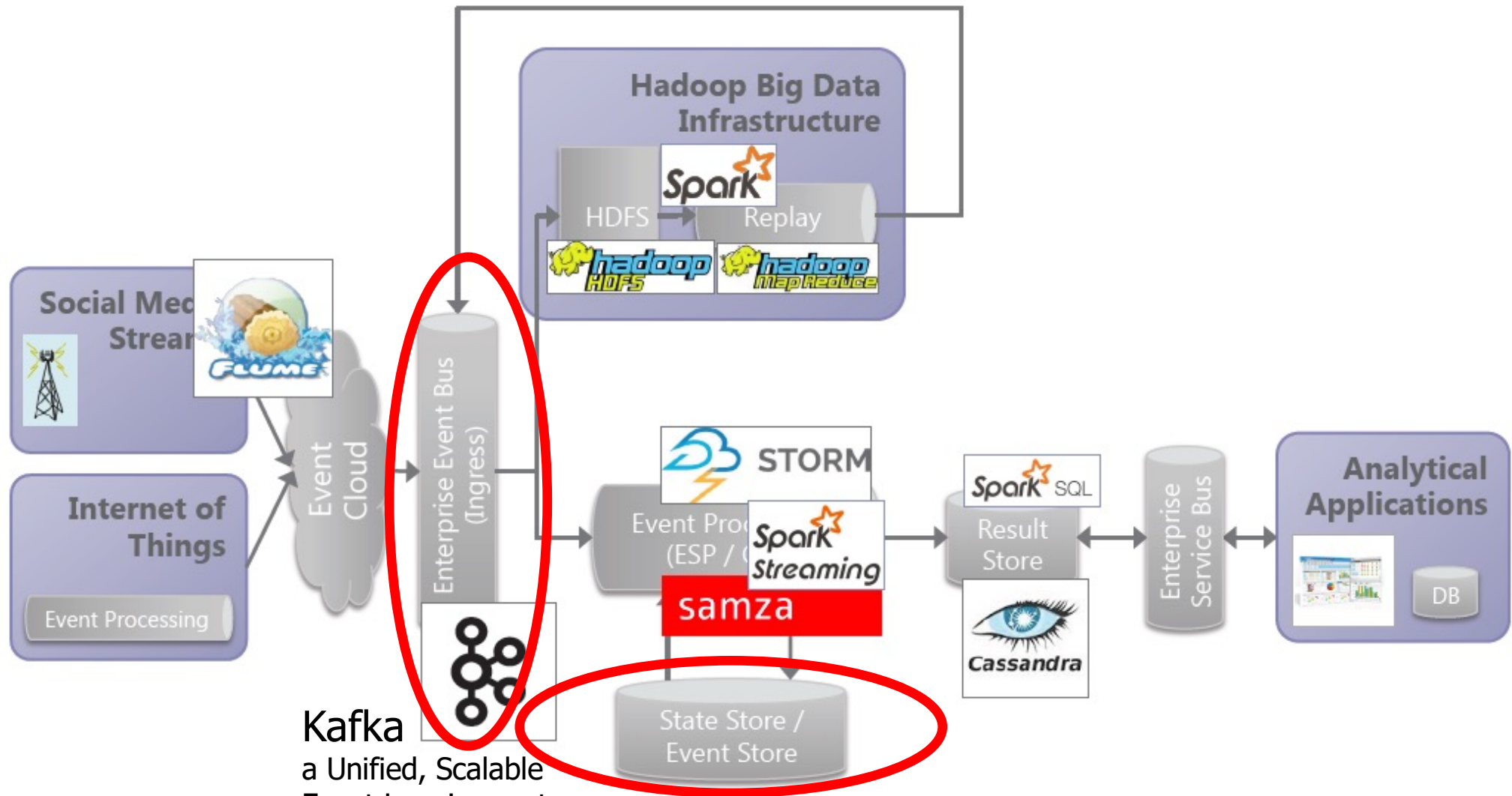
<http://lambda-architecture.net>

# Summing Bird

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

Write the same code (script) and then compile to be run on Storm as well as on 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/>

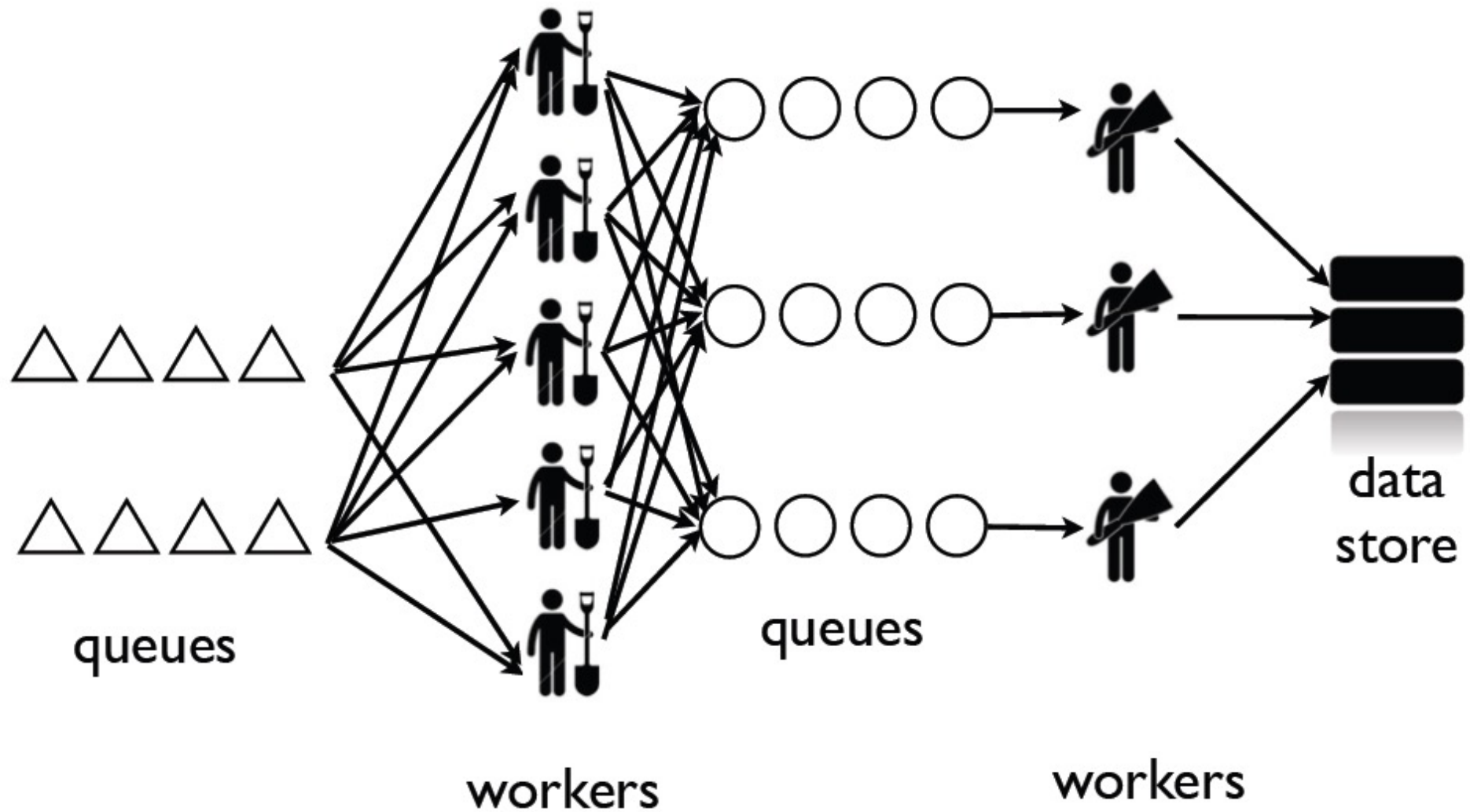
# Stream Processing with Apache Storm



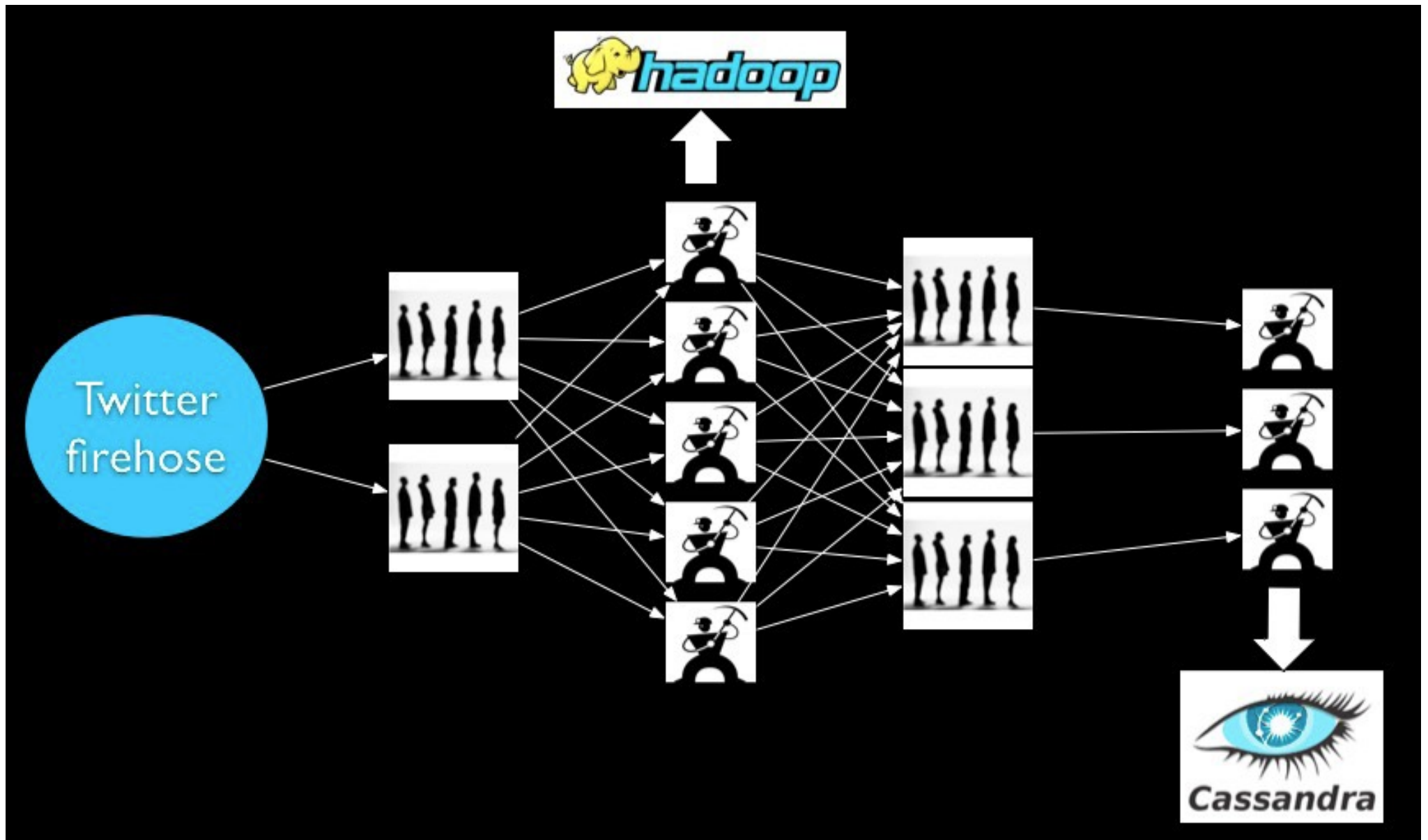
# Agenda

- 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
  - ~~Operational Guidelines for Storm~~
  - ~~Adoption Statistics and Real-time Use Cases~~
  - Future Extensions (Researchie)

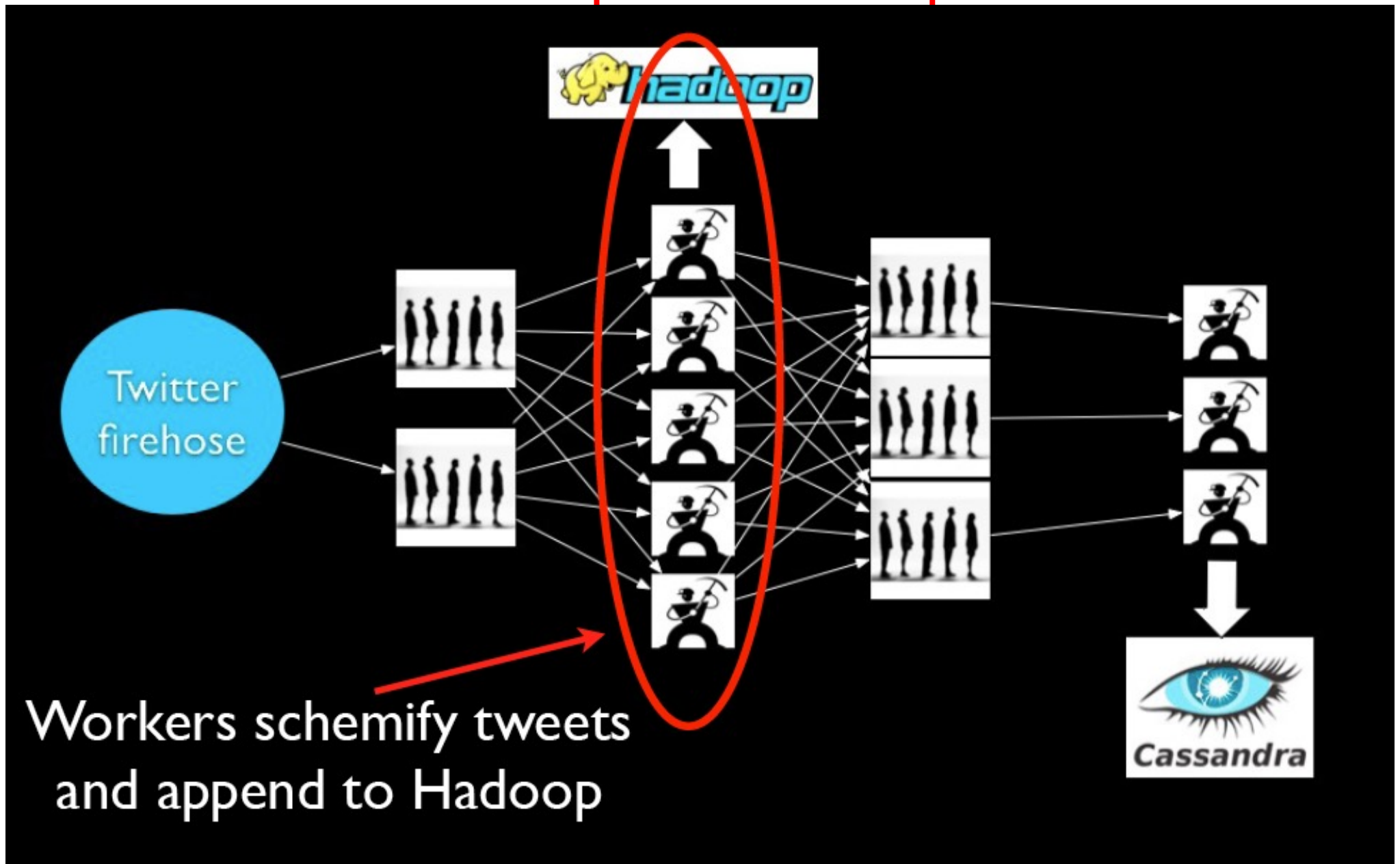
# Traditional Workflow under the Queues-Workers Model for Event Processing



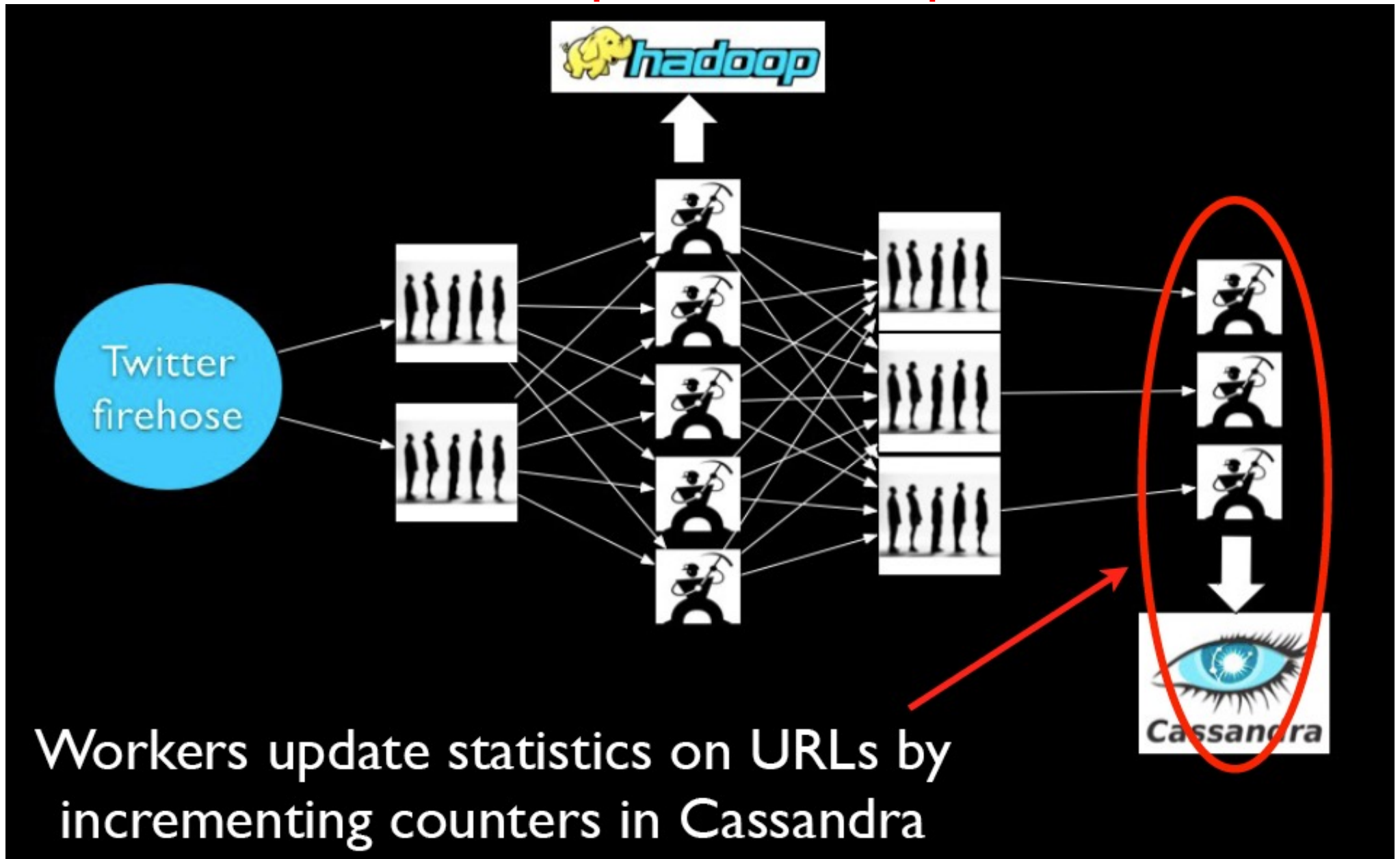
# A Simplified Example



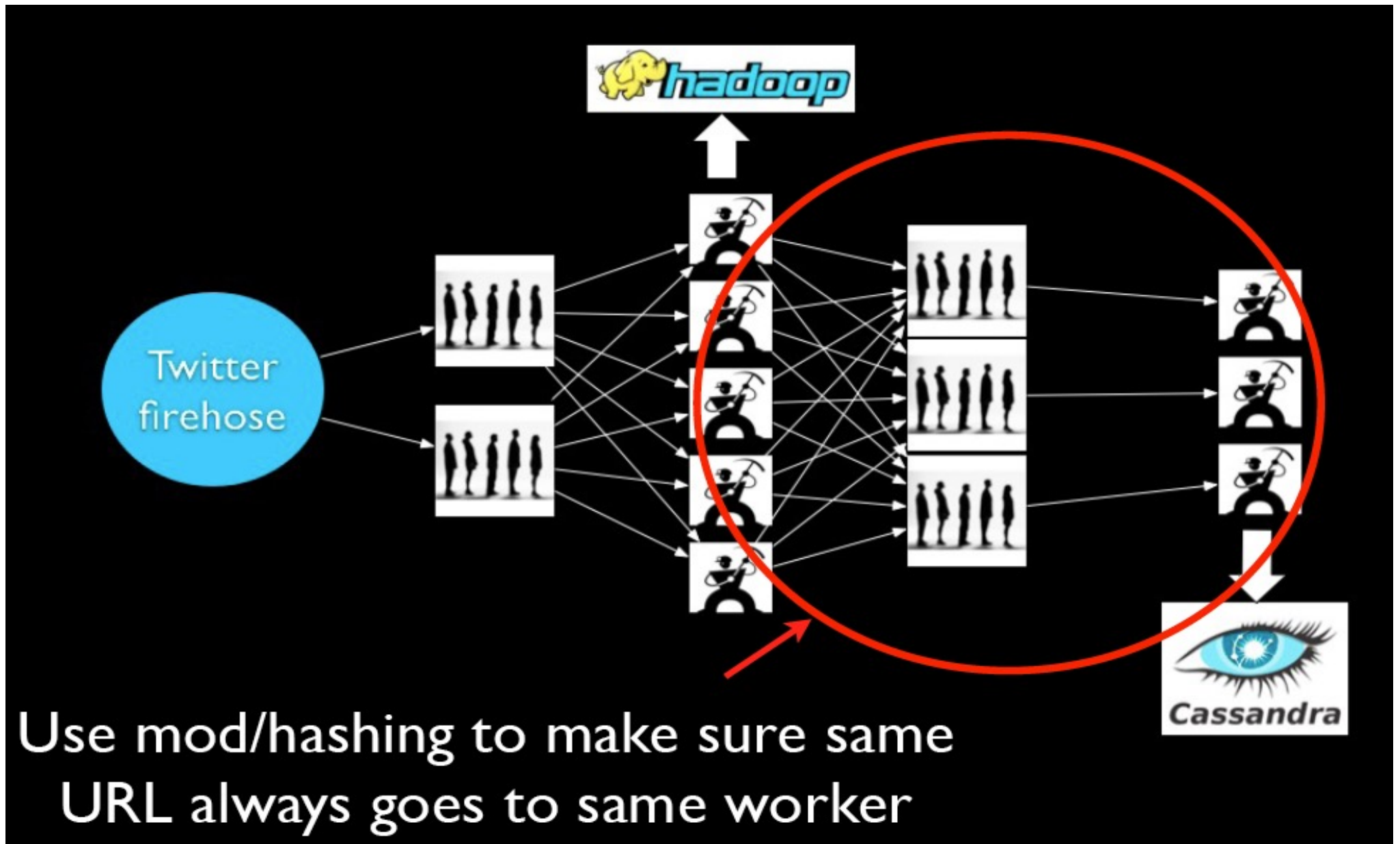
# A Simplified Example



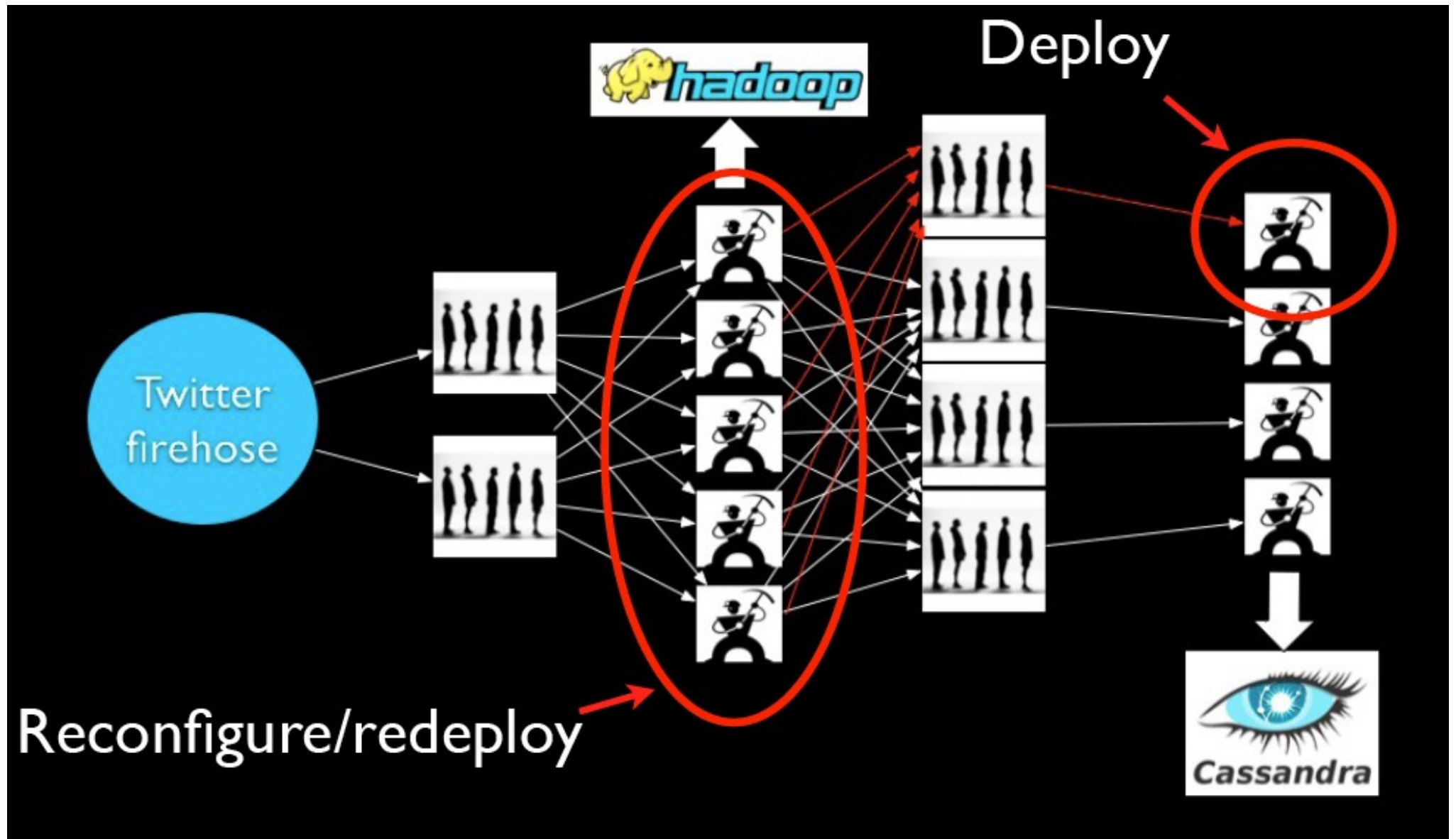
# A Simplified Example



# 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



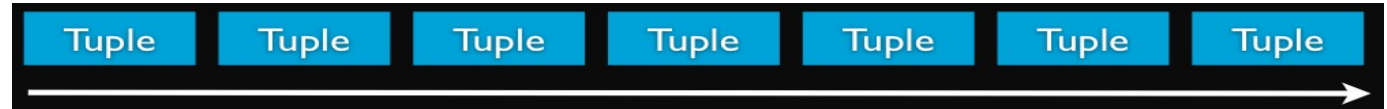
# A Solution: Apache Storm



- 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!
- Support Guaranteed data processing (at least once) ; default: at most 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
  - Heron uses **the SAME programming abstraction** and is **100% API-compatible with Storm**
  - [Under Apache incubation \(as of Feb 2019\) https://github.com/apache/incubator-storm](#) & [Kafka](#)

# Key Concepts in Storm

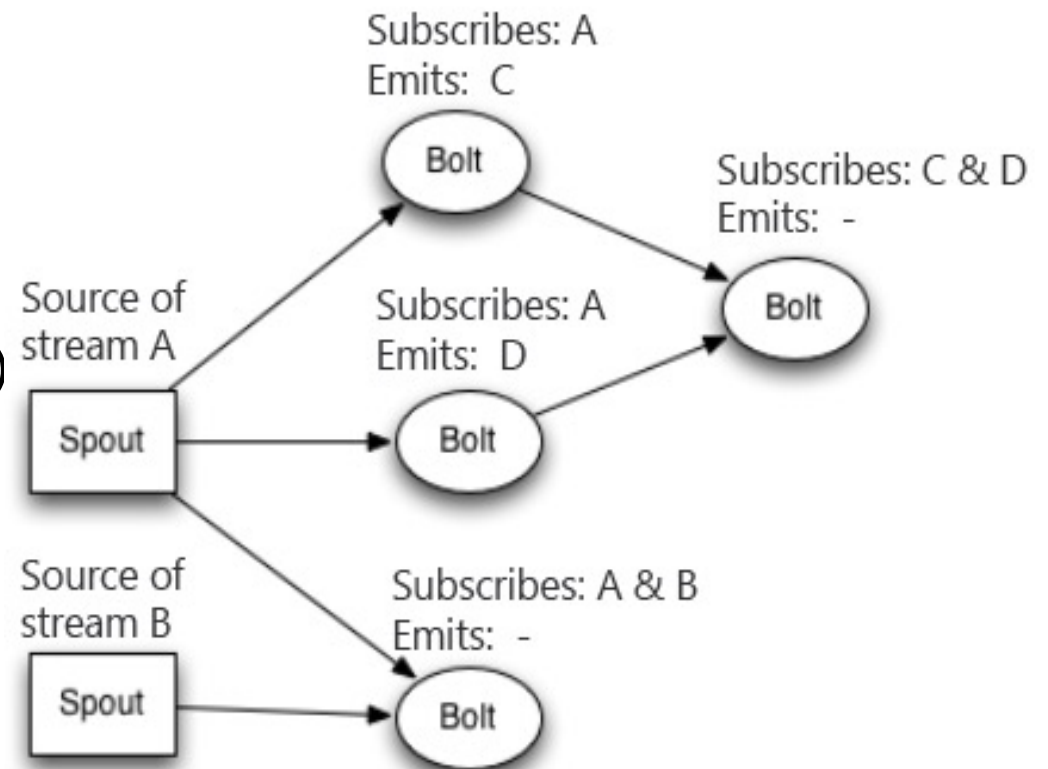
## Stream



- 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

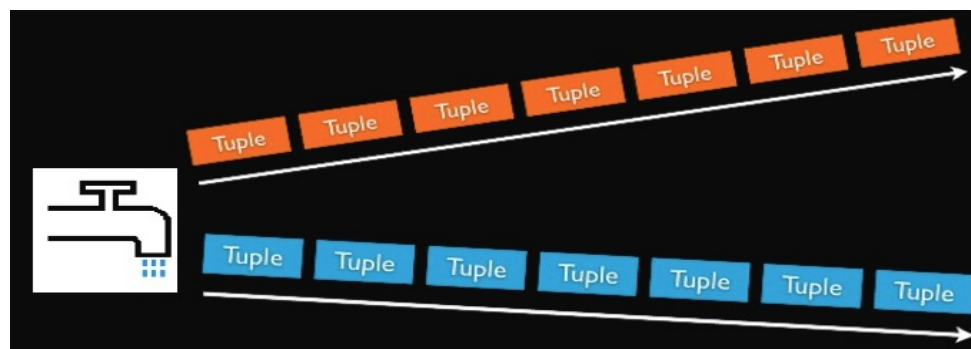
- A Directed Acyclic Graph (DAG) where each node is either a Data source (Spout) or a Processing node (Bolt)
- An Edge indicates which Bolt subscribes to which Stream



# Key Concepts in Storm (cont'd)

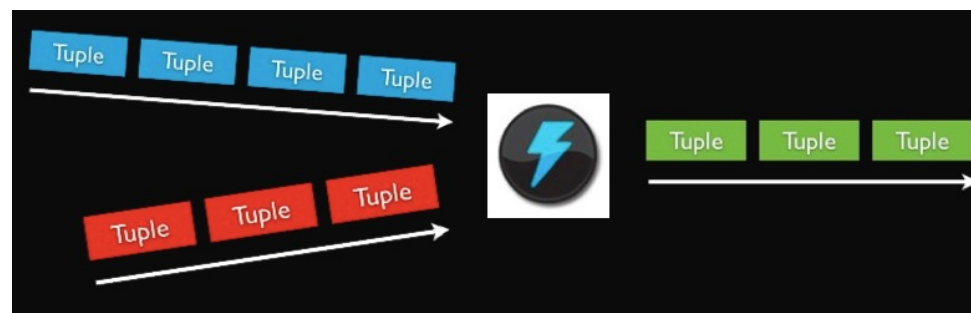
## ■ Spout

- 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)



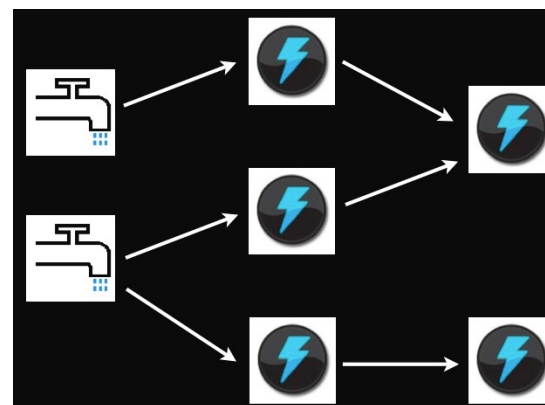
## ■ Bolt

- Processes 1+ input stream(s) and produces 1+ new stream(s)
  - e.g. Calculate, Functions, Filters, Aggregation, Joins, talk to database



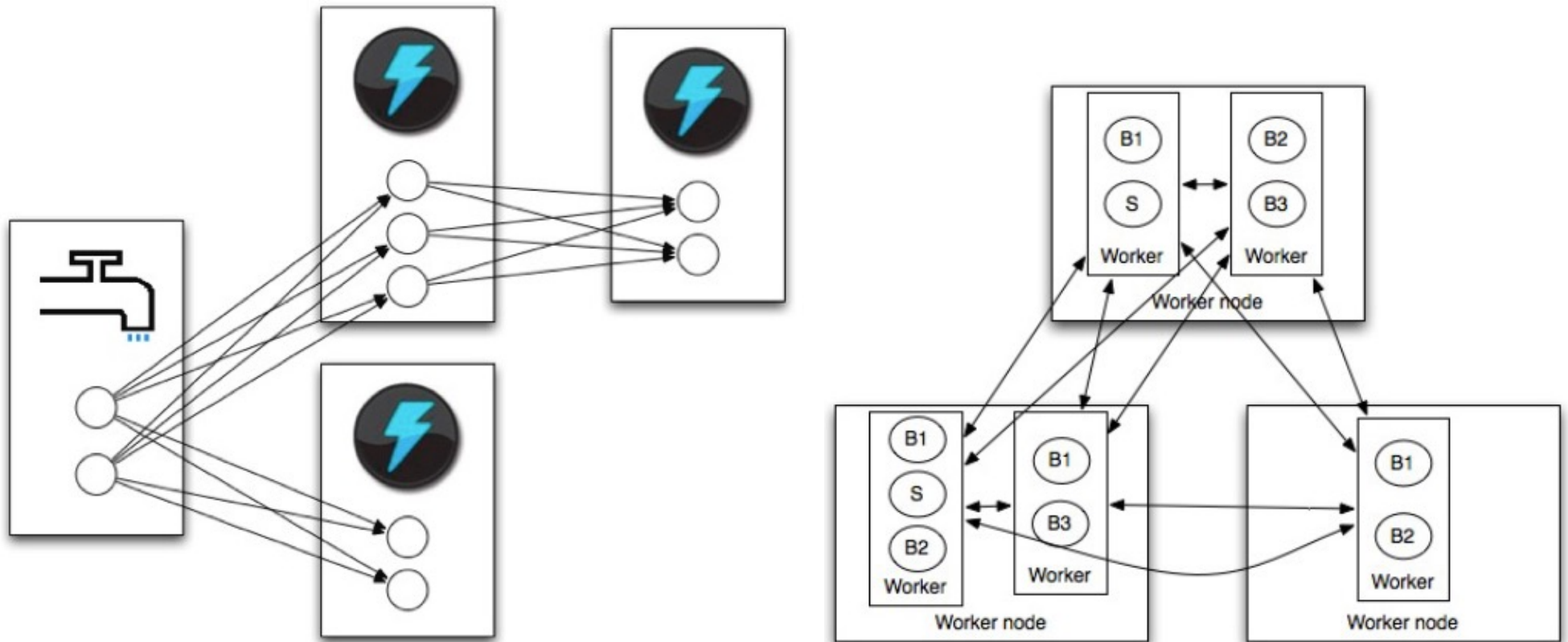
## ■ A sample Storm Topology

- Compiled to be executed on many machines similar to a MapReduce job in Hadoop



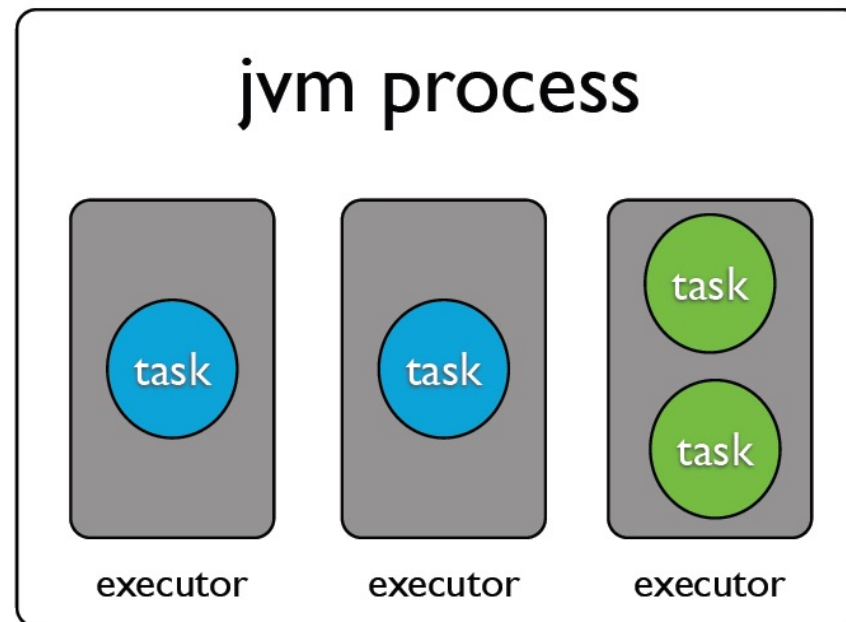
# Storm Tasks

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

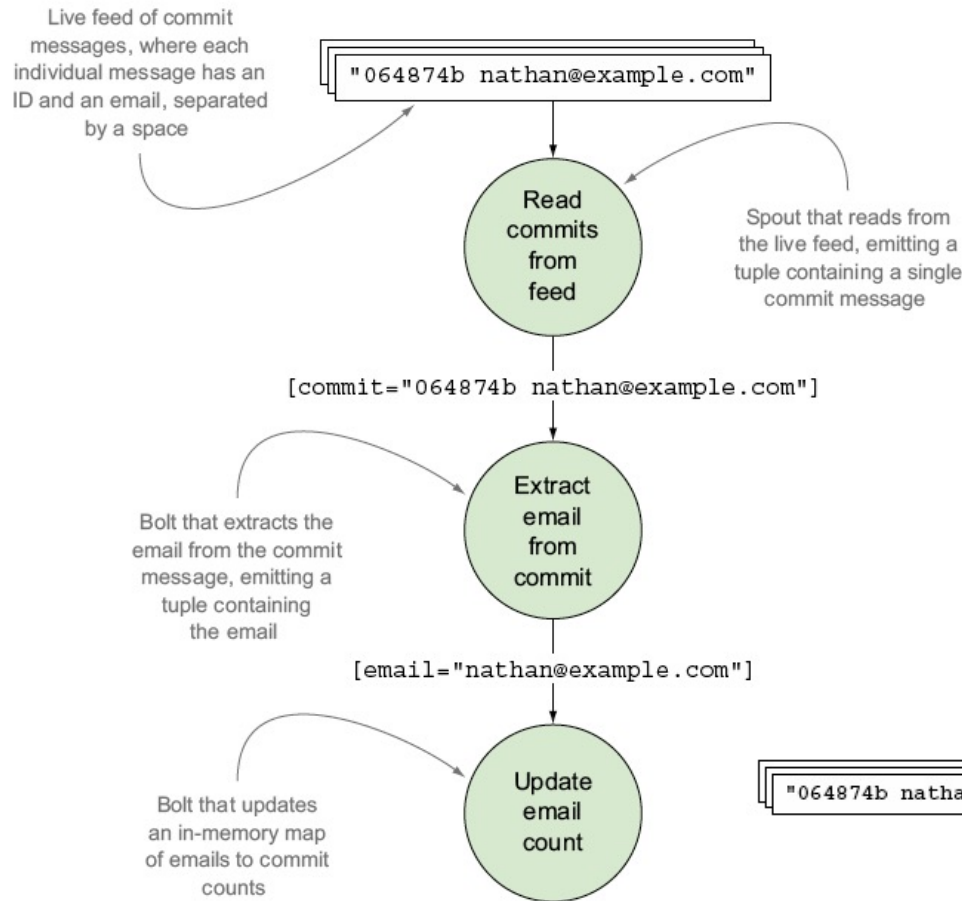


# Key Concepts in Storm (cont'd)

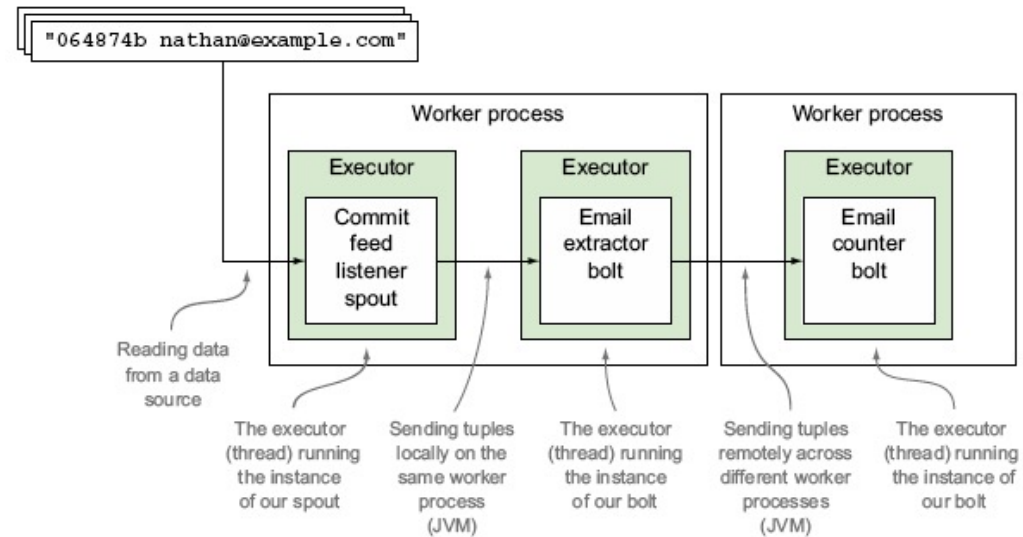
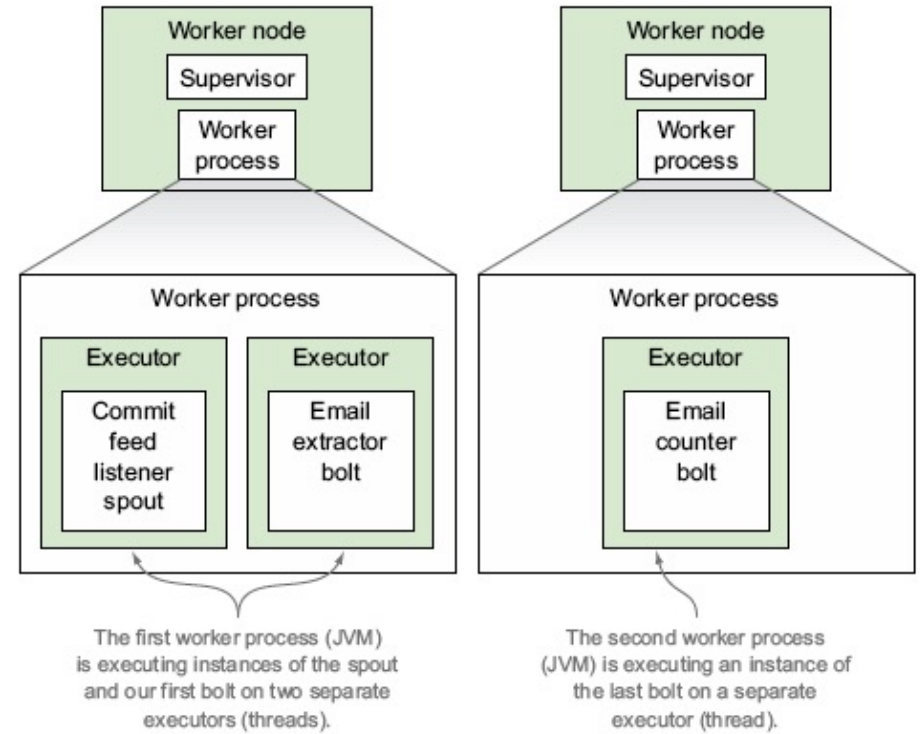
- Worker (JVM) Process
  - Executes subset of a Topology ;
  - May run 1 or more threads (Executors) for one or more components
  - One Thread per Executor
- Task
  - The actual data processing instance executing by the thread
  - It is possible for multiple tasks to share one thread (Why ? to facilitate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



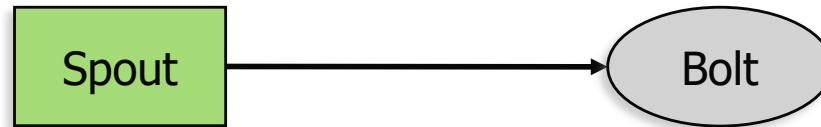
# An Example on deploying a Topology across a Cluster



Our topology is executing across two worker nodes (physical or virtual machines), where each worker node is running a single worker process (JVM).



# A trivial “Hello, Storm” topology



“emit random  
number < 100”



“multiply  
by 2”

**(74)**

**(148)**

# Code

## Spout

```
1 ▾ public void nextTuple() {  
2     | final Random rand = new Random(); // normally this should be an instance field  
3     | int nextRandomNumber = rand.nextInt(100);  
4     | collector.emit(new Values(nextRandomNumber)); // auto-boxing  
5     | }
```

## 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    | }
```



# 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 `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.

```
1 @Override
2 public Map<String, Object> getComponentConfiguration() {
3     Config config = new Config();
4     config.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 10);
5     return config;
6 }
```

- 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.

```
1  @Override
2  public void execute(Tuple tuple) {
3  if (isTickTuple(tuple)) {
4      // now you can trigger e.g. a periodic activity
5  }
6  else {
7      // do something with the normal tuple
8  }
9  }
10
11 private static boolean isTickTuple(Tuple tuple) {
12     return tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID)
13         && tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID);
14 }
```

- 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/>

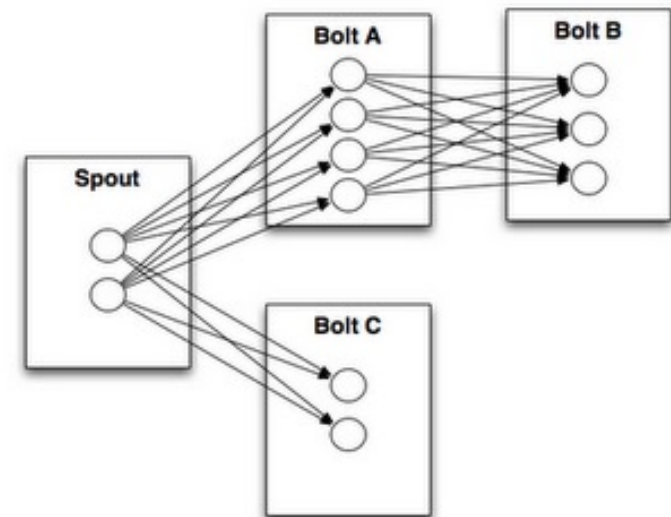
# Common spout/bolt gotchas

- `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()`

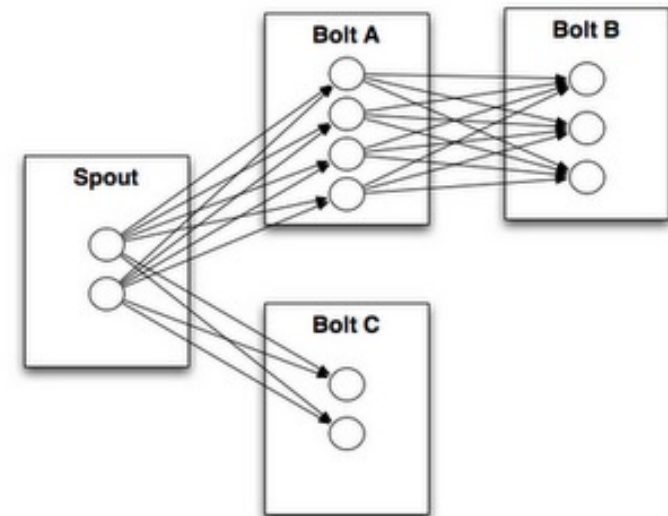
```
1 Config conf = new Config();
2 conf.setNumWorkers(200);
3
4 TopologyBuilder builder = new TopologyBuilder();
5 builder.setSpout("my-spout", new MySpout(), 100);
6 builder.setBolt("my-bolt", new MyBolt(), 200)
7     .shuffleGrouping("my-spout");
8 StormTopology topology = builder.createTopology();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



# 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/ Submitting 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
```

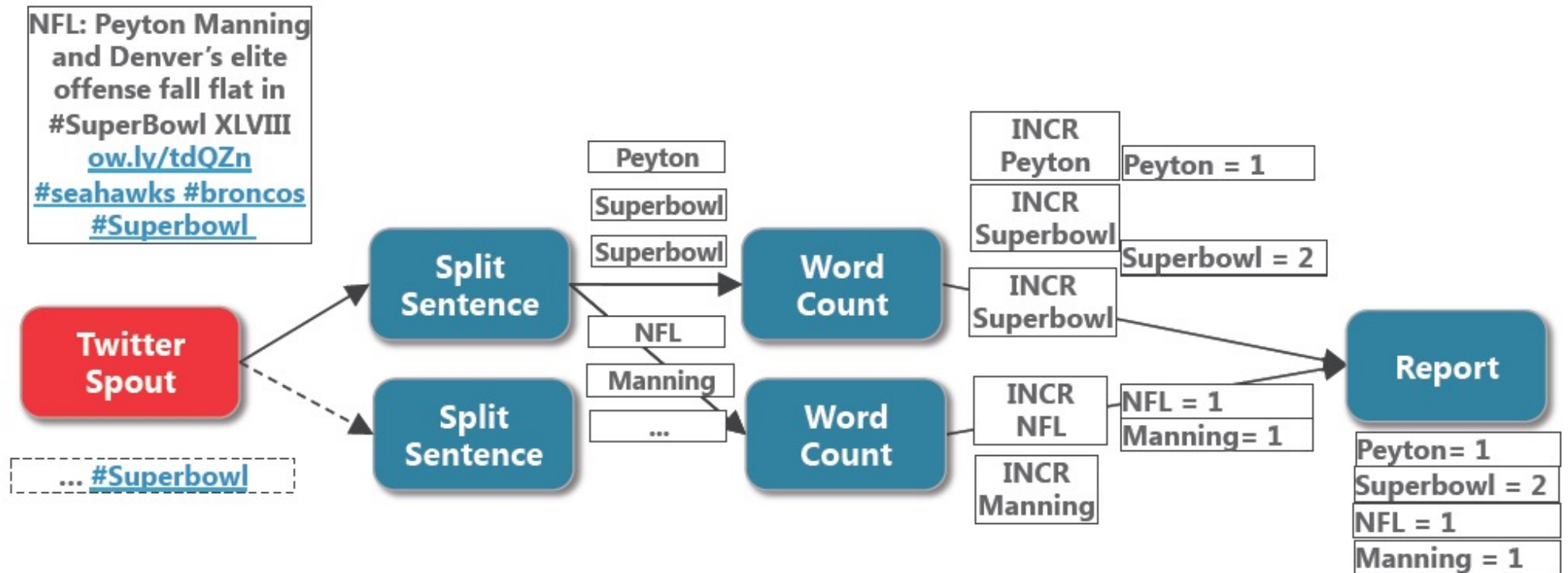
- Will connect to Nimbus, upload your jar, and run the topology.
- Use any machine that can run "storm jar" and talk to Nimbus' Thrift port.
- You can pass additional JVM options via \$STORM\_JAR\_JVM\_OPTS.



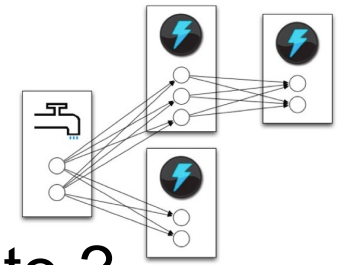
# Alright, my topology runs – now what?

- 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
    - One (new) way to overcome this is to use “Stateful Bolts with Automatic Checkpointing” by extending from the **BaseStatefulBolt** class available from the recently released Storm Ver1.0.
  - 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



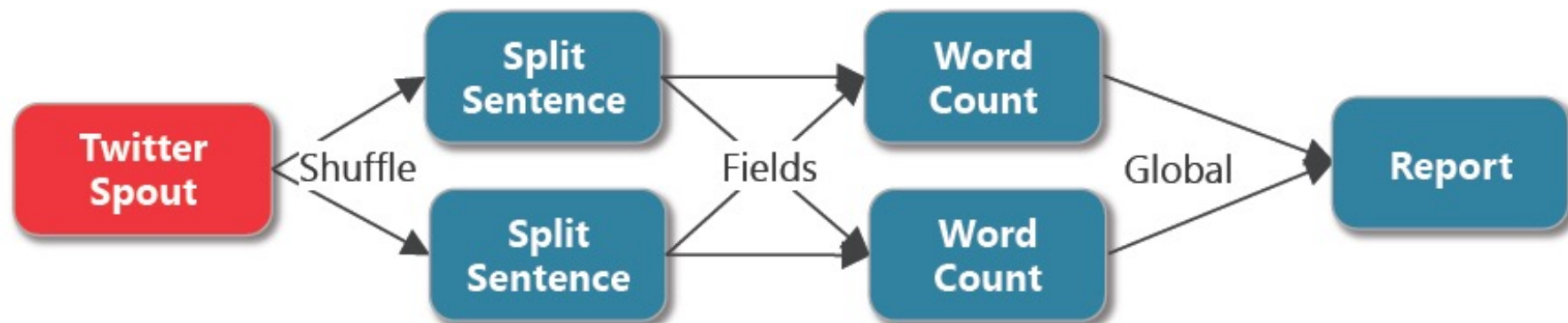
# How/ Where to route a tuple?



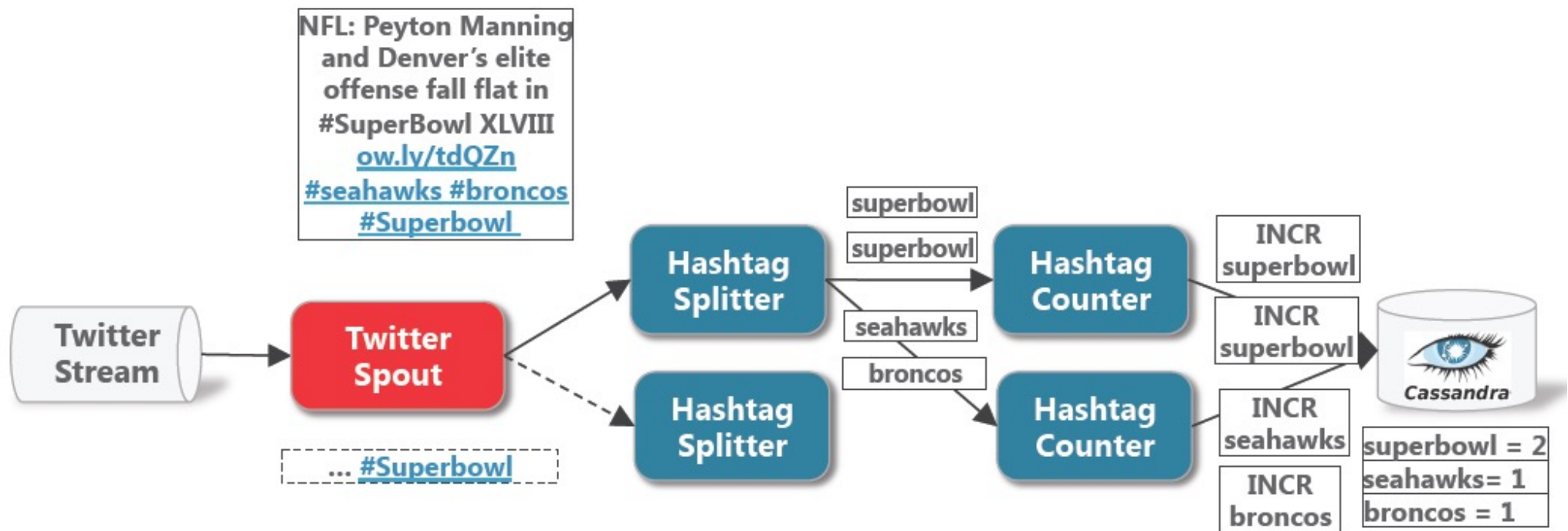
When a tuple is emitted, which task should it be routed to ?  
Ans: It is User-Programmable

<b>Shuffle grouping</b>	is random grouping
<b>Fields grouping</b>	is grouped by value, such that equal value results in equal task
<b>All grouping</b>	replicates to all tasks
<b>Global grouping</b>	makes all tuples go to one task
<b>None grouping</b>	makes bolt run in the same thread as bolt/spout it subscribes to
<b>Direct grouping</b>	producer (task that emits) controls which consumer will receive
<b>Local or Shuffle grouping</b>	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 TopologyBuilder();
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"));

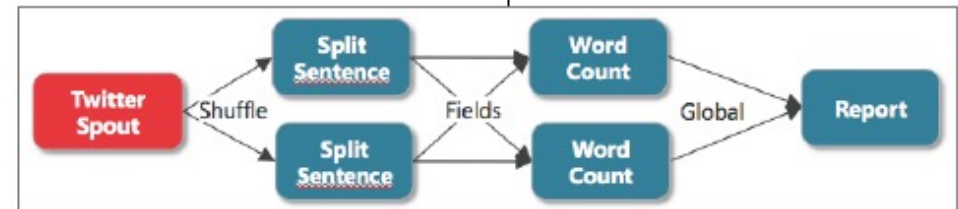
//=====
public static class SplitSentence extends ShellBolt implements TRichBolt {
    //Code to split a sentence
}

public static class WordCount implements IBasicBolt {
    //Code to count words, have to override the execute method
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        //...
    }
}
//=====
StormSubmitter.submitTopology("word-count", builder.createTopology());
```

**Parallelism Degree**  
=(Number of threads for a Spout or Bolt) ;  
Default = 1 task per thread ;  
Can be overridden by setTaskNum( )

# 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());  
        }  
    }  
}
```

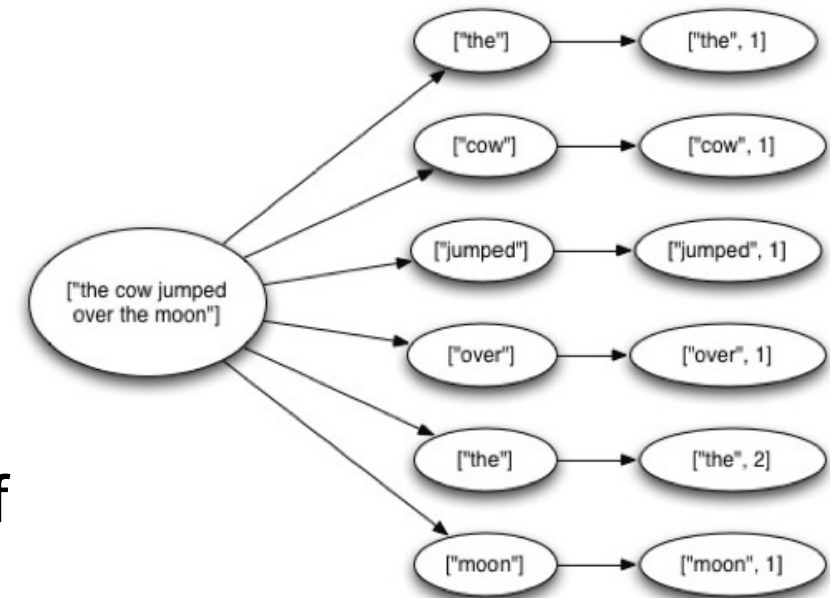


# Storm supports 3 different flavors of Message/Tuple Processing Guarantee

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 feature has been deprecated
  - Now, one should use Trident (is built on 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

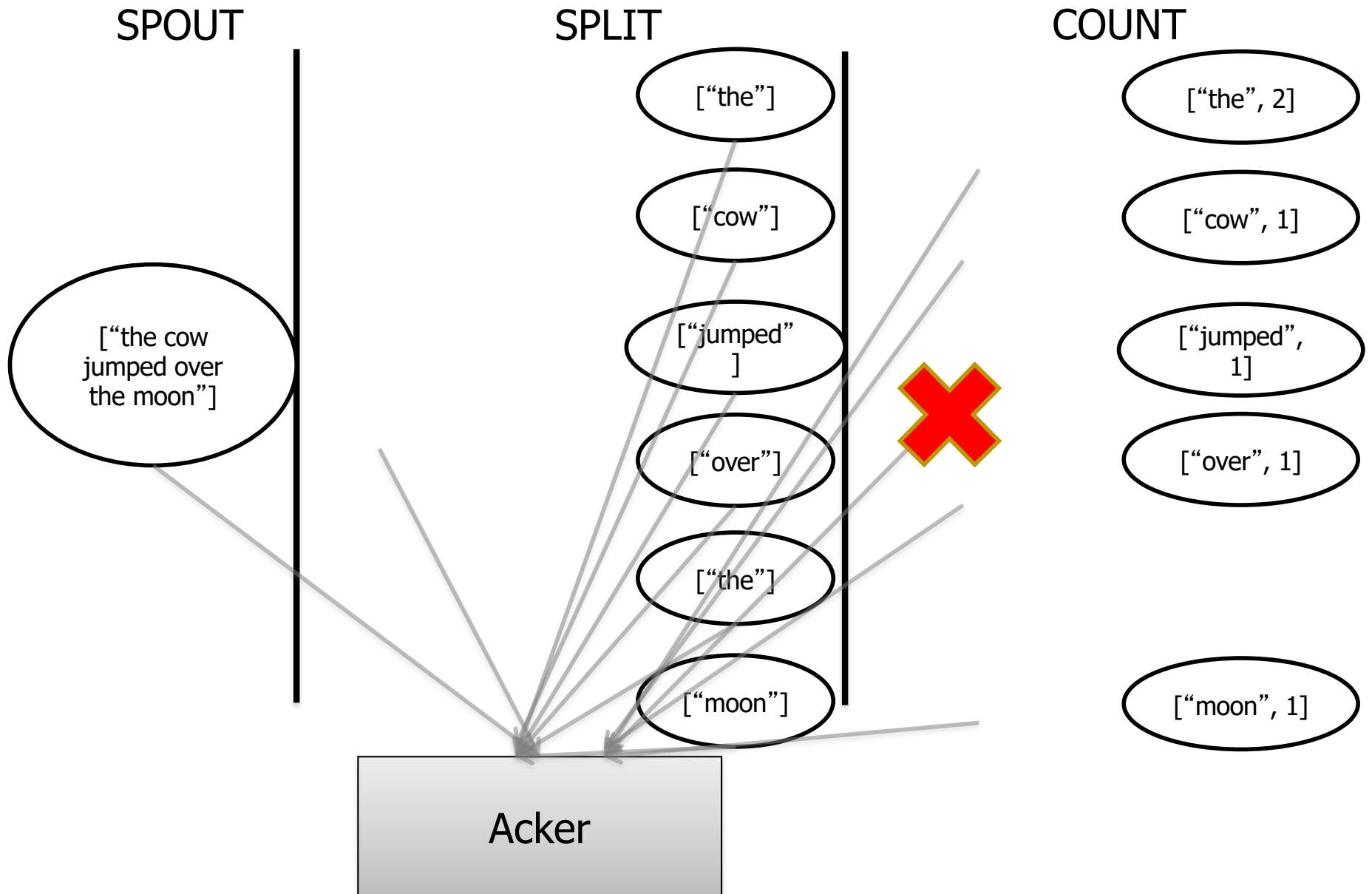
Marks a single  
node in the tree as  
complete Storm&Kafka 56



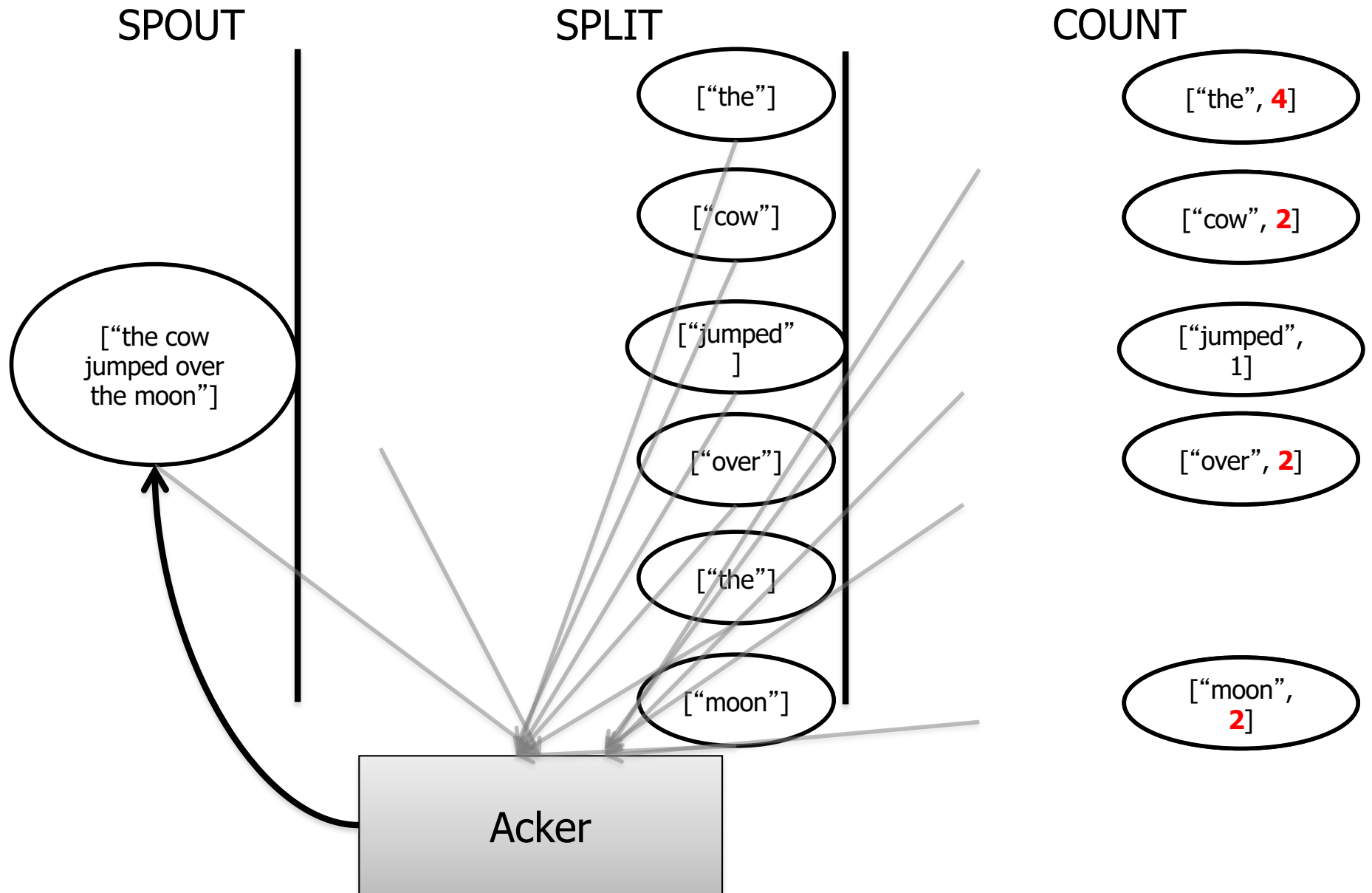
# At Least Once

- 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)

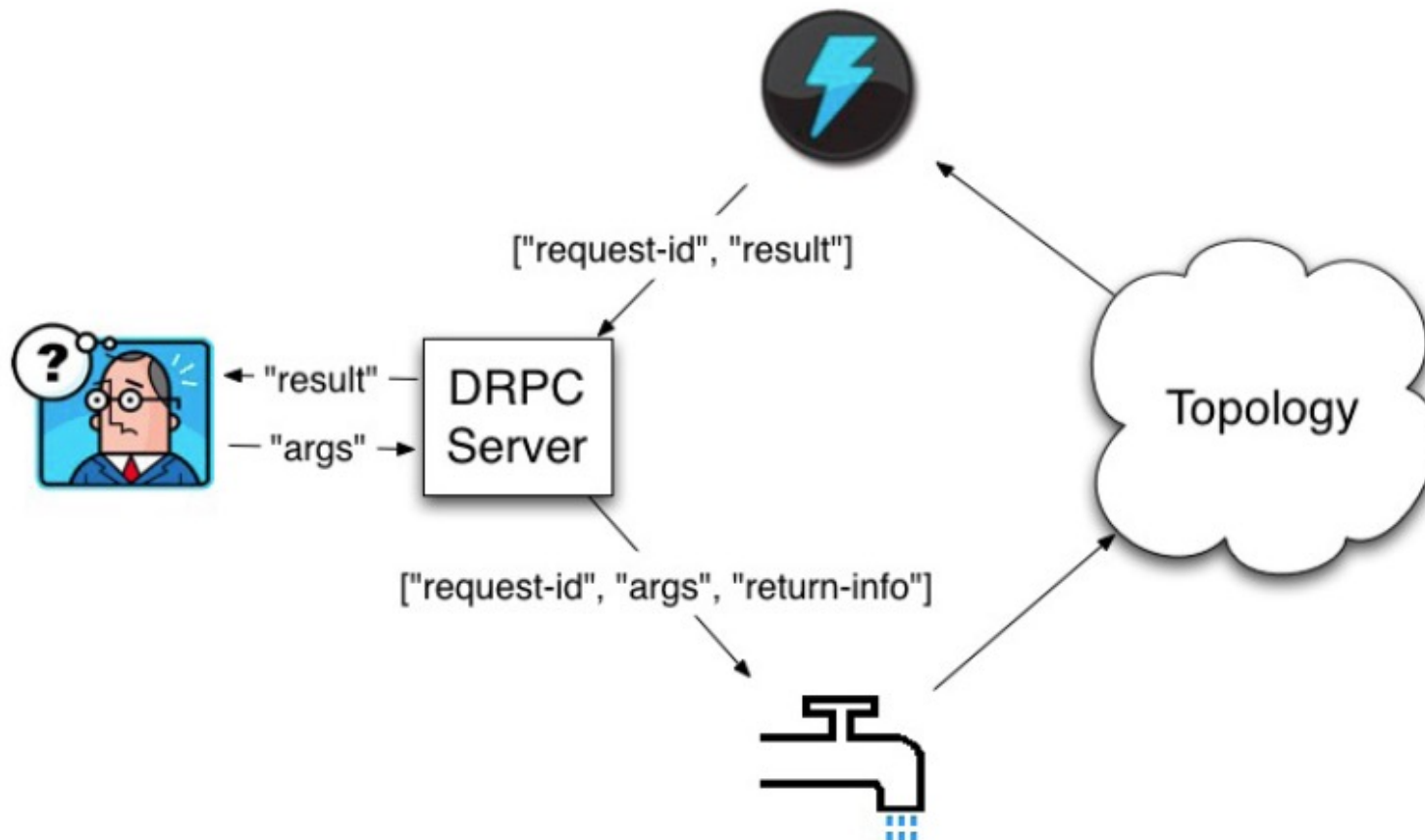
# Distributed Remote Procedure Call (DRPC)

# DRPC

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

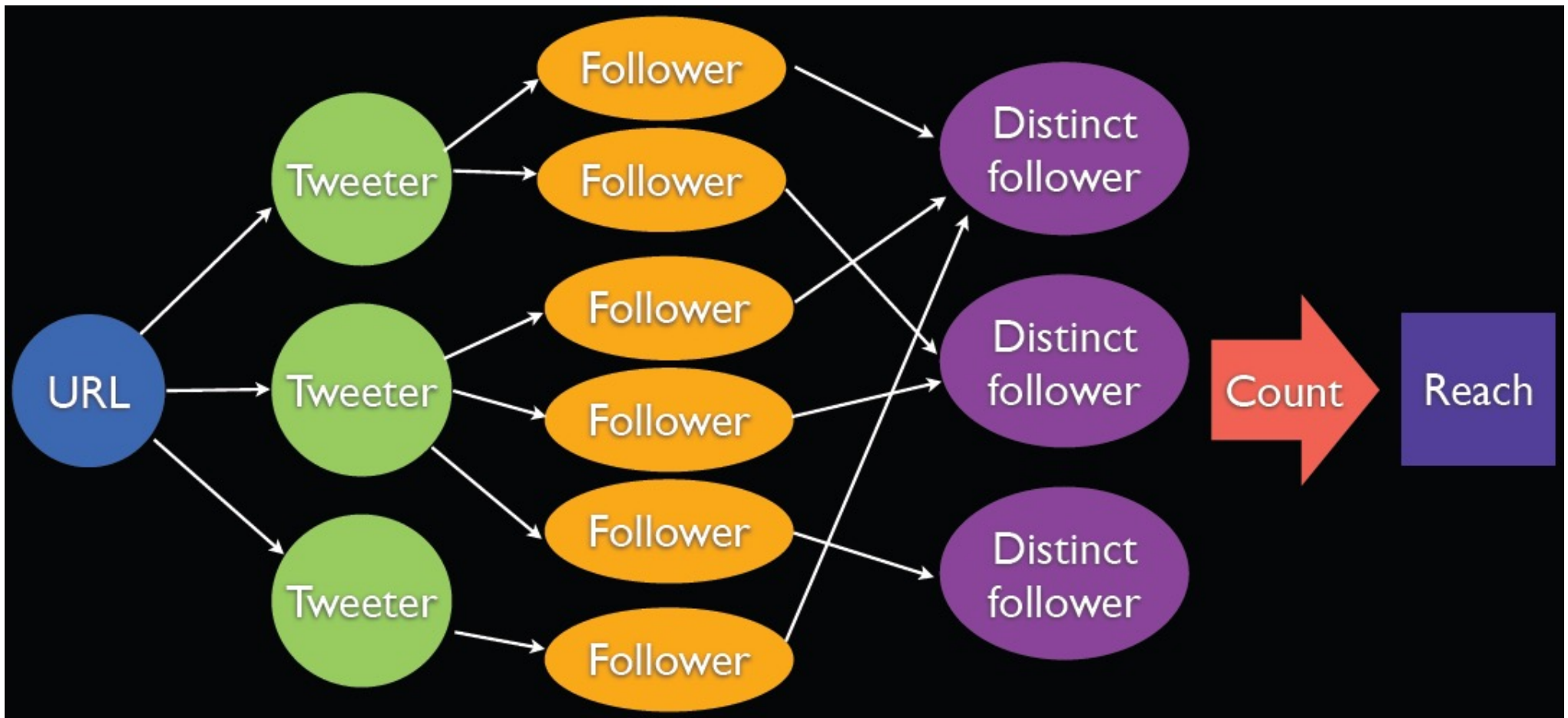




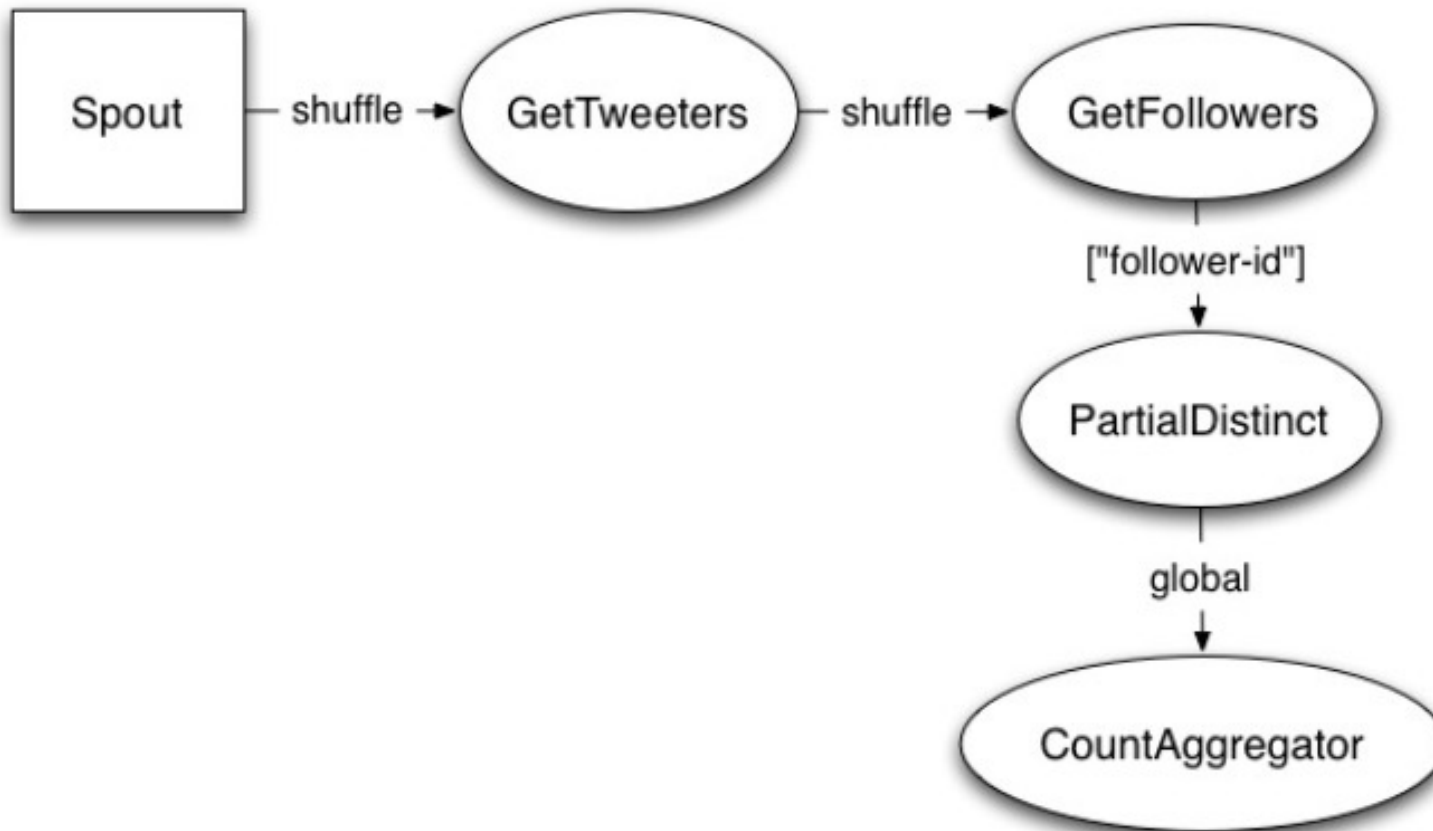
# An Example using DRPC

- 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));
    }
}
```

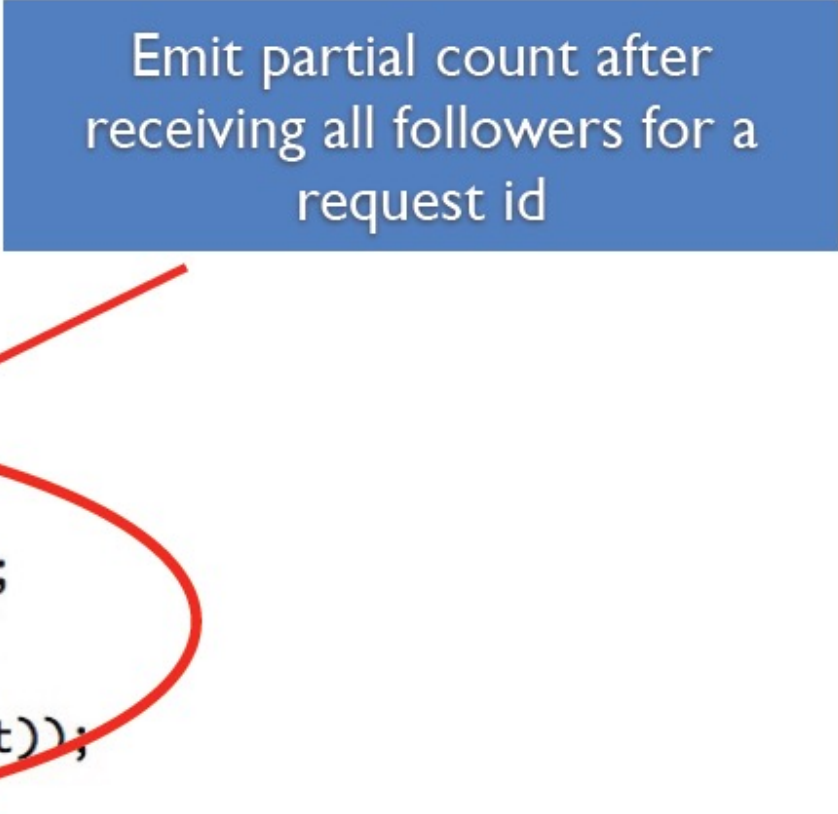
Update followers set when  
receive a new follower

# 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));
    }
}
```



# 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 anywhere -- too bad.

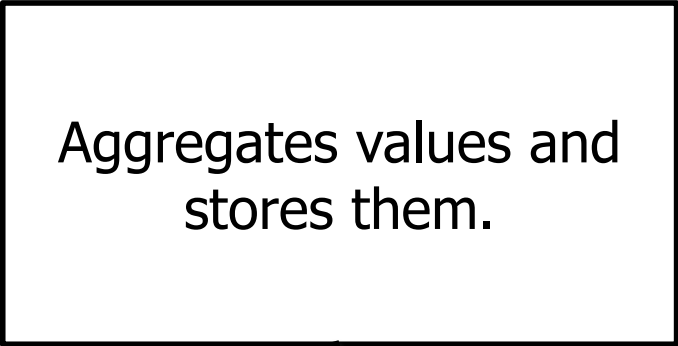


# Enter Trident

- 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("senten  
new Fields("word"))  
        .groupBy(new Fields("word"))  
        .persistentAggregate(new  
MemoryMapState.Factory(), new Count(), new  
Fields("count"))  
        .parallelismHint(6);
```



Aggregates values and stores them.

The code is annotated with a red oval around the `.persistentAggregate` call and a callout box with arrows pointing to the `new Count()` and `new Fields("count")` arguments.

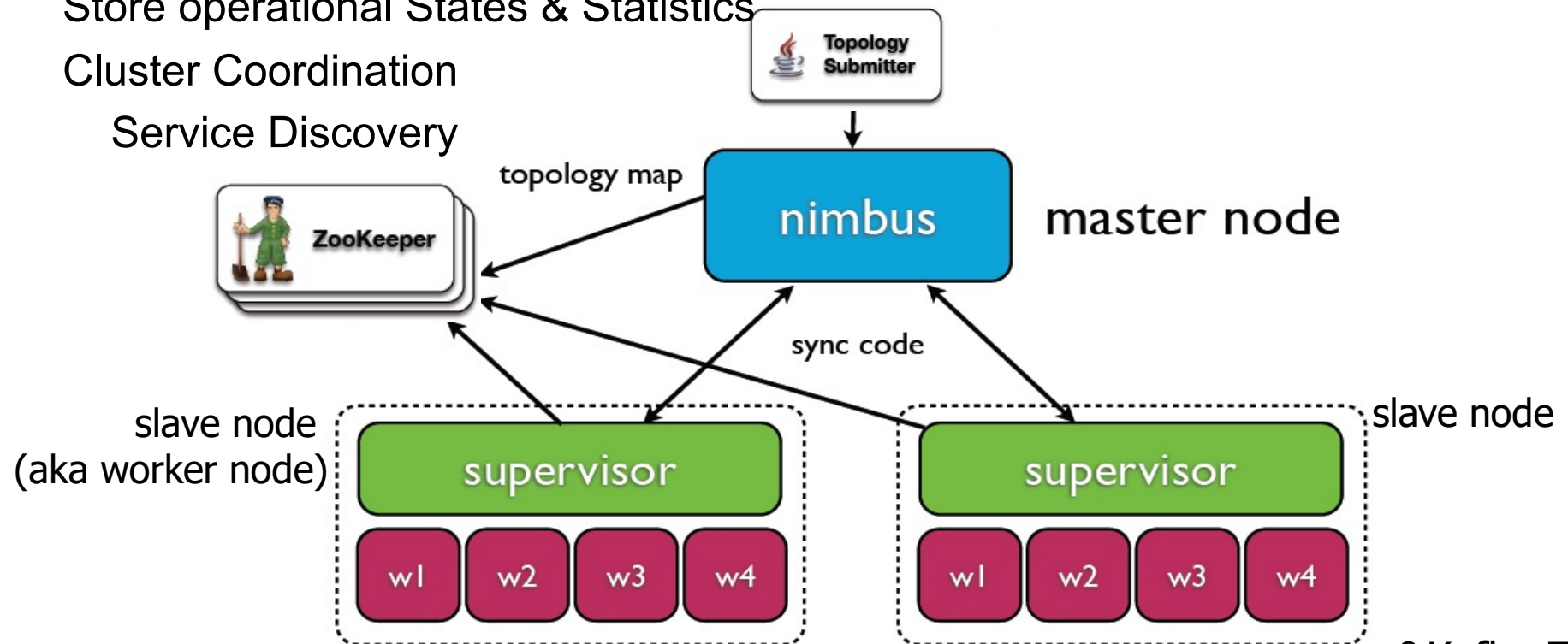
# Trident Example

```
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

# 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



## A Summary on What Storm does

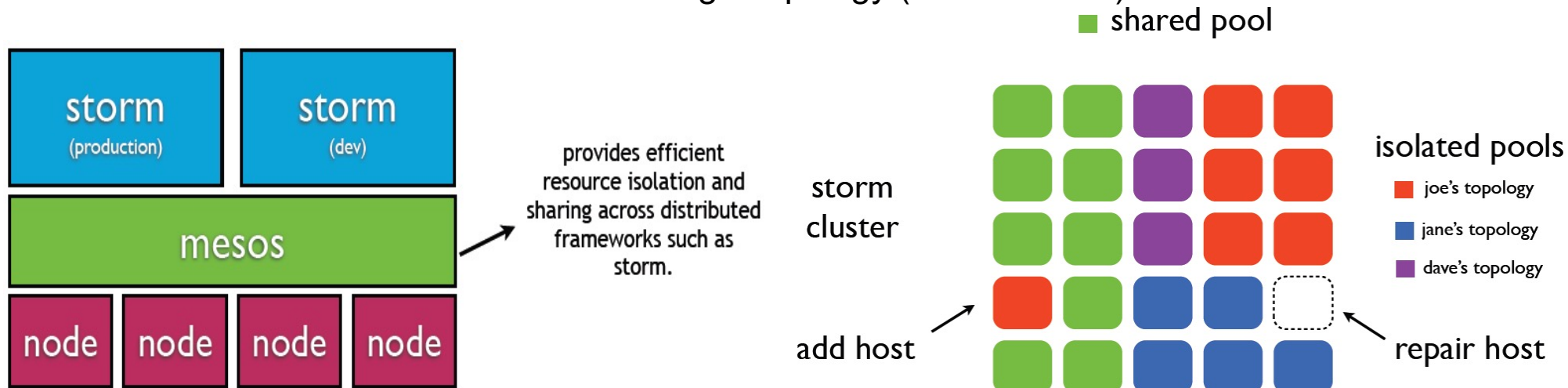
- Distribute Code and Configuration
- Robust Process Management
- Monitors Topologies and Assign Failed Tasks
- Provide Reliability by Tracking Tuple Tree
- Routing and Partitioning of Streams
- Serialization
- Fine-grained Performance Statistics/ Status of Topologies

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

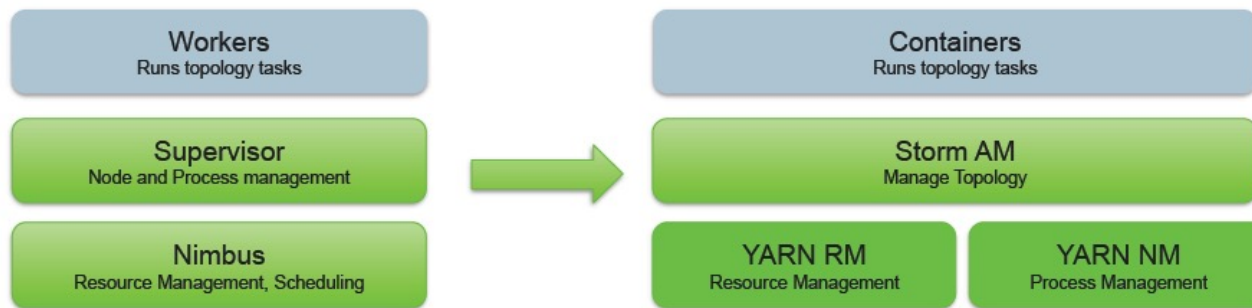
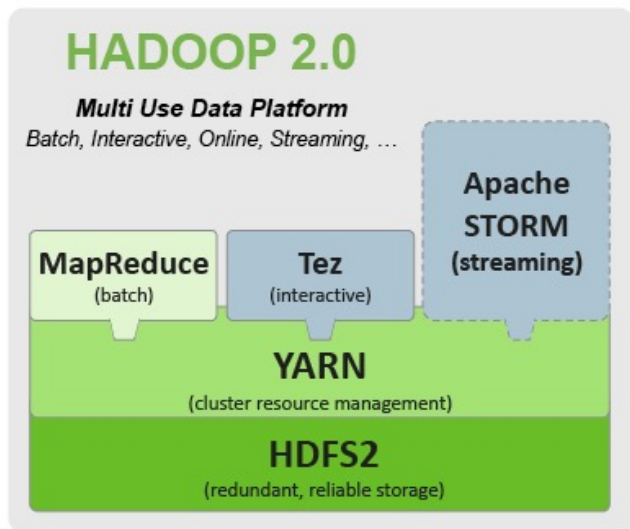
Hadoop v1	Storm	Functions in Storm
JobTracker	<b>Nimbus (only 1)</b>	<ul style="list-style-type: none"><li>▪ distributes code around cluster</li><li>▪ assigns tasks to machines/supervisors</li><li>▪ failure monitoring</li><li>▪ is fail-fast and stateless (you can "kill -9" it)</li></ul>
TaskTracker	<b>Supervisor (many)</b>	<ul style="list-style-type: none"><li>▪ listens for work assigned to its machine</li><li>▪ starts and stops worker processes as necessary based on Nimbus</li><li>▪ is fail-fast and stateless (you can "kill -9" it)</li><li>▪ shuts down worker processes with "kill -9", too</li></ul>
MR job	<b>Topology</b>	<ul style="list-style-type: none"><li>▪ processes messages forever (or until you kill it)</li><li>▪ a running topology consists of many worker processes spread across many machines</li></ul>

# 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)



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



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

# 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  $\geq 3$  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

- 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_childdopts: '-Xmx128m -Djava.net.preferIPv4Stack=true'
storm::nimbus_host: 'nimbus1'
storm::supervisor_childdopts: '-Xmx256m -Djava.net.preferIPv4Stack=true'
storm::worker_childdopts: '-Xmx1024m -Djava.net.preferIPv4Stack=true'
storm::supervisor_slots_ports:
  - 6700
  - 6701
  - 6702
  - 6703
```

# Operating Storm

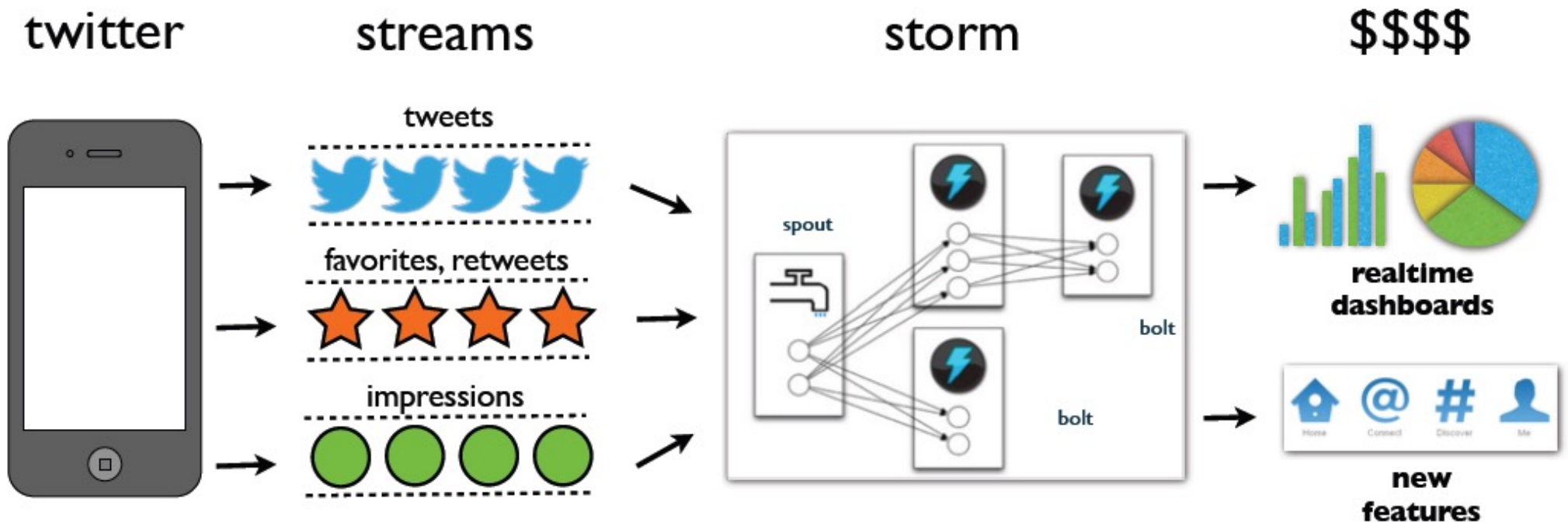
- Typical operations tasks include:
  - Monitoring topologies for Performance and Scalability (P&S) : “Don’t let our pipes blow up!”
    - Tackling P&S in Storm is a joint Ops-Dev effort.
  - Adding or removing slave nodes, i.e. nodes that run Supervisors
  - Apps management: new topologies, swapping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



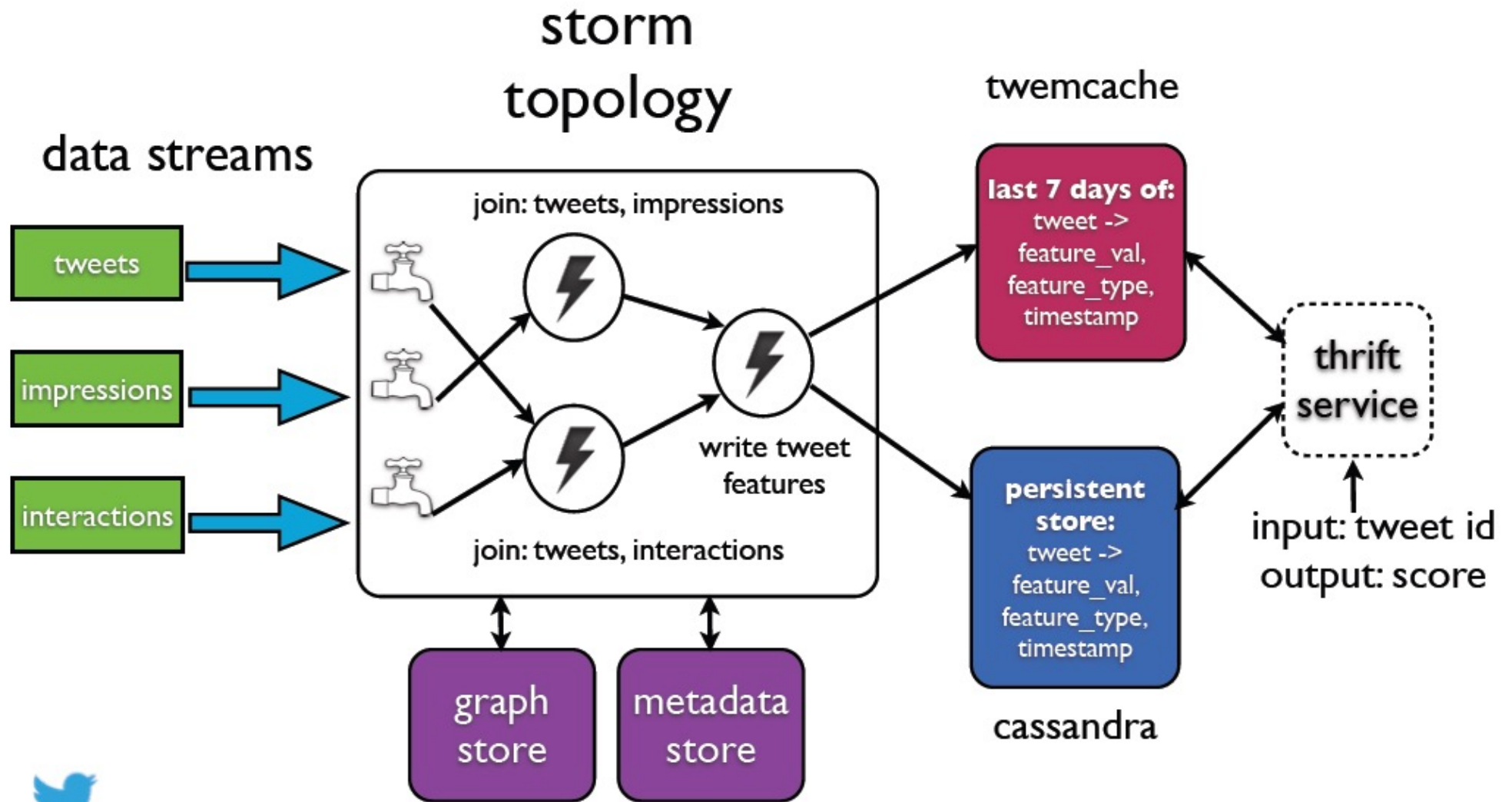
# Stream Processing Applications at Twitter



## 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



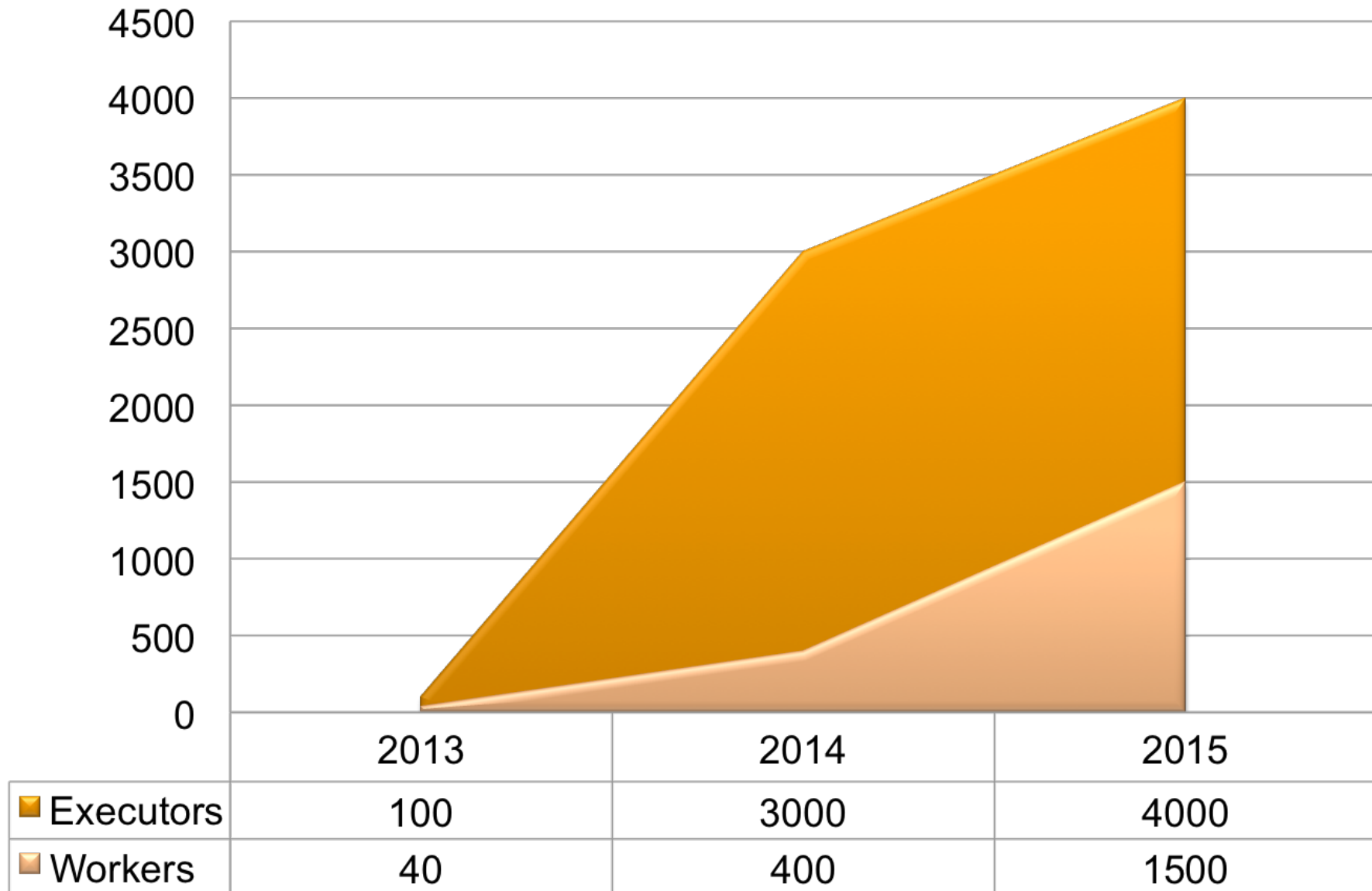
# Storm adoption and use cases

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

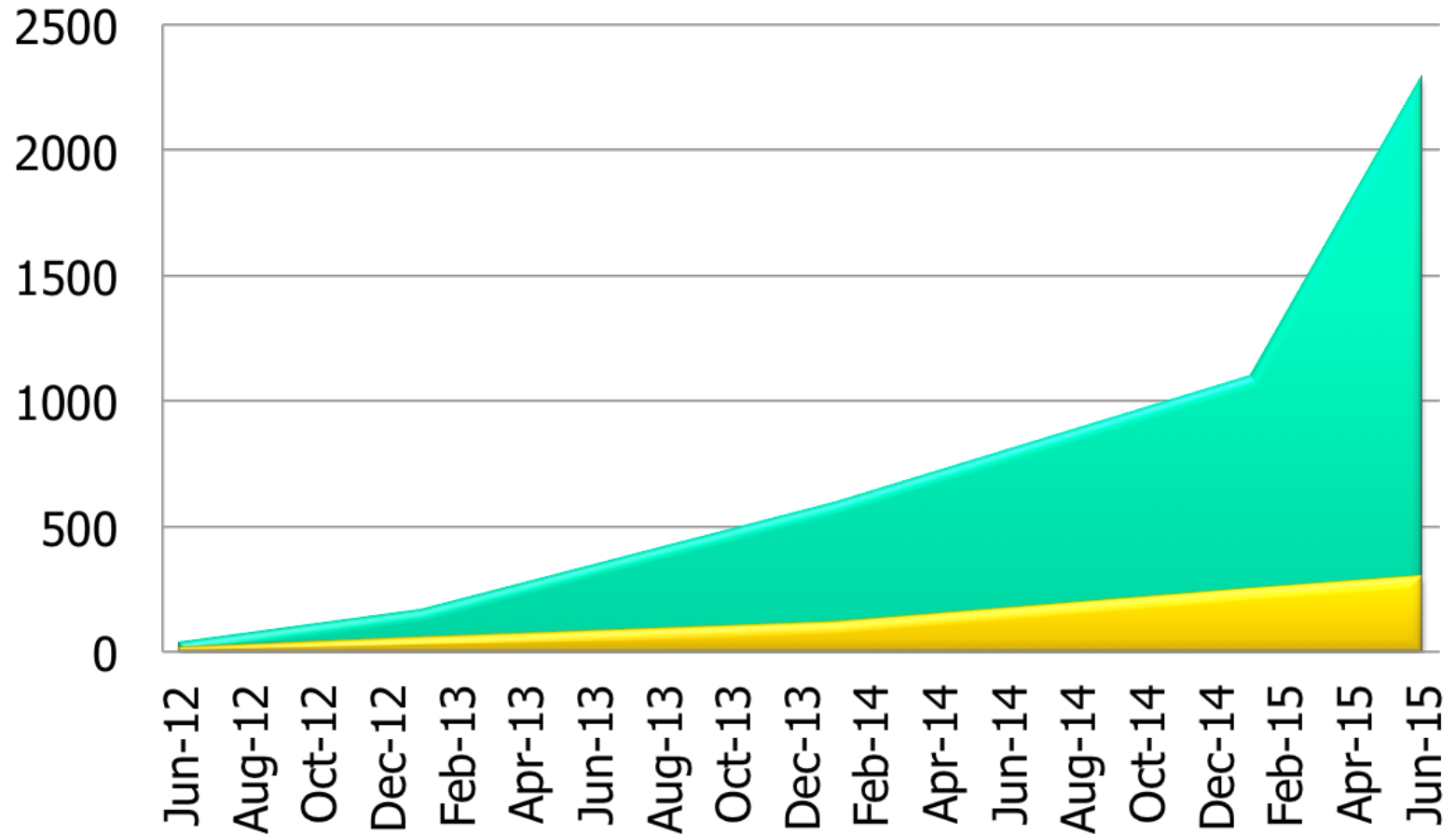
- **Twitter:** personalization, search, revenue optimization, ...
  - 200 nodes, 30 topos, 50B msg/day, avg latency <50ms, Jun 2013
- **Yahoo:** user events, content feeds, and application logs
  - 320 nodes (YARN), 130k msg/s, June 2013
- **Spotify:** recommendation, ads, monitoring, ...
  - v0.8.0,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



# Topology Growth at Yahoo



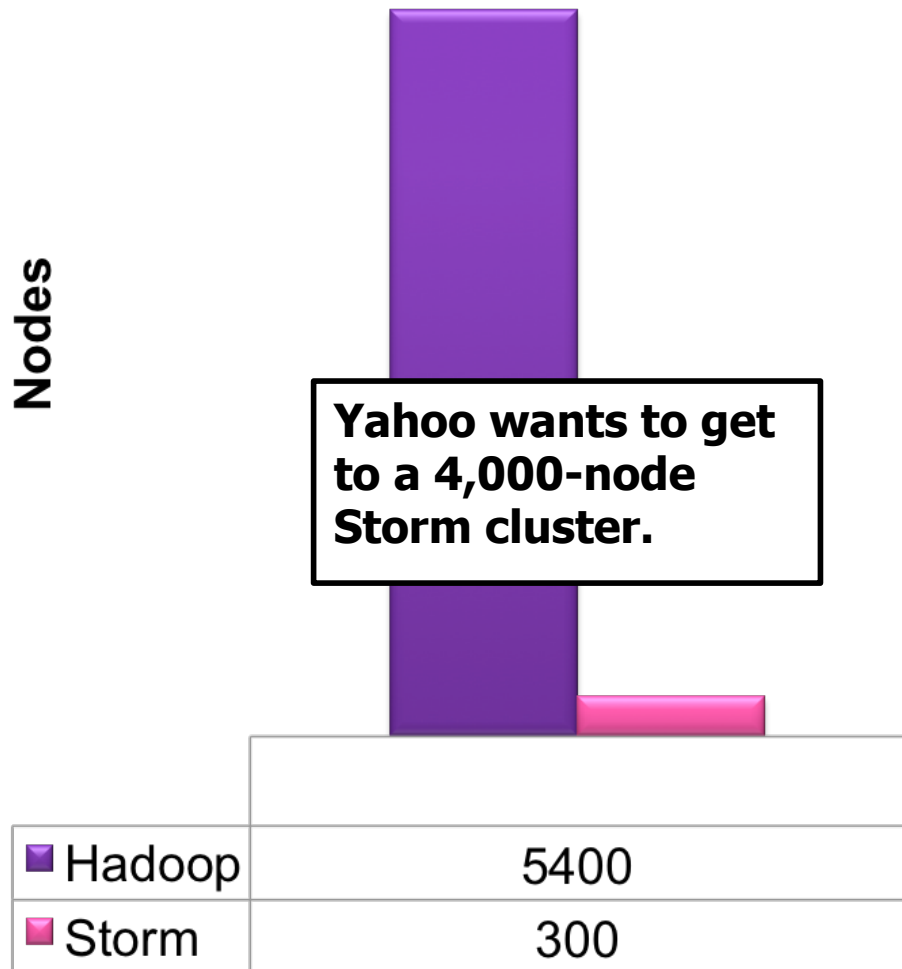
# Cluster Growth at Yahoo



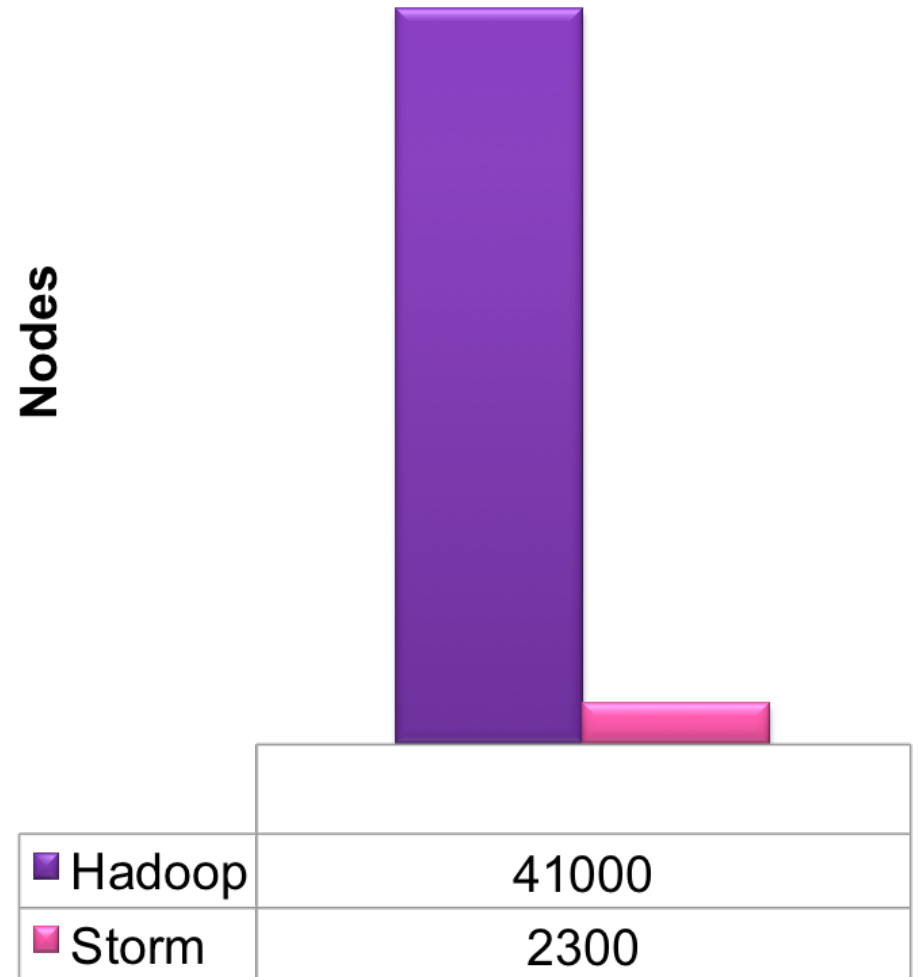
	Jun-12	Jan-13	Jan-14	Jan-15	Jun-15
Total Nodes	40	170	600	1100	2300
Largest Cluster	20	60	120	250	300

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

## Largest Cluster Size



## Total Nodes



# 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.aol.com/2015/06/15/twitter-heron-stream-processing-at-scale/)
    - <http://blog.aol.com/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  
 $O(\text{num\_execs} * \text{resched\_rate})$
- Supervisor  
 $O(\text{num\_supervisors} * \text{hb\_rate})$
- Topology Metrics (worst case)  
 $O(\text{num\_execs} * \text{num\_comps} * \text{num\_streams} * \text{hb\_rate})$

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

## Theoretical Limit:

$80 \text{ MB/sec} / 16 \text{ MB/sec} * 240 \text{ nodes} = 1,200 \text{ nodes}$

# Solution: Pacemaker

(option available since Storm 1.0.0 release, Apr 2016)

An alternative Heartbeat server

Simple Secure In-Memory Store for Worker Heartbeats.

- Removes Disk Limitation
- Writes Scale Linearly

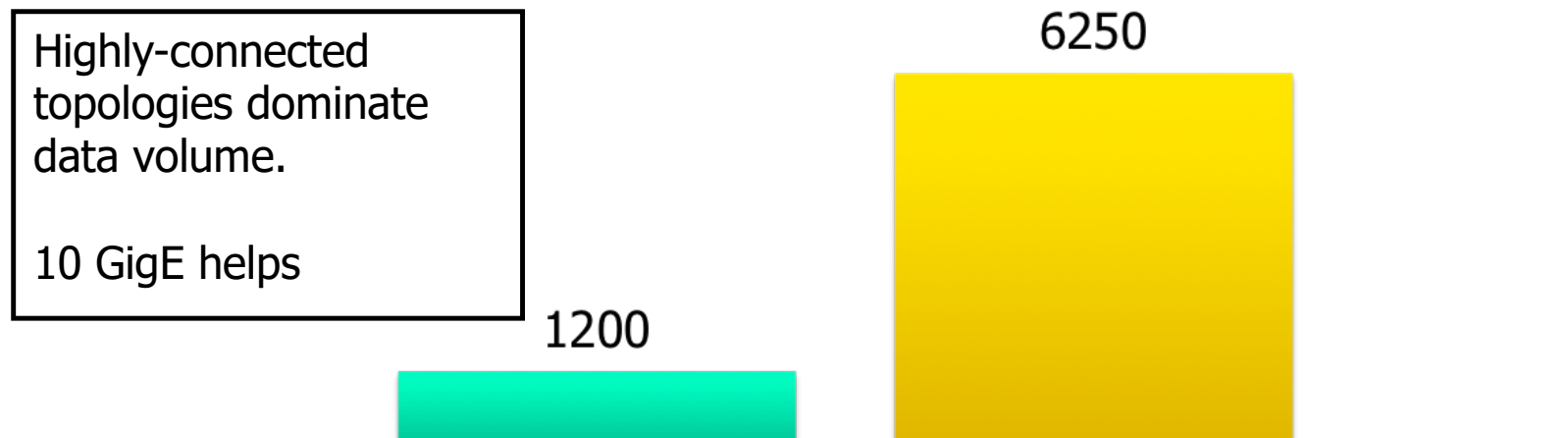
(but nimbus still needs to read it all, ideally in 10 sec or less)

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

$10 \text{ s} / (48 \text{ MB} / 125 \text{ MB/s}) * 240 \text{ nodes} = 6,250 \text{ nodes}$

## Theoretical Maximum Cluster Size

■ Zookeeper   ■ PaceMaker Gigabit



# 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

# Open 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)

# 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)

Solution: Resource & Network Topography Aware Scheduler (Storm V1.0.0, released Apr 2016) ; Also allow Pluggable Scheduler provided by user

## One slow node slows the entire topology.

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



## Other Notable New Features from Storm V1.0.0 Rel. Apr 2016

Automatic Backpressure, Native Streaming Window API,  
Stateful Bolts with Automatic Checkpointing, HA Nimbus

<https://storm.apache.org/2016/04/12/storm100-released.html>

# 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 (Some proto-type effort but...)
- Native Window-based Processing API available from Storm V1.0.0.

# 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 ( <http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf> )
  - Open Source API, BUT NOT implementation
  - Based on technologies from Google's FlumeJava & MillWheel
  - Great stream processing concepts
- Spark Streaming (BDAS of Berkeley/Databricks)
  - Micro-batch processing instead of true real-time streaming
- Microsoft's Naiad, Apache Apex (DataTorrent), Flink, Samza (LinkedIn), Amazon's Kinesis, etc

## 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

# 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
  - [https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.5/bk\\_storm-component-guide/bk\\_storm-component-guide.pdf](https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.5/bk_storm-component-guide/bk_storm-component-guide.pdf)
  - <http://storm.apache.org/releases/1.2.2/index.html>
- Storm Kafka Integration
  - <http://storm.apache.org/releases/1.2.2/storm-kafka-client.html>
- Mailing lists
  - [https://mail-archives.apache.org/mod\\_mbox/storm-user/](https://mail-archives.apache.org/mod_mbox/storm-user/)
- 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)

Apache Kafka  -- Unified Logging Platform

# Acknowledgements

- The slides are adapted from the following source materials:
  - Jay Kreps, “The Log: What every software engineer should know about real-time data’s unifying abstraction,” LinkedIn Engineering Blog, <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>, Dec 2013.
  - Jun Rao, “Intra-cluster Replication for Apache Kafka,” ApacheCon 2013.
  - Joel Koshy, “Building a Real-Time Data Pipeline: Apache Kafka at LinkedIn,” Hadoop Summit, June 2013.
  - Martin Kleppmann, “Apache Samza: Taking stream processing to the next level,” 2014
  - Martin Kleppmann: “Moving faster with data streams: The rise of Samza at LinkedIn.” 14 July 2014. <http://engineering.linkedin.com/stream-processing/moving-faster-data-streams-rise-samza-linkedin>
  - Jeff Holoman, Cloudera, “Kafka Introduction,” Apache Kafka ATL Meetup, 2015.
  - Michael G. Noll, Verisign, “Apache Kafka 0.8 basic training,” July 2014.
  - David Tucker (Confluent), David Ostrovsky (Couchbase), “State of the Streaming Platform 2016 – What’s new in Apache Kafka and Confluent platform,” Nov 2016.
  - Cloudurable, “Introduction to Kafka”, May 2017.
  - Cloudurable, “Kafka Tutorial,” May 2017. <http://cloudurable.com/blog/kafka-tutorial/index.html>
- All copyrights belong to the original authors of the materials.



# Apache Kafka

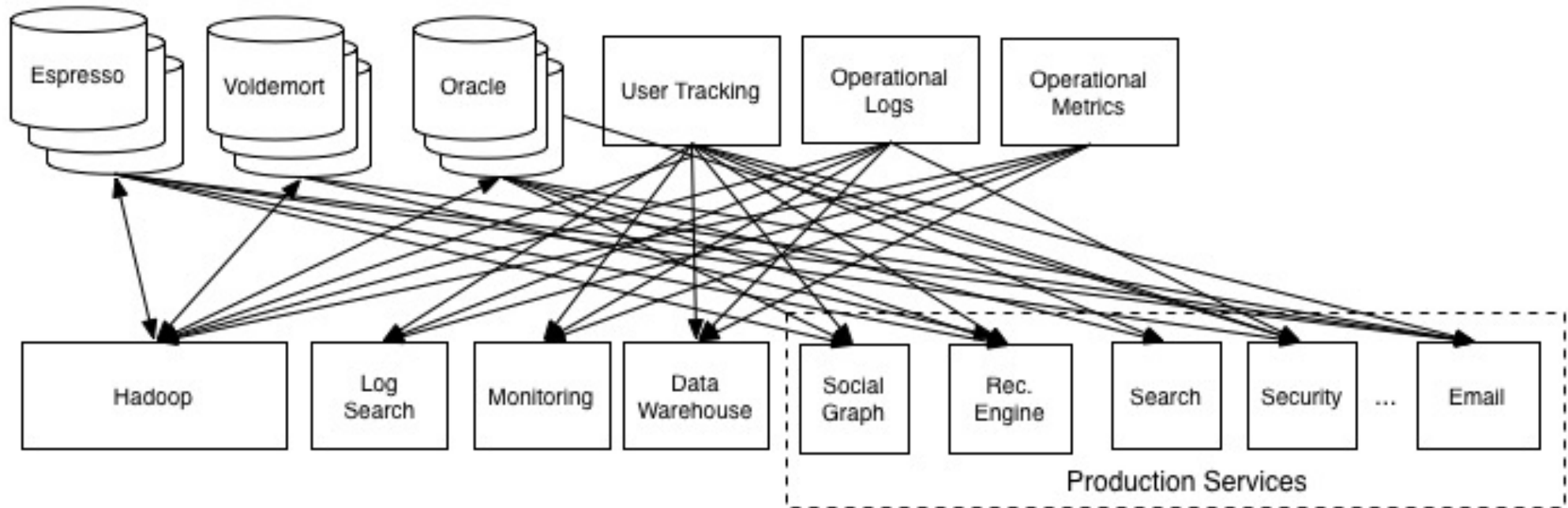
A high-throughput distributed messaging system.

## Outline

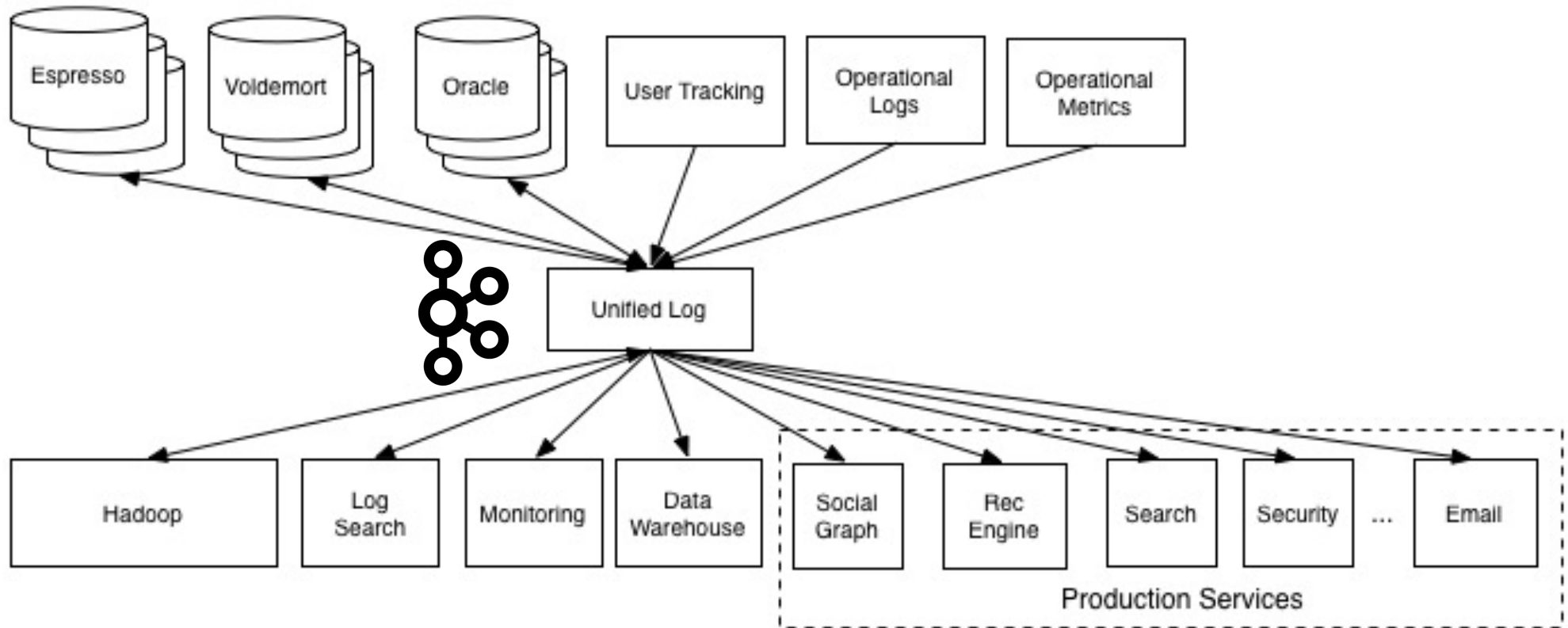
- Motivation for Kafka
- What is Kafka ?
- Real-world Use Cases
- System Architecture
- Key Concepts/ Terminologies in Kafka
- Replication Support
- Performance



# Motivation: LinkedIn Before Kafka



# LinkedIn After Kafka



# Motivation for Building Kafka in LinkedIn

- Goal: To provide a High-Performance, Reliable Distributed Unified Logging Platform for LinkedIn
  - Named after German author/writer Franz Kafka by one of its developers (Jay Kreps) because “It is a system optimized for writing”.
- Originated at LinkedIn, open sourced in early 2011
  - <http://kafka.apache.org/>
  - In 2014, core development team of Kafka at LinkedIn formed **Confluent**, a start-up to further develop a Kafka-centric real-time big data processing platform.
- Implemented in Scala, some Java

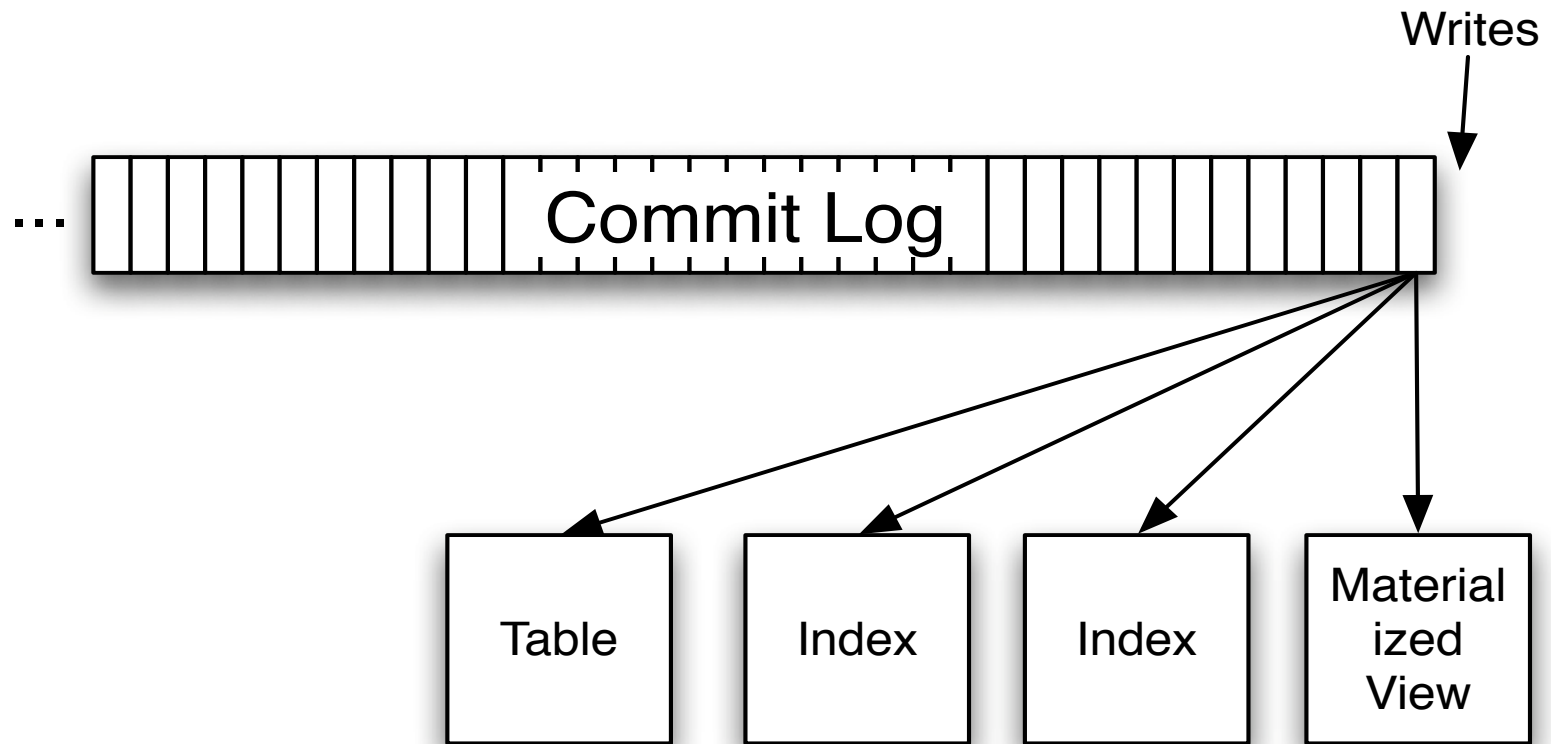
# Requirements for Kafka

- LinkedIn's motivation for Kafka was:
  - “A unified platform for handling all the real-time data feeds a large company might have.”
- Must have:
  - High throughput to support **high volume event feeds**.
  - Support real-time processing of these feeds to create **new, derived feeds**.
  - Support large data backlogs to handle periodic ingestion from **offline systems**.
  - Support low-latency delivery to handle more traditional **messaging use cases**.
  - Guarantee **fault-tolerance** in the presence of machine failures.

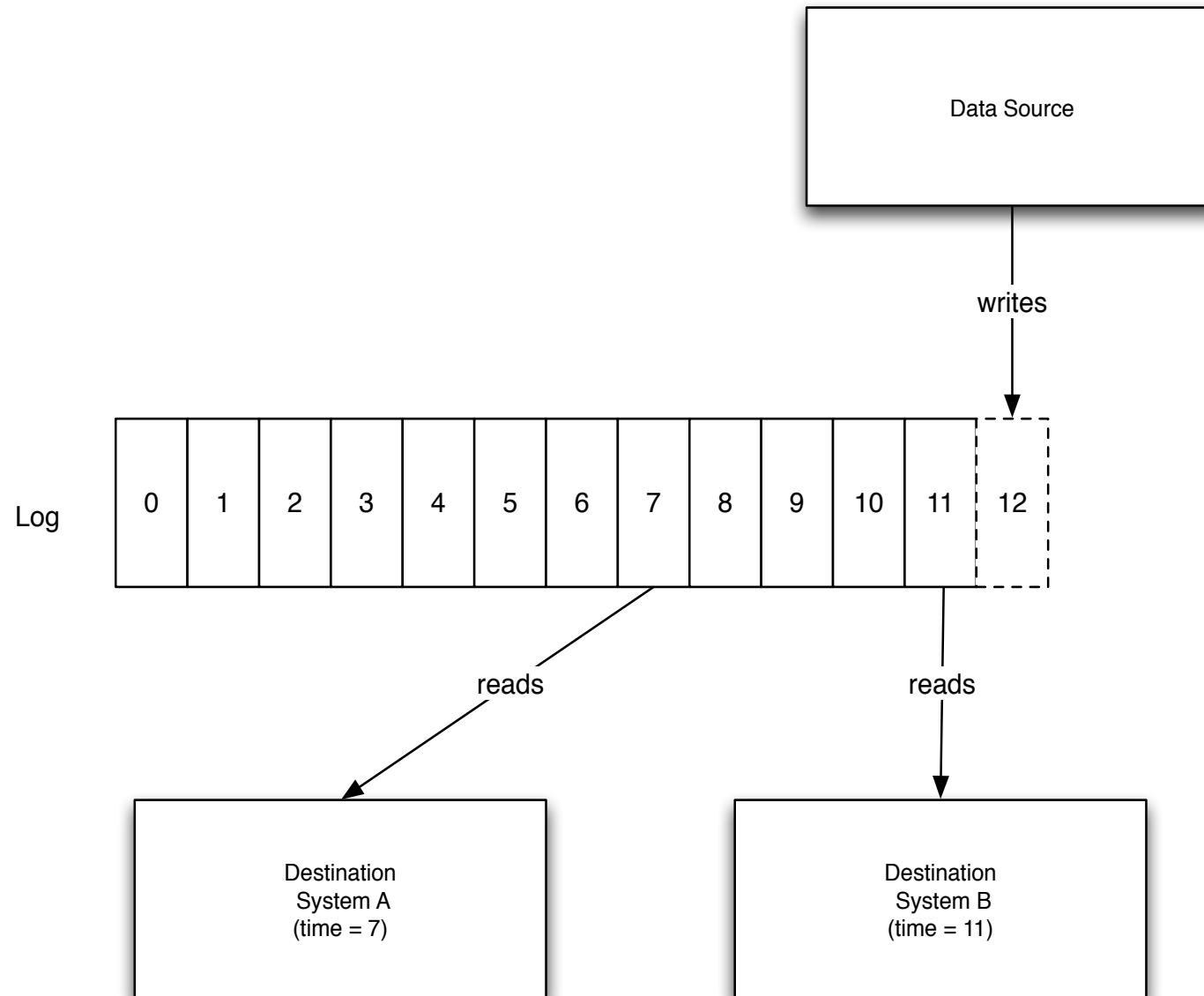
# What is Kafka ?

- A Distributed Unified Logging Platform
  - Kafka maintains feeds (streams) of records (messages) organized in different categories called topics.
  - Support the “Publish and Subscribe” model for streams of records (messages)
  - Fault Tolerant Storage via Replication to multiple servers
  - Process records as they occur
  - Fast, Efficient I/O, Batching, Compression and more
  - Decouple the Producers and Consumers of those records (messages)

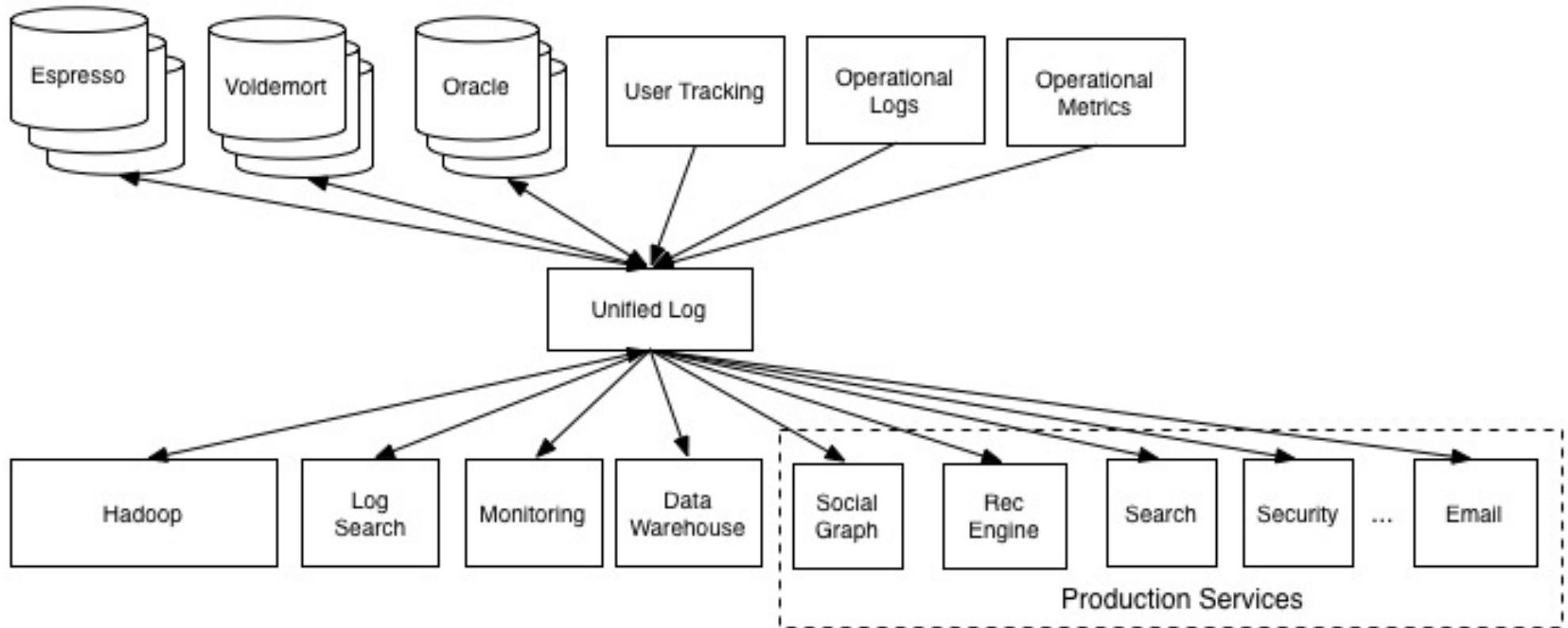
# What is a commit log?



# The Log as a Messaging system



# Linkedin After Kafka





# Kafka @ LinkedIn, 2014



<https://twitter.com/SalesforceEng/status/466033231800713216/photo/1>  
<http://www.hakka Labs.co/articles/site-reliability-engineering-linkedin-kafka-service>

# Kafka @ LinkedIn, 2014

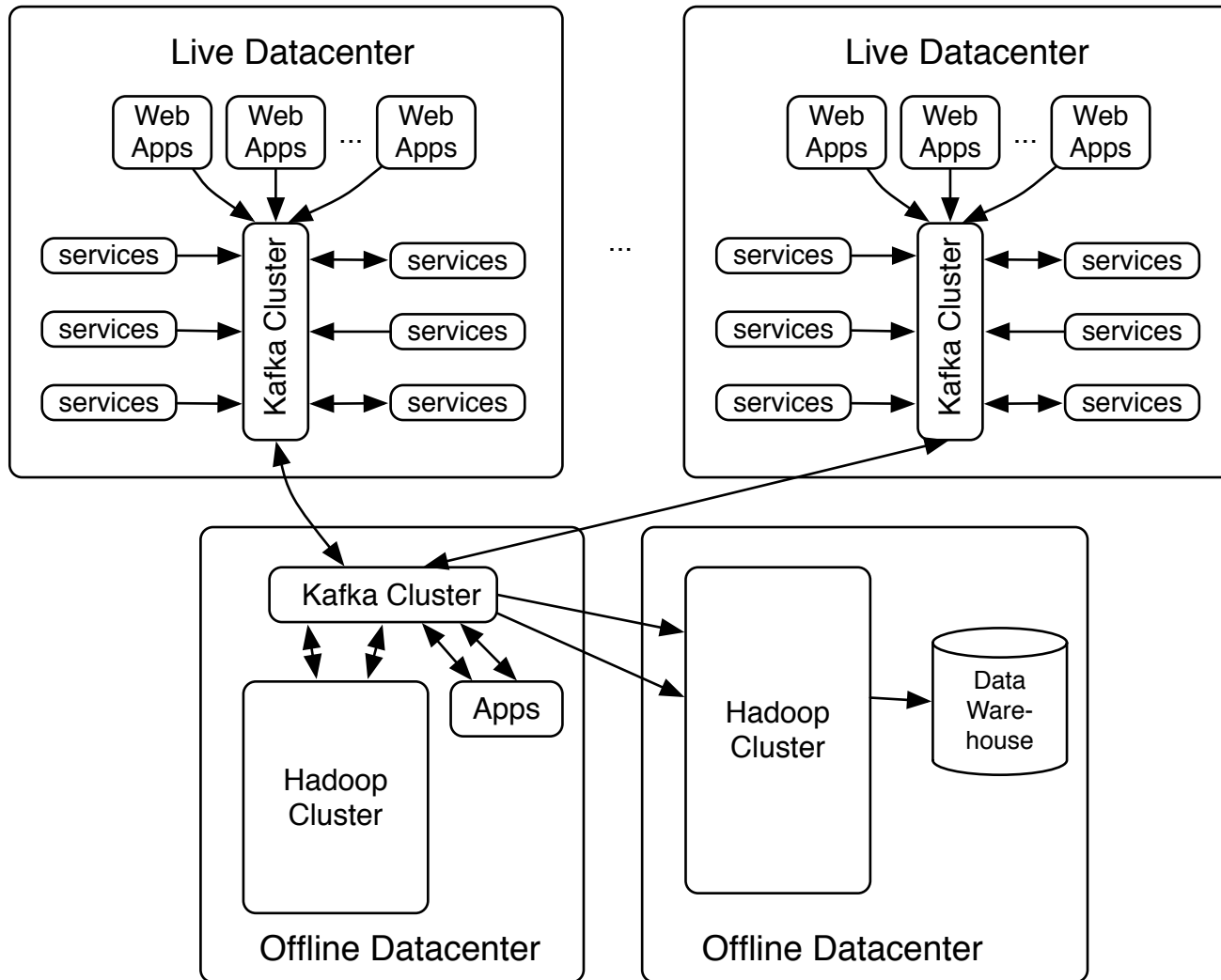
- Multiple data centers, multiple clusters
  - Mirroring between clusters / data centers
- What type of data is being transported through Kafka?
  - **Metrics**: operational telemetry data
  - **Tracking**: everything a LinkedIn.com user does
  - **Queuing**: between LinkedIn apps, e.g. for sending emails
- To transport data from LinkedIn's apps to Hadoop, and back
- In total ~ **200 billion events/day** via Kafka
  - Tens of thousands of data producers, thousands of consumers
  - 7 million events/sec (write), 35 million events/sec (read) <<< may include replicated events
  - But: LinkedIn is not even the largest Kafka user anymore as of 2014

<http://www.hakka-labs.co/articles/site-reliability-engineering-linkedin-kafka-service>

<http://www.slideshare.net/JayKreps1/i-32858698>

<http://search-hadoop.com/m/4TaT4qAFQW1>

# Kafka Usage at LinkedIn (circa 2013)



# Kafka @ LinkedIn, 2014

“For reference, here are the stats on one of LinkedIn's busiest clusters (at peak):

15 brokers

15,500 partitions (replication factor 2)

400,000 msg/s inbound

70 MB/s inbound

400 MB/s outbound”

<https://kafka.apache.org/documentation.html#java>

# General Use Cases of Kafka

Major Role (via the original Kafka “Core components”):

- Log Aggregation
- Capture and Ingest Data into Spark/ Hadoop/ Storm/ Flink etc
- Command-Query Responsibility Segregation (CRQS), Replay, Error Recovery
- Guaranteed Distributed Commit Log for in-memory computing
- Metrics Collection and Monitoring

Supporting Role (with recent extensions from Confluent):

- Stream Processing
- Website Activity Tracking
- Real Time Analytics

# Sample Use Cases of Kafka

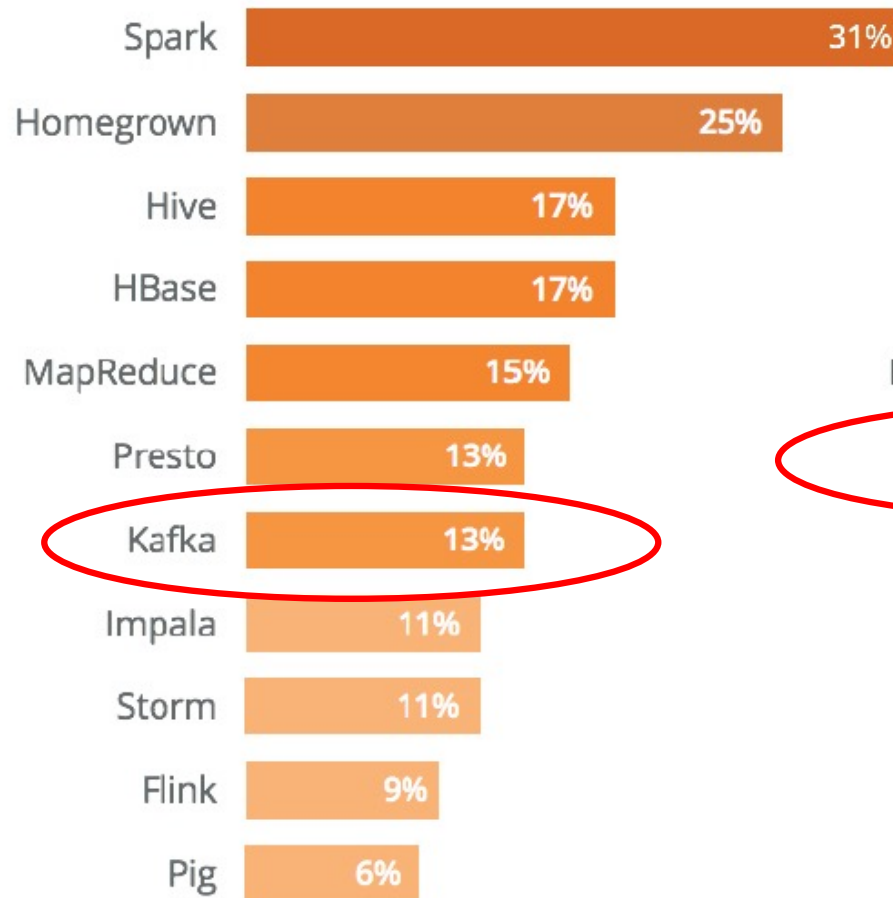
- LinkedIn: Activity streams, Operational Metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/sec) (circa May 2014)
- Netflix: Real-time Monitoring and Event Processing
- Twitter: Use it with Storm for stream processing pipelines
- Spotify: Log delivery (for 4 hrs down to 10sec), Hadoop
- Loggy: Log collection and processing
- Mozilla: Telemetry data
- Square: Kafka as “data-bus” to move all system events to various Square datacenters (logs, custom events, metrics, etc). Outputs to other systems, e.g. Alert generation

# Other Users of Kafka

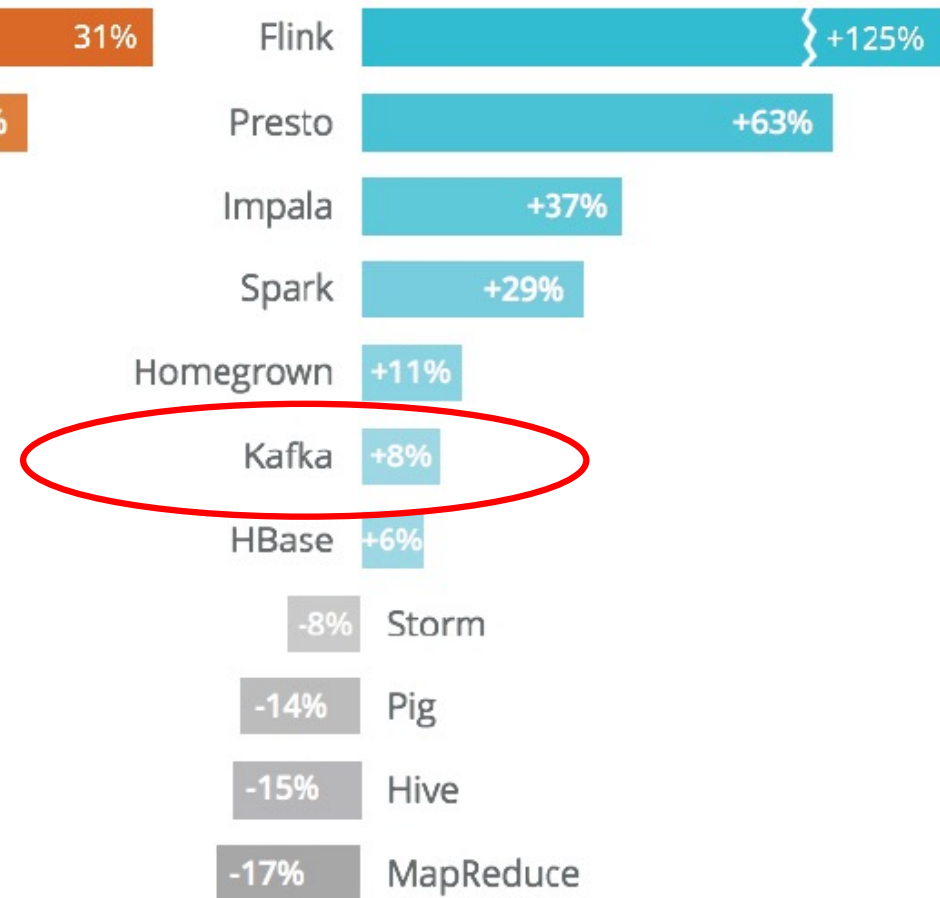
- 1/3 of all Fortunate 500 companies
  - Top 10 Travel companies ; 7 of Top 10 Banks ; 8 of Top 10 Insurance companies ; 9 of Top 10 Telecom coms.
- Airbnb, Uber, Tumbler, Goldman Sachs, PayPal, Cisco, CloudFlare, etc.
- LinkedIn, Microsoft and Netflix process 4 comma messages a day with Kafka (1,000,000,000,000) (circa 2017)

# Big Data Frameworks Adoption Trends

## Frameworks in Use in 2018

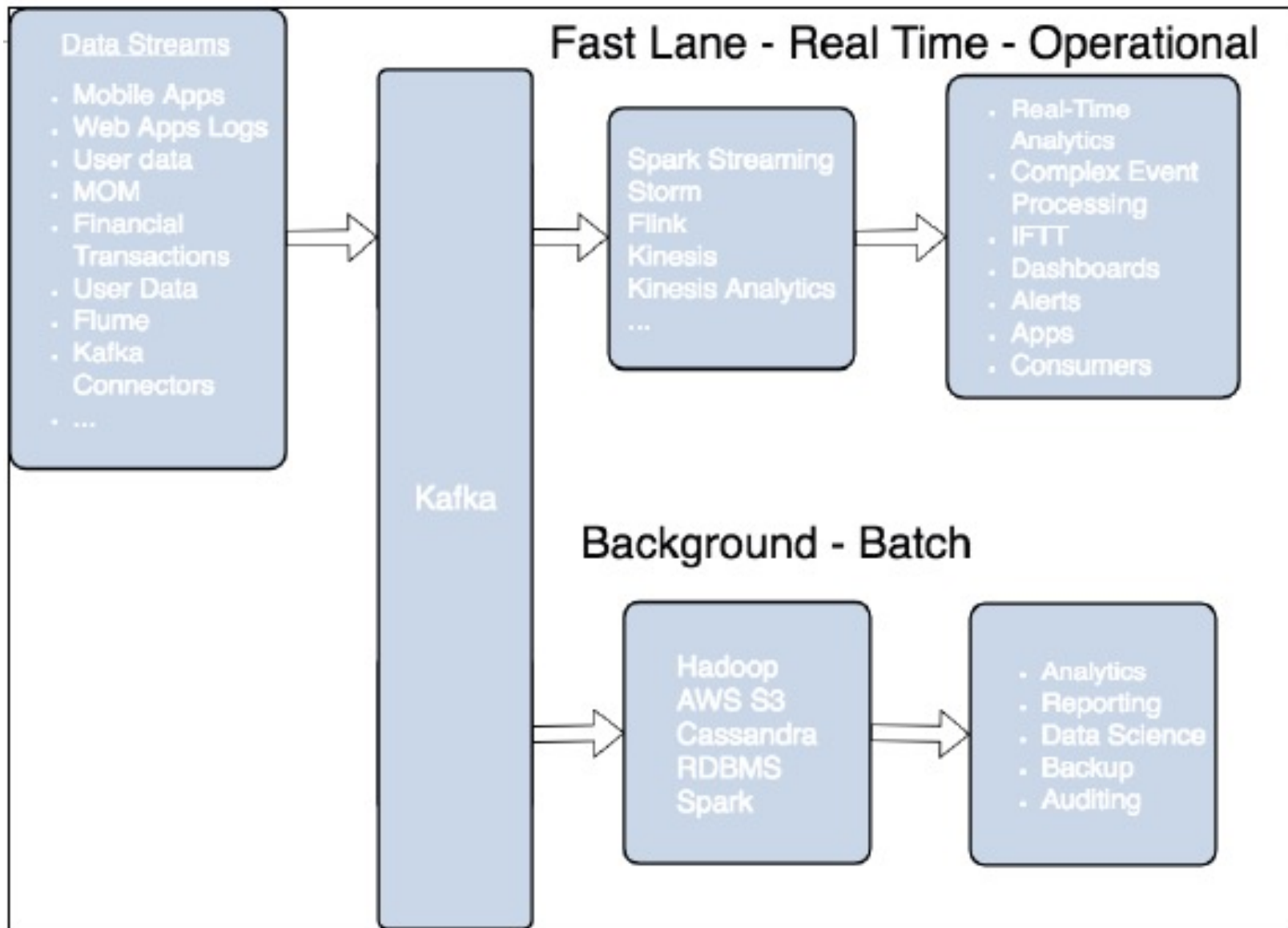


## Percent Change from 2017





# Typical Service Architecture with Kafka



# Kafka + X for processing the data?

- Kafka + **Storm** often used in combination, e.g. Twitter

- Kafka + **custom**

- “Normal” Java multi-threaded setups
- Akka actors with Scala or Java, e.g. Ooyala

- Additional “partners”:

- **Samza** (since Aug '13) – also by LinkedIn
- **Spark Streaming**, part of Spark (since Feb '13)

- Kafka + **Camus** for Kafka->Hadoop ingestion

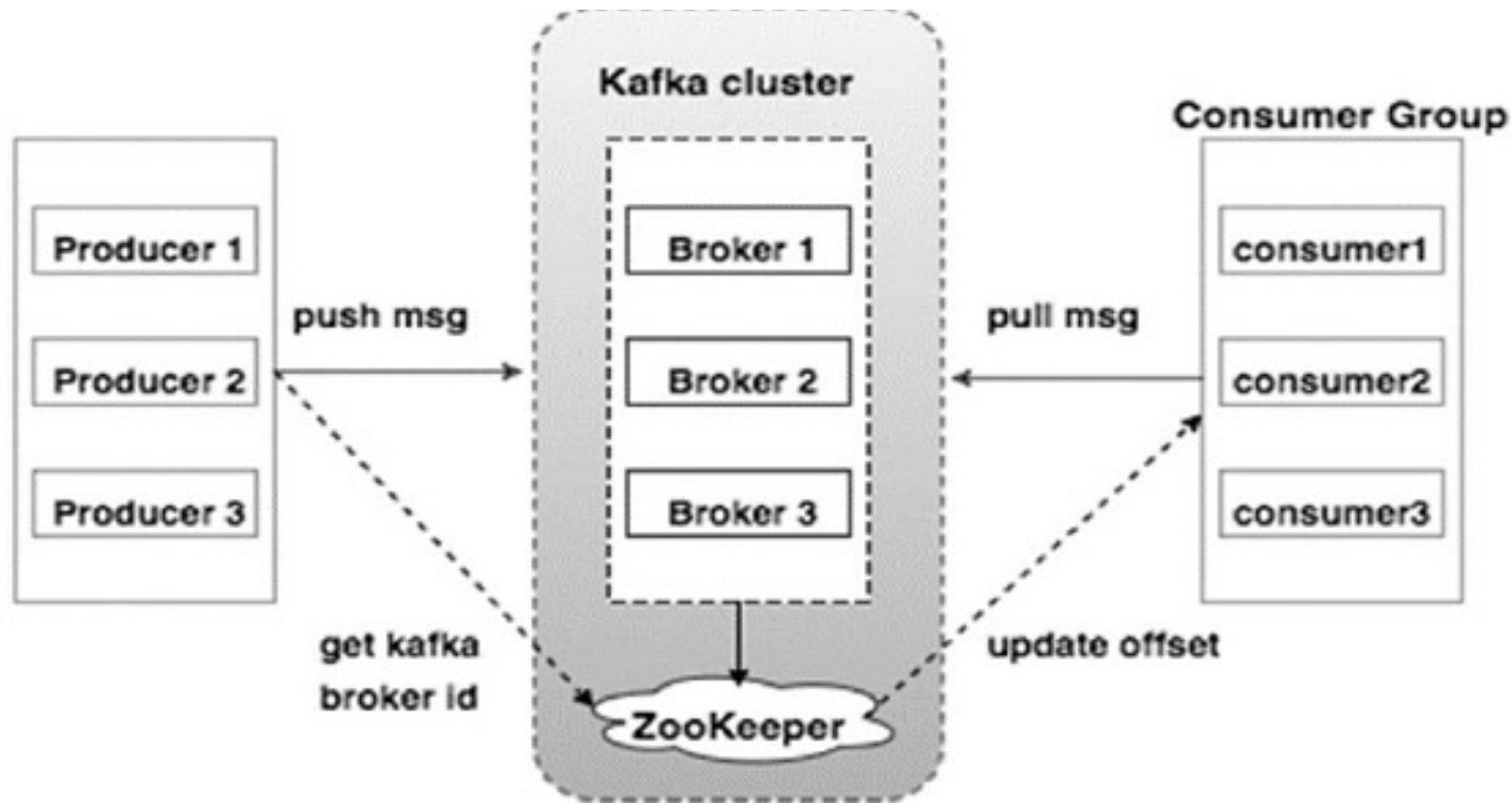
- Camus phased out/ replaced by **Globbin**

<https://cwiki.apache.org/confluence/display/KAFKA/Powered+By>

- Kafka (core) + extended Open-source frameworks from **Confluent** e.g.

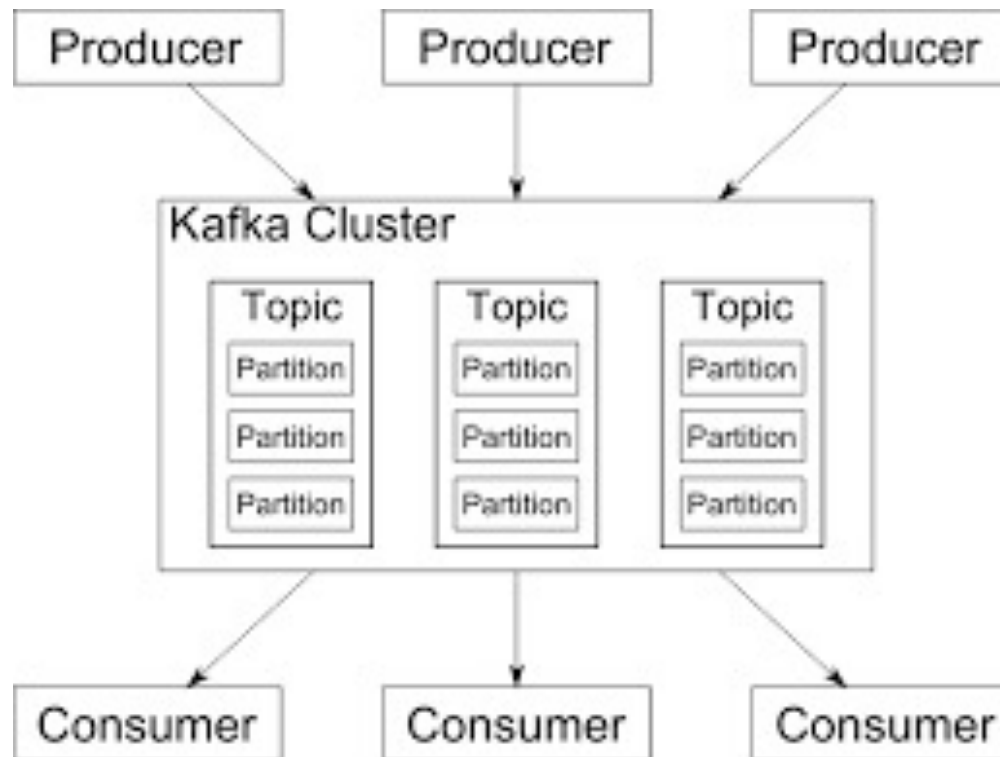
- KSQL (Kafka SQL), Kafka Streams, Kafka Connect & Connectors, Schema Registry, REST Proxy, MQTT Proxy ;

# System Architecture of Kafka (its Core Components)



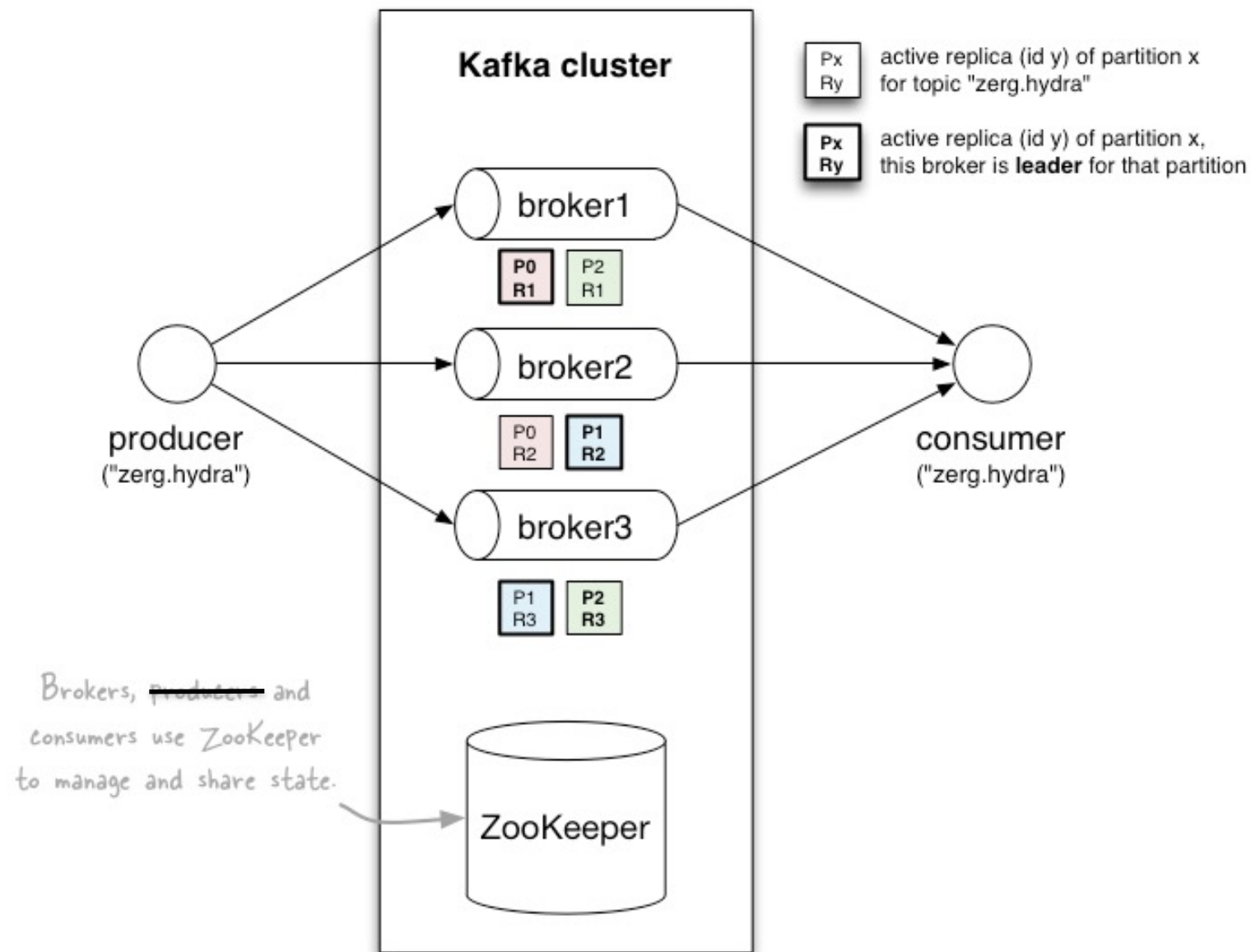
- **Broker:** Kafka server that runs in a Kafka Cluster. Each Cluster has 1+ Broker ; Each Broker has a Unique Broker ID
- **ZooKeeper:** Coordinate Brokers/Cluster topology:
  - Stable storage for Cluster configuration
  - Election for Broker and Partition Leaders ; Coordinate Cluster changes
  - Used by Consumers to track message (reading) offsets in v0.8 [replaced by the use of Special Topics in v0.9]

# Key Concepts/ Terminologies of Kafka



- **Producers** write data to **Brokers**.
- **Consumers** read data from **Brokers**.
- Data is stored in **Topics**.
- **Topics** are split into **Partitions**, which are **Replicated**.
- **All this is distributed.**

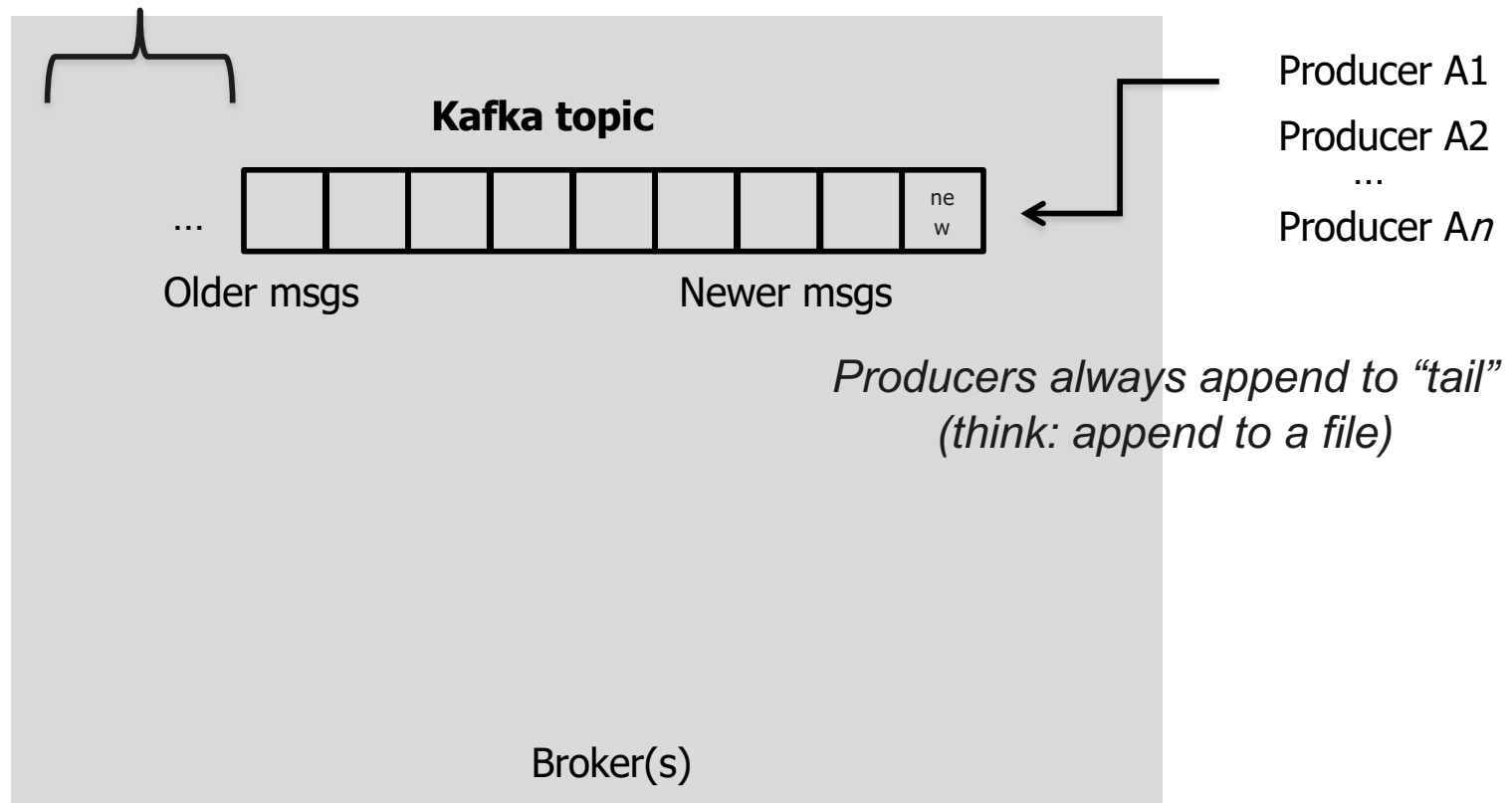
# Illustration of a Multi-Broker Kafka Cluster



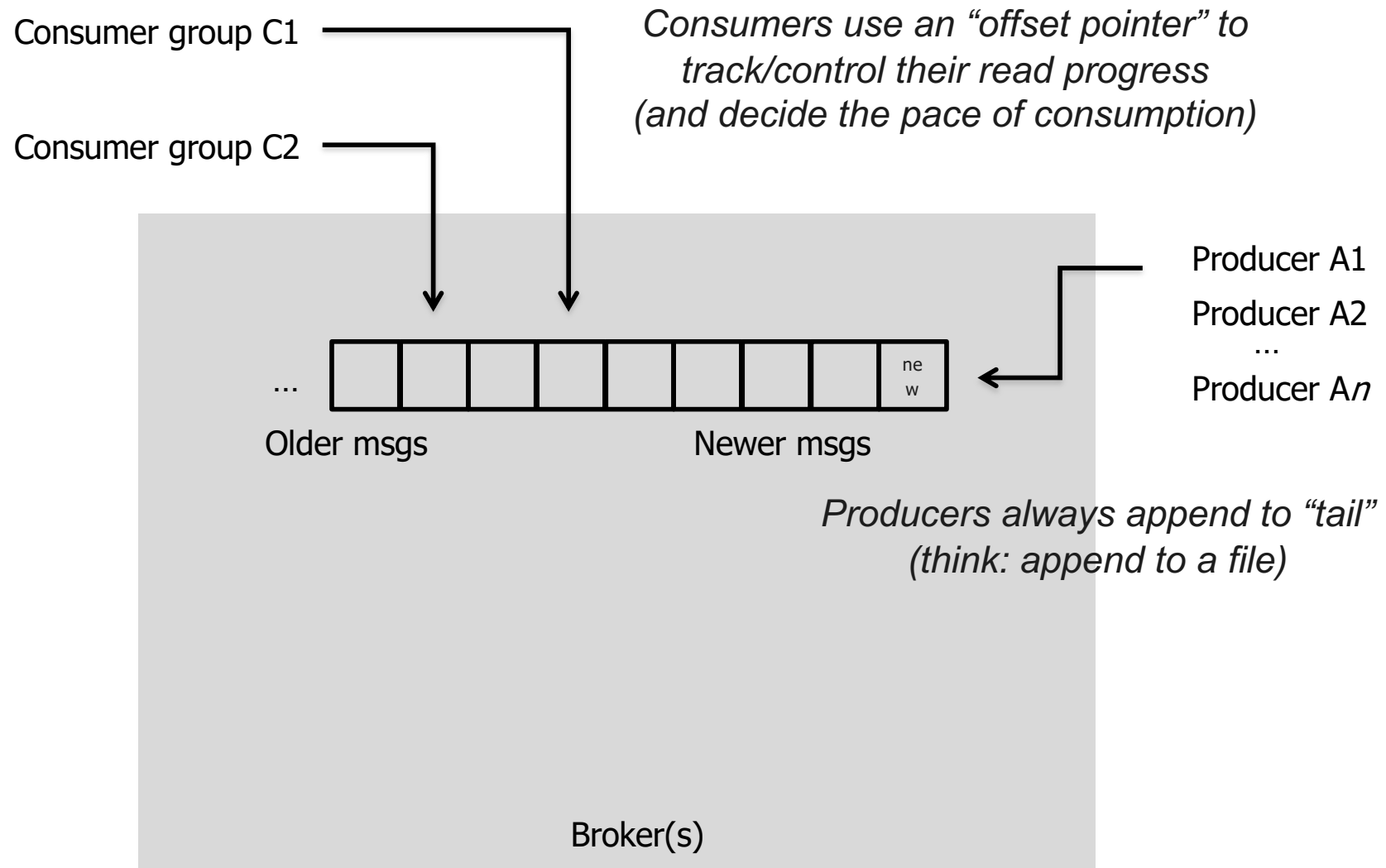
# Topics

- **Topic:** feed name to which messages are published
  - Example: “zerg.hydra”

*Kafka prunes “head” based on **age** or **max size** or “key”*



# Topics



# Topics

- Creating a topic

- CLI (following example for Kafka version 0.8x only)

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --create --topic zerg.hydra \  
  --partitions 3 --replication-factor 2 \  
  --config x=y
```

- API

<https://github.com/miguno/kafka-storm-starter>

- Auto-create via `auto.create.topics.enable = true`

- Modifying a topic

- [https://kafka.apache.org/documentation.html#basic\\_ops\\_modify\\_topic](https://kafka.apache.org/documentation.html#basic_ops_modify_topic)

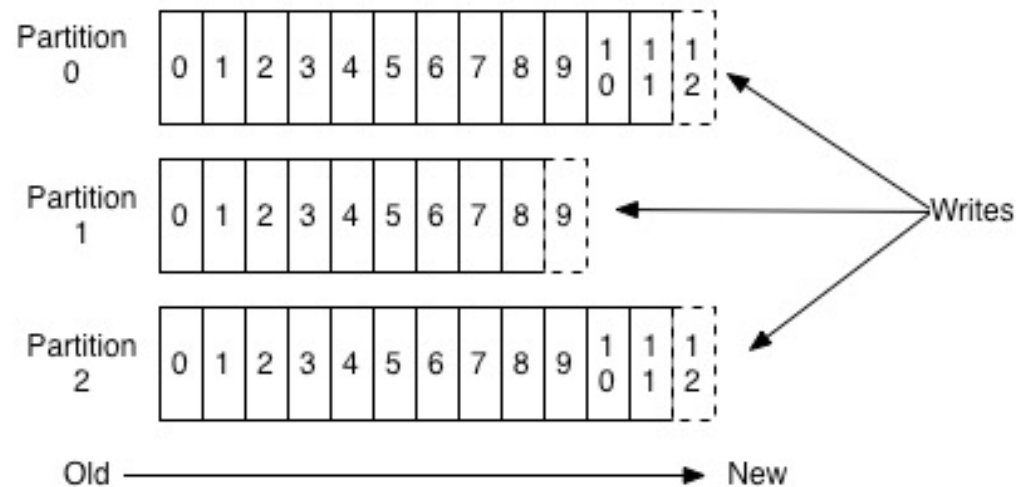
- Deleting a topic



# Partitions

- A topic consists of **partitions**.
- Partition: **ordered + immutable** sequence of messages that is continually appended to

## Anatomy of a Topic

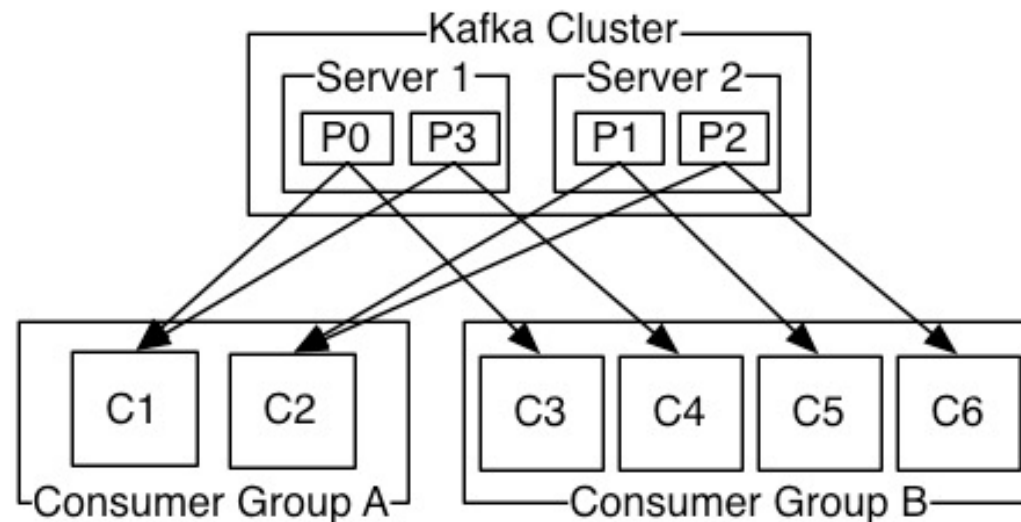


# Producers

- Producers publish data to the **topics** of their choice.
- Producer is responsible for choosing which message to assign to which **partition** within the topic.
- This can be done in a **round-robin** fashion simply to balance load or it can be done according to some **semantic partition function**

# Partitions

- #partitions of a topic is configurable
- #partitions determines **max** # of consumers (threads) in each Consumer Group
  - Parallelism (same idea as sharding in database)
  - Cf. parallelism of Storm's KafkaSpout via `builder.setSpout(,,N)`

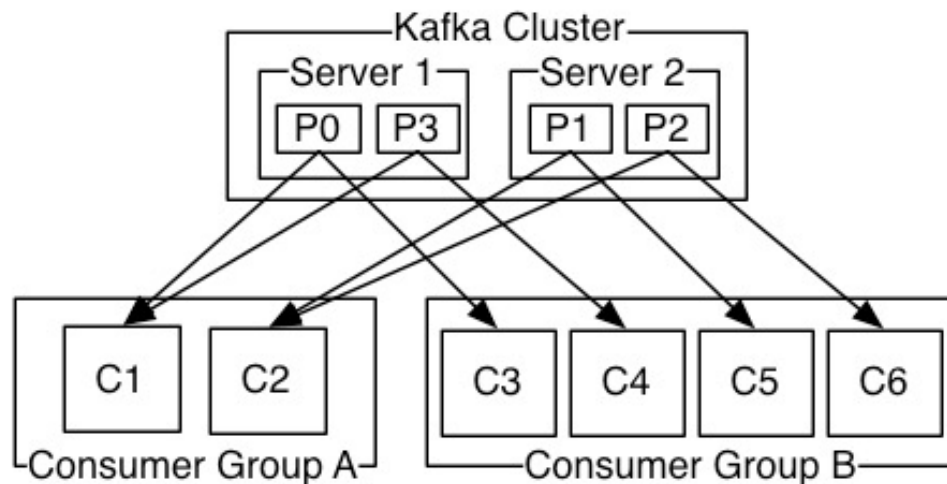


- Consumer group A, with 2 consumers, reads from a 4-partition topic
- Consumer group B, with 4 consumers, reads from the same topic

# Reading data from Kafka

## Consumer “groups”

- Allow multi-threaded and/or multi-machine consumption from Kafka topics.
- Consumers “join” a group by using the same group.id
- **Kafka guarantees a message is only ever read by a single consumer in a group => processing order guarantee for messages within a partition.**
  - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
  - Maximum parallelism of a consumer group:  
**#consumers (in the group) =< #partitions**

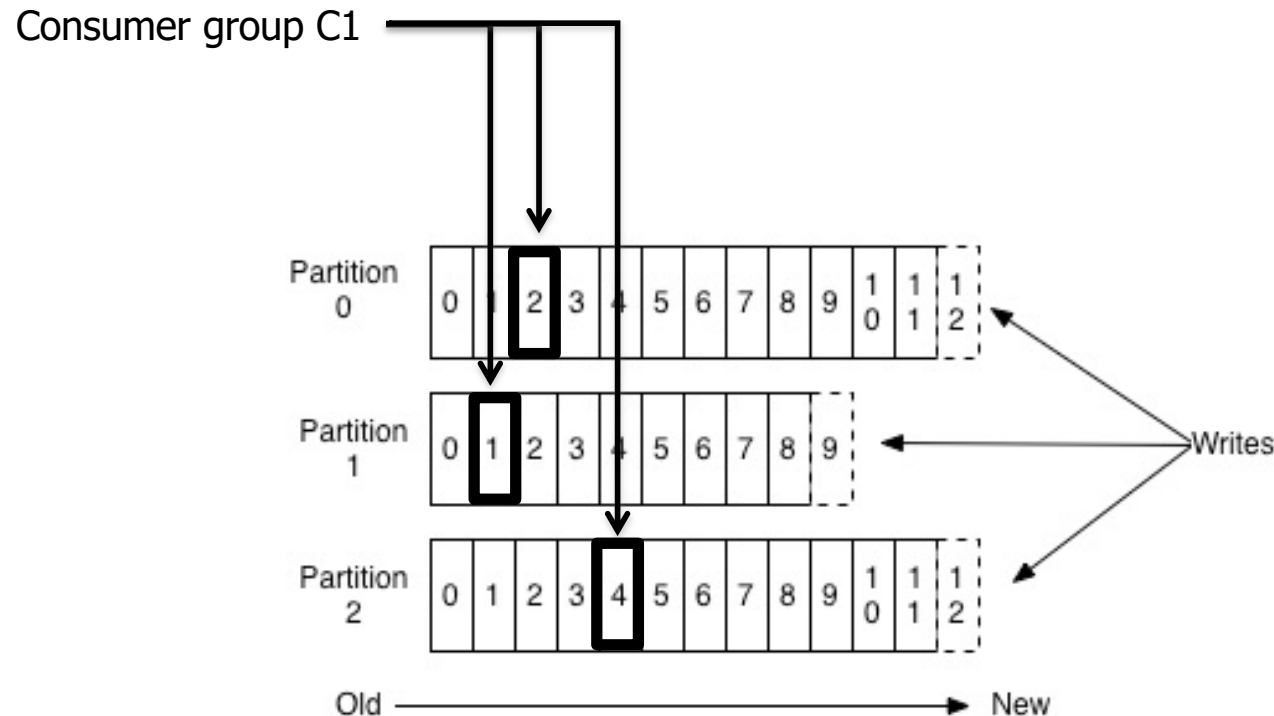


# Consumers

- Kafka offers a single consumer abstraction that generalizes both queuing and publish-subscribe mode.
- Consumers label themselves with a consumer group name, and each message published to a topic is delivered to **one consumer instance** within each subscribing consumer group.
- Kafka is able to provide both **ordering guarantees** and **load balancing** over a pool of consumer processes, by guarantee *each partition is consumed by exactly one consumer in the group*.
  - => There cannot be more consumer instances in a consumer group than partitions.
- Kafka only provides a total order over messages **within** a partition, not between different partitions in a topic.

# Partition offsets

- **Offset:** messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via (*offset, partition, topic*) tuples ; This offset is controlled by the Consumer



## Guarantees supported by Kafka

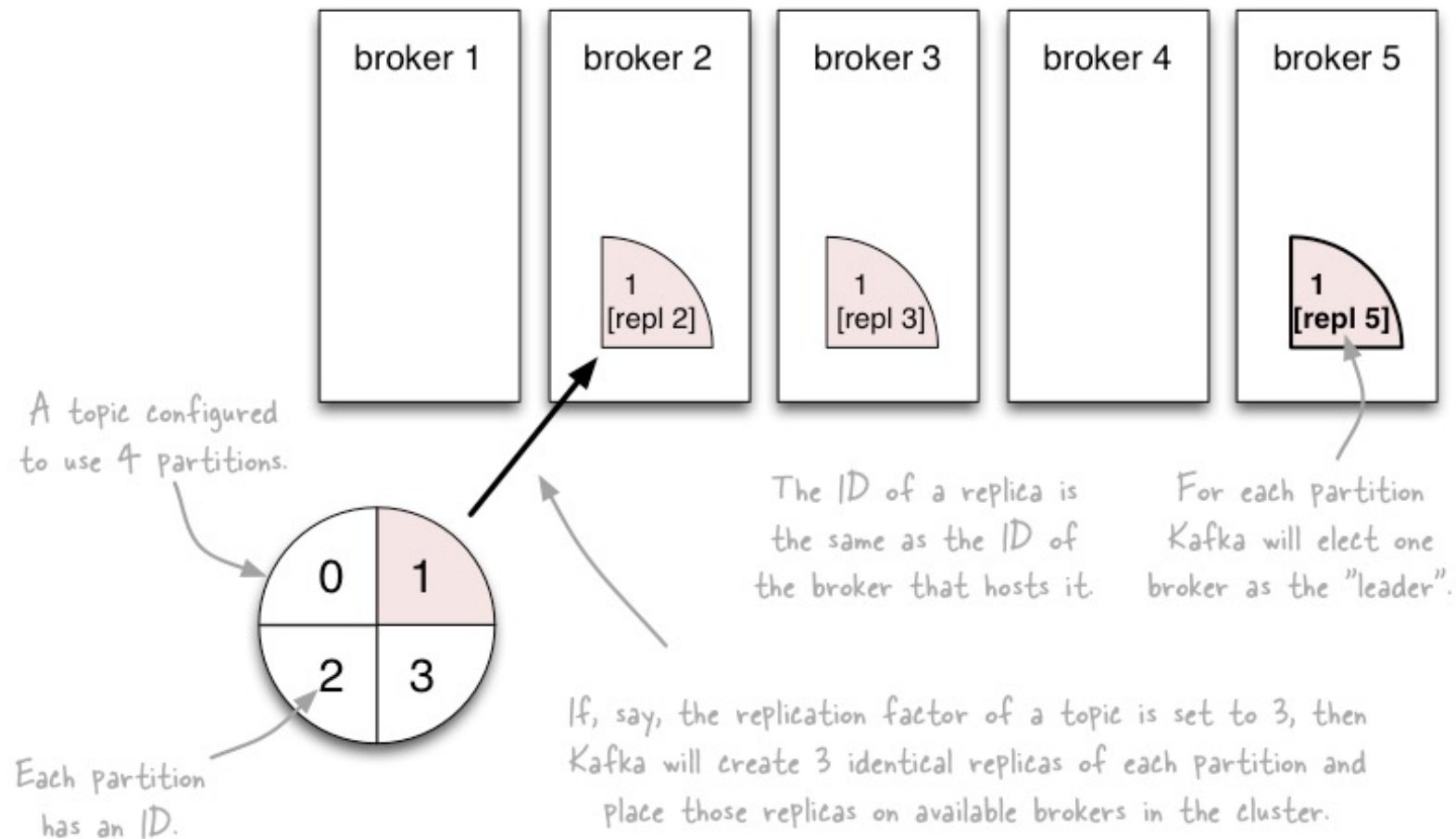
- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees messages in the order they are stored in the log.

# Distribution and Replication of a Partition

- The partitions of the log are distributed over the servers (brokers) in the Kafka cluster
- Each partition is replicated across a **configurable number** of servers for fault tolerance.
- Each partition has one server which acts as the "**leader**" and zero or more servers which act as "**followers**".
- If the leader fails, one of the followers will automatically become the new leader.
- Each server acts as a leader for some of its partitions and a follower for others so load is well **balanced within the cluster**



# Topics vs. Partitions vs. Replicas



<http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/>

# Distribution and Replication of a Partition (cont'd)

- The leader of a partition handles all read and write requests for the partition while the followers (Replicas) ONLY passively replicate the leader.
  - Replicas exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do NOT help to increase producer or consumer parallelism!
  - Kafka tolerates  $(numReplicas - 1)$  dead brokers before losing data
    - LinkedIn:  $numReplicas == 2 \rightarrow 1$  broker can die

# Inspecting the current state of a topic

- `--describe` the topic

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --describe --topic zerg.hydra
Topic:zerg2.hydra PartitionCount:3 ReplicationFactor:2 Configs:
  Topic: zerg2.hydra Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0
  Topic: zerg2.hydra Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1
  Topic: zerg2.hydra Partition: 2 Leader: 1 Replicas: 1,0 Isr: 1,0
```

- Leader: brokerID of the currently elected leader broker
  - Replica ID's = broker ID's
- ISR = “in-sync replica”, replicas that are in sync with the leader
- In this example:
  - Broker 0 is leader for partition 1.
  - Broker 1 is leader for partitions 0 and 2.
  - All replicas are in-sync with their respective leader partitions.

# Message Delivery Semantics

- **At Least Once** (default)
  - Messages are never lost but may be redelivered
- **At Most Once**
  - Messages can be lost but never redelivered
- **Exactly Once**

# Achieving Exactly Once Semantics

- Must consider 2 components
  - Durability guarantees when publishing a message (by the Producer)
  - Durability guarantees when reading a message (by a Consumer)
- For the Producer
  - What happens when a produce request was sent but a network error returned before an ACK ?
  - Use a Single writer per partition and check the latest committed value after network errors
- For a Consumer
  - Include a Unique ID (e.g. UUID) and de-duplicate records
  - Consider storing offsets with data

# Kafka Record Retention

- Kafka cluster retains all published records (as long as there is enough storage)
  - Time-based – Configurable Retention period, e.g. 3 days, 2 weeks or 1 month
  - Size-based – Configurable based on size
  - Compaction – keeps latest records
- Records written to Kafka are persisted to disk and replicated to other servers for fault-tolerance
- Records are available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size as Kafka always write to the end of the (topic) log
- Record (message) Producers *can* wait on Acknowledgement
  - s.t. Write not complete until properly replicated

# Writing Data to Kafka

For more detail tutorials, see:

<http://cloudurable.com/blog/kafka-tutorial-kafka-producer/index.html>

# Writing data to Kafka

- Use Kafka “Producers” to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
- Two modes of writing: “async” and “sync”
  - Different semantics
  - Sync Producer “send” call will block !
  - Async Producer is preferred to achieve high throughput (non-blocking)
- Important Configuration settings for Producers:

<code>client.id</code>	identifies producer app, e.g. in system logs
<code>producer.type</code>	async or sync
<code>acks</code>	acking semantics
<code>serializer.class</code>	configure encoder, e.g. using Avro
<code>Bootstrap.servers</code>	For bootstrapping from a list of well-known brokers



# Async Producer

- Sends messages in background = no blocking in client.
- Provides more powerful batching of messages.
- Wraps a sync producer, or rather a pool of them.
- Communication from async->sync producer happens via a queue.

## Caveats

- Async producer may drop messages if its queue is full.
  - Solution 1: Don't push data to producer faster than it is able to send to brokers.
  - Solution 2: Queue full == need more brokers, add them now! Use this solution in favor of solution 3 particularly if your producer cannot block (async producers).
  - Solution 3: Set `queue.enqueue.timeout.ms` to -1 (default). Now the producer will block indefinitely and will never willingly drop a message.
  - Solution 4: Increase `queue.buffering.max.messages` (default: 10,000).

# Message ACKing for Producers

- Background:
  - In Kafka, a message is considered *committed* when “any required” ISR (in-sync replicas) for that partition have applied it to their data log.
  - Message acking is about conveying this “Yes, committed!” information back from the brokers to the producer client.
  - Exact meaning of “any required” is defined by `request.required.acks`.
- Only **producers** must configure acking
  - Exact behavior is configured via `request.required.acks`, which determines when a produce request is considered completed.
  - Allows you to trade **latency (speed)** <-> **durability (data safety)**.
  - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don't need to worry about potentially seeing a message that could be lost if the leader fails.

# Message ACKing (cont'd)

- Typical values of `request.required.acks`
  - **0**: producer never waits for an ack from the broker.
    - Gives **the lowest latency** but the weakest durability guarantees.
  - **1**: producer gets an ack after the leader replica has received the data.
    - Gives better durability as we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.
  - **-1**: producer gets an ack after *all* ISR have received the data.
    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one ISR remains.

better  
latency  
↑  
↓  
durability  
better

- Beware of interplay with `request.timeout.ms`!
  - "The amount of time the broker will wait trying to meet the `request.required.acks` requirement before sending back an error to the client."
  - Caveat: Message may be committed even when broker sends timeout error to client (e.g. because not all ISR ack'ed in time). One reason for this is that the producer acknowledgement is independent of the leader-follower replication, and ISR's send their acks to the leader, the latter of which will reply to the client.

# Sample Java code for a Kafka Producer

```
package de.predic8.h_performance;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import javax.json.Json;
import javax.json.JsonObject;
import java.util.Properties;
import static java.lang.Math.random;
import static java.lang.Math.round;
import static org.apache.kafka.clients.producer.ProducerConfig.*;

public class PerformanceProducer {

    public static void main(String[] args) throws InterruptedException {

        Properties props = new Properties();
        props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ACKS_CONFIG, "all");
        props.put(RETRIES_CONFIG, 0);
        props.put(BATCH_SIZE_CONFIG, 32000);
        props.put(LINGER_MS_CONFIG, 100);
        props.put(BUFFER_MEMORY_CONFIG, 33554432);
        props.put(KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.LongSerializer");
        props.put(VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.LongSerializer");

        Producer<Long, Long> producer = new KafkaProducer<>(props);

        long t1 = System.currentTimeMillis();

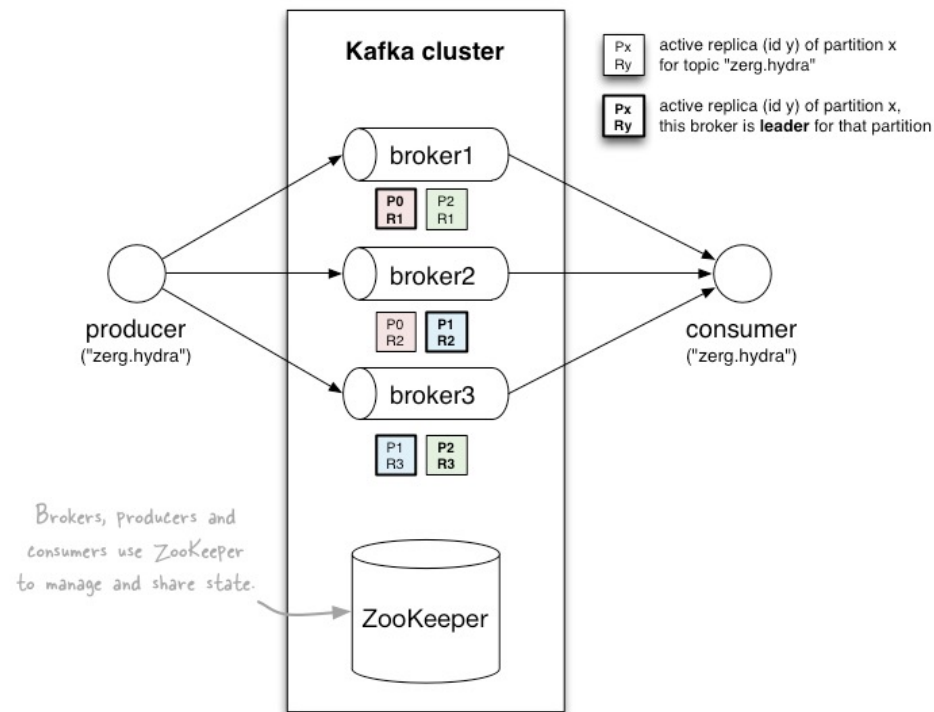
        long i = 0;
        for(; i < 1000000; i++) {

            producer.send(new ProducerRecord<>("produktion", i, i));
        }
        producer.send(new ProducerRecord<Long,Long>("production", (long) -1, (long)-1));
        System.out.println("fertig " + i + " Nachrichten in " + (System.currentTimeMillis() - t1 + " ms"));

        producer.close();
    }
}
```

# Write operations behind the scenes

- When writing to a topic in Kafka, producers write directly to the partition leaders (brokers) of that topic
  - Remember: Writes always go to the leader ISR of a partition!



- This raises two questions:
  - How to know the "right" partition for a given topic?
  - How to know the current leader broker/replica of a partition?

# 1) How to know the “right” partition when sending?

- In Kafka, a producer – i.e. the **client** – decides to which target partition a message will be sent.
  - Can be random ~ load balancing across receiving brokers.
  - Can be semantic based on message “key”, e.g. by user ID or domain name.
    - Here, Kafka guarantees that all data for the same key will go to the same partition, so consumers can make locality assumptions.

```
1 // Java example. Topic is "zerg.hydra".
2 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
3 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- But there’s one catch with line 2 (i.e. no key)

# Keyed vs. non-keyed messages in Kafka

- If a key **is not** specified:

```
2 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
```

- Producer will *ignore* any configured partitioner.
- It will pick a random partition from the list of *available* partitions **and stick to it** for some time before switching to another one = NOT round robin or similar!
  - Why? To reduce number of open sockets in large Kafka deployments ([KAFKA-1017](#)).
  - Default: 10mins, cf. `topic.metadata.refresh.interval.ms`
  - See implementation in `DefaultEventHandler#getPartition()`
- If there are fewer producers than partitions at a given point of time, some partitions may not receive any data. How to fix if needed?
  - Try to reduce the metadata refresh interval `topic.metadata.refresh.interval.ms`
  - Specify a message key and a customized random partitioner.
- In practice it is not trivial to implement a correct “random” partitioner in Kafka 0.8.
  - Partitioner interface in Kafka 0.8 lacks sufficient information to let a partitioner select a random and *available* partition. Same issue with `DefaultPartitioner`.

# Keyed vs. non-keyed messages in Kafka

- If a key is specified:

```
3 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- Key is retained as part of the msg, will be stored in the broker.
- One can design a partition function to route the msg based on key.
- The *default partitioner* assigns messages to a partition based on their key hashes, via `key.hashCode % numPartitions`.
- Caveat:
  - If you specify a key for a message but do not explicitly wire in a custom partitioner via `partitioner.class`, your producer will use the default partitioner.
  - So without a custom partitioner, messages with the same key will still end up in the same partition! (cf. default partitioner's behavior above)



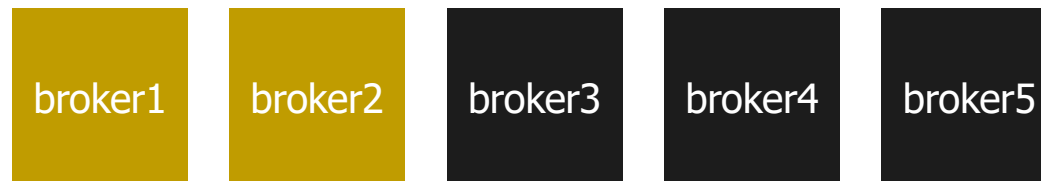
## 2) How to know the current leader of a partition?

- Producers: broker discovery aka bootstrapping
  - Producers don't talk to ZooKeeper, so it's not through ZK.
  - Broker discovery is achieved by providing producers with a “bootstrapping” broker list, cf. `metadata.broker.list`
    - These brokers inform the producer about all alive brokers and where to find current partition leaders. The bootstrap brokers do use ZK for that.
- Impacts on failure handling
  - In Kafka 0.8 the bootstrap list is static/immutable during producer run-time. This has limitations and problems as shown in next slide.
  - The bootstrap approach has been improved in Kafka 0.9. This change makes the life of Ops easier.

# Bootstrapping in Kafka

- Scenario: N=5 brokers total, 2 of which are for bootstrap

```
1 Properties props = new Properties();
2 props.put("metadata.broker.list", "broker1:9092,broker2:9092");
3 ProducerConfig config = new ProducerConfig(props);
```



- Do's:
  - Take down one bootstrap broker (e.g. **broker2**), repair it, and bring it back.
  - In terms of impacts on *broker discovery*, you can do whatever you want to brokers 3-5.
- Don'ts:
  - Stop all bootstrap **brokers 1+2**. If you do, the producer stops working!
- To improve operational flexibility, use VIP's or similar for values in `metadata.broker.list`.

# Reading Data from Kafka

For more detail tutorials, see:

<http://cloudurable.com/blog/kafka-tutorial-kafka-consumer/index.html>

# Reading data from Kafka

- You use Kafka “consumers” to read data from Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
  - The Kafka project only provides the JVM implementation.
    - Has risk that a new Kafka release will break non-JVM clients.
- Three API options for JVM users:
  1. High-level consumer API <<< in most cases you want to use this one!
  2. Simple consumer API
  3. Hadoop consumer API
- Most noteworthy: The “simple” API is anything but simple. 😊
  - Prefer to use the high-level consumer API if it meets your needs (it should).
  - Counter-example: Kafka spout in Storm 0.9.2 uses simple consumer API to integrate well with Storm’s model of guaranteed message processing.

# Reading data from Kafka

- Consumers *pull* from Kafka (there's no push)
  - Allows consumers to control their pace of consumption.
  - Allows to design downstream apps for **average** load, not peak load ([cf. Loggly talk](#))
- Consumers' responsibility to track their read positions i.e. offsets
  - High-level consumer API: takes care of this for you, stores offsets in ZooKeeper
  - Simple consumer API: nothing provided, it's totally up to you (the programmer)
  - What does this offset management allow you to do?
    - Consumers can deliberately rewind "in time" (up to the point where Kafka prunes), e.g. to replay older messages.
      - Cf. Kafka spout in Storm 0.9.2.
    - Consumers can decide to only read a specific subset of partitions for a given topic.
      - Cf. Loggly's setup of (down)sampling a production Kafka topic to a manageable volume for testing
    - Run offline, batch ingestion tools that write (say) from Kafka to Hadoop HDFS every hour.
      - Cf. LinkedIn Camus, Pinterest Secor

# Reading data from Kafka

- Important consumer configuration settings

<code>group.id</code>	assigns an individual consumer to a "group"
<code>zookeeper.connect</code>	to discover brokers/topics/etc., and to store consumer state (e.g. when using the high-level consumer API)
<code>fetch.message.max.bytes</code>	number of message bytes to (attempt to) fetch for each partition; must be $\geq$ broker's <code>message.max.bytes</code>

# Sample Java code for a Kafka Consumer

```
package com.cloudurable.kafka;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
        "localhost:9092,localhost:9093,localhost:9094";
    ...
}

public class KafkaConsumerExample {

    private static Consumer<Long, String> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
            "KafkaExampleConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            LongDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        // Create the consumer using props.
        final Consumer<Long, String> consumer =
            new KafkaConsumer<>(props);

        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(TOPIC));
        return consumer;
    }
}
```

```
...
static void runConsumer() throws InterruptedException {
    final Consumer<Long, String> consumer = createConsumer();

    final int giveUp = 100; int noRecordsCount = 0;

    while (true) {
        final ConsumerRecords<Long, String> consumerRecords =
            consumer.poll(1000);

        if (consumerRecords.count()==0) {
            noRecordsCount++;
            if (noRecordsCount > giveUp) break;
            else continue;
        }

        consumerRecords.forEach(record -> {
            System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
                record.key(), record.value(),
                record.partition(), record.offset());
        });

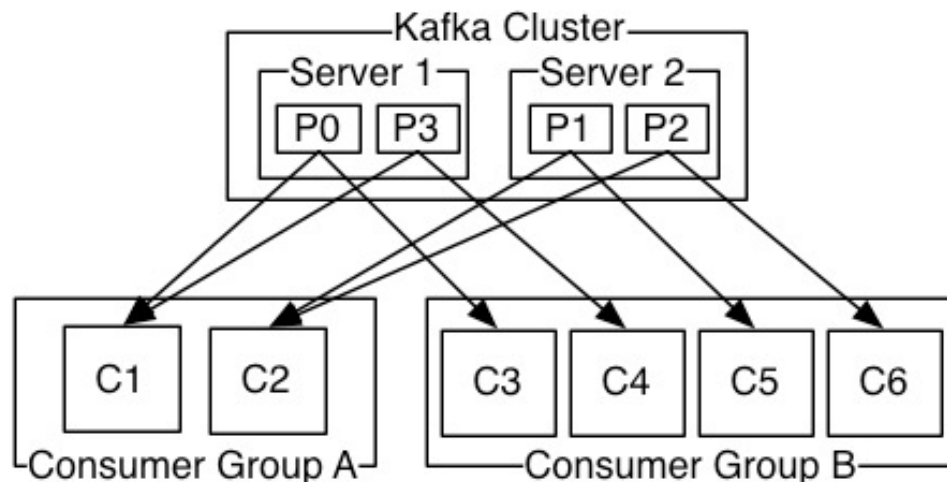
        consumer.commitAsync();
    }
    consumer.close();
    System.out.println("DONE");
}

public static void main(String... args) throws Exception {
    runConsumer();
}
}
```

# Recall: Reading data from Kafka

## Consumer “groups”

- Allow multi-threaded and/or multi-machine consumption from Kafka topics.
- Consumers “join” a group by using the same group.id
- **Kafka guarantees a message is only ever read by a single consumer in a group.**
  - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
  - Maximum parallelism of a consumer group: **#consumers** (in the group)  $\leq$  **#partitions**



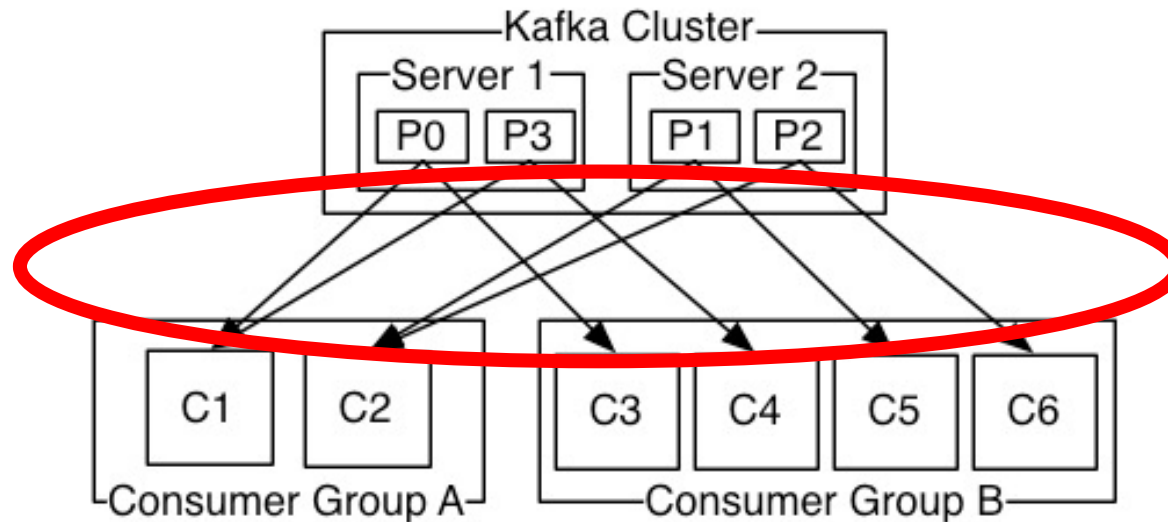


# Guarantees when reading data from Kafka

- A message is only ever read by a single consumer in a group.
- A consumer sees messages in the order they were stored in the log.
- The order of messages is only guaranteed within a partition.
  - No order guarantee across partitions, which includes no order guarantee per-topic.
  - If total order (per topic) is required you can consider, for instance:
    - Use `#partition = 1`. Good: total order. Bad: Only 1 consumer process at a time.
    - “Add” total ordering in your consumer application, e.g. a Storm topology.
- Some gotchas:
  - If you have multiple partitions per thread there is NO guarantee about the order you receive messages, other than that within the partition the offsets will be sequential.
    - Example: You may receive 5 messages from partition 10 and 6 from partition 11, then 5 more from partition 10 followed by 5 more from partition 10, even if partition 11 has data available.
  - Adding more processes/threads will cause Kafka to rebalance, possibly changing the assignment of a partition to a thread (whoops).

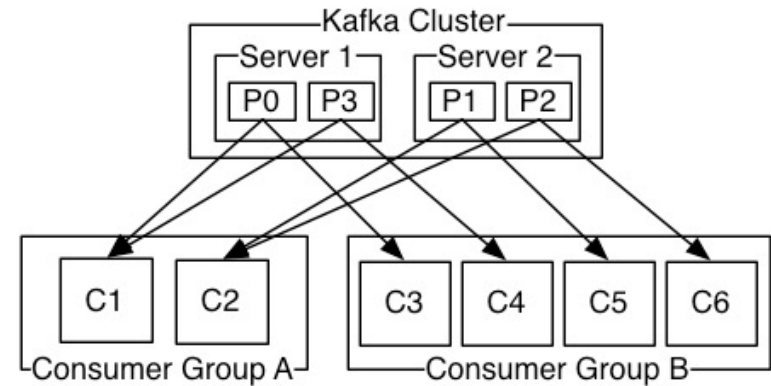
# Rebalancing: how consumers meet brokers

Remember?



- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

# Rebalancing: how consumers meet brokers



- Why “dynamic at run-time”?
  - Machines can die, be added, ...
  - Consumer apps may die, be re-configured, added, ...
- Whenever this happens a **rebalancing** occurs.
  - Rebalancing is a normal and expected lifecycle event in Kafka.
  - But it’s also a nice way to shoot yourself or Ops in the foot.
- Why is this important?
  - Most Ops issues are due to
    - 1) **rebalancing** and 2) **consumer lag**.
  - So Dev + Ops must understand what goes on.

# Rebalancing: how consumers meet brokers

- Rebalancing?
  - Consumers in a group come into consensus on which consumer is consuming which partitions → required for distributed consumption
  - Divides broker partitions evenly across consumers, tries to reduce the number of broker nodes each consumer has to connect to
- When does it happen? Each time:
  - a **consumer** joins or leaves a consumer group, OR
  - a **broker** joins or leaves, OR
  - a topic “joins/leaves” via a filter, cf. `createMessageStreamsByFilter()`
- Examples:
  - If a consumer or broker fails to heartbeat to ZK → rebalance!
  - `createMessageStreams()` registers consumers for a topic, which results in a rebalance of the consumer-broker assignment.

# Kafka Polyglot Clients/ Wire Protocol

- Kafka communication from clients and servers wire protocol over TCP
- Protocol versioned
- Maintains backwards compatibility
- Many Languages supported
- Kafka REST proxy allows easy integration with other systems via HTTP and JSON
- Also provide Avro/ Schema registry support via an extended Kafka ecosystem

# Serialization in Kafka

- Kafka does not care about data format of msg payload
- Up to developer (= you) to handle serialization/deserialization
  - Common choices in practice: Avro, JSON

```
1  /**
2   * Create a list of message streams of type T for each topic.
3   *
4   * @param topicCountMap a map of (topic, #streams) pair
5   * @param decoder a decoder that converts from Message to T
6   * @return a map of (topic, list of KafkaStream) pairs.
7   *         The number of items in the list is #streams. Each stream supports
8   *         an iterator over message/metadata pairs.
9   */
10 public <K,V> Map<String, List<KafkaStream<K,V>>>
11     createMessageStreams(Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, Decoder<V> valueDecoder);
12
13 /**
14 * Create a list of message streams of type T for each topic, using the default decoder.
15 */
16 public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String, Integer> topicCountMap);
```

(Code above is from the High Level Consumer API)

# Serialization in Kafka: using Avro

- One way to use Avro in Kafka is via Twitter Bijection.
  - <https://github.com/twitter/bijection>
- Approach: Convert pojo to byte[], then send byte[] to Kafka.
  - Bijection in Scala:

```
5  val tweet = new Tweet("miguno", "Terran is the cheese race.", 1366190000)
6
7  // From POJO to byte array
8  val bytes = Injection[Tweet, Array[Byte]](tweet)
9
10 // From byte array back to POJO
11 val recovered_tweet = Injection.invert[Tweet, Array[Byte]](bytes)
```

- Bijection in Java: <https://github.com/twitter/bijection/wiki/Using-bijection-from-java>
- Full Kafka/Bijection example:
  - [KafkaSpec in kafka-storm-starter](#)
- Alternatives to Bijection:
  - e.g. <https://github.com/miguno/kafka-avro-codec>

# Data compression in Kafka

- Kafka provides E2E Batch Compression
- Bottleneck is not always CPU or disk but often network bandwidth
  - Especially in Cloud, Containerized and Virtualized environments
  - Especially when communicating between Datacenters or via WAN
- Instead of compressing messages one at a time, compresses whole batch and sent to Kafka Broker in one go !
  - Interplay with batching of messages, e.g. larger batches typically achieve better compression ratios.
  - Message batch gets written in compressed form in log partition and do NOT get decompressed until they are consumed.
- GZIP, Snappy and LZ4 compression algorithms supported
- Details about compression in Kafka:
  - <https://cwiki.apache.org/confluence/display/KAFKA/Compression>

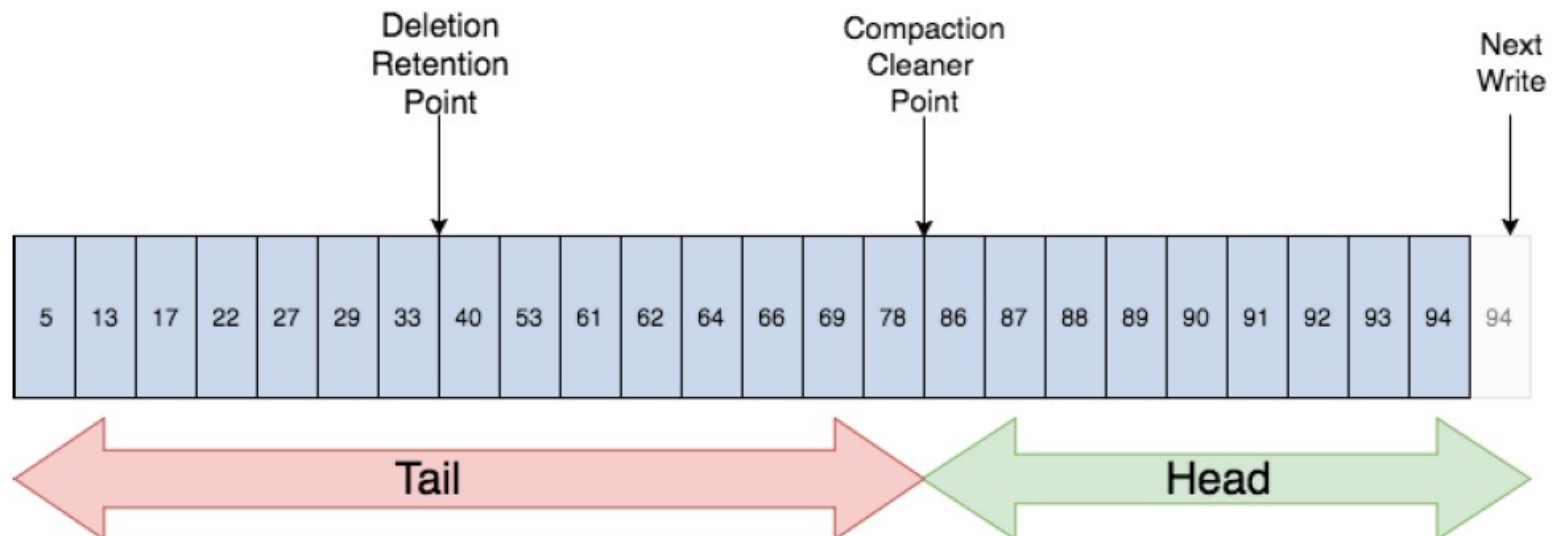


# Log Compaction Overview

- Recall Kafka can delete older records based on
  - time period
  - size of a log
- Kafka also supports log compaction for **Record Key** Compaction
- Log compaction: keep latest version of record and delete older versions

# Log Compaction Structure

- Log has head and tail
- Head of compacted log identical to a traditional Kafka log
- New records get appended to the head
- Log compaction works at tail of the log
- Tail gets compacted
- Records in tail of log retain their original offset when written after compaction



# Log Compaction Cleaning for Key-value Data

Before  
Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

## Cleaning

Only keeps latest version  
of key. Older duplicates not  
needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After  
Compaction

# Log Compaction Overview

- Log compaction retains last known value for each record key
- Useful for restoring state after a crash or system failure, e.g., in memory service, persistent data store, reloading a cache
- Data streams is to log changes to keyed, mutable data,
  - e.g., changes to a database table, changes to object in in-memory microservice
- Topic log has full snapshot of final values for every key - not just recently changed keys
- Downstream consumers can restore state from a log compacted topic

# Log Compaction Details

- All offsets remain valid, even if record at offset has been compacted away (next highest offset)
- Compaction also allows for deletes. A message with a key and a null payload acts like a tombstone (a delete marker for that key)
  - Tombstones get cleared after a period.
- Log compaction periodically runs in background by recopying log segments.
- Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers

# Log Compaction Guarantee

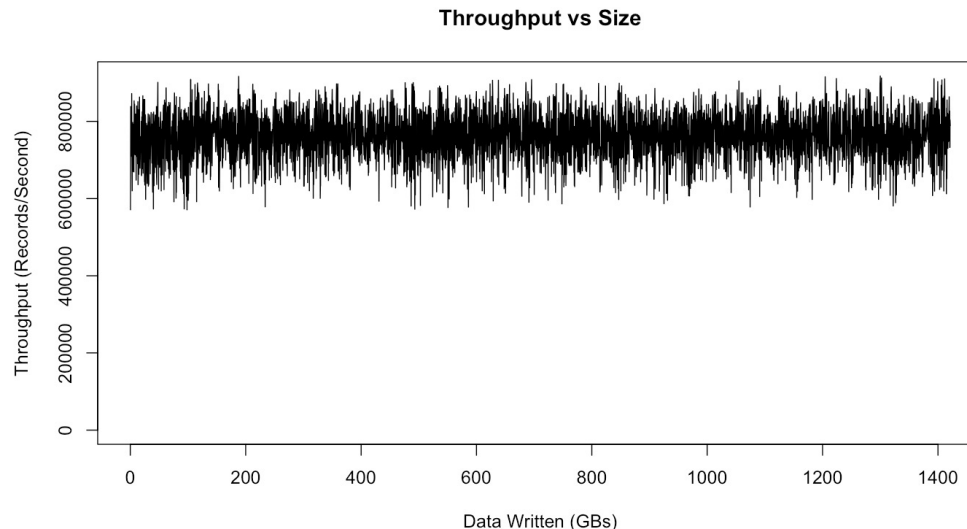
- If consumer stays caught up to head of the log, it sees every record that is written.
  - Topic config ***min.compaction.lag.ms*** used to guarantee minimum period that must pass before message can be compacted.
  - Consumer sees all tombstones as long as the consumer reaches head of log in a period less than the topic config ***delete.retention.ms*** (the default is 24 hours).
- Compaction will never re-order messages, just remove some.
- Offset for a message never changes.
- Any consumer reading from start of the log, sees at least final state of all records in order they were written

# Failover vs. Disaster Recovery

- Kafka Replication is for Failover
  - Replication of Kafka Topic Log partitions allows for failure of a rack or AWS availability zone
- Mirror Maker is used for Disaster Recovery
  - Mirror Maker replicates a Kafka cluster to another datacenter or AWS region
  - Called Mirroring since replication happens within a cluster.

# How fast is Kafka?

- **“Up to 2 million writes/sec on 3 cheap machines”**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated
- **Sustained throughput as stored data grows**
  - Slightly different test config than 2M writes/sec above.



- Test setup
  - Kafka trunk as of April 2013, but 0.8.1+ should be similar.
  - 3 machines: 6-core Intel Xeon 2.5 GHz, 32GB RAM, 6x 7200rpm SATA, 1GigE



# Why is Kafka so Fast ?

- Adopt the principle of Zero Copy
  - Using sendfile (Java's NIO FileChannel transferTo method)
  - Implement Linux sendfile( ) which skips unnecessary copies
- Batch Data in Chunks during Disk and Network access
  - The I/O scheduler will batch together consecutive small writes into bigger physical writes which improve system throughput
  - The I/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves system throughput
  - Provide more efficient data compression
- Sequential Disk Writes
  - Write to immutable commit log => Disk accessed in sequential manner
  - Eliminate slow disk seek or random I/O operations

# Why is Kafka so Fast ? (cont'd)

- Heavily relies on Linux PageCache
  - It automatically uses all free memory on the machine
  - In a system where consumers are roughly caught up with producers, you are essentially reading data from cache
- Scale out Horizontally
  - Use 100's to 1000's of partitions (i.e. sharding) for a single topic (group of messages of interest)
  - Spread workload to thousands of servers to handle massive load

# Why is Kafka so fast (cont'd) ?

- **Fast writes:**

- While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.
- Cf. hardware specs and OS tuning

- **Fast reads:**

- Very efficient to transfer data from page cache to a network **socket**
- Linux: **sendfile()** system call

- **Combination of the two = fast Kafka!**

- Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache.

<http://kafka.apache.org/documentation.html#persistence>

# Why is Kafka so fast?

- Example: Loggly.com, who run Kafka & Co. on Amazon AWS
  - “99.99999% of the time our data is coming from disk cache and RAM; only very rarely do we hit the disk.”
  - “One of our consumer groups (8 threads) which maps a log to a customer can process about 200,000 events per second draining from 192 partitions spread across 3 brokers.”
    - Brokers run on m2.xlarge Amazon EC2 instances backed by provisioned IOPS

<http://www.developer-tech.com/news/2014/jun/10/why-loggly-loves-apache-kafka-how-unbreakable-ininitely-scalable-messaging-makes-log-management-better/>

## Recap: Key Strength of Kafka

- High Performance – High Throughput, Low-Latency
- Stable, Robust Replication => Reliable, Durability
- Support Flexible Publish-Subscribe model for Data Producers and Consumers
- Producer Tunable Consistency Guarantees
- Ordering Preserved (at Shard Level, i.e. Topic Partition)
- Works well with Systems that have Data Streams to process, aggregate, transform and load to other stores
  - “Connector” modules available for most mainstream Big Data processing frameworks.

# Comparing Kafka with other Alternatives

# Kafka vs.

## Message Oriented Middleware (MOM) systems

- MOM = JMS, ActiveMQ, RabbitMQ, IBM MQ Series, Tibco, etc.
- Is Kafka a Queue or a Pub/Sub/Topic?
  - Yes
- Kafka is like a Queue per consumer group
  - Kafka is a queue system per consumer in consumer group so load balancing like JMS, RabbitMQ queue
- Kafka is like Topics in JMS, RabbitMQ, MOM
  - Topic/pub/sub by offering Consumer Groups which act like subscriptions
  - Broadcast to multiple consumer groups

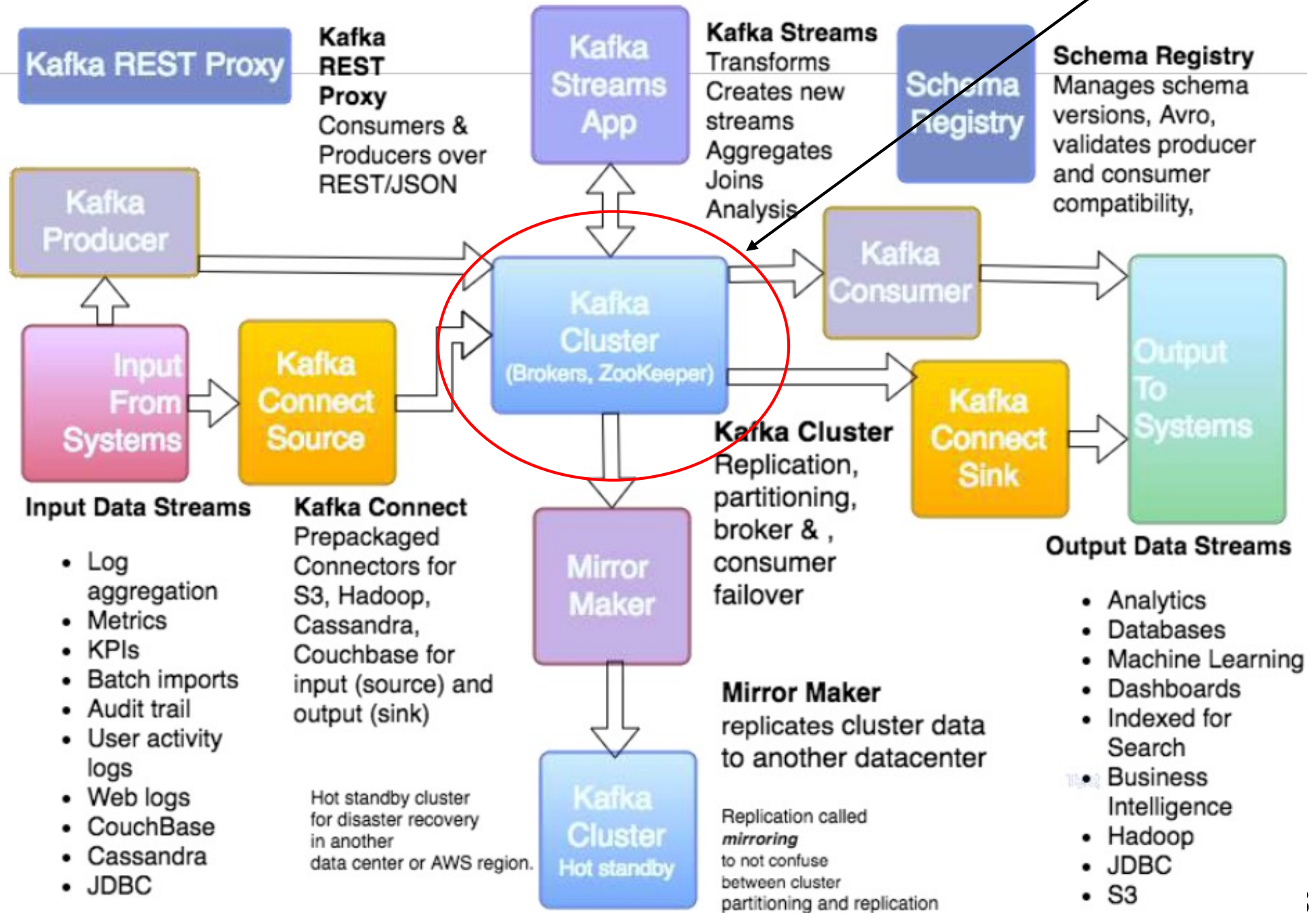
# Kafka vs.

## Message Oriented Middleware (MOM) systems

- By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log
  - Load divided among Consumers for read by partition
  - Handle parallel consumers better than traditional MOM
- Also by moving location (partition offset) in log to client/consumer side of equation instead of the broker, less tracking required by Broker and more flexible consumers
- Kafka written with mechanical sympathy, modern hardware, cloud in mind
  - Disks are faster
  - Servers have tons of system memory
  - Easier to spin up servers for scale out



# The Extended Kafka Universe (beyond “Kafka Core”)



# Kafka and Amazon Kinesis are similar

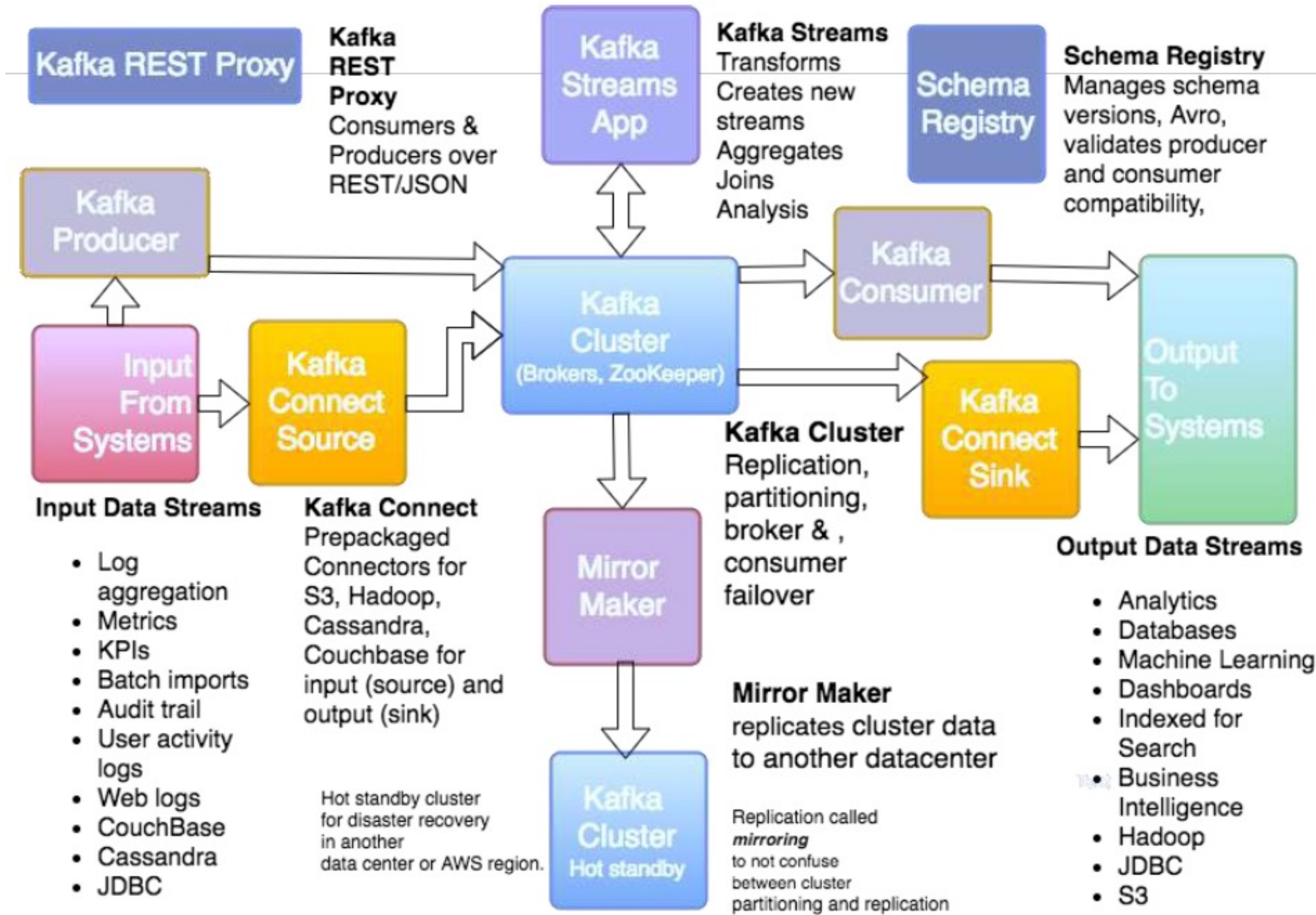
- Kinesis Streams is like Kafka Core
- Kinesis Analytics is like Kafka Streams
- Kinesis Shard is like Kafka Partition
- Similar and get used in similar use cases
- In Kinesis, data is stored in shards. In Kafka, data is stored in partitions
- Kinesis Analytics allows you to perform SQL like queries on data streams
- Kafka Streaming allows you to perform functional aggregations and mutations
- Kafka integrates well with Spark and Flink which allows SQL like queries on streams

# Kafka vs. Amazon Kinesis

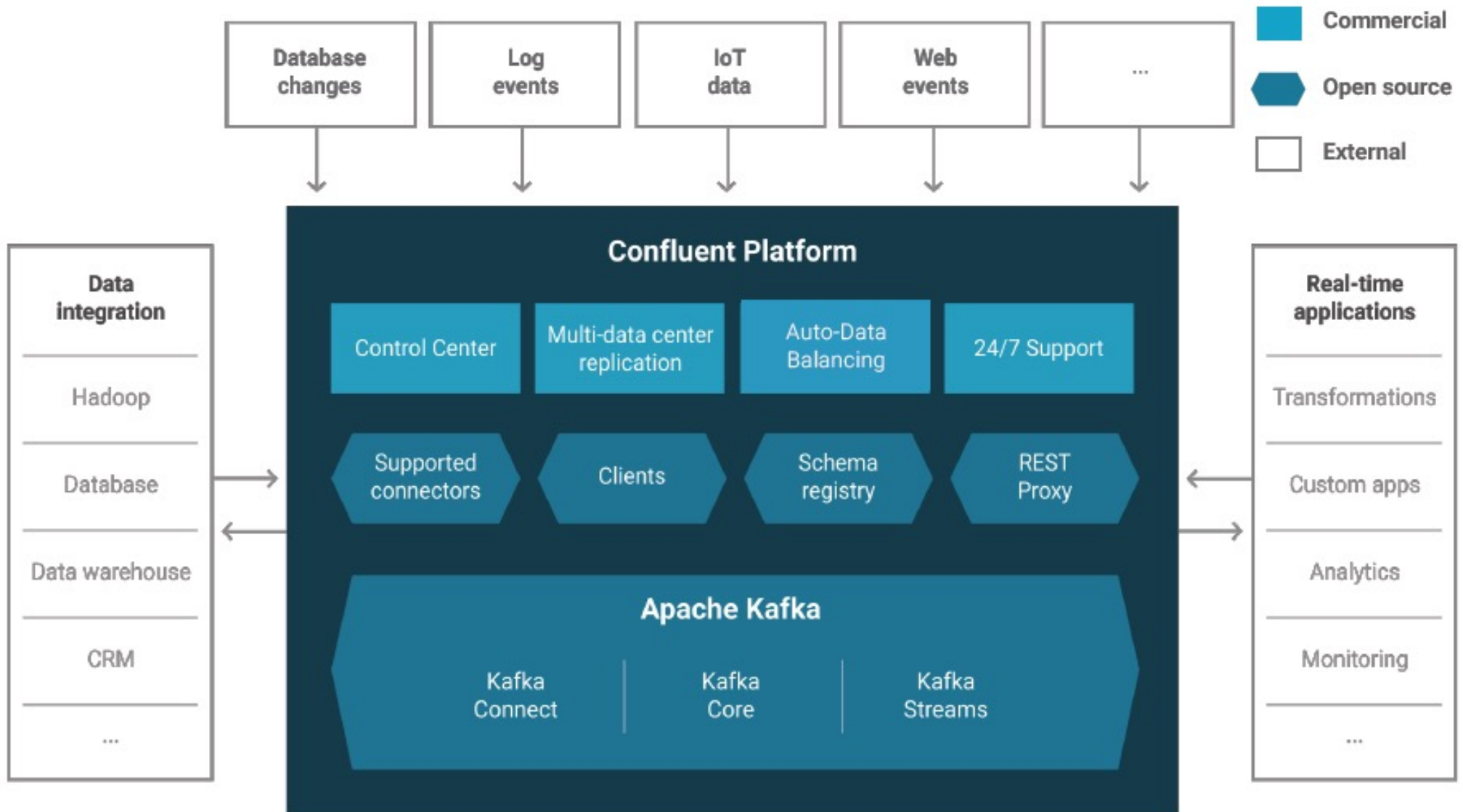
- Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
- Kafka records default stored for 7 days
  - can increase until you run out of disk space.
  - Decide by the size of data or by date.
  - Can use compaction with Kafka so it only stores the latest timestamp per key per record in the log
- With Kinesis data can be analyzed by Lambda before it gets sent to S3 or RedShift
- With Kinesis you pay for use, by buying read and write units.
- Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going
- Kinesis is sold as a service and does not require a DevOps team to keep it going (can be more expensive and less flexible, but much easier to setup and run)

# The Extended Universe of Kafka by Confluent

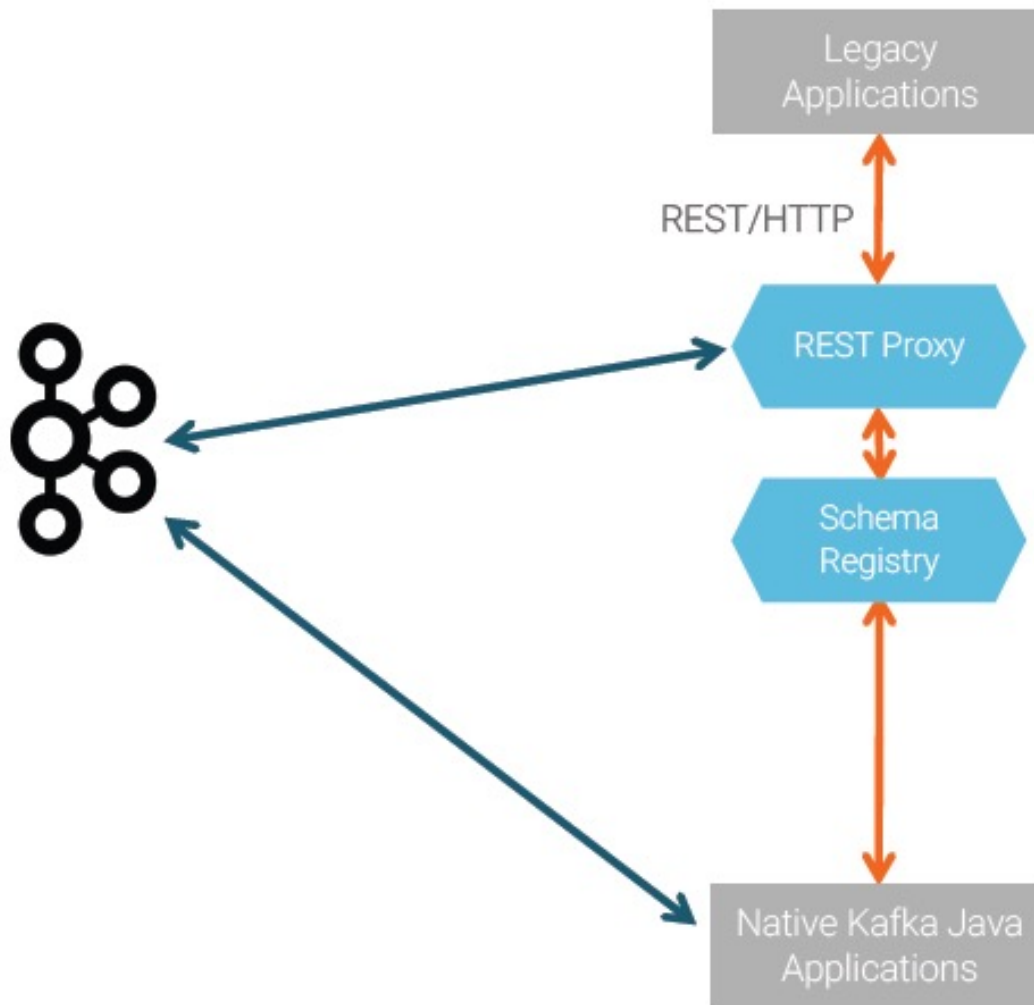
# The "Extended" Kafka Universe



# The Confluent Enterprise Streaming Platform

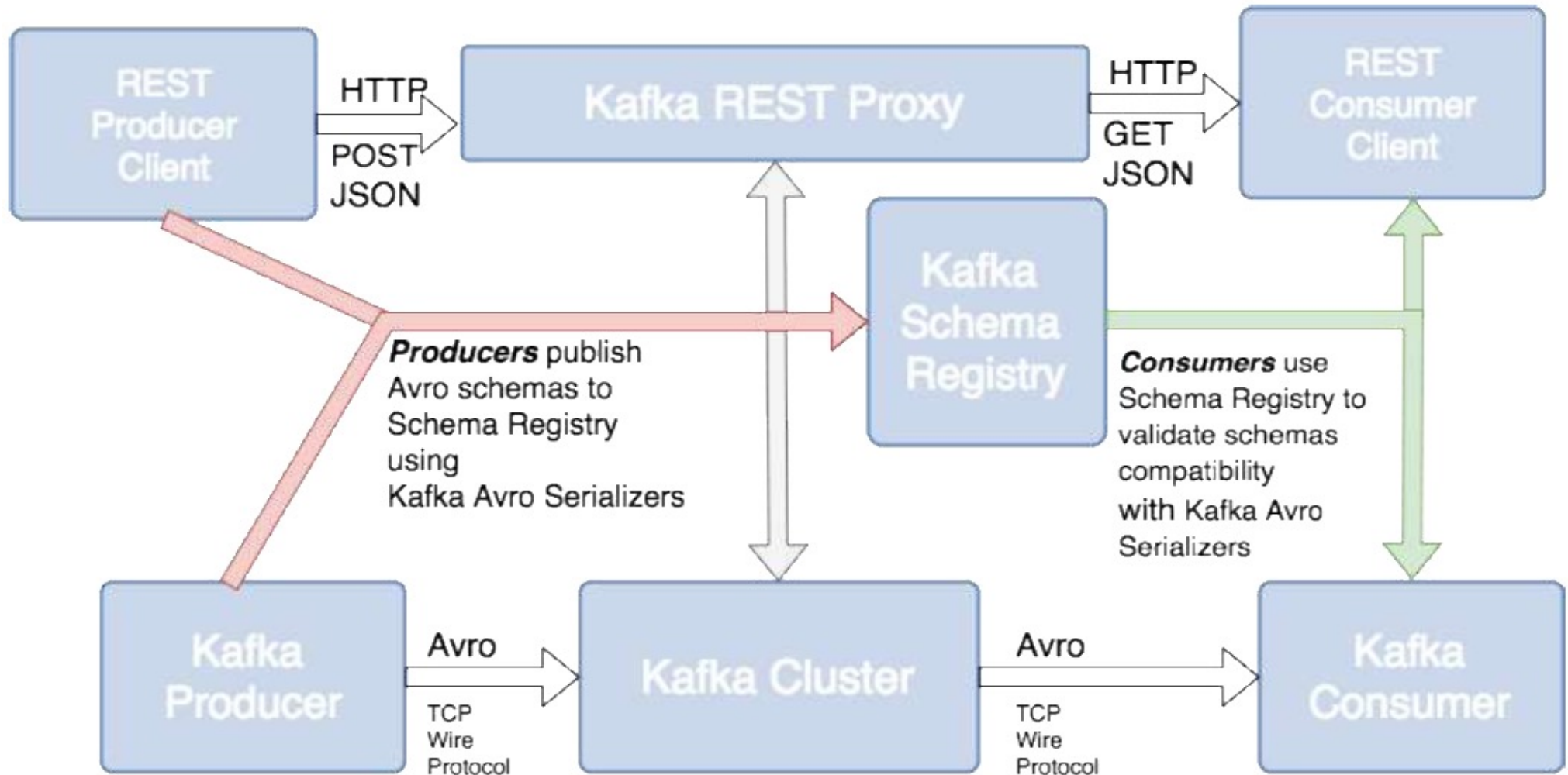


# REST Proxy: Enable other Applications to access Kafka data



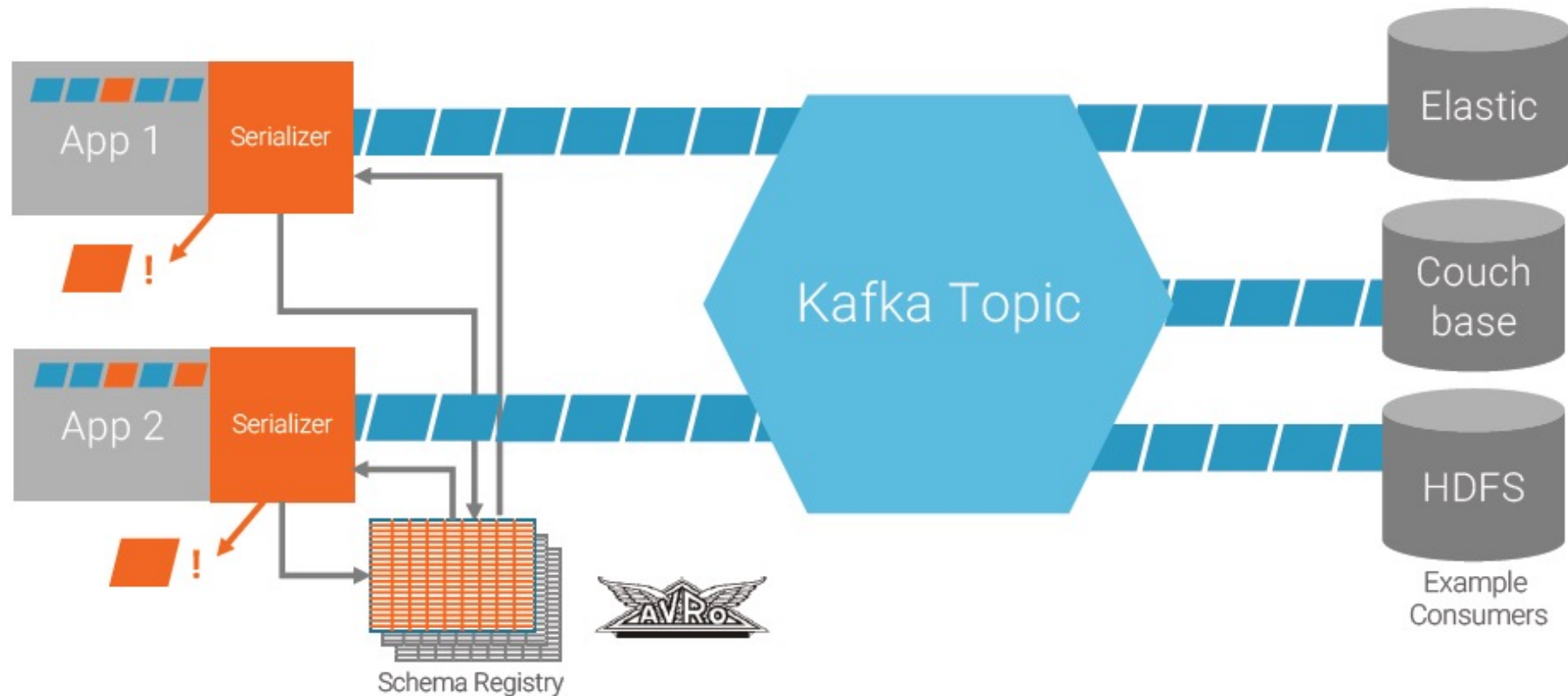
- Provide a RESTful Interface to a Kafka Cluster
  - Simplify message creation and consumption
  - Simplify system administration

# Kafka REST Proxy and Schema Registry





# Sample use of the Confluent Schema Registry



- Define the expected fields (schema) for each Kafka topic
- Leverage Avro to automatically handle schema changes (e.g. new fields) => enhance backward compatibility

# Kafka Clients



Apache Kafka Native Clients

---



Confluent Native Clients

---



Proxy http/REST

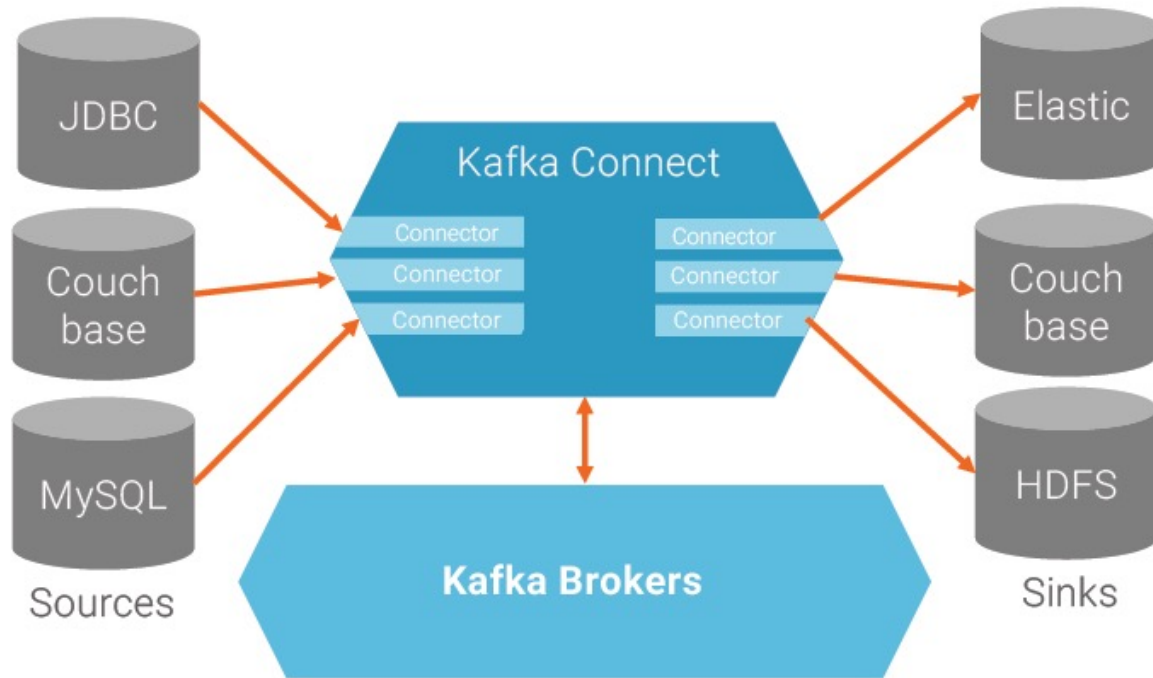


Stdin/stdout

Community Supported Clients

- Define the expected fields (schema) for each Kafka topic
- Leverage Avro to automatically handle schema changes (e.g. new fields) => enhance backward compatibility

# Apache Kafka Connect Library of Connectors for Streaming Data Capture



Databases
JDBC*
Couchbase
Datastax / Cassandra
GoldenGate
JustOne
DynamoDB
MongoDB
Hbase
InfluxDB
Kudu
RethinkDB

Datstore / File Store
HDFS*
Apache Ignite
FTP
Syslog
Hazelcast

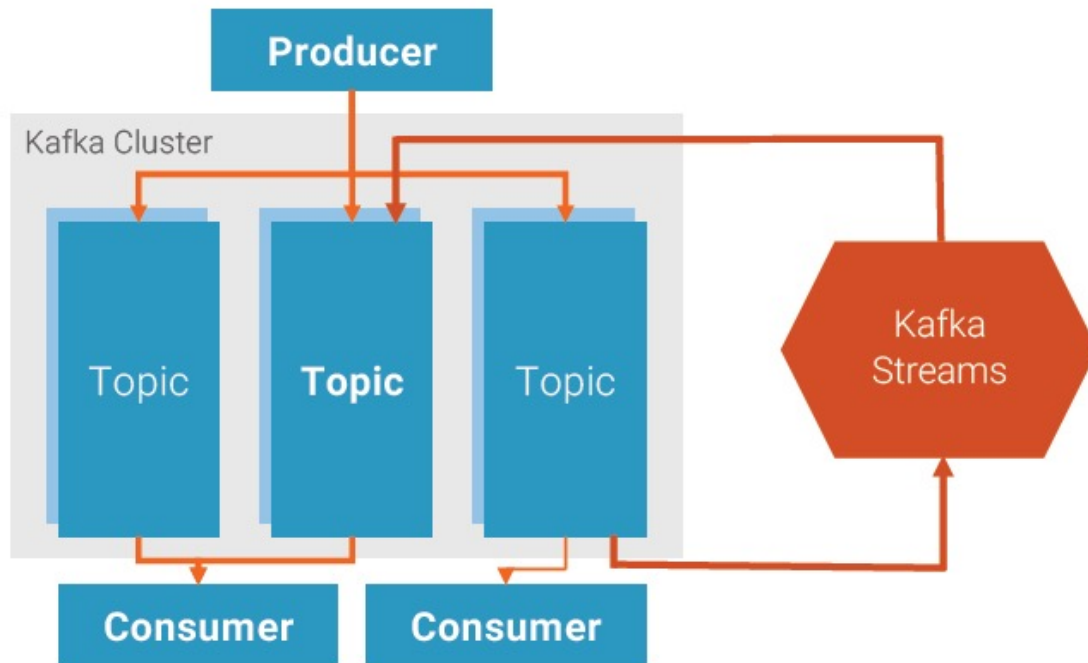
Analytics
Elasticsearch*
Veritca
Mixpanel

Applications / Other
Attunity
AWS / S3
Bloomberg Ticker
Striim
Solr
Syncsort
Twitter

- Fault tolerant
- Preserve Data Schema
- Manage/Support Heterogeneous Sources and Sinks

\* Denotes Connectors developed at Confluent and distributed with the Confluent Platform.

# Architecture of Kafka Streams



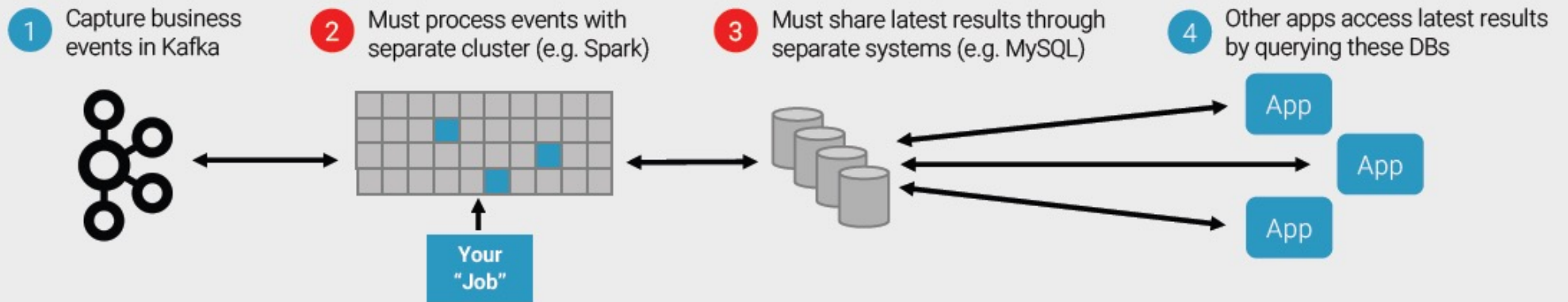
## Example Use Cases for “Kafka Streams”

- Microservices
- Continuous queries
- Continuous transformations
- Event-triggered processes

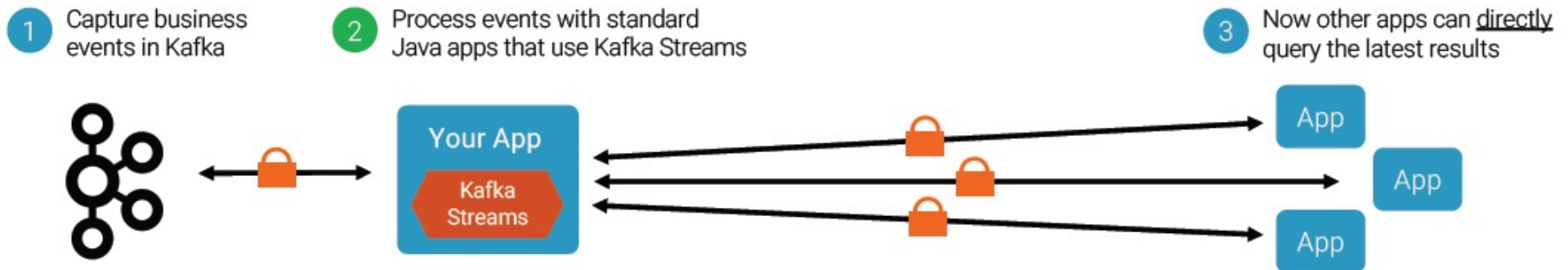
- Available as high-level DSL as well as low-level API to enable flexible application development
- No additional cluster required
- Security and permission integrated from Kafka

# Comparison of Alternative Service Architectures to support Enterprise-level Stream Processing

**Before:** Undue complexity, heavy footprint, many technologies, split ownership with conflicting priorities

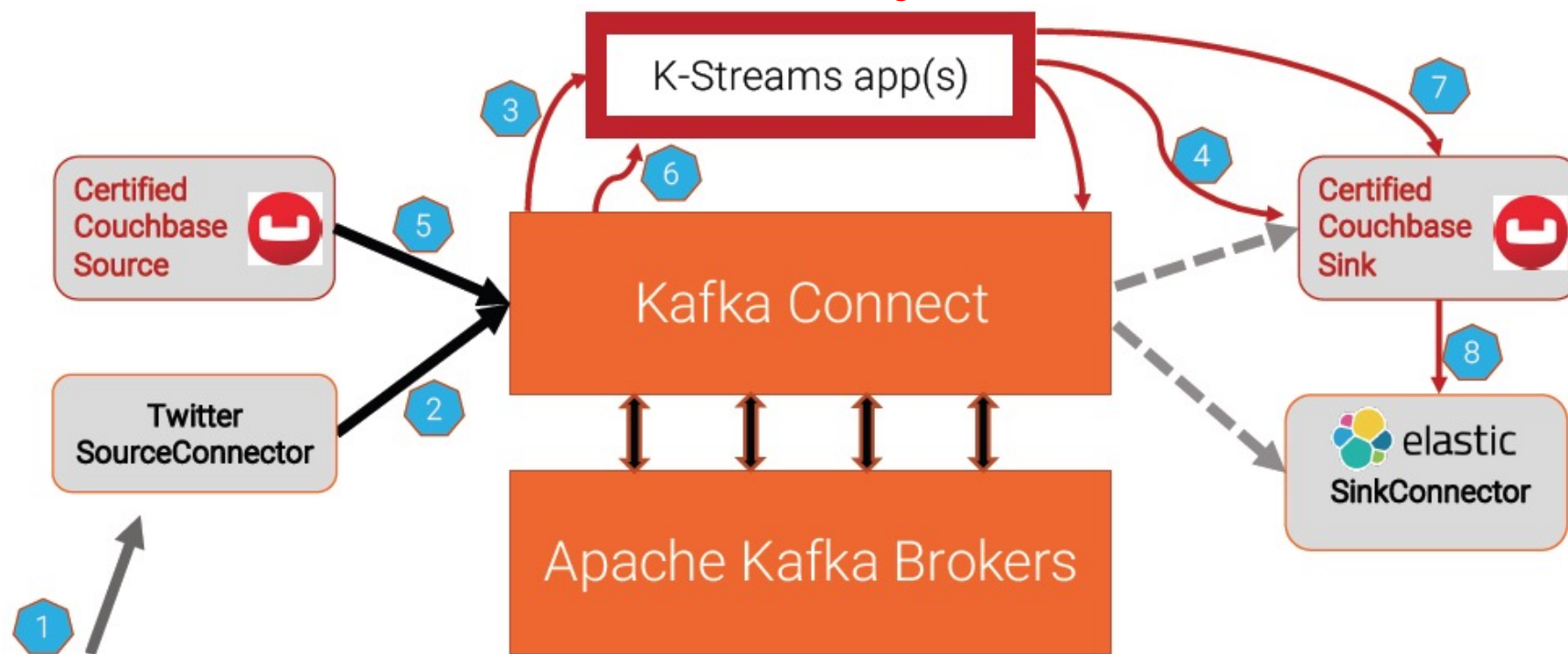


**With Kafka Streams:** simplified, app-centric architecture, puts app owners in control



“Grain of Salt” Alert: This slide is from Confluent

# A Sample Streaming Data Pipeline using the Kafka ecosystem



1. Twitter feed with sentiment data
2. Use Twitter Source connector to publish data to Kafka topic
3. Kafka Streams application augments Twitter records with sentiment analysis
4. Kafka Streams output to Couchbase
5. Couchbase Source Connector to pull data from Couchbase bucket back to Kafka topic
6. 2<sup>nd</sup> stage Kafka Streams app saves data to another Couchbase bucket and then onto Elastic Search

# Operating Kafka

# Kafka broker hardware specs @ LinkedIn (circa 2014)

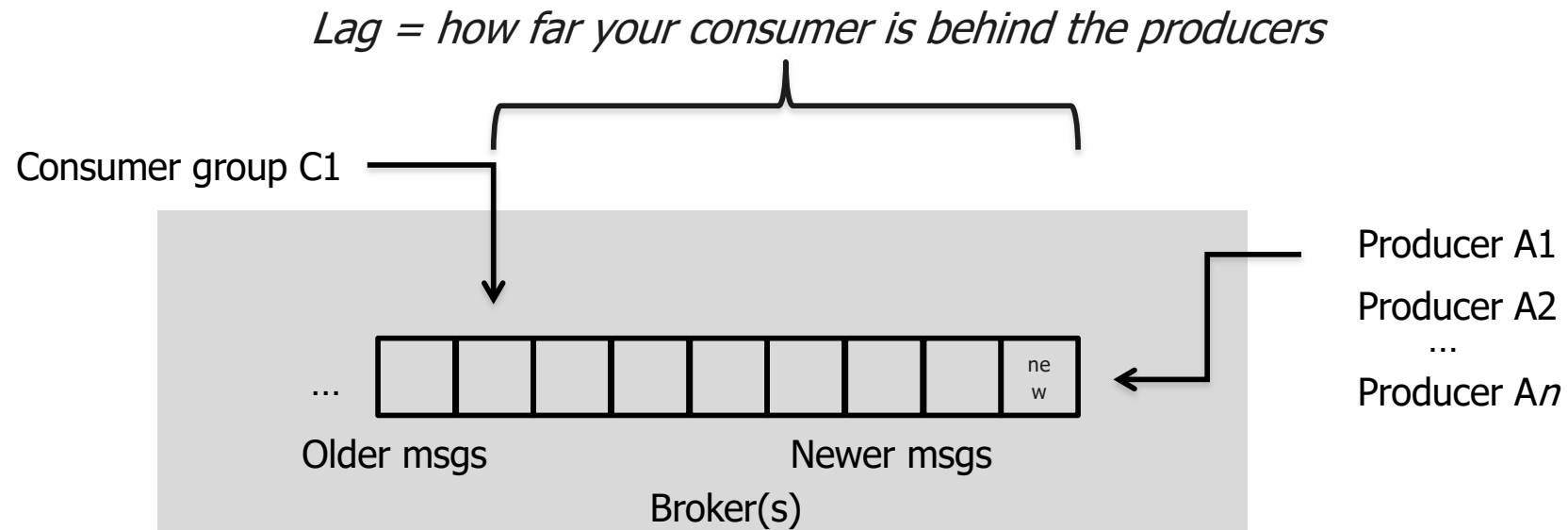
- Solely dedicated to running Kafka, run nothing else.
  - 1 Kafka broker instance per machine
- 2x 4-core Intel Xeon (info outdated?)
- **64 GB RAM (up from 24 GB)**
  - Only 4 GB used for Kafka broker, remaining 60 GB for page cache
  - Page cache is what makes Kafka **fast**
- **RAID10 with 14 spindles**
  - More spindles = higher disk throughput
  - Cache on RAID, with battery backup
  - Before H/W upgrade: 8x SATA drives (7200rpm), not sure about RAID
- 1 GigE (?) NICs
- EC2 example: m2.2xlarge @ \$0.34/hour, with provisioned IOPS



# Operating Kafka

- Typical operations tasks include:
  - Adding or removing brokers
    - Example: ensure a newly added broker actually receives data, which requires moving partitions from existing brokers to the new broker
    - Kafka provides helper scripts but still manual work involved
  - Balancing data/partitions to ensure best performance
  - Add new topics, re-configure topics
    - Example: Increasing #partitions of a topic to increase max parallelism
  - Apps management: new producers, new consumers
- Biggest Challenges to handle growth of Kafka adoption, increase in Producers, Consumers
  - Most problems are due to: 1) Consumer Lag ; and 2) Rebalancing
- Original design was not created with security in mind.
  - Discussion started in June 2014 to add security features.
  - Covers transport layer security, data encryption at rest, non-repudiation, A&A, ...

# Monitoring Kafka apps: Consumer lag



- **Lag** is a consumer problem:
  - Too slow, too much GC, losing connection to ZK or Kafka, ...
  - Bug or design flaw in Consumer
  - Operational mistakes: e.g. you brought up 6 servers in parallel, each one in turn triggering rebalancing, then hit Kafka's rebalance limit;  
cf. `rebalance.max.retries` (default: 4) & friends

# Monitoring Kafka itself (1 of 3)

- **Under-replicated partitions**

- For example, because a broker is down.
- Means cluster runs in degraded state.
  - FYI: LinkedIn runs with replication factor of 2 => 1 broker can die.

- **Offline partitions**

- Even worse than under-replicated partitions!
- *Serious* problem (data loss) if anything but 0 offline partitions.

# Monitoring Kafka itself (1 of 3)

- Data size on disk
  - Should be balanced across disks/brokers
  - Data balance even more important than partition balance
  - There are scripts for balancing data/partitions across brokers
- Broker partition balance
  - Count of partitions should be balanced evenly across brokers

# Monitoring Kafka itself (1 of 3)

- Leader partition count
  - Should be balanced across brokers so that each broker gets the same amount of load
  - Only 1 broker is ever the leader of a given partition, and only this broker is going to talk to producers + consumers for that partition
    - Non-leader replicas are used solely as safeguards against data loss
  - Recent Feature to support auto-rebalance the leaders and partitions in case a broker dies
- Network utilization
  - Maxed network one reason for under-replicated partitions
  - LinkedIn don't run anything but Kafka on the brokers, so network max is due to Kafka. Hence, when they max the network, they need to add more capacity across the board.

# Monitoring ZooKeeper

- Ensemble (= cluster) availability
  - LinkedIn run 5-node ensembles = tolerates 2 dead
  - Twitter run 13-node ensembles = tolerates 6 dead
- Latency of requests
  - Metric target is 0 ms when using SSD's in ZooKeeper machines.
    - Why? Because SSD's are so fast they typically bring down latency below ZK's metric granularity (which is per-ms).
- Outstanding requests
  - Metric target is 0.
  - Why? Because ZK processes all incoming requests serially. Non-zero values mean that requests are backing up.

# “Auditing” Kafka

- Every **producer** is also writing messages into a special topic about how many messages it produced, every 10mins.
  - Example: "Over the last 10mins, I sent N messages to topic X."
  - This metadata gets mirrored like any other Kafka data.
- **Audit consumer**
  - 1 audit consumer per Kafka cluster
  - Reads *every* single message out of “its” Kafka cluster. It then calculates counts for each topic, and writes those counts back into the same special topic, every 10mins.
    - Example: "I saw M messages in the last 10mins for topic X in THIS cluster"
  - And the next audit consumer in the next, downstream cluster does the same thing.

# “Auditing” Kafka

- Monitoring audit consumers
  - Completeness check
    - “#msgs according to producer == #msgs seen by audit consumer?”
  - Lag
    - “Can the audit consumers keep up with the incoming data rate?”
    - If audit consumers fall behind, then all your tracking data falls behind as well, and you don't know how many messages got produced.



# “Auditing” Kafka

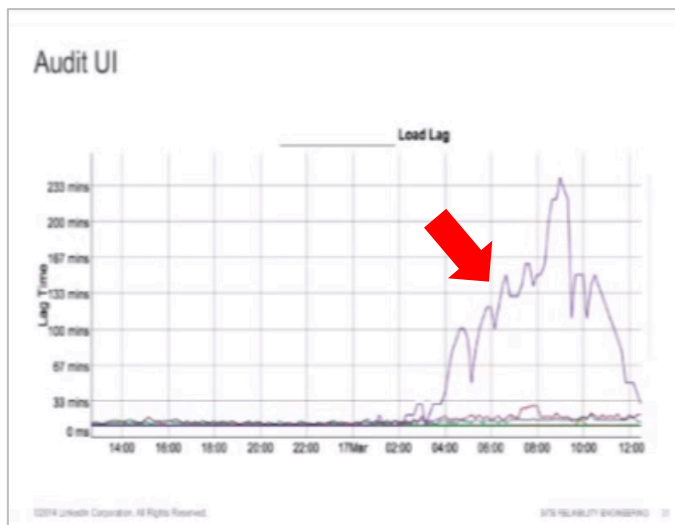
## ■ Audit UI

- Only reads data from that special "metrics/monitoring" topic, but this data is read from every Kafka cluster at LinkedIn.
  - What they producers said they wrote in.
  - What the audit consumers said they saw.
- Shows correlation graphs (producers vs. audit consumers)
  - For each tier, it shows how many messages there were in each topic over any given period of time.
  - Percentage of how much data got through (from cluster to cluster).
  - If the percentage drops below 100%, then emails are sent to Kafka SRE+DEV as well as their Hadoop ETL team because that stops the Hadoop pipelines from functioning properly.

# LinkedIn's Audit UI: a closing look



- Example 1: Count discrepancy
  - Caused by messages failing to reach a downstream Kafka cluster



- Example 2: Load lag

# Kafka Performance Tuning

## ■ OS Kernel tuning

- Don't swap! `vm.swappiness = 0` (RHEL 6.5 onwards: 1)
- Allow more dirty pages but less dirty cache.
  - LinkedIn have lots of RAM in servers, most of it is for page cache (60 of 64 GB). They let dirty pages built up, but cache should be available as Kafka does lots of disk and network I/O.
  - See `vm.dirty*_ratio` & friends

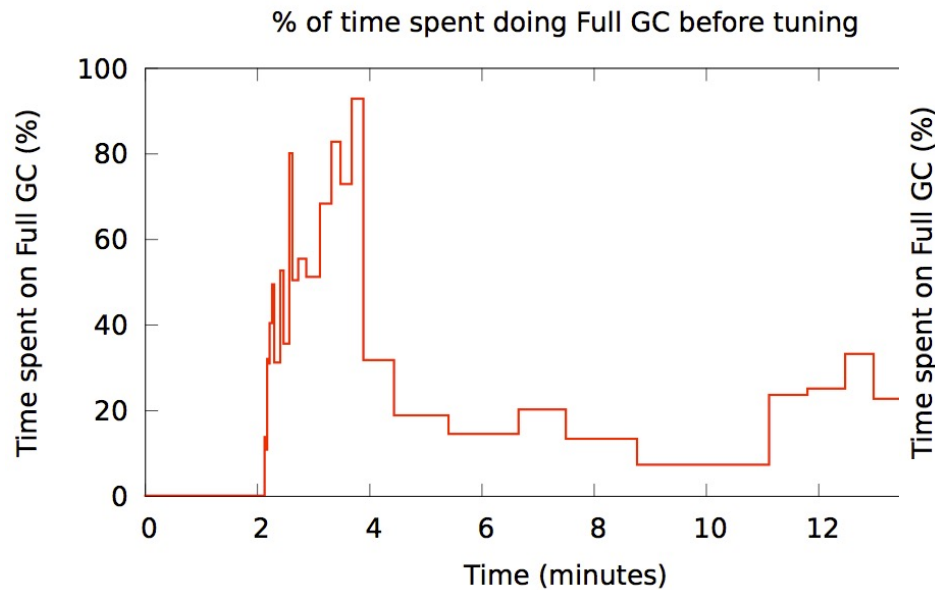
## ■ Disk throughput

- Longer commit interval on mount points. (ext3 or ext4)
  - Normal interval for ext3 mount point is 30s between flushes; LinkedIn: 120s. They can tolerate losing 2mins worth of data (because of partition replicas) so they rather prefer higher throughput here.
- More spindles (RAID10 w/ 14 disks)

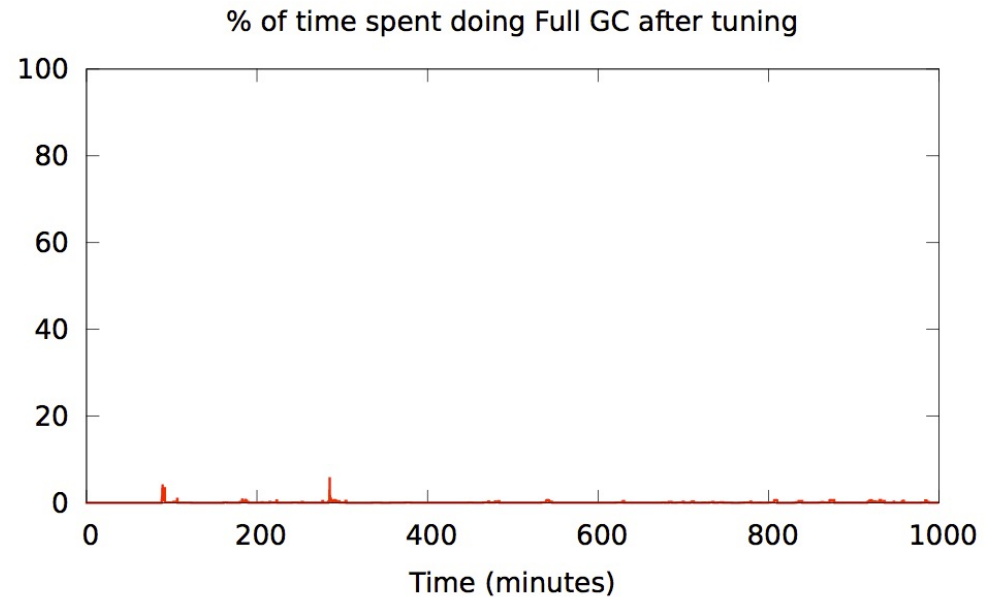
# Java/ JVM tuning for Kafka

- Biggest issue: Garbage Collection (GC)
  - And, most of the time, the only issue
  
- Goal is to **minimize GC pause times**
  - Aka “stop-the-world” events – apps are halted until GC finishes

# Java garbage collection in Kafka @ Spotify



**Before tuning**



**After tuning**

<https://www.jfokus.se/jfokus14/preso/Reliable-real-time-processing-with-Kafka-and-Storm.pdf>

# Kafka configuration tuning

- Often not much to do beyond using the defaults,
- Key candidates for tuning:

`num.io.threads`

should be  $\geq$  `#disks` (start testing with  $=$  `#disks`)

`num.network.threads`

adjust it based on (concurrent) `#producers`,  
`#consumers`, and replication factor

# Kafka usage tuning – lessons learned from others

- Don't break things up into separate topics unless the data in them is truly independent.
  - Consumer behavior can (and will) be extremely variable, don't assume you will always be consuming as fast as you are producing.
- Keep time related messages in the same partition.
  - Consumer behavior can be extremely variable, don't assume the lag on all your partitions will be similar.
  - Design a partitioning scheme, so that the owner of one partition can stop consuming for a long period of time and your application will be minimally impacted (for example, partition by transaction id)

<http://grokbase.com/t/kafka/users/145qtx4z1c/topic-partitioning-strategy-for-large-data>

## Further Reading

- Neha Narkhede, Gwen Shapira, Todd Palino, “Kafka – The Definitive Guide,” published by O’Reilly, July 2017.
  - **Complimentary Free Copy available** from the Confluent website  
<https://www.confluent.io/wp-content/uploads/confluent-kafka-definitive-guide-complete.pdf>