Big Data - Lecture 19

Realtime (Streaming) Processing / Storm

The only reason for time is so that everything doesn’t happen at once.

~Albert Einstein

What is real-time?

From Wikipedia…

- In computer science, **real-time computing** (**RTC**), or **reactive computing**, is the study of hardware and software systems that are subject to a "real-time constraint"— e.g. operational deadlines from event to system response.
- Real-time programs must guarantee response within strict time constraints, often referred to as "deadlines".
- Real-time responses are often **understood to be in the order of milliseconds, and sometimes microseconds**. Conversely, a system without real-time facilities, cannot *guarantee* a response within any timeframe (regardless of *actual* or *expected* response times).
- Jeb Corlis: http://www.youtube.com/watch?v=TWfph3iNC-k

Realtime vs. Batch

- The past decade has seen a revolution in data processing.
- MapReduce, Hadoop, and related technologies have made it possible to store and process data at scales previously unthinkable.
- Unfortunately, these data processing technologies are **not realtime systems, nor are they meant to be**.
- There's no hack that will turn Hadoop into a real-time system; realtime data processing has a fundamentally different set of requirements than batch processing.
- However, real-time data processing at massive scale is becoming more and more of a requirement for businesses. The lack of a "Hadoop of realtime" has become the biggest hole in the data processing ecosystem.

Previously...

Before Storm, you would typically have to manually build a network of queues and workers to do real-time processing.

Workers would process messages off a queue, update databases, and send new messages to other queues for further processing.

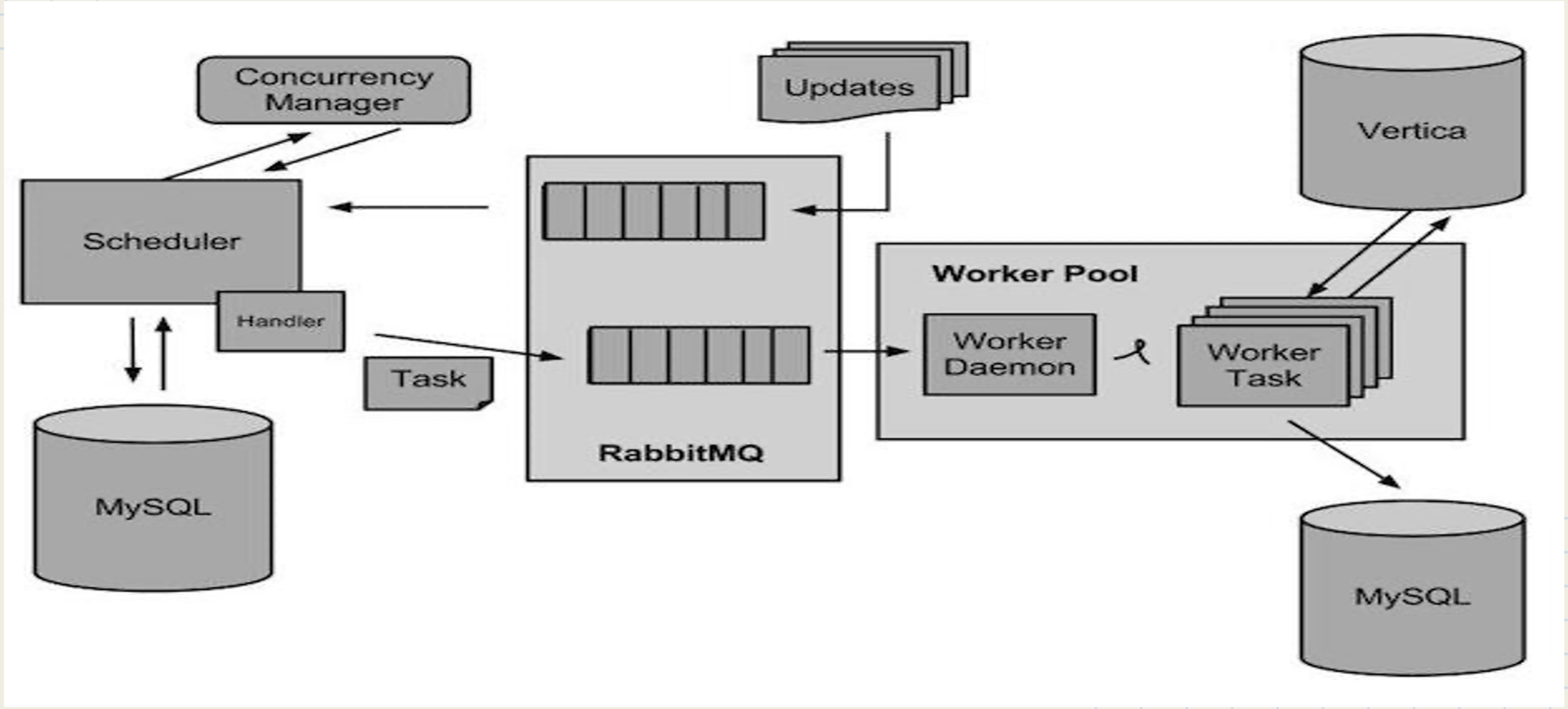
Disadvantages:

1. **Tedious**: You spend most of your development time configuring where to send messages, deploying workers, and deploying intermediate queues. The real-time processing logic that you care about corresponds to a relatively small percentage of your codebase.

1. **Brittle**: There's little fault-tolerance. You're responsible for keeping each worker and queue up.

1. **Painful to scale**: When the message throughput get too high for a single worker or queue, you need to partition how the data is spread around. You need to reconfigure the other workers to know the new locations to send messages. This introduces moving parts and new pieces that can fail.

Queue & Workers Design



Message processing

- Although the queues and workers paradigm breaks down for large numbers of messages, **message processing is clearly the fundamental paradigm for real-time computation**.
- The question is: how do you do it in a way that doesn't lose data, scales to huge volumes of messages, and is dead-simple to use and operate?

Storm

Heron - the successor by Twitter.

Nathan Marz on Storm: http://www.youtube.com/watch?v=bdps8tE0gYo

Storm

- **Storm** is a distributed computation framework written predominantly in the Clojure programming language. Originally created by Nathan Marz and team at BackType, the project was open sourced after being acquired by Twitter.

- Storm exposes a set of primitives for doing realtime computation. Like how MapReduce greatly eases the writing of parallel batch processing, Storm's primitives greatly ease the writing of parallel realtime computation.

Storm’s key properties

1. **Extremely broad set of use cases**: Storm can be used for processing messages and updating databases (stream processing), doing a continuous query on data streams and streaming the results into clients (continuous computation), parallelizing an intense query like a search query on the fly (distributed RPC), and more. Storm's small set of primitives satisfy a stunning number of use cases.
2. **Scalable**: Storm scales to massive numbers of messages per second. To scale a topology, all you have to do is add machines and increase the parallelism settings of the topology. As an example of Storm's scale, one of Storm's initial applications processed 1,000,000 messages per second on a 10 node cluster, including hundreds of database calls per second as part of the topology. Storm's usage of Zookeeper for cluster coordination makes it scale to much larger cluster sizes.
3. **Guarantees no data loss**: A realtime system must have strong guarantees about data being successfully processed. A system that drops data has a very limited set of use cases. Storm guarantees that every message will be processed.
4. **Extremely robust**: Unlike systems like Hadoop, which are notorious for being difficult to manage, Storm clusters just work. It is an explicit goal of the Storm project to make the user experience of managing Storm clusters as painless as possible.
5. **Fault-tolerant**: If there are faults during execution of your computation, Storm will reassign tasks as necessary. Storm makes sure that a computation can run forever (or until you kill the computation).
6. **Programming language agnostic**: Robust and scalable realtime processing shouldn't be limited to a single platform. Storm topologies and processing components can be defined in any language, making Storm accessible to nearly anyone.

Storm Clusters

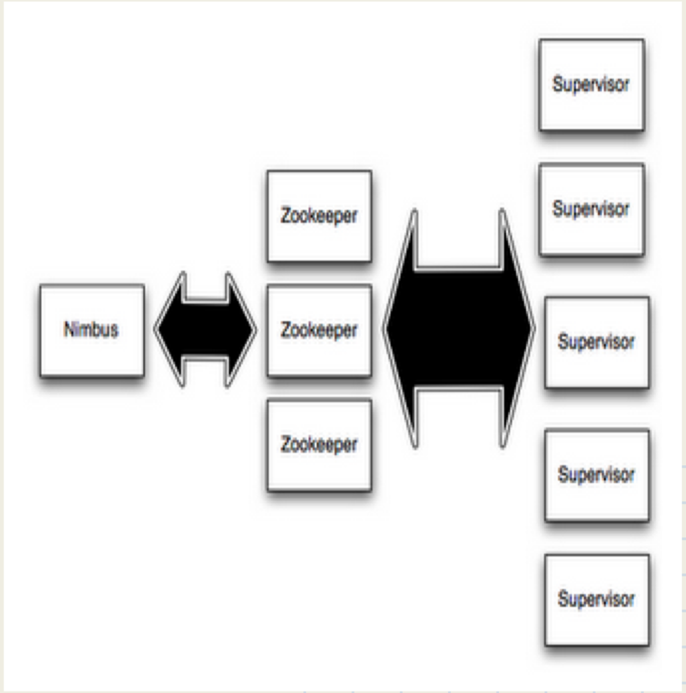
- A Storm cluster is superficially similar to a Hadoop cluster. Whereas on Hadoop you run "MapReduce jobs", on Storm you run "topologies".

- There are two kinds of nodes on a Storm cluster: the **master** node and the **worker** nodes.

- The master node runs a daemon called "Nimbus" that is similar to Hadoop's "JobTracker". Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

- Each worker node runs a **daemon called the "Supervisor"**. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it.

- Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

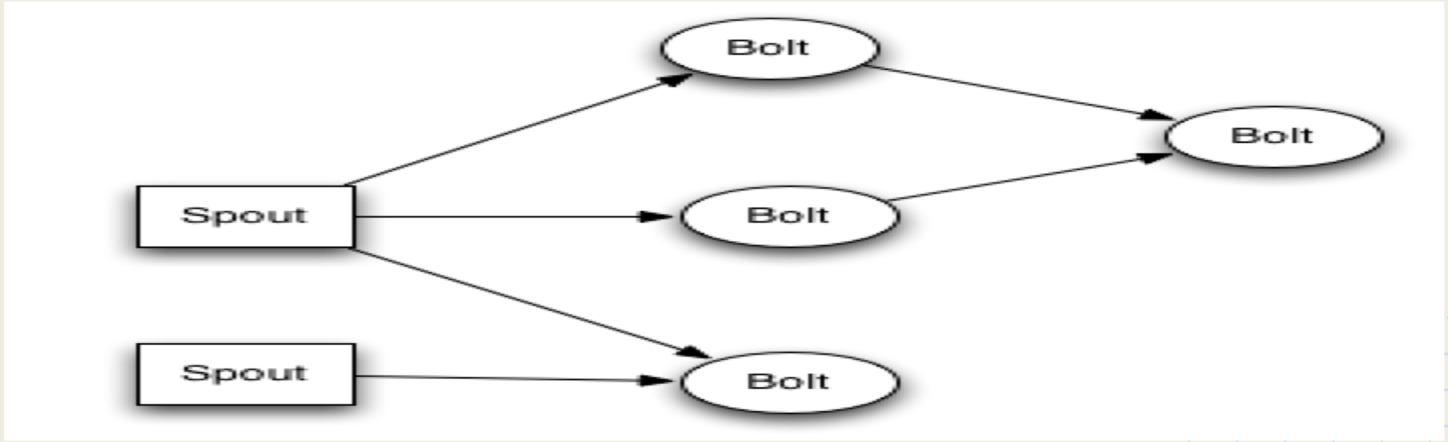


Streams

- The core abstraction in Storm is the **"stream"**- an **unbounded sequence of tuples**.

- Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. For example, you may transform a stream of tweets into a stream of trending topics.

- The basic primitives Storm provides for doing stream transformations are
  - **"spouts"** and
  - **"bolts"**.



What is a Tuple (again)?

- Storm uses tuples as its data model.
- **A tuple is a named list of values**, and a field in a tuple can be an **object of any type**.
- Out of the box, Storm supports all the primitive types, strings, and byte arrays as tuple field values. To use an object of another type, you just need to implement a serializer for the type.
- Every node in a topology must declare the output fields for the tuples it emits.

Spouts

- Spouts and bolts have interfaces that you implement to run your application-specific logic.
- A spout is a source of streams. For example, a spout may read tuples off of a Kestrel queue and emit them as a stream. Or a spout may connect to the Twitter API and emit a stream of tweets.

Example: Word Spout

```
public void nextTuple() {
    Utils.sleep(100);
    final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
    final Random rand = new Random();
    final String word = words[rand.nextInt(words.length)];
    _collector.emit(new Values(word));
}
```

Bolts

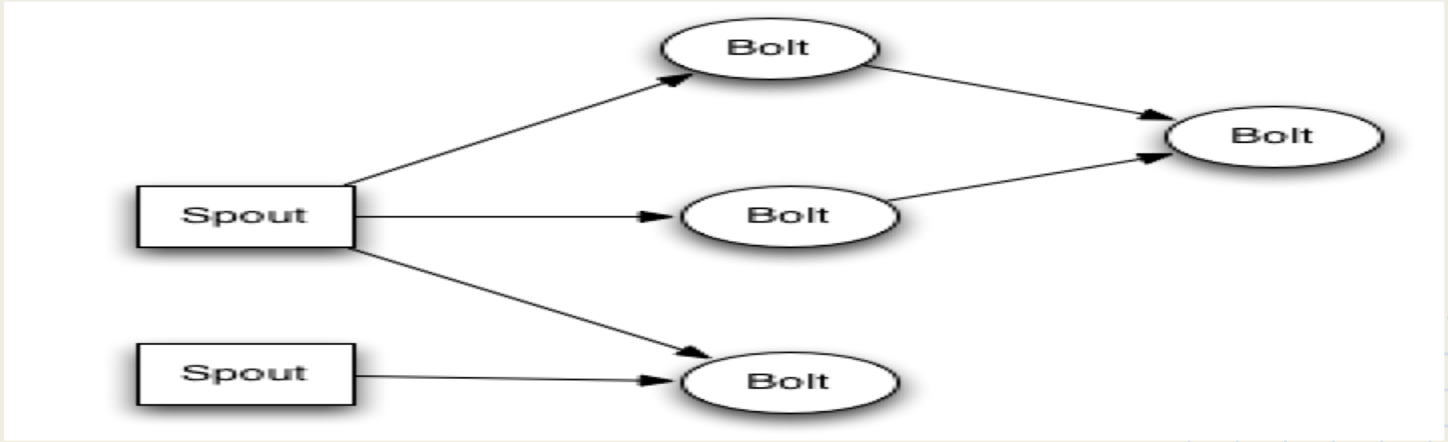
- A bolt consumes any number of input streams, does some processing, and possibly emits new streams.
- Complex stream transformations, like computing a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts.
- Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

Example Bolt (!!!)

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

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

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

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

Topology

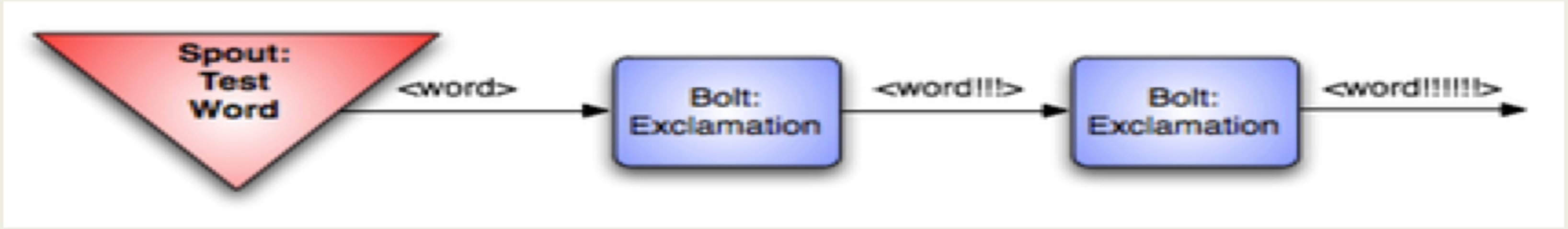
- Networks of spouts and bolts are packaged into a "topology" which is the top-level abstraction that you submit to Storm clusters for execution.
- A topology is a graph of stream transformations where each node is a spout or bolt.
- Edges in the graph indicate which bolts are subscribing to which streams.
- When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.



Example

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3).shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2).shuffleGrouping("exclaim1");
```

This topology contains a spout and two bolts.



The spout emits words, and each bolt appends the string "!!!" to its input.

The nodes are arranged in a line: the spout emits to the first bolt which then emits to the second bolt.

If the spout emits the tuples ["bob"] and ["john"], then the second bolt will emit the words ["bob!!!!!!"] and ["john!!!!!!"].

Multiple inputs

If you wanted component "exclaim2" to read all the tuples emitted by both component "words" and component "exclaim1", you would write component "exclaim2"'s definition like this:

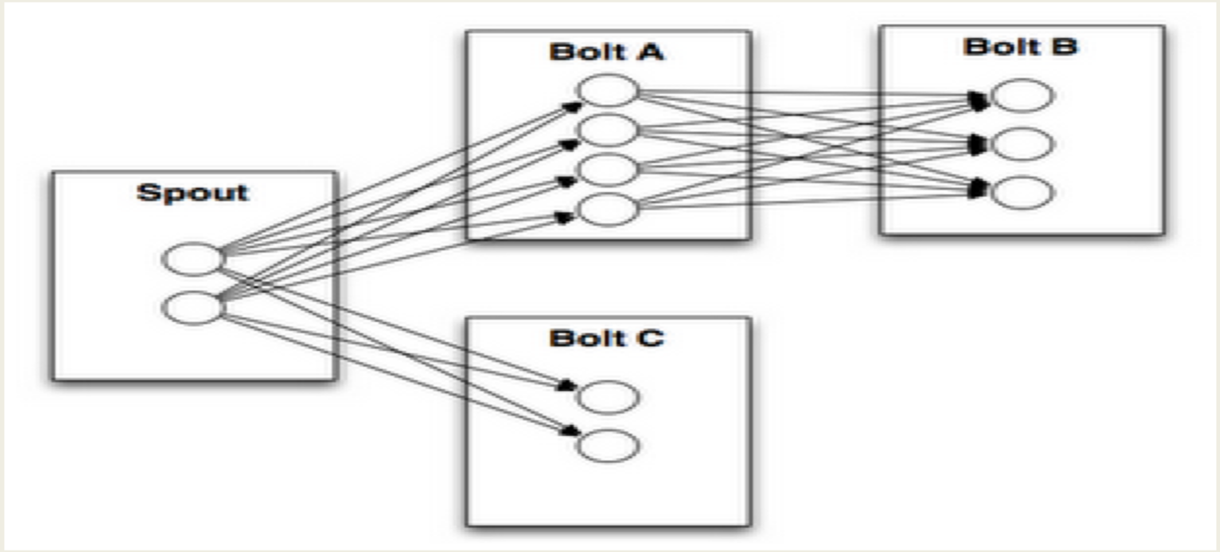
```
builder.setBolt("exclaim2", new ExclamationBolt(), 5)
            .shuffleGrouping("words")
            .shuffleGrouping("exclaim1");
```

As you can see, input declarations can be chained to specify multiple sources for the Bolt.

Stream Groupings

A stream grouping tells a topology how to send tuples between two components. Remember, spouts and bolts execute in parallel as many tasks across the cluster.

If you look at how a topology is executing at the task level, it looks something like this:



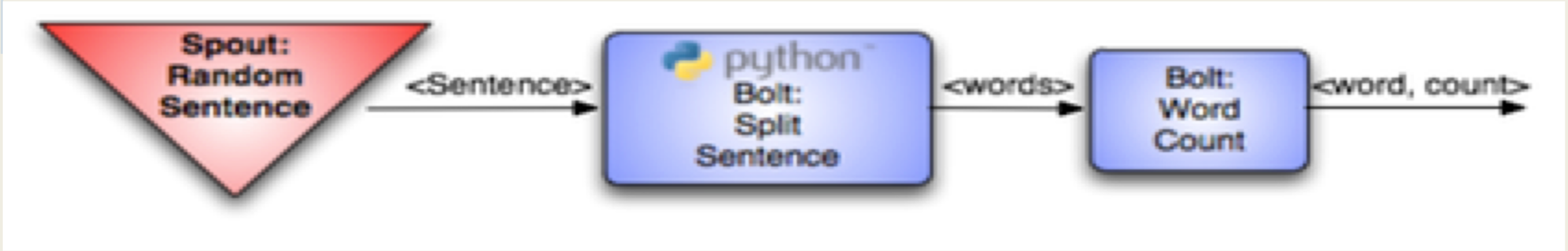
When a task for Bolt A emits a tuple to Bolt B, which task should it send the tuple to?

Shuffle Grouping

- The simplest kind of grouping is called a "shuffle grouping" which sends the tuple to a random task.
- It has the effect of evenly distributing the work of processing the tuples across all of a bolt's tasks.

Fields Grouping

A more interesting kind of grouping is the "fields grouping".



A fields grouping is used between the SplitSentence bolt and the WordCount bolt.

It is critical for the functioning of the WordCount bolt that the same word always go to the same task. Otherwise, more than one task will see the same word, and they'll each emit incorrect values for the count since each has incomplete information.

A fields grouping lets you group a stream by a subset of its fields. This causes equal values for that subset of fields to go to the same task. Since WordCount subscribes to SplitSentence's output stream using a fields grouping on the "word" field, the same word always goes to the same task and the bolt produces the correct output.

Fields groupings are the basis of implementing streaming joins and streaming aggregations as well as a plethora of other use cases. Underneath the hood, fields groupings are implemented using mod hashing.

Topologies implemented

- To do realtime computation on Storm, you create what are called "topologies". **A topology is a graph of computation**. Each node in a topology contains processing logic, and links between nodes indicate how data should be passed around between nodes.

- Running a topology is straightforward. First, you package all your code and dependencies into a single jar. Then, you run a command like the following:

```
storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2
```

- This runs the class backtype.storm.MyTopology with the arguments arg1 and arg2.

- The main function of the class defines the topology and submits it to Nimbus. The storm jar part takes care of connecting to Nimbus and uploading the jar.

- Since topology definitions are just Thrift structs, and Nimbus is a Thrift service, **you can create and submit topologies using any programming language**.

Local vs. Distributed Mode

- Storm has two modes of operation:
  - local mode and
  - distributed mode.

- In local mode, Storm executes completely in process by simulating worker nodes with threads. Local mode is useful for testing and development of topologies.

- In distributed mode, Storm operates as a cluster of machines. When you submit a topology to the master, you also submit all the code necessary to run the topology. The master will take care of distributing your code and allocating workers to run your topology. If workers go down, the master will reassign them somewhere else.

Guaranteed Message Processing

Storm's reliability API: how Storm guarantees that every message coming off a spout will be fully processed.

Lenghty discussion of why here:
http://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html

Transactional Topologies

- Storm guarantees that every message will be played through the topology at least once.

- A common question asked is "how do you do things like counting on top of Storm? Won't you overcount?"

- Storm has a feature called transactional topologies that let you achieve exactly-once messaging semantics for most computations.

- Detailed explanation here: http://storm.apache.org/documentation/Transactional-topologies.html

Resources

- http://storm.apache.org/
- Heron (next gen Storm at Twitter): https://twitter.github.io/heron/
- http://nathanmarz.com/ (author of Storm)
- Nathan’s Youtube video from class: http://www.youtube.com/watch?v=bdps8tE0gYo
- http://aws.amazon.com/kinesis/ (Amazon’s entry in realtime)
- Heron (successor to Storm): https://twitter.github.io/heron/

Other Realtime Frameworks:

- Apache Flink: https://flink.apache.org/
- Kafka Streams: https://kafka.apache.org/documentation/streams/