Storm
Distributed and fault-tolerant realtime computation

Nathan Marz
Twitter
Basic info

- Open sourced September 19th
- Implementation is 15,000 lines of code
- Used by over 25 companies
- >2700 watchers on Github (most watched JVM project)
- Very active mailing list
  - >2300 messages
  - >670 members
Before Storm

Queues

Workers
Example

(simplified)
Example

Workers schemify tweets and append to Hadoop
Example

Workers update statistics on URLs by incrementing counters in Cassandra
Example

Use mod/hashing to make sure same URL always goes to same worker
Scaling

Deploy

Reconfigure/redeploy

Twitter firehose

Hadoop

Cassandra
Problems

• Scaling is painful
• Poor fault-tolerance
• Coding is tedious
What we want

• Guaranteed data processing
• Horizontal scalability
• Fault-tolerance
• No intermediate message brokers!
• Higher level abstraction than message passing
• “Just works”
Storm

- Guaranteed data processing
- Horizontal scalability
- Fault-tolerance
- No intermediate message brokers!
- Higher level abstraction than message passing
- “Just works”
Use cases

Stream processing

Distributed RPC

Continuous computation
Storm Cluster
Storm Cluster

Master node (similar to Hadoop JobTracker)
Storm Cluster

Used for cluster coordination
Storm Cluster

Run worker processes
Starting a topology

```
storm jar mycode.jar twitter.storm.MyTopology demo
```
Killing a topology

storm kill demo
Concepts

- Streams
- Spouts
- Bolts
- Topologies
Streams

Unbounded sequence of tuples
Spouts

Source of streams
Spout examples

- Read from Kestrel queue
- Read from Twitter streaming API
Bolts

Processes input streams and produces new streams
Bolts

- Functions
- Filters
- Aggregation
- Joins
- Talk to databases
Topology

Network of spouts and bolts
Tasks

Spouts and bolts execute as many tasks across the cluster
Task execution

Tasks are spread across the cluster
Task execution

Tasks are spread across the cluster
Stream grouping

When a tuple is emitted, which task does it go to?
Stream grouping

- **Shuffle grouping**: pick a random task
- **Fields grouping**: mod hashing on a subset of tuple fields
- **All grouping**: send to all tasks
- **Global grouping**: pick task with lowest id
Streaming word count

TopologyBuilder builder = new TopologyBuilder();

TopologyBuilder is used to construct topologies in Java
Streaming word count

```java
builder.setSpout("spout",
    new KestrelSpout(
        "kestrel.twitter.com"
        22133,
        "sentence_queue"),
    5);
```

Define a spout in the topology with parallelism of 5 tasks
Streaming word count

Split sentences into words with parallelism of 8 tasks

```java
builder.setBolt("split", new SplitSentence(), 8)
  .shuffleGrouping("spout");
```
Streaming word count

Consumer decides what data it receives and how it gets grouped

Split sentences into words with parallelism of 8 tasks

```java
builder.setBolt("split", new SplitSentence(), 8)
   .shuffleGrouping("spout");
```
Streaming word count

```java
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word"));
```

Create a word count stream
Streaming word count

```python
import storm

class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])
```

```java
public static class SplitSentence extends ShellBolt implements IRichBolt {
    public SplitSentence() {
        super("python", "splitsentence.py");
    }

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

```java
public static class WordCount implements IBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void prepare(Map conf, TopologyContext context) {
    }

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    public void cleanup() {
    }

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

Map conf = new HashMap();
conf.put(Config.TOPOLOGY_WORKERS, 10);
StormSubmitter.submitTopology("word-count", conf, builder.createTopology());

Submitting topology to a cluster
Streaming word count

```
LocalCluster cluster = new LocalCluster();

Map conf = new HashMap();
conf.put(Config.TOPOLOGY_DEBUG, true);

cluster.submitTopology("demo", conf, builder.createTopology());
```

Running topology in local mode
Distributed RPC

Data flow for Distributed RPC
DRPC Example

Computing “reach” of a URL on the fly
Reach

Reach is the number of unique people exposed to a URL on Twitter
Computing reach

URL → Tweeter → Follower → Distinct follower → Count → Reach
Reach topology
Reach topology

```java
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"));
```
public static class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();

    public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, Object id) {
        _collector = collector;
        _id = id;
    }

    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }

    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }
}
Reach topology

```java
public static class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();

    public void prepare(Map conf, TopologyContext context,
                         BatchOutputCollector collector, Object id) {
        _collector = collector;
        _id = id;
    }

    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }

    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }
}
```

Keep set of followers for each request id in memory
Reach topology

```java
public static class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();

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

    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }

    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }
}
```

Update followers set when receive a new follower
Reach topology

```java
public static class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();

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

    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }

    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }
}
```
Guaranteeing message processing

“Tuple tree”
Guaranteeing message processing

- A spout tuple is not fully processed until all tuples in the tree have been completed
Guaranteeing message processing

- If the tuple tree is not completed within a specified timeout, the spout tuple is replayed
Guaranteeing message processing

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

Reliability API
Guaranteeing message processing

```
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
Guaranteeing message processing

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

Marks a single node in the tree as complete
Guaranteeing message processing

• Storm tracks tuple trees for you in an extremely efficient way
Transactional topologies

How do you do idempotent counting with an at least once delivery guarantee?
Transactional topologies

Won’t you overcount?
Transactional topologies solve this problem
Transactional topologies

Built completely on top of Storm's primitives of streams, spouts, and bolts
Transactional topologies

Enables fault-tolerant, exactly-once messaging semantics
Transactional topologies

Batch 1  Batch 2  Batch 3

Process small batches of tuples
Transactional topologies

If a batch fails, replay the whole batch
Transactional topologies

Once a batch is completed, commit the batch
Transactional topologies

Batch 1  Batch 2  Batch 3

Bolts can optionally be “committers”
Commits are ordered. If there’s a failure during commit, the whole batch + commit is retried.
Example

```java
public class IdempotentCountingBolt extends BaseTransactionalBolt implements ICommitter {
    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context, 
                        BatchOutputCollector collector, 
                        TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
        CurrentValue current = getCurrentValue();
        if (current.txid!= _attempt.getTransactionId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }
}
```
Example

```java
public class IdempotentCountingBolt extends BaseTransactionalBolt
    implements ICommitter {

    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context,
        BatchOutputCollector collector,
        TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
       CurrentValue current = getCurrentValue();
        if (current.txid != _attempt.getTransactionId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }
```
Example

```java
public class IdempotentCountingBolt extends BaseTransactionalBolt
    implements ICommitter {
    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context,
                         BatchOutputCollector collector,
                         TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
        CurrentValue current = getCurrentValue();
        if(current.txid!=_attempt.getTransactionId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }
}
```
Example

```java
public class IdempotentCountingBolt extends BaseTransactionalBolt
    implements ICommitter {
    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context,
        BatchOutputCollector collector,
        TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
       CurrentValue current = getCurrentValue();
        if(current.txid != _attempt.getTransactionId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }
}
```

Only update database if transaction ids differ
Example

public class IdempotentCountingBolt extends BaseTransactionalBolt
    implements ICommitter {

    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context,
                         BatchOutputCollector collector,
                         TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
       CurrentValue current = getCurrentValue();
        if (current.txid != _attempt.getTransactionIdId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }

    This enables idempotency since commits are ordered
public class IdempotentCountingBolt extends BaseTransactionalBolt
    implements ICommitter {
    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _count = 0;

    public void prepare(Map conf, TopologyContext context,
                         BatchOutputCollector collector,
                         TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    public void execute(Tuple tuple) {
        _count += 1;
    }

    public void finishBatch() {
       CurrentValue current = getCurrentValue();
        if(current.txid!=_attempt.getTransactionId()) {
            setCurrentValue(current.count + _count, _attempt.getTransactionId());
        }
    }

    (Credit goes to Kafka devs for this trick)
Transactional topologies

Multiple batches can be processed in parallel, but commits are guaranteed to be ordered.
Transactional topologies

• Requires a more sophisticated source queue than Kestrel or RabbitMQ

• storm-contrib has a transactional spout implementation for Kafka
### Storm UI

#### Topology summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Id</th>
<th>Uptime</th>
<th>Num workers</th>
<th>Num tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>poseidon</td>
<td>poseidon-1-1314658150</td>
<td>23h 17m 0s</td>
<td>80</td>
<td>765</td>
</tr>
</tbody>
</table>

#### Topology stats

<table>
<thead>
<tr>
<th>Window</th>
<th>Emitted</th>
<th>Transferred</th>
<th>Complete latency (ms)</th>
<th>Acked</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10m 0s</td>
<td>24786020</td>
<td>24786000</td>
<td>4131.688</td>
<td>2338940</td>
<td>0</td>
</tr>
<tr>
<td>3h 0m 0s</td>
<td>621695800</td>
<td>621694600</td>
<td>4463.830</td>
<td>59353840</td>
<td>0</td>
</tr>
<tr>
<td>1d 0h 0m 0s</td>
<td>4447725560</td>
<td>4447716960</td>
<td>4278.459</td>
<td>438710100</td>
<td>0</td>
</tr>
<tr>
<td>All time</td>
<td>4447725560</td>
<td>4447716960</td>
<td>4278.459</td>
<td>438710100</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Spouts (All time)

<table>
<thead>
<tr>
<th>Id</th>
<th>Parallelism</th>
<th>Emitted</th>
<th>Transferred</th>
<th>Complete latency (ms)</th>
<th>Acked</th>
<th>Failed</th>
<th>Last error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>160</td>
<td>877453060</td>
<td>877453060</td>
<td>4278.459</td>
<td>438710100</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### Bolts (All time)

<table>
<thead>
<tr>
<th>Id</th>
<th>Parallelism</th>
<th>Emitted</th>
<th>Transferred</th>
<th>Process latency (ms)</th>
<th>Acked</th>
<th>Failed</th>
<th>Last error</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>4</td>
<td>438716440</td>
<td>438716440</td>
<td>0.009</td>
<td>2223890060</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>877451720</td>
<td>877451720</td>
<td>0.320</td>
<td>438725980</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>160</td>
<td>1264258160</td>
<td>1264258160</td>
<td>5.438</td>
<td>438724980</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>55946080</td>
<td>55946080</td>
<td>0.215</td>
<td>55946040</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>55947280</td>
<td>55947280</td>
<td>0.121</td>
<td>55947280</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>55945660</td>
<td>55945660</td>
<td>0.229</td>
<td>55945660</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>55946480</td>
<td>55946480</td>
<td>0.145</td>
<td>55946580</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>81512620</td>
<td>81512620</td>
<td>0.209</td>
<td>81512620</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>438710060</td>
<td>438710060</td>
<td>4205.639</td>
<td>438710140</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>163904580</td>
<td>163904580</td>
<td>0.181</td>
<td>81512990</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Storm on EC2

https://github.com/nathanmarz/storm-deploy

One-click deploy tool
Starter code

https://github.com/nathanmarz/storm-starter

Example topologies
Documentation

Storm is a distributed realtime computation system. Similar to how Hadoop provides a set of general primitives for doing batch processing, Storm provides a set of general primitives for doing realtime computation. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Read these first

- Rationale
- Setting up development environment
- Creating a new Storm project
- Tutorial

Getting help

Feel free to ask questions on Storm's mailing list: http://groups.google.com/group/storm-user

You can also come to the #storm-user room on freenode. You can usually find a Storm developer there to help you out.

Related projects
Ecosystem

- Scala, JRuby, and Clojure DSL’s
- Kestrel, Redis, AMQP, JMS, and other spout adapters
- Multilang adapters
- Cassandra, MongoDB integration
Questions?

http://github.com/nathanmarz/storm
Future work

• State spout
• Storm on Mesos
• “Swapping”
• Auto-scaling
• Higher level abstractions
Implementation

KafkaTransactionalSpout

[Diagram showing connections between components]
Implementation
TransactionalSpout is a subtopology consisting of a spout and a bolt
The spout consists of one task that coordinates the transactions.
The bolt emits the batches of tuples
The coordinator emits a “batch” stream and a “commit stream”
Implementation

Batch stream
Implementation

Commit stream
Coordinator reuses tuple tree framework to detect success or failure of batches or commits and replays appropriately.