Implementing Real-Time Trending Topics with a Distributed Rolling Count Algorithm in Storm

Jan 18, 2013 · 20 min read

A common pattern in real-time data workflows is performing rolling counts of incoming data points, also known as sliding window analysis. A typical use case for rolling counts is identifying trending topics in a user community – such as on Twitter – where a topic is considered trending when it has been among the top N topics in a given window of time. In this article I will describe how to implement such an algorithm in a distributed and scalable fashion using the Storm real-time data processing platform. The same code can also be used in other areas such as infrastructure and security monitoring.

Update 2014-06-04: I updated several references to point to the latest version of storm-starter, which is now part of the official Storm project.

Table of Contents

About Trending Topics and Sliding Windows

First, let me explain what I mean by "trending topics" so that we have a common understanding. Here is an explanation taken from Wikipedia:

> ***Trending topics***
>
> *A word, phrase or topic that is tagged at a greater rate than other tags is said to be a trending topic. Trending topics become popular either through a concerted effort by users or because of an event that prompts*

> *people to talk about one specific topic. These topics help Twitter and their users to understand what is happening in the world.*
>
> ***Wikipedia page on Twitter*** *en.wikipedia.org/wiki/…*

In other words, it is a measure of "What's hot?" in a user community. Typically, you are interested in trending topics for a given *time span*; for instance, the most popular topics in the past five minutes or the current day. So the question "What's hot?" is more precisely stated as "What's hot *today*?" or "What's hot *this week*?".

In this article we assume we have a system that uses the Twitter API to pull the latest tweets from the live Twitter stream. We assume further that we have a mechanism in place that extracts topical information in the form of words from those tweets. For instance, we could opt to use a simple pattern matching algorithm that treats #hashtags in tweets as topics. Here, we would consider a tweet such as

```
@miguno The #Storm project rocks for real-time distributed #data process
```

to "mention" the topics

```
storm
data
```

We design our system so that it considers topic A more popular than topic B (for a given time span) if topic A has been mentioned more often in tweets than topic B. This means we only need to *count* the number of occurrences of topics in tweets.

$$popularity(A) \geq popularity(B) \iff mentions(A) \geq mentions(B)$$

For the context of this article we do not care how the topics are actually derived from user content or user activities as long as the derived topics are represented as textual words. Then, the Storm topology described in this article will be able to identify in real-time the *trending* topics in this input data using a time-sensitive rolling count algorithm (rolling counts are also known as *sliding windows*) coupled with a ranking step. The former aspect takes care of filtering user input by time span, the latter of ranking the most trendy topics at the top the list.

Eventually we want our Storm topology to periodically produce the top N of trending topics similar to the following example output, where *t0* to *t2* are different points in time:

```
Rank @ t0   ----->   t1   ----->   t2
--------------------------------------------
1.    java   (33)   ruby   (41)   scala  (32)
2.    php    (30)   scala  (28)   python (29)
3.    scala  (21)   java   (27)   ruby   (24)
```

```
4.    ruby   (16)   python (21)   java   (22)
5.    python (15)   php    (14)   erlang (18)
```

In this example we can see that over time "scala" has become the hottest trending topic.

Sliding Windows

The last background aspect I want to cover are sliding windows aka rolling counts. A picture is worth a thousand words:

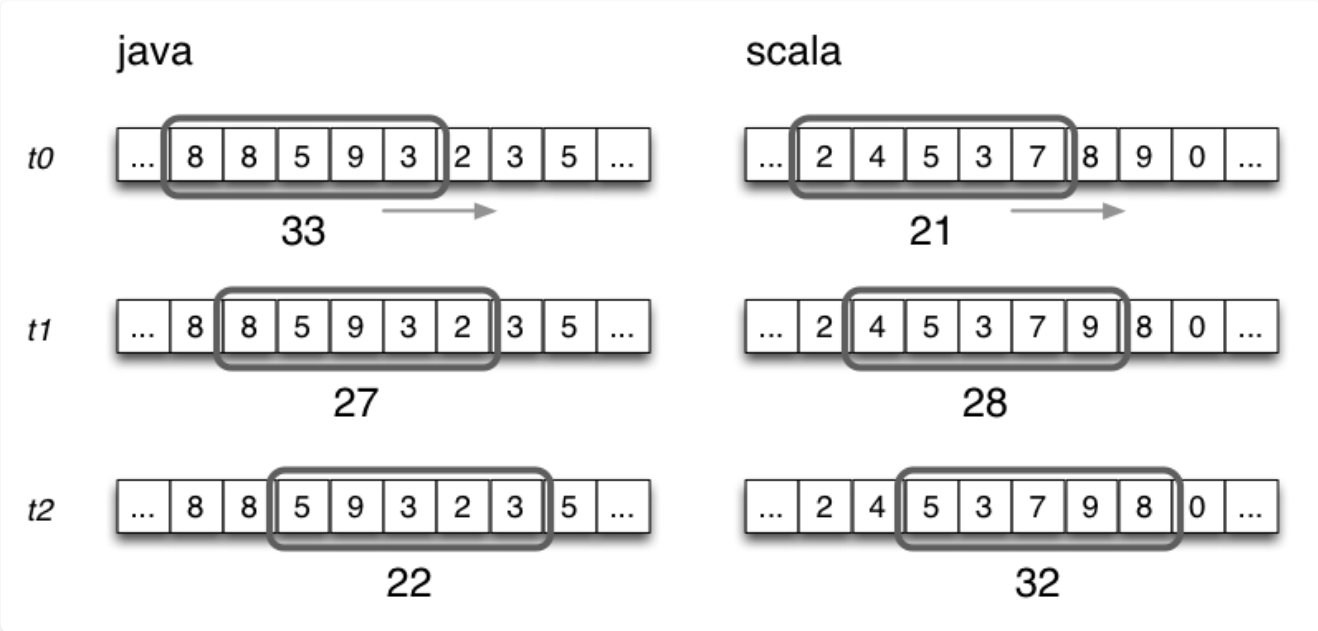

Figure 1: As the sliding window advances, the slice of its input data changes. In the example above the algorithm uses the current sliding window data to compute the sum of the window's elements.

A formula might also be worth a bunch of words – ok, ok, maybe not a full *thousand* of them – so mathematically speaking we could formalize such a sliding-window sum algorithm as follows:

$$\text{m-sized rolling sum} = \sum_{i=t}^{i+m} element(i)$$

where *t* continually advances (most often with time) and *m* is the window size.

From size to time: If the window is advanced with time, say every *N* minutes, then the individual elements in the input represent data collected over the same interval of time (here: *N* minutes). In that case the window size is equivalent to *N x m* minutes. Simply speaking, if *N=1* and *m=5*, then our sliding window algorithm emits the latest five-minute aggregates every one minute.

Now that we have introduced *trending topics* and *sliding windows* we can finally start talking about writing code for Storm that implements all this in practice – large-scale, distributed,

in real time.

Before We Start

About storm-starter

The storm-starter project on GitHub provides example implementations of various real-time data processing topologies such as a simple streaming WordCount algorithm. It also includes a Rolling Top Words topology that can be used for computing trending topics, the purpose of which is exactly what I want to cover in this article.

When I began to tackle trending topic analysis with Storm I expected that I could re-use most if not all of the Rolling Top Words code in `storm-starter`. But I soon realized that the old code would need some serious redesigning and refactoring before one could actually use it in a real-world environment – including being able to efficiently maintain and augment the code in a team of engineers across release cycles.

In the next section I will briefly summarize the state of the Rolling Top Words topology before and after my refactoring to highlight some important changes and things to consider when writing your own Storm code. Then I will continue with covering the most important aspects of the new implementation in further detail. And of course I contributed the new implementation back to the Storm project.

The Old Code and My Goals for the New Code

> Just to absolutely clear here: I am talking about the defects of the old code to highlight some typical pitfalls during software development for a distributed system such as Storm. My intention is to make other developers aware of these gotchas so that we make less mistakes in our profession. I am by no means implying that the authors of the old code did a bad job (after all, the old code was perfectly adequate to get me started with trending topics in Storm) or that the new implementation I came up with is the pinnacle of coding. :-)

My initial reaction to the old code was that, frankly speaking, I had no idea what and how it was doing its job. The various logical responsibilities of the code were mixed together in the existing classes, clearly not abiding by the Single Responsibility Principle. And I am not talking about academic treatments of SRP and such – I was hands-down struggling to wrap my head around the old code because of this.

Also, I noticed a few `synchronized` statements and threads being launched manually, hinting at additional parallel operations beyond what the Storm framework natively provides you with. Here, I was particularly

In practice this dirty-write bug in the old rolling count

concerned with those functionalities that interacted with the system time (calls to `System.currentTimeMillis()`). I couldn't help the feeling that they looked prone to concurrency issues. And my suspicions were eventually confirmed when I discovered a dirty-write bug in the `RollingCountObjects` bolt code for the slot-based counting (using `long[]`) of object occurrences. In practice this dirty-write bug in the old rolling count implementation caused data corruption, i.e. the code was not carrying out its main responsibility correctly – that of counting objects. That said I'd argue that it would not have been trivial to spot this error in the old code prior to refactoring (where it was eventually plain to see), so please don't think it was just negligence on the part of the original authors. With the new tick tuple feature in Storm 0.8 I was feeling confident that this part of the code could be significantly simplified and fixed.

implementation caused data corruption, i.e. the code was not carrying out its main responsibility correctly – that of counting objects.

In general I figured that completely refactoring the code and untangling these responsibilities would not only make the code more approachable and readable for me and others – after all the storm-starter code's main purpose is to jumpstart Storm beginners – but it would also allow me to write meaningful unit tests, which would have been very difficult to do with the old code.

What	Before refactoring	After refactoring
Storm Bolts	RollingCountObjects, RankObjects, MergeObjects	RollingCountBolt, IntermediateRankingsBolt, TotalRankingsBolt,
Storm Spouts	TestWordSpout	TestWordSpout (not modified)
Data Structures	-	SlotBasedCounter, SlidingWindowCounter, Rankings, Rankable, RankableObjectWithFields
Unit Tests	-	Every class has its own suite of tests.
Additional Notes	Uses manually launched background threads instead of native Storm features to execute periodic activities.	Uses new tick tuple feature in Storm 0.8 to trigger periodic activities in Storm components.

Table 1: The state of the trending topics Storm implementation before and after the refactoring.

The design and implementation that I will describe in the following sections are the result of a number of refactoring iterations. I started with smaller code changes that served me primarily to understand the existing code better (e.g. more meaningful variable names, splitting long methods into smaller logical units). The more I felt comfortable the more I started to introduce substantial changes. Unfortunately the existing code was not accompanied by any unit tests, so while refactoring I was in the dark, risking to break something that I was not even aware of breaking. I considered writing unit tests for the existing code first and then go back to refactoring but I figured that this would not be the best approach given the state of the code and the time I had available.

In summary my goals for the new trending topics implementation were:

1. The new code should be clean and easy to understand, both for the benefit of other developers when adapting or maintaining the code and for reasoning about its correctness. Notably, the code should decouple its data structures from the Storm sub-system and, if possible, favor native Storm features for concurrency instead of custom approaches.
2. The new code should be covered by meaningful unit tests.
3. The new code should be good enough to contribute it back to the Storm project to help its community.

Implementing the Data Structures

Eventually I settled down to the following core data structures for the new distributed Rolling Count algorithm. As you will see, an interesting characteristic is that these data structures are completely decoupled from any Storm internals. Our Storm bolts will make use of them, of course, but there is no dependency in the opposite direction from the data structures to Storm.

- Classes used for counting objects: SlotBasedCounter, SlidingWindowCounter
- Classes used for ranking objects by their count: Rankings, Rankable, RankableObjectWithFields

Another notable improvement is that the new code removes any need and use of concurrency-related code such as `synchronized` statements or manually started background threads. Also, none of the data structures are interacting with the system time. Eliminating direct calls to system time and manually started background threads makes the new code much simpler and testable than before.

Eliminating direct calls to system time and manually started background threads makes the new code much simpler and testable than before.

```
// such code from the old RollingCountObjects
long delta = millisPerBucket(_numBuckets)
```

```
                - (System.currentTimeMillis() %
Utils.sleep(delta);
```

SlotBasedCounter

The SlotBasedCounter class provides per-slot counts of the occurrences of objects. The number of slots of a given counter instance is fixed. The class provides four public methods:

```
// SlotBasedCounter API
public void incrementCount(T obj, int slot);
public void wipeSlot(int slot):
public long getCount(T obj, int slot)
// get the *total* counts of all objects across all slots
public Map<T, Long> getCounts();
```

Here is a usage example:

```
// Using SlotBasedCounter

// we want to count Object's using five slots
SlotBasedCounter counter = new SlotBasedCounter<Object>(5);

// counting
Object trackMe = ...;
int currentSlot = 0;
counter.incrementCount(trackMe, currentSlot);

// the counts of an object for a given slot
long counts = counter.getCount(trackMe, currentSlot);

// the total counts (across all slots) of all objects
Map<Object, Long> counts = counter.getCounts();
```

Internally `SlotBasedCounter` is backed by a `Map<T, long[]>` for the actual count state. You might be surprised to see the low-level `long[]` array here – wouldn't it be better OO style to introduce a new, separate class that is just used for the counting of a single slot, and then we use a couple of these single-slot counters to form the SlotBasedCounter? Well, yes we could. But for performance reasons and for not deviating too far from the old code I decided not to go down this route. Apart from updating the counter – which is a WRITE operation – the most common operation in our use case is a READ operation to get the *total* counts of tracked objects. Here, we must calculate the sum of an object's counts *across all slots*. And for this it is preferable to have the individual data points for an object close to each other (kind of data locality), which the `long[]` array allows us to do. Your mileage may vary though.

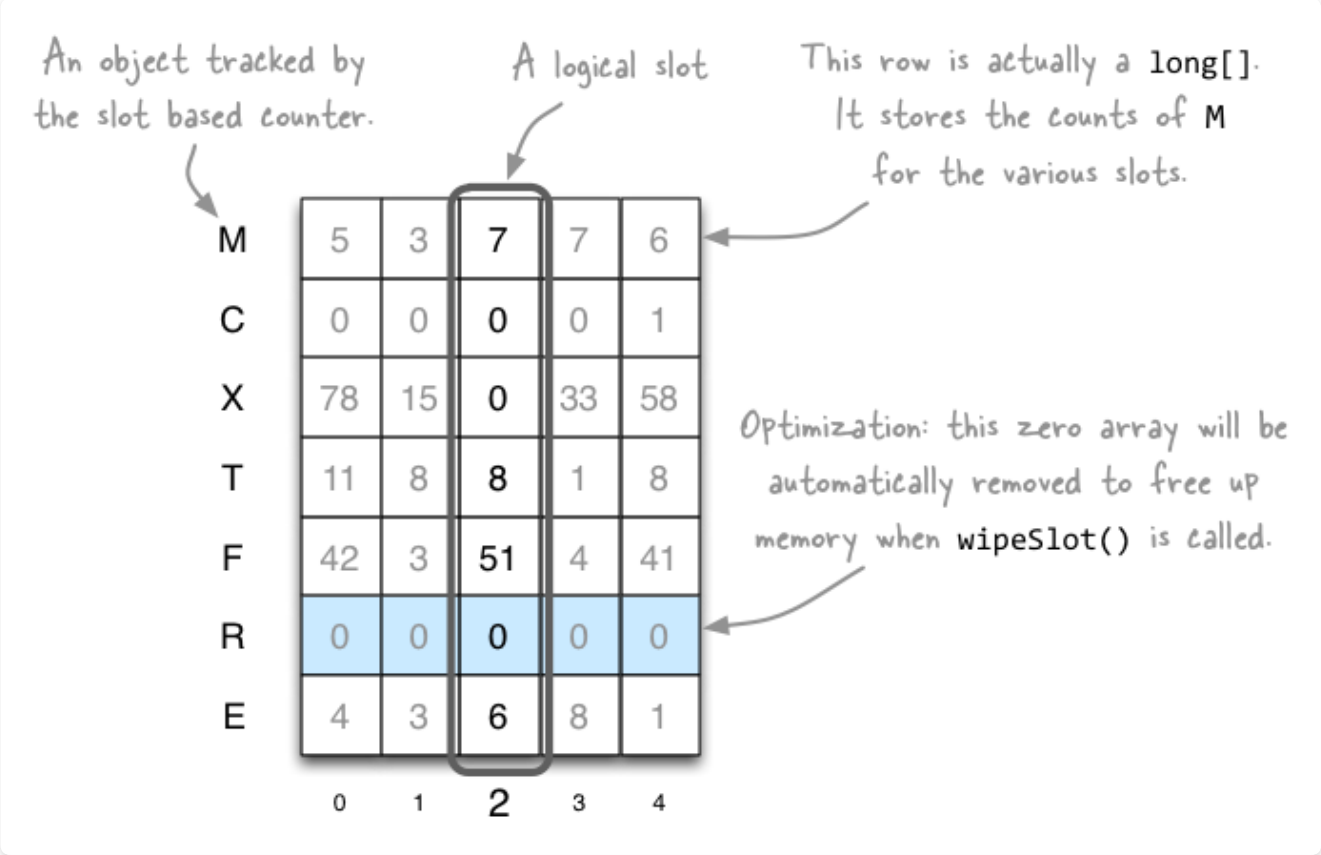

Figure 2: The `SlotBasedCounter` class keeps track of multiple counts of a given object. In the example above, the SlotBasedCounter has five logical slots which allows you to track up to five counts per object.

The `SlotBasedCounter` is a primitive class that can be used, for instance, as a building block for implementing sliding window counting of objects. And this is exactly what I will describe in the next section.

SlidingWindowCounter

The SlidingWindowCounter class provides *rolling* counts of the occurrences of "things", i.e. a sliding window count for each tracked object. Its counting functionality is based on the previously described `SlotBasedCounter`. The size of the sliding window is equivalent to the (fixed) number of slots number of a given `SlidingWindowCounter` instance. It is used by `RollingCountBolt` for counting incoming data tuples.

The class provides two public methods:

```
// SlidingWindowCounter API
public void incrementCount(T obj);
Map<T, Long> getCountsThenAdvanceWindow();
```

What might be surprising to some readers is that this class does not have any notion of time even though "sliding window" normally means a time-based window of some kind. In our case however the window does not advance with time but whenever (and only when) the

method `getCountsThenAdvanceWindow()` is called. This means `SlidingWindowCounter` behaves just like a normal ring buffer in terms of advancing from one window to the next.

> Note: While working on the code I realized that parts of my redesign decisions -- teasing apart the concerns -- were close in mind to those of the LMAX Disruptor concurrent ring buffer, albeit much simpler of course. Firstly, to *limit concurrent access* to the relevant data structures (here: mostly what SlidingWindowCounter is being used for). In my case I followed the SRP and split the concerns into new data structures in a way that actually allowed me to eliminate the need for ANY concurrent access. Secondly, to put a *strict sequencing concept* in place (the way `incrementCount(T obj)` and `getCountsThenAdvanceWindow()` interact) that would prevent dirty reads or dirty writes from happening as was unfortunately possible in the old, system time based code.
>
> If you have not heard about LMAX Disruptor before, make sure to read their LMAX technical paper (PDF) on the LMAX homepage for inspirations. It's worth the time!

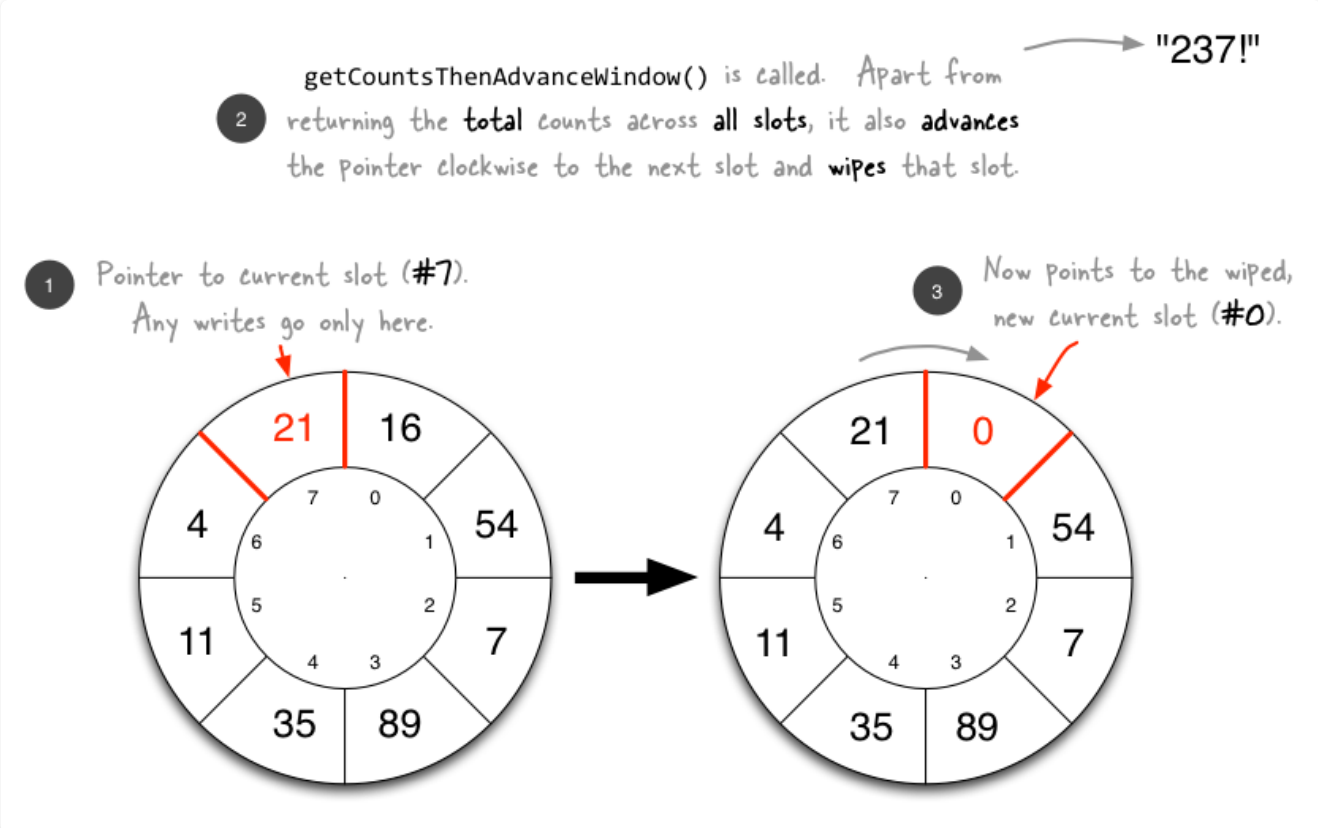

Figure 3: The `SlidingWindowCounter` class keeps track of multiple rolling counts of objects, i.e. a sliding window count for each tracked object. Please note that the example of an 8-slot sliding window counter above is simplified as it only shows a single count per slot. In reality `SlidingWindowCounter` tracks multiple counts for multiple objects.

Here is an illustration showing the behavior of `SlidingWindowCounter` over multiple iterations:

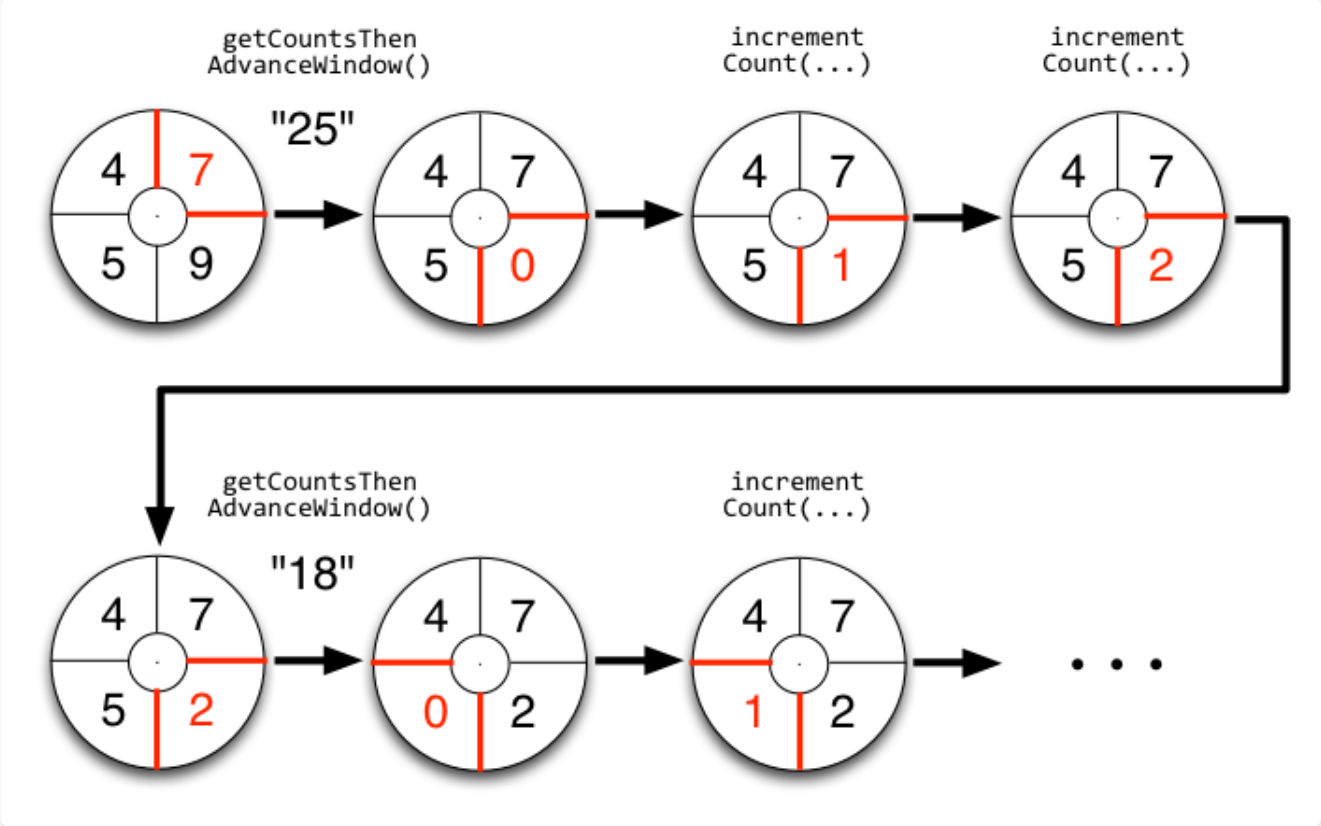

Figure 4: Example of `SlidingWindowCounter` behavior for a counter of size 4. Again, the example is simplified as it only shows a single count per slot.

Rankings and Rankable

The Rankings class represents fixed-size rankings of objects, for instance to implement "Top 10" rankings. It ranks its objects descendingly according to their **natural order**, i.e. from largest to smallest. This class is used by `AbstractRankerBolt` and its derived bolts to track the current rankings of incoming objects over time.

> Note: The `Rankings` class itself is completely unaware of the bolts' time-based behavior.

The class provides five public methods:

```
// Rankings API
public void updateWith(Rankable r);
public void updateWith(Rankings other);
public List<Rankable> getRankings();
public int maxSize(); // as supplied to constructor
public int size(); // current size, might be less than maximum size
```

Whenever you update `Rankings` with new data, it will discard any elements that are smaller than the updated top `N`, where `N` is the maximum size of the `Rankings` instance (e.g. `10` for a top 10 ranking).

Now the sorting aspect of the ranking is driven by the *natural order* of the ranked objects. In my specific case, I created a Rankable interface that in turn implements the Comparable interface. In practice, you simply pass a `Rankable` object to the `Rankings` class, and the latter will update its rankings accordingly.

```
// Using the Rankings class
Rankings topTen = new Rankings(10);
Rankable C = ...;
topTen.updateWith(r);

List<Rankable> rankings = topTen.getRankings();
```

As you can see it is really straight-forward and intuitive in its use.

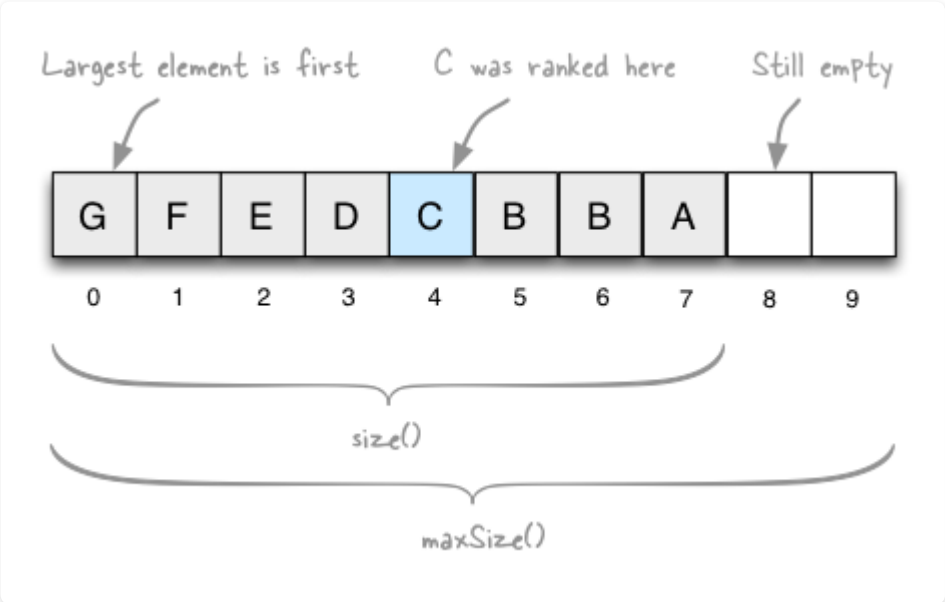

*Figure 5: The `Rankings` class ranks `Rankable` objects descendingly according to their **natural order**, i.e. from largest to smallest. The example above shows a Rankings instance with a maximum size of 10 and a current size of 8.*

The concrete class implementing `Rankable` is RankableObjectWithFields. The bolt `IntermediateRankingsBolt`, for instance, creates `Rankables` from incoming data tuples via a factory method of this class:

```
// IntermediateRankingsBolt.java
@Override
void updateRankingsWithTuple(Tuple tuple) {
  Rankable rankable = RankableObjectWithFields.from(tuple);
```

```
    super.getRankings().updateWith(rankable);
}
```

Have a look at Rankings, Rankable and RankableObjectWithFields for details. If you run into a situation where you have to implement classes like these yourself, make sure you follow good engineering practice and add standard methods such as `equals()` and `hashCode()` as well to your data structures.

Implementing the Rolling Top Words Topology

So where are we? In the sections above we have already discussed a number of Java classes but not even a single one of them has been directly related to Storm. It's about time that we start writing some Storm code!

In the following sections I will describe the Storm components that make up the Rolling Top Words topology. When reading the sections keep in mind that the "words" in this topology represent the topics that are currently being mentioned by the users in our imaginary system.

Overview of the Topology

The high-level view of the Rolling Top Words topology is shown in the figure below.

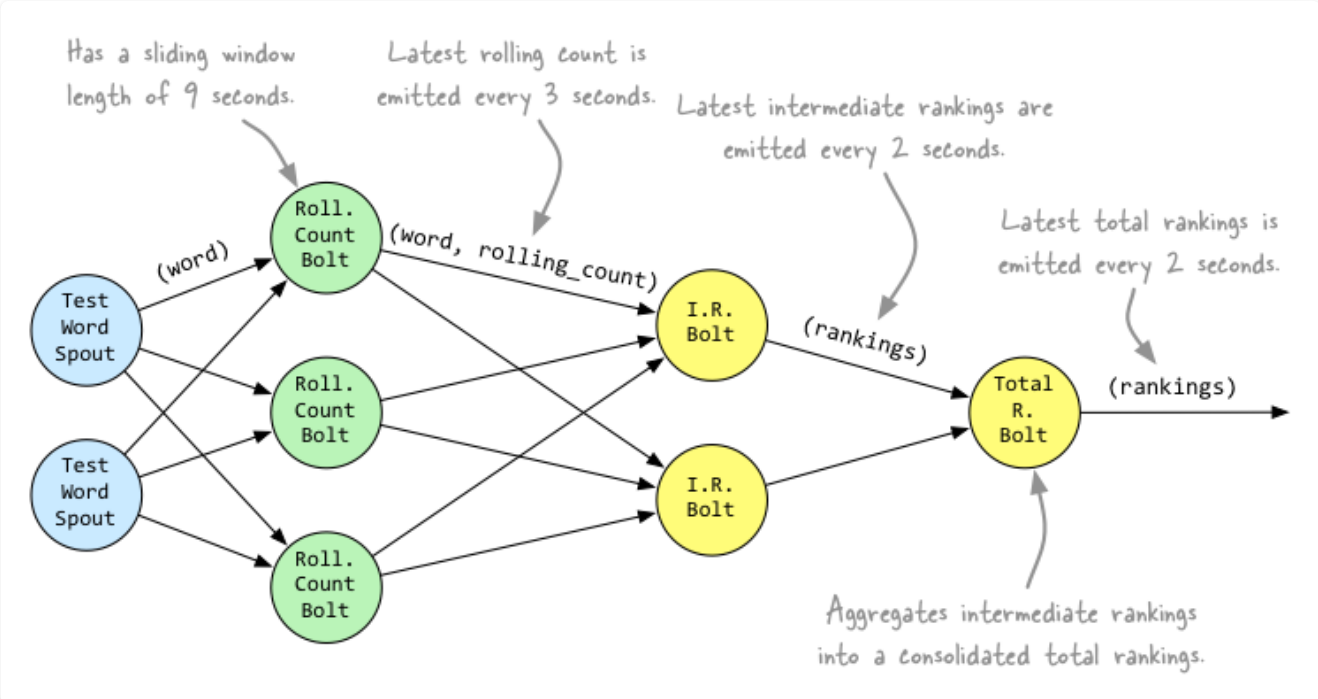

Figure 6: The Rolling Top Words topology consists of instances of `TestWordSpout`, `RollingCountBolt`, `IntermediateRankingsBolt` and `TotalRankingsBolt`. The length of the sliding window (in secs) as well as the various emit frequencies (in secs) are just example values -- depending on your use case you would, for instance, prefer to have a sliding window of five minutes and emit the latest rolling counts every minute.

The main responsibilities are split as follows:

1. In the first layer the topology runs many `TestWordSpout` instances in parallel to simulate the load of incoming data – in our case this would be the names of the topics (represented as words) that are currently being mentioned by our users.
2. The second layer comprises multiple instances of `RollingCountBolt`, which perform a rolling count of incoming words/topics.
3. The third layer uses multiple instances of `IntermediateRankingsBolt` ("I.R. Bolt" in the figure) to distribute the load of pre-aggregating the various incoming rolling counts into intermediate rankings. Hadoop users will see a strong similarity here to the functionality of a *combiner* in Hadoop.
4. Lastly, there is the final step in the topology. Here, a single instance of `TotalRankingsBolt` aggregates the incoming intermediate rankings into a global, consolidated total ranking. The output of this bolt are the currently trending topics in the system. These trending topics can then be used by downstream data consumers to provide all the cool user-facing and backend features you want to have in your platform.

In code the topology wiring looks as follows in RollingTopWords:

```
// RollingTopWords.java
builder.setSpout(spoutId, new TestWordSpout(), 2);
builder.setBolt(counterId, new RollingCountBolt(9, 3), 3)
            .fieldsGrouping(spoutId, new Fields("word"));
builder.setBolt(intermediateRankerId, new IntermediateRankingsBolt(TOP_N
            .fieldsGrouping(counterId, new Fields("obj"));
builder.setBolt(totalRankerId, new TotalRankingsBolt(TOP_N))
            .globalGrouping(intermediateRankerId);
```

> Note: The integer parameters of the `setSpout()` and `setBolt()` methods (do not confuse them with the integer parameters of the bolt constructors) configure the parallelism of the Storm components. See my article Understanding the Parallelism of a Storm Topology for details.

TestWordSpout

The only spout we will be using is the TestWordSpout that is part of `backtype.storm.testing` package of Storm itself. I will not cover the spout in detail because it is a trivial class. The only thing it does is to select a random word from a fixed list of five words ("nathan", "mike", "jackson", "golda", "bertels") and emit that word to the downstream topology every 100ms. For the sake of this article, we consider these words to be our "topics", of which we want to identify the trending ones.

> Note: Because `TestWordSpout` selects its output words at random (and each word having the same probability of being selected) in most cases the counts of the various words are pretty close to each other. This is ok for example code such as ours. In a production setting though you most likely want to generate "better" simulation data.

The spout's output can be visualized as follows. Note that the `@XXXms` milliseconds timeline is not part of the actual output.

```
@100ms: nathan
@200ms: golda
@300ms: golda
@400ms: jackson
@500ms: mike
@600ms: nathan
@700ms: bertels
...
```

Excursus: Tick Tuples in Storm 0.8+

A new and very helpful feature of Storm 0.8 is the so-called *tick tuple*. Whenever you want a spout or bolt execute a task at periodic intervals – in other words, you want to trigger an event or activity – using a tick tuple is normally the best practice.

Nathan Marz described tick tuples in the Storm 0.8 announcement as follows:

> *Tick tuples: It's common to require a bolt to "do something" at a fixed interval, like flush writes to a database. Many people have been using variants of a ClockSpout to send these ticks. The problem with a ClockSpout is that you can't internalize the need for ticks within your bolt, so if you forget to set up your bolt correctly within your topology it won't work correctly. 0.8.0 introduces a new "tick tuple" config that lets you specify the frequency at which you want to receive tick tuples via the "topology.tick.tuple.freq.secs" component-specific config, and then your bolt will receive a tuple from the `__system` component and `__tick` stream at that frequency.*
>
> ***Nathan Marz on the Storm mailing list** groups.google.com/forum/#!msg/…*

Here is how you configure a bolt/spout to receive tick tuples every 10 seconds:

```
// Configuring a bolt/spout to receive tick tuples every 10 seconds
@Override
public Map<String, Object> getComponentConfiguration() {
  Config conf = new Config();
  int tickFrequencyInSeconds = 10;
  conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, tickFrequencyInSeconds)
```

```
  return conf;
}
```

Usually you will want to add a conditional switch to the component's `execute` method to tell tick tuples and "normal" tuples apart:

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

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

Be aware that tick tuples are sent to bolts/spouts just like "regular" tuples, which means they will be queued behind other tuples that a bolt/spout is about to process via its `execute()` or `nextTuple()` method, respectively. As such the time interval you configure for tick tuples is, in practice, served on a "best effort" basis. For instance, if a bolt is suffering from high execution latency – e.g. due to being overwhelmed by the incoming rate of regular, non-tick tuples – then you will observe that the periodic activities implemented in the bolt will get triggered later than expected.

I hope that, like me, you can appreciate the elegance of solely using Storm's existing primitives to implement the new tick tuple feature. :-)

RollingCountBolt

This bolt performs rolling counts of incoming objects, i.e. sliding window based counting. Accordingly it uses the `SlidingWindowCounter` class described above to achieve this. In contrast to the old implementation only this bolt (more correctly: the instances of this bolt that run as Storm tasks) is interacting with the `SlidingWindowCounter` data structure. Each instance of the bolt has its own private `SlidingWindowCounter` field, which eliminates the need for any custom inter-thread communication and synchronization.

The bolt combines the previously described tick tuples (that trigger at fix intervals in time) with the time-agnostic behavior of `SlidingWindowCounter` to achieve time-based sliding window counting. Whenever the bolt receives a tick tuple, it will advance the window of its

private `SlidingWindowCounter` instance and emit its latest rolling counts. In the case of normal tuples it will simply count the object and ack the tuple.

```
// RollingCountBolt
@Override
public void execute(Tuple tuple) {
  if (TupleHelpers.isTickTuple(tuple)) {
    LOG.info("Received tick tuple, triggering emit of current window cou
    emitCurrentWindowCounts();
  }
  else {
    countObjAndAck(tuple);
  }
}

private void emitCurrentWindowCounts() {
  Map<Object, Long> counts = counter.getCountsThenAdvanceWindow();
  ...
  emit(counts, actualWindowLengthInSeconds);
}

private void emit(Map<Object, Long> counts) {
  for (Entry<Object, Long> entry : counts.entrySet()) {
    Object obj = entry.getKey();
    Long count = entry.getValue();
    collector.emit(new Values(obj, count));
  }
}

private void countObjAndAck(Tuple tuple) {
  Object obj = tuple.getValue(0);
  counter.incrementCount(obj);
  collector.ack(tuple);
}
```

That's all there is to it! The new tick tuples in Storm 0.8 and the cleaned code of the bolt and its collaborators also make the code much more testable (the new code of this bolt has 98% test coverage). Compare the code above to the old implementation of the bolt and decide for yourself which one you'd prefer adapting or maintaining:

```
// RollingCountObjects BEFORE Storm tick tuples and refactoring
public void prepare(Map stormConf, TopologyContext context, OutputCollec
    _collector = collector;
    cleaner = new Thread(new Runnable() {
        public void run() {
            Integer lastBucket = currentBucket(_numBuckets);
```

```
            while(true) {
              int currBucket = currentBucket(_numBuckets);
              if(currBucket!=lastBucket) {
                  int bucketToWipe = (currBucket + 1) % _numBuckets;
                  synchronized(_objectCounts) {
                      Set objs = new HashSet(_objectCounts.keySet());
                      for (Object obj: objs) {
                        long[] counts = _objectCounts.get(obj);
                        long currBucketVal = counts[bucketToWipe];
                        counts[bucketToWipe] = 0;
                        long total = totalObjects(obj);
                        if(currBucketVal!=0) {
                            _collector.emit(new Values(obj, total));
                        }
                        if(total==0) {
                            _objectCounts.remove(obj);
                        }
                      }
                  }
                  lastBucket = currBucket;
              }
              long delta = millisPerBucket(_numBuckets) - (System.curren
              Utils.sleep(delta);
            }
        }
    });
    cleaner.start();
}

public void execute(Tuple tuple) {
    Object obj = tuple.getValue(0);
    int bucket = currentBucket(_numBuckets);
    synchronized(_objectCounts) {
        long[] curr = _objectCounts.get(obj);
        if(curr==null) {
            curr = new long[_numBuckets];
            _objectCounts.put(obj, curr);
        }
        curr[bucket]++;
        _collector.emit(new Values(obj, totalObjects(obj)));
        _collector.ack(tuple);
    }
}
```

Unit Test Example

Since I mentioned unit testing a couple of times in the previous section, let me briefly discuss this point in further detail. I implemented the unit tests with TestNG, Mockito and FEST-Assert. Here is an example unit test for `RollingCountBolt`, taken from RollingCountBoltTest.

```
// Example unit test
@Test
public void shouldEmitNothingIfNoObjectHasBeenCountedYetAndTickTupleIsRe
  // given
  Tuple tickTuple = MockTupleHelpers.mockTickTuple();
  RollingCountBolt bolt = new RollingCountBolt();
  Map conf = mock(Map.class);
  TopologyContext context = mock(TopologyContext.class);
  OutputCollector collector = mock(OutputCollector.class);
  bolt.prepare(conf, context, collector);

  // when
  bolt.execute(tickTuple);

  // then
  verifyZeroInteractions(collector);
}
```

AbstractRankerBolt

This abstract bolt provides the basic behavior of bolts that rank objects according to their natural order. It uses the template method design pattern for its `execute()` method to allow actual bolt implementations to specify how incoming tuples are processed, i.e. how the objects embedded within those tuples are retrieved and counted.

This bolt has a private `Rankings` field to rank incoming tuples (those must contain `Rankable` objects, of course) according to their natural order.

```
// AbstractRankerBolt
// This method functions as a template method (design pattern).
@Override
public final void execute(Tuple tuple, BasicOutputCollector collector) {
  if (TupleHelpers.isTickTuple(tuple)) {
    getLogger().info("Received tick tuple, triggering emit of current ra
    emitRankings(collector);
  }
  else {
    updateRankingsWithTuple(tuple);
  }
}
```

```
abstract void updateRankingsWithTuple(Tuple tuple);
```

The two actual implementations used in the Rolling Top Words topology, `IntermediateRankingsBolt` and `TotalRankingsBolt`, only need to implement the `updateRankingsWithTuple()` method.

IntermediateRankingsBolt

This bolt extends `AbstractRankerBolt` and ranks incoming objects by their count in order to produce intermediate rankings. This type of aggregation is similar to the functionality of a *combiner* in Hadoop. The topology runs many of such intermediate ranking bolts in parallel to distribute the load of processing the incoming rolling counts from the `RollingCountBolt` instances.

This bolt only needs to override `updateRankingsWithTuple()` of `AbstractRankerBolt`:

```
// IntermediateRankingsBolt
@Override
void updateRankingsWithTuple(Tuple tuple) {
  Rankable rankable = RankableObjectWithFields.from(tuple);
  super.getRankings().updateWith(rankable);
}
```

TotalRankingsBolt

This bolt extends `AbstractRankerBolt` and merges incoming intermediate `Rankings` emitted by the `IntermediateRankingsBolt` instances.

Like `IntermediateRankingsBolt`, this bolt only needs to override the `updateRankingsWithTuple()` method:

```
// TotalRankingsBolt
@Override
void updateRankingsWithTuple(Tuple tuple) {
  Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
  super.getRankings().updateWith(rankingsToBeMerged);
}
```

Since this bolt is responsible for creating a global, consolidated ranking of currently trending topics, the topology must run only a single instance of `TotalRankingsBolt`. In other words, it must be a singleton in the topology.

The bolt's current code in `storm-starter` does not enforce this behavior though – instead it relies on the `RollingTopWords` class to configure the bolt's parallelism correctly (if you ask yourself why it doesn't: that was simply oversight on my part, oops). If you want to improve that, you can provide a so-called *per-component* Storm configuration for this bolt that sets its maximum task parallelism to 1:

```
// TotalRankingsBolt
@Override
public Map<String, Object> getComponentConfiguration() {
  Map<String, Object> conf = new HashMap<String, Object>();
  conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, emitFrequencyInSeconds);
  // run only a single instance of this bolt in the Storm topology
  conf.setMaxTaskParallelism(1);
  return conf;
}
```

RollingTopWords

The class RollingTopWords ties all the previously discussed code pieces together. It implements the actual Storm topology, configures spouts and bolts, wires them together and launches the topology in local mode (Storm's local mode is similar to a pseudo-distributed, single-node Hadoop cluster).

By default, it will produce the top 5 rolling words (our trending topics) and run for one minute before terminating. If you want to twiddle with the topology's configuration settings, here are the most important:

- Configure the number of generated trending topics by setting the `TOP_N` constant in `RollingTopWords`.
- Configure the length and emit frequencies (both in seconds) for the sliding window counting in the constructor of `RollingCountBolt` in `RollingTopWords#wireTopology()`.
- Similarly, configure the emit frequencies (in seconds) of the ranking bolts by using their corresponding constructors.
- Configure the parallelism of the topology by setting the `parallelism_hint` parameter of each bolt and spout accordingly.

Apart from this there is nothing special about this class. And because we have already seen the most important code snippet from this class in the section *Overview of the Topology* I will not describe it any further here.

Running the Rolling Top Words topology

Update 2014-06-04: I updated the instructions below based on the latest version of storm-starter, which is now part of the official Storm project.

Now that you know how the trending topics Storm code works it is about time we actually launch the topology! The topology is configured to run in local mode, which means you can just grab the code to your development box and launch it right away. You do not need any special Storm cluster installation or similar setup.

First you must checkout the latest code of the storm-starter project from GitHub:

```
$ git clone git@github.com:apache/incubator-storm.git
$ cd incubator-storm
```

Then you must build and install the (latest) Storm jars locally, see the storm-starter README:

```
# Must be run from the top-level directory of the Storm code repository
$ mvn clean install -DskipTests=true
```

Now you can compile and run the RollingTopWords topology:

```
$ cd examples/storm-starter
$ mvn compile exec:java -Dstorm.topology=storm.starter.RollingTopWords
```

By default the topology will run for one minute and then terminate automatically.

Example Logging Output

Here is some example logging output of the topology. The first colum is the current time in milliseconds since the topology was started (i.e. it is `0` at the very beginning). The second colum is the ID of the thread that logged the message. I deliberately removed some entries in the log flow to make the output easier to read. For this reason please take a close look on the timestamps (first column) when you want to compare the various example outputs below.

Also, the Rolling Top Words topology has debugging output enabled. This means that Storm itself will by default log information such as what data a bolt/spout has emitted. For that reason you will see seemingly duplicate lines in the logs below.

Lastly, to make the logging output easier to read here is some information about the various thread IDs in this example run:

Thread ID	Java Class
Thread-37	TestWordSpout
Thread-39	TestWordSpout
Thread-19	RollingCountBolt

Thread ID	Java Class
Thread-21	RollingCountBolt
Thread-25	RollingCountBolt
Thread-31	IntermediateRankingsBolt
Thread-33	IntermediateRankingsBolt
Thread-27	TotalRankingsBolt

Note: The Rolling Top Words code in the `storm-starter` repository runs more instances of the various spouts and bolts than the code used in this article. I downscaled the settings only to make the figures etc. easier to read. This means your own logging output will look slightly different.

The topology has just started to run. The spouts generate their first output messages:

```
2056 [Thread-37] INFO  backtype.storm.daemon.task  - Emitting: wordGener
2057 [Thread-19] INFO  backtype.storm.daemon.executor  - Processing rece
2063 [Thread-39] INFO  backtype.storm.daemon.task  - Emitting: wordGener
2064 [Thread-25] INFO  backtype.storm.daemon.executor  - Processing rece
2069 [Thread-37] INFO  backtype.storm.daemon.task  - Emitting: wordGener
2069 [Thread-21] INFO  backtype.storm.daemon.executor  - Processing rece
```

The three RollingCountBolt instances start to emit their first sliding window counts:

```
4765 [Thread-19] INFO  backtype.storm.daemon.executor  - Processing rece
4765 [Thread-19] INFO  storm.starter.bolt.RollingCountBolt  - Received t
4765 [Thread-25] INFO  backtype.storm.daemon.executor  - Processing rece
4765 [Thread-25] INFO  storm.starter.bolt.RollingCountBolt  - Received t
4766 [Thread-21] INFO  backtype.storm.daemon.executor  - Processing rece
4766 [Thread-21] INFO  storm.starter.bolt.RollingCountBolt  - Received t
4766 [Thread-19] INFO  backtype.storm.daemon.task  - Emitting: counter d
4766 [Thread-25] INFO  backtype.storm.daemon.task  - Emitting: counter d
4766 [Thread-21] INFO  backtype.storm.daemon.task  - Emitting: counter d
```

The two `IntermediateRankingsBolt` instances emit their intermediate rankings:

```
5774 [Thread-31] INFO  backtype.storm.daemon.task  - Emitting: intermedi
5774 [Thread-33] INFO  backtype.storm.daemon.task  - Emitting: intermedi
5774 [Thread-31] INFO  storm.starter.bolt.IntermediateRankingsBolt  - Ra
5774 [Thread-33] INFO  storm.starter.bolt.IntermediateRankingsBolt  - Ra
```

The single `TotalRankingsBolt` instance emits its global rankings:

```
3765 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings:
5767 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings:
7768 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings:
9770 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings:
11771 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings
13772 [Thread-27] INFO  storm.starter.bolt.TotalRankingsBolt  - Rankings
```

> Note: During the first few seconds after startup you will observe that `IntermediateRankingsBolt` and `TotalRankingsBolt` instances will emit empty rankings. This is normal and the expected behavior -- during the first seconds the `RollingCountBolt` instances will collect incoming words/topics and fill their sliding windows before emitting the first rolling counts to the `IntermediateRankingsBolt` instances. The same kind of thing happens for the combination of `IntermediateBolt` instances and the `TotalRankingsBolt` instance. This is an important behavior of the code that must be understood by downstream data consumers of the trending topics emitted by the topology.

What I Did Not Cover

I introduced a new feature to the Rolling Top Words code that I contributed back to `storm-starter`. This feature is a metric that tracks the difference between the configured length of the sliding window (in seconds) and the actual window length as seen in the emitted output data.

```
4763 [Thread-25] WARN  storm.starter.bolt.RollingCountBolt  - Actual win
```

This metric provides downstream data consumers with additional meta data, namely the time range that a data tuple actually covers. It is a nifty addition that will make the life of your fellow data scientists easier. Typically, you will see a difference between configured and actual window length a) during startup for the reasons mentioned above and b) when your machines are under high load and therefore not respond perfectly in time. I omitted the discussion of this new feature to prevent this article from getting too long.

Also, there are some minor changes in my own code that I did not contribute back to `storm-starter` because I did not want to introduce too many changes at once (such as a refactored `TestWordSpout` class).

Summary

In this article I described how to implement a distributed, real-time trending topics algorithm in Storm. It uses the latest features available in Storm 0.8 (namely tick tuples) and should be a good starting point for anyone trying to implement such an algorithm for their own application. The new code is now available in the official storm-starter repository, so feel free to take a closer look.

The sliding window analysis described here applies to a broader range of problems than computing trending topics.

You might ask whether there is a use of a distributed sliding window analysis beyond the use case I presented in this article. And for sure there is. The sliding window analysis described here applies to a broader range of problems than computing trending topics. Another typical area of application is real-time infrastructure monitoring, for instance to identify broken servers by detecting a surge of errors originating from problematic machines. A similar use case is identifying attacks against your technical infrastructure, notably flood-type DDoS attacks. All of these scenarios can benefit from sliding window analyses of incoming real-time data through tools such as Storm.

If you think the starter code can be improved further, please contribute your changes back to the storm-starter component in the official Storm project.

Related Links

- Understanding the Parallelism of a Storm Topology

Interested in more? You can subscribe to this blog and follow me on Twitter.

Michael G. Noll

I am a software engineer turned product manager based in Switzerland. In my day job I am working on stream processing products at Confluent, the company founded by the creators of Apache Kafka. Opinions my own. Read more »

michael@michael-noll.com
miguno
miguno