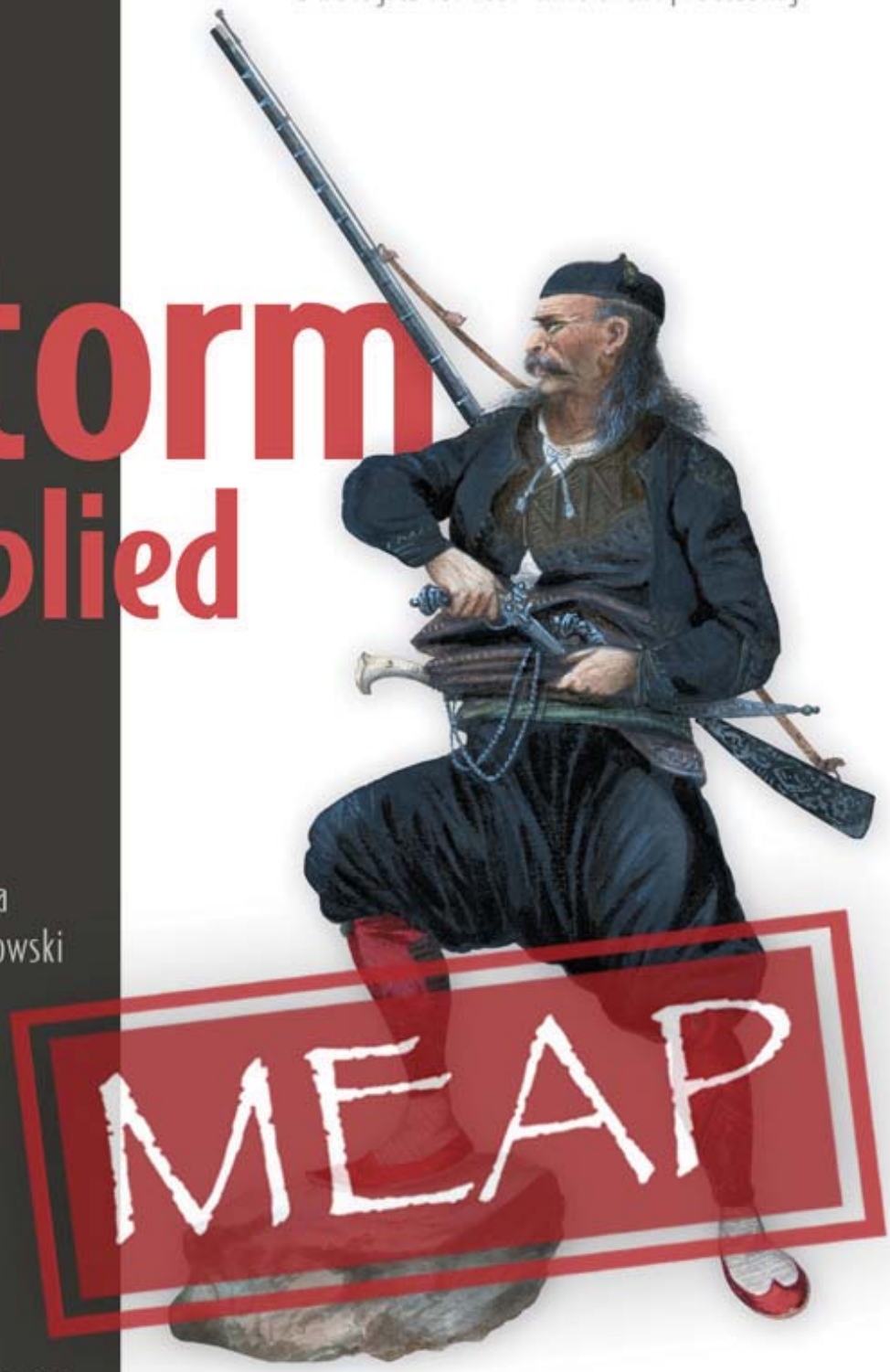
Strategies for real-time event processing

Storm Applied

Sean T. Allen
Peter Pathirana
Matthew Jankowski



MEAP

MANNING

MEAP Edition
Manning Early Access Program
Storm Applied
Version 3

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *Storm Applied*. We can hardly believe this is happening. Just a few months ago, the idea of writing a book wasn't on our radar. When Manning approached us, we were initially hesitant. A few months later, we are excited that the initial product of our hard work is now in your hands. *Storm Applied* is an advanced book however, it recognizes an important reality about the current Storm landscape. There is a limited amount of information currently out for getting started with Storm, so our first section attempts to do as gentle an introduction to Storm and its concepts as we could muster. We believe we have struck the right balance, we hope you do as well.

In *Storm Applied*, we are trying to accomplish three things:

1. Teach you about Storm and its concepts and terminology
2. Teach you how to map problems from domains you might be familiar with onto Storm.
3. Teach you the in's and out's of maintaining Storm topologies in a production environment

We believe that while the first goal might be what initially draws you in, the second and third are what is going to keep you coming back. We've been running Storm in production for quite a while now and learning how to map our problems onto Storm was the hardest part of the journey. Hopefully, by the time you finish *Storm Applied*, you'll have a leg up.

What you have in your hands now is section one of the book. In it, we endeavor to introduce the primary concepts from Storm and start to get you thinking in Storm.

Each chapter in the book is built around a use case. The use cases serve to guide you in thinking about problems and mapping them to Storm while teaching you important concepts and techniques.

Chapter 2 introduces the most basic Storm concepts through a use case that almost every programmer should be familiar with: analyzing commits to a version control repository.

In Chapter 3, we start to introduce some best practice design patterns to follow when working with Storm. Additionally, we build on the concepts in chapter 1 and introduce the basic parallelism primitives that Storm provides.

At its core, Chapter 4 introduces once of Storm's most important features: guaranteed at least once message processing. It also builds on chapter 3's initial foray into parallelism and digs deeper by moving from running in a local test mode to running on a production cluster and introduces the various moving parts of a production Storm system.

Looking ahead, section 2 will introduce additional use cases and use them to explore production issues with Storm such as debugging & managing resource contention. Section 3 will dive into Trident and the high level abstractions over Storm that it affords.

We hope that you will join us in the Manning Author's Forum and help us improve *Storm Applied* with your feedback.

All the best,

—Matt, Peter & Sean

brief contents

1

Introducing Storm

This chapter covers

- Answering the question "What is Storm?"
- How Storm fits into the big data landscape
- Answering the question "Why would I want to use Storm?"
- How we are going to teach you Storm

1.1 What is Storm

Storm is a distributed real-time computational framework that makes processing unbounded streams of data ("big data") easy. Storm can be integrated with your existing queuing and persistence technologies, consuming streams of data and processing/transforming these streams in many different ways.

Still with us? Some of you are probably feeling smart because you know what that means. Others are searching for the proper animated gif to express your level of frustration. There's a lot in that description, so if you don't understand what all of that means right now, don't worry. We have devoted the remainder of this chapter to clarifying exactly what we mean, starting with explaining where Storm falls within the big data landscape.

1.2 Storm Within the Big Data Landscape

In order to appreciate what Storm is and when it should be used, it is important to understand where Storm falls within the big data landscape. What technologies can it be used with? What technologies can it replace? Being able to answer questions like these requires some context. Let's get started by talking about "big data" in all its various shapes and guises.

1.2.1 What is Big Data?

It's easiest to comprehend big data by considering its three main properties: velocity, volume and variety. Understanding these properties will help you classify Storm as a big data tool along with understanding where exactly it fits within the variety of big data tools available. We start with explaining "volume" as that is what most people think of when they say "big data".

VOLUME

Volume is the most obvious property of big data, and the first that comes to most people's mind when they hear the term "big data". Data is constantly being generated every day from a multitude of sources: data generated by people via social media, data generated by software itself (website tracking, application logs, etc.) and data generated to be publicly available on the web, such as Wikipedia, only scratch the surface of sources of data.

When people think volume, companies such as Google, Facebook and Twitter come to mind. Sure, all deal with enormous amounts of data and I'm sure you can name others, but what about our employer: TheLadders? By the definition of volume alone, we don't have big data, yet we leverage Storm. Why?

> **WHAT IS THELADDERS?** TheLadders (www.theladders.com) is a job-matching service for career-driven professionals. This involves helping professionals find the jobs and meet the employers that are right for their career goals.

VELOCITY

Velocity deals with the pace at which data flows into a system, both in terms of amount of data and the fact that it's a continuous flow of data. The amount of data (maybe just a series of links on your website that a visitor is clicking on) might be relatively small, but the rate it is flowing into your system could be rather high. Velocity matters. It doesn't matter how much data you have if you aren't processing it fast enough to provide value. It could be a couple terabytes; it could be 5 million URLs making up a much smaller volume of data. All that matters is can I extract meaning from this data before it goes stale.

VARIETY

What about Variety? Well, that's a special case. Let's step back and look at extracting meaning from data. Often, that can mean taking data from multiple different sources and putting them together into something that tells a story. When you start though, you might have some data over here in Google Analytics, maybe some in an append-only log, and perhaps some more over in a relational database. You need to bring all of these together and shape them into something you can work with to drill down to extract meaningful answers from questions such as:

Q: *"Who are my best customers?"*
A: *Coyotes in New Mexico.*

Q: *"What do they usual purchase?"*
A: *Some paint but mostly large heavy items.*

Q: *"Can I look at each of these customers individually and find items others have liked and market it to them?"*
A: *That depends on how quickly you can turn your variety of data into something you can use and operate on.*

So now that we have established some boundaries around big data, we will touch upon the different types of tools used with big data. With this knowledge in hand, it will start to make sense where Storm fits within all of this.

1.2.2 Big Data Tools

There are a variety of tools designed to address different parts of the Volume, Velocity and Variety problems. We are running under the assumption that most companies have some sort of messaging system, such as Kafka, along with some sort of data store, such as a NoSQL or relational database. The tools we are more interested in are two classes of tools that have established themselves as valuable big data tools: batch-oriented and stream processing. To understand where Storm fits within the big data landscape, it's important to understand the difference between the two.

BATCH-ORIENTED

A batch-oriented tool is one that ingests a large batch of data, such as a day's worth of website click data, and processes it in some fashion. Figure 1.1 provides an overview how data flows into a batch-oriented tool.

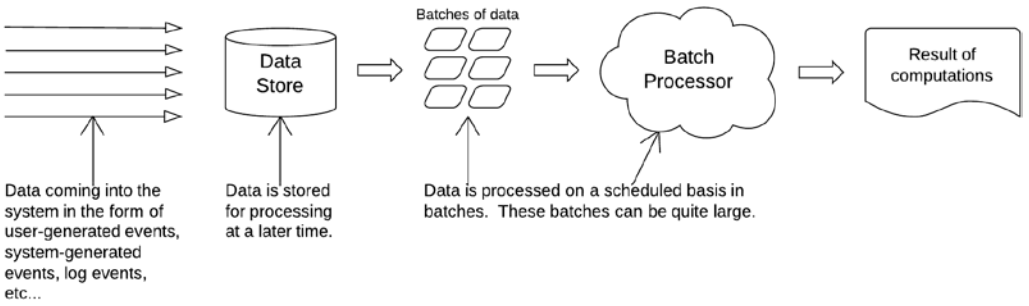


Figure 1.1 Batch-processor & how data flows into it

Perhaps it's finding top URLs or top URLs by visitors as broken down by geography. Our batches might be much smaller. Maybe you are an auction site, like EBay, and you have a search index you need to keep up to date: all auctions on site. You could use a batch-oriented

process to load all the new auctions from the last ten minutes and all those that have closed up into memory to update the index customers use to find what's available on your site.

STREAM PROCESSING

On the other end is stream processing. Stream processing can be broken down into a couple of classes:

- An unbounded stream of data
- A stream of data within the context of a batch.

Figure 1.2 provides an overview of how data flows into a stream processing system.

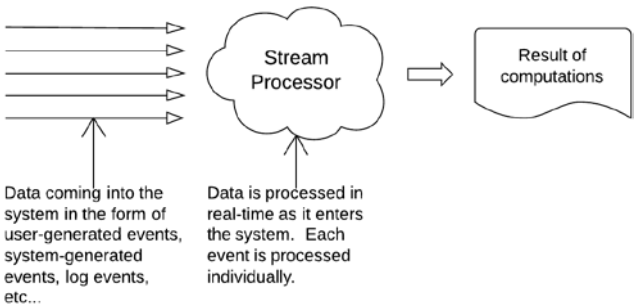


Figure 1.2 Stream-processor & how data flows into it

Let's start with the unbounded stream of data, which is what we see in Figure 1.2. We are continually ingesting new data (a "stream") that we must do some sort of analysis on. One of the most well known examples of this is Twitter's Trending Topics. The amount of data in the form of tweets flowing through Twitter's system is immense and they need to be able to tell users what everyone in their area is talking about right now.

Think about that one for a moment. Not only do they have the requirement to operate at high volume, but they also need to operate with high velocity. Twitter has a massive, never-ending stream of tweets coming in and they need to be able to extract, in real-time, what people are talking about. That is a serious feat of engineering. In fact, Chapter 3 is built around a use case that is similar to this idea.

You can also use stream processing in more of a batch-oriented fashion. If you have a large amount of data that you need to process at one time but the processing of any individual item doesn't depend on the processing of any other, you are effectively doing stream processing over a batch. Let's go back to our employer again, TheLadders. We allow job seekers on our site to save searches, which can be automatically run. At one time, we did this as a single application on a single machine that created a thread pool and a queue of searches to run. Each thread would grab a search from the queue, run it and store the results for emailing to the job seeker. That's a batch job. But none of the results matter at all in aggregate. We don't care for this use case: how often a job appears in the results. We just

want new jobs that match the job seekers' searches so we can send a corresponding email with the results. You can think of each search as an independent "stream of data" flowing through the search process. In Chapter 5, we'll cover a use case that aligns with this general idea and discuss more in depth why you might want to use Storm for this sort of problem.

1.2.3 How Storm Fits Into the Picture

So where does Storm fit within all of this? Going back to our original definition, we said:

> Storm is a distributed real-time computational framework that makes processing unbounded streams of data easy.

Storm is a stream-processing tool, plain and simple. Storm will run indefinitely listening to a stream of data, doing "something" any time it receives data from the stream. Storm is also a distributed system, allowing additional machines to be easily added in order to process as much data in "real-time" as we can.

What is real-time?

When we use the term *real-time* throughout this book, what exactly do we mean? Well, technically speaking, *near real-time* is more accurate. In software systems, real-time constraints are defined to set operational deadlines for how long it takes a system to respond to a particular event. Normally, this latency is along the order of milliseconds (or at least sub-second level), with no perceivable delay to the end-user. Within the context of Storm, both real-time (sub-second level) and near real-time (matter of seconds or few minutes depending on use case) latencies are quite possible.

And what about the second sentence in our initial definition?

> Storm can be integrated with your existing queuing and persistence technologies, consuming streams of data and processing/transforming these streams in many different ways.

As we will show you throughout the book, Storm is extremely flexible in that the source of a stream can be anything: usually this means a queuing system but Storm doesn't put any limits on where your stream comes from (we will be using Kafka & RabbitMQ for several of our use cases). Same thing goes for the result of a stream transformation produced by Storm. We've seen many cases where the result is persisted to a database somewhere for later

access. However, the result may also be pushed onto a separate queue for another system (maybe even another Storm topology) to process.

The point here is that Storm is simply a piece that can be plugged in to your existing architecture. The chapters throughout the book will give use cases of how exactly this can be done. We'll give you a little sneak peak, however, with a hypothetical architecture that involves Storm. Figure 1.3 shows this hypothetical scenario for analyzing a stream of tweets.

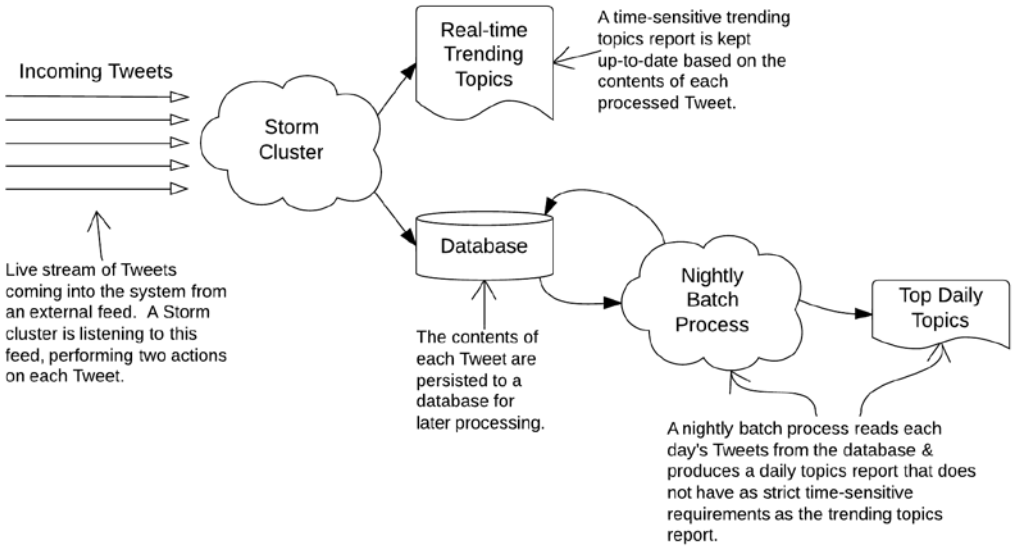


Figure 1.3 Example of how Storm may be used within a system

This high-level hypothetical solution is exactly just that: hypothetical. It was put here to show you where Storm could fall within a system. It also shows how the co-existence of batch and stream-processing tools is possible.

What about the different technologies that can be used with Storm? We give you Figure 1.2 to hopefully shed some light on this question. Figure 1.4 shows you a small sampling of some of the technologies that may be used in this architecture. The point of this diagram is to illustrate just how flexible Storm is in terms of a) the technologies it can be worked with and b) where it can be plugged into a system.

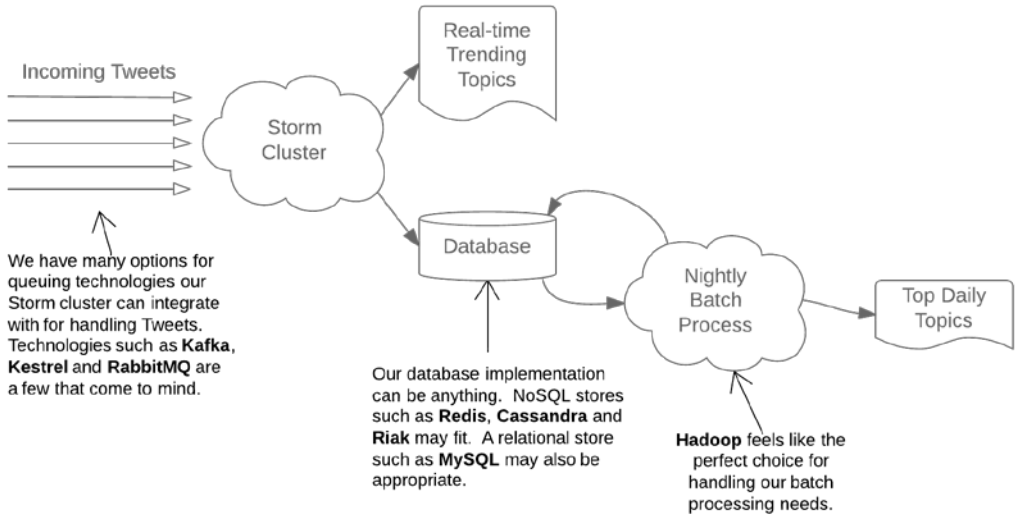


Figure 1.4 Example of how Storm may be used with other technologies

For our queuing system, we could choose from a number of technologies, including Kafka, Kestrel and RabbitMQ. Same thing goes for our database choice: Redis, Cassandra, Riak and MySQL only scratch the surface in terms of options. And look at that, we've even managed to include a Hadoop cluster in our solution for performing the required batch computation for our "Top Daily Topics" report.

Hopefully you're starting to gain a clearer understanding of where Storm fits and what it can be used with. A wide range of technologies can work with Storm within a system, including Hadoop. Wait, did we just say: "Storm can work with Hadoop?"

1.2.4 Storm Versus Hadoop

In a lot of conversations we've had with other engineers, Storm and Hadoop often come up in the same sentence. Part of this is probably because the original description of Storm on the Storm wiki mentioned Hadoop:

> Storm makes it easy to write and scale complex realtime computations on a cluster of computers, doing for realtime processing what Hadoop did for batch processing…

In addition, Hadoop is one of the prominent big data processing tools and, as a result, is oftentimes the standard by which other tools are measured.

Whatever the reasons are for mentioning both in the same breath, we feel there may be misconceptions when it comes to using Storm and Hadoop. Many people feel they should use

either Storm or Hadoop, when the truth is they are different tools meant for different purposes. They may actually be used together given the right circumstances.

HADOOP

Remember earlier when we talked about batch-oriented systems? That's Hadoop. Figure 1.5 gives you a refresher of how data would be fed into Hadoop for batch processing.

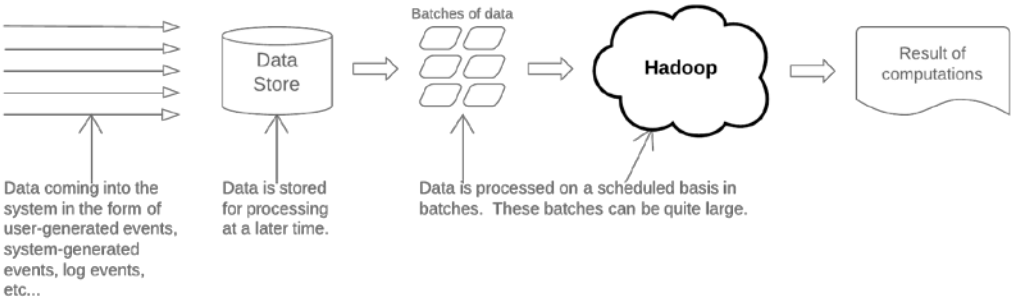


Figure 1.5 Hadoop & how data flows into it

Load a bunch of data into a Hadoop job and process it. In particular, Hadoop is a "MapReduce" system. We take a chunk of data and run it through two steps: 1) map and then 2) reduce.

MapReduce is a pretty flexible paradigm and you can find a way to make a large number of problems map on it (apologies for the pun). Hadoop is a tool that takes a batch of data in as input and runs one or more MapReduce jobs on it to produce a batch of data. Most jobs people consider using Hadoop for will take quite a while to complete. There's a component in a Hadoop cluster called the Job Tracker that is a single point of failure; lose the Job Tracker and lose your work. One of the primary design goals in Storm was to avoid that single point of failure. You should be able to (and can) lose any part of a Storm cluster and continue processing. If you know anyone who has worked with Hadoop, ask them if they've ever written a job, perhaps in Pig, and had it run for hours only to die somewhere near the end and they lose all their data. We're pretty sure you won't have to look too far to find someone you know who has experienced that.

Now don't get us wrong, Hadoop is a great tool for batch processing over an entire data set, but it isn't the only tool out there. There are other tools with different strengths meant for handling other types of problems, some of which may even be used in conjunction with Hadoop.

STORM

Storm, being a general framework for doing "real-time" computation, allows you to run incremental functions over data in a fashion that Hadoop can't. Figure 1.6 gives you a refresher on how data would be fed into Storm.

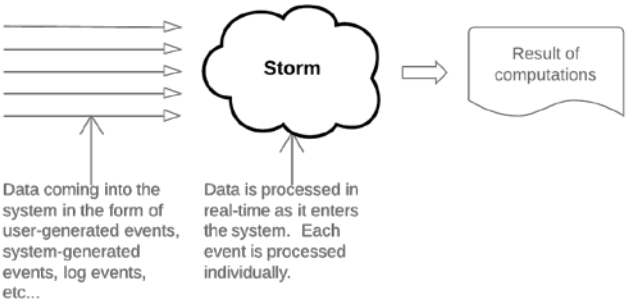


Figure 1.6 Storm & how data flows into it

As we alluded to earlier, there is a tradeoff in choosing Storm over Hadoop. You can't easily run over the entire data set at one time. Given enough batch size, you can approximate a lot of what Storm does with Hadoop and given a large enough chunk of memory, you can approximate a giant Hadoop batch size with Storm. There are reasons you might try. Perhaps you are a small shop and don't want or cannot support both a Storm and a Hadoop cluster. There is nothing out there that says you can't use them together. In fact, using a batch oriented system in conjunction with a stream oriented one is the subject of Big Data (ISBN #9781617290343) by Nathan Marz, the original author of Storm.

1.3 Why You Would Want to Use Storm

Now that we have explained where Storm fits within the big data landscape, let's give you some reasons for why you would want to use Storm. As we will demonstrate throughout the book, there are some fundamental properties that make Storm an attractive option. They include the following:

- Can be applied to a wide variety of use cases.
- Works well with a multitude of technologies.
- Is scalable. Storm makes it very easy to break down work over a series of threads over a series of JVMs over a series of machines. All of this without having to change your code to scale in that fashion, only changing some configuration.
- Guarantees to process every piece of input you give it at least once.
- Is very robust; one might even call it fault-tolerant. There are four major components within Storm and at various times, we've had to kill off any of the four while continuing to process data.
- Is programming language agnostic. If you can run it on the JVM, you can run it easily on Storm. Even if you can't run it on the JVM, if you can call it from a *nix command line, you can probably use it with Storm (although in this book, we are going to confine ourselves to the JVM and specifically Java).

Hopefully, that sounds impressive. We meant it to be. Storm has become our go to toolkit not just for scaling, but also for fault tolerance and guaranteed message processing. We have a variety of Storm topologies (a chunk of Storm code that performs a given task) that could easily run as a Python script on a single machine. However, if that script crashes, it does not compare to Storm in terms of recoverability; Storm will restart and pick up work from our point-of-crash. No 3 AM pager-duty alerts, no 9 AM explanations to VP of Engineering why something died. One of the great things about Storm is you can come for the fault tolerance and stay for the easy scaling.

1.4 How We Are Going to Teach You Storm

We've broken Storm Applied down into a series of use cases. Each one is designed to introduce a particular Storm concept and provide a simplified example of a real use case. Each chapter introduces new Storm concepts and a use case that demonstrates those concepts. With each use case, we aim to give you a way to think about Storm (to think "in" Storm if you will). Consider each chapter to be the addition of another tool to your kit for working with Storm.

The first section of the book is as gentle an introduction as we could give to the most important concepts. By the time you are done with it, you should understand the fundamental concepts that make Storm tick and should be well positioned to start writing your own Storm topologies with confidence.

The second section of the book targets issues you're going to hit when you go to production. How to tune and troubleshoot Storm; the hard stuff. Again, we will approach it as we did the first section: provide you the tools to do it on your own. Teach you reusable tuning and troubleshooting concepts that you can apply to your own unique use case.

And the third and final section? We'll get you going with Trident. Trident is an abstraction layer over the top of Storm that provides you with the ability to handle data in batches and guarantee exactly-once processing. Right now, that might not mean much to you but by the time you get through the third section, you'll understand why Trident will be a valuable tool in your kit. We hope you enjoy the ride.

1.5 Conclusion

In this chapter you've learned the following key facts:

- Storm is a stream-processing tool that can work with a myriad of technologies and for a multitude of use cases.
- Some of the benefits of Storm include its scalability, ability to process each message at least once, its robustness and its ability to be developed with any programming language.
- The three key properties of big data: volume, velocity and variety.
- The difference between batch and stream processing big data tools.

You now have a high-level understanding of what Storm is and how it may be used. Up next is establishing an understanding of the core concepts in Storm. This understanding will serve as the foundation for everything else we discuss in this book.

2

Storm Concepts

This chapter covers

- Storm core concepts and terminology
- Basic code for our first Storm project

The core concepts in Storm are simple once you understand them, but this understanding can be hard to come by. Wrapping your head around them all at once can be very difficult. Encountering a description of "executors and tasks" on your first day can be hard to grok. There were just too many concepts we needed to hold in our heads at one time. As we go forward in *Storm Applied*, we will introduce concepts in a progressive fashion; trying to minimize the number of concepts you need to think about at one time. This will often mean that an explanation isn't entirely "true", but it will be accurate enough at that point in your journey through Storm Applied. As you slowly pick up on different pieces of the puzzle, we will point out where our earlier definitions can be expanded on.

2.1 Problem definition: GitHub Commit Count Dashboard

We would start teaching you about Storm by doing work in a domain that should be familiar: source control and in particular, GitHub. Most developers are familiar with GitHub, having used it for either a personal project, for work or for interacting with other open source projects.

STORM SOURCE Hosted on GitHub at https://github.com/apache/incubator-storm

Let's say we want to implement a dashboard that shows a running count of the most active developers against any repository. This count has some real-time requirements in that it must be updated immediately after any change is made to the repository. The dashboard requested by GitHub may look something like Figure 2.1.

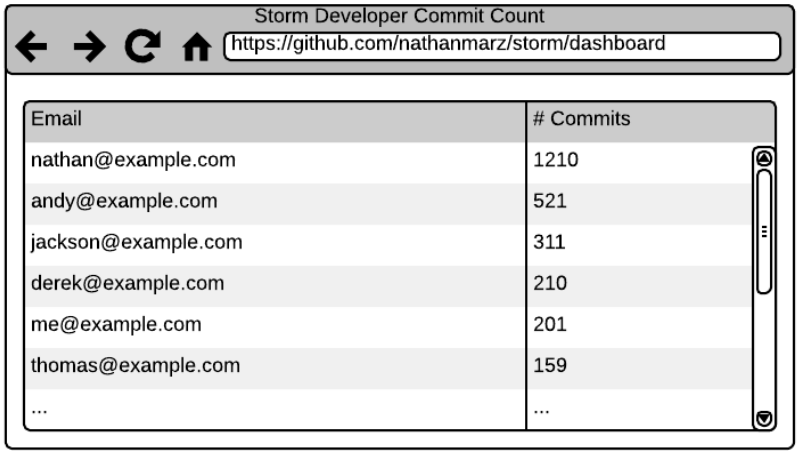


Figure 2.1 Mock-up of dashboard for running count of changes made to a repository

The dashboard is quite simple. It contains a listing of the email of every developer that has made a commit to the repository along with a running total of the number of commits they have made. Before we dive into how we would design a solution with Storm, let's break down the problem a little bit further in terms of the data that will be used.

2.1.1 Data – starting and ending points

GitHub provides a live feed of commits being made to any repository. This feed is a simple list of space-separated strings containing the commit ID followed by the email of the developer that made the commit. The following listing is a sample of ten commits in the feed.

STARTING POINT

```
b20ea50 nathan@example.com
064874b andy@example.com
28e4f8e andy@example.com
9a3e07f andy@example.com
cbb9cd1 nathan@example.com
0f663d2 jackson@example.com
0a4b984 nathan@example.com
1915ca4 derek@example.com
```

So we have a starting point. We have a live feed of commits containing a commit ID with an associated email.

ENDING POINTS

We will somehow need to go from this live feed to a UI containing a running count of commits per email address. For sake of simplicity, let's say all we need to do is maintain an in-memory map with email address as the key and number of commits as the value. The map may look something like the following in code:

```
Map<String, Integer> countsByEmail = new HashMap<String, Integer>();
```

Now that we have defined the data involved with our problem, let's break down the steps we need to take to make sure our in-memory map correctly reflects the commit data.

2.1.2 Breaking down the problem into a series of steps

Let's break down how we want to tackle this problem into a series of steps.

1. A component that reads from the live feed of commits and passes along a single commit to our next component.
2. A component that accepts a single commit, extracts the developer's email from that commit, and passes along the email to the next component.
3. A component that accepts the developer's email and updates an in-memory map containing emails and the number of commits for those emails.

If you look carefully, you will see that each component we introduced has well-defined input and output. Figure 2.2 breaks down our modular approach to the design.

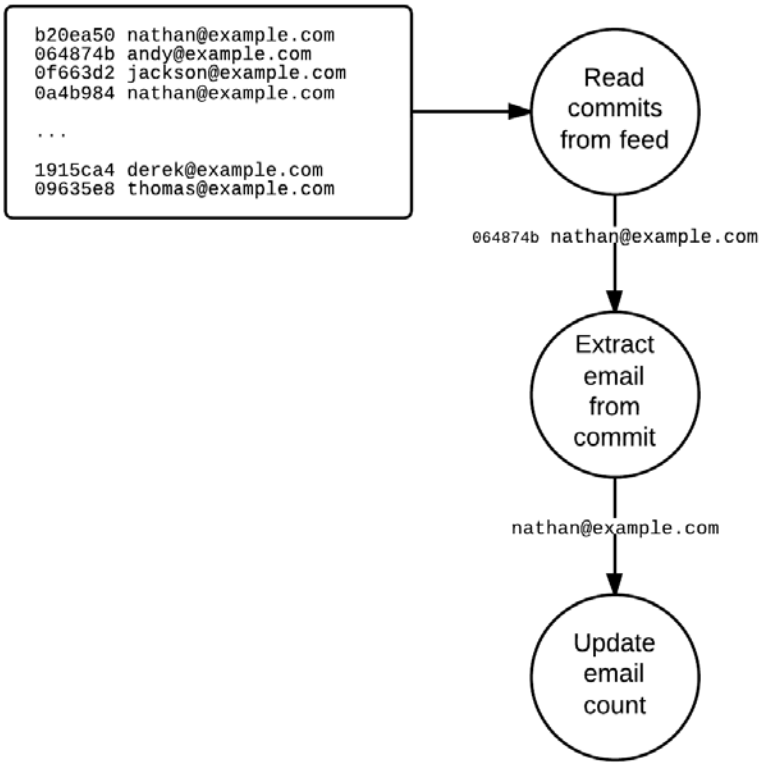


Figure 2.2 Steps in our solution shown with their corresponding input and output

So there it is, the solution to our problem broken down into simple components, each with a singular purpose. Now that we have a well-formed idea of how we want to solve this problem, lets frame our solution within the context of Storm.

2.2 Basic Storm Concepts

To help you understand the core concepts in Storm, it makes sense to go over the common terminology used in Storm. We will do this within the context of our sample design. We will begin our explanation with the most basic component in Storm: the topology.

2.2.1 Topology

Let's take a step back from our example really quick in order to understand what a topology is. Think of a very simple linear graph with some nodes and directed edges between those nodes. Now imagine that each one of those nodes represents a single process or computation and each edge represents the result of one computation being passed as input to the next computation. Figure 2.3 illustrates this more clearly.

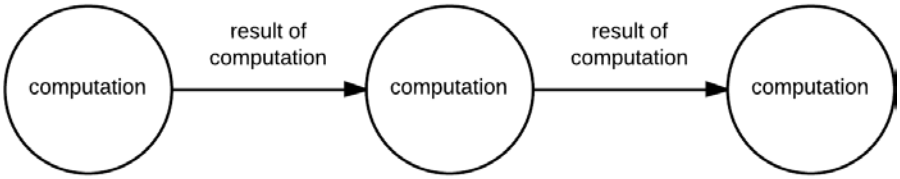


Figure 2.3 Graph with nodes representing computations and edges representing results of computations

This is a topology in its simplest form. A Storm topology is a graph of computation where the nodes represent some individual computations and the edges represent the data being passed between nodes. We then feed data into this graph of computation in order to achieve some goal. What does this mean exactly? Let's go back to our dashboard example to show you exactly what we're talking about.

Looking at the modular breakdown of our problem, we are able to identify each of the components we just mentioned in the definition of a topology. Figure 2.4 illustrates this correlation; there is a lot to take in here, so take your time.

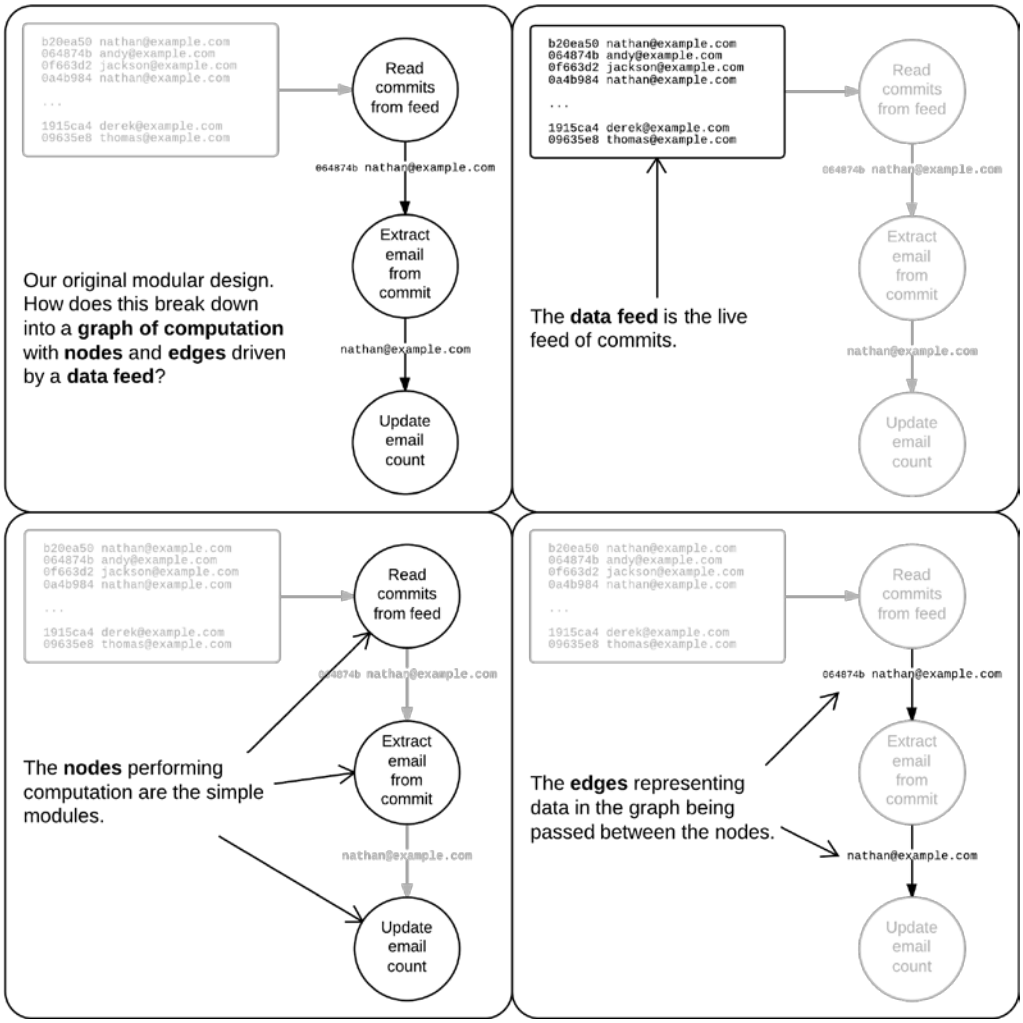


Figure 2.4 Our design mapped to the definition of a Storm topology

Each concept we mentioned in the definition of a topology can be found in our design. The actual topology consists of the nodes and edges. This topology is then driven by the continuous live feed of commits. Our design fits in quite well within the framework of Storm. Now that we understand what a topology is, we'll dive into the individual components that make up a topology.

2.2.2 Tuple

The nodes in our topology send data between one another in the form of tuples. A tuple is an ordered list of values, where each value is assigned a name. There are two tuples in our example: 1) the commit containing the commit id and developer email and 2) the developer email. These can be seen in Figure 2.5.

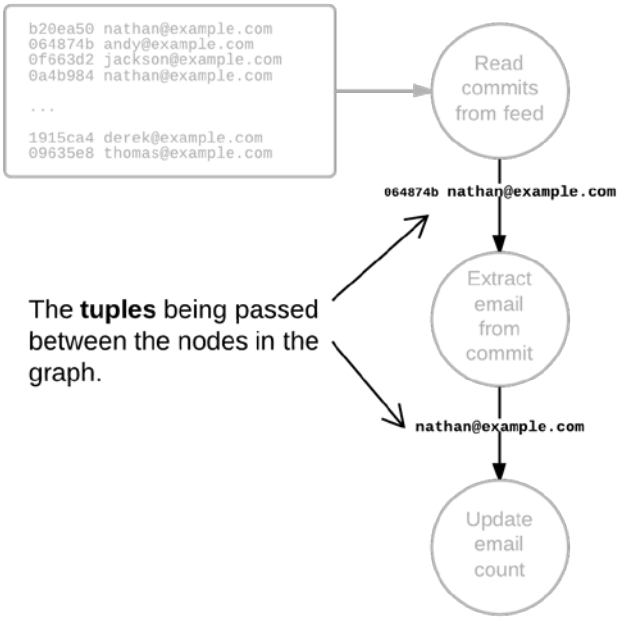


Figure 2.5 The tuples in our topology

The types of the values within a tuple are dynamic and do not need to be declared. However, Storm does need to know how to serialize these values so it can send the tuple between nodes in the topology. Storm already knows how to serialize primitive types but will require custom serializers for any custom type you define. We will discuss custom serializers later in the book.

The nodes in our topology can create tuples and then (optionally) send those tuples to any number of nodes in the graph. The process of sending a tuple out to be processed by any number of nodes is called **emitting** a tuple. If a node emits a tuple, it must define the names for each of the values in the ordered list. As an example, a node that emits a tuple containing an email would need the following two lines of code:

1. `outputFieldsDeclarer.declare(new Fields("email"))` to declare that all tuples emitted by the node contain a field named "email".
2. Something like `outputCollector.emit(new Values("nathan@example.com"))` to emit an instance of a tuple.

A more complete example explaining `outputFieldsDeclarer`, `outputCollector` and where this code needs to go is seen later in the full code listing for this chapter, so don't worry if the picture is still a little muddied right now. For now, this understanding should suffice and we'll move on from tuples to the core abstraction in Storm, the stream.

2.2.3 Stream

According to the Storm wiki, a stream is an "unbounded sequence of tuples". This is a great explanation of what a stream is with maybe one addition. A stream is an unbounded sequence of tuples between two nodes in the topology. A topology can contain any number of streams. Other than the very first node in the topology that reads from the data feed, nodes can accept one or more streams as input. Nodes will then normally perform some computation or transformation on the input tuples and emit new tuples, thus creating a new output stream. These output streams then act as input streams for other nodes, and so on.

Looking at our GitHub commit count topology, we have two streams. The first stream starts with the node that continuously reads commits from a feed. This node emits a tuple with the commit to another node that extracts the email. This stream can be seen in Figure 2.6.

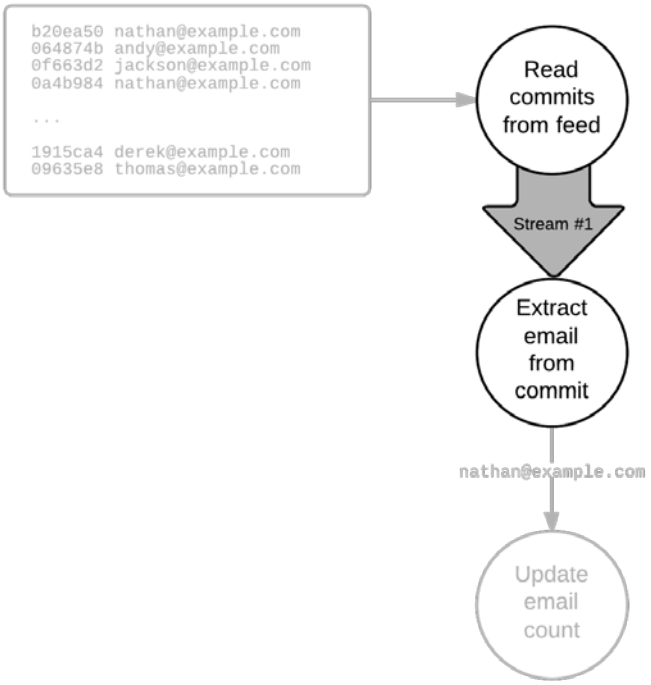


Figure 2.6 Stream between node that reads commits & the node that extracts emails

The second stream starts with the node that extracts the email from the commit. This node transforms its input stream (containing commits) by emitting a new stream containing only emails. The resulting output stream serves as input into the node that updates the in-memory map. This is seen in Figure 2.7.

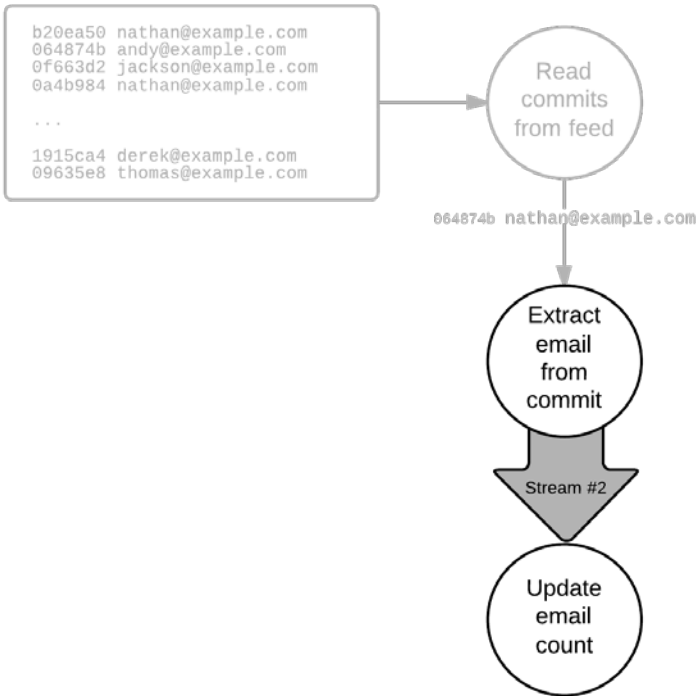


Figure 2.7 Stream between node that extracts emails & node updating the in-memory map

Our Storm GitHub scenario is an example of a very simple chained stream (multiple streams chained together). Streams may not always be this straightforward however. Take the example in Figure 2.8. This shows a topology with four different streams. The first node emits a tuple that is actually processed by two different nodes. This results in two separate streams. Each of those nodes then emits tuples to their own new output stream.

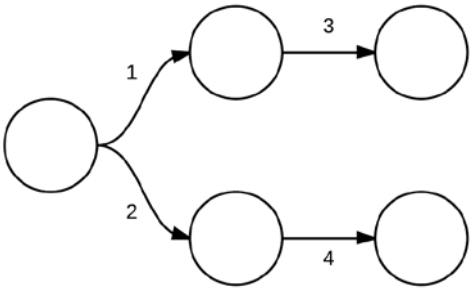


Figure 2.8 Topology with four streams

The combinations are endless with the number of streams that may be created, split and then joined again. The examples later in this book will get into the more complex chains of streams and why it is beneficial to design a topology in such a way. For now, we will continue with our straightforward example and move on to the source of a stream for a topology.

2.2.4 Spout

A spout is the source of a stream in the topology. Spouts normally read data from an external data source and emit tuples into the topology. Spouts can listen to message queues for incoming messages, listen to a database for changes, or listen to any other source of a feed of data. In our example, the spout is listening to the real-time feed of commits being made to the Storm repository. This can be seen in Figure 2.9.

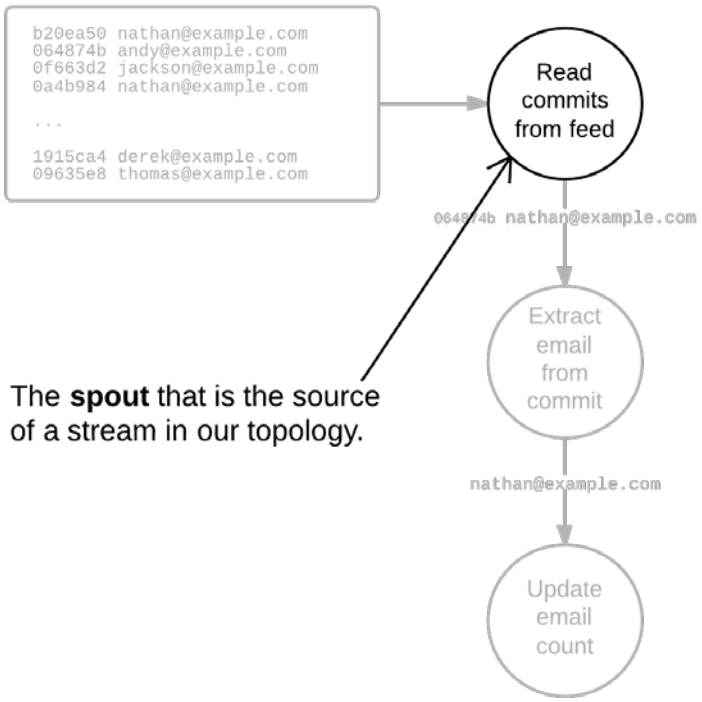


Figure 2.9 The spout in the Storm project commit count topology

Spouts are only part of the picture however. We need something to actually perform the processing within the topology. This is where bolts come into play.

2.2.5 Bolt

A bolt is a node that does some type of computation or transformation to its input stream. Bolts can do anything from filtering, to aggregations, to joins, to talking to databases. Bolts accept tuples from an input stream and can optionally emit new tuples, thus creating an output stream. The bolts in our example are the following:

- A bolt that extracts the developer's email from the commit. This bolt accepts a tuple containing a commit with a commit id and email from its input stream. It transforms that input stream, emitting a new tuple containing only email address to its output stream.
- A bolt that updates the map of emails to commit counts. This bolt accepts a tuple containing an email address from its input stream. Since this bolt updates an in-memory map and does not emit a new tuple, it does not produce an output stream.

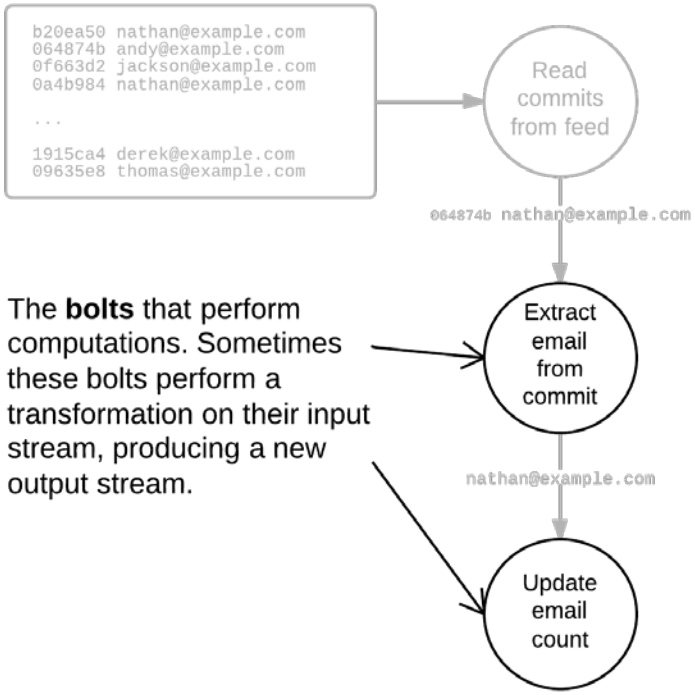


Figure 2.9 The bolts in the Storm project commit count topology

The bolts in our example are extremely simple. As we move along in the book, we will be creating bolts that do much more complex transformations, sometimes even reading from multiple input streams and producing multiple output streams. We're getting ahead of ourselves here however. Before we get that far, we need to understand how exactly bolts and spouts work in practice.

How Bolts & Spouts Work Under the Covers

In our previous diagrams, both the spout and bolts were shown as single components. This is true from a logical standpoint. However when it comes to how spouts and bolts work in reality, there is a little more to it. In a running topology, there are normally numerous instances of each type of spout/bolt performing computations in parallel. This can be seen in Figure 2.10 where the bolt for extracting the email from the commit and the bolt for updating the email count are each running across three different instances. Notice how a single instance of one bolt is emitting a tuple to a single instance of another bolt.

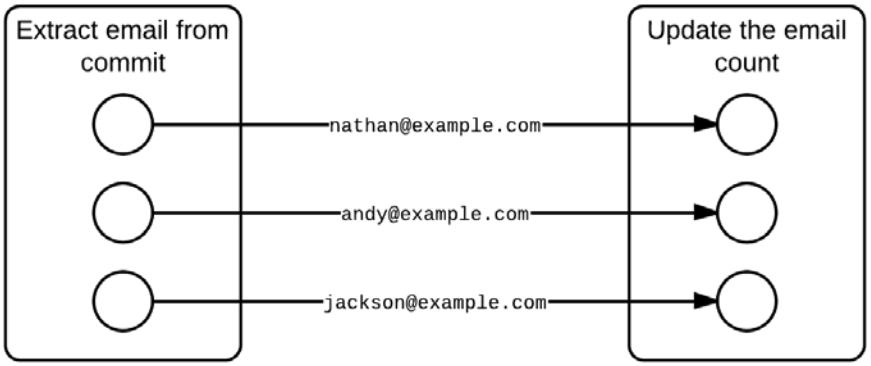


Figure 2.10 Several instances of a bolt handling the processing

Figure 2.10 shows just one possible scenario of how the tuples would be sent between instances of the two bolts. In reality, the picture is more like Figure 2.11, where each bolt instance on the left is emitting tuples to several different bolt instances on the right.

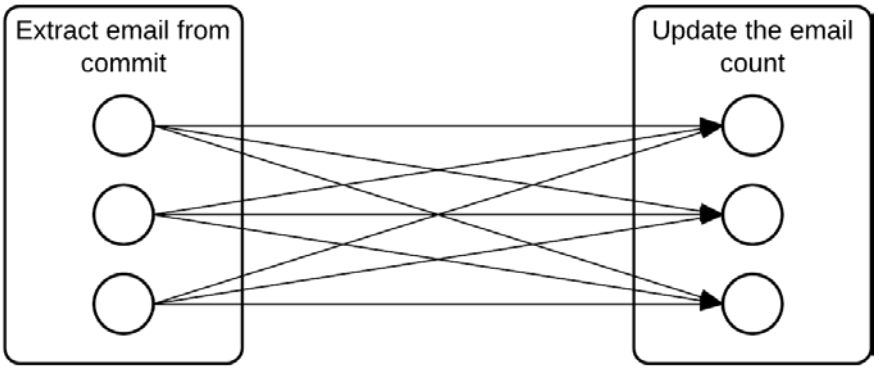


Figure 2.11 Bolt instances emitting tuples to various other bolt instances

Understanding the breakdown of spout/bolt instances is extremely important before moving forward. In fact, let's pause for a moment and rehash where we are before diving our final concept. So far we have a **topology** that consists of nodes and edges. The nodes represent either spouts or bolts and the edges represent streams of tuples between these spouts and bolts. A **tuple** is an ordered list of named values. A **stream** is an unbounded sequence of tuples between a spout and a bolt or between two bolts. A **spout** is the source of a stream in a topology, usually listening to some sort of live feed of data. A **bolt** accepts a stream of tuples from another spout or bolt, normally performing some sort of computation or transformation on these input tuples. The bolt can then optionally emit new tuples that become that serve as the input stream into another bolt in the topology. Now each spout and

bolt will have one or many individual instances that perform all of this processing in parallel. That's quite a bit of material, so be sure to let this sink in before we move to our next topic. Ready? Good. Time to move on to stream groupings.

2.2.6 Stream Grouping

A stream grouping determines how tuples are sent between instances of spouts and bolts. There are many types of stream groupings that Storm provides out of the box. We are going to cover two common groupings in this chapter, both of which are being used in our Storm GitHub commit count example.

> **NOTE** There are several types of stream groupings and we will cover them in case studies where they can be applied in different parts of the book.

SHUFFLE GROUPING

A shuffle grouping is a type of stream grouping where tuples are emitted to instances of bolts at random. A shuffle grouping guarantees that each bolt instance receives an equal number of input tuples. This is the simplest of all the different stream groupings.

This grouping is useful in many basic cases. In our example, it is a useful stream grouping to use when sending tuples between the spout that continuously reads from the feed of commits and the bolt that extracts the email from the commit.

The same cannot be said when emitting tuples to the bolt that updates the in-memory map. Think of a case where two tuples for commits by `nathan@example.com` are going through our topology. In a shuffle grouping, the first tuple might go to one instance and the second tuple might go to another instance. Given the timing of things, each instance could be manipulating a map with an incorrect number of commits for that email. A different type of grouping solves this problem for us: the fields grouping.

FIELDS GROUPING

A fields grouping is a type of stream grouping where tuples with the same value for a particular field name are always emitted to the same instance of a bolt. This works well for counting emails, as the same email will always go to the same bolt instance, resulting in an accurate count for that email. With the fields grouping now covered, you have a basic understanding of the core concepts and the terminology used in Storm. Let's dive into the code and implement our design.

2.3 Implementing GitHub Commit Count Dashboard in Storm

This section will take you through the code for implementing the GitHub dashboard. After going through the code, you will have an understanding of how spouts, bolts and topologies are implemented. Before we dive into the code, however, we will get you set up with the necessary Storm jar files.

2.3.1 Setting up a Storm Project

The easiest way to get the Storm jars on your classpath for development is to use Maven. The following must be added to your project's pom.xml file.

Listing 1.1 pom.xml

```
<project>
..
  <repositories>
    ..
    <repository>
      <id>clojars.org</id>
      <url>http://clojars.org/repo</url> #1
    </repository>
    ..
  </repositories>
  ..
  <dependencies>
  ..
    <dependency>
      <groupId>storm</groupId>
      <artifactId>storm</artifactId>
      <version>0.9.0.1</version> #2
      <!-- <scope>provided</scope> --> #3
    </dependency>
  ..
  </dependencies>
</project>
```

#1 Storm is hosted on the Clojars Maven repository.
#2 Most recent version of Storm at the time of writing.
#3 For topologies that will be deployed to a real cluster, scope should be set to provided. But we are leaving it commented out as we're still in learning mode.

Once these additions have been made to your pom.xml, you should have all of the necessary dependencies for writing code and running Storm topologies locally on your development machine.

2.3.2 Storm API Hierarchy

Storm API supports these concepts by having interfaces that directly map to them.

Spout. All interfaces within Storm are prefixed with an uppercase I.

TOP LEVEL INTERFACES:

- `IComponent` – Common interface for all topology components (bolts and spouts). Defines a components configuration and its output.
- `ISpout` – Defines the responsibilities of a Spout.
- `IBolt` – Defines the responsibilities of a Bolt

These interface are not directly used in building a Storm topology by users of Storm but you should know them to understand the hierarchy inherent within Storm's API.

RICH INTERFACES/BASE CLASSES:

- `IRichSpout` – A complete Spout interface that combines `ISpout` & `IComponent`.
- `IRichBolt` – A complete Bolt interface that combines `IBolt` & `IComponent`.
- `BaseRichSpout` – Partial implementation of `IRichSpout`.
- `BaseRichBolt` – Partial implementation of `IRichBolt`.

These interface and base classes are used for building Storm topologies when you need complete access to the fluency of Storm API.

BASIC BOLT INTERFACES/BASE CLASSES:

- `IBasicBolt` – A complete Bolt interface extends `IComponent` but only supports subset of the features of `IRichBolt`
- `BaseBasicBolt` – Partial implementation of `IBasicBolt`.

Basic bolt interfaces are used for incredibly simple implementations of bolts. They take over the responsibilities usually accessible to a developer with `IRichBolt` so it makes simpler implementations more concise but takes some of the fluency of the rich feature set made accessible through `IRichBolt`.

For our GitHub Commiter Dashboard, we will be using `BaseRichSpout` for the spout. For all of our bolts, we will use the simpler `BaseBasicBolt`.

NOTE The differences between `BaseRichBolt` and `BaseBasicBolt` and when to use one over the other will be covered as part of guaranteed message processing in Chapter 4.

2.3.3 Building a Spout

We will begin our coding at the point where data flows into our topology: the spout. In our design, it listens to a live feed of commits being made to a particular GitHub project using GitHub API. To keep things simple, we won't actually set up our spout to listen to a live feed. Instead, our spout will simulate this by continuously reading from a file containing commit data, emitting a tuple for each line in the file.

Listing 2.2 CommitFeedListener.java

```
public class CommitFeedListener extends BaseRichSpout { #1
  private SpoutOutputCollector outputCollector; #2
  private List<String> commits; #3

     @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) { #4
    declarer.declare(new Fields("commit")); #5
  }

     @Override
  public void open(Map configMap,
                   TopologyContext context,
                   SpoutOutputCollector outputCollector) { #6
    this.outputCollector = outputCollector;
```

```
    try {
      commits = IOUtils.readLines(ClassLoader.getSystemResourceAsStream(
        "changelog.txt"), Charset.defaultCharset().name()); #7
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

     @Override
  public void nextTuple() { #8
    for (String commit : commits) {
      outputCollector.emit(new Values(commit)); #9
    }
  }
}
```

#1 Our spout extends BaseRichSpout.
#2 Used for emitting tuples.
#3 In-memory list of commits.
#4 Storm calls this method to determine the tuple field names.
#5 How a spout declares the names of the fields for each tuple it emits. Order here must match the order of emitted values.
#6 Method Storm calls upon initializing a spout instance.
#7 Obtain the file containing the list of commits & read the contents of the file into our list of strings.
#8 Method Storm calls when it is ready for the next tuple. You are free to emit more than one. The frequency with which this method gets called is configurable, with a default of 1ms.
#9 Loop through the list of commits emitting a tuple for each commit

2.3.4 Building the Bolts

We have a spout that is serving as the source of the stream, emitting tuples containing commits we saved in a file. Now we need to actually do something with that commit data, namely:

1. Extract the email from an individual commit.
2. Update the commit count for that email.

Let's go through the implementation of the two bolts that perform these tasks. Starting with the `EmailExtractor` bolt:

Listing 2.3 EmailExtractor.java

```
public class EmailExtractor extends BaseBasicBolt { #1
     @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) { #2
    declarer.declare(new Fields("email")); #3
  }

     @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) { #4
    String commit = tuple.getStringByField("commit"); #5
    String[] parts = commit.split(" ");
    outputCollector.emit(new Values(parts[1])); #6
  }
}
```

#1 Our bolt extends BaseBasicBolt.
#2 Storm calls this method to determine the tuple field names.
#3 How a bolt declares the names of the fields for each tuple it emits. Order here must match the order of emitted values.
#4 Method Storm calls to process an incoming tuple.
#5 Using a convenience method on the tuple, extract the value of the 'commit' field from the incoming tuple.
#6 Emit a new tuple containing the email.

The next bolt in our topology will process the email that was just emitted from the `EmailExtractor`. This bolt, named `EmailCounter`, will update an in-memory map of emails to commit counts as explained earlier. The `EmailCounter` extends/overrides all of the same fields as our previous bolt, with some differences in what it does within the overridden methods.

Listing 2.4 EmailCounter.java

```
public class EmailCounter extends BaseBasicBolt {
  private Map<String, Integer> counts; #1

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // This bolt does not emit anything and therefore does
    // not declare any output fields.
  }

  @Override
  public void prepare(Map config,
                      TopologyContext context) { #2
    counts = new HashMap<String, Integer>();
  }

  @Override
  public void execute(Tuple tuple, ,
                      BasicOutputCollector outputCollector) {
    String email = tuple.getStringByField("email"); #3
    counts.put(email, countFor(email) + 1);
    printCounts();
  }

  private Integer countFor(String email) {
    Integer count = counts.get(email);
    return count == null ? 0 : count;
  }

  private void printCounts() {
    for (String email : counts.keySet()) {
      System.out.println(
        String.format("%s has count of %s", email, counts.get(email)));
    }
  }
}
```

#1 In-memory map used for mapping commit counts to developer email addresses.
#2 Storm calls this method when initializing a bolt instance.
#3 Using a convenience method on the tuple, extract the value of the 'email' field from the incoming tuple.

2.3.5 Wiring everything together to form the Topology

We now have implementations for our spout and bolts. What's left is to actually create the topology, defining how the stream flows between these components. We would then like to run our topology and see our first topology in action. The following code listing will show you how to wire up all of the components of the topology along with defining how this topology can be run locally in process.

Listing 1.5 LocalTopologyRunner.java

```
public class LocalTopologyRunner {
  private static final int TEN_MINUTES = 600000;

  public static void main(String[] args) { #1
    TopologyBuilder builder = new TopologyBuilder(); #2

    builder.setSpout("commit-feed-listener", new CommitFeedListener()); #3

    builder
      .setBolt("email-extractor", new EmailExtractor()) #4
      .shuffleGrouping("commit-feed-listener"); #5

    builder
      .setBolt("email-counter", new EmailCounter()) #6
      .fieldsGrouping("email-extractor", new Fields("email")); #7

    Config config = new Config(); #8
    config.setDebug(true);

    StormTopology topology = builder.createTopology(); #9

    LocalCluster cluster = new LocalCluster(); #10
    cluster.submitTopology("github-commit-count-topology",
      config,
      topology); #11

    Utils.sleep(TEN_MINUTES);
    cluster.killTopology("github-commit-count"); #12
    cluster.shutdown(); #13
  }
}
```

#1 This main method will need to be executed in order to run our topology locally in process.
#2 Convenient builder class for creating and submitting topologies to a cluster.
#3 Sets the spout of our topology to the commit feed listener. We give the spout an id and pass in a new instance.
#4 Adds the email extractor bolt to the topology. We give the bolt an id and pass in a new instance.
#5 Set the stream grouping on this bolt as a shuffle grouping, with the commit feed listener as the source of input tuples.
#6 Adds the email counter bolt to the topology. We give the bolt an id and pass in a new instance.
#7 Sets the stream grouping on this bolt as a fields grouping, with the email extractor as the source of input tuples.
#8 Provide a very basic topology configuration by setting debug output to true. This will allow us to see messages being passed between components.
#9 Create the topology.

#10 LocalCluster allows you to run topologies locally on your machine in process. This is great for development and testing purposes, which is why we are using it here.
#11 Submit our topology to the local cluster, providing an id for the topology, the topology configuration and our actual Storm topology.
#12 Kill the topology after a certain amount of time.
#13 Shut down the local cluster after killing the topology.

All that is left to be done is to run the main method of `LocalTopologyRunner` and watch the topology do its thing!

2.4 Summary

In this chapter we've covered:

- The core concepts and terminology used in Storm. Including topology, tuple, stream, spout, bolt and stream grouping.
- The basic code for building a topology. This includes defining a spout, bolt, wiring them up with a topology builder, and running within a local cluster.

The foundation has been set to start building bigger, more complex and efficient topologies to address a wide range of scenarios presented throughout the book. In the next chapter, we will discuss some best design patterns to follow when designing topologies. We will also dive into one of the aspects that make Storm so powerful, parallelism.

3

Topology Design

This chapter covers

- How to break down a problem to fit into the constructs of a Storm topology
- Working with unreliable data sources
- Integration with external services and data stores
- Basic understanding of parallelism within a Storm topology
- Best practices for topology design

In the last chapter, we got our feet wet by building a simple topology. We built a topology that counts commits made to a Github project. We broke it down into Storm's two primary components: spouts and bolts, but we did not go into detail on why the problem was broken down in that manner. Neither did we go over why we broke down the problem into multiple steps involving multiple bolts. In this chapter we will take it a step further and expand upon those basic Storm concepts by teaching you how to think about modeling & designing solutions with Storm. You will learn strategies for problem analysis that can help you end up with a good design: a model for representing the workflow of the problem at hand.

In addition, it is important that we start building intuition on how scalability (or parallelization of units of work) is built into Storm as that affects the approach that should be taken with topology design. We will also emphasize different strategies suitable for gaining the most out of your topology in terms of speed.

After reading this chapter, not only will you be able to easily take apart a problem and see how it fits within Storm, but you will also be able to determine whether Storm is the right solution for tackling that problem. This will give you a solid understanding of topology design so that you can envision solutions to big data problems.

3.1 Approaching topology design

The approach for topology design could be broken down as follows:

1. Problem definition / Formation of a conceptual solution

   This serves to provide a clear understanding of the problem being tackled. It also serves as a place to document the requirements to be placed on any potential solution (incl. requirements with regards to speed which is a common criterion in big data problems). Then we will model a solution (not an implementation) that addresses the core need(s) of the problem.

2. Precepts for mapping the solution to Storm

   Then we will follow a set of tenets for breaking down the proposed solution in a manner that allows us to envision how it will map to Storm primitives (a.k.a. Storm concepts). At this stage, we will come up with a design for our topology. This design will be tuned and adjusted as needed in the following steps.

3. Initial implementation of a solution

   Each of the components will be implemented at this point.

4. Scaling the topology

   Then we will turn the knobs that Storm provides for us to run this topology at scale.

5. Tuning based on observations

   Finally, we will make adjustments to the topology based on observed behavior after it is actually running. This may involve additional tuning for achieving scale as well as design changes that may be warranted for efficiency.

3.2 Problem definition – A Social Heatmap

Let's pick a problem that will serve as a good use case and encapsulate several challenging topics around topology design.

STORM GOES SOCIAL - A HEATMAP FOR POPULARITY

Imagine this… It's Saturday night and you are out drinking at a bar; enjoying the good life with your friends. You are finishing your third drink and you are starting to feel like you need a change of scene. Maybe switch it up and go to different bar? So many choices… How do you even choose? Being a socialite, of course you would want to end up in the bar that's most popular. Now you don't want to go somewhere that was voted best in your neighborhood glossy magazine. That was so last week. You want to be where the action is right now, not last week, not even last hour. You are the trendsetter. You have a responsibility to show your friends a good time.

Okay, maybe that's not you. But does that represent the average social network user? Now what can we do to help this person? If we can represent the answer this person is looking for in a graphical form factor, it would be ideal. A map that identifies the neighborhoods with highest density of activity in bars as hot zones can convey everything

quickly. A heat map to our friend's aid – a heat map can identify the general neighborhood in a big city like New York or San Francisco, and generally when a picking a popular bar, it's always better to have a few choices in within close proximity to one another, just in case.

Other case studies for heat maps

What kind of problems benefit from visualization using a heat map? A good candidate would allow us to use heat map's intensity to model the relative importance of a set of data points are compared to others within an area (geographical or otherwise).

- Spread of a wild fire in California or an approaching hurricane in the east coast or disease outbreak can be modeled and represented as a heat map to warn residents.
- On an election day, we might want to know
  - Which political districts had the most voters turn out? We can depict this on a heat map by modeling the turnout numbers to reflect the intensity.
  - Which political party/candidate/issue received the most votes by modeling the party, candidate or issue as a different color with the intensity of the color reflecting the number of votes.

3.2.1 Formation of a conceptual solution

Now where do we begin? There are multiple social networks with the concept of check-ins. Let's say we have access to a fire-hose that collects check-ins for bars from all of these different networks. This fire-hose will emit a bar's address for every check in. Okay, we have a starting point, but it's also good to have an end goal in mind. Let's say that our end goal is a geographical map with a heat map overlay identifying neighborhoods with most popular bars.

Figure 3.1 Using check-ins to build heat map of bars

Figure 3.1 illustrates our proposed solution in which we will transform multiple check-ins from different venues to be shown in a heat map. So the solution that we need to model within Storm becomes the method of transforming (or aggregating) check-ins into a data set that can be depicted on a heat map.

3.3 Precepts for mapping the solution to Storm

The best way to start is to contemplate the nature of data flowing through this system. When we better understand the peculiarities contained within the data stream, we can become more attuned to requirements that can be placed on this system realistically.

3.3.1 Consider the requirements imposed by the data stream

We have a fire-hose emitting addresses of bars for each check-in. But this stream of check-ins does not reliably represent every single user that went to a bar. A check-in is not equivalent to physical presence at a location. It's better to think of it as a sampling of real life as not every single user checks in. But that leads us to question whether check-in data is even useful for solving this problem. I think we can safely assume that check-ins at bars are proportional to people actually at those locations

So we know that,

1. Check-ins are sampling of real life scenario, it's not complete.
2. But they're proportionately representative.

Caveat: Data Loss Assumptions

Let's make the assumption here that the data volume is large enough to compensate for data loss and that any data loss is intermittent and not sustained long enough to cause a noticeable disruption in service. These assumptions help us portray a case of working with an unreliable data source.

Now we have our first insight about our data stream: a proportionately representative but possibly incomplete stream of check-ins. What's next? We know our users want to be notified about the latest trends in activity as soon as possible. In other words, we have a strict speed requirement. Get the results to the user as quickly as possible as the value of data diminishes with time.

What emerges from consideration of the data stream is that we do not need to worry too much about data loss. We can come to this conclusion because we know that our incoming data set is incomplete, so accuracy down to some arbitrary, minute degree of precision is not necessary. But it's proportionately representative and that's good enough for determining popularity. Combine this with the requirement on speed, we now know that as long as we get recent data quickly to our users, they will be happy. Even if there is data loss, the past results will be replaced soon anyway. This maps directly to the idea of working with an ***unreliable***

data source in Storm. With an unreliable data source, you do not have the ability to retry a failed operation. The data source may not have the ability to replay a data point. In our case, we are sampling real life by way of check-ins and that mimics the availability of an incomplete data set.

In other cases, you may have a reliable data source that does have the ability to replay data points that fail, but you may not want to. Perhaps accuracy is less important than speed. Then approximations can be just as acceptable. In such instances it's possible to treat a reliable data source as if it was unreliable by choosing to ignore any reliability measures it provides like ability to replay failed data points.

RELIABLE DATA SOURCES We will cover reliable data sources along with fault tolerance in the next chapter.

3.3.2 Represent data points as Tuples

Our next step is to identify the individual data points that flow through this stream. It's easy to accomplish by considering the beginning and end. We begin with a series of data points composed of street addresses of bars with activity. We will also need to know the time the check-in occurred. So our input data point can be represented as `(9:00:07 PM, 287 Hudson St New York NY 10013)`. That is time and an address where the check-in happened. This would be our input *tuple*. If you remember from Chapter 1, a tuple is a storm primitive for representing a data point.

We have the end goal of building a heat map with latest activity at bars. So we need to end up with data points representing timely coordinates on a map. We can attach a time interval (say 9:00:00 PM to 9:00:15 PM, if we wanted 15 second increments) to a set of coordinates that occurred within that interval. Then at the point of display within the heat map, we can pick the latest available time interval. Coordinates on a map can be expressed by way of latitude and longitude (say 40.7142° N, 74.0064° W for New York, NY). It's standard form to represent 40.7142° N, 74.0064° W as (40.7142, -74.0064). But there might be multiple coordinates representing multiple check-ins within a time window. So we need a list of coordinates for a time interval. Then our end data point is starting to look like `(9:00:00 PM to 9:00:15 PM, List((40.719908,-73.987277),(40.72612,-74.001396))`. That's an end data point containing a time interval and two corresponding check-ins at two different bars.

What if there is two or more check-ins at the same bar within that time interval? Then that coordinate will be duplicated. How would we handle that? One option is to keep counts of occurrences within that time window for that coordinate. This involves determining same-ness of coordinates based on some arbitrary but useful degree of precision. To avoid all that, let's keep duplicates of any coordinate within a time interval with multiple check-ins. By adding multiples of the same coordinates to a heat map, we can let the map generator make use of multiple occurrences as a level of hotness (rather than using occurrence count for that purpose).

Then our end data point will look like `(9:00:00 PM to 9:00:15 PM, List((40.719908,-73.987277),(40.72612,-74.001396),(40.719908,-73.987277))`. Note that the first coordinate is duplicated. This is our end tuple that will be served up in the form of a heat map.

Having a list of coordinates grouped by a time interval…

1. Allows us to easily build a heat map by using Google Maps API. We can do this by adding a heat map overlay on top of a regular Google Map
2. Lets us to go back in time to any particular time interval and see the heat map for that point in time

3.3.3 Steps for determining the topology composition

Our approach for designing a Storm topology can be broken down into three steps.

1. Determine the input data points and how they can be represented as tuples.
2. Determine the final data points needed to solve the problem and how they can be represented as tuples.
3. Bridge the gap between the input tuples and the final tuples by creating a series of operations that transform them.

We already know our input and desired output.

```
Input tuples – (9:00:07 PM, 287 Hudson St New York NY 10013)
End tuples –  (9:00:00 PM to 9:00:15 PM, List((40.719908,-73.987277),
                                              (40.72612,-74.001396),
                                              (40.719908,-73.987277))
```

Somewhere along the way, we need to transform the addresses of bars into these end tuples. Let's see how we can break down the problem into these series of operations.

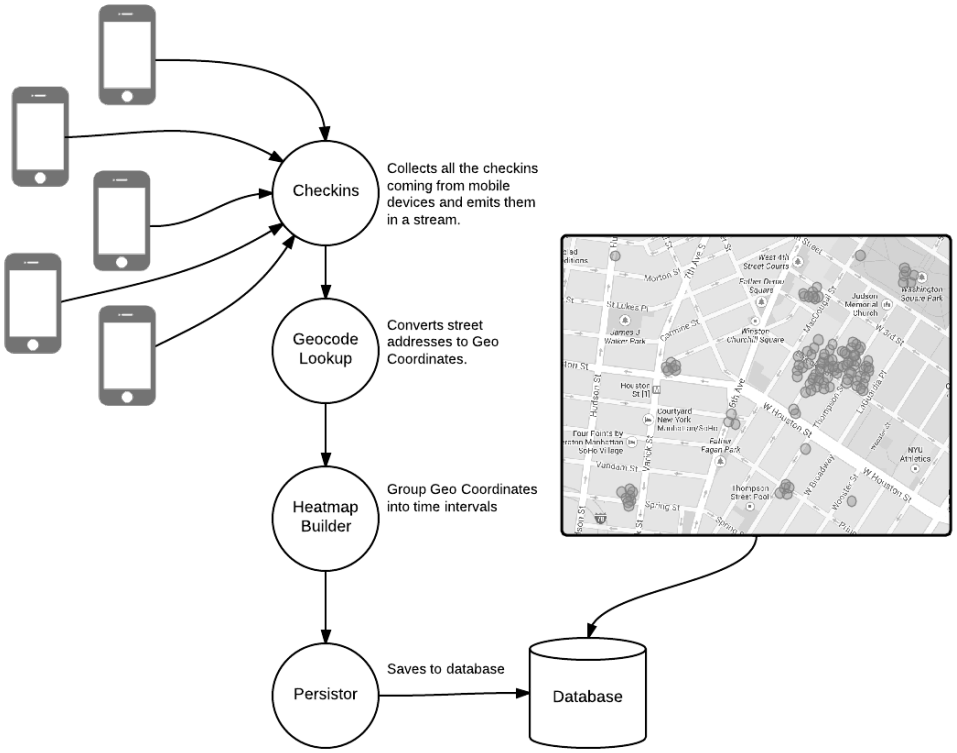


Figure 3.2 Transforming input tuples to end tuples via a series of operations

OPERATIONS AS SPOUTS AND BOLTS

We have come up with a series of operations to transform input tuples to end tuples. Let's see how these four operations map to Storm primitives (we are using terms 'storm primitives' and 'storm concepts' interchangeably).

1. Checkins – This will be source of input tuples into the topology, so in terms of Storm concepts this will be our spout. In this case, since we're using an unreliable data source, we will build a spout that has no capability of retrying failures.
2. Geocode Lookup – This will take our input tuple and convert the street address to a Geo Coordinate by querying Google Maps Geocoding API. This is the first bolt in the topology.
3. Heat Map Builder – This is the second bolt in the topology and it will keep a data structure in memory to map each incoming tuple to a time interval, thereby grouping check-ins by time interval. When each time interval is completely passed, it will emit the list of coordinates associated with that time interval.

4. Persistor – We will use this third and final bolt in our topology to save our end tuples to a database.

Figure 3.3 provides an illustration of the design mapped to Storm concepts. We've touched upon everything but the stream groupings, which will cover when we get into the code for this topology.

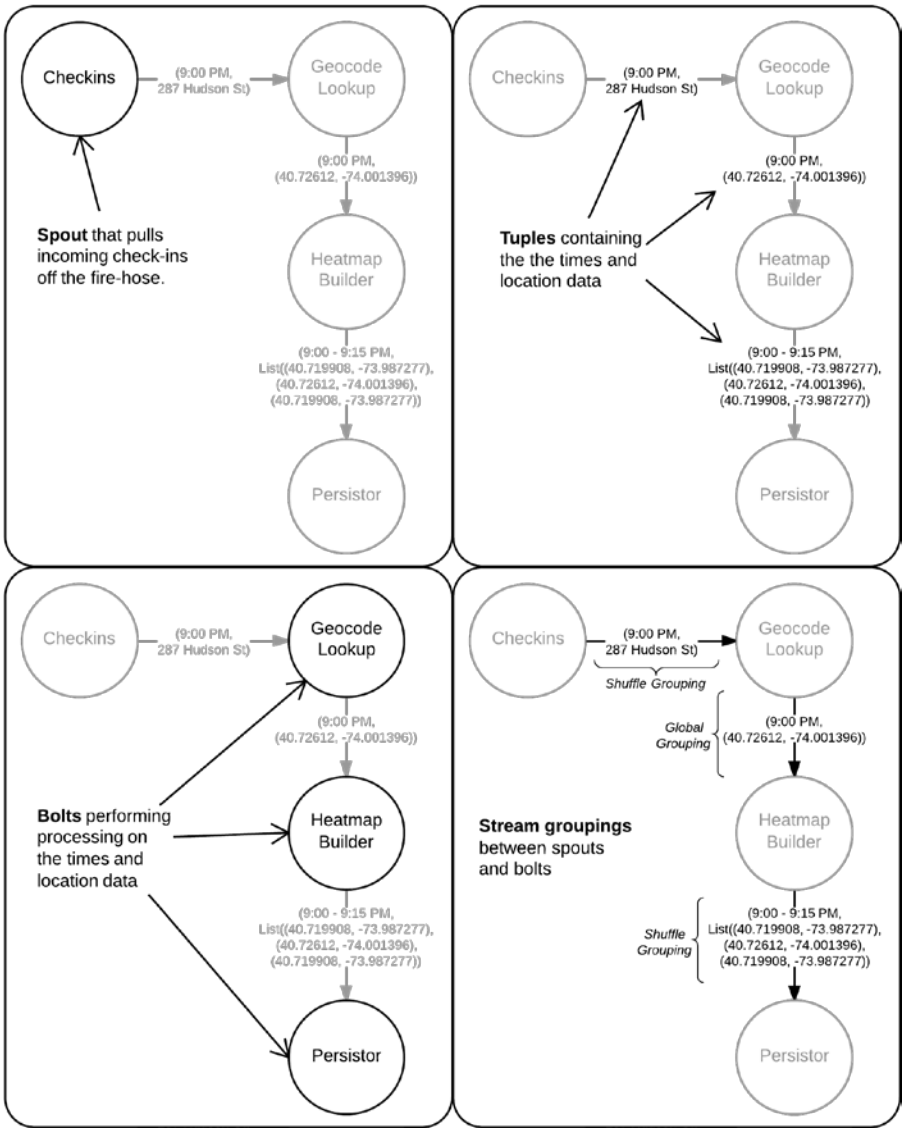


Figure 3.3 Heat map design mapped to Storm concepts

We have identified the general makeup of the topology as part of this design. We are now ready to move forward with the implementation.

3.4 Initial implementation of the design

Let's start with simple implementations of each of the components within our proposed design. Later we will adjust each of these implementations for efficiency or to address some of their shortcomings.

3.4.1 Spout – Read data from a source

For the purpose of this chapter, we will use a text file as our source of data for check-ins. To feed this data set into our Storm topology, we need to write a Spout that reads from this file and emits a tuple for each line. Since our input tuple is a time and address, let us represent the time as `Long` (standard Unix time) and the address as a `String` with the two separated by a comma in our text file.

Listing 3.1 - Checkins.java

```
public class Checkins extends BaseRichSpout {  #1
  private List<String> checkins; #2
  private int nextEmitIndex; #3
  private SpoutOutputCollector outputCollector;

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("time", "address")); #4
  }

  @Override
  public void open(Map config,
                   TopologyContext topologyContext,
                   SpoutOutputCollector spoutOutputCollector) {
    this.outputCollector = spoutOutputCollector;
    this.nextEmitIndex = 0;

    try {
      checkins =
IOUtils.readLines(ClassLoader.getSystemResourceAsStream("checkins.txt"),
                                    Charset.defaultCharset().name()); #5
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  public void nextTuple() {
    String checkin = checkins.get(nextEmitIndex); #6
    String[] parts = checkin.split(",");
    Long time = Long.valueOf(parts[0]);
    String address = parts[1];
    outputCollector.emit(new Values(time, address)); #7

    nextEmitIndex = (nextEmitIndex + 1) % checkins.size(); #8
  }
}
```

#1 Checkins spout extends BaseRichSpout as we saw in Chapter #2.
#2 Store the static checkins from a file in List.
#3 nextEmitIndex will keep track of current position in list as we will recycle the static list of checkins.
#4 Let Storm know that this Spout will emit a tuple containing two fields named 'time' and 'address'
#5 Use Java 7 File IO API to read the lines from 'checkins.txt' file into an in-memory List
#6 When Storm requests the next tuple from the Spout, look up the next checkin from our in-memory List and parse it into a 'time' and 'address' components
#7 Use SpoutOutputCollector provided in the spout open method to emit the relevant fields
#8 Advance the index of next item to be emitted (recycling if at end of list)

Since we're treating this as an unreliable data source, the spout remains very simple, as it does not need to keep track of which tuples failed and which ones succeeded in order to provide fault tolerance. That not only simplifies the spout implementation, it removes quite a bit of bookkeeping Storm needs to do internally and speeds things up. When fault tolerance is not necessary and we can define a service level agreement (SLA) that allows us to discard data at will, an unreliable data source can be beneficial as it is easier to maintain and provides fewer points of failure.

3.4.2 Bolt – Connect to an external service

Our first bolt in the topology will take the address data point from the tuple that got emitted from the Checkins spout and translate that address into a coordinate by querying the Google Maps Geocoding Service. For this we will use the Google Geocoder Java API from https://code.google.com/p/geocoder-java/

Listing 3.2 - GeocodeLookup.java

```
public class GeocodeLookup extends BaseBasicBolt { #1
  private Geocoder geocoder;

  @Override
  public void declareOutputFields(OutputFieldsDeclarer fieldsDeclarer) {
    fieldsDeclarer.declare(new Fields("time", "geocode")); #2
  }

  @Override
  public void prepare(Map config,
                      TopologyContext context) {
    geocoder = new Geocoder(); #3
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    String address = tuple.getStringByField("address"); #4
    Long time = tuple.getLongByField("time");

    GeocoderRequest request = new GeocoderRequestBuilder()
                                    .setAddress(address)
                                    .setLanguage("en")
                                    .getGeocoderRequest();
    GeocodeResponse response = geocoder.geocode(request); #5
```

```
      GeocoderStatus status = response.getStatus();
      if (GeocoderStatus.OK.equals(status)) {
        GeocoderResult firstResult = response.getResults().get(0);
        LatLng latLng = firstResult.getGeometry().getLocation();
        outputCollector.emit(new Values(time, latLng)); #6
      }
    }
  }
```

#1 GeocodeLookup bolt extends BaseBasicBolt
#2 Inform Storm that this bolt will emit two fields, 'time' and 'geocode'
#3 Initialize the Google Geocoder
#4 Extract the 'time' and 'address' fields from the tuple sent by Checkins spout
#5 Query Google Maps Geocoding API with the 'address' value from the tuple
#6 Use first result from Google Geocoding API for Geocoordinate and emit it along with time

We intentionally kept our interaction with Google Geocoding API quite simple. In a real implementation we should be handling for error cases when addresses may not be valid. Additionally the Google Geocoding API imposes a quota when used in this manner that is quite small and not really practical for big data applications. For a big data application like this, you will need to obtain an access level with higher quota from Google if you wish to use them as a provider for Geocoding. Other approaches to consider may include locally caching geocoding results within your datacenter to avoid making unnecessary invocations to Google's API.

We now how have the time and `Geocoordinate` of every check-in. We took our input tuple of `(9:00:07 PM, 287 Hudson St New York NY 10013)` and transformed it into `(9:00:07 PM, (40.72612,-74.001396))`.

3.4.3 Bolt – Collect data in-memory

At this step, we will build the data structure that will represent the heat map. What kind of data structure is suitable?

We have tuples coming into this bolt from the previous `GeocodeLookup` bolt in the form of `(9:00:07 PM, (40.72612,-74.001396))`. We need to group these by time intervals. Let's say 15 second intervals as we want to display a new heat map every 15 seconds. As we know from earlier, our end tuples need to be in the form of `(9:00:00 PM to 9:00:15 PM, List((40.719908,-73.987277),(40.72612,-74.001396),(40.719908,-73.987277))`.

In order to group geo coordinates by time interval, let's maintain a data structure in memory and collect incoming tuples into that data structure isolated by time interval. We can model this as a map.

```
Map<Long, List<LatLng>> heatmaps;
```

This map is keyed by the time that starts our interval. We can omit the end of the time interval as each interval is of the same length. The value will be the list of coordinates that fall into that time interval (incl. duplicates as duplicates or coordinates in closer proximity would indicate a hot zone or intensity with the heat map).

Let us start building the heat map in three steps.

1. Collect incoming tuples into an in-memory map

2. Configure this bolt to receive a signal at a given frequency
3. Emit the aggregated heat map for elapsed time intervals

We will look each step individually and then put everything together.

Listing 3.3 - HeatMapBuilder: Step 1 – Collect incoming tuples into an in-memory map

```
private Map<Long, List<LatLng>> heatmaps;

@Override
public void prepare(Map config,
                    TopologyContext context) {
  heatmaps = new HashMap<>(); #1
}

@Override
public void execute(Tuple tuple,
                    BasicOutputCollector outputCollector) {
  Long time = tuple.getLongByField("time");
  LatLng geocode = (LatLng) tuple.getValueByField("geocode");

  Long timeInterval = selectTimeInterval(time); #2
  List<LatLng> checkins = getCheckinsForInterval(timeInterval);
  checkins.add(geocode); #3
}

private Long selectTimeInterval(Long time) {
  return time / (15 * 1000);
}

private List<LatLng> getCheckinsForInterval(Long timeInterval) {
  List<LatLng> hotzones = heatmaps.get(timeInterval);
  if (hotzones == null) {
    hotzones = new ArrayList<>();
    heatmaps.put(timeInterval, hotzones);
  }
  return hotzones;
}
```

#1 Initialize the in-memory map
#2 Select the time interval that tuple falls into
#3 Adds the geo coordinate to the list of checkins associated with that time interval

The absolute time interval the incoming tuple falls into is selected by taking the check-in time and dividing it by the length of the interval, in this case 15 seconds. For example, if check-in time is 9:00:07.535 PM, then it should fall into the time interval that is 9:00:00.000 – 9:00:15.000 PM. What we are extracting here is the beginning of that time interval which is 9:00:00.000 PM.

Now that we are collecting all the tuples into a heat map, we need to periodically inspect it and emit the coordinates from completed time intervals, so that they can persisted into a data store by the next bolt.

TICK TUPLES

Very often when you are working with a Storm topology, you would need to trigger certain actions periodically, like aggregating a batch of data or flushing some writes to a database. Storm has a feature called Tick Tuples just for that. You can configure a bolt to receive a Tick Tuple that can serve as a signal to take some action. It can be configured so that this Tick Tuple is received at a user-defined frequency. When configured, the execute method on the bolt will receive the Tick Tuple at the given frequency. You need to inspect the tuple to determine whether it is one of these system-emitted Tick Tuples or whether it is a normal tuple. Normal tuples within your topology will flow through the default stream while Tick Tuples are flowing through a system tick stream, making them easily identifiable.

Listing 3.4 - HeatMapBuilder: Step 2 – Configure this bolt to receive a signal at a given frequency

```
@Override
public Map<String, Object> getComponentConfiguration() { #1
  Config conf = new Config();
  conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 60);
  return conf;
}

@Override
public void execute(Tuple tuple,
                    BasicOutputCollector outputCollector) { #2
  if (isTickTuple(tuple)) {
    // . . . take periodic action
  } else {
    Long time = tuple.getLongByField("time");
    LatLng geocode = (LatLng) tuple.getValueByField("geocode");

    Long timeInterval = selectTimeInterval(time);
    List<LatLng> checkins = getCheckinsForInterval(timeInterval);
    checkins.add(geocode);
  }
}

private boolean isTickTuple(Tuple tuple) {
  String sourceComponent = tuple.getSourceComponent();
  String sourceStreamId = tuple.getSourceStreamId();
  return sourceComponent.equals(Constants.SYSTEM_COMPONENT_ID) #3
      && sourceStreamId.equals(Constants.SYSTEM_TICK_STREAM_ID);
}
```

#1 Override this method to Storm to send this bolt a Tick Tuple every 60 seconds.
#2 When we get a tick tuple, do something different For a regular tuple, we take the same action as before.
#3 Tick Tuples are easily identifiable because they are emitted on the system tick stream by system component instead of being emitted by one of our own components on the default stream for our topology.

Emit Frequencies of Tick Tuples

We configured our tick tuples to be emitted at a frequency of every 60 seconds. This does not mean they will be emitted exactly every 60 seconds; it is done on a best effort basis. Tick Tuples that get sent to a bolt are queued behind the other tuples currently waiting to be consumed by the execute method on that bolt. A bolt may not necessarily process the Tick Tuples at the frequency that they get emitted if the bolt is lagging behind to due to high latency in processing it's regular stream of tuples.

Another thing to point out is that tick tuples are configured at the bolt level as evident by using the `getComponentConfiguration` to set it up. The tick tuple in question here will only be sent to instances of this bolt.

Now let's use that Tick Tuple as a signal to select time periods that have passed for which we no longer expect any incoming coordinates and emit them from this bolt so that next bolt down the line can take them on.

Listing 3.5 - HeatMapBuilder: Step 3 – Emit the aggregated heatmap for elapsed time intervals

```
@Override
public void execute(Tuple tuple,
                    BasicOutputCollector outputCollector) {
  if (isTickTuple(tuple)) {
    emitHeatmap(outputCollector); #1
  } else {
    Long time = tuple.getLongByField("time");
    LatLng geocode = (LatLng) tuple.getValueByField("geocode");

    Long timeInterval = selectTimeInterval(time);
    List<LatLng> checkins = getCheckinsForInterval(timeInterval);
    checkins.add(geocode);
  }
}

private void emitHeatmap(BasicOutputCollector outputCollector) {
  Long now = System.currentTimeMillis();
  Long emitUpToTimeInterval = selectTimeInterval(now); #2
  Set<Long> timeIntervalsAvailable = heatmaps.keySet();
  for (Long timeInterval : timeIntervalsAvailable) { #3
    if (timeInterval <= emitUpToTimeInterval) {
      List<LatLng> hotzones = heatmaps.remove(timeInterval);
      outputCollector.emit(new Values(timeInterval, hotzones));
    }
  }
}

private Long selectTimeInterval(Long time) {
  return time / (15 * 1000);
}
```

#1 If we got a tick tuple, interpret that as a signal to see whether there are any heatmaps that can be emitted.
#2 Heatmaps that can be emitted are considered to have checkins that occurred before the beginning of current time-interval. That is why we're passing current time into selectTimeInterval() which will give us the beginning of the current time interval.
#3 For all time intervals we're currently keeping track of if they have elapsed, remove them from in-memory data structure and emit them.

Thread safety

We are collecting coordinates into an in-memory map, but we created it as an instance of a regular HashMap. Storm is highly scalable and there are multiple tuples coming in that get added to this map and also we're periodically removing entries from that map. Is modifying an in-memory data structure like this thread safe?

Yes. It is thread safe because execute() is only processing one tuple at a time. Whether it is your regular stream of tuples or a tick tuple, only one JVM thread of execution will be going through and processing code within an instance of this bolt. So within a given bolt instance, there will never be multiple threads running through it.

Does that mean you never need to worry about thread safety within the confines of your bolt? No, in certain cases you might. While there will only be one Storm thread running through it, you may have threads of your own. For example, instead of using tick tuples, if your spawned a background thread that periodically emits heat maps, then you will need to concern yourself with thread safety. As you will have your own thread and Storm's thread of execution both running through your bolt.

Steps 1, 2 and 3 above sum up the implementation of the `HeatMapBuilder` bolt.

3.4.4 Bolt – Persisting to a data store

We have the end tuples that represent a heat map. At this point, we are ready to persist that data to some data store. Our JavaScript-based web application can read the heat map values from this data store and interact with Google Maps API to build a geographical visualization from these calculated values.

Since we're storing and accessing heat maps based on time interval, it makes sense to use a key/value data model for storage. For this case study, we will use Redis, but any data store that supports the key/value model will suffice (Membase, Memcached, Riak, etc…). We will store the heat maps keyed by time interval with the heat map itself as a JSON representation of list of coordinates. We will use Jedis as a Java client for Redis and the Jackson JSON library for converting the heat map to JSON.

NoSQL and other data stores with Storm

It is outside the scope of this book to examine the different NoSQL and data storage solutions available for working with large data sets, but we want to make sure you start off on the right foot when making your selections with regards to data storage solutions.

It is quite common for people to consider the various options available to them and ask themselves which one of these NoSQL solutions should I pick? This is the wrong approach. Instead, ask yourself questions about the functionality you are implementing and the requirements they impose upon any data storage solution.

You should be asking whether your use case requires a data store that supports:

- Random reads or random writes.
- Sequential reads or sequential writes.
- High read throughput or high write throughput.
- Will the data change or remain immutable once written?
- Storage model suitable for your data access patterns
  - column/column-family oriented
  - key/value
  - documented oriented
  - schema/schemaless
- Whether consistency or availability is most desirable?

Once you determined your mix of requirements, then it's easy to figure out which of the available NoSQL, NewSQL or any other solutions are suitable for you. There's no right NoSQL solution for all problems. There's also no perfect data store for use with Storm. It depends on the use case.

Listing 3.6 - Persistor.java

```
public class Persistor extends BaseBasicBolt {
  private final Logger logger = LoggerFactory.getLogger(Persistor.class);

  private Jedis jedis;
  private ObjectMapper objectMapper;

  @Override
  public void prepare(Map stormConf,
                      TopologyContext context) {
    jedis = new Jedis("localhost"); #1
    objectMapper = new ObjectMapper(); #2
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
```

```
    Long timeInterval = tuple.getLongByField("time-interval");
    List<LatLng> hz = (List<LatLng>) tuple.getValueByField("hotzones");
    List<String> hotzones = asListOfStrings(hz); #3

    try {
      String key = "checkins-" + timeInterval;
      String value = objectMapper.writeValueAsString(hotzones); #4
      jedis.set(key, value); #5
    } catch (Exception e) {
      logger.error("Error persisting for time: " + timeInterval, e); #6
    }
  }

  private List<String> asListOfStrings(List<LatLng> hotzones) {
    List<String> hotzonesStandard = new ArrayList<>(hotzones.size());
    for (LatLng geoCoordinate : hotzones) {
      hotzonesStandard.add(geoCoordinate.toUrlValue());
    }
    return hotzonesStandard;
  }

  @Override
  public void cleanup() {
    if (jedis.isConnected()) {
      jedis.quit(); #7
    }
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // No output fields to be declared #8
  }
}
```

#1 Instantiate Jedis and have it connect to Redis instance running on localhost
#2 Instantiate the Jackson JSON ObjectMapper for serializing our heatmap
#3 Convert the list of LatLng to list of String where each String is of the form (latitude,longitude)
#4 Serialize the list of geo-coordinates (currently in String form) to JSON
#5 Write the heatmap JSON to Redis keyed by the time interval
#6 We are not retrying any database failures as this is an unreliable stream
#7 Close connection to Redis when Storm topology is stopped
#8 As this is the last bolt, no tuples are emitted from it. So no fields need to be declared.

Working with Redis is really simple and it serves as a good store for our use case. But for larger scale applications and datasets, a different data store may be necessary. One thing to note is that because we are working with an unreliable data stream, we are simply logging any errors that may occur while saving to the database. Some errors may be retriable (say, a timeout), and when working with a reliable data stream, we'd consider how to retry them, as you will see in the Chapter 3.

3.4.5 Building a Topology for Running in Local Cluster Mode

We're almost done. We just need wire everything together and run the topology in Local Cluster mode, just like we did for Chapter 1.

Listing 2.7 - HeatMapTopology.java

```
public class HeatmapTopology {
  public static void main(String[] args) { #1
    Config config = new Config(); #2

    StormTopology topology = wireTopology();

    LocalCluster localCluster = new LocalCluster(); #3
    localCluster.submitTopology("local-heatmap", config, topology); #4
  }


  public static StormTopology wireTopology() { #5
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("checkins", new Checkins());
    builder.setBolt("geocode-lookup", new GeocodeLookup())
              .shuffleGrouping("checkins"); #6
    builder.setBolt("heatmap-builder", new HeatMapBuilder())
           .globalGrouping("geocode-lookup"); #7
    builder.setBolt("persistor", new Persistor())
           .shuffleGrouping("heatmap-builder"); #6

    return builder.createTopology();
  }
}
```

#1 A simple java class with main() method is usually used to start the topology
#2 A basic Storm config with no changes to default configuration
#3 Create a local cluster, just like in chapter one
#4 Submit the topology and start running it in Local Cluster mode
#5 Wire in the topology and connect the all bolts and spout together in order. In this particular topology, these components are connected one another in order, in serial fashion
#6 These 2 bolts are connected to their corresponding previous components using shuffle grouping. So these bolts receive their incoming tuples in a random but evenly distributed manner.
#7 We use global grouping to connect HeatMapBuilder to Checkins.

In Chapter 1, we discussed two ways of connecting components within a topology to one another. Shuffle grouping and fields grouping. To recap, we use shuffle grouping to distribute outgoing tuples from one component to next in a manner that's random but evenly spread out. We use fields grouping when we want to ensure tuples with same values for a selected set of fields always go to the same instance of the next bolt.

But we need to send the entire stream of outgoing tuples from `GeocodeLookup` bolt to `HeatMapBuilder` bolt. If different tuples from `GeocodeLookup` end up going to different instances of `HeatMapBuilder`, then we won't be able to group them into time-intervals as they are spread out among different instances of `HeatMapBuilder`. This is where **Global Grouping** comes in. **Global Grouping** will ensure that the entire stream of tuples will go to one specific instance of `HeatMapBuilder`. Specifically, the entire stream will go to the instance of `HeatMapBuilder` with the lowest task id (an id assigned internally by Storm). Now we have

every tuple in one place and we can easily determine which time interval any tuple falls into, grouping them into their corresponding time intervals.

Now we have a working topology. We read check-in data from our spout and in the end we persist the coordinates grouped by time intervals into Redis. Heat map topology is complete. All we have left to do is read the data from Redis using a Javascript application and use the Heat Map Overlay feature of Google Maps API to build the visualization.

Voila. We're done… With the simple implementation anyway. This will run… but will it scale? Will it be fast enough? Let's do some digging and find out.

3.5 Scaling the topology

Let's review where we are so far. We have a working topology for our service that looks about like:

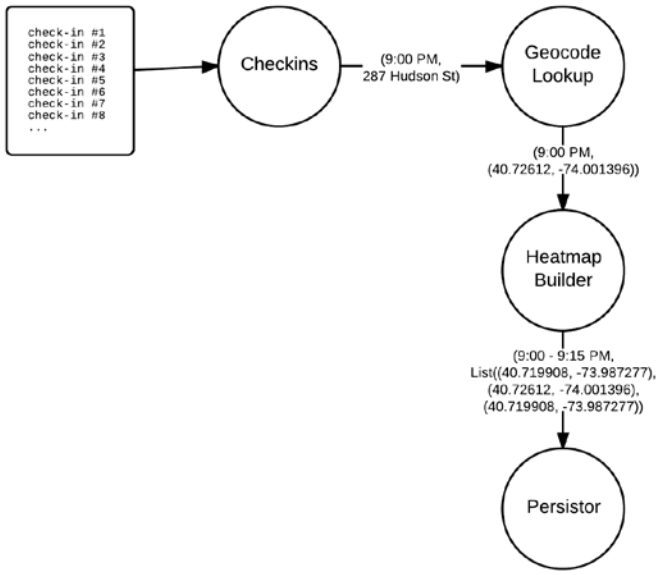


Figure 3.4 Heat map topology

There are problems with it. As it stands right now, this topology operates in a serial fashion, processing one check-in at a time. That isn't web-scale. That's Apple IIe scale. If we were to put this live, everything would back up and we would end up with unhappy customers, an unhappy ops team and probably, unhappy investors. What we need is to process multiple check-ins at a time. We need to introduce parallelism into our topology. One property that makes Storm so alluring is how easy it is to parallelize workflows such as our heat map. Let's take a look at each part of the topology again and discuss how they can be parallelized, starting with check-ins.

What is web-scale?

A system is web-scale when it can grow simply without downtime to cater to the demand brought about by the network effect that is the web. When each happy user tells ten of their friends about our heat map, service and demand increases "exponentially". This increase in demand is "web-scale".

3.5.1 Understanding Parallelism in Storm

Storm has additional primitives that serve as knobs for tuning how it can scale. If you don't touch them, the topology can still work, but all components will run in a more or less linear fashion. This may be fine for topologies that only have a small stream of data flowing through them. For something like the heat map topology that will be receiving data from a large fire-hose, we would want to address the bottlenecks within it. In this section, we will look at two of the primitives that deal with scaling. There are additional primitives for scaling that we will consider later in the next chapter.

PARALLELISM HINTS

We know we are going to need to process many check-ins rapidly. So we are going to want to parallelize the spout that handles check-ins. Figure 3.5 gives you an idea of what part of the topology we are working on here.

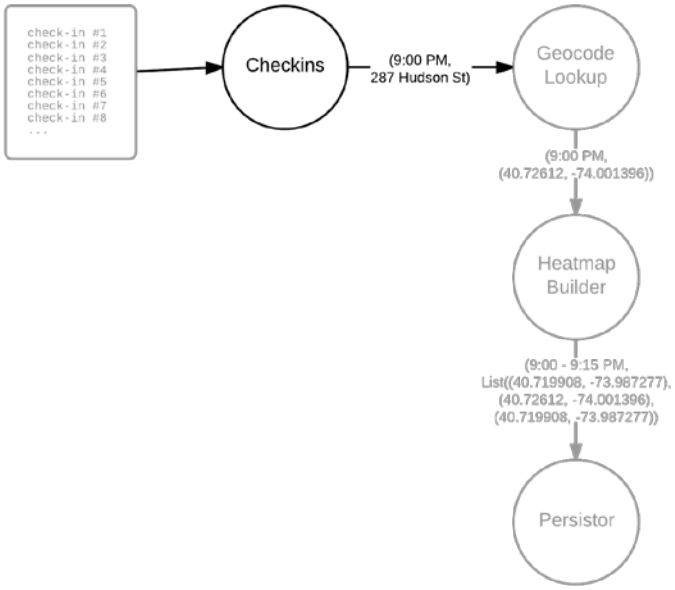


Figure 3.5 Focusing our parallelization changes on the Checkins spout

Storm allows you to provide a parallelism hint when you define any spouts or bolts. In code, this would be transforming

```
builder.setSpout("checkins", new Checkins());
```

to

```
builder.setSpout("checkins", new Checkins(), 4);
```

The additional parameter we provide to `setSpout` is the parallelism hint. That's a bit of a mouthful... "*parallelism hint*". So what is a parallelism hint? For right now, let's just say that the parallelism hint tells Storm how many check-in spouts to create. In our example, this results in four spout instances being created. There's more to it than that but we'll get to that in a bit. Figure 3.6 illustrates the result of this parallelism hint.

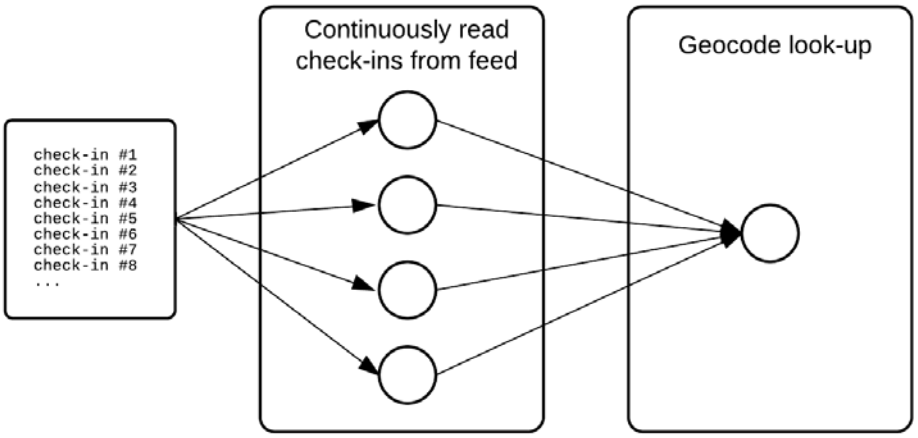


Figure 3.6 Four spout instances emitting tuples to one bolt instance

Great, so now when we run our topology, we should be able to process check-ins four times as fast. Except simply introducing more spouts/bolts into our topology is not enough. Parallelism in a topology is about both input and output. Our spout can now process more check-ins at a time, but the Geocode lookup is still being handled serially. Simultaneously passing four check-ins to a single Geocode instance is not going to work out well. If you look at this diagram again in Figure 3.7, you can also see the geocode bolt has become a bottleneck!

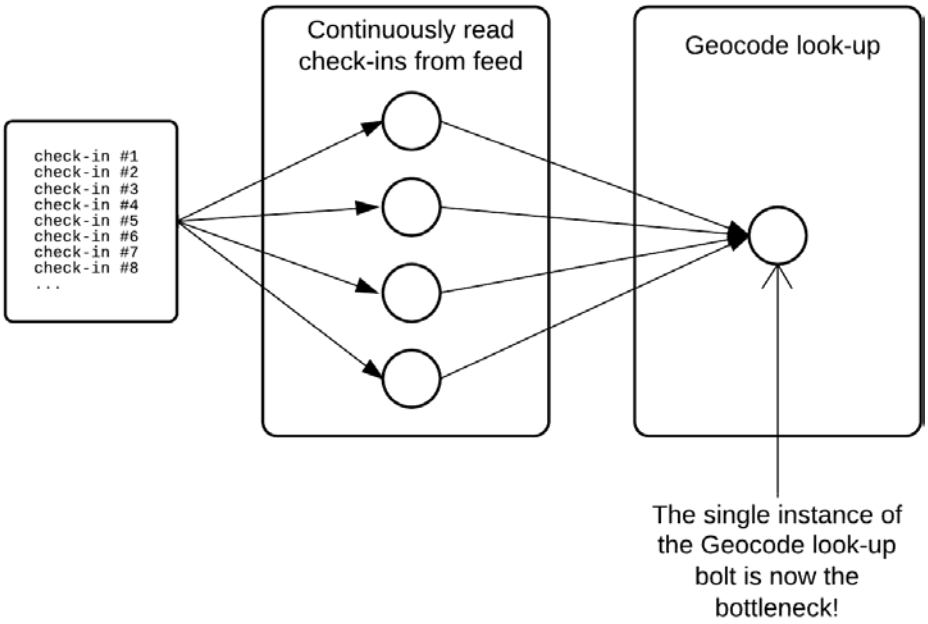


Figure 3.7 The one bolt instance has become a bottleneck

Right now, what we have is akin to a circus clown car routine where many clowns all try to simultaneously pile into a car through the same door. This bottleneck needs to be resolved; let's try parallelizing the geocode look-up bolt as well. We could just parallelize the geocode bolt in the same way we did check-ins. Going from

```
builder.setBolt("geocode-lookup", new GeocodeLookup());
```

to

```
builder.setBolt("geocode-lookup", new GeocodeLookup(), 4);
```

That will certainly help. Now we have one geocode lookup bolt for each check-in handling spout. However, Geocode lookup is going to take a lot longer than receiving a check-in and handing it off to our bolt. So perhaps we can do something more like this:

```
builder.setBolt("geocode-lookup", new GeocodeLookup(), 8);
```

Now if Geocode lookup takes two times as long as check-in handling, tuples should continue to flow through our system smoothly. We can turn:

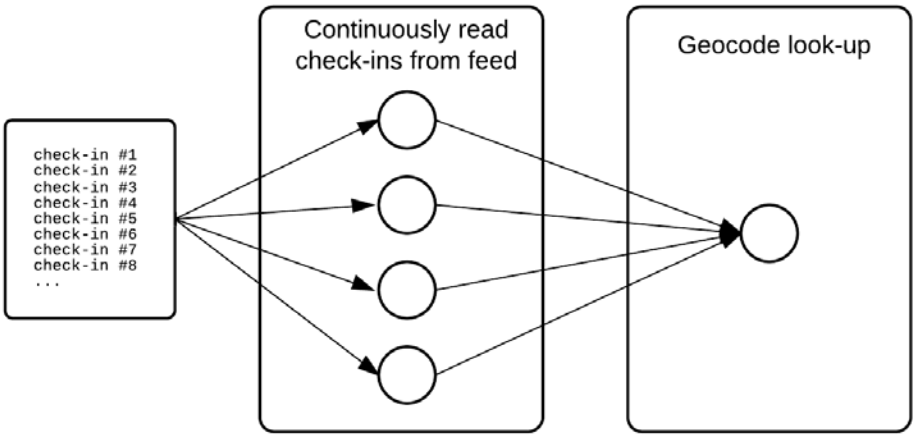


Figure 3.8 Four spout instances emitting tuples to one bolt instance

into

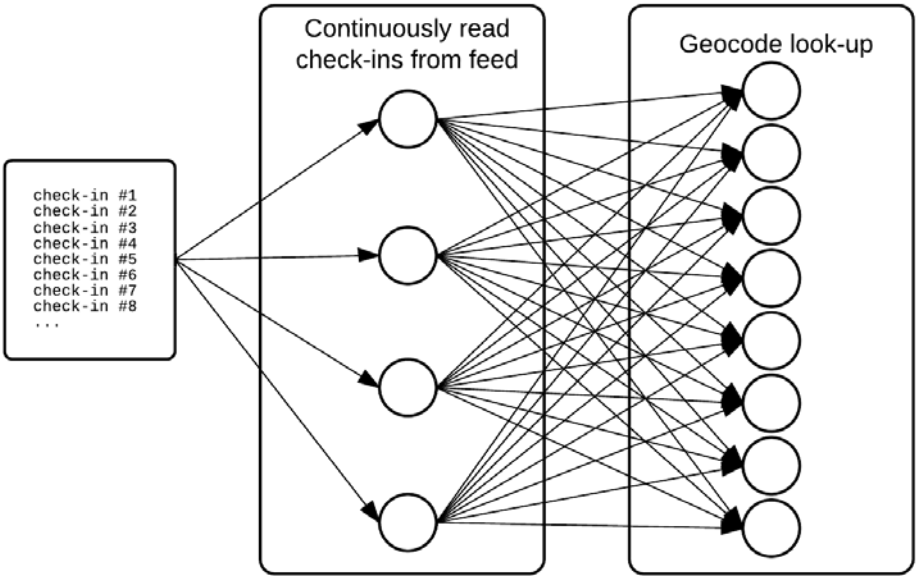


Figure 3.9 Four spout instances emitting tuples to eight bolt instances

We're making progress here but there is something else to think about… What happens as our service becomes more popular? We're going to need to be able to continue to scale to keep pace with our ever expanding traffic without taking our application offline; or at least not taking it offline very often. Luckily Storm provides a way to do that. We loosely defined the

parallelism hint earlier but said there was a little more to it. Well, here we are. That parallelism hint maps into two Storm concepts we haven't covered yet: **executors** and **tasks**.

EXECUTORS AND TASKS

So what are executors and tasks really? These are quite abstract concepts and Figure 3.10 does a good job of mapping them to terms that are a little more familiar.

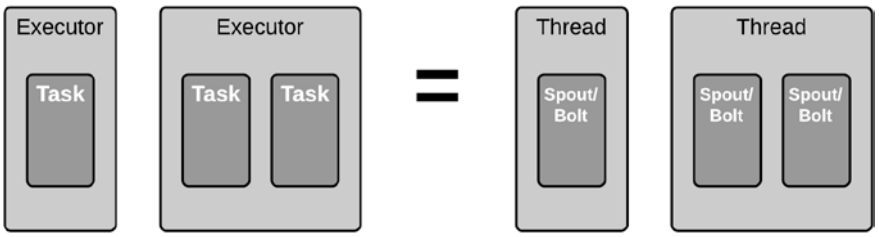


Figure 3.10 Executors and tasks as threads and instances of spouts/bolts

You can draw the following conclusions from Figure 3.10:

1. An executor is a thread of execution within a JVM.
2. A task is an instance of a spout or bolt.

It's really that simple. An executor (thread) has one or more tasks (spout/bolt instances) running on it. This should provide a clearer understanding of what is actually happening when we increase the number of executors and tasks when tuning our topology with the parallelism hints. Let's get back to our code and revisit what this means.

Setting the parallelism hint to eight as we did with `GeocodeLookup` is telling Storm to create eight executors (threads), running eight tasks (instances) of `GeocodeLookup`. This is seen with the following code:

```
builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
```

By default, the parallelism hint is setting both the number of executors and tasks to the same value. We can override the number of tasks with the `setNumTasks` method as follows:

```
builder.setBolt("geocode-lookup", new GeocodeLookup(), 8).setNumTasks(8)
```

Why provide the ability to set the number of tasks to something different than the number of executors? Before we answer this question, let's take a step back and refresh how we got here. We were talking about how we will want to scale our heat map in the future without taking it offline. What is the easiest way to do this? The answer: increase the parallelism. Fortunately Storm provides a useful feature that allows us to increase the parallelism of a running topology by dynamically increasing the number of executors (threads) (more on how this is exactly done in a later chapter).

What does this mean for our `GeocodeLookup` bolt with eight instances being run across eight threads? Well, each of those instances will be spending most of their time waiting on

network I/O. We suspect that this means GeocodeLookup is going to be a source of contention in the future and will need to be scaled up. We can allow for this possibility with:

```
builder.setBolt("geocode-lookup", new GeocodeLookup(), 8).setNumTasks(64)
```

Now we have 64 tasks (instances) of GeocodeLookup running across eight executors (threads). As we need to increase the parallelism of GeocodeLookup, we can keep increasing the number of executors up to a maximum of 64 without stopping our topology. We repeat, without stopping the topology. As we mentioned earlier, we'll get into the details of how to do this in a later chapter, but the key point to understand here is the number of executors (threads) can be dynamically changed in a running topology.

Storm breaks parallelism down into two distinct concepts of executors and tasks to deal situations like we have with our geocode bolt. In order to understand why, let's go back to the definition of a fields grouping:

> A fields grouping is a type of stream grouping where tuples with the same value for a particular field name are always emitted to the same instance of a bolt.

Within that definition lurks our answer. Fields groupings work by consistently hashing tuples across a set number of bolts. In order to keep keys with the same value going to the same bolt, the number of bolts can't change. If it did, tuples would start going to different bolts. That would defeat the purpose of what we were trying to accomplish with a fields grouping.

It was easy to configure the executors and tasks on Checkins spout and Geocode bolt in order to scale them at a later point in time. Sometimes however, parts of our design simply will not work well for scaling. Let's look at that next.

3.5.2 Adjusting the topology to address bottlenecks inherent within design

HeatMapBuilder is up next. Earlier we hit a bottleneck on GeocodeLookup when we increased the parallelism hint on Checkins spout. But we were able to address this easily by increasing the parallelism on the GeocodeLookup bolt accordingly. We can't quite do that here. It does not make sense to increase the parallelism on HeatMapBuilder as it is connected to the previous bolt using Global Grouping. Since global grouping dictates that every tuple goes to one specific instance of HeatMapBuilder, increasing parallelism on it does not have any effect as only once instance will be actively working on the stream. There's a bottleneck that is inherent in the design of our topology.

This is the downside of using global grouping. With global grouping, you're trading your ability to scale and introducing an intentional bottleneck for being able to see the entire stream of tuples in one specific bolt instance.

So what can we do? Is there no way we can parallelize this step in our topology? If we cannot parallelize this bolt, then it makes little sense to parallelize the bolts that follow. This

is the choke point. It cannot be parallelized with the current design. When we come across a problem like this the best approach is to take a step back and see what we can change about the topology design to achieve our goal.

Current topology wiring from HeatMapTopology.java

```
public static StormTopology wireTopology() {
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("checkins", new Checkins(), 4);
  builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
         .setNumTasks(64)
            .shuffleGrouping("checkins");
  builder.setBolt("heatmap-builder", new HeatMapBuilder())
         .globalGrouping("geocode-lookup");
  builder.setBolt("persistor", new Persistor())
         .shuffleGrouping("heatmap-builder");

  return builder.createTopology();
}
```

The reason why we can't parallelize the `HeatMapBuilder` is because all tuples need to go in to the same instance. All tuples need to go to the same instance because we need to ensure that every tuple that falls into any given time-interval can be grouped together. So if we can ensure that every tuple that falls into given time-interval goes into the same instance, we can have multiple instances of `HeatMapBuilder`.

Right now, we use the `HeatMapBuilder` bolt to do two things:

1. Determine which time-interval a given tuple falls into
2. Group by tuples by time-interval

If we can separate these two actions into separate bolts, then we can get closer to our goal. Let's look at the part of the `HeatMapBuilder` bolt that determines which time-interval a tuple falls into.

Determining time-interval for a tuple in HeatMapBuilder.java

```
public void execute(Tuple tuple,
                    BasicOutputCollector outputCollector) {
  if (isTickTuple(tuple)) {
    emitHeatmap(outputCollector);
  } else {
    Long time = tuple.getLongByField("time");
    LatLng geocode = (LatLng) tuple.getValueByField("geocode");

    Long timeInterval = selectTimeInterval(time);
    List<LatLng> checkins = getCheckinsForInterval(timeInterval);
    checkins.add(geocode);
  }
}

private Long selectTimeInterval(Long time) {
  return time / (15 * 1000);
}
```

HeatMapBuilder receives a check-in time and a geo coordinate from GeocodeLookup. Let's move this simple task of extracting the time-interval out of tuple emitted by GeocodeLookup into another bolt. This bolt, let's call it TimeIntervalExtractor, can emit a time-interval and a coordinate that can be picked up by HeatMapBuilder instead.

TimeIntervalExtractor

```
public class TimeIntervalExtractor extends BaseBasicBolt {
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("time-interval", "geocode"));
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    Long time = tuple.getLongByField("time");
    LatLng geocode = (LatLng) tuple.getValueByField("geocode");

    Long timeInterval = time / (15 * 1000); #1
    outputCollector.emit(new Values(timeInterval, geocode));
  }
}
```

#1 This will calculate the time-interval and emit that time-interval and geo coordinate (instead of time and geo coordinate) to be picked up HeatMapBuilder.

Update execute() in HeatMapBuilder.java to use the precalculated time-interval

```
  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    if (isTickTuple(tuple)) {
      emitHeatmap(outputCollector);
    } else {
      Long timeInterval = tuple.getLongByField("time-interval");
      LatLng geocode = (LatLng) tuple.getValueByField("geocode");

      List<LatLng> checkins = getCheckinsForInterval(timeInterval);
      checkins.add(geocode);
    }
  }
```

So now we have,

1. Checkins spout – emits time, address
2. GeocodeLookup bolt – emits time, geo coordinate
3. TimeIntervalExtractor bolt – emits time-interval, geo coordinate
4. HeatMapBuilder bolt – emits time-interval, a list of grouped geo coordinates
5. Persistor bolt – emits nothing as it's the last bolt in our topology

Figure 3.11 shows our updated topology design.

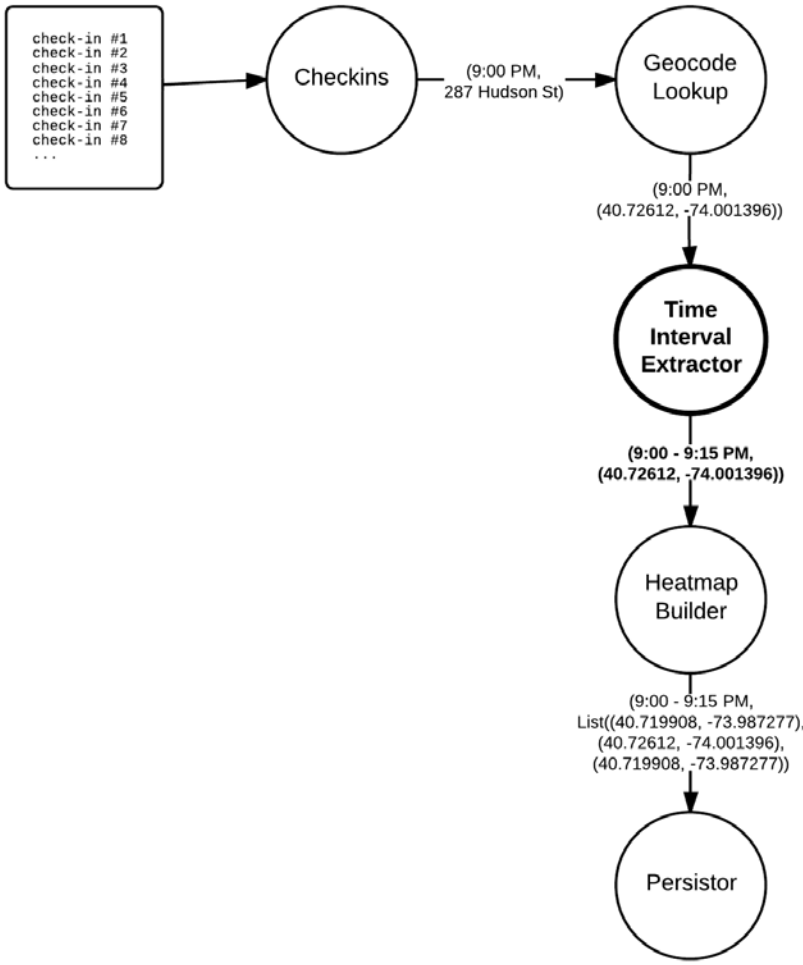


Figure 3.11 Updated topology with the TimeIntervalExtractor bolt

Now when we wire `HeatMapBuilder` to `TimeIntervalExtractor` we don't need to use *Global Grouping*. Now that we have time-interval pre-calculated, we just need to ensure the same `HeatMapBuilder` bolt instance receives all values for given time-interval. It does not matter if different time-intervals go to different instances. We can use *Fields Grouping* for this. Fields Grouping lets us group values by a specified field and send all tuples that arrive with that given value to a specific bolt instance. What we have done is segment the tuples into time-intervals, and send each segment into different `HeatMapBuilder` instances, thereby allowing us to achieve parallelism by running the segments in parallel. Figure 3.12 shows the updated stream groupings between our spout and bolts.

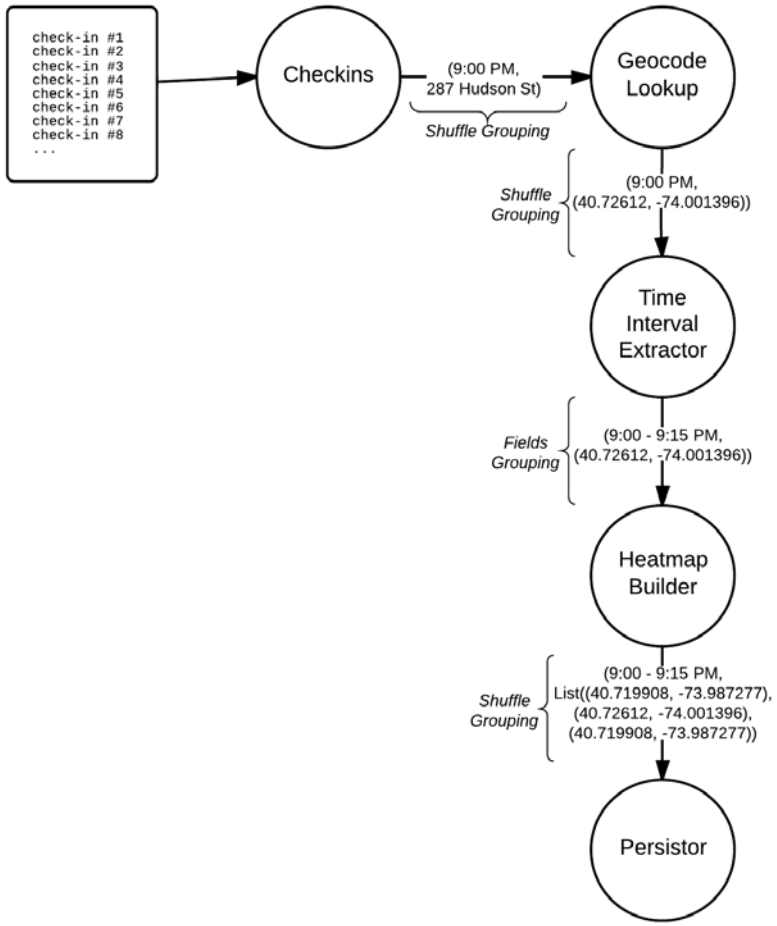


Figure 3.12 Updated topology stream groupings

Updated topology wiring of HeatMapTopology.java with new bolt

```
public static StormTopology wireTopology() {
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("checkins", new Checkins(), 4);
  builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
         .setNumTasks(64)
            .shuffleGrouping("checkins");
  builder.setBolt("time-interval-extractor", new TimeIntervalExtractor())
         .shuffleGrouping("geocode-lookup"); #1
  builder.setBolt("heatmap-builder", new HeatMapBuilder())
         .fieldsGrouping("time-interval-extractor", #2
                         new Fields("time-interval"));
  builder.setBolt("persistor", new Persistor())
         .shuffleGrouping("heatmap-builder");
```

```
    return builder.createTopology();
  }
```

#1 New bolt is being inserted between GeocodeLookup and HeatMapBuilder and it will bind to GeocodeLookup with shuffle grouping as random assignment of tuples is ok.
#2 HeatMapBuilder can now receive tuples from the new bolt and we can change it to use fields grouping to so that we can parallelize it next

Global Grouping

We scaled this bolt by replacing *Global Grouping* with *Fields Grouping* after some minor design changes. So does global grouping fit well with any real world scenarios where we actually need scale? Global grouping should not be discounted; it does serve a useful purpose when deployed at the right junction.

In this case study, we used global grouping at the point of aggregation (grouping coordinates by time interval). When used at the point of aggregation it does not indeed scale as we saw as we are forcing it to crunch a larger data set. But if we were to use to global grouping post-aggregation, then it would be dealing with a smaller stream of tuples and you would not have as high a need for scale as you would pre-aggregation.

If you need to see the entire stream of tuples, then global grouping is highly useful. What you would need to do first is aggregate them in some manner (shuffle grouping for randomly aggregating sets of tuples or fields grouping of aggregating a selected sets of tuples) and then use global grouping on the aggregation to get the complete picture.

```
builder.setBolt(“aggregation-bolt”, new AggregationBolt(), 10)
   .shuffleGrouping(“previous-bolt”);
builder.setBolt(“world-view-bolt”, new WorldViewBolt())
   .globalGrouping(“aggregation-bolt”);
```

`AggregationBolt` in this case can be scaled and it will trim down the stream into a smaller set. Then `WorldViewBolt` can look at the complete stream by using global grouping on already aggregated tuples coming from `AggregationBolt`. We don’t have to scale `WorldViewBolt` as it’s looking at a smaller data set.

Parallelizing `TimeIntervalExtractor` is simple. To start, we can give it the same level of parallelism as the `Checkins` spout as there is no waiting on an external service like with the `GeocodeLookup` bolt.

```
builder.setBolt("time-interval-extractor", new TimeIntervalExtractor(), 4)
    .shuffleGrouping("geocode-lookup");
```

Next up, we can clear our troublesome choke point in the topology.

```
builder.setBolt("heatmap-builder", new HeatMapBuilder(), 4)
       .fieldsGrouping("time-interval-extractor",
                       new Fields("time-interval"));
```

Finally we address the Persistor. This is similar to GeocodeLookup in the sense that we expect we'll need to scale it later on. So we will need more tasks than executors for the reasons we covered under GeocodeLookup earlier.

```
builder.setBolt("persistor", new Persistor(), 1)
    .setNumTasks(4)
    .shuffleGrouping("heatmap-builder");
```

So it looks like we are done with scaling this topology… or are we? We have configured every component (i.e. every spout and bolt) for parallelism within the topology. While each bolt or spout maybe configured for parallelism, this does not necessarily mean it will run at scale. Let's see why.

3.5.3 Adjusting the topology to address bottlenecks inherent within data stream

We have parallelized every component within the topology, and this is in line with the technical definition of how every grouping (shuffle grouping, field grouping, global grouping) we use affects the flow of tuples within our topology. Unfortunately, it is still not effectively parallel.

While we were able to parallelize the HeatMapBuilder with the changes from the last section, what we forgot to consider is how the nature of our data stream affects parallelism. We are grouping the tuples that flow through the stream into segments of 15 seconds, and that is the source of our problem. So for a given 15 second window, all tuples that fall into that window will go through one instance of the HeatMapBuilder. While with the design changes we made HeatMapBuilder became technically parallelizable, it is effectively not parallel yet. The shape of the data stream that flows through your topology can hide problems with scaling that may be hard to spot like this. It is wise to always question the impact of how data flow through your topology.

How can we parallelize this? We were right to group by time interval as that is the basis for our heat map generation. What we need is an additional level of grouping under the time interval. I think we can refine our higher-level solution so that we are delivering heat maps by time interval by city. When we add an additional level of grouping on city, we will have multiple data flows for a given time interval and they may flow through different instances of the HeatMapBuilder.

Emit city for GeoCoordinate along with it from GeocodeLookup.java

```
public class GeocodeLookup extends BaseBasicBolt {
  private Geocoder geocoder;

  @Override
  public void declareOutputFields(OutputFieldsDeclarer fieldsDeclarer) {
    fieldsDeclarer.declare(new Fields("time", "geocode", "city")); #1
  }

  @Override
  public void prepare(Map config,
```

```
                      TopologyContext context) {
    geocoder = new Geocoder();
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    String address = tuple.getStringByField("address");
    Long time = tuple.getLongByField("time");

    GeocoderRequest request = new GeocoderRequestBuilder()
        .setAddress(address)
        .setLanguage("en")
        .getGeocoderRequest();
    GeocodeResponse response = geocoder.geocode(request);
    GeocoderStatus status = response.getStatus();
    if (GeocoderStatus.OK.equals(status)) {
      GeocoderResult firstResult = response.getResults().get(0);
      LatLng latLng = firstResult.getGeometry().getLocation();
      String city = extractCity(firstResult); #2
      outputCollector.emit(new Values(time, latLng, city));
    }
  }

  private String extractCity(GeocoderResult result) {
    for (GeocoderAddressComponent component : result.getAddressComponents()) {
      if (component.getTypes().contains("locality")) return
component.getLongName();
    }
    return "";
  }
}
```

#1 Add city as an additional field that will be emitted from this bolt
#2 Extract city name from already available Geocoded result

Pass city field along in TimeIntervalExtractor.java

```
public class TimeIntervalExtractor extends BaseBasicBolt {
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("time-interval", "geocode", "city"));
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    Long time = tuple.getLongByField("time");
    LatLng geocode = (LatLng) tuple.getValueByField("geocode");
    String city = tuple.getStringByField("city");

    Long timeInterval = time / (15 * 1000);
    outputCollector.emit(new Values(timeInterval, geocode, city));
  }
}
```

Updated topology wiring of HeatMapTopology.java to add second level grouping

```
public static StormTopology wireTopology()
{
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("checkins", new Checkins(), 4);
  builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
         .setNumTasks(64)
            .shuffleGrouping("checkins");
  builder.setBolt("time-interval-extractor", new TimeIntervalExtractor(), 4)
         .shuffleGrouping("geocode-lookup");
  builder.setBolt("heatmap-builder", new HeatMapBuilder(), 4)
         .fieldsGrouping("time-interval-extractor",
                         new Fields("time-interval", "city")); #1
  builder.setBolt("persistor", new Persistor(), 1)
         .setNumTasks(4)
         .shuffleGrouping("heatmap-builder");

  return builder.createTopology();
}
```

#1 Secondary grouping being added allows effective parallelization of HeatMapBuilder

So now we have topology that is not only technically parallelized but also effectively running in parallel fashion.

We have covered the basics of parallelizing a Storm topology. The approach we followed here is based on making educated guesses driven by our understanding of how each topology component works. There is more work that can be done on parallelizing this topology including additional parallelism primitives and approaches to achieving optimal tuning based on observed metrics. We will visit them at appropriate points throughout the book. In this chapter, we built up the understanding of parallelism needed to properly design a Storm topology as the ability to scale a topology depends heavily on the makeup of a topology's underlying component breakdown and design.

3.6 Topology Design Paradigms

Let's recap how we designed the heat map topology.

1. We examined our data stream, and determined what our input tuples are based on what we start with. Then we decided the resulting tuples we need to end up with in order to achieve our goal (the end tuples).
2. We created a series of operations (as bolts) that transform the input tuples into end tuples.
3. We carefully examined each operation to understand its behavior and scaled it by making educated guesses based on our understanding of its behavior (by adjusting its executors/tasks).
4. At points of contention where we could no longer scale, we re-thought our design and refactored the topology into scalable components.

This is a good approach to topology design. It is quite common for most people to fall into the trap of not having scalability in mind when creating their topologies. If we don't do this early on and leave scalability concerns for later on, the amount of work you have to do refactor or re-design your topology will increase by an order of magnitude.

> "premature optimization is the root of all evil"
>
> - Donald Knuth

As engineers we are fond of stating this quote from Donald Knuth whenever we talk about performance considerations early on. This is indeed true in most cases, but let us look at the complete quote to give us more context to what Dr. Knuth was really trying to say (rather than the sound bite we engineers normally use to make our point).

> "We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil"
>
> - Donald Knuth

We are not trying to achieve small efficiencies… we are working with "Big Data". Every efficiency enhancement we make counts. One minor performance block can be the difference in not achieving the performance SLA you need when working with large data sets. If you are building a racecar, you need to keep performance in mind starting on day one. You cannot refactor your engine to improve it later if it was not built for performance from the ground up. So steps three and four are quite critical and important pieces in designing a topology.

The only caveat here is lack of knowledge about the problem domain. If your knowledge about problem domain is limited, then it might work against you to try scale it too early. When we say knowledge about the problem domain, what we are referring to is both the nature of data that is flowing through your system as well as the inherent choke points within your operations. It is always ok to defer scaling concerns until you have a good understanding of it. Similar to building an expert system, when you have a true understanding of the problem domain, you might have to scrap your initial solution and start over.

3.6.1 Design by Breakdown into Functional Components

Let's observe how we broke down the series of operations within our topology (Figure 3.3).

Topology components of HeatMapTopology.java

```
public static StormTopology wireTopology() {
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("checkins", new Checkins(), 4);
  builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
         .setNumTasks(64)
            .shuffleGrouping("checkins");
  builder.setBolt("time-interval-extractor",
                  new TimeIntervalExtractor(),
                  4)
         .shuffleGrouping("geocode-lookup");
  builder.setBolt("heatmap-builder", new HeatMapBuilder(), 4)
         .fieldsGrouping("time-interval-extractor",
                         new Fields("time-interval"));
  builder.setBolt("persistor", new Persistor(), 1)
         .setNumTasks(4)
         .shuffleGrouping("heatmap-builder");

  return builder.createTopology();
}
```

We decomposed the topology makeup into separate bolts by giving each bolt a very specific responsibility. This is in line with the *principle of single responsibility*. We encapsulated a very specific responsibility within each bolt and everything within each bolt is narrowly aligned with its responsibility and nothing else. In other words, each bolt represents a functional whole.

There is a lot of value in this approach to design. By giving each bolt a single responsibility, it makes very easy to work with a given bolt in isolation. It also makes it easy to scale a single bolt without interference from the rest of the topology as parallelism is tuned at the bolt level. Whether it's scaling or troubleshooting a problem, when you can zoom in and focus your attention on a single component, the productivity gains to be had from that will allow you to reap the benefits of the effort spent on designing your components in this manner.

3.6.2 Design by Breakdown into Components at points of Re-partition

There's a slightly different approach to breaking down a problem into its constituent parts. It provides a marked improvement in terms of performance over the approach of breaking down into functional components we discussed earlier. With this pattern, instead of decomposing the problem into its simplest possible functional components, we think in terms of separation points (or join points) between the different components. In other words, we think of the points of connection between the different bolts. In Storm, the different stream groupings are markers between different bolts (as the groupings define how the outgoing tuples from one bolt are distributed to the next).

At these points, the stream of tuples flowing through the topology gets re-partitioned. During a **stream re-partition**, the way tuples are distributed changes. That is in fact the functionality of a stream grouping.

Repartition operations from HeatMapTopology.java

```
public static StormTopology wireTopology() {
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("checkins", new Checkins(), 4);
  builder.setBolt("geocode-lookup", new GeocodeLookup(), 8)
         .setNumTasks(64)
         .shuffleGrouping("checkins");
  builder.setBolt("time-interval-extractor",
                  new TimeIntervalExtractor(),
                  4)
         .shuffleGrouping("geocode-lookup");
  builder.setBolt("heatmap-builder", new HeatMapBuilder(), 4)
         .fieldsGrouping("time-interval-extractor",
                         new Fields("time-interval"));
  builder.setBolt("persistor", new Persistor(), 1)
         .setNumTasks(4)
         .shuffleGrouping("heatmap-builder");

  return builder.createTopology();
}
```

With this pattern of topology design, we strive to minimize the number of repartitions within a topology. Every time there is a re-partitioning, tuples will be sent from one bolt to another across the network. This is an expensive operation due to a number of different reasons.

1. The topology operates within a distributed cluster. When tuples get emitted they may travel across the cluster and this may incur network overhead.
2. With every emit, a tuple will need to be serialized and de-serialized at the receiving point.
3. The higher the number of partitions, the higher number of resources needed. Each bolt will require a number of executors and tasks and a queue in front of for all the incoming tuples.

The moving parts of a Storm Cluster

We will discuss the makeup of a Storm cluster and the internals that support a bolt in later chapters.

So for our topology, what can we do to minimize the number of partitions within our topology? We will have to collapse a few bolts together to do that. The way you figure out

which bolts can be collapsed together is by figuring out what is different about each functional component that makes it a need its own bolt (and the resources that come with a bolt).

1. `Checkins` (spout) – 4 executors (reads a file)
2. `GeocodeLookup` – 8 executor, 64 tasks (hits an external service)
3. `TimeIntervalExtractor` – 4 executors (internal computation – transforms data)
4. `HeatMapBuilder` – 4 executors (internal computation – aggregates tuples)
5. `Persistor` – 1 executors, 4 tasks (writes to a data store)

Analysis:

- `GeocodeLookup` and `Persistor` interact with an external entity and the time spent waiting on interactions with that external entity will dictate the way executors and tasks are allocated to these two bolts. It's unlikely that we will able to coerce the behavior of these bolts to fit within another. Maybe something else might be able to fit within the resources necessary for one of these two.
- `HeatMapBuilder` does the aggregation of geo coordinates by time interval and city. It is somewhat unique compared to others as it buffers data in memory and you cannot proceed to the next step until the time interval has elapsed. It's peculiar enough that collapsing it with another will require careful consideration.
- `Checkins` is a spout and normally you would not modify a spout to contain operations that involve computation. Also since the spout is responsible for keeping track of the data that has been emitted, rarely would we perform any computation within one. However, certain things around adapting the initial tuples (parsing, extracting, converting, etc..) do fit within the responsibilities of a spout.
- That leaves `TimeIntervalExtractor`. This is quite simple as all it does is transform a "time" entry into a "time interval". We extracted it out of `HeatMapBuilder` as we needed to know the time interval prior to `HeatMapBuilder` so that we can group by the time interval. This allowed us to scale the `HeatMapBuilder` bolt. Work done by `TimeIntervalExtractor` can technically happen at any point before the `HeatMapBuilder`.
    a. If we merge `TimeIntervalExtractor` with `GeocodeLookup`, it will need to fit within resources allocated to `GeocodeLookup`. While they have different resource configurations, the simplicity of `TimeIntervalExtractor` will allow it to fit within resources allocated to `GeocodeLookup`. On a purely idealistic sense, they also fit together as both operations are data transformations (time -> time-interval and address -> geo coordinate). One of them is incredibly simple and the other requires network overhead of hitting an external service.
    b. Can we merge `TimeIntervalExtractor` with `Checkins` spout? They have the exact same resource configurations. Also transforming a "time" into a

> "time-interval" is one of few types of operations from a bolt that can make sense within a spout. The answer is a resounding yes. This begs the question whether GeocodeLookup can also be merged with Checkins spout? While GeocodeLookup is also a data transformer, it is a much more heavyweight computation as it depends on an external service, meaning it does not fit within the type of actions that should happen in a spout.

So should we merge TimeIntervalExtractor with GeocodeLookup or Checkins spout? From an efficiency perspective, either will do, and that is the right answer. The authors would merge it with the spout as they have a preference for keeping external service interactions untangled with much simpler tasks like TimeIntervalExtractor. We would leave it as an exercise to the reader to make the needed changes in our topology to make this happen.

You might question why we chose not to merge HeatMapBuilder with Persistor. HeatMapBuilder emits the aggregated geo coordinates periodically (whenever it receives a tick tuple) and at the point of emitting, it can be modified to write the value to the data store instead (the responsibility of Persistor). While this makes sense conceptually, it changes the observable behavior of the combined bolt. The combined HeatMapBuilder/Persistor behaves very differently on the two types of tuples it receives. The regular tuple from the stream will perform with very low latency while the tick tuple for writing to data store will have comparably higher latency. If we were to monitor and gather data about the performance of this combined bolt, it will be very hard to isolate the observed metrics and make intelligent decisions on how to tune it further. This unbalanced nature of latency makes it very inelegant.

Designing a topology by considering points of re-partition of stream and trying to minimize those will give you the most efficient usage of resources with a topology makeup that has a higher likelihood of performing with low latency.

3.6.3 Simplest functional components vs Lowest number of re-partitions

We discussed two approaches to topology design. Which one is better? Having the lowest number of re-partitions will provide the best performance as long as careful consideration is given to what kind of operations can be grouped into one bolt.

Usually it is not one or the other. As a Storm beginner, you should always start by designing the simplest functional components as it allows you to reason about different operations very easily. Also if you start with more complex components tasked with multiple responsibilities, it is much harder to break down into simpler components if your design is wrong.

You can always start with simplest functional components and then advance towards combining different operations together to reduce the number of partitions. It is much harder to go the other way around. As you gain more experience with working with Storm and develop intuition around topology design, you will be able start with the lowest number of re-partitions from the get-go.

3.7 Summary

In this chapter we learned:

- How to take a problem and break it down into constructs that fit within a Storm topology
- How to take a topology that runs in a serial fashion and introduce parallelism
- How to spot problems in our design and refine and refactor
- The importance of paying attention to the effects of the data stream on the limitations it imposes on the topology
- Two different approaches to topology design and the delicate balance between the two

These design guidelines serve as best practices for building Storm topologies. Later on in the book, we will see why these design decisions aid us greatly in tuning Storm for optimal performance.

4

Creating Robust Topologies

This chapter covers:

- Guaranteed message processing
- Fault tolerance
- The components of a Storm cluster
- Parallelism within a Storm cluster

So far, we have defined many of Storm's core concepts. Along the way, we have also implemented two separate topologies, each of which runs in a local cluster. This chapter is no different in that we will be designing and implementing another topology for a new scenario. However, the problem we are solving has stricter requirements around guaranteeing tuples are processed and fault tolerance. In order to meet these requirements, we will introduce some new concepts and deploy our topology to a Storm cluster (time to move on from the local mode kiddie pool). This will lead to a more in-depth discussion on parallelism that will build on the parallelism concepts introduced in Chapter 2.

4.1 Requirements on Reliability

In the previous chapter, our heat map application needed to quickly process a large amount of time sensitive data. Further, merely sampling a portion of that data could provide us with what we needed: an approximation of the popularity of establishments within a given geographic area right now. If we failed to process a given tuple within a short time window, it lost its value. The heat map was all about right now. We didn't need to guarantee that each message was processed: "most" was good enough.

There are domains, however, where this is strictly unacceptable; each tuple is sacred. In these scenarios, we need to guarantee that each and every one is processed. **Reliability** is more important than timeliness here. If we have to keep retrying a tuple for 30 seconds or 10 minutes or an hour (or up to some threshold that makes sense), it has just as much value in our system as it did when we first tried. There is a need for reliability.

Storm provides us with the ability to guarantee that each tuple is processed. This serves as a reliability measure that we can count on to ensure accurate implementation of functionality. On a very high level, Storm provides reliability by keeping track of which tuples get successfully processed and which ones do not and then replaying the ones that have failed until they succeed.

PIECES OF THE PUZZLE FOR SUPPORTING RELIABILITY

Storm has many moving parts that need to come together in order to deliver reliability.

- A reliable data source with a corresponding reliable spout
- An anchored tuple stream
- A topology that acknowledges each tuple as it gets processed or notifies of failure
- A fault tolerant Storm cluster infrastructure

In this chapter, we will look at how all of these components fall into place to enable reliability.

4.2 Problem definition: A Credit Card Authorization System

When you are thinking about using Storm to solve a problem within your domain, take time to think about what guarantees you need to have around processing; it's an important part of "thinking in Storm". So, let's dive into a problem that has a reliability requirement.

Imagine for a moment, that you run a large e-commerce site that deals with shipping physical goods to people. You know that the vast majority of orders placed on your site authorize for payment successfully and only a small percentage get declined. Traditionally in e-commerce, the higher the number of steps that your user needs to take to place an order, the higher the risk of losing the sale. When you are doing billing at the time an order is placed, you are losing business. Handling billing as a separate, "offline" operation improves conversions and thereby directly affects your bottom line. You also need this offline billing process to scale well to support peak seasons such as the holidays (think Amazon) or even flash sales (think Gilt).

This is a scenario that requires reliability. Each order has to be authorized before it is shipped. If we encounter a problem during our attempts to authorize, we should retry. In short, we need guaranteed message processing.

Other case studies requiring reliability

Add additional case studies here

4.2.1 A conceptual solution with retry characteristics

What does this system look like? Well, it deals solely with authorizing credit cards related to orders that have already been placed. Our system does not actually deal with customers placing orders; that happens earlier in the pipeline.

ASSUMPTIONS ON UPSTREAM AND DOWNSTREAM SYSTEMS

- The same order will never be sent to our system more than once. This is guaranteed by some upstream system that handles the placing of customer orders.
- The upstream system that places orders will put the order on a queue and our system will pull the order off of the queue so it can be authorized.
- A separate downstream system will handle a processed order, either fulfilling the order if the credit card was authorized or notifying the customer of a denied credit card.

With these assumptions in hand, we can move forward with a design that is limited in scope but maps well to the Storm concepts we wish to cover.

FORMATION OF A CONCEPTUAL SOLUTION

We will begin with how orders flow through our system. The following steps are taken when the credit card for an order must be authorized:

1. Pull the order off the message queue.
2. Determine if the order has been marked as "ready-to-ship" and do one of two things:
    a. If the order has been marked as "ready-to-ship", do nothing.
    b. If the order has not been marked as "ready-to-ship", continue to step 3.
3. Attempt to authorize the credit card by calling an external credit card authorization service.
    a. If the service call succeeded, update the order status.
    b. If it fails, we can try again later
4. Notify a separate downstream system that the order has been processed.

RETRY/REPLAY CHARACTERISTICS

Is the check in step #2 is even necessary since there is a precondition on the same order never being sent to our system more than once? It is indeed true that the upstream system will not send the same order twice. But within the context of our system itself, any of these steps can fail and will cause that order to be retried. So step 2 is our built in safety mechanism for this. After we are logically done processing a given order/tuple, we update its status. It is quite possible that we might see that tuple again. As you will see in a bit, we do this by placing a call out to an external system of record to verify we haven't already processed this order. In some systems, there might not be any meaningful impact of processing a tuple more than once in which case we could just ignore this entirely. However,

in our case, we'd repeatedly authorize charges against someone's credit card. We'd like to avoid that as much as possible and keep our customer satisfaction high while keeping customer service costs low. These steps can be visualized as shown in figure 4.1

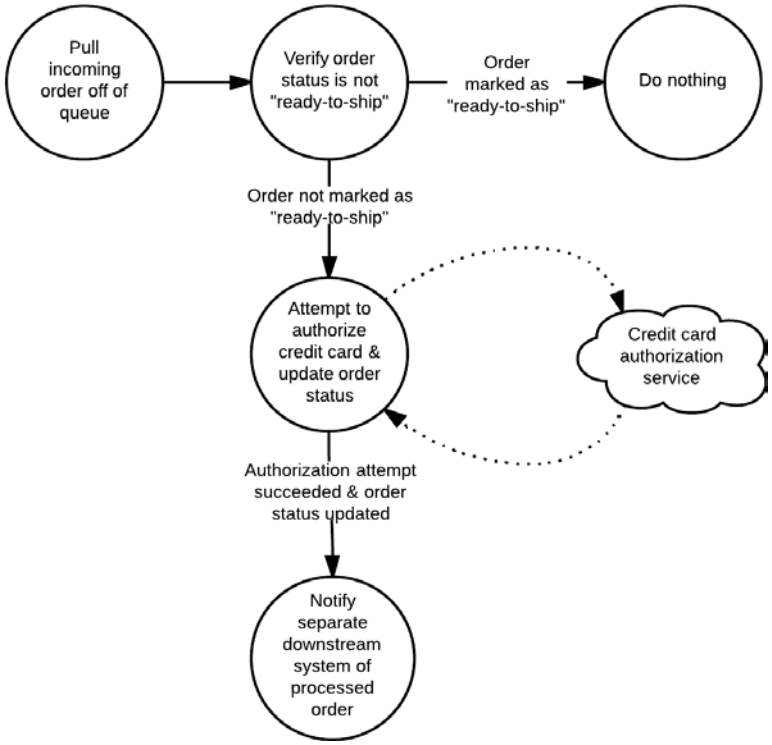


Figure 4.1 E-Commerce Credit Card Authorization Flow

4.2.2 Defining the data points

With the flow of transactions defined, we can define the data that will be involved in that flow. Incoming orders will be accepted as JSON. A sample JSON order is given in Listing 4.1. This same order JSON will be passed between each of the modules in our design:

Listing 4.1 – Order JSON

```
{
  "id" : 1234,
  "customer_id": 5678,
  "credit_card_number" : 1111222233334444,
  "credit_card_expires" : "01/2001",
  "credit_card_code" : 123,
  "amount" : 42.23
}
```

With this in mind, let's take a look at our design from a slightly different point of view. Figure 4.2 shows our design solely in terms of the data points as they travel through the components that perform processing.

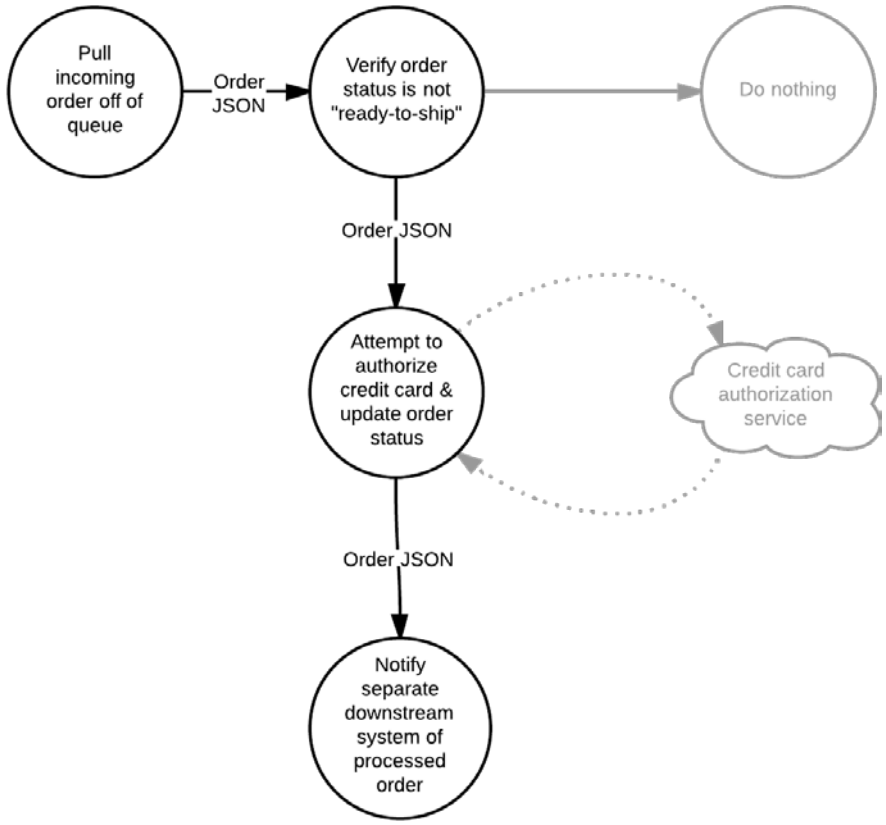


Figure 4.2 Credit Card Authorization components & data points

This approach of defining a problem in terms of data points and components that act on them should feel familiar. This is exactly how we have broken down the problems in previous chapters when creating our topologies.

4.2.3 Mapping the solution to Storm with retry characteristics

Now that we have a basic design and have identified the data that will flow through our system, we can map both our data and components to Storm concepts. The component that pulls orders off of the queue will be the **spout**. This spout will emit **tuples** containing the order JSON. There will be three **bolts** in our topology and they are as follows:

- `VerifyOrderStatus` bolt - If the order is not “ready-to-ship”, this bolt will emit a tuple containing the order JSON to the next bolt in the stream.
- `AuthorizeCreditCard` bolt - If the credit card was authorized, this bolt will update the status of the order to “ready-to-ship”. If the credit card was denied, this bolt will update the status of the order to “denied”. Regardless of the status, this bolt will emit a tuple containing the order JSON to the next bolt in the stream.
- `ProcessedOrderNotification` bolt - A bolt that notifies a separate system that an order has been processed.

In addition to the spout, bolts and tuples, we must define stream groupings for how tuples are emitted between each of the components. The following stream groupings will be used:

- Fields grouping between the spout and `VerifyOrderStatus` bolt
- Fields grouping between `VerifyOrderStatus` bolt and `AuthorizeCreditCard` bolt
- Fields grouping between `AuthorizeCreditCard` bolt and the `ProcessedOrderNotification` bolt.

In previous chapters when we used fields grouping to ensure that when we maintain counts by GitHub Committer (or grouping Geo-coordinates by time interval), the same committers tuples (or same time interval’s tuples) were routed to the same bolt instance. Here we are using fields grouping to ensure that same bolt instance will process each of these actions. All of these Storm concepts we just discussed can be seen in Figure 4.3.

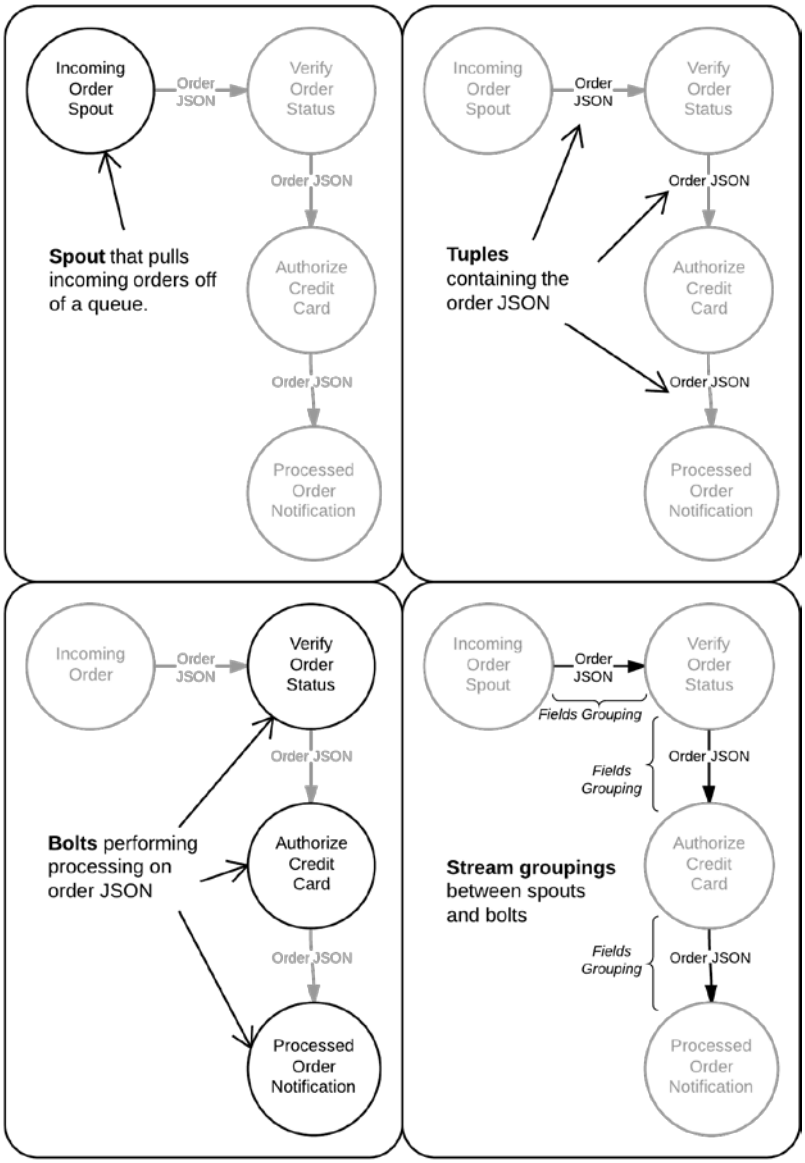


Figure 4.3 E-Commerce Credit Card Authorization Mapped to Storm Concepts

4.3 Implementation of our design

With the design in hand, we will move to the code for implementing the topology.

Listing 4.2 VerifyOrderStatus.java

```
public class VerifyOrderStatus extends BaseBasicBolt {
  private OrderDao orderDao; #1

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("order")); #2
  }

  @Override
  public void prepare(Map config,
                      TopologyContext context) {
    orderDao = new OrderDao();
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    Order order = (Order) tuple.getValueByField("order"); #3
    if (orderDao.isNotReadyToShip(order)) { #4
      outputCollector.emit(new Values(order)); #5
    }
  }
}
```

#1 Data access object (DAO) for accessing the status of the order in the database.
#2 Declares the bolt emits a tuple with a value named "order".
#3 Obtain the order from the input tuple.
#4 Check to see if the status of the order is not "ready-to-ship" in the database.
#5 Emit a tuple containing the order down the stream.

Listing 4.3 AuthorizeCreditCard.java

```
public class AuthorizeCreditCard extends BaseBasicBolt {
  private AuthorizationService authorizationService; #1
  private OrderDao orderDao; #2

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("order")); #3
  }

  @Override
  public void prepare(Map config,
                      TopologyContext context) {
    orderDao = new OrderDao();
    authorizationService = new AuthorizationService();
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    Order order = (Order) tuple.getValueByField("order"); #4
    boolean isAuthorized = authorizationService.authorize(order); #5
    if (isAuthorized) {
      orderDao.updateStatusToReadyToShip(order); #6
    } else {
      orderDao.updateStatusToDenied(order); #7
```

```
    }
    outputCollector.emit(new Values(order)); #8
  }
}
```

#1 Service for authorizing the credit card.
#2 DAO for updating the status of the order in the database.
#3 Declares the bolt emits a tuple with a value named "order".
#4 Obtain the order from the input tuple.
#5 Attempt to authorize the credit card by calling out to the authorization service.
#6 The status of the order is updated to "ready-to-ship" in the database.
#7 The status of the order is updated to "denied" in the database.
#8 Emit a tuple containing the order down the stream.

Listing 4.4 ProcessedOrderNotification.java

```
public class ProcessedOrderNotification extends BaseBasicBolt {
  private NotificationService notificationService; #1

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // This bolt does not emit anything. No output fields will be declared.
  }

  @Override
  public void prepare(Map config,
                      TopologyContext context) {
    notificationService = new NotificationService();
  }

  @Override
  public void execute(Tuple tuple,
                      BasicOutputCollector outputCollector) {
    Order order = (Order) tuple.getValueByField("order"); #2
    notificationService.notifyOrderHasBeenProcessed(order); #3
  }
}
```

#1 Notification service that notifies some downstream system the order has been processed.
#2 The order is obtained from the tuple.
#3 The notification service notifies the downstream system the order has been processed.

We now have a pretty well defined solution for our Storm topology. The steps we took to come up with a design here match those we used in previous chapters. Where this design will differ from our previous two designs is the requirement to ensure tuples are processed by all of the bolts in the stream. We didn't necessarily have this for showing GitHub commit counts and generating heat maps for social media check-ins. Dealing with financial transactions is different; we must take steps in our code to ensure these tuples get fully processed.

We have implemented the design, but what we didn't tell you is that this encapsulates basically everything you need to code-wise to ensure reliability for this use case. But let's examine in detail how this works to provide us the reliability we need. To do that need to understand how Storm provides guarantees within it.

4.4 Guaranteed Message Processing

What is a message and how does Storm guarantee it gets processed? A message is synonymous with a tuple, and Storm has the ability to ensure a tuple being emitted from a spout gets fully processed by the topology. So if a tuple were to fail at some point in the stream, Storm knows a failure occurred and can replay the tuple, making sure it gets processed. The Storm documentation commonly uses the phrase ***guaranteed message processing***, so we will follow suit throughout the book.

Understanding guaranteed message processing is essential if you want to develop **reliable** topologies. This section will help you gain that understanding by covering the following:

- What it means for a tuple to be "fully processed" or "failed" and using the Storm API to achieve each of these.
- The lifecycle of a tuple emitted from our spout and what Storm is doing behind the scenes to track that tuple.
- A spout's role in guaranteed message processing.
- A simpler way to make sure your tuple is "fully processed" or "failed" in code.

4.4.1 Tuple States: Fully Processed vs. Failed

A tuple that is emitted from a spout can trigger many additional tuples to be emitted by the downstream bolts. What this creates is a tuple tree, with the tuple emitted by the spout acting as the root. Figure 4.4 shows a tuple tree for our order authorization topology.

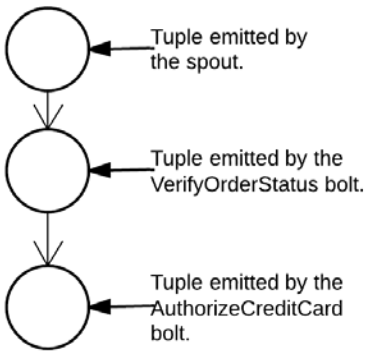


Figure 4.4 Example Tuple Tree

Storm creates and tracks a tuple tree for every tuple emitted by the spout. *Storm will consider a tuple emitted by a spout to be fully processed when all of the leaves in the tree for that tuple have been marked as processed.* There are two things you need to do with the Storm API to make sure Storm can create and track the tuple tree:

1. Make sure we ***anchor*** to input tuples when emitting new tuples from a bolt. It's a bolt's way of saying, "Okay, I'm emitting a new tuple and here's the initial input tuple as well so you can make a connection between the two."

2. Make sure our bolts tell Storm when they have finished processing an input tuple. This is called ***acking*** and is a bolt's way of saying "Hey Storm, I'm done processing this tuple so feel free to mark it as processed in the tuple tree."

By doing these two things, Storm has all it needs to create and track a tuple tree. Now in an ideal world, you could stop here and tuples emitted by our spout would always be fully processed without any problems. Unfortunately, the world of software isn't always ideal; in fact failures should be expected. Our tuples are no different and will be considered failed in one of two scenarios:

1. If all of the leaves in a tuple tree are not marked as processed (acked) within a certain timeframe. This timeframe is actually configurable at the topology level via the `TOPOLOGY_MESSAGE_TIMEOUT_SECS` setting, which defaults to 30 seconds. Here's how you would override this default:

```
Config config = new Config();
config.setMessageTimeoutSecs(60);
```

2. If a tuple is manually failed in a bolt, which triggers an immediate failure of the tuple tree.

GOING DOWN THE RABBIT-HOLE WITH ALICE… OR A TUPLE

Figure 4.5 starts things off by showing the initial state of the tuple tree after our spout emits a tuple. We have a tree with a single root node.

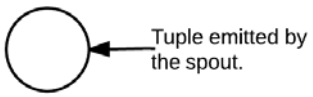


Figure 4.5 Initial State of the Tuple Tree

The next bolt in the stream is the `VerifyOrderStatus` bolt. The order has not yet been processed, so `VerifyOrderStatus` will emit a tuple, which results in the tuple tree in Figure 4.6.

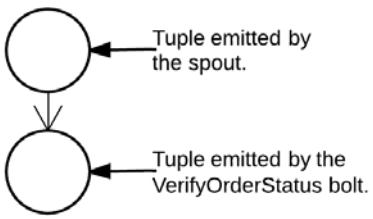


Figure 4.6 Tuple tree after the VerifyOrderStatus bolt emits a tuple

We will need to ack the input tuple in the `VerifyOrderStatus` bolt so Storm can mark that tuple as processed. Figure 4.7 shows the tuple tree after this ack has been performed.

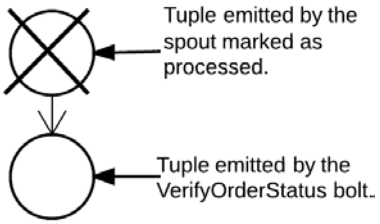


Figure 4.7 Tuple tree after the VerifyOrderStatus bolt acks its input tuple

Once a tuple has been emitted by VerifyOrderStatus, it makes its way to the AuthorizeCreditCard bolt. This bolt will perform the authorization and then emit a new tuple. Figure 4.8 shows the tuple tree after this happens.

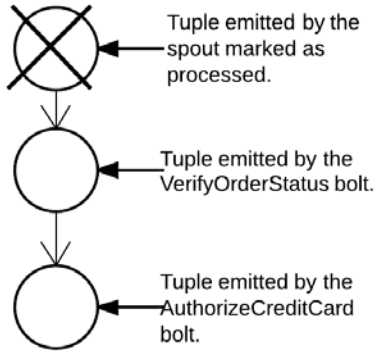


Figure 4.8 Tuple tree after the AuthorizeCreditCard bolt emits a tuple

Again, don't forget that ack! Figure 4.9 shows the tuple tree after the `AuthorizeCreditCard` bolt acks its input tuple.

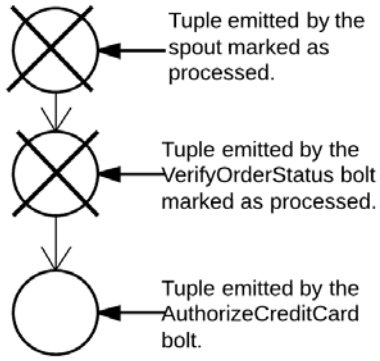


Figure 4.9 Tuple tree after the AuthorizeCreditCard bolt acks its input tuple

Finally, we get to the `ProcessedOrderNotification` bolt. This bolt does not emit a tuple, so no tuples will be added to the tuple tree. However, we do need to ack the input tuple, telling Storm this bolt has completed processing. Figure 4.10 shows the tuple tree after this ack has been performed. At this point the tuple is considered fully processed.

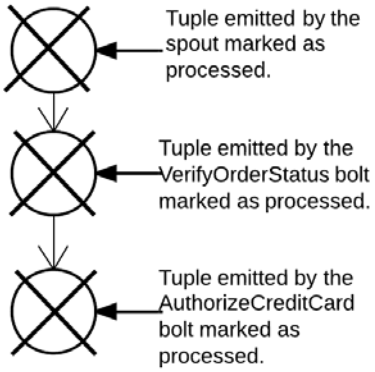


Figure 4.10 Tuple tree after the ProcessedOrderNotification bolt acks its input tuple

You now know what it means for a tuple to be either fully processed or failed. Now let's see how this can be achieved with code.

Directed Acyclic Graph & Tuple Trees

While we call it a tuple tree, it is actually a directed acyclic graph (DAG). A directed graph is a set of nodes connected by edges, where the edges have a direction to them. A DAG is a directed graph such that you cannot start at one node and follow a sequence of edges to eventually get back to that same node. Early versions of Storm only worked with trees, even though now it supports DAGs, the term "tuple tree" has stuck.

4.4.2 Implicit Anchoring, Acking and Failing

In our implementation, all of our bolts extended the `BaseBasicBolt` abstract class. The beauty of using `BaseBasicBolt` as your base class is that it will automatically provide anchoring and acking for you.

- Anchoring – Within the `execute` method of your `BaseBasicBolt` implementation, you will be emitting a tuple to be passed on to the next bolt. At this point of emitting, the provided `BasicOutputCollector` will take on the responsibility of anchoring the output tuple to the input tuple.
- In both `VerifyOrderStatus` and `AuthorizeCreditCard` bolts, we emit the order. This outgoing order tuple will anchored with the incoming order tuple at this time automatically.

```
outputCollector.emit(new Values(order));
```

- Acking – When the execute method of your BaseBasicBolt implementation completes, the tuple that was sent to it will be automatically acked.
- Failing – If there's a failure within the execute method, the way to handle that is notify BaseBasicBolt by throwing a FailedException. Then BaseBasicBolt will take care of marking that tuple as failed.

Using BaseBasicBolt to keep track of tuple states through implicit anchoring, acking and failing is quite easy. But BaseBasicBolt is not suitable for every use case. It is in fact generally useful only in use cases where a single tuple enters the bolt and a single corresponding tuple is emitted from that bolt immediately. That was the case with our credit card authorization topology so it worked there, but for more complex examples, it is not sufficient.

4.4.3 Explicit Anchoring, Acking and Failing

When you have bolts that perform more complex tasks such as:

- Aggregating on multiple input tuples (collapsing)
- Taking in a single tuple but emitting multiple out tuples in response (expanding)
- Joining multiple incoming streams (we won't cover multiple streams in this chapter, but we did have two streams going through a bolt in heat map chapter when we had a tick tuple stream in addition to the default stream).

Then we will have to move beyond the functionality provided by BaseBasicBolt. BaseBasicBolt is suitable when behavior is predictable. When you need to programmatically decide when a tuple batch is complete (e.g. when aggregating) or at runtime decide to split an incoming tuple into multiple ones (e.g. a split line of text in a word count example), then you need to programmatically decide when to anchor, ack or fail. In these cases, you need to use BaseRichBolt as a your base class instead of BaseBasicBolt.

- Anchoring – we will be passing the input tuple to the emit method on the outputCollector within the bolt's execute method

```
~~outputCollector.emit(new Values(order));~~
outputCollector.emit(tuple, new Values(order));
```

- Acking – we will be calling the ack method on the outputCollector within the bolt's execute method

```
outputCollector.ack(tuple);
```

- Failing - This is achieved by calling the fail method on the outputCollector within the bolt's execute method

```
~~throw new FailedException();~~
outputCollector.fail(tuple);
```

While the BaseBasicBolt cannot be used for all use cases, BaseRichBolt can indeed be used for everything that the former can do and more since it provides more fine-grained control over when and how you anchor, ack or fail. Even though our credit card authorization topology can be expressed in terms of BaseBasicBolt with desired reliability, it can also be written with BaseRichBolt just as easily. Let us attempt to re-write one of the bolts from credit card authorization topology using BaseRichBolt to provide an example.

Listing 4.5 Explicit Anchoring & Acking in VerifyOrderStatus.java

```
public class VerifyOrderStatus extends BaseRichBolt { #1
  private OrderDao orderDao;
  private OutputCollector outputCollector;

  @Override
  public void prepare(Map config,
                      TopologyContext topologyContext,
                      OutputCollector collector) {
    orderDao = new OrderDao();
    outputCollector = collector; #2
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("order"));
  }

  @Override
  public void execute(Tuple tuple) {
    Order order = (Order) tuple.getValueByField("order");
    if (orderDao.isNotReadyToShip(order)) {
      outputCollector.emit(tuple, new Values(order)); #3
    }
    outputCollector.ack(tuple); #4
  }
}
```

#1 Switch to extending BaseRichBolt from BaseBasicBolt
#2 Keep the OutputCollector given in the prepare method during initialization in an instance variable
#3 Anchor to the input tuple.
#4 Ack the input tuple

One thing to note is that with BaseBasicBolt, we were given a BasicOutputCollector with each call of the execute method. But with BaseRichBolt, we are responsible for maintaining tuple state by using an OutputCollector that will be provided via prepare method at the time of bolt initialization. BasicOutputCollector is a more stripped down version of OutputCollector that is available in here as BasicOutputCollector encapsulates an OutputCollector but hides the more finger grained functionality with a simpler interface.

Another thing to be mindful of is that when using BaseRichBolt, if you do not anchor your outgoing tuple(s) to the incoming tuple, you will no longer have any reliability downstream from that point onwards. BaseBasicBolt did the anchoring on your behalf.

- Anchored - `outputCollector.emit(tuple, new Values(order));`

- Unanchored - `outputCollector.emit(new Values(order));`

4.4.4 Handling failure – Knowing when to retry

We have covered a lot of concepts around guaranteed message processing. We have anchoring and acking down pat. We have yet to address how we want to handle failures. We know that we can fail a tuple by either throwing a `FailedException` (usually when using `BaseBasicBolt`) or calling `fail` on the `OutputCollector` (when using `BaseRichBolt`). Let us look at this in the context of `AuthorizeCreditCard` bolt shown in Listing 4.6. It has been re-written as a `BaseRichBolt` but we are only showing the changes to the `execute` method as the rest is exactly the same as was presented for `VerifyOrderStatus` in Listing 4.5.

Listing 4.6 Anchoring, Acking, & Failing in AuthorizeCreditCard.execute()

```
public void execute(Tuple tuple) {
  Order order = (Order) tuple.getValueByField("order");
  try {
    boolean isAuthorized = authorizationService.authorize(order);
    if (isAuthorized) {
      orderDao.updateStatusToReadyToShip(order);
    } else {
      orderDao.updateStatusToDenied(order);
    }
    outputCollector.emit(tuple, new Values(order)); #1
    outputCollector.ack(tuple); #2
  } catch (ServiceException e) {
    outputCollector.fail(tuple); #3
  }
}
```

#1 Anchor to the input tuple.
#2 Ack the input tuple.
#3 Fail the input tuple in the case of a service exception.

Failing a tuple in this manner will cause the entire tuple tree to be replayed starting at the spout. This is the key to guaranteed message processing, as this is the main trigger for the retry mechanism. It is important to know when a tuple should be failed or not.

Seems obvious, but tuples should be failed when they can be retried. Then the question becomes what can be or what should be retried?

- Known errors
  - Retriable:

    For known specific, retriable errors (say a socket timeout exception while connecting to a service), then you will want to fail them so that they will be replayed and retried.

  - Non-Retriable:

    For known errors that cannot be safely retried (like a POST to REST API) or something that does not make sense to be retried (like a ParseException while handling JSON or XML), then these shouldn't be failed. When you have one of

these non-retriable errors, instead of failing the tuple you will need to ack the tuple (without emitting a new one), as you don't want to engage the replay mechanism for it.

- Unknown errors

  Generally, unknown or unexpected errors will be a very small percentage of errors observed. So it is customary to fail them and retry them. Once you have seen them once, they become a known error (assuming logging is in place) and then you take action on them like mentioned above for known errors.

Logging and Metrics around errors

Having data on errors within a Storm topology can be quite useful, as you will see in a later chapter when we discuss metrics.

Up until now, we've focused solely on the bolts in our implementation. It's time to shift gears and move to the spout. We mentioned that when the replay mechanism gets engaged, the replaying happens starting at the spout and works it way down. Let's see how that works.

4.4.5 A Spout's Role in Guaranteed Message Processing

So far our focus has been centered on what we need to do in our bolts to achieve guaranteed message processing. This section will complete the loop and discuss the role a spout plays in guaranteeing a tuple it emits gets fully processed or replayed on failure. Listing 4.7 shows a spout's interface. Some of these methods should already look familiar.

Listing 4.7 Spout Interface

```
public interface ISpout extends Serializable {
  void open(Map config,
            TopologyContext context,
            SpoutOutputCollector outputCollector);

  void close();

  void nextTuple();            #A

  void ack(Object messageId);  #B

  void fail(Object messageId); #C
}
```

So how does a spout tie in to guaranteeing messages are processed? Here's a hint: the ack (`#B`) and `fail` (`#C`) methods have something to do with it. The following steps give a more complete picture in terms of what happens before a spout emits a tuple and after that tuple is either fully processed or failed.

1. Storm requests a tuple by calling `nextTuple` (`#A`) on the spout.

2. The spout uses the `SpoutOutputCollector` to emit a tuple to one of its streams.
3. When emitting the tuple, the spout provides a `messageId` that is used to identify that particular tuple. This may look something like:
   `spoutOutputCollector.emit(tuple, messageId);`
4. The tuple gets sent to the bolts downstream and Storm tracks the tuple tree of messages that are created. Remember, this is done via anchoring and acking within the bolts so Storm can build up the tree and mark leaves as processed.
5. If Storm detects that a tuple is fully processed, it will call the `ack` (#B) method on the originating spout task with the message id the spout provided to Storm.
6. If the tuple timed-out or one of the consuming bolts explicitly failed the tuple (such as in our `AuthorizeCreditCard` bolt), Storm will call the `fail` (#C) method on the originating spout task with the message id the spout provided to Storm.

Steps 3, 5 & 6 are the keys to guaranteed message processing from a spout's perspective. First off, everything starts with providing a `messageId` when emitting a tuple. Not doing this means Storm cannot track the tuple tree. You should add code to the `ack` method to perform any required cleanup for a fully processed tuple, if necessary. You should also add code to the `fail` method to replay the tuple.

Storm Acker Tasks

Storm uses special "acker" tasks to keep track of tuple trees in order to determine if a spout tuple has been fully processed. If an acker task sees a tuple tree is complete, it will send a message to the spout that originally emitted the tuple, resulting in that spout's `ack` method being called.

So it looks like we need to write an implementation of a spout that supports all these criteria. In the last chapter, we introduced the concept of an **unreliable data source**. An unreliable data source will not be able to support acking or failing. Once that data source hands your spout a message, it assumes you have taken responsibility for that message. A **reliable data source** on the other hand will pass messages to the spout but will not assume you have assumed responsibility for them until you have provided an acknowledgement of some sort. In addition, reliable data source will allow you to fail any given tuple with the guarantee that it will later be able to replay it. In short, a reliable data source will support #3, #5 and #6.

So the best way to demonstrate how a reliable data source's capabilities tie in to a spout API is to implement a solution with a commonly used data source. Kafka, RabbitMQ and Kestrel are all commonly used with Storm. Kafka is a valuable tool in your arsenal of infrastructure that works great with Storm, but because of its complexity and a better match

for a different use case, we will study it later. For variety's sake and for matching this use case really well, let's pick RabbitMQ as our guinea pig.

A Reliable Spout Implementation

We will go over a RabbitMQ-based spout implementation that will provide all the reliability we need for this use case. Keep in mind our main point of interest is not RabbitMQ, but rather how a well implemented spout together with reliable data source work together to provide guaranteed message processing. If you do not follow the underpinnings of RabbitMQ client API too well, don't be too worried about it. We have **emphasized** the important parts that you need to follow and de-emphasized the less relevant bits.

Listing 4.8 RabbitMQSpout

```
public class RabbitMQSpout extends BaseRichSpout { #1
  private Connection connection;
  private Channel channel;
  private QueueingConsumer consumer;
  private SpoutOutputCollector outputCollector;

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("order"));
  }

  @Override
  public void open(Map config,
                   TopologyContext topologyContext,
                   SpoutOutputCollector spoutOutputCollector) {
    outputCollector = spoutOutputCollector;
    connection = new ConnectionFactory().newConnection(); #2
    channel = connection.createChannel();
    channel.basicQos(25); #3
    consumer = new QueueingConsumer(channel); #4
    channel.basicConsume("orders", false /*auto-ack=false*/, consumer); #5
  }

  @Override
  public void nextTuple() { #6
    QueueingConsumer.Delivery delivery = consumer.nextDelivery(1L); #7
    if (delivery == null) return; /* no messages yet */
    Long msgId = delivery.getEnvelope().getDeliveryTag(); #8
    byte[] msgAsbytes = delivery.getBody();
    String msgAsString = new String(msgAsbytes, Charset.forName("UTF-8"));
    Order order = new Gson().fromJson(msgAsString, Order.class); #9
    outputCollector.emit(new Values(order), msgId); #10
  }

  @Override
  public void ack(Object msgId) { #11
    channel.basicAck((Long) msgId, false /* only acking this msgId */); #12
  }

  @Override
  public void fail(Object msgId) { #13
    channel.basicReject((Long) msgId, true /* requeue enabled */); #14
  }
```

```
  @Override
  public void close() {
    channel.close();
    connection.close();
  }
}
```

#1 Extend off the BaseRichSpout abstract class which implements ISpout that was discussed earlier
#2 Connect to a RabbitMQ node running on localhost with default credentials and settings
#3 Consume and locally buffer 25 messages at a time from RabbitMQ
#4 When we take message off a RabbitMQ queue, we will buffer them locally inside this consumer
#5 Set up subscription that consumes off a RabbitMQ queue. The messages consumed will be kept in local buffer in consumer object and we won't ack them till downstream bolts send their acks back.
#6 Storm will call next tuple when it's ready to send the next message to downstream bolt.
#7 Pick the next message off of the local buffer of RabbitMQ queue. A timeout of 1ms is imposed as nextTuple should not be allowed to block for thread safety issues within Storm.
#8 Hang to this message by the message ID assigned to it by RabbitMQ (common vernacular between RabbitMQ and Storm to refer to this message)
#9 Deserialize the message into Order object by using Google GSON JSON parsing library
#10 Emit the order tuple but anchor it with message-id provided by RabbitMQ
#11 Storm will call ack on the spout when all downstream bolts have fully processed this message. This will be called with same message id that the spout anchored to when it emitted that message.
#12 Since all downstream bolts have fully processed this message, we can tell RabbitMQ to we're done with it and remove it from the queue
#13 Storm will call fail when anything downstream fails this messages
#14 We tell RabbitMQ to requeue this message to be retried

Storm gives you the tools to guarantee that the tuples being emitted by your spout are fully processed while they are in transit within the Storm infrastructure. However for guaranteed message processing to take effect, you must use a reliable data source that has the capability of replaying a tuple. Additionally, the spout implementation has to make use of the replay mechanism provided by its data source. Understanding this is essential if you want to be successful with guaranteed message processing in your topologies.

A better RabbitMQ spout

The spout implementation for RabbitMQ in Listing 4.8 is sufficient for demonstrative purposes. But real world implementation will need to be more robust and more configurable. For those that are curious, there is such a robust, configurable and more performant implementation on GitHub that has been used in production with great success by the authors.

https://github.com/ppat/storm-rabbitmq

Whatever data source you end up using, you are encouraged to find existing spout implementations for that data source within the Storm community. Make sure you verify that the one you pick actually supports all the capabilities of your chose data source.

You are now ready to write a robust topology and introduce it to the world. We have the robustness we need in our topologies at the code level but failures don't always occur in code. Sometimes they occur within the Storm infrastructure or your hardware. For Storm to be truly fault-tolerant, we will need to understand how failures at the infrastructure level get handled.

4.5 Introducing the Storm Production Cluster

Up until this point, we have run all of our topologies in local mode, simulating a Storm cluster within a single process. Local mode has served our needs so far and is very useful for development and testing purposes. But local mode does not provide first class guaranteed processing. Everything we have discussed so far depends on anchored stream of tuples flowing from the spout. Local mode operation does not provide anchoring. Also since local mode is only intended for testing, it's about time we start looking at a real production class Storm Cluster. Within the remainder of this chapter, we will explore the role that a Storm Cluster plays in providing fault tolerance. Thereby enabling guaranteed message processing.

Storm Cluster Coverage

In this chapter, we will be covering the parts of Storm Cluster necessary to understand guaranteed message processing. In later chapters, we will cover different segments of the Storm cluster in detail under use cases where those segments fit better. Actual installation and configuration of a production class Storm Cluster will be covered in appendix A as that topic does not fit with any of the use cases.

First, we will take you through the various parts that make up a Storm cluster.

4.5.1 What is in a Storm Cluster?

A Storm cluster consists of three types of nodes.

- Master node
- Worker node(s)
- Zookeeper node(s)

The master node runs a daemon called "Nimbus" that is responsible for distributing code around the cluster to the worker nodes, assigning tasks to worker nodes and monitoring the worker nodes for failures.

The worker nodes each run a daemon called a Supervisor. The Supervisor listens for work assigned to its machine from Nimbus and starts/stops worker processes as necessary based on what Nimbus assigns it. We will get more into worker processes later in this section.

Zookeeper maintains and co-ordinates cluster state, which means if the Nimbus and/or Supervisor daemons go down, they will start back up and the topologies will continue to process like nothing happened as both Nimbus and Supervisors rely on Zookeeper to be the source of truth.

APACHE ZOOKEEPER Zookeeper is an Apache project that allows distributed processes to coordinate with each other. It is a centralized service that maintains configuration information and naming along with providing distributed synchronization and group services. For more information, check out http://zookeeper.apache.org/.

Figure 4.11 illustrates the types of nodes, their responsibilities and the flow of messages between them.

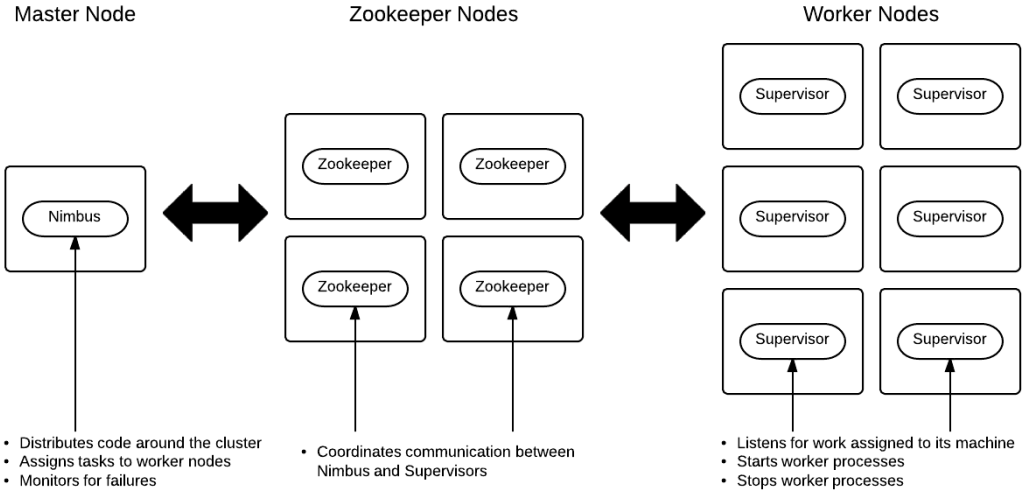


Figure 4.11 Production Cluster with Master, Worker and Zookeeper Nodes

The rest of this section will focus on worker nodes and their internals. Understanding the various parts of a worker node are necessary in order to be able to debug, monitor and tune topologies like we will be doing later in the book. In addition, a further understanding of working nodes goes a long way towards gaining deeper insights into how parallelism in Storm works.

4.5.2 Anatomy of a Worker Node As a VM

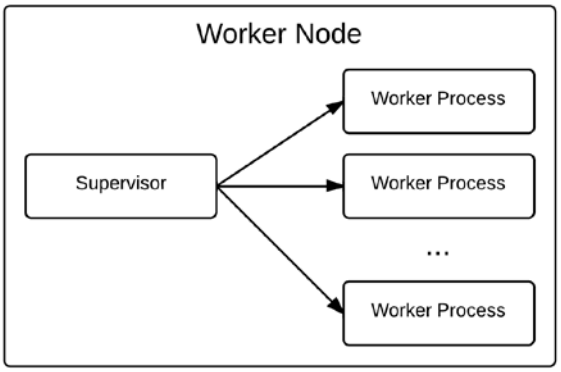


Figure 4.12 Supervisor manages the Worker Processes' JVMs

Each worker node is equivalent to actual bare-metal hardware or virtual machine (VM) running an operating system. Within this VM we have the Supervisor running as a daemon. It is tasked with administering the Worker Processes and keeping them in running state. If the Supervisor notices the ones of the Worker Processes is down, it will immediately restart it.

4.5.3 Anatomy of a Worker Process

Each worker process executes a subset of a topology. This means that each worker process belongs to a specific topology and each topology will be run across one or more worker processes. Normally, these worker processes are run across many machines within the Storm cluster. Listing 4.9 shows the code for setting the number of worker processes in a topology.

Listing 3.9 Setting the Number of Worker Processes

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

EXECUTORS/TASKS WITHIN A WORKER PROCESS

In Chapter 3, we dove into what executors (threads) and tasks (instances of a spout/bolt) are. We will take it a step further here and explain these concepts within the context of a worker process. The breakdown of a worker process can be seen in Figure 4.13. A worker process consists of one or more executors, each of which consists of one or more tasks.

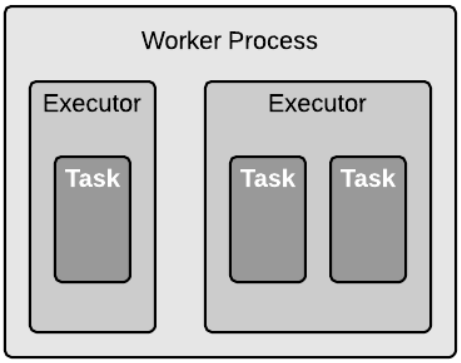


Figure 4.13 Worker Process

WORKER PROCESS AS A JVM

This is all very abstract, so Figure 4.14 should help you map the concepts of worker process, executor and task to concepts that may be more familiar.

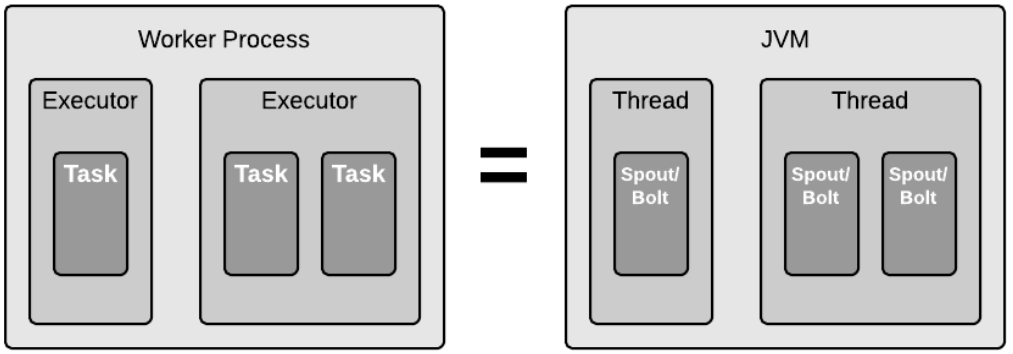


Figure 4.14 Worker Process maps to JVM equivalent

Similar to what we saw in Chapter 3, the following conclusions can be drawn from Figure 4.14:

- A worker process maps to a JVM.
- An executor maps to a thread of execution on that JVM.
- A task maps to an instance of a spout or bolt that is run in the thread of execution.

It's really that simple. A worker process is simply a JVM that is running one or more threads, each with one of more instances of a spout or bolt being executed.

4.5.4 Fail fast philosophy for fault tolerance within a Storm Cluster

The components of a Storm cluster have been designed with fault-tolerance in mind. The easiest way to explain how Storm handles fault tolerance is to answer questions in the form of *"what does Storm do when x happens?"* The most important fault tolerance questions are addressed in Table 4.1.

Table 4.1 Fault Tolerance Questions & Answers

Question	Answer
What if a worker node dies?	Supervisor will restart it and new tasks will be assigned to it. All tuples that were not fully acked at time of death will be fully replayed by the spout. This is why not only the spout needs to support replaying (reliable spout) but the data source behind the spout also needs to be reliable (support replay).
What if a worker node continuously fails to start up?	Nimbus will reassign tasks to another worker.
What if an actual machine that runs worker nodes dies?	Nimbus will reassign the tasks on that machine to healthy machines.
What if Nimbus dies?	Since Nimbus is being run under supervision (using a tool like daemontools or monit), it will restart like nothing happened.
What if a Supervisor dies?	Since Supervisors are being run under supervision (using a tool like daemontools or monit), they will restart like nothing happened.
Is Nimbus a single point of failure?	Not necessarily. Supervisors and worker nodes will continue to process. However, you lose the ability to reassign workers to other machines or deploy new topologies.
What if a zookeeper node dies?	As long as quorum is maintained in the Zookeeper ensemble, you will be fine. Zookeeper is the weakest link in the Storm infrastructure, it should be in your best interest to keep it well-configured and under monitoring. Zookeeper should be configured to run under supervision and then it should restart itself when it dies. Please refer to Zookeeper documentation.

You can see that Storm maintains a fail fast philosophy in the sense that every piece within this infrastructure can be re-started and it will re-calibrate itself and move on. So if tuples

were in mid-process during a failure, they will be failed automatically. In the case of using RabbitMQ as a data source, then tuples in mid-process on the failed unit of infrastructure will be automatically re-queued to be replayed. This happens automatically because that's how RabbitMQ operates when one of its client connections are closed or lost while the client had outstanding un-acked messages. Here again we see why it is critical to select a **reliable data source** if you want fault tolerance.

It doesn't matter if the unit of infrastructure that failed is an instance (task) or a thread (executor) or a JVM (worker process) or a VM (worker node). At each level, there are safeguards in place to ensure that everything gets restarted automatically (because everything runs under supervision) and cluster state is restored from Zookeeper and finally, the failed data points will be replayed from the data source.

Anchored vs. Unanchored Tuples

The topologies we created in earlier chapters did not take advantage of guaranteed message processing or fault tolerance. We may have used `BaseBasicBolt` in those chapters and that may have bought us implicit anchoring and acking. But our tuples in those chapters did not originate from a reliable spout. Because of that unreliable nature of those data source, when we emitted tuples at the spout, they were sent "unanchored" via `outputCollector.emit(new Values(order))`. When you don't anchor to the input tuple starting from the spout, it cannot guarantee that they will get fully processed. This is because replaying always starts at the spout. So decision to emit tuples unanchored should always be conscious one, like we made in the heat map case study.

4.6 Scaling a topology at the cluster level

In the previous chapter, we learned how to scale the heat map topology by tuning the executors and tasks. Now that we have some additional Storm primitives that apply at the cluster level, let's see what role they play when it scaling. First, let's recap how we can configure executors and tasks for a topology.

Listing 4.10 Setting the Number of Executors and Tasks

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout-name", new SomeSpout(), 1) #1
       .setNumTasks(1); #2
builder.setBolt("first-bolt", new FirstBolt(), 2) #3
       .shuffleGrouping("spout-name")
       .setNumTasks(4); #4
```

#1 One executor for the spout.
#2 One task for the spout.
#3 Two executors for the first bolt.
#4 Four tasks for the first bolt.

4.6.1 Designing a Worker Node for our Credit Card Authorization Topology

To bring our discussion about worker processes full circle, we will present a hypothetical configuration for our topology. This illustration assumes a single worker node with two worker processes. The breakdown of executors and tasks can be seen in Figure 4.15.

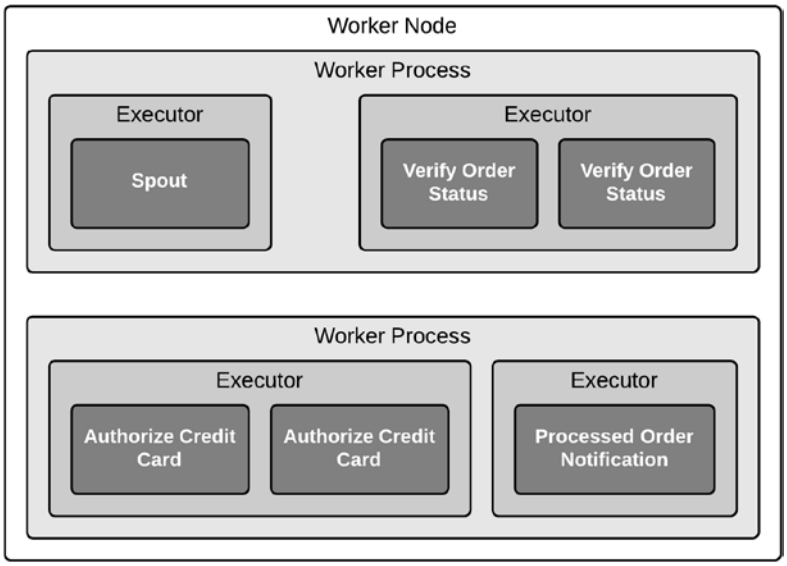


Figure 4.15 Worker Processes With Actual Spouts & Bolts from our Topology

The setup in Figure 4.15 would be done with the code in Listing 4.11.

Listing 4.11 Configuration for Our Hypothetical Storm Cluster

```
Config config = new Config();
config.setNumWorkers(2); #1
config.setMessageTimeoutSecs(60); #7

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RabbitMQSpout(), 1); #2

builder.setBolt("check-status", new VerifyOrderStatus(), 1) #3
       .fieldsGrouping("spout")
       .setNumTasks(2); #4

builder.setBolt("authorize-card", new AuthorizeCreditCard(), 1) #3
       .fieldsGrouping("check-status")
       .setNumTasks(2); #4

builder.setBolt("notification", new ProcessedOrderNotification(), 1) #3
       .fieldsGrouping("authorize-card")
       .setNumTasks(1); #5
```

```
StormSubmitter.submitTopology("credit-card-topology",
                              config,
                              builder.createTopology()); #6
```

#1 Sets the number of worker processes (JVMs) to 2.
#2 Parallelism hint that sets the number of executors (threads) to 1 for this spout, with the default number of tasks (instances) also being set to 1.
#3 Parallelism hint that sets the number of executors (threads) to 1 for this bolt.
#4 Sets the number of tasks (instances) to 2 for this bolt.
#5 Sets the number of tasks (instances) to 1 for this bolt.
#6 Submits the topology to the cluster with the provided configuration and builder.
#7 Configure how long each tuple tree has to complete before it gets failed automatically

When we set the `numWorkers` in the `Config`, we are configuring the Worker Processes desired for running this topology. We don't actually force both Worker Processes to end up on the same Worker Node (VM) as depicted in the Figure 4.15. Storm will pick where they end up based on which Worker Nodes in our cluster has vacant slots for running Worker Processes.

Parallelism vs. Concurrency, What's the Difference

Parallelism is when two threads are executing simultaneously. Concurrency when at least two threads are making progress on some sort of computation. Concurrency doesn't necessarily mean the two threads are executing simultaneously as something like time slicing may be used to simulate parallelism.

4.7 Replay Semantics

We have completed everything we need to understand reliability within Storm. There are number of things within Storm that come together to provide this reliability.

- A reliable spout backed by a reliable data source
- An anchored stream
- Acking and Failing tuples
- Message timeout for failing tuple trees that too long to complete
- Fail fast philosophy within the Storm cluster

Every single of one these play a key role. All of these are necessary to build a robust topology. But when you examine the replay characteristics of streams flowing through your topology, you will gain a different perspective into different kinds of reliability available within Storm. The idea behind this is not too different from how we saw a different kind of scaling problem with the heat map case study when we carefully examined the data streams. We can assign different semantics to reliability when we become mindful of different requirements being met by our streams.

- At most once processing

You would use this mechanism when you want to guarantee that no single tuple ever gets processed more than once. If it succeeds, great, but if it fails it will be discarded. In this case, no replaying will happen. Regardless, this semantic *provides no reliability* and provides the simplest implementation. This is what we did in preceding chapters, as those use cases did not dictate a need for reliability. We may have used `BaseBasicBolt` (with its automated anchoring and acking) in previous chapters, but we did not anchor the tuples when we first emitted them from the spout.

Requirements - None

- At least once processing

  This mechanism can be used when we want to guarantee that every single tuple must get processed successfully, at least once. If a single tuple gets replayed several times for some reason and it succeeds more than once, that is okay under this replay semantic. But the key is that it must succeed at least once.

  Requirements – Reliable spout with reliable data source, an anchored stream with acked or failed tuples, a real Storm cluster. This is what we covered in this chapter.

- Exactly once processing

  This is similar to "at least once" processing in the sense that it can guarantee that every tuple gets processed successfully. But it will take care to ensure that once a tuple is processed, there is no risk of it ever being replayed again.

  Requirements – This is more complex and we will cover this in a later chapter.

In this chapter, we successfully demonstrated how to build a robust topology obeying "at least once" reliability semantics.

At least once vs. Exactly once

It is important not to dismiss "at least once" and always favor "exactly once". At least once processing is useful –

- If you want to ensure processing of a tuple and do not care about duplication. That might be the case if each time it gets processed, it replaces the previous result and you don't care about the previous value (which can be case more often than not) or the order in which the replays happened.
- If you already have infrastructure that takes care of de-duplication for you as in the case, you will just be using Storm reliability measures to ensure that tuples flow through your topology smoothly and no data loss occurs.

On the other hand, "exactly once" should be preferred when you want Storm and its supporting infrastructure to take care of not only processing guarantees but also de-duplication guarantees.

4.8 Summary

In this chapter you have seen:

- What it means for a tuple to be fully processed versus failed.
- How Storm is able to track whether or not a tuple is fully processed or failed by creating a tuple tree.
- The importance of reliable spout backed by a reliable data source for guaranteed message processing.
- The steps Storm takes to guarantee that messages get processed.
- The core components of a Storm cluster.
- How a worker node is broken down into worker processes, executors and tasks, and how to configure each of these elements.
- How parallelism primitives get expanded within a cluster to include Worker Processes

This chapter wraps up Section 1, getting you up to speed with the fundamentals of Storm. Understanding these fundamentals is essential if you want to take advantage of the full power of Storm. However, being able to do things such as guarantee "at least once" processing of your messages and parallelism to make things faster is only part of the picture. What separates a beginner from a more advanced Storm user is the ability to solve those tricky problems that always seem to pop up when you are running in production. You know, the *"why on Earth did my topology slow down by an order of magnitude of n?"* problems. Skills around monitoring, debugging and tuning your topologies are a must if you want to take those initial Storm topology deployments to the next level.