In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.1 "Distributed Word Count"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

P. Taylor Goetz is an Apache Storm committer and release manager and has been involved with the usage and development of Storm since it was first released as open source in October of 2011. As an active contributor to the Storm user community, Taylor leads a number of open source projects that enable enterprises to integrate Storm into heterogeneous infrastructure.

Presently, he works at Hortonworks where he leads the integration of Storm into Hortonworks Data Platform (HDP). Prior to joining Hortonworks, he worked at Health Market Science where he led the integration of Storm into HMS’ next generation Master Data Management platform with technologies including Cassandra, Kafka, Elastic Search, and the Titan graph database.

I would like to thank my amazing wife, children, family, and friends whose love, support, and sacrifices made this book possible. I owe you all a debt of gratitude.

For More Information:
**Brian O'Neill** is a husband, hacker, hiker, and kayaker. He is a fisherman and father as well as big data believer, innovator, and distributed computing dreamer.

He has been a technology leader for over 15 years and is recognized as an authority on big data. He has experience as an architect in a wide variety of settings, from start-ups to Fortune 500 companies. He believes in open source and contributes to numerous projects. He leads projects that extend Cassandra and integrate the database with indexing engines, distributed processing frameworks, and analytics engines. He won InfoWorld's Technology Leadership award in 2013. He authored the Dzone reference card on Cassandra and was selected as a Datastax Cassandra MVP in 2012 and 2013.

In the past, he has contributed to expert groups within the **Java Community Process (JCP)** and has patents in artificial intelligence and context-based discovery. He is proud to hold a B.S. in Computer Science from Brown University.

Presently, Brian is Chief Technology Officer for **Health Market Science (HMS)**, where he heads the development of their big data platform focused on data management and analysis for the healthcare space. The platform is powered by Storm and Cassandra and delivers real-time data management and analytics as a service.

---

For my family...To my wife Lisa, We put our faith in the wind. And our mast has carried us to the clouds. Rooted to the earth by our children, and fastened to the bedrock of those that have gone before us, our hands are ever entwined by the fabric of our family. Without all of you, this ink would never have met this page.

---

For More Information:

Storm Blueprints: Patterns for Distributed Real-time Computation

The demand for timely, actionable information is pushing software systems to process an increasing amount of data in a decreasing amount of time. Additionally, as the number of connected devices increases and as these devices are applied to a broadening spectrum of industries, that demand is becoming increasingly pervasive. Traditional enterprise operational systems are being forced to operate on scales of data that were originally associated only with Internet-scale companies. This monumental shift is forcing the collapse of more traditional architectures and approaches that separated online transactional systems and offline analysis. Instead, people are reimagining what it means to extract information from data. Frameworks and infrastructure are likewise evolving to accommodate this new vision.

Specifically, data generation is now viewed as a series of discrete events. Those event streams are associated with data flows, some operational and some analytical, but processed by a common framework and infrastructure.

Storm is the most popular framework for real-time stream processing. It provides the fundamental primitives and guarantees required for fault-tolerant distributed computing in high-volume, mission-critical applications. It is both an integration technology as well as a data flow and control mechanism. Many large companies are using Storm as the backbone of their big data platforms.

Using design patterns from this book, you will learn to develop, deploy, and operate data processing flows capable of processing billions of transactions per hour/day.

*Storm Blueprints: Patterns for Distributed Real-time Computation* covers a broad range of distributed computing topics, including not only design and integration patterns but also domains and applications to which the technology is immediately useful and commonly applied. This book introduces the reader to Storm using real-world examples, beginning with simple Storm topologies. The examples increase in complexity, introducing advanced Storm concepts as well as more sophisticated approaches to deployment and operational concerns.

For More Information:
What This Book Covers

Chapter 1, Distributed Word Count, introduces the core concepts of distributed stream processing with Storm. The distributed word count example demonstrates many of the structures, techniques, and patterns required for more complex computations. In this chapter, we will gain a basic understanding of the structure of Storm computations. We will set up a development environment and understand the techniques used to debug and develop Storm applications.

Chapter 2, Configuring Storm Clusters, provides a deeper look into the Storm technology stack and the process of setting up and deploying to a Storm cluster. In this chapter, we will automate the installation and configuration of a multi-node cluster using the Puppet provisioning tool.

Chapter 3, Trident Topologies and Sensor Data, covers Trident topologies. Trident provides a higher-level abstraction on top of Storm that abstracts away the details of transactional processing and state management. In this chapter, we will apply the Trident framework to process, aggregate, and filter sensor data to detect a disease outbreak.

Chapter 4, Real-time Trend Analysis, introduces trend analysis techniques using Storm and Trident. Real-time trend analysis involves identifying patterns in data streams. In this chapter, you will integrate with Apache Kafka and will implement a sliding window to compute moving averages.

Chapter 5, Real-time Graph Analysis, covers graph analysis using Storm to persist data to a graph database and query that data to discover relationships. Graph databases are databases that store data as graph structures with vertices, edges, and properties and focus primarily on relationships between entities. In this chapter, you will integrate Storm with Titan, a popular graph database, using Twitter as a data source.

Chapter 6, Artificial Intelligence, applies Storm to an artificial intelligence algorithm typically implemented using recursion. We expose some of the limitations of Storm, and examine patterns to accommodate those limitations. In this chapter, using Distributed Remote Procedure Call (DRPC), you will implement a Storm topology capable of servicing synchronous queries to determine the next best move in tic-tac-toe.

Chapter 7, Integrating Druid for Financial Analytics, demonstrates the complexities of integrating Storm with non-transactional systems. To support such integrations, the chapter presents a pattern that leverages ZooKeeper to manage the distributed state. In this chapter, you will integrate Storm with Druid, which is an open source infrastructure for exploratory analytics, to deliver a configurable real-time system for analysis of financial events.

For More Information:
Chapter 8, *Natural Language Processing*, introduces the concept of Lambda architecture, pairing real time and batch processing to create a resilient system for analytics. Building on the Chapter 7, *Integrating Druid for Financial Analytics* you will incorporate the Hadoop infrastructure and examine a MapReduce job to backfill analytics in Druid in the event of a host failure.

Chapter 9, *Deploying Storm on Hadoop for Advertising Analysis*, demonstrates converting an existing batch process, written in Pig script running on Hadoop, into a real-time Storm topology. To do this, you will leverage Storm-YARN, which allows users to leverage YARN to deploy and run Storm clusters. Running Storm on Hadoop allows enterprises to consolidate operations and utilize the same infrastructure for both real time and batch processing.

Chapter 10, *Storm in the Cloud*, covers best practices for running and deploying Storm in a cloud-provider hosted environment. Specifically, you will leverage Apache Whirr, a set of libraries for cloud services, to deploy and configure Storm and its supporting technologies to infrastructure provisioned via Amazon Web Services (AWS) Elastic Compute Cloud (EC2). Additionally, you will leverage Vagrant to create clustered environments for development and testing.

For More Information:
In this chapter, we will introduce you to the core concepts involved in creating distributed stream processing applications with Storm. We do this by building a simple application that calculates a running word count from a continuous stream of sentences. The word count example involves many of the structures, techniques, and patterns required for more complex computation, yet it is simple and easy to follow.

We will begin with an overview of Storm's data structures and move on to implementing the components that comprise a fully fledged Storm application. By the end of the chapter, you will have gained a basic understanding of the structure of Storm computations, setting up a development environment, and techniques for developing and debugging Storm applications.

This chapter covers the following topics:

- Storm's basic constructs – topologies, streams, spouts, and bolts
- Setting up a Storm development environment
- Implementing a basic word count application
- Parallelization and fault tolerance
- Scaling by parallelizing computation tasks

For More Information:
Introducing elements of a Storm topology – streams, spouts, and bolts

In Storm, the structure of a distributed computation is referred to as a topology and is made up of streams of data, spouts (stream producers), and bolts (operations). Storm topologies are roughly analogous to jobs in batch processing systems such as Hadoop. However, while batch jobs have clearly defined beginning and end points, Storm topologies run forever, until explicitly killed or undeployed.

Streams

The core data structure in Storm is the tuple. A tuple is simply a list of named values (key-value pairs), and a Stream is an unbounded sequence of tuples. If you are familiar with complex event processing (CEP), you can think of Storm tuples as events.

Spouts

Spouts represent the main entry point of data into a Storm topology. Spouts act as adapters that connect to a source of data, transform the data into tuples, and emit the tuples as a stream.

As you will see, Storm provides a simple API for implementing spouts. Developing a spout is largely a matter of writing the code necessary to consume data from a raw source or API. Potential data sources include:

For More Information:
Chapter 1

- Click streams from a web-based or mobile application
- Twitter or other social network feeds
- Sensor output
- Application log events

Since spouts typically don't implement any specific business logic, they can often be reused across multiple topologies.

**Bolts**

Bolts can be thought of as the *operators or functions* of your computation. They take as input any number of streams, process the data, and optionally emit one or more streams. Bolts may subscribe to streams emitted by spouts or other bolts, making it possible to create a complex network of stream transformations.

Bolts can perform any sort of processing imaginable and like the Spout API, the bolt interface is simple and straightforward. Typical functions performed by bolts include:

- Filtering tuples
- Joins and aggregations
- Calculations
- Database reads/writes

**Introducing the word count topology data flow**

Our word count topology (depicted in the following diagram) will consist of a single spout connected to three downstream bolts.

![Word count topology](image)

For More Information:
**Sentence spout**
The SentenceSpout class will simply emit a stream of single-value tuples with the key name "sentence" and a string value (a sentence), as shown in the following code:

```json
{ "sentence":"my dog has fleas" }
```

To keep things simple, the source of our data will be a static list of sentences that we loop over, emitting a tuple for every sentence. In a real-world application, a spout would typically connect to a dynamic source, such as tweets retrieved from the Twitter API.

**Introducing the split sentence bolt**
The split sentence bolt will subscribe to the sentence spout's tuple stream. For each tuple received, it will look up the "sentence" object's value, split the value into words, and emit a tuple for each word:

```json
{ "word" : "my" }  
{ "word" : "dog" }  
{ "word" : "has" }  
{ "word" : "fleas" }
```

**Introducing the word count bolt**
The word count bolt subscribes to the output of the SplitSentenceBolt class, keeping a running count of how many times it has seen a particular word. Whenever it receives a tuple, it will increment the counter associated with a word and emit a tuple containing the word and the current count:

```json
{ "word" : "dog", "count" : 5 }
```

**Introducing the report bolt**
The report bolt subscribes to the output of the WordCountBolt class and maintains a table of all words and their corresponding counts, just like WordCountBolt. When it receives a tuple, it updates the table and prints the contents to the console.

For More Information:
Implementing the word count topology

Now that we've introduced the basic Storm concepts, we're ready to start developing a simple application. For now, we'll be developing and running a Storm topology in local mode. Storm's local mode simulates a Storm cluster within a single JVM instance, making it easy to develop and debug Storm topologies in a local development environment or IDE. In later chapters, we'll show you how to take Storm topologies developed in local mode and deploy them to a fully clustered environment.

Setting up a development environment

Creating a new Storm project is just a matter of adding the Storm library and its dependencies to the Java classpath. However, as you'll learn in Chapter 2, Configuring Storm Clusters, deploying a Storm topology to a clustered environment requires special packaging of your compiled classes and dependencies. For this reason, it is highly recommended that you use a build management tool such as Apache Maven, Gradle, or Leinengen. For the distributed word count example, we will use Maven.

Let's begin by creating a new Maven project:

$ mvn archetype:create -DgroupId=storm.blueprints -DartifactId=Chapter1 -DpackageName=storm.blueprints.chapter1.v1

Next, edit the pom.xml file and add the Storm dependency:

<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
  <version>0.9.1-incubating</version>
</dependency>

Then, test the Maven configuration by building the project with the following command:

$ mvn install

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Maven will download the Storm library and all its dependencies. With the project set up, we're now ready to begin writing our Storm application.

For More Information:
Implementing the sentence spout

To keep things simple, our SentenceSpout implementation will simulate a data source by creating a static list of sentences that gets iterated. Each sentence is emitted as a single field tuple. The complete spout implementation is listed in Example 1.1.

Example 1.1: SentenceSpout.java

```java
public class SentenceSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "i don't think i like fleas"
    };
    private int index = 0;

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

    public void open(Map config, TopologyContext context,
                      SpoutOutputCollector collector) {
        this.collector = collector;
    }

    public void nextTuple() {
        this.collector.emit(new Values(sentences[index]));
        index++;
        if (index >= sentences.length) {
            index = 0;
        }
        Utils.waitForMillis(1);
    }
}
```

The BaseRichSpout class is a convenient implementation of the ISpout and IComponent interfaces and provides default implementations for methods we don't need in this example. Using this class allows us to focus only on the methods we need.

For More Information:
The `declareOutputFields()` method is defined in the `IComponent` interface that all Storm components (spouts and bolts) must implement and is used to tell Storm what streams a component will emit and the fields each stream's tuples will contain. In this case, we're declaring that our spout will emit a single (default) stream of tuples containing a single field ("sentence").

The `open()` method is defined in the `ISpout` interface and is called whenever a spout component is initialized. The `open()` method takes three parameters: a map containing the Storm configuration, a `TopologyContext` object that provides information about a components placed in a topology, and a `SpoutOutputCollector` object that provides methods for emitting tuples. In this example, we don't need to perform much in terms of initialization, so the `open()` implementation simply stores a reference to the `SpoutOutputCollector` object in an instance variable.

The `nextTuple()` method represents the core of any spout implementation. Storm calls this method to request that the spout emit tuples to the output collector. Here, we just emit the sentence at the current index, and increment the index.

### Implementing the split sentence bolt

The `SplitSentenceBolt` implementation is listed in *Example 1.2*.

#### Example 1.2 – SplitSentenceBolt.java

```java
public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for (String word : words) {
            this.collector.emit(new Values(word));
        }
    }

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

For More Information:

The **BaseRichBolt** class is another convenience class that implements both the **IComponent** and **IBolt** interfaces. Extending this class frees us from having to implement methods we're not concerned with and lets us focus on the functionality we need.

The **prepare()** method defined by the **IBolt** interface is analogous to the **open()** method of **ISpout**. This is where you would prepare resources such as database connections during bolt initialization. Like the **SentenceSpout** class, the **SplitSentenceBolt** class does not require much in terms of initialization, so the **prepare()** method simply saves a reference to the **OutputCollector** object.

In the **declareOutputFields()** method, the **SplitSentenceBolt** class declares a single stream of tuples, each containing one field ("word").

The core functionality of the **SplitSentenceBolt** class is contained in the **execute()** method defined by **IBolt**. This method is called every time the bolt receives a tuple from a stream to which it subscribes. In this case, it looks up the value of the "sentence" field of the incoming tuple as a string, splits the value into individual words, and emits a new tuple for each word.

### Implementing the word count bolt

The **WordCountBolt** class (Example 1.3) is the topology component that actually maintains the word count. In the bolt's **prepare()** method, we instantiate an instance of **HashMap<String, Long>** that will store all the words and their corresponding counts. It is common practice to instantiate most instance variables in the **prepare()** method. The reason behind this pattern lies in the fact that when a topology is deployed, its component spouts and bolts are serialized and sent across the network. If a spout or bolt has any non-serializable instance variables instantiated before serialization (created in the constructor, for example) a **NotSerializableException** will be thrown and the topology will fail to deploy. In this case, since **HashMap<String, Long>** is serializable, we could have safely instantiated it in the constructor. However, in general, it is best to limit constructor arguments to primitives and serializable objects and instantiate non-serializable objects in the **prepare()** method.

In the **declareOutputFields()** method, the **WordCountBolt** class declares a stream of tuples that will contain both the word received and the corresponding count. In the **execute()** method, we look up the count for the word received (initializing it to 0 if necessary), increment and store the count, and then emit a new tuple consisting of the word and current count. Emitting the count as a stream allows other bolts in the topology to subscribe to the stream and perform additional processing.

For More Information:

Example 1.3 – WordCountBolt.java

public class WordCountBolt extends BaseRichBolt{
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;

    public void prepare(Map config, TopologyContext context,
            OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }

    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if(count == null){
            count = 0L;
        }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }

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

Implementing the report bolt

The purpose of the ReportBolt class is to produce a report of the counts for each word. Like the WordCountBolt class, it uses a HashMap<String, Long> object to record the counts, but in this case, it just stores the count received from the counter bolt.

One difference between the report bolt and the other bolts we've written so far is that it is a terminal bolt—it only receives tuples. Because it does not emit any streams, the declareOutputFields() method is left empty.

The report bolt also introduces the cleanup() method defined in the IBolt interface. Storm calls this method when a bolt is about to be shutdown. We exploit the cleanup() method here as a convenient way to output our final counts when the topology shuts down, but typically, the cleanup() method is used to release resources used by a bolt, such as open files or database connections.

For More Information:
One important thing to keep in mind about the IBolt.cleanup() method when writing bolts is that there is no guarantee that Storm will call it when a topology is running on a cluster. We'll discuss the reasons behind this when we talk about Storm’s fault tolerance mechanisms in the next chapter. But for this example, we’ll be running Storm in a development mode where the cleanup() method is guaranteed to be called.

The full source for the ReportBolt class is listed in Example 1.4.

Example 1.4 – ReportBolt.java

```java
public class ReportBolt extends BaseRichBolt {

    private HashMap<String, Long> counts = null;

    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.counts = new HashMap<String, Long>();
    }

    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");
        this.counts.put(word, count);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // this bolt does not emit anything
    }

    public void cleanup() {
        System.out.println("--- FINAL COUNTS ---");
        List<String> keys = new ArrayList<String>();
        keys.addAll(this.counts.keySet());
        Collections.sort(keys);
        for (String key : keys) {
            System.out.println(key + " : " + this.counts.get(key));
        }
        System.out.println("--------------");
    }
}
```

For More Information:
Implementing the word count topology

Now that we've defined the spout and bolts that will make up our computation, we're ready to wire them together into a runnable topology (refer to Example 1.5).

Example 1.5 – WordCountTopology.java

```java
public class WordCountTopology {

    private static final String SENTENCE_SPOUT_ID = "sentence-spout";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String TOPOLOGY_NAME = "word-count-topology";

    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout(SENTENCE_SPOUT_ID, spout);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt(SPLIT_BOLT_ID, splitBolt)
            .shuffleGrouping(SENTENCE_SPOUT_ID);
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt(COUNT_BOLT_ID, countBolt)
            .fieldsGrouping(SPLIT_BOLT_ID, new Fields("word"));
        // WordCountBolt --> ReportBolt
        builder.setBolt(REPORT_BOLT_ID, reportBolt)
            .globalGrouping(COUNT_BOLT_ID);

        Config config = new Config();
        LocalCluster cluster = new LocalCluster();

        cluster.submitTopology(TOPOLOGY_NAME, config, builder.
            createTopology());
        waitForSeconds(10);
        cluster.killTopology(TOPOLOGY_NAME);
        cluster.shutdown();
    }
}
```

For More Information:
Storm topologies are typically defined and run (or submitted if the topology is being deployed to a cluster) in a Java main() method. In this example, we begin by defining string constants that will serve as unique identifiers for our Storm components. We begin the main() method by instantiating our spout and bolts and creating an instance of TopologyBuilder. The TopologyBuilder class provides a fluent-style API for defining the data flow between components in a topology. We start by registering the sentence spout and assigning it a unique ID:

```java
builder.setSpout(SENTENCE_SPOUT_ID, spout);
```

The next step is to register SplitSentenceBolt and establish a subscription to the stream emitted by the SentenceSpout class:

```java
builder.setBolt(SPLIT_BOLT_ID, splitBolt)
   .shuffleGrouping(SENTENCE_SPOUT_ID);
```

The setBolt() method registers a bolt with the TopologyBuilder class and returns an instance of BoltDeclarer that exposes methods for defining the input source(s) for a bolt. Here we pass in the unique ID we defined for the SentenceSpout object to the shuffleGrouping() method establishing the relationship. The shuffleGrouping() method tells Storm to shuffle tuples emitted by the SentenceSpout class and distribute them evenly among instances of the SplitSentenceBolt object. We will explain stream groupings in detail shortly in our discussion of parallelism in Storm.

The next line establishes the connection between the SplitSentenceBolt class and the WordCountBolt class:

```java
builder.setBolt(COUNT_BOLT_ID, countBolt)
   .fieldsGrouping(SPLIT_BOLT_ID, new Fields("word"));
```

As you'll learn, there are times when it's imperative that tuples containing certain data get routed to a specific instance of a bolt. Here, we use the fieldsGrouping() method of the BoltDeclarer class to ensure that all tuples containing the same "word" value get routed to the same WordCountBolt instance.

The last step in defining our data flow is to route the stream of tuples emitted by the WordCountBolt instance to the ReportBolt class. In this case, we want all tuples emitted by WordCountBolt routed to a single ReportBolt task. This behavior is provided by the globalGrouping() method, as follows:

```java
builder.setBolt(REPORT_BOLT_ID, reportBolt)
   .globalGrouping(COUNT_BOLT_ID);
```

For More Information:
With our data flow defined, the final step in running our word count computation is to build the topology and submit it to a cluster:

```java
Config config = new Config();
LocalCluster cluster = new LocalCluster();

    cluster.submitTopology(TOPOLOGY_NAME, config, builder.createTopology());
    waitForSeconds(10);
    cluster.killTopology(TOPOLOGY_NAME);
    cluster.shutdown();
```

Here, we're running Storm in local mode using Storm's `LocalCluster` class to simulate a full-blown Storm cluster within our local development environment. Local mode is a convenient way to develop and test Storm applications without the overhead of deploying to a distributed cluster. Local mode also allows you to run Storm topologies within an IDE, setting breakpoints, halting execution, inspecting variables and profiling the application in ways that are much more time consuming or near impossible when deploying to a Storm cluster.

In this example, we create a `LocalCluster` instance and call the `submitTopology()` method with the topology name, an instance of `backtype.storm.Config`, and the `Topology` object returned by the `TopologyBuilder` class' `createTopology()` method. As you'll see in the next chapter, the `submitTopology()` method used to deploy a topology in local mode has the same signature as the method to deploy a topology in remote (distributed) mode.

Storm's `Config` class is simply an extension of `HashMap<String, Object>`, which defines a number of Storm-specific constants and convenience methods for configuring a topology's runtime behavior. When a topology is submitted, Storm will merge its predefined default configuration values with the contents of the `Config` instance passed to the `submitTopology()` method, and the result will be passed to the `open()` and `prepare()` methods of the topology spouts and bolts respectively. In this sense, the `Config` object represents a set of configuration parameters that are global to all components in a topology.

We're now ready to run the `WordCountTopology` class. The `main()` method will submit the topology, wait for ten seconds while it runs, kill (undeploy) the topology, and finally shut down the local cluster. When the program run is complete, you should see console output similar to the following:

For More Information:
Introducing parallelism in Storm

Recall from the introduction that Storm allows a computation to scale horizontally across multiple machines by dividing the computation into multiple, independent tasks that execute in parallel across a cluster. In Storm, a task is simply an instance of a spout or bolt running somewhere on the cluster.

To understand how parallelism works, we must first explain the four main components involved in executing a topology in a Storm cluster:

- **Nodes (machines)**: These are simply machines configured to participate in a Storm cluster and execute portions of a topology. A Storm cluster contains one or more nodes that perform work.

- **Workers (JVMs)**: These are independent JVM processes running on a node. Each node is configured to run one or more workers. A topology may request one or more workers be assigned to it.

- **Executors (threads)**: These are Java threads running within a worker JVM process. Multiple tasks can be assigned to a single executor. Unless explicitly overridden, Storm will assign one task for each executor.

- **Tasks (bolt/spout instances)**: Tasks are instances of spouts and bolts whose `nextTuple()` and `execute()` methods are called by executor threads.

--- FINAL COUNTS ---

a : 1426
ate : 1426
beverages : 1426
cold : 1426
cow : 1426
dog : 2852
don't : 2851
deas : 2851
has : 1426
have : 1426
homework : 1426
i : 4276
like : 2851
man : 1426
my : 2852
the : 1426
think : 1425

For More Information:
**WordCountTopology parallelism**

So far in our word count example, we have not explicitly used any of Storm's parallelism APIs; instead, we allowed Storm to use its default settings. In most cases, unless overridden, Storm will default most parallelism settings to a factor of one.

Before changing the parallelism settings for our topology, let's consider how our topology will execute with the default settings. Assuming we have one machine (node), have assigned one worker to the topology, and allowed Storm to one task per executor, our topology execution would look like the following:

As you can see, the only parallelism we have is at the thread level. Each task runs on a separate thread within a single JVM. How can we increase the parallelism to more effectively utilize the hardware we have at our disposal? Let's start by increasing the number of workers and executors assigned to run our topology.

**Adding workers to a topology**

Assigning additional workers is an easy way to add computational power to a topology, and Storm provides the means to do so through its API as well as pure configuration. Whichever method we choose, our component spouts and bolts do not have to change, and can be reused as is.

In the previous version of the word count topology, we introduced the Config object that gets passed to the `submitTopology()` method at deployment time but left it largely unused. To increase the number of workers assigned to a topology, we simply call the `setNumWorkers()` method of the Config object:

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

For More Information:

Distributed Word Count

This assigns two workers to our topology instead of the default of one. While this will add computation resources to our topology, in order to effectively utilize those resources, we will also want to adjust the number of executors in our topology as well as the number of tasks per executor.

Configuring executors and tasks

As we've seen, Storm creates a single task for each component defined in a topology, by default, and assigns a single executor for each task. Storm's parallelism API offers control over this behavior by allowing you to set the number of executors per task as well as the number of tasks per executor.

The number of executors assigned to a given component is configured by setting a parallelism hint when defining a stream grouping. To illustrate this feature, let's modify our topology definition to parallelize `SentenceSpout` such that it is assigned two tasks and each task is assigned its own executor thread:

```java
builder.setSpout(SENTENCE_SPOUT_ID, spout, 2);
```

If we're using one worker, the execution of our topology now looks like the following:

![Diagram](image-url)
Next, we will set up the split sentence bolt to execute as four tasks with two executors. Each executor thread will be assigned two tasks to execute (4 / 2 = 2). We'll also configure the word count bolt to run as four tasks, each with its own executor thread:

```java
builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2)
    .setNumTasks(4)
    .shuffleGrouping(SENTENCE_SPOUT_ID);

builder.setBolt(COUNT_BOLT_ID, countBolt, 4)
    .fieldsGrouping(SPLIT_BOLT_ID, new Fields("word"));
```

With two workers, the execution of the topology will now look like the following diagram:
Distributed Word Count

With the topology parallelism increased, running the updated `WordCountTopology` class should yield higher total counts for each word:

```
--- FINAL COUNTS ---
a : 2726
ate : 2722
beverages : 2723
cold : 2723
cow : 2726
dog : 5445
don't : 5444
fleas : 5451
has : 2723
have : 2722
homework : 2722
i : 8175
like : 5449
man : 2722
my : 5445
the : 2727
think : 2722
--------------
```

Since spout emits data indefinitely and only stops when the topology is killed, the actual counts will vary depending on the speed of your computer and what other processes are running on it, but you should see an overall increase in the number of words emitted and processed.

It's important to point out that increasing the number of workers has no effect when running a topology in local mode. A topology running in local mode always runs in a single JVM process, so only task and executor parallelism settings have any effect. Storm's local mode offers a decent approximation of cluster behavior and is very useful for development, but you should always test your application in a true clustered environment before moving to production.

Understanding stream groupings

Based on the previous example, you may wonder why we did not bother increasing the parallelism of `ReportBolt`. The answer is that it does not make any sense to do so. To understand why, you need to understand the concept of stream groupings in Storm.

For More Information:
A stream grouping defines how a stream's tuples are distributed among bolt tasks in a topology. For example, in the parallelized version of the word count topology, the `SplitSentenceBolt` class was assigned four tasks in the topology. The stream grouping determines which one of those tasks will receive a given tuple.

Storm defines seven built-in stream groupings:

- **Shuffle grouping**: This randomly distributes tuples across the target bolt's tasks such that each bolt receives an equal number of tuples.
- **Fields grouping**: This routes tuples to bolt tasks based on the values of the fields specified in the grouping. For example, if a stream is grouped on the "word" field, tuples with the same value for the "word" field will always be routed to the same bolt task.
- **All grouping**: This replicates the tuple stream across all bolt tasks such that each task will receive a copy of the tuple.
- **Global grouping**: This routes all tuples in a stream to a single task, choosing the task with the lowest task ID value. Note that setting a parallelism hint or number of tasks on a bolt when using the global grouping is meaningless since all tuples will be routed to the same bolt task. The global grouping should be used with caution since it will route all tuples to a single JVM instance, potentially creating a bottleneck or overwhelming a specific JVM/machine in a cluster.
- **None grouping**: The none grouping is functionally equivalent to the shuffle grouping. It has been reserved for future use.
- **Direct grouping**: With a direct grouping, the source stream decides which component will receive a given tuple by calling the `emitDirect()` method. It can only be used on streams that have been declared direct streams.
- **Local or shuffle grouping**: The local or shuffle grouping is similar to the shuffle grouping but will shuffle tuples among bolt tasks running in the same worker process, if any. Otherwise, it will fall back to the shuffle grouping behavior. Depending on the parallelism of a topology, the local or shuffle grouping can increase topology performance by limiting network transfer.

In addition to the predefined groupings, you can define your own stream grouping by implementing the `CustomStreamGrouping` interface:

```java
public interface CustomStreamGrouping extends Serializable {

    void prepare(WorkerTopologyContext context,
                     GlobalStreamId stream, List<Integer> targetTasks);

    List<Integer> chooseTasks(int taskId, List<Object> values);
}
```

**For More Information:**

Distributed Word Count

The `prepare()` method is called at runtime to initiate the grouping with information the grouping implementation can use to make decisions on how to group tuples to receiving tasks. The `WorkerTopologyContext` object provides contextual information about the topology, and the `GlobalStreamId` object provides metadata about the stream being grouped on. The most useful parameter is `targetTasks`, which is a list of all the task identifiers the grouping needs to take into account. You will usually want to store the `targetTasks` parameter as an instance variable for reference in the implementation of the `chooseTasks()` method.

The `chooseTasks()` method returns a list of task identifiers to which a tuple should be sent. Its parameters are the task identifier of the component emitting the tuple and the values of the tuple.

To illustrate the importance of stream groupings, let’s introduce a bug into our topology. Begin by modifying the `nextTuple()` method of `SentenceSpout` so it only emits each sentence once:

```java
public void nextTuple() {
  if(index < sentences.length){
    this.collector.emit(new Values(sentences[index]));
    index++;
  }
  Utils.waitForMillis(1);
}
```

Now run the topology to get the following output:

```
--- FINAL COUNTS ---
a : 2
ate : 2
beverages : 2
cold : 2
cow : 2
dog : 4
don't : 4
fleas : 4
has : 2
have : 2
homework : 2
i : 6
like : 4
man : 2
my : 4
the : 2
think : 2
--------------
```

For More Information:
Now change the field grouping on the CountBolt parameter to a shuffle grouping and rerun the topology:

```java
builder.setBolt(COUNT_BOLT_ID, countBolt, 4)
    .shuffleGrouping(SPLIT_BOLT_ID);
```

The output should look like the following:

```
--- FINAL COUNTS ---
a : 1
ate : 2
beverages : 1
cold : 1
cow : 1
dog : 2
don't : 2
fleas : 1
has : 1
have : 1
homework : 1
i : 3
like : 1
man : 1
my : 1
the : 1
think : 1
--------------
```

Our counts are off because the CountBolt parameter is stateful: it maintains a count for each word it's seen. In this case, the accuracy of our computation depends on the ability to group based on a tuple's content when components have been parallelized. The bug we introduced will only be manifested if the parallelism of the CountBolt parameter is greater than one. This underscores the importance of testing topologies with various parallelism configurations.

In general, you should avoid storing state information in a bolt since any time a worker fails and/or has its tasks reassigned, that information will be lost. One solution is to periodically take a snapshot of state information to a persistent store, such as a database, so it can be restored if a task is reassigned.

For More Information:

Guaranteed processing
Storm provides an API that allows you to guarantee that a tuple emitted by a spout is fully processed. So far in our example, we've not worried about failures. We've seen that a spout stream can be split and can generate any number of streams in a topology, depending on the behavior of downstream bolts. What happens in the event of a failure? As an example, consider a bolt that persists information to tuple data based on a database. How do we handle situations where the database update fails?

Reliability in spouts
In Storm, guaranteed message processing begins with the spout. A spout that supports guaranteed processing needs a way to keep track of tuples it has emitted and be prepared to re-emit a tuple if downstream processing of that tuple, or any child tuples, fails. A child tuple can be thought of as any tuple emitted as a result of a tuple originating from a spout. Another way to look at it is to consider the spout's stream(s) as the trunk of a tuple tree (shown in the following diagram):

In the preceding diagram, the solid lines represent the original trunk tuples emitted by a spout, and the dotted lines represent tuples derived from the original tuple. The resulting graph represents the tuple tree. With guaranteed processing, each bolt in the tree can either acknowledge (ack) or fail a tuple. If all bolts in the tree acknowledge tuples derived from the trunk tuple, the spout's ack method will be called to indicate that message processing is complete. If any of the bolts in the tree explicitly fail a tuple, or if processing of the tuple tree exceeds the time-out period, the spout's fail method will be called.

For More Information:
Storm's ISpout interface defines three methods involved in the reliability API: `nextTuple`, `ack`, and `fail`.

```java
public interface ISpout extends Serializable {
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);
    void close();
    void nextTuple();
    void ack(Object msgId);
    void fail(Object msgId);
}
```

As we've seen before, when Storm requests that a spout emit a tuple, it calls the `nextTuple()` method. The first step in implementing guaranteed processing is to assign the outbound tuple a unique ID and pass that value to the `emit()` method of `SpoutOutputCollector`:

```java
collector.emit(new Values("value1", "value2") , msgId);
```

Assigning the tuple a message ID tells Storm that a spout would like to receive notifications either when the tuple tree is completed or if it fails at any point. If processing succeeds, the spout's `ack()` method will be called with the message ID assigned to the tuple. If processing fails or times out, the spout's `fail` method will be called.

### Reliability in bolts

Implementing a bolt that participates in guaranteed processing involves two steps:

1. Anchoring to an incoming tuple when emitting a derived tuple.
2. Acknowledging or failing tuples that have been processed successfully or unsuccessfully, respectively.

Anchoring to a tuple means that we are creating a link between an incoming tuple and derived tuples such that any downstream bolts are expected to participate in the tuple tree by acknowledging the tuple, failing the tuple, or allowing it to time out.

You can anchor to a tuple (or a list of tuples) by calling one of the overloaded `emit` methods of `OutputCollector`:

```java
collector.emit(tuple, new Values(word));
```
Here, we're anchoring to the incoming tuple and emitting a new tuple that downstream bolts should acknowledge or fail. An alternative form of the `emit` method will emit unanchored tuples:

```
collector.emit(new Values(word));
```

Unanchored tuples do not participate in the reliability of a stream. If an unanchored tuple fails downstream, it will not cause a replay of the original root tuple.

After successfully processing a tuple and optionally emitting new or derived tuples, a bolt processing a reliable stream should acknowledge the inbound tuple:

```
this.collector.ack(tuple);
```

If tuple processing fails in such a way that the spout must replay (re-emit) the tuple, the bolt should explicitly fail the tuple:

```
this.collector.fail(tuple)
```

If tuple processing fails as a result of a time out or through an explicit call, the `OutputCollector.fail()` method, the spout that emitted the original tuple, will be notified, allowing it to re-emit the tuple, as you'll see shortly.

### Reliable word count

To further illustrate reliability, let's begin by enhancing the `SentenceSpout` class to make it support guaranteed delivery. It will need to keep track of all tuples emitted and assign each one a unique ID. We'll use a `HashMap<UUID, Values>` object to store the tuples that are pending. For each tuple we emit, we'll assign a unique identifier and store it in our map of pending tuples. When we receive an acknowledgement, we'll remove the tuple from our pending list. On failure, we'll replay the tuple:

```java
public class SentenceSpout extends BaseRichSpout {

    private ConcurrentHashMap<UUID, Values> pending;
    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "i don't think i like fleas"
    };

}
private int index = 0;

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

public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    this.pending = new ConcurrentHashMap<UUID, Values>();
}

public void nextTuple() {
    Values values = new Values(sentences[index]);
    UUID msgId = UUID.randomUUID();
    this.pending.put(msgId, values);
    this.collector.emit(values, msgId);
    index++;
    if (index >= sentences.length) {
        index = 0;
    }
    Utils.waitForMillis(1);
}

public void ack(Object msgId) {
    this.pending.remove(msgId);
}

public void fail(Object msgId) {
    this.collector.emit(this.pending.get(msgId), msgId);
}

Modifying the bolts to provide guaranteed processing simply involves anchoring outbound tuples to the incoming tuple and then acknowledging the inbound tuple:

public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
}
public void execute(Tuple tuple) {
    String sentence = tuple.getStringByField("sentence");
    String[] words = sentence.split(" ");
    for(String word : words){
        this.collector.emit(tuple, new Values(word));
    }
    this.collector.ack(tuple);
}

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

Summary
In this chapter, we've built a simple distributed computation application using
Storm's core API and covered a large part of Storm's feature set, all without even
installing Storm or setting up a cluster. Storm's local mode is powerful in terms of
productivity and ease of development, but to see Storm's true power and horizontal
scalability, you'll want to deploy applications to a real cluster.

In the next chapter, we'll walk through the process of installing and setting up a
clustered Storm environment and deploying topologies in a distributed environment.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.