Storm Real-time Processing Cookbook

Quinton Anderson

Chapter No. 2
"Log Stream Processing"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Log Stream Processing"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Quinton Anderson is a software engineer with a background and focus on real-time computational systems. His career has been split between building real-time communication systems for defense systems and building enterprise applications within financial services and banking. Quinton does not align himself with any particular technology or programming language, but rather prefers to focus on sound engineering and polyglot development. He is passionate about open source, and is an active member of the Storm community; he has also enjoyed delivering various Storm-based solutions.

Quinton's next area of focus is machine learning; specifically, Deep Belief networks, as they pertain to robotics. Please follow his blog entries on Computational Theory, general IT concepts, and Deep Belief networks for more information.

You can find more information on Quinton via his LinkedIn profile (http://au.linkedin.com/pub/quinton-anderson/37/422/11b/) or more importantly, view and contribute to the source code available at his GitHub (https://github.com/quintona) and Bitbucket (https://bitbucket.org/qanderson) accounts.

For More Information:  
I would like to thank the Storm community for their efforts in building a truly awesome platform for the open source community; a special mention, of course, to the core author of Storm, Nathan Marz.

I would like to thank my wife and children for putting up with my long working hours spent on this book and other related projects. Your effort in making up for my absence is greatly appreciated, and I love you all very dearly. I would also like to thank all those who participated in the review process of this book.

For More Information:
Open source has changed the software landscape in many fundamental ways. There are many arguments that can be made for and against using open source in any given situation, largely in terms of support, risk, and total cost of ownership. Open source is more popular in certain settings than others, such as research institutions versus large institutional financial service providers. Within the emerging areas of web service providers, content provision, and social networking, open source is dominating the landscape. This is true for many reasons, cost being a large one among them. These solutions that need to grow to "Web scale" have been classified as "Big Data" solutions, for want of a better term. These solutions serve millions of requests per second with extreme levels of availability, all the while providing customized experiences for customers across a wide range of services.

Designing systems at this scale requires us to think about problems differently, architect solutions differently, and learn where to accept complexity and where to avoid it. As an industry, we have come to grips with designing batch systems that scale. Large-scale computing clusters following MapReduce, Bulk Synchronous Parallel, and other computational paradigms are widely implemented and well understood. The surge of innovation has been driven and enabled by open source, leaving even the top software vendors struggling to integrate Hadoop into their technology stack, never mind trying to implement some level of competition to it.

Customers, however, have grown an insatiable desire for more. More data, more services, more value, more convenience, and they want it now and at lower cost. As the sheer volume of data increases, the demand for real-time response time increases too. The next phase of computational platforms has arrived, and it is focused on real time, at scale. It represents many unique challenges, and is both theoretically and practically challenging.

This cookbook will help you master a platform, the Storm processor. The Storm processor is an open source, real-time computational platform written by Nathan Marz at Backtype, a social analytics company. It was later purchased by Twitter and released as open source. It has since thrived in an ever-expanding open source community of users, contributors, and success stories within production sites. At the time of writing this preface, the project was enjoying more than 6,000 stars on GitHub, 3,000 Twitter followers, has been benchmarked at over a million transactions per second per node, and has almost 80 reference customers with production instances of Storm.

These are extremely impressive figures. Moreover, you will find by the end of this book that it is also extremely enjoyable to deliver systems based on Storm, using whichever language is congruent with your way of thinking and delivering solutions.

For More Information:
This book is designed to teach you Storm with a series of practical examples. These examples are grounded in real-world use cases, and introduce various concepts as the book unfolds. Furthermore, the book is designed to promote DevOps practice around the Storm technology, enabling the reader to develop Storm solutions and deliver them reliably into production, where they create value.

**An introduction to the Storm processor**

A common criticism of open source projects is their lack of documentation. Storm does not suffer from this particular issue; the documentation for the project is excellent, well-written, and well-supplemented by the vibrant user community. This cookbook does not seek to duplicate this documentation but rather supplement it, driven largely by examples with conceptual and theoretical discussion where required. It is highly recommended that the reader take the time to read the Storm introductory documentation before proceeding to *Chapter 1, Setting Up Your Development Environment*, specifically the following pages of the Storm wiki:

- https://github.com/nathanmarz/storm/wiki/Rationale
- https://github.com/nathanmarz/storm/wiki/Concepts
- https://github.com/nathanmarz/storm/wiki/Understanding-theparallelism-of-a-Storm-topology

**What This Book Covers**

*Chapter 1, Setting Up Your Development Environment*, will demonstrate the process of setting up a local development environment for Storm; this includes all required tooling and suggested development workflows.

*Chapter 2, Log Stream Processing*, will lead the reader through the process of creating a log stream processing solution, complete with a base statistics dashboard and log-searching capability.

*Chapter 3, Calculating Term Importance with Trident*, will introduce the reader to Trident, a data-flow abstraction that works on top of Storm to enable highly productive enterprise data pipelines.

*Chapter 4, Distributed Remote Procedure Calls*, will teach the user how to implement distributed remote procedure calls.

*Chapter 5, Polyglot Topology*, will guide the reader to develop a Polyglot technology and add new technologies to the list of already supported technologies.

*Chapter 6, Integrating Storm with Hadoop*, will guide the user through the process of integrating Storm with Hadoop, thus creating a complete Lambda architecture.

For More Information:  
Chapter 7, *Real-time Machine Learning*, will provide the reader with a very basic introduction to machine learning as a topic, and provides various approaches to implementing it in realtime projects based on Storm.

Chapter 8, *Continuous Delivery*, will demonstrate how to set up a Continuous Delivery pipeline and deliver a Storm cluster reliably into an environment.

Chapter 9, *Storm on AWS*, will guide the user through various approaches to automated provisioning of a Storm cluster into the Amazon Computing Cloud.
In this chapter we will cover:

- Creating a log agent
- Creating the log spout
- Rule-based analysis of the log stream
- Indexing and persisting the log data
- Counting and persisting log statistics
- Creating an integration test for the log stream cluster
- Creating a log analytics dashboard

Introduction

This chapter will present an implementation recipe for an enterprise log storage and a search and analysis solution based on the Storm processor. Log data processing isn't necessarily a problem that needs solving again; it is, however, a good analogy.

Stream processing is a key architectural concern in the modern enterprise; however, streams of data are often semi-structured at best. By presenting an approach to enterprise log processing, this chapter is designed to provide the reader with all the key elements to achieve this level of capability on any kind of data. Log data is also extremely convenient in an academic setting given its sheer abundance. A key success factor for any stream processing or analytics effort is a deep understanding of the actual data and sourcing data can often be difficult.

It is, therefore, important that the reader considers how the architectural blueprint could be applied to other forms of data within the enterprise.

For More Information:
Creating a log agent

Modern enterprise architectures consist of a huge number of solutions, each comprising many nodes. Some MapReduce clusters contain hundreds of nodes. Each node contains an array of applications and services, both at the operating system and Application layers. These services and applications generate varying volumes of log data. There is an increasing recognition of the importance of log data within the enterprise community for the following reasons:

- It is a key source of information for any IT operations team to maintain available systems
- It is key to finding and resolving problems, both in the production and systems testing phases
- It is increasingly becoming a source of business value, where valuable business data is trapped in this semi-structured data, including:
  - Risk and compliance data
  - Business operational data
  - Web analytics
  - Security
  - Financial forecasts

In order to leverage valuable log data, it must be sourced from these nodes and delivered securely and easily to a centralized log service for storage, indexing, and analysis. This recipe demonstrates how to achieve this through an open source log agent called logstash.
There are many good commercial and open source log solutions available. This chapter uses portions of logstash; further logstash recipes can be found at http://cookbook.logstash.net/ and http://logstash.net/docs/1.1.13/tutorials/getting-started-centralized. A good commercial equivalent is Splunk (http://www.splunk.com/).

How to do it...

1. To start, the logs on your local node will be streamed into the topology. Start by downloading and configuring logstash as follows:

   wget https://logstash.objects.dreamhost.com/release/logstash-1.1.7-monolithic.jar

2. Then, using your favorite text editor, create a file called shipper.conf containing the following:

   ```
   input {
     file {
       type => "syslog"

       # Wildcards work here :)
       path => [ "/var/log/messages", "/var/log/system.*", "/var/log/*.log" ]
     }
   }
   
   output {
     # Output events to stdout for debugging. Feel free to remove
     # this output if you don't need it.
     stdout {}

     redis {
       host => "localhost"
       data_type => "list"
       key => "rawLogs"
     }
   }
   ```

3. After starting a local instance of Redis, you can start this logging agent by issuing the following command:

   ```
   java –jar logstash-1.1.7-monolithic.jar -f shipper.conf
   ```

For More Information:
**How it works...**

logstash implements a very simple model of input-filter-output with an ever-expanding list of plugins for any of these three elements. The preceding configuration file (shipper.conf) configures at least one input and output.

The file input plugin will tail files based on filenames or wildcards in the specified paths. Many file inputs can be configured, each with a different type. The log type is important for later processing and categorization. As we have not configured any filters in this configuration file, the raw log will be passed to the output plugin. The output plugin is the Redis plugin that will output the log to the Redis instance on localhost to a list called rawLogs.

logstash can easily be included in the baseline provisioning of each node provisioned on your infrastructure, including key exchange for secure log delivery via any transport mechanism you are comfortable with.

**Creating the log spout**

The log topology will read all logs through the Redis channel that is fed by logstash; these logs will be emitted into the topology through the spout described in this recipe. As this is a new topology, we must first create the new topology project.

**How to do it...**


1. Create the POM as per the Creating a "Hello World" topology recipe in Chapter 1, Setting Up Your Development Environment, updating the <artifactId> and <name> tag values and including the following dependencies:

   ```xml
   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.11</version>
     <scope>test</scope>
   </dependency>
   <dependency>
     <groupId>org.slf4j</groupId>
     <artifactId>slf4j-log4j12</artifactId>
     <version>1.6.1</version>
   </dependency>
   ``
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-legacy</artifactId>
  <version>2.5.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>storm</groupId>
  <artifactId>storm</artifactId>
  <version>0.8.1</version>
  <!-- keep storm out of the jar-with-dependencies -->
  <scope>provided</scope>
  <exclusions>
    <exclusion>
      <artifactId>slf4j-api</artifactId>
      <groupId>org.slf4j</groupId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.googlecode.json-simple</groupId>
  <artifactId>json-simple</artifactId>
  <version>1.1</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.1.0</version>
</dependency>
<dependency>
  <groupId>commons-httpclient</groupId>
  <artifactId>commons-httpclient</artifactId>
  <version>3.1</version>
</dependency>
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-junit4</artifactId>
  <version>2.5.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.github.ptgoetz</groupId>
  <artifactId>storm-cassandra</artifactId>
  <version>0.3.1-SNAPSHOT</version>
</dependency>

For More Information:
Log Stream Processing

<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch</artifactId>
  <version>0.20.2</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-core</artifactId>
  <version>5.5.0.Final</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-compiler</artifactId>
  <version>5.5.0.Final</version>
</dependency>

2. Import the project into Eclipse after generating the Eclipse project files as follows:

   mvn eclipse:eclipse

3. Tuples in the log topology will carry a log domain object that encapsulates the data and parsing logic for a single log record or an entry in a logfile. In the created project, create this domain object:

   public class LogEntry {
      public static Logger LOG = Logger.getLogger(LogEntry.class);
      private String source;
      private String type;
      private List<String> tags = new ArrayList<String>();
      private Map<String,String> fields = new HashMap<String, String>();
      private Date timestamp;
      private String sourceHost;
      private String sourcePath;
      private String message = ""
      private boolean filter = false;
      private NotificationDetails notifyAbout = null;
      private static String[] FORMATS = new String[] {"yyyy-MM-dd'T'HH:mm:ss.SSS",
         "yyyy.MM.dd G 'at' HH:mm:ss z",
         "yyyyy.MMMMM.dd GGG hh:mm aaa",
         "EEE, d MMM yyyy HH:mm:ss Z",
         "yyMMddHHmmssZ"};
   }

For More Information:

@SuppressWarnings("unchecked")
public LogEntry(JSONObject json){
    source = (String)json.get("@source");
    timestamp =
        parseDate((String)json.get("@timestamp"));
    sourceHost = (String)json.get("@source_host");
    sourcePath = (String)json.get("@source_path");
    message = (String)json.get("@message");
    type = (String)json.get("@type");

    JSONArray array = (JSONArray)json.get("@tags");
    tags.addAll(array);
    JSONObject fields = (JSONObject)json.get("@fields");
    fields.putAll(fields);
}

public Date parseDate(String value){
    SimpleDateFormat format = new
        SimpleDateFormat(FORMATS[i]);
    for(int i = 0; i < FORMATS.length; i++){
        Date temp;
        try {
            temp = format.parse(value);
            if(temp != null)
                return temp;
        } catch (ParseException e) {} 
    }
    LOG.error("Could not parse timestamp for log");
    return null;
}

@SuppressWarnings("unchecked")
public JSONObject toJSON(){
    JSONObject json = new JSONObject();
    json.put("@source", source);
    json.put("@timestamp",DateFormat
        .getDateTimeInstance().format(timestamp));
    json.put("@source_host",sourceHost);
    json.put("@source_path",sourcePath);
}

For More Information:
json.put("@message", message);
json.put("@type", type);
JSONArray temp = new JSONArray();
temp.addAll(tags);
json.put("@tags", temp);
JSONArray fieldTemp = new JSONArray();
fieldTemp.putAll(fields);
json.put("@fields", fieldTemp);
return json;
}

The getter, setter, and equals methods have been excluded from this code snippet; however, they must be implemented in order. The equals method is vital for unit testing purposes.

4. Then create the Logspout class that extends the BaseRichSpout interface and implements the same pattern as described in Chapter 1, Setting Up Your Development Environment, declaring a single field as follows:

```
outputFieldsDeclarer.declare(new Fields(FieldNames.LOG_ENTRY));
```

And then emitting the received log entries into the topology as follows:

```
public void nextTuple() {
    String content = jedis.rpop(LOG_CHANNEL);
    if(content==null || "nil".equals(content)) {
        //sleep to prevent starving other threads
        try { Thread.sleep(300); } catch (InterruptedException e) {}
    } else {
        JSONObject obj = (JSONObject) JSONValue.parse(content);
        LogEntry entry = new LogEntry(obj);
        collector.emit(new Values(entry));
    }
}
```

Literals should be avoided in the code as far as possible; tuples allow for effective runtime coupling; however, peppering code with field name literals for elements that are effectively coupled prior to runtime doesn’t add any value. Hence the usage of static field name definitions.

For More Information:
How it works...

The Redis spout implementation is already familiar; the key logic implemented in this recipe is the parsing logic within the domain object of the `LogEntry` class. logstash submits log lines as separate JSON values into the Redis channel. These JSON values are in the following format:

```
{
  "@source":"file://PATH",
  "@tags":[],
  "@fields":{},
  "@timestamp":"yyyy-MM-ddThh:mm:ss.SSS",
  "@source_host":"hostname",
  "@source_path":"path",
  "@message":"Syslog log line contents",
  "@type":"syslog"
}
```

There's more...

The JSON format provides for two key structures, namely the JSON object and the JSON array. The JSON website (www.json.org) provides a concise definition of each and has also been provided here for the sake of convenience. An object is an unordered set of name/value pairs. An object begins with `{` (left brace) and ends with `}` (right brace). Each name is followed by `:` (colon) and the name/value pairs are separated by `,` (comma).

```
object
  |  string         |  value
  v   +           +
  1   2           1
```

An array is an ordered collection of values. An array begins with `[` (left bracket) and ends with `]` (right bracket). The values are separated by `,` (comma).

```
array
  |  value
  v   +
  1   1
```

A value can be a string in double quotes; a number; true, false, or null; an object; or an array. These structures can be nested.

For More Information:
Log Stream Processing

The constructor of the LogEntry object takes JSONObject as the only parameter and initializes its internal values based on the contained values. The LogEntry object is also able to convert itself into a JSONObject through the toJSON() method, which will become useful later. LogEntry makes heavy use of the com.googlecode.json-simple library in order to achieve the first level of parsing from string to a workable structure.

Although the structure is well-defined, the date-time format can vary. The parseDate() method, therefore, provides a best-effort approach to parse the date. A static list of supported date-time formats is defined as the FORMATS class member variable.

Rule-based analysis of the log stream

Any reasonable log management system needs to be able to achieve the following:

- Filter logs that aren't important, and therefore should not be counted or stored. These often include log entries at the INFO or DEBUG levels (yes, these exist in production systems).
- Analyze the log entry further and extract as much meaning and new fields as possible.
- Enhance/update the log entry prior to storage.
- Send notifications on when certain logs are received.
- Correlate log events to derive new meaning.
- Deal with changes in the log's structure and formatting.

This recipe integrates the JBoss Library and Drools into a bolt to make these goals easily achievable in a declarative and clear manner. Drools is an open source implementation of a forward-chaining rules engine that is able to infer new values and execute the logic based on matching logic. You can find more details on the Drools project at http://www.jboss.org/drools/.

How to do it...

1. Within Eclipse, create a class called LogRulesBolt, which extends BaseRichBolt, within the storm.cookbook.log package. As with the LogSpout class, the LogRulesBolt class will emit a single value containing a LogEntry instance.
   
   declarer.declare(new Fields(FieldNames.LOG_ENTRY));

2. Add a private member-level variable ksession of the StatelessKnowledgeSession class and initialize it within the bolt's prepare method.
   
   KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
   newKnowledgeBuilder();

For More Information:
kbuilder.add( ResourceFactory.newClassPathResource("/Syslog.drl", getClass() ), ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
    LOG.error( kbuilder.getErrors().toString() );
}
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
ksession = kbase.newStatelessKnowledgeSession();

The initialization of this knowledge session includes only a single set of rules for the syslog logs. It is recommended that rules management be extracted out into Drools Guvnor, or similar, and rules resources be retrieved via an agent. This is outside the scope of this book but more details are available at the following link:
http://www.jboss.org/drools/drools-guvnor.html

3. In the bolt's execute method, you need to pass the LogEntry object from the tuple into the knowledge session.

LogEntry entry = (LogEntry)input.getValueByField(FieldNames.LOG_ENTRY);
if(entry == null){
    LOG.fatal( "Received null or incorrect value from tuple" );
    return;
}
ksession.execute( entry );
if(!entry.isFilter()){
    collector.emit(new Values(entry));
}

4. You next need to create the rules resource file; this can simply be done with a text editor or using the Eclipse plugin available from the update site (http://download.jboss.org/drools/release/5.5.0.Final/org.drools.updatesite/). The rules resource file should be placed at the root of the classpath; create the file named Syslog.drl in src/main/resources and add this folder to the build path within Eclipse by right-clicking on the folder and going to Build Path | Use as source folder.

5. Add the following content to the rules resource:

package storm.cookbook.log.rules

import storm.cookbook.log.model.LogEntry;

For More Information:
import java.util.regex.Matcher
import java.util.regex.Pattern

rule "Host Correction"
    when
        l: LogEntry(sourceHost == "localhost")
    then
        l.setSourceHost("localhost.example.com");
end

rule "Filter By Type"
    when
        l: LogEntry(type != "syslog")
    then
        l.setFilter(true);
end

rule "Extract Fields"
    salience 100//run later
    when
        l: LogEntry(filter != true)
    then
        String logEntryPattern = "^([^\d.]+) (\S+) (\S+) \[(\[\w:/\]+\s[+\-]\d{4})\] "(.+?)" (\d{3}) (\d+) "([^""]+)" "([^""]+)";  
        Matcher matcher = Pattern.compile(logEntryPattern).matcher(l.getMessage());
        if(matcher.find()){
            l.addField("_pid",matcher.group(1));
            l.addField("_src",matcher.group(2));
        }
end

How it works...

Drools supports two types of knowledge sessions, namely **stateful** and **stateless**. For this use case, a stateless session is all that is required.

Stateful sessions are always to be used with caution as they can lead to performance problems. They essentially maintain facts in memory between session executions. There are use cases where this is vital; however, the nature of a forward-chaining rete engine is that it will degrade in performance exponentially as facts are added to its knowledge base.

For More Information:
A knowledge session is used to evaluate facts against a known set of rules. This is set up within the `prepare` method of the bolt, with the rules provided at that point. Within the execution of the bolt, `LogEntry` is extracted from the tuple and passed into the knowledge session through the following call:

```java
ksession.execute(entry);
```

The knowledge session will act as an entry during execution and we can expect it to be potentially different once the call has completed. Contained within the `LogEntry` object is a control field called `filter`. If a rule sets this to `true`, the log entry is to be dropped; this is implemented by checking prior to emitting a tuple containing the entry after the rules execution.

```java
if(!entry.isFilter()){
    collector.emit(new Values(entry));
}
```

Within the rules resource file, there are three rules currently defined.

- Host Correction
- Filter By Type
- Extract Fields

These rules are implemented for demonstration purposes only and aren't necessarily viable production rules. The `Host Correction` rule tries to correct the host name value such that it is fully qualified. The autonomy of a rule is that, when a matching criterion is met, the result is displayed. The `when` clause of this rule will match against the `LogEntry` instance whose `sourceHost` field is `localhost`.

```java
l: LogEntry(sourceHost == "localhost")
```

This clause also assigns any matching instance to a local variable `l` within the scope of this rule. The functionality specified in the `then` clause is simply plain old Java, which is added into the current classpath after compilation at runtime. These rules imply making the `localhost` value fully qualified.

```java
l.setSourceHost("localhost.example.com");
```

The `Filter By Type` rule will set the filter to `true` for all entries whose type doesn't match `syslog`.

The `Extract Fields` rule is more interesting. Firstly, because it includes a salience value, which ensures it is evaluated last. This ensures that it never extracts fields from filtered logs. It then uses regular expression matching to extract more fields and structure from the log file. While regular expressions are outside the scope of this book, they are widely understood and well documented.

For More Information:
Log Stream Processing

For completeness's sake, here are some more useful examples of expressions for log entries:

- Match the date: `((?<=\d\x20))\w+\x20\d+`
- Match the time: `((?<=\d\x20))\d+:\d+:\d+`
- Match an IP address: `((?<=[,])\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})`
- Match the protocol: `((?<=\[\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\s))+\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})`

Further reading on regular expressions can be found at Wikipedia:

http://en.wikipedia.org/wiki/Regular_expression

These extra fields can then be added to the fields or tags and used later for analysis or search and grouping.

Drools also includes a module called Drools Fusion that essentially supports Complex Event Processing (CEP). It is often referred to as an emerging enterprise approach, which may be true, but practically it simply means that the rules engine understands temporal concerns. Using temporal operators, it can correlate events over time and derive new knowledge or trigger actions. These temporal operators are supported based on the bolt implementation in this recipe. For more information, browse to the following link:

http://www.jboss.org/drools/drools-fusion.html

Indexing and persisting the log data

Log data needs to be stored for some defined period of time in order to be useful; it also needs to be searchable. In order to achieve this, the recipe integrates with an open source product call Elastic Search, which is a general-use, clustered search engine with a RESTful API (http://www.elasticsearch.org/).

How to do it...

1. Create a new BaseRichBolt class called IndexerBolt and declare the `org.elasticsearch.client.Client` client as a private member variable. You must initialize it as follows within the prepare method:

```java
if((Boolean)stormConf.get(backtype.storm.Config.TOPOLOGY_DEBUG) ==
  true){
    node = NodeBuilder.nodeBuilder().local(true).node();
  } else {
    String clusterName = (String) stormConf.get(Conf.ELASTIC_]
    CLUSTER_NAME);
```

For More Information:

if(clusterName == null)
    clusterName = Conf.DEFAULT_ELASTIC_CLUSTER;
node = NodeBuilder.nodeBuilder().
    clusterName(clusterName).node();
client = node.client();

2. The LogEntry object can then be indexed during the execute method of the bolt:

LogEntry entry = (LogEntry)input.getValueByField(FieldNames.LOG_ENTRY);
if(entry == null){
    LOG.fatal( "Received null or incorrect value from tuple" );
    return;
}
String toBeIndexed = entry.toJSONString();
IndexResponse response = client.prepareIndex(INDEX_NAME,INDEX_TYPE)
    .setSource(toBeIndexed)
    .execute().actionGet();
if(response == null)
    LOG.error("Failed to index Tuple: " + input.toString());
else{
    if(response.getId() == null)
        LOG.error("Failed to index Tuple: " + input.toString());
    else{
        LOG.debug("Indexing success on Tuple: " + input.toString());
        collector.emit(new Values(entry,response.getId()));
    }
}

3. The unit test of this bolt is not obvious; it is therefore worthwhile to give some explanation here. Create a new JUnit 4 unit test in your test source folder under the storm.cookbook.log package. Add a private inner class called StoringMatcher as follows:

private static class StoringMatcher extends BaseMatcher<Values> {
    private final List<Values> objects = new ArrayList<Values>();
    @Override
    public boolean matches(Object item) {
        if (item instanceof Values) {
            objects.add((Values) item);
    
For More Information:
Log Stream Processing

```
return true;
}
return false;

@Override
public void describeTo(Description description) {
    description.appendText("any integer");
}

public Values getLastValue() {
    return objects.remove(0);
}
```

4. Then implement the actual test method as follows:

```
@Test
public void testIndexing() throws IOException {
    //Config, ensure we are in debug mode
    Map config = new HashMap();
    config.put(backtype.storm.Config.TOPOLOGY_DEBUG, true);
    Node node = NodeBuilder.nodeBuilder()
        .local(true).node();
    Client client = node.client();
    final OutputCollector collector =
        context.mock(OutputCollector.class);
    IndexerBolt bolt = new IndexerBolt();
    bolt.prepare(config, null, collector);
    final LogEntry entry = getEntry();
    final Tuple tuple = getTuple();
    final StoringMatcher matcher = new StoringMatcher();
    context.checking(new Expectations(){
        oneOf(tuple).getValueByField(FieldNames.LOG_ENTRY);will
        (returnValue(entry));
        oneOf(collector).emit(with(matcher));
    });
    bolt.execute(tuple);
    context.assertIsSatisfied();
    //get the ID for the index
    String id = (String) matcher.getLastValue().get(1);
```
//Check that the indexing working
GetResponse response =
client.prepareGet(IndexerBolt.INDEX_NAME,
IndexerBolt.INDEX_TYPE,id)
  .execute()
  .actionGet();
assertTrue(response.exists());
}

How it works...

Elastic Search provides a complete client API for Java (given that it is implemented in Java), making integration with it quite trivial. The prepare method of the bolt will create a cluster node in either the local or clustered mode. The cluster mode will join an existing cluster based on the name provided with a local storage node being created on the current node; this prevents the double-hop latency of a write over a different transport.

Elastic Search is a large complex system in its own right; it is recommended that you read the provided documentation in order to understand the operational and provisioning concerns.

When Storm is in the debug mode, the Elastic Search node will run an embedded cluster, with many nodes (if requested) being executed within the same JVM. This is obviously useful for unit testing purposes. This is all enabled in the prepare method of the bolt.

if((Boolean)stormConf.get(backtype.storm.Config.TOPOLOGY_DEBUG) ==
true){
    node = NodeBuilder.nodeBuilder().local(true).node();
} else {

When a tuple is received, the LogEntry object is extracted and the JSON contents of LogEntry are sent to Elastic Search.

String toBeIndexed = entry.toJSON().toJSONString();
IndexResponse response =
    client.prepareIndex(INDEX_NAME,INDEX_TYPE)
    .setSource(toBeIndexed)
    .execute().actionGet();

The ID of the log within the Elastic Search cluster is then extracted from the response and emitted with the LogEntry objects to downstream bolts. In this particular recipe, we will only use this value for unit testing; however, downstream bolts could easily be added to persist this value against some log statistics that would be extremely useful within a user interface for drilldown purposes.

    collector.emit(new Values(entry,response.getId()));

For More Information:
Log Stream Processing

The unit test for this particular bolt is quite tricky. This is because, in a typical unit test, we know what the expected outcome is before we run the test. In this case, we don't know the ID until we have received the response from the Elastic Search cluster. This makes expressing expectations difficult, especially if we want to validate the log in the search engine. To achieve this, we make use of a custom matcher for JMock. The key method in the custom matcher is the matches method.

```java
public boolean matches(Object item) {
    if (item instanceof Values) {
        objects.add((Values) item);
        return true;
    }
    return false;
}
```

This method simply ensures that an instance of Values is returned but it also holds onto the value for later evaluation. This allows us to set the following set of expectations:

```java
context.checking(new Expectations(){
    oneOf(tuple).getValueByField(FieldNames.LOG_ENTRY);will(returnValue(entry));
    oneOf(collector).emit(with(matcher));
});
```

And then retrieve the record ID and validate it against the embedded Elastic Search cluster.

```java
String id = (String) matcher.getLastValue().get(1);
GetResponse response = client.prepareGet(IndexerBolt.INDEX_NAME,IndexerBolt.INDEX_TYPE,id)
    .execute().actionGet();
assertTrue(response.exists());
```

If you would like to be able to search the logfiles in the cluster, download and install the excellent log search front engine, Kibana, from kibana.org. This recipe has maintained the JSON log structure from logstash and Kibana is designed as the frontend for logstash on Elastic Search; it will work seamlessly with this recipe. It also uses the Twitter Bootstrap GUI framework, meaning that you can integrate it with the analytics dashboard quite easily.

Counting and persisting log statistics

There are many statistics that can be gathered for log streams; for the purposes of this recipe and to illustrate the concept, only a single-time series will be dealt with (log volume per minute); however, this should fully illustrate the design and approach for implementing other analyses.

For More Information:
1. Download and install the storm-cassandra contrib project into your Maven repository:
   
   ```
   git clone https://github.com/quintona/storm-cassandra
   cd storm-cassandra
   mvn clean install
   ```

2. Create a new BaseRichBolt class called VolumeCountingBolt in the storm. cookbook.log package. The bolt must declare three output fields:
   
   ```java
   public void declareOutputFields(OutputFieldsDeclarer declarer) {
       declarer.declare(new Fields(FIELD_ROW_KEY,
                                   FIELD_COLUMN, FIELD_INCREMENT));
   }
   ```

3. Then implement a static utility method to derive the minute representation of the log's time:
   
   ```java
   public static Long getMinuteForTime(Date time) {
       Calendar c = Calendar.getInstance();
       c.setTime(time);
       c.set(Calendar.SECOND, 0);
       c.set(Calendar.MILLISECOND, 0);
       return c.getTimeInMillis();
   }
   ```

4. Implement the execute method (yes, it is that short):
   
   ```java
   LogEntry entry = (LogEntry) input.getValueByField(FieldNames.LOG_ENTRY);
   collector.emit(new Values(getMinuteForTime(entry.getTimestamp()), entry.getSource(), 1L));
   ```

5. Finally, create the LogTopology class as per the pattern presented in Chapter 1, Setting Up Your Development Environment, and create the topology as follows:
   
   ```java
   builder.setSpout("logSpout", new LogSpout(), 10);
   builder.setBolt("indexer", new IndexerBolt(), 10).shuffleGrouping("logRules");
   CassandraCounterBatchingBolt logPersistenceBolt = new CassandraCounterBatchingBolt(
       Conf.COUNT_CF_NAME, VolumeCountingBolt.FIELD_ROW_KEY,
   ```
Log Stream Processing

```java
VolumeCountingBolt.FIELD_INCREMENT );
logPersistenceBolt.setAckStrategy
(AckStrategy.ACK_ON_RECEIVE);
builder.setBolt("countPersistor",
    logPersistenceBolt, 10).shuffleGrouping("counter");

conf.put(Conf.REDIS_PORT_KEY, Conf.DEFAULT_JEDIS_PORT);
conf.put(CassandraBolt.CASSANDRA_KEYSPACE, Conf.LOGGING_KEYSPACE);
```

**How it works...**

This implementation looks surprisingly simple, and it is. It makes use of the storm-cassandra project (a type of storm-contrib) to abstract all the persistence complexity away. The storm-cassandra project integrates Storm and Cassandra by providing a generic and configurable backtype.storm.Bolt implementation that writes the StormTuple objects to a Cassandra column family.

Cassandra is a column family database (http://cassandra.apache.org/). Cassandra's column family data model offers the convenience of column indexes with the performance of log-structured updates, strong support for materialized view, and powerful built-in caching. A recent addition to the Cassandra functionality is that of counter columns, which are essentially persistent columns, within a given column family, that can be incremented safely from anywhere in the cluster.

The storm-cassandra project provides two styles of persistence, firstly for standard Cassandra column families and secondly for counter-based columns. We will focus on the second type, as it is appropriate for our use case; you can read about the other style on the project's README file but it is essentially the same.

An instance of the CassandraCounterBatchingBolt class does all the work for us. It expects to be told which column family to use, which tuple field to use for the row key and which tuple field to use for the increment amount. It will then increment columns by that amount based on the remaining fields in the tuple.

Consider the following constructor:

```java
CassandraCounterBatchingBolt logPersistenceBolt = new CassandraCounterBatchingBolt("columnFamily", "RowKeyField", "IncrementAmountField" );
```

And the following tuple as input:

```java
{rowKey: 12345, IncrementAmount: 1L, IncrementColumn: 'SomeCounter'}
```

This will increment the SomeCounter counter column by 1L in the columnFamily column family.

For More Information:

A key mind shift for any developer with a relational database background is data modeling in a column family database. Column family databases, as part of the big data family of databases, promote the use of highly denormalized data models. This approach removes table relationships and their locking concerns and enables massive-scale parallel read and write processing on the database. This promotes data duplication; however, given the cost of commodity disks, this is seen as a small sacrifice in order to meet the scaling objectives of today. The mind shift is to think of data models in terms of the queries that we will perform on the dataset, rather than modeling real-world entities into concise normalized structures. The query we are trying to answer with this data model is essentially this: select all total count for all logfiles for a given point in time.

This approach allows us to easily derive this data by emitting a tuple to count that total; we can easily emit a tuple to answer any other question, and examples could include the following:

- What do my volume trends look like across any given time period, be it day, month, or year?
- What are the most popular stems within my logfiles?

Column families can contain more than counts, design any denormalized structure, and emit a tuple to represent a set of columns for that row; if the row already exists, the columns will simply be added or updated. This can become extremely powerful.

### Creating an integration test for the log stream cluster

Integration testing is obviously a vital task in the delivery process. There are many types of integration testing. Unit integration testing involves integration testing a topology, typically as part of the continuous integration build cycle, and should be seen as complementary to the necessary functional style of integration testing of a deployed cluster. The integration test presented here is essentially the same as that of the integration test presented in Chapter 1, Setting Up Your Development Environment; however, it is sufficiently complex to warrant an explanation here.

### How to do it...

Start by creating the unit test.

1. Using Eclipse, create a JUnit 4 test case called `IntegrationTestTopology` under the unit testing source folder of your project in the `storm.cookbook.log` package. Add a setup method that should be invoked before the class:

```java
@BeforeClass
public static void setup() throws Exception {
    // Insert setup code here
}
```

For More Information:  
2. Then create each of the associated setup methods; first set up an embedded version of Cassandra:

```java
private static void setupCassandra() throws Exception {
    cassandra = new EmbeddedCassandra(9171);
    cassandra.start();
    // Allow some time for it to start
    Thread.sleep(3000);

    AstyanaxContext<Cluster> clusterContext = new AstyanaxContext.Builder()
        .forCluster("ClusterName")
        .withAstyanaxConfiguration(
            new AstyanaxConfigurationImpl()
                .setDiscoveryType(NodeDiscoveryType.NONE))
        .withConnectionPoolConfiguration(
            new ConnectionPoolConfigurationImpl("MyConnectionPool")
                .setMaxConnsPerHost(1).setSeeds(
                    "localhost:9171")
                .withConnectionPoolMonitor(new CountingConnectionPoolMonitor()))
        .buildCluster(ThriftFamilyFactory.getInstance());

    clusterContext.start();
    Cluster cluster = clusterContext.getEntity();
    KeyspaceDefinition ksDef = cluster.makeKeyspaceDefinition();

    Map<String, String> stratOptions = new HashMap<String, String>();
    stratOptions.put("replication_factor", "1");
    ksDef.setName(Conf.LOGGING_KEYSPACE)
        .setStrategyClass("SimpleStrategy")
        .setStrategyOptions(stratOptions)
        .addColumnFamily(
            cluster.makeColumnFamilyDefinition()
                .setName(Conf.COUNT_CF_NAME)
                .setComparatorType("UTF8Type")
                .setKeyValidationClass("UTF8Type")
                .setDefaultValidationClass("CounterColumnType"));
```

For More Information:

3. Then set up a local, embedded instance of Elastic Search:

```java
private static void setupElasticSearch() throws Exception {
    Node node = NodeBuilder.nodeBuilder().local(true).node();
    client = node.client();
    //allow time for the node to be available
    Thread.sleep(5000);
}
```

4. Finally, set up the actual topology to be tested:

```java
private static void setupTopology() {
    // We want all output tuples coming to the mock for
    // testing purposes
    topology.getBuilder().setBolt("testBolt", testBolt, 1)
        .globalGrouping("indexer");
    // run in local mode, but we will shut the cluster
    // down when we are finished
    topology.runLocal(0);
    // jedis required for input and output of the cluster
    jedis = new Jedis("localhost",
        Integer.parseInt(Conf.DEFAULT_JEDIS_PORT));
    jedis.connect();
    jedis.flushDB();
    // give it some time to startup before running the
    // tests.
    Utils.sleep(5000);
}
```

5. This will set up the fixtures we require in order to test our topology; we also need to
   shut these down gracefully at the end of the test, so add the AfterClass method
   for the test suite:

```java
@AfterClass
public static void shutDown() {
    topology.shutDownLocal();
    jedis.disconnect();
    client.close();
    cassandra.stop();
}
```

---

For More Information:
6. Finish off by implementing the actual test case:

```java
@Test
public void inputOutputClusterTest() throws Exception {
    String testData = UnitTestUtils.readFile("/testData1.json");
    jedis.rpush("log", testData);
    LogEntry entry = UnitTestUtils.getEntry();
    long minute = VolumeCountingBolt.getMinuteForTime(entry.getTimestamp());
    Utils.sleep(6000);
    String id = jedis.rpop(REDIS_CHANNEL);
    assertNotNull(id);
    // Check that the indexing working
    GetResponse response = client
        .prepareGet(IndexerBolt.INDEX_NAME,
                    IndexerBolt.INDEX_TYPE, id).execute().actionGet();
    assertTrue(response.exists());
    // now check that count has been updated in cassandra
    AstyanaxContext<Keyspace> astyContext = new AstyanaxContext.Builder()
        .forCluster("ClusterName")
        .forKeyspace(Conf.LOGGING_KEYSPACE)
        .withAstyanaxConfiguration(
            new AstyanaxConfigurationImpl()
        .setDiscoveryType(NodeDiscoveryType.NONE))
        .withConnectionPoolConfiguration(
            new ConnectionPoolConfigurationImpl("MyConnectionPool")
        .setMaxConnsPerHost(1).setSeeds(
            "localhost:9171")
        .withConnectionPoolMonitor(
            new CountingConnectionPoolMonitor())
        .buildKeyspace(ThriftFamilyFactory.getInstance());
    astyContext.start();
    Keyspace ks = astyContext.getEntity();
    Column<String> result = ks.prepareQuery(
        new ColumnFamily<String, String>(
            Conf.COUNT_CF_NAME, StringSerializer.get(),
            StringSerializer.get()))
```

For More Information:

This test case works by creating embedded instances of the required clusters for this topology, namely Cassandra and Elastic Search. As with the previous integration test, it then injects test data into the input channel and allows the log entry to flow through the topology, after which it validates the entry in the search engine and validates that the counter has incremented appropriately.

This test will take longer to run than a standard unit test and therefore should not be included in your standard Maven build. The test should, however, be used as part of your local development workflow and validated further on a continuous integration server.

Creating a log analytics dashboard

The log analytics dashboard is a web application that presents aggregated data to the user, typically in a graphical manner. For achieving this, we must take cognizance of the following user interface design principles:

- **Laser focus**: This only shows what is required, creates a focal point based on what the user is trying to achieve, and doesn't detract from it with unnecessary clutter
- **Minimalistic**: This only incorporates required graphical features based on the usability concerns
- **Responsive**: This is a design approach that ensures that the display is clear and consistent regardless of the device it is viewed on, be it a PC or a tablet
- **Standards based**: This means that you shouldn't use any vendor-specific technologies that would preclude the viewing of the dashboard on devices such as the iPad

For More Information:
Log Stream Processing

The dashboard in this recipe will present a single dynamic graph of the log volume by minute per log file. The following screenshot illustrates the relative expanded view output:

![Relative Expanded View](image1)

The following screenshot illustrates the detail inspection support output:

![Detail Inspection Support](image2)

For More Information:

How to do it...

1. Create a new project called log-web using the standard Maven archetype command:

   mvn archetype:generate -DgroupId=storm.cookbook -DartifactId=log-web -DarchetypeArtifactId=maven-archetype-webapp

   This will generate a standard project structure and Maven POM file for you.

2. Open the pom.xml file and remove the default dependencies, replacing them with the following dependencies:

   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.8.1</version>
     <scope>test</scope>
   </dependency>
   <dependency>
     <groupId>org.hectorclient</groupId>
     <artifactId>hector-core</artifactId>
     <version>1.1-2</version>
   </dependency>
   <dependency>
     <groupId>org.slf4j</groupId>
     <artifactId>slf4j-log4j12</artifactId>
     <version>1.6.1</version>
   </dependency>
   <dependency>
     <groupId>com.sun.jersey</groupId>
     <artifactId>jersey-server</artifactId>
     <version>1.16</version>
   </dependency>
   <dependency>
     <groupId>com.sun.jersey</groupId>
     <artifactId>jersey-grizzly2</artifactId>
     <version>1.16</version>
   </dependency>
   <dependency>
     <groupId>com.sun.jersey</groupId>
     <artifactId>jersey-servlet</artifactId>
     <version>1.16</version>
   </dependency>

For More Information:
3. Then add the following build plugins to the build section of the POM:

```xml
<plugins>
  <plugin>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
  </plugin>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <executions>
      <execution>
        <goals>
          <goal>java</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3</version>
    <configuration>
      <source>1.6</source>
      <target>1.6</target>
      <optimize>true</optimize>
    </configuration>
  </plugin>
</plugins>
```

For More Information:
4. Then import the project into Eclipse using the `mvn eclipse:eclipse` command and the Eclipse project import process.

5. The excellent Twitter Bootstrap GUI library will be used in the creation of the user interface. Start by downloading this into a separate location on your drive and expanding it.

   ```
   wget http://twitter.github.com/bootstrap/assets/bootstrap.zip
   unzip bootstrap.zip
   ```

6. The bootstrap gives us a rapid start by providing many practical examples; we will simply copy one and adapt it:

   ```
   cp bootstrap/docs/examples/hero.html log-web/src/main/webapp/index.html
   cp bootstrap/docs/about log-web/src/main/webapp/about.html
   cp bootstrap/docs/assets log-web/src/main/webapp/
   cp bootstrap/docs/templates log-web/src/main/webapp/
   ```

   The Twitter Bootstrap is really quite an excellent departure point for any web-based GUI; it is highly recommended that you read the self-contained documentation in the downloaded package.

7. While there is much HTML to update, we will focus on the important elements: the central content and graph. Update the `index.html` file, replacing the existing `</div class="container">` tag and its contents with the following:

   ```
   <div class="container">
     <div class="hero-unit">
       <div id="chart">
         <svg style="height: 300px;"/>
       </div>
     </div>
     <div class="row">
       <div class="span4">
         For More Information:
       </div>
     </div>
   </div>
   ```

For More Information:
For More Information:
Chapter 2

```
.axisLabel('Time')
.showMaxMin(false);

chart.yAxis
  .axisLabel('Volume')
  .tickFormat(d3.format(',' + 2f));

d3.select('#chart svg')
  .datum(getdata())
  .transition().duration(500)
  .call(chart);

nv.utils.windowResize(chart.update);

chart.dispatch.on('stateChange', function (e) {
  nv.log('New State:', JSON.stringify(e));
});

return chart;
});

function update() {
  fetch();
  if (continueUpdates)
    setTimeout(update, 60000);
}

update();

$(document).ready(function () {
  $('#updateToggleButton').bind('click', function () {
    if (continueUpdates)
      continueUpdates = false;
    else {
      continueUpdates = true;
      update();
    }
  });
});

13. And then add the code to fetch the data from the server:

```
v alreadyFetched = {};

function getUrl(){
  var today = new Date();
today.setSeconds(0);
```

For More Information:
Log Stream Processing

```javascript
today.setMilliseconds(0);
var timestamp = today.valueOf();
var dataurl = "http://localhost:8080/services/LogCount/
    TotalsForMinute/" + timestamp + "/";
return dataurl;
}

function fetch() {
    // find the URL in the link right next to us
    var dataurl = getUrl();

    // then fetch the data with jQuery
    function onDataReceived(series) {
        // append to the existing data
        for(var i = 0; i < series.length; i++){
            if(alreadyFetched[series[i].FileName] == null){
                alreadyFetched[series[i].FileName] = {
                    FileName: series[i].FileName,
                    values: [{
                        Minute: series[i].Minute,
                        Total: series[i].Total
                    }]
                };
            } else {
                alreadyFetched[series[i].FileName].values.push({
                    Minute: series[i].Minute,
                    Total: series[i].Total
                });
                if(alreadyFetched[series[i].FileName].values.
                    length > 30){
                    alreadyFetched[series[i].FileName].values.
                        pop();
                }
            }
        }

        //update the graph
        d3.select('#chart svg')
            .datum(getdata())
            .transition().duration(500)
            .call(chart);
    }

    function onError(request, status, error){
        console.log("Received Error from AJAX: " +
            request.responseText);
    }
}

For More Information:
$.ajax({
    url: dataurl,
    type: 'GET',
    dataType: 'json',
    crossDomain: true,
    xhrFields: {
        withCredentials: true
    },
    success: onDataReceived,
    error: onError
});
}

function getdata() {
    var series = [];
    var keys = [];
    for (key in alreadyFetched) {
        keys.push(key);
    }
    for (var i = 0; i < keys.length; i++) {
        var newValues = [];
        for (var j = 0; j < alreadyFetched[keys[i]].values.length; j++) {
            newValues.push([alreadyFetched[keys[i]].values[j].Minute, alreadyFetched[keys[i]].values[j].Total]);
        }
        series.push({
            key: alreadyFetched[keys[i]].FileName,
            values: newValues
        });
    }
    return series;
}

14. This completes the client-side part of the implementation. In order to expose the data to the client layer, we need to expose services to retrieve the data.

15. Start by creating a utility class called CassandraUtils in the storm.cookbook.services.resources package and add the following content:

   public class CassandraUtils {
      public static Cluster cluster;
      public static Keyspace keyspace;
   }

For More Information:
protected static Properties properties;

public static boolean initCassandra(){
    properties = new Properties();
    try {
        properties.load(Main.class.getResourceAsStream("/cassandra.properties"));
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
    cluster = HFactory.getOrCreateCluster(properties.getProperty("cluster.name", "DefaultCluster"), properties.getProperty("cluster.hosts", "127.0.0.1:9160"));
    ConfigurableConsistencyLevel ccl = new ConfigurableConsistencyLevel();
    ccl.setDefaultReadConsistencyLevel(HConsistencyLevel.ONE);
    String keyspaceName = properties.getProperty("logging.keyspace", "Logging");
    keyspace = HFactory.createKeyspace(keyspaceName, cluster, ccl);
    return (cluster.describeKeyspace(keyspaceName) != null);
}

16. Then create the LogCount class in the same package, which essentially exposes a RESTful lookup service:

@Path("/LogCount")
public class LogCount {

    @GET
    @Path("/TotalsForMinute/{timestamp}")
    @Produces("application/json")
    public String getMinuteTotals(@PathParam("timestamp") String timestamp){
        SliceCounterQuery<String, String> query =
            HFactory.createCounterSliceQuery(CassandraUtils.keyspace,
                                               StringSerializer.get(),
                                               StringSerializer.get());

        // For More Information:
query.setColumnFamily("LogVolumeByMinute");
query.setKey(timestamp);
query.setRange("", ",", false, 100);

QueryResult<CounterSlice<String>> result = query.execute();

Iterator<HCounterColumn<String>> it =
    result.get().getColumns().iterator();
JSONArray content = new JSONArray();
while (it.hasNext()) {
    HCounterColumn<String> column = it.next();
    JSONObject fileObject = new JSONObject();
    fileObject.put("FileName", column.getName());
    fileObject.put("Total", column.getValue());
    fileObject.put("Minute", Long.parseLong(timestamp));
    content.add(fileObject);
}
return content.toJSONString();
}

17. Finally, you expose the service by creating the LogServices class:

    @ApplicationPath("/")
    public class LogServices extends Application {

        public LogServices()
        {
            CassandraUtils.initCassandra();
        }        
    }

        @Override
        public Set<Class<?>> getClasses() {
            final Set<Class<?>> classes = new HashSet<Class<?>>();
            // register root resource
            classes.add(LogCount.class);
            return classes;
        }
    }

18. Then configure the web.xml file:

    <web-app>
        <display-name>Log-Web</display-name>
        <servlet>
            <servlet-name>
                storm.cookbook.services.LogServices</servlet-name>
            <servlet-class>
                
            </servlet-class>
        </servlet>
    </web-app>

For More Information:
Log Stream Processing

com.sun.jersey.spi.container.servlet.ServletContainer
</init-param>
</servlet>
</servlet-mapping>
</web-app>

19. You can now run your project using the following command from the root of your web-log project:

mvn jetty:run

Your dashboard will then be available at localhost:8080.

How it works...

At a high level, the dashboard works by periodically querying the server for counts for a given time. It maintains an in-memory structure on the client side to hold the results of these queries and then feeds the consolidated two-dimensional array into the graph class. Take a look at the HTML; the following code defines where the graph will be displayed:

```html
<div id="chart">
  <svg style="height: 300px;">"</svg>
</div>
```

The chart is defined by the following:

```javascript
nv.addGraph(function () {
  chart = nv.models.stackedAreaChart()
    .x(function(d) { return d[0] })
    .y(function(d) { return d[1] })
    .clipEdge(true);

  chart.xAxis
    .tickFormat(function(d) { return d3.time.format('%X')(new Date(d)) })
```

For More Information:

The in-memory structure is essentially simply a two-dimensional array of values and so it is important to map these onto the x and y axes on the graph, which is done through the following:

```
chart = nv.models.stackedAreaChart()
   .x(function(d) { return d[0] })
   .y(function(d) { return d[1] })
   .clipEdge(true);
```

Data is fetched through the `fetch()` method, which issues an Ajax asynchronous request to the server. Once the response is received, it is added to the in-memory structure in the `onDataReceived(series)` method. Finally, the `getdata()` method maps the log structure into a two-dimension array to be displayed by the graph.

On the server side, the service is exposed via Jersey. It is the open source, production-quality, JSR 311 Reference Implementation for building RESTful web services. Services are defined using annotations. For this recipe, only the single service is defined by the following annotations to the `LogCount` class:

```
@Path("/LogCount")
public class LogCount {

    @GET
    @Path("/TotalsForMinute/{timestamp}")
    @Produces("application/json")
    public String getMinuteTotals(@PathParam("timestamp") String timestamp){
```
This service will then essentially be available from localhost:8080/services/LogCount/TotalForMinutes/[timestamp]. The value passed into the timestamp variable will be used in performing the lookup against Cassandra. The results of the query are then mapped onto a JSON object and returned to the caller:

```java
Iterator<HCounterColumn<String>> it = result.get().getColumns().iterator();
JSONArray content = new JSONArray();
while (it.hasNext()) {
    HCounterColumn<String> column = it.next();
    JSONObject fileObject = new JSONObject();
    fileObject.put("FileName", column.getName());
    fileObject.put("Total", column.getValue());
    fileObject.put("Minute", Long.parseLong(timestamp));
    content.add(fileObject);
} return content.toJSONString();
```

It is usually quite difficult to bring up the entire topology and set of clusters in order to simply test the web application; a convenient main class is provided in the supporting material that populates the column family with random data, allowing for easy testing of the web application in isolation.

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.