

# Moving data in and out of Hadoop

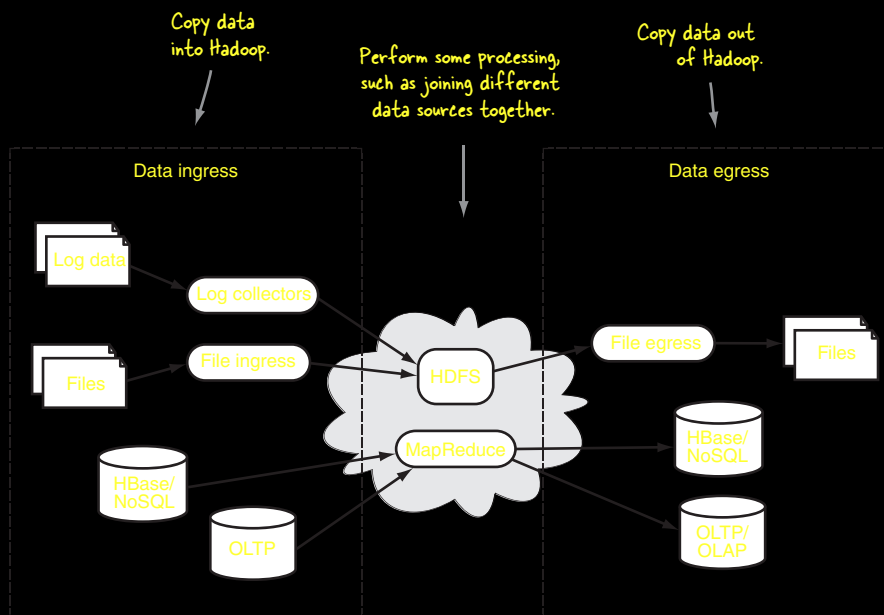
---

## ***This chapter covers***

- Understanding key design considerations for data ingress and egress tools
- Techniques for moving log files into HDFS and Hive
- Using relational databases and HBase as data sources and data sinks

Moving data in and out of Hadoop, which I'll refer to in this chapter as data *ingress* and *egress*, is the process by which data is transported from an external system into an internal system, and vice versa. Hadoop supports ingress and egress at a low level in HDFS and MapReduce. Files can be moved in and out of HDFS, and data can be pulled from external data sources and pushed to external data sinks using MapReduce. Figure 2.1 shows some of Hadoop's ingress and egress mechanisms.

The fact that your data exists in various forms and locations throughout your environments complicates the process of ingress and egress. How do you bring in data that's sitting in an OLTP (online transaction processing) database? Or ingress log data that's being produced by tens of thousands of production servers? Or work with binary data sitting behind a firewall?



**Figure 2.1** Hadoop data ingress and egress transports data to and from an external system to an internal one.

Further, how do you automate your data ingress and egress process so that your data is moved at regular intervals? Automation is a critical part of the process, along with monitoring and data integrity responsibilities to ensure correct and safe transportation of data.

In this chapter we'll survey the tools that simplify the process of ferrying data in and out of Hadoop. We'll also look at how to automate the movement of log files, ubiquitous data sources for Hadoop, but which tend to be scattered throughout your environments and therefore present a collection and aggregation challenge. In addition, we'll cover using Flume for moving log data into Hadoop, and in the process we'll evaluate two competing log collection and aggregation tools, Chukwa and Scribe.

We'll also walk through how to move relational data in and out of Hadoop. This is an emerging usage pattern where you can use Hadoop to join data sitting in your databases with data ingressed from other sources, such as log files, and subsequently push result data back out to databases. Finally, we'll cover how to use Sqoop for database ingress and egress activities, and we'll look at how to ingress and egress data in HBase.

We'll cover a lot of ground in this chapter, and it's likely that you have specific types of data you need to ingress or egress. If this is the case, feel free to jump directly to a particular section that provides the details you need. In addition, if you're looking for lower-level HDFS ingress and egress options, take a look at appendix B where I cover using tools such as WebHDFS and Hoop.

Let's start things off with a look at key ingress and egress system considerations.

## 2.1 Key elements of ingress and egress

Moving large quantities of data in and out of Hadoop has logistical challenges that include consistency guarantees and resource impacts on data sources and destinations. Before we dive into the techniques, however, we need to discuss the design elements to be aware of when working with data ingress and egress.

### IDEMPOTENCE

An idempotent operation produces the same result no matter how many times it's executed. In a relational database the inserts typically aren't idempotent, because executing them multiple times doesn't produce the same resulting database state. Alternatively, updates often are idempotent, because they'll produce the same end result.

Any time data is being written idempotence should be a consideration, and data ingress and egress in Hadoop is no different. How well do distributed log collection frameworks deal with data retransmissions? How do you ensure idempotent behavior in a MapReduce job where multiple tasks are inserting into a database in parallel? We'll examine and answer these questions in this chapter.

### AGGREGATION

The data aggregation process combines multiple data elements. In the context of data ingress this can be useful because moving large quantities of small files into HDFS potentially translates into NameNode memory woes, as well as slow MapReduce execution times. Having the ability to aggregate files or data together mitigates this problem, and is a feature to consider.

### DATA FORMAT TRANSFORMATION

The data format transformation process converts one data format into another. Often your source data isn't in a format that's ideal for processing in tools such as MapReduce. If your source data is multiline XML or JSON form, for example, you may want to consider a preprocessing step. This would convert the data into a form that can be split, such as a JSON or an XML element per line, or convert it into a format such as Avro. Chapter 3 contains more details on these data formats.

### RECOVERABILITY

Recoverability allows an ingress or egress tool to retry in the event of a failed operation. Because it's unlikely that any data source, sink, or Hadoop itself can be 100 percent available, it's important that an ingress or egress action be retried in the event of failure.

### CORRECTNESS

In the context of data transportation, checking for correctness is how you verify that no data corruption occurred as the data was in transit. When you work with heterogeneous systems such as Hadoop data ingress and egress tools, the fact that data is being transported across different hosts, networks, and protocols only increases the potential for problems during data transfer. Common methods for checking correctness of raw data such as storage devices include Cyclic Redundancy Checks (CRC), which are what HDFS uses internally to maintain block-level integrity.

**RESOURCE CONSUMPTION AND PERFORMANCE**

Resource consumption and performance are measures of system resource utilization and system efficiency, respectively. Ingress and egress tools don't typically incur significant load (resource consumption) on a system, unless you have appreciable data volumes. For performance, the questions to ask include whether the tool performs ingress and egress activities in parallel, and if so, what mechanisms it provides to tune the amount of parallelism. For example, if your data source is a production database, don't use a large number of concurrent map tasks to import data.

**MONITORING**

Monitoring ensures that functions are performing as expected in automated systems. For data ingress and egress, monitoring breaks down into two elements: ensuring that the process(es) involved in ingress and egress are alive, and validating that source and destination data are being produced as expected.

On to the techniques. Let's start with how you can leverage Hadoop's built-in ingress and egress mechanisms.

## **2.2 Moving data into Hadoop**

The first step in working with data in Hadoop is to make it available to Hadoop. As I mentioned earlier in this chapter, there are two primary methods that can be used for moving data into Hadoop: writing external data at the HDFS level (a data push), or reading external data at the MapReduce level (more like a pull). Reading data in MapReduce has advantages in the ease with which the operation can be parallelized and fault tolerant. Not all data is accessible from MapReduce, however, such as in the case of log files, which is where other systems need to be relied upon for transportation, including HDFS for the final data hop.

In this section we'll look at methods to move source data into Hadoop, which I'll refer to as *data ingress*. I'll use the data ingress design considerations in the previous section as the criteria to examine and understand the different tools as I go through the techniques.

We'll look at Hadoop data ingress across a spectrum of data sources, starting with log files, then semistructured or binary files, then databases, and finally HBase. We'll start by looking at data ingress of log files.

**LOW-LEVEL HADOOP INGRESS MECHANISMS**

This section will focus on high-level data ingress tools that provide easy and automated mechanisms to get data into Hadoop. All these tools use one of a finite set of low-level mechanisms, however, which Hadoop provides to get data in and out. These mechanisms include Hadoop's Java HDFS API, WebHDFS, the new Hadoop 0.23 REST API, and MapReduce. An extensive evaluation of these mechanisms and tools is outside the scope of this chapter, but I provide them for reference in appendix B.

### 2.2.1 Pushing log files into Hadoop

Log data has long been prevalent across all applications, but with Hadoop came the ability to process the large volumes of log data produced by production systems. Various systems produce log data, from network devices and operating systems to web servers and applications. These log files all offer the potential for valuable insights into how systems and applications operate as well as how they're used. What unifies log files is that they tend to be in text form and line-oriented, making them easy to process.

In this section we'll look at tools that can help transport log data from source to HDFS. We'll also perform a deep dive into one of these tools and look at how to transport system log files into HDFS and Hive. I'll provide what you need to know to deploy, configure, and run an automated log collection and distribution infrastructure, and kick-start your own log data-mining activities.

#### COMPARING FLUME, CHUKWA, AND SCRIBE

Flume, Chukwa, and Scribe are log collecting and distribution frameworks that have the capability to use HDFS as a data sink for that log data. It can be challenging to differentiate between them because they share the same features.

#### FLUME

Apache Flume is a distributed system for collecting streaming data. It's an Apache project in incubator status, originally developed by Cloudera. It offers various levels of reliability and transport delivery guarantees that can be tuned to your needs. It's highly customizable and supports a plugin architecture where you can add custom data sources and data sinks.

Figure 2.2 shows Flume's architecture.

#### CHUKWA

Chukwa is an Apache subproject of Hadoop that also offers a large-scale mechanism to collect and store data in HDFS. And it's also in incubator status. Chukwa's reliability

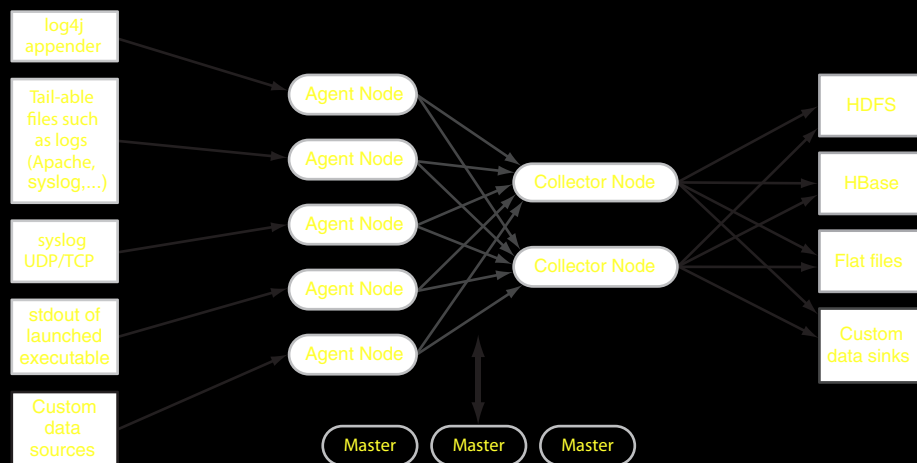
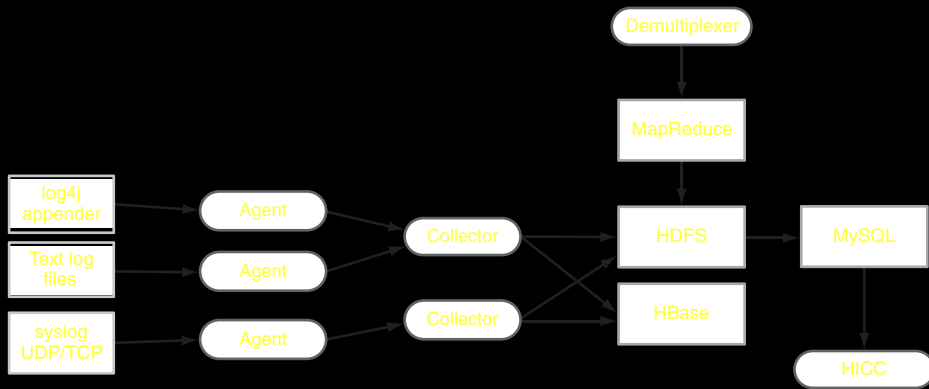


Figure 2.2 Flume architecture for collecting streaming data



**Figure 2.3** Chukwa architecture for collecting and storing data in HDFS

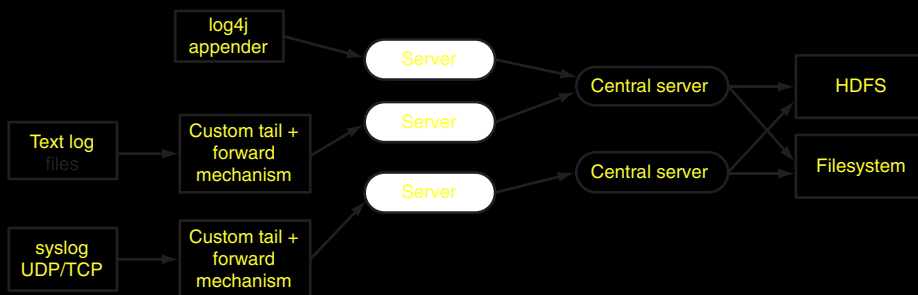
model supports two levels: end-to-end reliability, and fast-path delivery, which minimizes latencies. After writing data into HDFS Chukwa runs a MapReduce job to demultiplex the data into separate streams. Chukwa also offers a tool called Hadoop Infrastructure Care Center (HICC), which is a web interface for visualizing system performance.

Figure 2.3 shows Chukwa’s architecture.

#### SCRIBE

Scribe is a rudimentary streaming log distribution service, developed and used heavily by Facebook. A scribe server that collects logs runs on every node and forwards them to a central Scribe server. Scribe supports multiple data sinks, including HDFS, regular filesystems, and NFS. Scribe’s reliability comes from a file-based mechanism where the server persists to a local disk in the event it can’t reach the downstream server.

Unlike Flume or Chukwa, Scribe doesn’t include any convenience mechanisms to pull log data. Instead the onus is on the user to stream the source data to the Scribe server running on the local system. For example, if you want to push your Apache log files, you would need to write a daemon that tails and forwards the log data to the Scribe server. Figure 2.4 shows Scribe’s architecture. The Scribe project is hosted on GitHub at <https://github.com/facebook/scribe>.



**Figure 2.4** Scribe architecture also pushes log data into HDFS.

**Table 2.1 Feature comparison of log collecting projects**

Project	Centralized configuration	Reliability	Failover	Level of documentation	Commercial support	Popularity (number of mailing list messages from 08/11/2011)
Flume	Yes	Yes <ul style="list-style-type: none"> <li>• Best effort</li> <li>• Disk failover</li> <li>• End-to-end</li> </ul>	Yes <ul style="list-style-type: none"> <li>• None</li> <li>• Manually configurable</li> <li>• Automated configuration</li> </ul>	High	Yes (Cloudera)	High (348)
Chukwa	No	Yes <ul style="list-style-type: none"> <li>• Fast</li> <li>• End-to-end</li> </ul>	Yes <ul style="list-style-type: none"> <li>• None</li> <li>• Manually configurable</li> <li>• Automated</li> </ul>	Low	No	Medium (85)
Scribe	No	Yes <ul style="list-style-type: none"> <li>• Disk-based (not end-to-end)</li> </ul>	No	Low	No	Medium (46)

Each of these three tools can fulfill the criteria of providing mechanisms to push log data into HDFS. Table 2.1 compares the various tools based on features such as reliability and configuration.

From a high-level perspective there's not much feature difference between these tools, other than that Scribe doesn't offer any end-to-end delivery guarantees. It's also clear that the main downside for Chukwa and Scribe is their limited user documentation. Their mailing list activities were also moderate.

I had to pick one of these three tools for this chapter, so I picked Flume. My reasons for selecting Flume include its centralized configuration, its flexible reliability and failover modes, and also the popularity of its mailing list.

Let's look at how you can deploy and set up Flume to collect logs, using a problem/solution scenario. I'll continue to introduce techniques in this manner throughout the rest of this chapter.

## TECHNIQUE 1 Pushing system log messages into HDFS with Flume

You have a bunch of log files being produced by multiple applications and systems across multiple servers. There's no doubt there's valuable information to be mined out of these logs, but your first challenge is a logistical one of moving these logs into your Hadoop cluster so that you can perform some analysis.

### **Problem**

You want to push all of your production server system log files into HDFS.

### **Solution**

For this technique you'll use Flume, a data collection system, to push a Linux log file into HDFS. We will also cover configurations required to run Flume in a distributed environment, as well as an examination of Flume's reliability modes.

**Discussion**

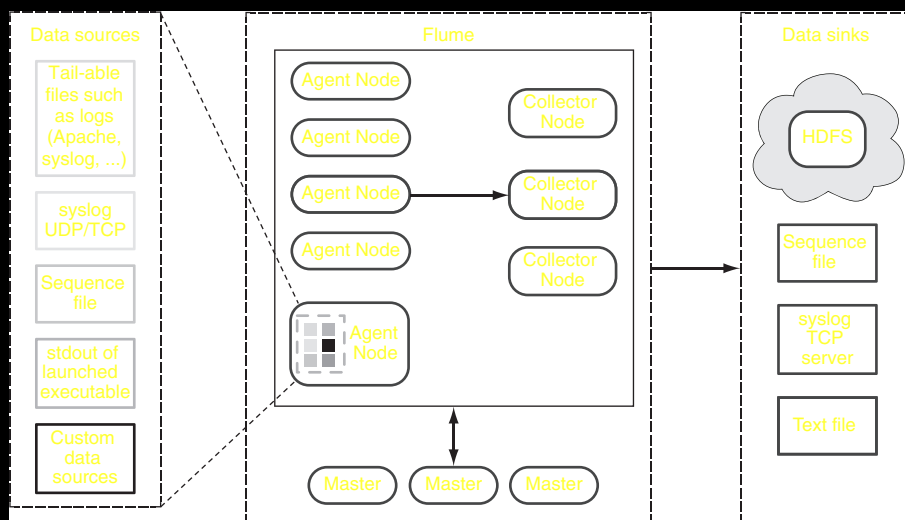
Figure 2.5 shows a full-fledged Flume deployment, and its four primary components:

- *Nodes*—Flume data paths that ferry data from a data source to a data sink. Agents and Collectors are simply Flume Nodes that are deployed in a way to efficiently and reliably work with a large number of data sources.
- *Agents*—Collect streaming data from the local host and forward it to the Collectors.
- *Collectors*—Aggregate data sent from the Agents and write that data into HDFS.
- *Masters*—Perform configuration management tasks and also help with reliable data flow.

This figure also shows data sources and data sinks. Data sources are streaming data origins whose data you want to transport to a different destination. Examples include application logs and Linux system logs, as well as nontext data that can be supported with custom data sources. Data sinks are the destination of that data, which can be HDFS, flat files, and any data target that can be supported with custom data sinks.

You'll run Flume in pseudo-distributed mode, which means you'll run the Flume Collector, Agent, and Master daemons on your single host. The first step is to install Flume, the Flume Master, and the Flume Node packages from the CDH3 Installation Guide.<sup>1</sup> After you've installed these packages, start up the Master and Agent daemons:

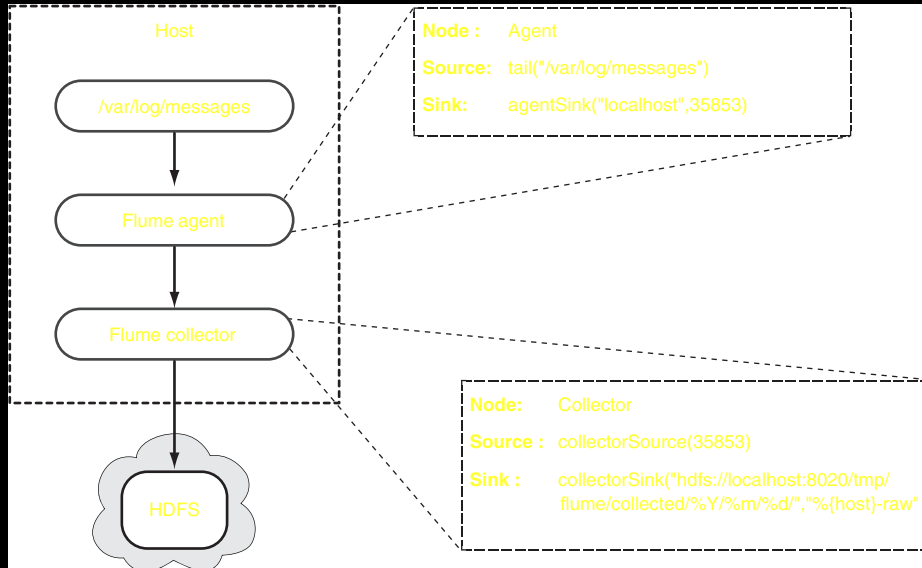
```
$ /etc/init.d/flume-master start
$ /etc/init.d/flume-node start
```



**Figure 2.5** Example of Flume deployment for collecting streaming data

<sup>1</sup> Appendix A contains installation instructions and additional resources for working with Flume.





**Figure 2.6** Data flow from `/var/log/messages` into HDFS

The data source in this exercise is the file `/var/log/messages`, the central file in Linux for system messages, and your ultimate data destination is HDFS. Figure 2.6 shows this data flow, and the Agent and Collector configuration settings you'll use to make it work.

By default, Flume will write data in Avro JSON format, which we'll discuss shortly. You'll want to preserve the original format of your syslog file, so you'll need to create and edit `flume-site.xml` and indicate the raw output format. The file should look like this:

```
$ cat /etc/flume/conf/flume-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>flume.collector.output.format</name>
    <value>raw</value>
  </property>
</configuration>
```

If you've set up your cluster with LZO compression, you'll need to create a `flume-env.sh` file and set the directory that contains the native compression codecs:

```
$ cp /usr/lib/flume/bin/flume-env.sh.template \
  /usr/lib/flume/bin/flume-env.sh
$ vi /usr/lib/flume/bin/flume-env.sh
# add the following line for 64-bit environments
export JAVA_LIBRARY_PATH=/usr/lib/hadoop/lib/native/Linux-amd64-64
# or the following line for 32-bit environments
export JAVA_LIBRARY_PATH=/usr/lib/hadoop/lib/native/Linux-i386-32
```

**Table 2.2** Flume UI endpoints

Daemon	URL
Flume Master	http://localhost:35871
Flume Node (Agent)	http://localhost:35862
Flume Node (Collector)	http://localhost:35863

You'll also need to copy the LZO JAR into Flume's lib directory:

```
$ cp /usr/lib/hadoop/lib/hadoop-lzo-0.4.1* /usr/lib/flume/lib/
```

Flume runs as the `flume` user, so you need to make sure that it has permissions to read any data source files (such as files under `/var/log`, for example).

Previously, when you launched the Flume Master and Node you were a Node short. Let's launch another Node, which you'll name `collector`, to perform the Collector duties:

```
$ flume node_nowatch -n collector
```

Each of the Flume daemons have embedded web user interfaces. If you've followed the previous instructions, table 2.2 shows the locations where they'll be available.

The advantage of using a Flume Master is that you can make configuration changes in a central location, and they'll be pulled by the Flume Nodes. There are two ways you can make configuration changes in the Flume Master: using the UI or the Flume shell. I'll show the configuration in the UI.

You'll need to configure the Agent and Collector Nodes according to the setup illustrated in figure 2.6. You'll connect to the Flume Master UI, and select the config menu from the top, as highlighted in figure 2.7. The drop-down box contains all of the Nodes, and you'll select the Agent node. The Agent node has the same name as

Select this menu item  
to get to this screen.

This drop-down  
contains a list of all the  
Agent and Collector  
Nodes connected to the  
master.

35853 is the port that  
the Collector listens on.

**Figure 2.7** Flume Agent Node configuration to follow

master | config | raw commands | static config | env | extn

## Flume Master: Configure Nodes

### Configure a single node

Configure node:	Choose from list <input type="button" value="v"/>
or specify another node:	<input type="text"/>
Source:	collectorSource(35853)
Sink:	collectorSink("hdfs://localhost:8020/tmp/flume/collected/%Y%m/%d/", "%{host}-raw")
<input type="button" value="Submit Query"/>	

Figure 2.8 Flume Collector Node configuration to follow

the hostname that you're running on—in my case *cdh*. You should see one other Node in the drop-down called *collector*, which you'll configure next. For the Agent Node, you'll specify that the data source is a tail of the syslog file and the data sink is the port that your Collector will run on.

Now select the Collector from the drop-down, and in a similar fashion, set the data source, which is the local port that you'll listen on, and the data sink, which is the final destination in HDFS for your log data. Figure 2.8 shows the configuration.

The main Flume Master screen displays all of the Nodes and their configured data sources and sinks, as shown in figure 2.9.

All the actions you just performed can be executed in the Flume shell. Here's an example of how you can view the same information that you just saw in the UI. To better identify the text you entered, the shell prompts are surrounded by square brackets. Your first command is `connect ...` to connect to the Flume Master on port 35873 (the

# Flume Master

Version: 0.9.4-cdh3u2, runknown  
Compiled: 20111013-2105 by jenkins

ServerID: 0

Servers localhost

## Node status

logical node	physical node	host name	status	version	last seen delta (s)	last seen
cdh	cdh	cdh	ACTIVE	Sun Nov 20 15:15:50 EST 2011	3	Sun Nov 20 15:25:06 EST 2011
collector	collector	cdh	ACTIVE	Sun Nov 20 15:17:22 EST 2011	0	Sun Nov 20 15:25:08 EST 2011

## Node configuration

Node	Version	Flow ID	Source	Sink	Translated Version	Translated Source	Translated Sink
cdh	Sun Nov 20 15:15:50 EST 2011	default-flow	tail("/var/log/messages")	agentSink("localhost",35853)	Sun Nov 20 15:15:50 EST 2011	tail("/var/log/messages")	agentSink("localhost", 35853 )
collector	Sun Nov 20 15:17:22 EST 2011	default-flow	collectorSource(35853)	collectorSink("hdfs://localhost:8020/tmp/flume/collected/%Y/%m/%d/", "%{host}-raw")	Sun Nov 20 15:17:22 EST 2011	collectorSource(35853 )	collectorSink("hdfs://localhost:8020/tmp/flume/collected/%Y/%m/%d/", "%{host}-raw" )

## Physical/Logical Node mapping

physical node	logical node
collector	collector
cdh	cdh

Figure 2.9 The main screen in the Flume Master UI after the configuration of your Nodes

default Flume Master port), and your second command is `getconfigs`, which dumps the current configuration for the Nodes:

```
$ flume shell
[flume (disconnected)] connect localhost:35873
[flume localhost:35873:45678] getconfigs
NODE          FLOW          SOURCE          SINK
collector default-flow collectorSource(35853) collectorSink(...)
cdh           default-flow  tail("/var/log/messages") agentSink(...)
```

After you make the configuration changes in the UI, they'll be picked up by the Flume Nodes after a few seconds. The Agent will then start to pipe the syslog file, and feed the output to the Collector, which in turn will periodically write output into HDFS. Let's examine HDFS to check on the progress:

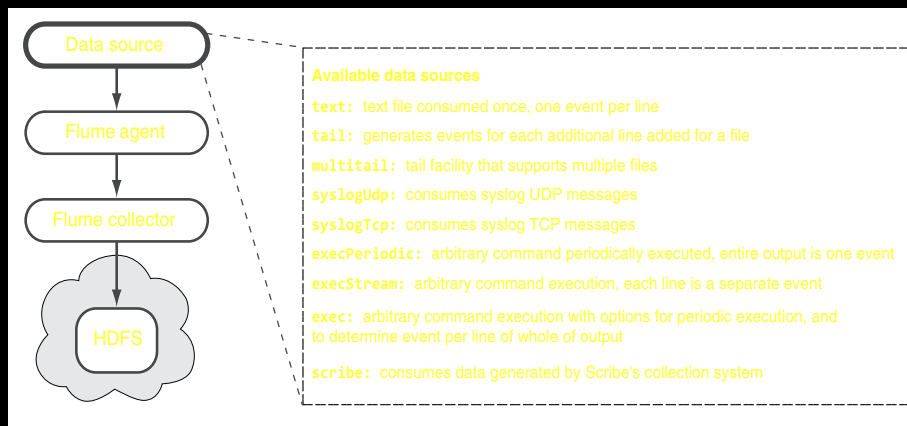
```
$ hadoop fs -lsr /tmp/flume
/tmp/flume/collected
/tmp/flume/collected/2011
/tmp/flume/collected/2011/11
/tmp/flume/collected/2011/11/20
/tmp/flume/collected/2011/11/20/cdh-raw201111120-133115126-...
/tmp/flume/collected/2011/11/20/cdh-raw201111120-134449503-...
/tmp/flume/collected/2011/11/20/cdh-raw201111120-135653300-...
...
```

We went through a whole lot of work without discussing some of the key concepts involved in setting up the data pipeline. Let's go back now and inspect the previous work in detail. The first thing we'll look at is the definition of the data source.

### FLUME DATA SOURCES

A data source is required for both the Agent and Collector Nodes; it determines where they collect their data. The Agent Node's data source is your application or system data that you want to transfer to HDFS, and the Collector Node's data source is the Agent's data sink. Figure 2.10 shows a subset of data sources supported by the Agent Node.

As you can see, Flume supports a number of Agent sources, a complete list of which can be viewed on the Cloudera website [http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#\\_flume\\_source\\_catalog](http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#_flume_source_catalog). Figure 2.7 uses the `tail` data



**Figure 2.10** Flume Agent Node data sources supported by the Agent Node

**Table 2.3** Flume Agent data sink reliability modes

Acronym	Description
E2E (end to end)	Guarantees that once an event enters Flume, the event will make its way to the end data sink.
DFO (disk failover)	Events are persisted on local disk in the event that a failure occurs when attempting to transmit the event to the downstream Node. Acknowledgement messages sent from downstream Nodes result in the persisted data being removed.
BE (best effort)	Events are dropped in the event of failed communication with a downstream Node.

source as an example, which works well for text files that are appended and rotated, specifying the file whose output you want to capture: `tail("/var/log/messages")`.

You can configure the `tail` data source to emit the complete file as events, or just start from the current end of the file. The default is to read the whole file. The `multi-tail` data source accepts a list of filenames, and the `tailDir` takes a directory name with a regular expression to filter files that should be tailed.

Flume also supports TCP/UDP data sources that can receive logs from syslog. Their data source names are `syslogUdp` for `syslogd`, and `syslogTcp` for `syslog-ng`.

In addition, Flume can periodically execute a command and capture its output as an event for processing, via the `execPeriodic` data source. And the `exec` data source gives you more control, allowing you to specify if each line of the processes output should be considered a separate message, or if the whole output should be considered the message. It also can be run periodically.

Flume supports a variety of other data sources out of the box as well, and it can be extended with a custom data source, as documented at [http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#\\_arbitrary\\_data\\_flows\\_and\\_custom\\_architectures](http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#_arbitrary_data_flows_and_custom_architectures).

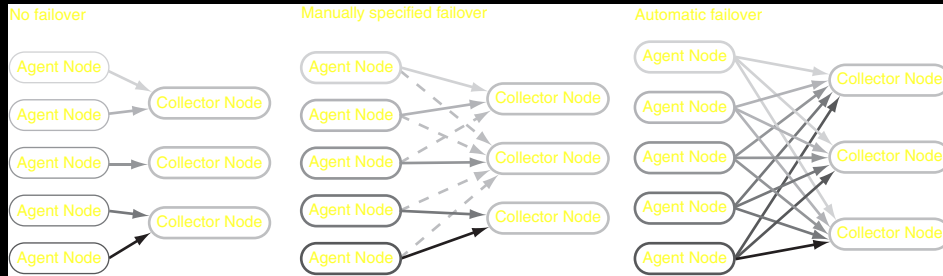
### AGENT SINKS

Agent sinks are destinations for an Agent's data source. Flume offers three different levels of reliability guarantees, which are summarized in table 2.3.

Flume also has three levels of availability, as described in table 2.4. Figure 2.11 shows how the Flume failover options work. These reliability and availability modes are combined to form nine separate Agent sink options, as shown in figure 2.12.

**Table 2.4** Flume failover options

Failover mode	Description
None	Configure each Agent to write to a single Collector. If the Collector goes down, the Agent waits until it comes back up again.
Manually specified failover	Configure each Agent with one or more Collectors in addition to the primary. If the primary Collector fails, the events will be routed to backup Collectors.
Automatic failover	In this mode the Flume Master will allocate failover Collectors for Agents whose Collectors fail. The advantage is that Flume balances the Agent/Collector event links to ensure that individual Collectors aren't overwhelmed. This rebalancing also occurs in normal situations where Collectors are added and removed.



**Figure 2.11** Flume failover architecture shows the three levels available.

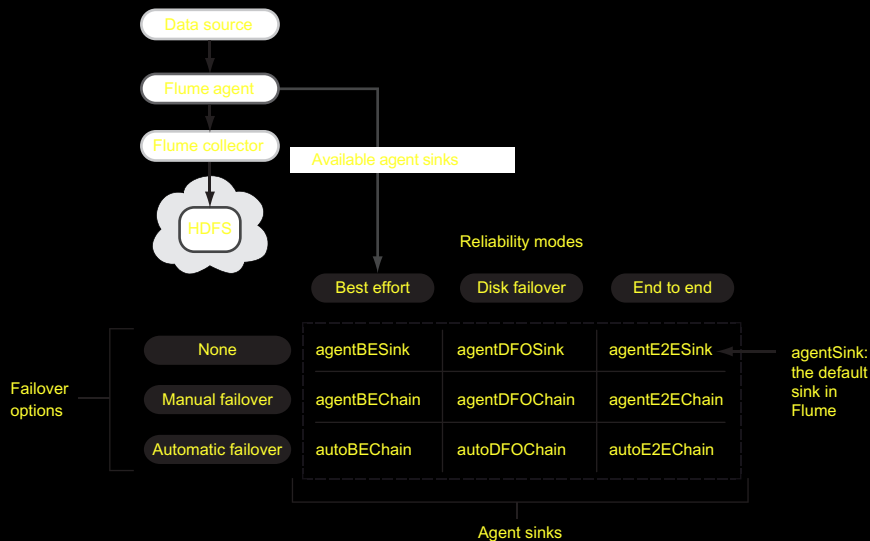
All of these Agent sinks take as arguments the Collector Node host, and the port that it's listening on. In the example in figure 2.7 I utilized the default `agentSink` option, which guarantees end-to-end delivery, but has no failover support. The Collector is running on the same host, on port 35853:

```
agentSink("localhost",35853)
```

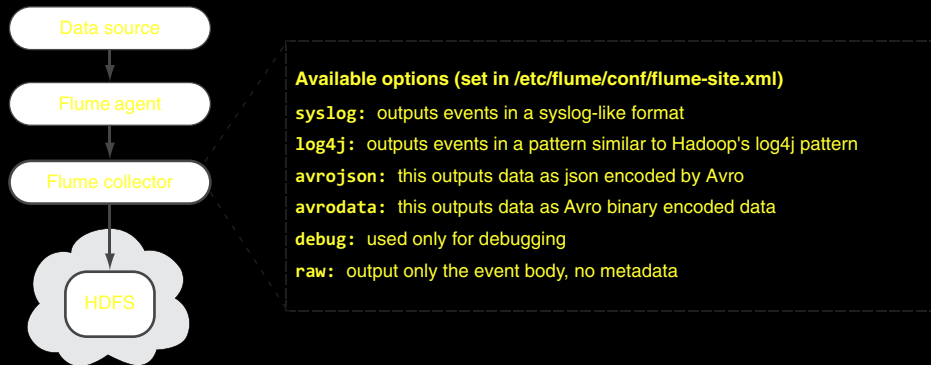
### FLUME COLLECTOR DATA SINK

Flume contains a single Collector data sink, called `collectorSink`, which you'll configure to write to a directory on a local disk or HDFS. The sink takes two parameters: the directory, and the filename prefix for files written in that directory. Both of these arguments support the Flume functionality called *output bucketing*, which permits some macro substitutions. Let's review how to configure the Collector sink:

```
collectorSink("hdfs://localhost:8020/tmp/flume/collected/%Y/%m/%d/", "%{host}-raw")
```



**Figure 2.12** Flume Agent sinks that are available



**Figure 2.13** Flume Collector data sink supports a variety of output formats.

The `%Y`, `%m`, and `%d` are date escape sequences that are substituted with the date at which the event was received. The `%{host}` is substituted with the agent host that generated the event. A full list of the escape sequences is available at [http://archive.cloudera.com/cdh/3/flume-0.9.1+1/UserGuide.html#\\_output\\_bucketing](http://archive.cloudera.com/cdh/3/flume-0.9.1+1/UserGuide.html#_output_bucketing).

The Collection data sink supports a variety of output formats for events, some of which are shown in figure 2.13.

Earlier in this chapter you created a `flume-site.xml` file and specified `raw` as your output format. By default Flume chooses `avrojson`, an example of which is shown in the following code. The body field contains the raw contents of a single line from the syslog:

```
{
  "body": "Nov 20 13:25:40 cdh aholmes: Flume test",
  "timestamp": 1321813541326,
  "pri": "INFO",
  "nanos": 2748053914369042,
  "host": "cdh",
  "fields": {
    "AckTag": "20111120-132531557-0500.2748044145743042.00000145",
    "AckType": "msg",
    "AckChecksum": "\u0000\u0000\u0000\u0000\u0004\u0002?g",
    "tailSrcFile": "messages",
    "rolltag": "20111120-132532809-0500.2748045397574042.00000021"
  }
}
```

This approach uses Flume to show how to capture syslog appends and write them into HDFS. You can use this same approach for a variety of line-based text files.

### Summary

We've used Flume on a single machine, using the default configuration settings, which assumes everything runs on the local host and on standard ports. In a fully distributed setup the Node hosts would need to specify where the Flume Master is located in `flume-site.xml`, as the next example demonstrates. Consult the user guide for more details at <http://goo.gl/8YNsU>.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
  <property>
    <name>flume.master.servers</name>
    <value>flume-master1,flume-master2,flume-master3</value>
  </property>
</configuration>
```

How do you determine the number of Flume masters that you should run? Let's say you want to be able to support the failure of two master Flume nodes. To figure out the number of Flume masters you should run, take the number of Flume master nodes that could fail, double it, and add 1. Flume uses an embedded ZooKeeper in each of the Master daemons, but you can configure it to use an external ZooKeeper if one already exists in your environment.

If you're capturing Apache logs, you can configure the web server to launch a Flume ad hoc Node and pipe the log output directly to Flume. If that Node is talking to a remote Collector, then, unfortunately, the client Node can't be configured for automatic failover or end-to-end reliability, because the Node can't be managed by the Flume Master. The workaround to this is to have the ad hoc Node forward to a local Flume Agent Node, which can have these reliability and failover properties.

Flume also includes a log4j appender, and you can find more details on this at [http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#\\_logging\\_via\\_log4j\\_directly](http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html#_logging_via_log4j_directly).

Using Flume for log distribution has many advantages over its peers, primarily because of its high level of documentation, its ease of use, and its customizable reliability modes.

We've looked at an automated way to shuttle log data into HDFS. But now imagine that the data you want to move into Hadoop isn't log data, but instead data that these tools have a harder time working with, such as semistructured or binary files.

### **2.2.2 Pushing and pulling semistructured and binary files**

You've learned how to use log collecting tools like Flume to automate moving data into HDFS. But these tools don't support working with semistructured or binary data out of the box. This section looks at techniques to help you automate moving such files into HDFS.

Production networks typically have network silos where your Hadoop clusters are segmented away from other production applications. In such cases it's possible that your Hadoop cluster isn't able to pull data from other data sources, leaving you only the option to push data into Hadoop.

You need a mechanism to automate the process of copying files of any format into HDFS, similar to the Linux tool `rsync`. The mechanism should be able to compress files written in HDFS and offer a way to dynamically determine the HDFS destination for data partitioning purposes.



## TECHNIQUE 2 An automated mechanism to copy files into HDFS

Existing file transportation mechanisms such as Flume, Scribe, and Chukwa are geared towards supporting log files. What if you have different file formats for your files, such as semistructured or binary? If the files were siloed in a way that the Hadoop slave nodes couldn't directly access, then you couldn't use Oozie to help with file ingress either.

### Problem

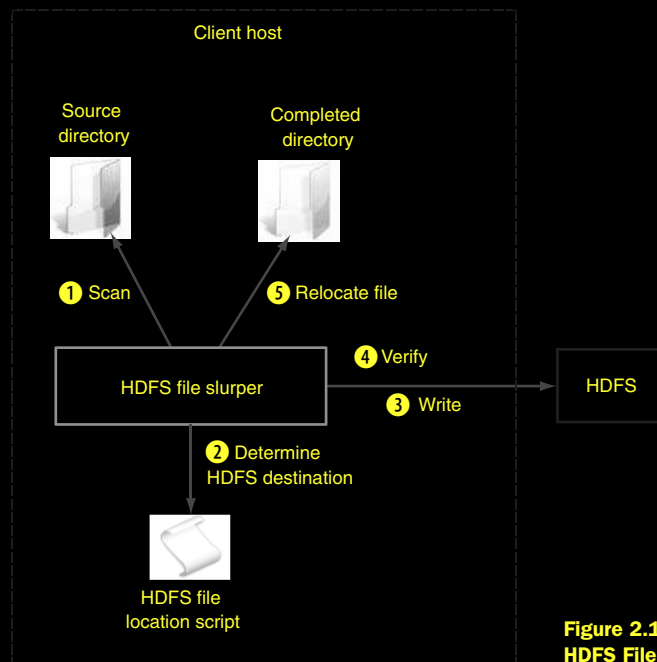
You need to automate the process by which files on remote servers are copied into HDFS.

### Solution

The HDFS File Slurper open source project can copy files of any format in and out of HDFS. This technique covers how it can be configured and used to copy data into HDFS.

### Discussion

You can use the HDFS File Slurper project that I wrote<sup>2</sup> to assist with your automation. The HDFS File Slurper is a simple utility that supports copying files from a local directory into HDFS and vice versa. Figure 2.14 provides a high-level overview of the Slurper (my nickname for the project), with an example of how you can use it to copy files. The Slurper reads any files that exist in a source directory and optionally consults with a script to determine the file placement in the destination directory. It then writes the file to the destination, after which there's an optional verification step. Finally, the



**Figure 2.14** The five-step process of the HDFS File Slurper data flow for copying files

<sup>2</sup> See <https://github.com/alexholmes/hdfs-file-slurper>.

Slurper moves the source file to a completed folder upon successful completion of all of the previous steps.

With this technique there are a few challenges you need to make sure you address:

- How do you effectively partition your writes to HDFS so that you don't lump everything into a single directory?
- How do you determine that your data is ready in HDFS for processing in order to avoid reading files that are mid-copy?
- How do you automate regular execution of your utility?

The first step is to download and build the code. The following assumes that you have git, Java, and version 3.0 or newer of Maven installed locally:

```
$ git clone git://github.com/alexholmes/hdfs-file-slurper.git
$ cd hdfs-file-slurper/
$ mvn package
```

Next you'll need to untar the tarball that the build created under `/usr/local`:

```
$ sudo tar -xzf target/hdfs-slurper-<version>-package.tar.gz \
-C /usr/local/

$ sudo ln -s /usr/local/hdfs-slurper-<version> /usr/local/hdfs-slurper
```

### CONFIGURATION

Before you can run the code you'll need to edit `/usr/local/hdfs-slurper/conf/slurper-env.sh` and set the Java home and Hadoop home directories.

The Slurper comes bundled with a `/usr/local/hdfs-slurper/conf/slurper.conf`, which contains details on the source and destination directory, along with other options. The file contains the following default settings, which you can change:

A name for the data being transferred. This is used for the log filename when launched via the Linux init daemon management system, which we'll cover shortly.

`DATASOURCE_NAME = test`

The source directory. Any files that are moved into here are automatically copied to the destination directory (with an intermediary hop to the staging directory).

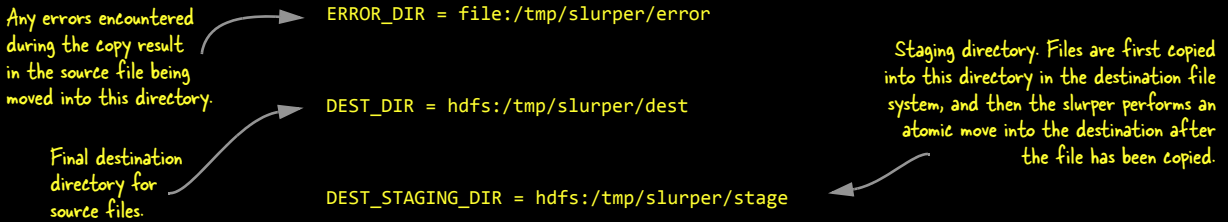
`SRC_DIR = file:/tmp/slurper/in`

The work directory. Files from the source directory are moved into this directory before the copy to the destination starts.

`WORK_DIR = file:/tmp/slurper/work`

After the copy has completed, the file is moved from the work directory into the complete directory. Alternatively the `REMOVE_AFTER_COPY` setting can be used to delete the source file, in which case the `COMPLETE_DIR` setting shouldn't be supplied.

`COMPLETE_DIR = file:/tmp/slurper/complete`



You'll notice that all of the directory names are HDFS URIs. HDFS distinguishes between different filesystems in this way. The `file:/` URI denotes a path on the local filesystem, and the `hdfs:/` URI denotes a path in HDFS. In fact, the Slurper supports any Hadoop filesystem, as long as you configure Hadoop to use it.

### RUNNING

Let's use the default settings, create a local directory called `/tmp/slurper/in`, write an empty file into it, and run the utility. If you're running your environment on a Hadoop distribution other than CDH, the `HADOOP_HOME` environment variable needs to be set with the location of your Hadoop installation:

```

$ mkdir -p /tmp/slurper/in
$ touch /tmp/slurper/in/test-file.txt

$ cd /usr/local/hdfs-slurper/
$ bin/slurper.sh --config-file conf/slurper.conf

Copying source file 'file:/tmp/slurper/work/test-file.txt'
to staging destination 'hdfs:/tmp/slurper/stage/1354823335'

Moving staging file 'hdfs:/tmp/slurper/stage/1354823335'
to destination 'hdfs:/tmp/slurper/dest/test-file.txt'

File copy successful, moving source
file:/tmp/slurper/work/test-file.txt to completed file
file:/tmp/slurper/complete/test-file.txt

$ hadoop fs -ls /tmp/slurper/dest
/tmp/slurper/dest/test-file.txt

```

A key feature in the Slurper's design is that it doesn't work with partially written files. Files must be atomically moved into the source directory (file moves in both the Linux<sup>3</sup> and HDFS filesystems are atomic). Alternatively, you can write to a filename that starts with a period (`.`), which is ignored by the Slurper, and after the file write completes, you'd rename the file to a name without the period prefix.

Another key consideration with the Slurper is the assurance that files being copied are globally unique. If they aren't, the Slurper will overwrite that file in HDFS, which is likely an undesirable outcome.

<sup>3</sup> Moving files is atomic only if both the source and destination are on the same partition. In other words, moving a file from a NFS mount to a local disk results in a copy, which isn't atomic.

**DYNAMIC DESTINATION PATHS**

The previous approach works well if you're working with a small number of files that you're moving into HDFS on a daily basis. But if you're dealing with a large volume of files you want to think about partitioning them into separate directories. This has the benefit of giving you more finely grained control over the input data for your MapReduce jobs, as well as helping with the overall organization of your data in the filesystem (you wouldn't want all the files on your computer to reside in a single flat directory).

How can you have more dynamic control over the destination directory and the filename that the Slurper uses? The Slurper configuration file (`slurper.conf`) has a `SCRIPT` option (which is mutually exclusive of the `DEST_DIR` option), where you can specify a script that can provide that dynamic mapping of the source files to destination files.

Let's assume that the files you're working with contain a date in the filename, and you've decided that you want to organize your data in HDFS by date. Let's write a script that can perform this mapping activity. The following example is a Python script that does this:

```
#!/usr/bin/python

import sys, os, re

# read the local file from standard input
input_file=sys.stdin.readline()

# extract the filename from the file
filename = os.path.basename(input_file)

# extract the date from the filename
match=re.search(r'([0-9]{4})([0-9]{2})([0-9]{2})', filename)

year=match.group(1)
mon=match.group(2)
day=match.group(3)

# construct our destination HDFS file
hdfs_dest="hdfs:/data/%s/%s/%s/%s" % (year, mon, day, filename)

# write it to standard output
print hdfs_dest,
```

Now you can update `/usr/local/hdfs-slurper/conf/slurper.conf`, set `SCRIPT`, and comment out `DEST_DIR`, which results in the following changes to the file:

```
# DEST_DIR = hdfs:/tmp/slurper/dest

SCRIPT = /usr/local/hdfs-slurper/bin/sample-python.py
```

Run the Slurper again and see what happens:

```
$ touch /tmp/slurper/in/apache-20110202.log

$ bin/slurper.sh --config-file conf/slurper.conf

Copying source file 'file:/tmp/slurper/work/apache-2011-02-02.log' to
staging destination 'hdfs:/slurper/staging/1787301476'

Moving staging file 'hdfs:/slurper/staging/1787301476' to destination
'hdfs:/slurper/in/2011-02-02/apache-2011-02-02.log'
```

### COMPRESSION AND VERIFICATION

What if you want to compress the output file in HDFS, and also verify that the copy is correct? You'd need to use the `COMPRESSION_CODEC` option, whose value, `CompressionCodec` classname, also needs to be used. The Slurper also supports verification, which rereads the destination file after the copy has completed, and ensures that the checksum of the destination file matches the source file. If you want to configure the Slurper to use Snappy for compression and verify the copy, you'd update the `slurper.conf` file and add the following lines:

```
COMPRESSION_CODEC = org.apache.hadoop.io.compress.SnappyCodec

VERIFY = true
```

Let's run the Slurper again:

```
$ touch /tmp/slurper/in/apache-20110202.log

$ bin/slurper.sh --config-file conf/slurper.conf

Verifying files
CRC's match (0)
Moving staging file 'hdfs:/tmp/slurper/stage/535232571'
to destination 'hdfs:/data/2011/02/02/apache-20110202.log.snappy'
```

### CONTINUOUS OPERATION

Now that you have the basic mechanics in place, your final step is to run your tool as a daemon, so that it continuously looks for files to transfer. To do this, you'll use a script called `bin/slurper-inittab.sh`, which is designed to work with the `inittab` respawn.<sup>4</sup> This script won't create a PID file or perform a `nohup`—because neither makes sense in the context of `respawn` since `inittab` is managing the process—and uses the `DATA-SOURCE_NAME` configuration value to create the log filename. This means that multiple Slurper instances can all be launched with different config files logging to separate log files.

#### Summary

The Slurper is a handy tool for data ingress from a local filesystem to HDFS. It also supports data egress by copying from HDFS to the local filesystem. It can be useful in

<sup>4</sup> Inittab is a Linux process management tool that you can configure to supervise and restart a process if it goes down. See <http://unixhelp.ed.ac.uk/CGI/man-cgi?inittab+5>.

situations where MapReduce doesn't have access to the filesystem, and the files being transferred are in a form that doesn't work with tools such as Flume.

Now let's look at automated pulls, for situations where MapReduce has access to your data sources.

### TECHNIQUE 3 **Scheduling regular ingress activities with Oozie**

If your data is sitting on a filesystem, web server, or any other system accessible from your Hadoop cluster, you'll need a way to periodically pull that data into Hadoop. While tools exist to help with pushing log files and pulling from databases (which we'll cover in this chapter), if you need to interface with some other system, it's likely you'll need to handle the data ingress process yourself.

There are two parts to this data ingress process: the first is how you import data from another system into Hadoop, and the second is how you regularly schedule the data transfer.

#### **Problem**

You want to automate a daily task to download content from an HTTP server into HDFS.

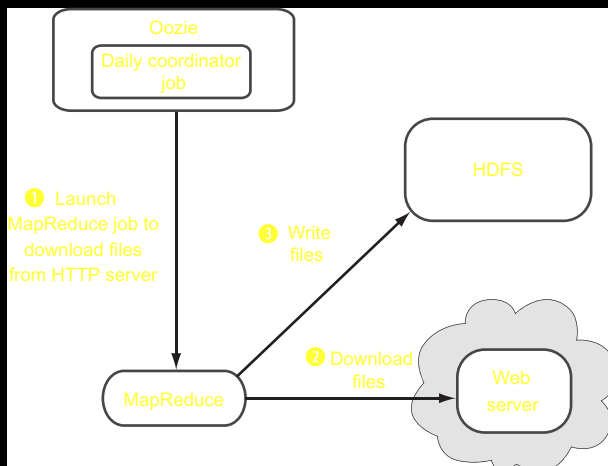
#### **Solution**

Oozie can be used to ingress data into HDFS, and can also be used to execute post-ingress activities such as launching a MapReduce job to process the ingressed data. An Apache project, Oozie started life inside Yahoo. It's a Hadoop workflow engine that manages data processing activities. For our scenario Oozie has a coordinator engine that can start workflows based on data and time triggers.

#### **Discussion**

In this technique, you perform a download of a number of URLs every 24 hours using Oozie to manage the workflow and scheduling. The flow for this technique is shown in figure 2.15.

You'll install Oozie from the instructions in Appendix A.



**Figure 2.15** Three-step data flow for Oozie ingress to manage workflow and scheduling

You'll use Oozie's data triggering capabilities to kick off a MapReduce job every 24 hours. Oozie has the notion of a coordinator job, which can launch a workflow at fixed intervals. The first step is to look at the coordinator XML configuration file. This file is used by Oozie's Coordination Engine to determine when it should kick off a workflow. Oozie uses the JSP expression language to perform parameterization, as you'll see in the following code. Create a file called `coordinator.xml` with the content shown in the next listing.<sup>5</sup>

### Oozie uses JSP to perform parameterization



What can be confusing about Oozie's coordinator is that the start and end times don't relate to the actual times that the jobs will be executed. Rather, they refer to the dates that will be materialized for the workflow. This is useful in situations where you have

<sup>5</sup> **GitHub source**—<https://github.com/alexholmes/hadoop-book/blob/master/src/main/oozie/ch2/http-download/coordinator.xml>

data being generated at periodic intervals, and you want to be able to go back in time up to a certain point and perform some work on that data. In this example, you don't want to go back in time, but instead want to schedule a job periodically every 24 hours going forward. However, you won't want to wait hours until the next day, so you can set the start date to be yesterday, and the end date to be some far-off date in the future.

Next you'll need to define the actual workflow, which will be executed for every interval in the past, and going forward when the wall clock reaches an interval. To do this, create a file called `workflow.xml`, with the content shown here.<sup>6</sup>

### Defining the past workflow using Oozie's coordinator

```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="download-http">
  <start to="download-http"/>
  <action name="download-http">
    <map-reduce>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${outputData}"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>com.manning.hip.ch2.HttpDownloadMap</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputData}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${outputData}</value>
        </property>
      </configuration>
    </map-reduce>
    <ok to="end"/>
    <error to="fail"/>
  </action>
  <kill name="fail">
    <message>Map/Reduce failed, error
      message[${wf:errorMessage(wf:lastErrorNode())}]</message>
  </kill>
  <end name="end"/>
</workflow-app>
```

*Map class.* → `<name>mapred.mapper.class</name>`

*Delete output directory before running MapReduce job.* → `<delete path="${outputData}"/>`

*Input directory for job.* → `<name>mapred.input.dir</name>`

*Output directory for job.* → `<name>mapred.output.dir</name>`

*Action needed if job fails; logs error message.* → `<error to="fail"/>`

The last step is to define your properties file, which specifies how to get to HDFS, MapReduce, and the location of the two XML files previously identified in HDFS. Create a file called `job.properties`, as shown in the following code:

<sup>6</sup> **GitHub source**—<https://github.com/alexholmes/hadoop-book/blob/master/src/main/oozie/ch2/http-download/workflow.xml>




```

nameNode=hdfs://localhost:8020
jobTracker=localhost:8021
queueName=default

oozie.coord.application.path=${nameNode}/user/${user.name}
    ➡ /http-download

```

HDFS location of two xml files.



In the previous snippet, the `oozie.coord.application.path` value is the HDFS location of the `coordinator.xml` and `workflow.xml` files that you wrote earlier in this chapter. Now you need to copy the XML files, your input file, and the JAR file containing your MapReduce code into HDFS:

```

$ hadoop fs -put src/main/oozie/ch2/http-download http-download
$ hadoop fs -put test-data/ch2/http-download/input/* http-download/
$ hadoop fs -mkdir http-download/lib
$ hadoop fs -put \
  target/hadoop-book-1.0.0-SNAPSHOT-jar-with-dependencies.jar \
  http-download/lib/

```

Finally, run your job in Oozie:

```

$ oozie job -config src/main/oozie/ch2/http-download/job.properties \
  -run
job: 0000000-111119163557664-oozie-oozi-C

```

You can use the job ID to get some information about the job:

```

$ oozie job -info 0000000-111119163557664-oozie-oozi-C
Job ID : 0000000-111119163557664-oozie-oozi-C
-----
Job Name : http-download
App Path : hdfs://user/aholmes/http-download/coordinator.xml
Status   : RUNNING
-----
ID              Status    Nominal Time
0000000-111119163557664-oozie-oozi-C@1  SUCCEEDED 2011-11-18 00:00
-----
0000000-111119163557664-oozie-oozi-C@2  SUCCEEDED 2011-11-19 00:00
-----

```

This output tells you that the job resulted in two runs, and you can see the nominal times for the two runs. The overall state is `RUNNING`, which means that the job is waiting for the next interval to occur. When the overall job has completed (after the end date has been reached), the status will transition to `SUCCEEDED`.

Because the job ran twice you should confirm that there are two output directories in HDFS corresponding to the two materialized dates:

```

$ hadoop fs -ls http-download/output/2011/11
/user/aholmes/http-download/output/2011/11/18
/user/aholmes/http-download/output/2011/11/19

```

If you wish to stop the job, use the `-suspend` option:

```
$ oozie job -suspend 0000000-111119163557664-oozie-oozi-C
```

As long as the job is running, it'll continue to execute until the end date, which in this example has been set as the year 2016.

### Summary

I showed you one example of the use of the Oozie coordinator, which offers cron-like capabilities to launch periodic Oozie workflows. The Oozie coordinator can also be used to trigger a workflow based on data availability. For example, if you had an external process, or even MapReduce generating data on a regular basis, you could use Oozie's data-driven coordinator to trigger a workflow, which could aggregate or process that data.

In this section we covered two automated mechanisms that can be used for data ingress purposes. The first technique covered a simple tool, the HDFS File Slurper, which automates the process of pushing data into HDFS. The second technique looked at how Oozie could be used to periodically launch a MapReduce job to pull data into HDFS or MapReduce.

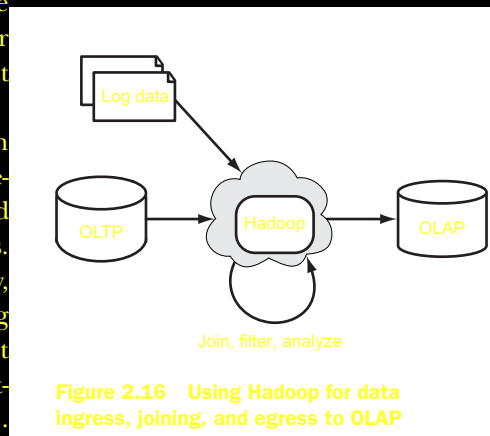
These techniques are particularly helpful in situations where the data you're working with is in a binary or semistructured form, or is only accessible via interfaces such as HTTP or FTP.

At this point in our review of data ingress we've looked at pushing log files, pushing files from regular filesystems, and pulling files from web servers. Another data source that will be of interest to most organizations is relational data sitting in OLTP databases. Next up is a look at how you can access that data.

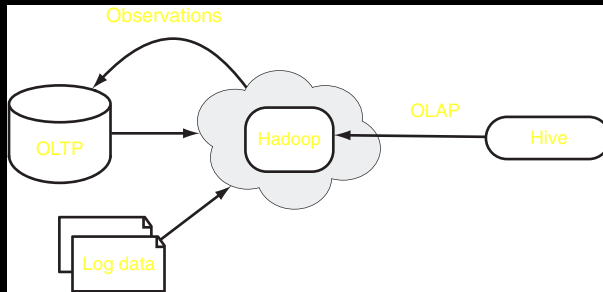
### 2.2.3 Pulling data from databases

Most organizations' crucial data exists across a number of OLTP databases. The data stored in these databases contains information about users, products, and a host of other useful items. If you want to analyze this data the traditional mechanism for doing so would be to periodically copy that data into a OLAP data warehouse.

Hadoop has emerged to play two roles in this space: as a replacement to data warehouses, and as a bridge between structured and unstructured data and data warehouses. Figure 2.16 shows the second role in play, where Hadoop is used as a large-scale joining and aggregation mechanism prior to export to an OLAP system (a commonly used platform for business intelligence applications).



**Figure 2.16** Using Hadoop for data ingress, joining, and egress to OLAP



**Figure 2.17** Using Hadoop for OLAP and feedback to OLTP systems

Facebook is an example of an organization that has successfully utilized Hadoop and Hive as an OLAP platform to work with petabytes of data. Figure 2.17 shows an architecture similar to that of Facebook's. This architecture also includes a feedback loop into the OLTP system, which can be used to push discoveries made in Hadoop, such as recommendations for users.

In either of the usage models shown in the previous figures, you need a way to bring relational data into Hadoop, and to also export it into relational databases. In the next techniques we'll cover two mechanisms you can use for database ingress. The first uses some built-in MapReduce classes, and the second provides an easy-to-use tool that removes the need for you to write your own code.

#### TECHNIQUE 4 Database ingress with MapReduce

Imagine you had valuable customer data sitting in a relational database. You used one of the previous techniques to push log data containing user activity from your web servers into HDFS. You now want to be able to do some analytics on your log data, and tie it back with your users in your relational database. How do you move your relational data into HDFS?

##### **Problem**

You want to import relational data using MapReduce.

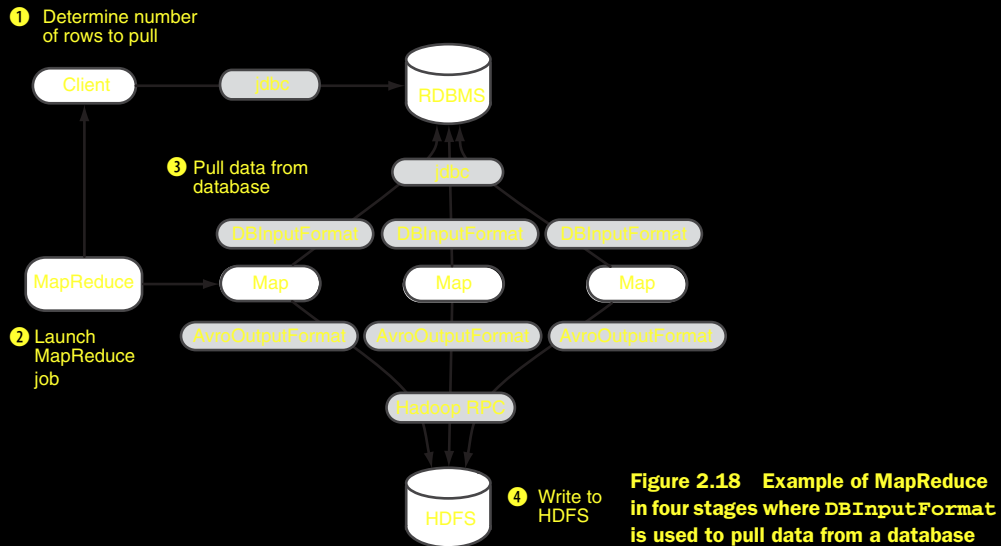
##### **Solution**

This technique covers using the `DBInputFormat` class to import relational data into HDFS. It also looks at mechanisms to guard against too many concurrent connections to your relational database.

##### **Discussion**

MapReduce contains `DBInputFormat` and `DBOutputFormat` classes, which can be used to read and write data from databases via JDBC. Figure 2.18 shows the classes you'll use.

You'll work with stock data (more details about stock data are contained in the preface to this book). In chapter 3 I'll go into more details on the `Writable` interface and an implementation of the `Writable` called `StockPriceWritable`, which represents the stock data. The `DBInputFormat` class requires a bean representation of the table being imported, which implements both the `Writable` and `DBWritable` interfaces. Because



you'll also need to implement the `DBWritable` interface, you'll do so by extending the `StockPriceWritable` class, as shown in the following code.<sup>7</sup>

#### Extending the `StockPriceWritable` class

Array of table  
column names.

```
public class StockRecord extends StockPriceWritable
    implements Writable, DBWritable {
    private final static SimpleDateFormat sdf =
        new SimpleDateFormat("yyyy-MM-dd");
    public static String [] fields = { "symbol", "quote_date",
        "open_price", "high_price", "low_price", "close_price",
        "volume", "adj_close_price"};

    @Override
    public void readFields(ResultSet resultSet)
        throws SQLException {
        int idx=2;
        setSymbol(resultSet.getString(idx++));
        setDate(sdf.format(resultSet.getDate(idx++)));
        setOpen(resultSet.getDouble(idx++));
        setHigh(resultSet.getDouble(idx++));
        setLow(resultSet.getDouble(idx++));
        setClose(resultSet.getDouble(idx++)); setVolume
        (resultSet.getInt(idx++)); setAdjClose
        (resultSet.getDouble(idx));
    }
}
```

← The implementation of the `DBWritable` interface's `readFields` method, which supplies a `JDBC` `ResultSet`. Read the values from it and set the bean properties.

<sup>7</sup> **GitHub source**—<https://github.com/alexholmes/hadoop-book/blob/master/src/main/java/com/manning/hip/ch2/StockRecord.java>

```

@Override
public void write(PreparedStatement statement)
    throws SQLException {
    int idx=1;
    statement.setString(idx++, getSymbol());
    try {
        statement.setDate(idx++,
            new Date(sdf.parse(getDate()).getTime()));
    } catch (ParseException e) {
        throw new SQLException("Failed to convert String to date", e);
    }
    statement.setDouble(idx++, getOpen());
    statement.setDouble(idx++, getHigh());
    statement.setDouble(idx++, getLow());
    statement.setDouble(idx++, getClose());
    statement.setInt(idx++, getVolume());
    statement.setDouble(idx, getAdjClose());
}
}

```

The implementation of the DBWritable interface's write method, which supplies a JDBC PreparedStatement. Write the bean properties to the statement.

Note that the array of column names in the previous code will be used later in your MapReduce configuration.

The MapReduce job, shown in the following listing,<sup>8</sup> will import from one table and write data into HDFS in Avro form.<sup>9</sup>

#### MapReduce job using DBInputFormat to import data from a relational database into HDFS

```

public static void runJob(String mysqlJar, String output)
    throws Exception {
    Configuration conf = new Configuration();
    JobHelper.addJarForJob(conf, mysqlJar);
    DBConfiguration.configureDB(conf,
        "com.mysql.jdbc.Driver",
        "jdbc:mysql://localhost/sqoop_test" +
        "?user=hip_sqoop_user&password=password");
    JobConf job = new JobConf(conf);
    job.setJarByClass(DBImportMapReduce.class);
    Path outputPath = new Path(output);
    outputPath.getFileSystem(job).delete(outputPath, true);
    job.setInputFormat(DBInputFormat.class);
    job.setOutputFormat(AvroOutputFormat.class);
    AvroJob.setOutputSchema(job, Stock.SCHEMA$);
    job.set(AvroJob.OUTPUT_CODEC, SnappyCodec.class.getName());
    job.setMapperClass(Map.class);
    job.setNumMapTasks(4);
}

```

Using a helper class to add the MySQL JAR to the distributed cache so that your map and reduce tasks have access to the JAR.

Database configuration step, where you specify the JDBC driver and the JDBC URL which contains the username and password of the MySQL user.

Set the DBInputFormat as the InputFormat class for the job.

Specify that your output should be in Avro form.

Limit the number of map tasks for the job. This should be a low number or you may bring down your database.

<sup>8</sup> **GitHub source**—<https://github.com/alexholmes/hadoop-book/blob/master/src/main/java/com/manning/hip/ch2/DBImportMapReduce.java>

<sup>9</sup> Avro is a data serialization library that we'll discuss in detail in chapter 3.

```

job.setNumReduceTasks(0);

job.setMapOutputKeyClass(AvroWrapper.class);
job.setMapOutputValueClass(NullWritable.class);

job.setOutputKeyClass(AvroWrapper.class);
job.setOutputValueClass(NullWritable.class);

FileOutputFormat.setOutputPath(job, outputPath);

DBInputFormat.setInput(
    job,
    StockRecord.class,
    "select * from stocks",
    "SELECT COUNT(id) FROM stocks");

JobClient.runJob(job);
}

public static class Map implements
    Mapper<LongWritable, StockRecord,
        AvroWrapper<Stock>, NullWritable> {

    public void map(LongWritable key,
        StockRecord value,
        OutputCollector<AvroWrapper<Stock>,
            NullWritable> output,
        Reporter reporter) throws IOException {
        output.collect(
            new AvroWrapper<Stock>(writableToAvro(value)),
            NullWritable.get());
    }
}

```

Specify your class that implements `DBWritable`, and also indicate the query to fetch the rows, as well as the query to determine the number of rows to fetch.

You want a map-only job, so set the reducers to 0.

Convert your `Stock` in `Writable` form into the `Avro` form and emit in your mapper.

Before you can continue, you'll need access to a MySQL database and have the MySQL JDBC JAR available.<sup>10</sup> You have access to a script (see note) that will create the necessary MySQL user and schema, and load the data for this technique. The script creates a `hip_sqoop_user` MySQL user, and creates a `sqoop_test` database with three tables: `stocks`, `stocks_export`, and `stocks_staging`. It then loads the `stocks` sample data (more details in the preface) into the `stocks` table. All of these steps are performed by running the following command:

```
$ bin/prep-sqoop-mysql.sh
```



#### SCRIPTS, SOURCE CODE, AND TEST DATA

All of the code in this book, sample data, and scripts (such as `run.sh`) to run the code are contained in the GitHub repository at <https://github.com/alexholmes/hadoop-book>. The preface has instructions to download, build, and run the examples.

<sup>10</sup> MySQL installation instructions can be found in appendix A, if you don't already have it installed. That section also includes a link to get the JDBC JAR.

Here's a quick peek at what the script did with the following MySQL client commands:

```
$ mysql
mysql> use sqoop_test;
mysql> show tables;
+-----+
| Tables_in_sqoop_test |
+-----+
| stocks                |
| stocks_export         |
| stocks_staging        |
+-----+
3 rows in set (0.00 sec)
mysql> select * from stocks;
+-----+-----+-----+-----+-----+-----+
| id | symbol | quote_date | open_price | high_price | low_price | ...
+-----+-----+-----+-----+-----+-----+
| 1 | AAPL | 2009-01-02 | 85.88 | 91.04 | 85.16 | ...
| 2 | AAPL | 2008-01-02 | 199.27 | 200.26 | 192.55 | ...
| 3 | AAPL | 2007-01-03 | 86.29 | 86.58 | 81.9 | ...
...
```

Now you're ready to run your MapReduce job. The utility `run.sh` will launch the `DBImportMapReduce` MapReduce job, as shown in the following code:

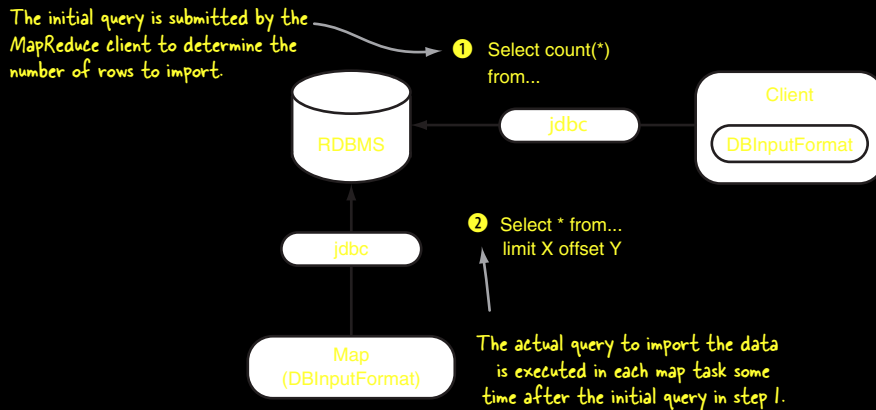


The result of this MapReduce job is a number of Avro files in the job output directory containing the results of the SQL query. The `AvroStockFileRead` class, which we'll examine in chapter 3, can be used to view the contents of the Avro files:

```
$ hadoop fs -ls output
/user/aholmes/output/part-00000.avro
/user/aholmes/output/part-00001.avro
/user/aholmes/output/part-00002.avro
/user/aholmes/output/part-00003.avro

$ bin/run.sh com.manning.hip.ch3.avro.AvroStockFileRead \
output/part-00000.avro

MSFT,2001-01-02,44.13,45.0,42.88,43.38,82413200,17.73
MSFT,2000-01-03,117.37,118.62,112.0,116.56,53228400,47.64
YHOO,2009-01-02,12.17,12.85,12.12,12.85,9514600,12.85
...
```



**Figure 2.19** The two-step process of SQL queries used for import

There's one file per map task and each contains a subset of the data that has been imported.

### Summary

There are several important considerations to bear in mind when using this technique. First, you need to make sure that you don't run with too many map tasks, because bombarding your database with thousands of concurrent reads will probably bring the DBA knocking on your door.

You should also ensure that your import and count queries are idempotent. Figure 2.19 illustrates the time differences between when the initial count query is executed as part of job submission and the subsequent queries from the map tasks. If data in the table being exported is inserted or deleted, you'll likely get duplicate records and potentially worse side effects. Therefore, either perform the import against an immutable table, or carefully choose your queries to guarantee that the results won't change during the import.

As the figure 2.19 indicates, because multiple mappers will be running with differing LIMIT and OFFSET settings, the query should be written in a way that will ensure consistent and repeatable ordering in mind, which means that the query needs to leverage a unique key, such as a primary key.

Phew! Importing data from a relational database seems more involved than it should be—which makes you wonder if there's a better way to perform the import.

## TECHNIQUE 5 Using Sqoop to import data from MySQL

The previous technique required a fair amount of work as you had to implement the Writable and DBWritable interfaces, and then write a MapReduce job to perform the import. Surely there's an easier way to import relational data!

### Problem

You want to load relational data into your cluster and ensure your writes are efficient and at the same time idempotent.



**Solution**

In this technique, we'll look at how to use Sqoop as a simple mechanism to bring relational data into Hadoop clusters. We'll walk through the process of importing data from MySQL into Sqoop. We'll also cover methods for using the regular connector, as well as how to do bulk imports using the fast connector.

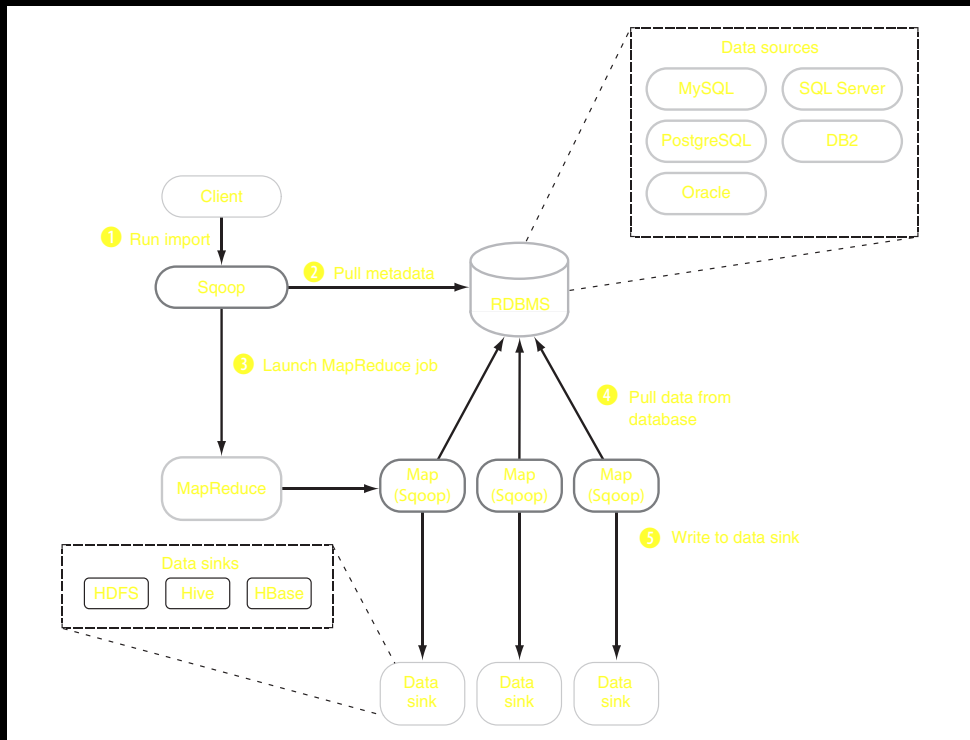
**Discussion**

Sqoop is a relational database import and export system. It was created by Cloudera, and is currently an Apache project in incubation status.

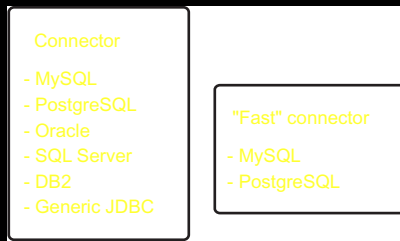
When you perform an import, Sqoop can write to HDFS, Hive, or HBase, and for exports it can do the reverse. Importing is broken down into two activities: connecting to the data source to gather some statistics, and then firing off a MapReduce job, which performs the actual import. Figure 2.20 shows these steps.

Sqoop has the notion of *Connectors*, which contain the specialized logic to read and write to external systems. Sqoop comes with two classes of connectors: a *common connector* for regular reads and writes, and a *fast connector* that uses database-proprietary batch mechanisms for efficient imports. Figure 2.21 shows these two classes of connectors and the databases that they support.

To get started, you'll need to install Sqoop, and the instructions for this are covered in appendix A. I recommend you read these instructions because they also contain steps for installing Sqoop dependencies, such as MySQL JDBC drivers. We covered



**Figure 2.20** Five-stage Sqoop import overview: connecting to the data source and using MapReduce to write to a data sink



**Figure 2.21** Sqoop connectors used to read and write to external systems

how to get MySQL installed and configured with the database, tables, and users in the previous technique, so if you haven't already done this step, go back and follow those instructions as well.

The first Sqoop command will be a basic import, where you'll specify connection information about your MySQL database, and the table you want to export:

```
$ sqoop import --username hip_sqoop_user --password password \
  --connect jdbc:mysql://localhost/sqoop_test --table stocks
```



#### **MYSQL TABLE NAMES**

MySQL table names in Linux are case sensitive. Make sure that the table name you supply in the Sqoop commands is also case sensitive.

It's generally not a good idea to have database passwords as arguments to a script because that allows other users to see your password using commands, such as `ps`, when the import is occurring. It'll also enter your shell history file. A best practice to follow is to write the password in Sqoop's option file and ensure that only you have read permissions on it:

```
$ cat > ~/.sqoop_import_options.txt << EOF
import
--username
hip_sqoop_user
--password
password
EOF
$ chmod 700 ~/.sqoop_import_options.txt
```

Sqoop also supports a `-P` option, which when present will result in you being prompted for the password.

Run the command again, this time specifying the options file you've created:

```
$ hadoop fs -rmr stocks
$ sqoop --options-file ~/.sqoop_import_options.txt \
  --connect jdbc:mysql://localhost/sqoop_test --table stocks
```

You may wonder why you had to delete the stocks directory in HDFS before rerunning the import command. Sqoop by default uses the table name as the destination in HDFS for the MapReduce job that it launches to perform the import. If you run the same command again, the MapReduce job will fail because the directory already exists. Let's take a look at the stocks directory in HDFS:

```
$ hadoop fs -ls stocks
624 2011-11-24 11:07 /user/aholmes/stocks/part-m-00000
644 2011-11-24 11:07 /user/aholmes/stocks/part-m-00001
642 2011-11-24 11:07 /user/aholmes/stocks/part-m-00002
686 2011-11-24 11:07 /user/aholmes/stocks/part-m-00003

$ hadoop fs -cat stocks/part-m-00000
1,AAPL,2009-01-02,85.88,91.04,85.16,90.75,26643400,90.75
2,AAPL,2008-01-02,199.27,200.26,192.55,194.84,38542100,194.84
3,AAPL,2007-01-03,86.29,86.58,81.9,83.8,44225700,83.8
...
```

### IMPORT DATA FORMATS

Sqoop has imported your data as comma-separated text files. It supports a number of other file formats, which can be activated with the arguments listed in table 2.5. If you're importing large amounts of data you may want to use a file format such as Avro, which is a compact data format, and use it in conjunction with compression. The following example uses the Snappy compression codec in conjunction with Avro files:

```
$ hadoop fs -rmr stocks
$ sqoop --options-file ~/.sqoop_import_options.txt \
  --as-avrodatafile \
  --compress \
  --compression-codec org.apache.hadoop.io.compress.SnappyCodec \
  --connect jdbc:mysql://localhost/sqoop_test \
  --table stocks
```

Note that the compression that's supplied on the command line must be defined in the config file, `core-site.xml`, under the property `io.compression.codecs`. The Snappy compression codec requires you to have the Hadoop native libraries installed. See chapter 5 for more details on compression setup and configuration.

You can introspect the structure of the Avro file to see how Sqoop has laid out the records by using an Avro dumper tool that I created. Sqoop uses Avro's `GenericRecord` for

**Table 2.5** Sqoop arguments that control the file formats of import commands

Argument	Description
<code>--as-avrodatafile</code>	Data is imported as Avro files.
<code>--as-sequencefile</code>	Data is imported as SequenceFiles.
<code>--as-textfile</code>	The default file format, with imported data as CSV text files.

record-level storage (more details on that in chapter 3). If you run your generic Avro dumper utility against the Sqoop-generated files in HDFS you'll see the following:

```
$ bin/run.sh com.manning.hip.ch3.avro.AvroGenericFileDumper \
  stocks/part-m-00000.avro
{"id": 1, "symbol": "AAPL", "quote_date": "2009-01-02",
 "open_price": 85.88, "high_price": 91.04, "low_price": 85.16,
 "close_price": 90.75, "volume": 26643400, "adj_close_price": 90.75}
...
```



### USING SQOOP IN CONJUNCTION WITH SEQUENCEFILES

One of the things that makes SequenceFiles hard to work with is that there isn't a generic way to access data in a SequenceFile. You must have access to the Writable class that was used to write the data. In Sqoop's case it code-generates this file. This introduces a major problem: if you move to a newer version of Sqoop, and that version modifies the code generator, there's a good chance your older, code-generated class won't work with SequenceFiles generated with the newer version of Sqoop. You'll either need to migrate all of your old SequenceFiles to the new version, or have code which can work with different versions of these SequenceFiles. Due to this restriction I don't recommend using SequenceFiles with Sqoop. If you're looking for more information on how SequenceFiles work, run the Sqoop import tool and look at the stocks.java file that's generated within your working directory.

In reality you'll more likely want to periodically import a subsection of your tables based on a query. But what if you want to import all of the Apple and Google stocks in 2007 and stick them into a custom HDFS directory? The following code shows how you would do this with Sqoop:

Store your query in variable `query`. The `%CONDITIONS` is a Sqoop macro that must be present in the `WHERE` clause of the query. It's used by Sqoop to substitute `LIMIT` and `OFFSET` options when issuing `mysql` queries.

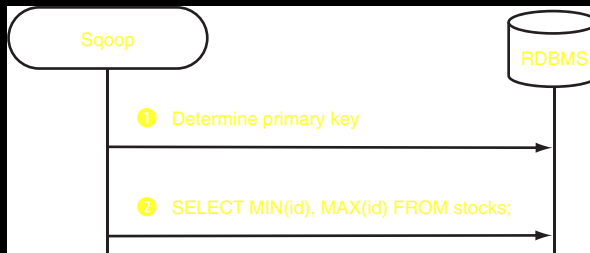
```
$ hadoop fs -rmr 2007-stocks
$ GLOBIGNORE=*
$ read -d '' query << "EOF"
select * from stocks
where symbol in ("AAPL", "GOOG")
  and quote_date between "2007-01-01" AND "2007-12-31"
  AND %CONDITIONS
EOF

$ sqoop --options-file ~/.sqoop_import_options.txt \
  --query "$query" \
  --split-by id \
  --target-dir /user/aholmes/2007-stocks \
  --connect jdbc:mysql://localhost/sqoop_test
```

Bash by default performs globbing, meaning that it'll expand wildcards like `*`. You use this command to turn this off so that the next line generates the SQL correctly.

This argument must be supplied so that Sqoop can determine which table column to use for splitting.

The `--query` SQL shown in the previous snippet can also be used to include only a subset of the columns in a table to be imported.



**Figure 2.22** Sqoop preprocessing in two steps to determine query splits

### DATA SPLITTING

How is Sqoop able to parallelize imports across multiple mappers?<sup>11</sup> In figure 2.20 I showed how Sqoop's first step is to pull metadata from the database. It inspects the table being imported to determine the primary key and runs a query to determine the lower and upper bounds of the data in the table (shown in figure 2.22). A somewhat even distribution of data within the minimum and maximum keys is assumed by dividing the delta by the number of mappers. Each mapper is then fed a unique query containing a range of the primary key.

You can configure Sqoop to use a nonprimary key with the `--split-by` argument. This can be useful in situations where the primary key doesn't have an even distribution of values between the min and max values. For large tables, however, you need to be careful that the column specified in `--split-by` is indexed to ensure optimal import times.

You can use the `--boundary-query` argument to construct an alternative query to determine the minimum and maximum values.

### INCREMENTAL IMPORTS

You can also perform incremental imports. Sqoop supports two types, *append*, which works for numerical data that's incrementing over time, such as auto-increment keys; and *lastmodified*, which works on timestamped data. In both cases you need to specify the column using `--check-column`, the mode via the `--incremental` argument (the value must be either *append* or *lastmodified*), and finally, the actual value to use to determine the incremental changes, `--last-value`. Using the example, if you want to import stock data that's newer than January 1, 2005, you'd do the following:

```

$ hadoop fs -rmr stocks
$ sqoop --options-file ~/.sqoop_import_options.txt \
  --check-column "quote_date" \
  --incremental "lastmodified" \
  --last-value "2005-01-01" \
  --connect jdbc:mysql://localhost/sqoop_test \
  --table stocks
...
tool.ImportTool: --incremental lastmodified
  
```

<sup>11</sup> By default Sqoop runs with four mappers. The number of mappers can be controlled with the `--num-mappers` argument.

```

tool.ImportTool: --check-column quote_date
tool.ImportTool: --last-value 2011-11-24 14:49:56.0
tool.ImportTool: (Consider saving this with 'sqoop job --create')
...

```

### SQOOP JOBS AND THE METASTORE

You can see in the command output the last value that was encountered for the increment column. How can you best automate a process that can reuse that value? Sqoop has the notion of a *job*, which can save this information and reuse it in subsequent executions:

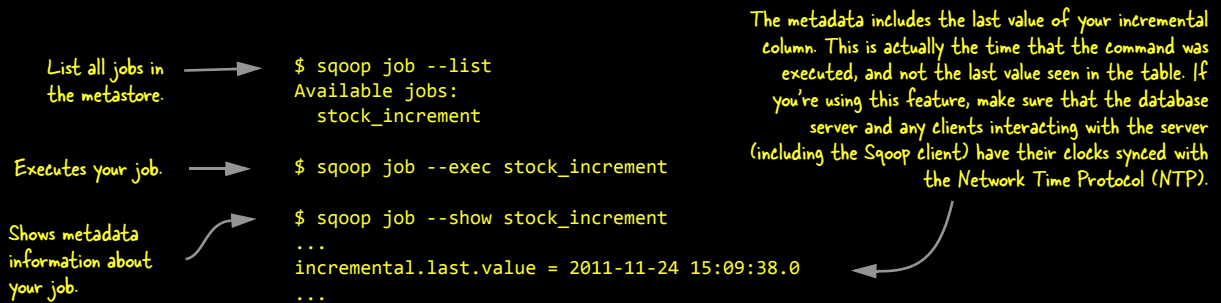
```

$ sqoop job --create stock_increment -- import \
--append \
--check-column "quote_date" \
--incremental "lastmodified" \
--last-value "2005-01-01" \
--connect jdbc:mysql://localhost/sqoop_test \
--username hip_sqoop_user \
--table stocks

```

This merely saves the notion of this command as a job in something called the Sqoop *metastore*. A Sqoop metastore keeps track of all jobs. By default, the metastore is contained in your home directory under `.sqoop`, and is only used for your own jobs. If you want to share jobs, you would need to install a JDBC-compliant database and use the `--meta-connect` argument to specify its location when issuing job commands.

The job create command executed in the previous example didn't do anything other than add the job to the metastore. To run the job you need to explicitly execute it as shown here:



Unfortunately, the `--options-file` argument, which referred to your local file with your username and password, doesn't work with jobs in Sqoop. The password also can't be specified when creating the job. Sqoop will instead prompt for the password when running the job. To make this work in an automated script you need to use Expect, a Linux automation tool, to supply the password from a local file when it detects Sqoop prompting for a password. The source of an Expect script that works with Sqoop is on GitHub at <http://goo.gl/yL4KQ>.

**FAST MYSQL IMPORTS**

What if you want to bypass JDBC altogether and use the fast MySQL Sqoop connector for a high-throughput load into HDFS? This approach uses the `mysqldump` utility shipped with MySQL to perform the load. You must make sure that `mysqldump` is in the `PATH` of the user running the MapReduce job. To enable use of the fast connector you must specify the `--direct` argument:

```
$ hadoop fs -rmr stocks
$ sqoop --options-file ~/.sqoop_import_options.txt \
  --direct \
  --connect jdbc:mysql://localhost/sqoop_test \
  --table stocks
```

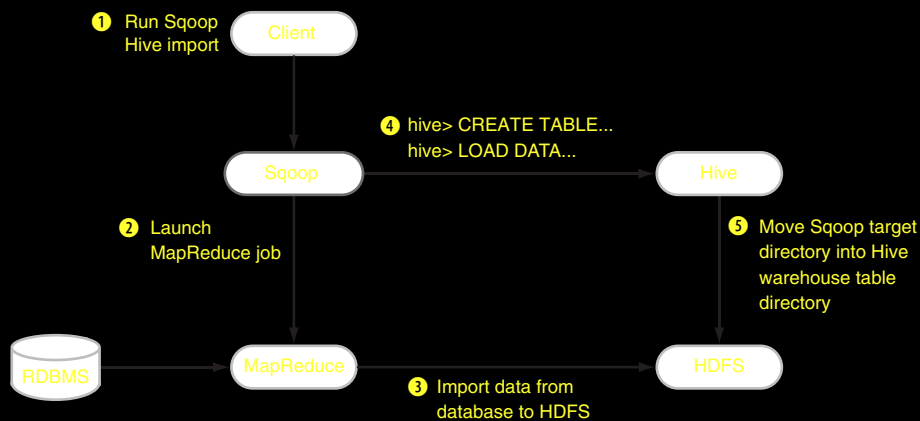
What are the disadvantages of fast connectors? First, only MySQL and PostgreSQL are currently supported. Fast connectors also only work with text output files—specifying Avro or SequenceFile as the output format of the import won't work.

**IMPORTING TO HIVE**

The final step in this technique is to use Sqoop to import your data into a Hive table. The only difference between an HDFS import and a Hive import is that the Hive import has a postprocessing step where the Hive table is created and loaded, as shown in figure 2.23.

When data is loaded into Hive from an HDFS file or directory, such as in the case of Sqoop Hive imports (step 4 in the figure diagram), for the sake of efficiency, Hive moves the directory into its warehouse rather than copying the data (step 5). The HDFS directory that the Sqoop MapReduce job writes to won't exist after the import.

Hive imports are triggered via the `--hive-import` argument. Just like with the fast connector, this option isn't compatible with the `--as-avrodatafile` and `--as-sequence-file` options.



**Figure 2.23** The five-stage Sqoop Hive import sequence of events

```
$ hadoop fs -rmr stocks
$ sqoop --options-file ~/.sqoop_import_options.txt \
  --hive-import \
  --connect jdbc:mysql://localhost/sqoop_test \
  --table stocks

$ hive
hive> select * from stocks;
OK
1 AAPL 2009-01-02 85.88 91.04 85.16 90.75 26643400 90.75
2 AAPL 2008-01-02 199.27 200.26 192.55 194.84 38542100 194.84
3 AAPL 2007-01-03 86.29 86.58 81.9 83.8 44225700 83.8
4 AAPL 2006-01-03 72.38 74.75 72.25 74.75 28829800 74.75
...
```



#### IMPORTING STRINGS CONTAINING HIVE DELIMITERS

You'll likely have downstream processing issues if you're importing columns that can contain any of Hive's delimiters ( `\n`, `\r` and `\01` characters). You have two options in such cases: either specify `--hive-drop-import-delims`, which will remove conflicting characters as part of the import, or specify `--hive-delims-replacement`, which will replace them with a different character.

If the Hive table already exists the data will be appended to the existing table. If this isn't the desired behavior, you can use the `--hive-overwrite` argument to indicate that the existing table should be replaced with the imported data.

Data in Hive can also be compressed. Since the LZOP compression codec is the only splittable codec<sup>12</sup> in Hadoop (see chapter 5 for details), it's the codec that should be used for Hive compression. The following example shows how to use the `--hive-overwrite` in conjunction with enabling LZOP compression. For this to work you'll need to have built and installed LZOP on your cluster, since it isn't bundled with Hadoop (or CDH) by default. Refer to chapter 5 for more details.

```
$ hive
hive> drop table stocks;

$ hadoop fs -rmr stocks

$ sqoop --options-file ~/.sqoop_import_options.txt \
  --hive-import \
  --hive-overwrite \
  --compress \
  --compression-codec com.hadoop.compression.lzo.LzopCodec \
  --connect jdbc:mysql://localhost/sqoop_test \
  --table stocks
```

<sup>12</sup> bzip2 is also a splittable compression codec which can be used in Hadoop, but its write performance is so poor that in practice it's rarely used.



Finally, you can use the `--hive-partition-key` and the `--hive-partition-value` to create different Hive partitions based on the value of a column being imported. For example, if you want to partition your input by date, you would do the following:

```
$ hive
hive> drop table stocks;

$ hadoop fs -rmr stocks

$ read -d '' query << "EOF"
SELECT id, quote_date, open_price
FROM stocks
WHERE symbol = "AAPL" AND $CONDITIONS
EOF

$ sqoop --options-file ~/.sqoop_import_options.txt \
  --query "$query" \
  --split-by id \
  --hive-import \
  --hive-table stocks \
  --hive-overwrite \
  --hive-partition-key symbol \
  --hive-partition-value "AAPL" \
  --connect jdbc:mysql://localhost/sqoop_test \
  --target-dir stocks

$ hadoop fs -lsr /user/hive/warehouse
/user/hive/warehouse/stocks/symbol=AAPL/part-m-00000
/user/hive/warehouse/stocks/symbol=AAPL/part-m-00001
...
```

Now, the previous example isn't optimal by any means. Ideally, a single import would be able to create multiple Hive partitions. Because you're limited to specifying a single key and value, you'd need to run the import once per unique partition value, which is laborious. You'd be better off importing into a nonpartitioned Hive table, and then retroactively creating partitions on the table after it had been loaded.

Also, the SQL query that you supply to Sqoop must also take care of filtering out the results, such that only those that match the partition are included. In other words, it would have been useful if Sqoop would have updated the `WHERE` clause with `symbol = "AAPL"` rather than having to do this yourself.

### **Summary**

Obviously, for Sqoop to work your Hadoop cluster nodes need to have access to the MySQL database. Common sources of error are either misconfiguration or lack of connectivity from the Hadoop nodes. It's probably wise to log on to one of the Hadoop nodes and attempt to connect to the MySQL server using the MySQL client, and/or attempt access with the `mysqldump` utility (if using a fast connector).

Another important note when using a fast connector is that it's assumed that `mysqldump` is installed on each Hadoop node, and is in the `PATH` of the user running the map tasks.