

All Your HBase Are Belong to Clojure

Posted on 12 Jan 2012

I'm sure you've heard a variation on this story before...

So I have this web crawler and it generates these super-detailed log files, which is great 'cause then we know what it's doing but it's also kind of bad 'cause when someone wants to know why the crawler did this thing but not that thing I have, like, literally *gajigabytes* of log files and I'm using grep and awk and, well, it's not working out. Plus what we really want is a nice web application the client can use.

I've never really had a good solution for this. One time I crammed this data into a big Lucene index and slapped a web interface on it. One time I turned the data into JSON and pushed it into CouchDB and slapped a web interface on that. Neither solution left me with a great feeling although both worked okay at the time.

This time I already had a Hadoop cluster up and running, I didn't have any experience with HBase but it looked interesting. After hunting around the internet, thought this might be the solution I had been seeking. Indeed, loading the data into HBase was fairly straightforward and HBase has been very responsive. I mean, very responsive now that I've structured my data in such a way that HBase *can* be responsive.

And that's the thing: if you are loading literally gajigabytes of data into HBase you need to be pretty sure that it's going to be able to answer your questions in a reasonable amount of time. Simply cramming it in there probably won't work (indeed, that approach probably won't work great for anything). I loaded and re-loaded a test set of twenty thousand rows until I had something that worked.

Setting Up Your Environment

Both [Hadoop](#) and [HBase](#) have pretty decent documentation that goes over installing and configuring these tools. I won't re-hash that here, especially given the variance in setup (single node for development verses multi-node for deployment). In my experience, getting a multi-node development setup with virtual machines (i.e. with [Vagrant](#)) was problematic; even with host names properly configured I had issues with connections timing out.

I'm using Clojure 1.3, the libraries that I needed were available in [Clojars](#) but are built for 1.2. I forked these libraries and moved them to 1.3 to keep things neat for myself. If you're using Clojure 1.3, you can clone these repositories and build your own copies.

- [Clojure Hadoop](#) (or [my fork for Clojure 1.3](#) compatibility)
- [Clojure HBase](#) (or [my fork for Clojure 1.3](#) compatibility)

My fork of the HBase library requires a 1.3 compatible version of the monolithic Clojure Contrib library. You can [download a copy if you need it](#).

What is HBase Again?

[HBase](#) provides a [BigTable](#)-like database on top of Hadoop. Uh-oh, I said *database!* Well, it's not at all like a SQL database; it's more like a really big sorted map. In the simplest scenario, given a particular key HBase can quickly give you all of the values associated with that key. Because it's sorted, getting all of the rows between one key and another is pretty fast. You can also quickly get all of the rows that begin with a particular key.

While Hadoop will let you store huge amounts of data and run jobs to analyze that data, you need to do something with the results of those jobs. Often making those results available for people to look at is good enough, in my case the client wants to be able to search for specific information (for instance, the log records produced when a particular host was crawled) and this isn't something I could do easily with Hadoop. My solution was to store the crawl logs on my Hadoop cluster and then run a Hadoop job that would load this data into HBase. HBase will let customer query for data on-demand.

This is important to note: it's a good solution for myself and my peculiar scenario. There are other data stores that may be a better fit for someone else. I already have a Hadoop cluster up, running and in active use. If I was already using [Riak](#), I most likely would have concentrated my efforts on making that work instead.

An HBase table is made up of rows, each row contains any number of columns and each one of those columns belongs to a *column family*. Each column family can then have a qualifier and a value. It took a bit of thinking to get my head around what a column family was, it's purpose and how it should be used. Generally speaking, items in the same column family are stored together physically and that makes fetching all of the data for a particular row or group of rows column family faster. There's a good discussion about how HBase actually stores this data [in the O'Reilly HBase book](#). It's definitely worth reading.

Goals: Web Application for Log Viewing

Before we move further down this path and try to figure out what data should go where, we're going to take a brief detour and look at the web application that will be presenting the log data. In my opinion, this will help us get a handle on what data we need and, at least in terms of idealized application usage, when we will need it.

The data set I'm working with looks like this: a set of files, each line of each file contains one web crawler transaction. This will have the date and time of the transaction, the URL that the crawler tried to crawl, the response from the host serving that resource and the status the indexer assigned to the result. Here's a sample row:

Timestamp	Response	Status	URL
2011-01-12-12:44:33	200	NEW	http://twitch.nervestaple.com
2011-01-12-12:45:12	200	NEW	http://twitch.nervestaple.com/index

The client wants the ability to see a summary page for each crawled host, this page should display the number of URLs for the host, how many returned content and how many had problems. They'd like to have a report page for each crawled URL that provides some summary information and a list of transactions for that URL. They want to be able to bang in an URL or host name into a search box and get dropped at the right page. They say that's everything but I can't help but have the feeling that this really only represents everything they've thought of *so far*.

Laying Out My Log Data

Given the way that column families work, it makes sense to store data that we want to access all at once in the same column family. While it's hard to know what data people will want, we can make some guesses. In my case, I know what my web application is going to look like and I can use that information to inform my choices.

First up, we want the ability to fetch all of the transactions for a given URL. This means that we'll

want an HBase table where the key will at least start with the URL. Each URL could be crawled multiple times, so we'll combine the URL with the date and time crawled. It'll look unpleasant but something like this:

```
http://twitch.nervestaple.com/index2011-01-12-12:45:12
```

HBase will store our data sorted by key, this means that when we want all of the transactions for a given URL it can jump to the first row that starts with our URL and then return each row there-after in sequence.

For our log data, we won't have particularly interesting column families. I chose to name my columns families "request", "transaction", "response" and "crawler". To follow through with our example, the data in HTable would look something like this:

Row	Column Family	Qualifier	Value
http://twitch...	request	url	http://twitch...
	request	host	twitch...
	transaction	date-time	2011-01-12-12:45:12
	response	code	200
	crawler	status	New

This layout makes it easy to pull out the rows for a particular URL. When we want to provide summary information for a specific host, this isn't as helpful. While we could scan through all of the records for the host in question, the client would have to crunch through the returned data and calculate the summary data. This will be problematic in practice, instead we're going to leverage another HBase function: counters.

HBase will let us treat a column as if it were a counter, this lets us atomically increment the counter in just one call to the server. As we load data into HBase, we'll increment these counters; this will make it easy to provide the summary data the client demands. For instance, we'll have a table called "host-stats" with the column families "transactions". We can use a qualifier like "total" to represent all of the transactions for that host. Some example rows might look like this:

Row	Column Family	Qualifier	Value
twitch.nervestaple.com	transactions	total	104
bakery.somewhere.com	transactions	total	83

In practice it'd make sense to break this down further. We could have a family called "transactions-yearly" and use the year portion of the crawl date as our qualifier. We could have a family called "transactions-monthly" and use a combination of the month and year portions of the crawl date as the qualifier. You get the idea.

HBase will let us have as many items of data for each column family as we want, in our last example we're using the qualifier to distinguish between each year and month combination in the "transactions-monthly" family.

Loading Data into HBase with Hadoop

There are a couple different ways to load data into HBase. Given the amount of computation we have to do (incrementing the counters based on host and URL) it makes sense to write a Hadoop map/reduce job to load in this data.

Setting Up Our Project

We'll be using both the Clojure Hadoop and the Clojure HBase libraries. I'm going to assume that you're using [Leinigen](#) to manage your Clojure projects. If not, you'll want to revisit that. Go ahead and create a new project with...

```
lein new crawl-log-loader
```

Next up you'll want to add the dependencies to your "project.clj" file. If you built the libraries from my repositories, you'll want to add the following:

```
(defproject crawl-log-loader "1.0-SNAPSHOT"
  :description "Load Crawler Log data"
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [clojure-hbase "0.90.5-SNAPSHOT"]
                 [clojure-hadoop "1.3.3-SNAPSHOT"]
                 [org.clojure/tools.logging "0.2.3"]
                 [org.clojure/tools.cli "0.2.1"]]
  :dev-dependencies [[org.codehaus.jackson/jackson-mapper-asl "1.9.2"]
                     [org.slf4j/jcl104-over-slf4j "1.4.3"]
                     [org.slf4j/slf4j-log4j12 "1.4.3"]]
  :main crawl-log-loader.core)
```

We're defining our dependencies on Clojure, the HBase library and the Hadoop library. We need the Clojure "tools.logging" library so that we can log message while our Hadoop job is running. I like to include "tools.cli" so I can look up documentation for functions from the REPL while I work. Lastly, the Hadoop libraries depend on Jackson and SLF4J; they'll be present on our Hadoop cluster but we'll need them around in order to build our application.

Use, Require and Import

With the project setup, it's time to add some code. Open up the core file (src/crawl_log_loader/core.clj), and add something like this...

```
(ns crawl-log-loader.core
  (:use [clojure.repl]
        [clojure.tools.logging]
        [clojure.tools.cli])
  (:require [clojure.string :as string]
            [clojure-hadoop.gen :as gen]
            [clojure-hadoop.imports :as imp]
            [clojure-hbase.core :as hb])
  (:import [org.apache.hadoop.util Tool]
           [org.apache.hadoop.hbase.client Increment]
           [org.apache.commons.logging LogFactory]
           [org.apache.commons.logging Log]
           [java.net URL]
           [org.apache.hadoop.hbase.util Bytes]))
```

I've added the Clojure REPL, logging and CLI tools to make it easier to bootstrap your application and parse out command line arguments. Since we're parsing log files, we'll need the Clojure String library as well. After we pull in our libraries for Hadoop and HBase, we import some of the Java classes we'll need to make those libraries work.

I haven't mentioned it before, but HBase stores everything as an array of bytes. It has no notion of type, it just sees byte arrays. Our last import provides a utility class that makes it easier for us to convert objects like strings and numbers into byte arrays.

Define Our Hadoop Job

We're going to do this a little backwards and add the function that sets up the Hadoop job next. This function will reference our map and reduce functions although we haven't written those yet.

```
(defn tool-run
  "Provides the main function needed to bootstrap the Hadoop application."
  [^Tool this args-in]

  ;; define our command line flags and parse out the provided
  ;; arguments
  (let [[options args banner]
        (cli args-in
          ["-h" "--help"
           "Show usage information" :default false :flag true]
          ["-p" "--path" "HDFS path of data to consume"]
          ["-o" "--output" "HDFS path for the output report"])]

    ;; display the help message
    (if (:help options)
        (do (println banner) 0)

        ;; setup and run our job
        (do
         (doto (Job.)
          (.setJarByClass (.getClass this))
          (.setJobName "crawl-log-load")
          (.setOutputKeyClass Text)
          (.setOutputValueClass LongWritable)
          (.setMapperClass (Class/forName "crawl-log-loader.core_mapper"))
          (.setReducerClass (Class/forName "crawl-log-loader.core_reducer"))
          (.setInputFormatClass TextInputFormat)
          (.setOutputFormatClass TextOutputFormat)
          (FileInputFormat/setInputPaths (:path options))
          (FileOutputFormat/setOutputPath (Path. (:output options)))
          (.waitForCompletion true))
         0))))
```

This article isn't about parsing command line arguments, but the above is a good habit to get into. We use the CLI library to both setup our arguments and to parse those arguments out into a hash-map. More information on how this library works [can be found on the project's page](#).

Hadoop wants our application to return a status code that indicates healthy completion of the job or exiting under an error condition. We return "0" to indicate that our job exited normally. In real life you may want to do something more clever.

If our app isn't invoked with the "-h" or "--help" flag, we setup the Hadoop job. We create a new Job object and set a bunch of fields. Note that we set the output key and value class. The main purpose of this job is to load data into HBase but we'll also output the number of transactions per host. This could be used for any number of things, perhaps we want to double-check the data stored in HBase.

We set the mapper and reducer classes, we'll write those up next. We set the input and output formats; the [TextInputFormat](#) reads plain text files line-by-line, a good fit our log input. The [TextOutputFormat](#) writes plain text files.

Mapping Function

We'll now add our mapping function. Make sure that you add your own mapping function *above* the definition of your "tool-run" function.

```
(defn mapper-map
  "Provides the function for handling our map task. We parse the data,
```

```

apply it to HBase and then write out the host and 1. This output is
used to provide a summary report that details the number of URLs
logged per host."
[this key value ^MapContext context]

;; parse the data
(let [parsed-data (parse-data value)]

    ;; apply the data to HBase
    (process-log-data parsed-data)

    ;; write our counter for our reduce task
    (.write context
     (Text. (:host parsed-data))
     (LongWritable. 1)))

```

This isn't so tricky! We read in a key and a value, they key will be the line number of the file being processed and the value will be the text of that line (a log entry). We don't really care which line of what file this entry came from so we ignore it. Then we parse the line of log data into a hash-map, apply that data to HBase with our "process-log-data" function (yet unwritten) and then write out data for our final report.

We're writing out the host for the URL that was crawled in this log entry and the number 1. During the reduce phase we'll sum the values for each host and output the total number of transactions. In fact, let's do that right now.

Reduce Function

```

(defn reducer-reduce
  "Provides the function for our reduce task. We sum the values for
  each host yeilding the number of URLs logged per host."
  [this key values ^ReduceContext context]

  ;; sum the values for each host
  (let [sum (reduce + (map (fn [^LongWritable value]
                           (.get value))
                          values))]

    ;; write out the total
    (.write context key (LongWritable. sum)))

```

Again, this function isn't scary at all. We map over the incoming values (Hadoop wraps the number in a LongWritable instance), pull out the actual values and reduce those values into our final sum. We write out the key, which is the name of the host and the sum, the total number of transactions for this host.

We're nearly through, the last function is the bit that applies our data to HBase.

Load Data into HBase

We need to parse out our line of log data into something easier to work with, a hash-map. My files are separated with spaces making this very easy.

```

(defn parse-data
  "Parses a String representing a row of data from an ESP crawler log
  into a hash-map of data."
  [text]

  ;; parse out the row of data
  (let [data (string/split (str text) #"\s+")]

```

```
;; return a map of data
{:timestamp (nth data 0)
 :reponse (nth data 1)
 :crawler-response (nth data 2)
 :url (nth data 3)
 :host (.getHost (URL. (nth data 3)))})
```

This is a bit simplistic, but you get the idea. In practice you'd probably want to be more careful and make sure the input data is valid. You'd likely want to split the date and time out or even parse it back into a real date instance.

Lastly, we want to add this row to HBase and update some counters.

```
(defn process-log-data
  "Handles the processing of log data by applying the map of data to
  the proper counters and adding the correct rows to our HBase
  tables."
  [parsed-data]

  ;; add our row of data
  (hb/with-table [urls (hb/table "urls")]
    (hb/put urls
      (str (:url parsed-data)
           (:timestamp parsed-data))
      :values [:request [:url (:url parsed-data)]]
      :values [:request [:host (:host parsed-data)]]
      :values [:transaction [:timestamp (:timestamp parsed-data)]]
      :values [:response [:http (:response parsed-data)]]
      :values [:response [:crawler (:crawler-response parsed-data)]]))

  ;; update our host stats table
  (hb/with-table [host-stats (hb/table "host-stats")]
    (.incrementColumnValue host-stats
      (hb/to-bytes (:host parsed-data))
      (hb/to-bytes "transactions")
      (hb/to-bytes "total")
      (.longValue 1))))
```

We're keeping this simple so that you get an idea of how this works. We add the row of data to our "urls" HBase table, then we increment to total number of transactions for this host in the "host-stats" table. We don't have to worry if there's already a row in the "host-stats" table for this host, if there isn't then HBase will create a new row with a value of zero and then increment it by our supplied value, 1.

Package and Deploy

Deployment is straightforward, create an "uberjar" with Leiningen and then copy that out to your Hadoop cluster. From there you can invoke the JAR with an input and output path.

```
lein uberjar
...watch Leiningen work...
```

```
scp crawl-log-loader-1.0-SNAPSHOT-standalone.jar hadoop1.local:/hadoop
```

```
ssh hadoop1.local
...log into your Hadoop node
```

And then run your job. It turns out this is trickier than you'd think. If you simply invoke the JAR with the "hadoop" command it will run on the local job runner (it won't run distributed across the

cluster) because it won't be able to find your HBase install. To fix this, create a new folder in your project directory called "resources". Files in this folder will be bundled up by Leiningen into your final JAR and they'll be present on the class-path when the application is launched.

Next, create a file called "hbase-site.xml" in this folder. It should look something like this:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>hadoop1.local</value>
  </property>
</configuration>
```

The Zookeeper Quorum value should list all of your Zookeeper nodes, if you're running in standalone development mode then there's just the one. If you are running in production this should be a comma separated list of host names. Lastly, you can use a script similar to the following to start your job.

```
export HBASE_HOME=/hbase
export HADOOP_HOME=/hadoop

HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` \
  ${HADOOP_HOME}/bin/hadoop jar crawl-log-loader-1.0-SNAPSHOT....jar \
  -p hdfs://hadoop1.local:54310/data/in \
  -o hdfs://hadoop1.local:54310/data/out
```

This sets up environment variables that point to our HBase and Hadoop installations. We then load our application in the context of our Hadoop instance and add our HBase installation to the class path. The "-p" and "-o" flags at the end are interpreted by our application.

For Great Justice

While this isn't production code it should be enough to get you started. We now have some data loaded and that will make it a lot easier to explore HBase and evaluate the product. Once again, I am amazed at how easy it is to work with these frameworks using Clojure! A big thanks goes out to the developers and maintainers of the Hadoop and HBase libraries that make this all so easy, as well as the developers and maintainers of Hadoop, HBase and Clojure themselves.

Related Posts

- 04 Dec 2011 - [Java Doesn't Need to Be So Bad](#)
- 09 Oct 2011 - [Blogging With Emacs](#)

Comments

•

[Disqus](#)

- [Login](#)

- [About Disqus](#)
- [Like](#)
- [Dislike](#)
- 
- 
- and 11 others liked this.

Glad you liked it. Would you like to share?

Facebook

Twitter

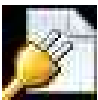
- [Share](#)
- [No thanks](#)


Sharing this page ...

Thanks! [Close](#)

[Login](#)


Add New Comment



- Post as ...
- Image 
-

Sort by popular now Sort by best rating Sort by newest first Sort by oldest first

Showing 2 comments

- 

cris_weber 1 comment collapsed [Collapse](#) [Expand](#)

Hi Christopher, great post! I've made some Hadoop Jobs using clojure-hadoop, but with wrappers and gen-job macro instead of dealing with Hadoop classes.

Will your fork to Clojure 1.3 be pulled to alexott/clojure-hadoop? I'm asking because I also made a fork and I have a DistributedCache wrapper to push to my fork, and it will be nice to merge all those features together.

- [A Like](#)
- [Reply](#)
- [1 day ago](#)
- [0 Like](#)
- [F](#)



•

tbatchelli 1 comment collapsed [Collapse](#) [Expand](#)

Great post. One missing piece is how to build the Hadoop cluster in the first place, for which we built a clojure solution based on Pallet: <https://github.com/pallet/pall...>