Cloudera: Using Cloudera's Hadoop AMIs to process EBS datasets on EC2

Cloudera's Notes

Using Cloudera's Hadoop AMIs to process EBS datasets on EC2

Share

Monday, May 11, 2009 at 5:50pm

> *A while back, we noticed a blog post From Arun Jacob over at Evri (if you haven't seen Evri before, it's a pretty impressive take on search UI). We were particularly interested in helping Arun and others use EC2 and Hadoop to process data stored on EBS as Amazon makes many public data sets available. After getting started, Arun volunteered to write up his experience, and we're happy to share it on the Cloudera blog.-Christophe*

Background

A couple of weeks ago I managed to get a Hadoop cluster up and running on EC2 using the /src/contrib/ec2 scripts found in the 0.18.3 version of Hadoop. This experience was not entirely pain free, and in order to spin up clusters without a lot of hand mods, I was going to have to modify those scripts to work with an explicit AMI in order to work around some of the issues I had run into.

Fortunately, before I could get going on that rewrite, Christophe from Cloudera contacted me and let me know that they had addressed all of those issues, and invited me to try instantiating and running my MapReduce job using their scripts and AMIs.

Prerequisites

Prior to running any of the scripts/code below, I did the following:

1. Signed up for Amazon S3 and EC2, see the EC2 Getting Started Guide .
2. Installed the Amazon EC2 command line API tools and put them on my path.
3. Set the following environment variables:
    1. AWS_ACCOUNT_ID — available through your AWS Account info
    2. AWS_ACCESS_KEY_ID –same
    3. AWS_SECRET_ACCESS_KEY — same
    4. EC2_CERT — fully qualified path to cert file.
    5. EC2_HOME — where you installed EC2 API.
    6. EC2_PRIVATE_KEY — fully qualified path to private key file. The file needs to have permissions set to 600 (rw for user only) in order to be used during initial cluster setup.
4. Installed the 0.18.3 version of Hadoop for local debugging purposes.
5. Installed the IBM MapReduce Tools for Eclipse plugin to my version of Eclipse (3.4.1). NOTE: I was unable to get the plugin to debug against an actual cluster, but I could run my MapReduce jobs in isolation against the local FS, and this allowed me to catch a lot of issues even before running on my local single node Hadoop cluster.
6. Installed the ElasticFox and S3 Organizer FireFox plugins.

Initializing the Cluster

There was no drama here. When it was time to create another (larger) cluster up on EC2 for another job that we were going to run, I downloaded the scripts from Cloudera, put in my Amazon Key and security group info as instructed , and ran

```
./hadoop-ec2 launch-cluster {name of cluster} {number of instances}
```

That's it. While I'm happy that I worked with the original ec2 scripts from hadoop/src/contrib, I'm even happier that Cloudera's rewrite makes this a one line operation. All HDFS initialization, master-slave connections, and other cluster setup was done for me.

I also set up access to the Hadoop JobTracker and NameNode (HDFS) Web UIs, which allow me to track and debug job and HDFS status. I modified the hadoop master security group, granting http access to ports 50030 (JobTracker) and 50070 (NameNode). Note that you should constrain access to those nodes using CIDR block notation, because they have no local auth. There are many ways to do this, including using the ec2 command line API , but I used ElasticFox, a great UI for doing one-off things like setting up security groups. The instructions for adding permissions to the master are found in the Cloudera EC2 setup page.

Slave node TaskTracker UIs are useful because they allow you to drill into specific task logs. However, the JobTracker UI tries to link to those UIs via their Amazon Internal DNS names. In order to get the slave node TaskTracker UIs to work, you would need to use hadoop-ec2 to start a local proxy, and use foxyproxy to set up your browser to work with the hadoop-ec2 proxy. Details on creating a pattern based proxy to access Amazon Internal DNS names using foxyproxy can be found in the troubleshooting section of the Amazon Elastic MapReduce Documentation. To start the proxy, run

```
hadoop-ec2 proxy myclustername
```

Make sure that foxyproxy is configured to talk to the proxy on port 6666.

Accessing the Cluster

In this note
No one.

The most logical way to get to the cluster is via the master instance. Starting jobs, uploading files to HDFS, and other tasks are all done from the master. The master (via NameNode and JobTracker) takes care of distributing input data to the nodes and running jobs (machine) local to those nodes. There are a couple of ways to access the master. From the elastic fox UI, select the master instance, then click on the key icon from the Instances tab. This opens up a shell to the master. You can also get command line access using the Cloudera hadoop-ec2 script:

```
{location of scripts}/bin/hadoop-ec2 login myclustername
```

Either way is required to actually run a job on the cluster. From the master it is very easy to ssh into the nodes — something I've found necessary when debugging map/reduce logic gone awry:

```
ssh {internal or external dns name of slave node}
```

is all that is required, since ssh keygen was taken care of during cluster setup.

Running A Job

Now that the cluster is up, accessible, and ready to go, I want to run a MapReduce job on it.

At Evri one of the many sources we process as we go about creating a data graph of people, places, and things is Wikipedia. Wikipedia not only contains relevant facts about entities — it also contains a lot of useful metadata about those entities. Some of that metadata is immediately available in the article. The more interesting data is obtained by traversing Wikipedia dumps and aggregating basic metadata.

One interesting piece of metadata is the number of inlinks (links to) for each Wikipedia article from other Wikipedia articles. An article with a lot of inlinks tends to be better known than an article with fewer inlinks. Collecting inlink count per article helps us get a baseline of how popular that article is.

In order to do that, I actually need to extract all of the outlinks (links away) from each Wikipedia article, tracking where they are linking to. Then I need to aggregate the link destinations and summarize the link counts for each unique destination. This is a perfect candidate for a MapReduce job — transforming data in one form to another. It is very analogous to the canonical word frequency count MapReduce job.

One of the great things about the intersection of Hadoop and EC2 is the ability to use Amazon's Public Datasets as input data for a MapReduce. That's right, huge amounts of maintained, clean data, there for the taking. The dataset that is most useful to us is the Freebase Wikipedia Extraction (WEX) data set. Freebase processes the raw (wikitext) Wikipedia dumps and converts the wiki text to XML, which makes programmatic analysis of Wikipedia much easier for the rest of us. This dump is available from their website, but they've gone the extra mile and have uploaded the results to Amazon in TSV (tab separated value) format, which is exactly what we need to process the data in Hadoop — thanks Freebase!

The Source Code

We've defined exactly what we need to do above: collate all links to all Wikipedia pages and aggregate their counts. We will do this by counting the frequency of all links in the Map phase as we process all Wikipedia articles during the Map phase. We will emit link URIs as keys, and counts as values. Hadoop will then collate the data and group it under the key we specified during the Map phase, and then call our Reduce logic during the Reduce phase. Our reduce logic will sum up all counts for each URI, giving us inlink counts for that URI.

I have attached the source code, but want to point out some of the relevant parts in the mapper and reducer. I use JDOM and some dependent libs (Xerces, Jaxen) to parse the XML. For more fundamental questions about Hadoop MapReduce classes, I recommend the basic Hadoop Tutorial.

The Mapper class is declared as follows: it takes in a line number and text field, and for each URI it finds, maps counts of outlinks to a count associated with the URI. This is done by implementing the Mapper interface with the appropriate input parameters — line number = LongWritable, line text = Text) and output (URI = Text, count = LongWritable). Hadoop needs to provide alternative implementations to standard Java classes like Long and String because it implements a custom serialization format.

```
public class LinkMapper extends MapReduceBase implements Mapper {
 ... }
```

The Mapping function is pretty simple. It is passed a line count and String representing each line in the freebase-articles.tsv. It parses the XML, and pushes all outlink counts to the passed in Collector. All mapping functionality updates the Collector with key/value pairs, and updates the Reporter with status, counter value updates, or errors.

```
/** * calculates frequency of targets in processed XML */ @Overri
de public void map(LongWritable lineCt, Text line, OutputCollecto
r collector, Reporter reporter) throws IOException {      String[
] parts = line.toString().split("\t");     // the xml should be i
n the 4th split.     if(parts.length == 4) {        String xml =
parts[3];          // fix up xml so that JDOM can parse it.
  xml = xml.replaceFirst("xmlns:xhtml=\" \"", "xmlns:xhtml=\"foo\
"");        String xmlProtocol = "";         String fullXml = xml
Protocol+xml;          for (Map.Entry entry : parseXml(fullXml).e
ntrySet()) {             collector.collect(entry.getKey(), entry
.getValue());          }      } }
```

The parseXML() function called by map() uses XPath to locate all outlinks and puts them into a map, so instances of the same outlink are detected and the count is incremented. The wikipedize() method creates a wikpedia specific URI from the target text.

```
/** * gets a map of counts by link for a given XML fragment. * @p
aram parser * @param xmlFragment * @return map of URIs mapped to
counts * @throws IOException */ protected  Map parseXml(String xm
lFragment) throws IOException { Map parsedLinks = new HashMap();
        logger.debug("parsing "+xmlFragment);      SAXBuilder bui
lder = new SAXBuilder();          // command line should offer UR
Is or file names         try {           StringReader reader = ne
w StringReader(xmlFragment);          Document doc = builder.buil
d(reader);           XPath xp = XPath.newInstance("//target");
        List targets = xp.selectNodes(doc);           for(Object
obj : targets) {              Element target = (Element)obj;
         Text wikiLink = new Text(wikipedize(target.getText()));
            LongWritable newCount  = ONE;              LongWrita
ble count = parsedLinks.get(wikiLink);             if(count != nu
ll) {               newCount = new LongWritable(count.get()+1);
           }              logger.debug("putting key="+wikiLink.t
oString()+", value="+newCount.get());             parsedLinks.put
(wikiLink,newCount);         }      }     catch (JDOMException e)
 {       logger.error(xmlFragment + " is not well-formed.");
  logger.error(e.getMessage());     }     catch (IOException e) {
       logger.error("Could not check " + xmlFragment);       logg
er.error(" because " + e.getMessage());     }      return parsedL
inks;  }
```

The Reducer is also very simple. It sums the counts for each key and updates the collector with the sum.

```
public class URIReducer extends MapReduceBase     implements Redu
cer {       @Override      public void reduce(Text key, Iterator
values,             OutputCollector collector,            Report
er reporter)         throws IOException {              long tota
lSum = 0;              while(values.hasNext()) {             Lon
gWritable value = values.next();                  totalSum += val
ue.get();              }              collector.collect(key,new L
ongWritable(totalSum));     } }
```

Note that since this reducer is simply an aggregator, I can also use it as a Combiner, which is in effect a local reduce for a slave node prior to the overall aggregation and sorting that happens in order to run the global reduce. The Combiner is set (just like the Mapper and Reducer) in the JobConf class used to run the job. This code is contained in the
WexLinkDriver class definition in the attached code:

```
 ... JobClient client = new JobClient(); JobConf conf = new JobCo
nf(getConf(), com.evri.infocloud.wexlinks.WexLinkDriver.class);
conf.setJobName("test job"); conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(LongWritable.class);  conf.setMapperClas
s(LinkMapper.class); conf.setCombinerClass(URIReducer.class); con
f.setReducerClass(URIReducer.class);  conf.setInputFormat(TextInp
utFormat.class); conf.setOutputFormat(TextOutputFormat.class);  F
ileInputFormat.setInputPaths(conf,new Path(input[1])); FileOutput
Format.setOutputPath(conf,new Path(output[1]));  client.setConf(c
onf); JobClient.runJob(conf); ...
```

Building the Jar File
The source code needs to be bundled into a jar and uploaded to the master node. Jars that the source code rely on also need to be bundled into that jar.

The jar contains the following directories:

- com.* (source code path)
- lib (all jars the source depends on)
- META-INF (contains MANIFEST.MF, specifying Main-Class to run)

I created a build.xml to copy the lib directory to the correct location where it can be jarred up relative to the classes and the META-INF directory:

```
    <property name="classes.dir" value="target/classes"/>      <t
arget name="init">         <mkdir dir="${classes.dir}"/>
<copy includeemptydirs="false" todir="${classes.dir}">
  <fileset dir="src" excludes="**/*.launch, **/*.java"/>
</copy>     <copy includeemptydirs="false" todir="${classes.dir}/
lib">             <fileset dir="lib" includes="**/*.jar"/>
  </copy>     </target>     <target depends="init" name="build">
        <echo message="${ant.project.name}: ${ant.file}"/>
  <javac debug="true" debuglevel="${debuglevel}" destdir="target/
classes"             source="${source}" target="${target}">
       <src path="src"/>             <classpath refid="fb-wex-inl
ink-aggregator.classpath"/>         </javac>     </target>
```

This creates the right structure under target/classes, which gets jarred up in a later step:

```
     <target depends="build, test" name="hadoop-jar">           <
jar destfile="target/inlink-aggregator.jar"             manifest=
"src/META-INF/MANIFEST.MF"              basedir="${classes.dir}"
```

```
>       </jar>     </target>
```

See the attached build.xml for the entire listing.

Getting The Dataset

The Dataset, as mentioned above, is available as an Elastic Block Store — available to be mounted by an EC2 instance. Mounting an EBS volume is as easy as (a) connecting the EBS volume to an EC2 instance (in this case, the master) as a virtual device on that EC2 instance, and (b) mounting that instance, i.e. running mount {device} {directory}.

I connect the EBS volume to my master EC2 instance via elastic fox, although, as mentioned above, it is possible to do this with the EC2 command line tools. Via ElasticFox, I

1. select the Volumes and Snapshots tab.
2. click on the '+' button to add a new volume.
3. input the size of the volume I need (Freebase-WEX needs 66GB).
4. Input the snapshot ID of the volume I want to attach to (the snapshot ID of Freebase WEX is snap-1781757e)

Now that I have the volume I need to attach it to the master.

1. right click on the volume.
2. select 'Attach this volume' menu option.
3. input the instance id of the master node.

Once I've attached the volume to the instance, I ssh into the instance and mount the device. If I've attached the EBS volume as /dev/sdb, I mount like this:

```
mount /dev/sdb freebase-wex
```

Uploading the Data to the Cluster

Now that I've mounted the Freebase WEX data, I am going to use {mount dir}/rawd/articles/freebase-wex-2009-01-12-articles.tsv as the input data for my hadoop job. In order to do this I need to get articles.tsv onto the cluster HDFS.

Since the Cloudera scripts have taken care of HDFS Namenode initialization, all I need to do is create an input directory and upload the tsv file to it:

```
hadoop fs -mkdir input
hadoop fs -copyFromLocal freebase-wex/rawd/articles/freebase-wex-
2009-01-12-articles.tsv input
```

Running the Job

In order to run the job, I need to upload the jar to the master node. I copy the jar to the master node:

```
hadoop-ec2 push my-cluster-name inlink-aggregator.jar
```

and then run the jar as follows

```
hadoop jar inlink-aggregator.jar input output
```

Note that the output directory cannot exist when the job is run.

Results

The results of the MapReduce job are in your output dir, which is in HDFS. Copy those results down to the master node:

```
hadoop fs -copyToLocal {HDFS output dir} {desired output dir on
local FS}
```

and (if you want) save them to S3 using the /usr/bgin/s3cmd.rb file that comes on the Cloudera AMI. In order to use s3cmd, you need to set the AWS_ACCESS_KEY and AWS_SECRET_ACCESS_KEY environment variables.

```
s3cmd put --force output/* s3://hadoop-freebase-wex-demo/output
```

From S3 you can download them to your machine using the S3 sync tool of your choice. I use S3Organizer to do this.

NOTE: you can upload directly to S3 and skip the above. In the cluster master:

(1) edit /etc/hadoop/conf/hadoop-site.xml, and add the following:

```
<property>  <name>fs.s3n.awsAccessKeyId</name>  <value>YOUR-AWS-A
CCESS-KEY-ID</value> </property>  <property>  <name>fs.s3n.awsSec
retAccessKey</name>  <value>YOUR-AWS-SECRET-ACCESS-KEY</value> </
property>
```

(2) restart NameNode so that HDFS picks up the changes
```
service hadoop-namenode restart
```

(3) run
```
hadoop jar inlink-aggregator.jar input s3n://infocloud-subject-ou
tput-1/run1
```

As a final node, I noticed that unless I specified the S3 output dir as a subdir of a top level S3 bucket, I recieved the following exception:

```
java.lang.IllegalArgumentException: Path must be absolute:
s3n://infocloud-subject-output-1
```

In subsequent MapReduce jobs, this logic also applies to the input dir (if you are using an S3 bucket as input). Details on this issue can be found here.

Debugging/Analyzing the Results

While the job ran fine, examination of the results shows that the counts are suspiciously low. For example Barack Obama only has 57 inlinks. At this point I'm going to refactor the mapping code and start tracking map state using Counters.

I have a suspicion that (a) the tsv format may not be consistent, or (b) the XML I'm parsing is not valid. In order to test those theories I'm going to instrument the code with counters.

In order to use Counters I need to create an enum that contains the states I want to represent.

```
public enum MapState { ERROR_INVALID_XML, ERROR_IO_EXCEPTION, XML
_PARSED, ERROR_RUNTIME_EXCEPTION, };
```

I first instrumented the existing code and ran it to see if I was getting exceptions or just not parsing XML.

I used the Reporter interface to instrument the code, using the enums I had created:

```
reporter.incrCounter(MapState.XML_PARSED, 1);
```

I ran, and saw no exceptions, but the XML_PARSED count was 574251, which seems very low, considering that the input file has 4183153 records.

I then modified the mapping code to check every line part to see if it was the XML, and re-ran:

```
@Override public void map(LongWritable lineCt, Text line, OutputC
ollector collector, Reporter reporter) throws IOException {
String[] parts = line.toString().split("\t");      String xml = n
ull;      try {      // the xml is not located in the same split
from row to row, find first one.            for(int i = 0; i < pa
rts.length;i++) {              if(parts[i].startsWith(ARTICLES_XM
LNS)) {                   xml = parts[i];                   break
;              }          }           if(xml != null) {
     // fix up the XML so that JDOM can parse it.              xm
l = xml.replaceFirst("xmlns:xhtml=\" \"", "xmlns:xhtml=\"foo\"");
              String xmlProtocol = "";              String fullXm
l = xmlProtocol+xml;             for (Map.Entry entry : parseXml(
fullXml).entrySet()) {                   collector.collect(entry.ge
tKey(), entry.getValue());              }           reporter.incr
Counter(MapState.XML_PARSED, 1);           }      } catch (JDOMEx
ception e) {           logger.error(xml+ " is not well-formed.");
           logger.error(e.getMessage());           reporter.incrC
ounter(MapState.ERROR_INVALID_XML, 1);       } catch (IOException
 e) {           logger.error("Could not check " + xml);
 logger.error(" because " + e.getMessage());           reporter.i
ncrCounter(MapState.ERROR_IO_EXCEPTION, 1);      } catch (Runtime
Exception ex) {          logger.error("RUNTIME EXCEPTION: "+ex.ge
tMessage());          logger.error("xml: "+xml);          reporte
r.incrCounter(MapState.ERROR_RUNTIME_EXCEPTION, 1);           thr
ow ex;     } }
```

When re-running this job, I noticed it ran a lot slower. When I checked my counter values via the web UI after the job completed the XML_PARSED counter was set to 4,180,888, much more in line with the line count . So apparently I _was_ missing a lot of data. A quick check on Barack Obama reveals that his page inlink count went from 77 to 11851, which is more in line with what I would expect.

The Web UI also provides access to the logs, provided that (a) you open up the TaskTracker port on the slave security group, and (b) you use the Amazon public DNS name instead of the Amazon private DNS name.

Summary

The ease with which I was able to get a job running on EC2 using the Cloudera scripts and AMIS helped me resolve focus on the logic related issues of my MapReduce code instead of thrashing around getting the cluster functional.

The other nice thing about the Cloudera scripts is that I didn't have to spend time learning the (182!) EC2 scripts. The hadoop-ec2 script, in combination with ElasticFox for finding and mounting an EBS volume, was all I really needed. If it isn't clear by now, I am really, really happy that Cloudera has taken the time to do this, and do it very well.

Bloopers

There were a couple of potholes along the way that I want to mention should anyone else fall into them.

(1) File permissions on my EC2 private key needed to be set to 0600, otherwise the hadoop-ec2 launch-cluster script would fail trying to set up passwordless ssh between master and slaves.

(2) Logging. I was logging the entire blurb of XML prior to refactoring the code to

look for the XML in 'non standard' locations. When I started up the job again, it would run just fine, and then I would start seeing:

```
/04/29 17:50:50 INFO mapred.JobClient: Task Id : attempt_20090429
1618_0002_m_000041_0, Status : FAILED java.io.IOException: Task p
rocess exit with nonzero status of 1.         at org.apache.hadoo
p.mapred.TaskRunner.runChild(TaskRunner.java:462)         at org.
apache.hadoop.mapred.TaskRunner.run(TaskRunner.java:403)  09/04/2
9 17:50:50 WARN mapred.JobClient: Error reading task outputhttp:/
/domU-12-31-39-02-CA-45.compute-1.internal:50060/tasklog?plaintex
t=true&taskid=attempt_200904291618_0002_m_000041_0&filter=stdout
09/04/29 17:50:50 WARN mapred.JobClient: Error reading task outpu
thttp://domU-12-31-39-02-CA-45.compute-1.internal:50060/tasklog?p
laintext=true&taskid=attempt_200904291618_0002_m_000041_0&filter=s
tderr
```

Eventually, this would happen across a majority of slaves and the master would consider the job 'failed'. Of course, I could not actually see what went wrong with the task because the JobClient couldn't read the error output. I did manage to look at the logs for several of the task trackers that failed, and decided to stop logging the XML. The issue resolved itself at that point.

(3) Debugging: I would recommend always running a MapReduce job over a small set of data on a local single node install prior to pushing it out to the cluster. While this didn't reproduce the logging issue above, it did catch some late night null pointer exception goofs. Since debugging failures over any distributed environment is a detective game at best, you want to show up with something that has been reasonably tested before throwing it across an N node cluster.

Updated about 10 months ago · View Original Post · Report Note