CSC561 NoSQL Databases
Lab 4: Column Family Database / Hbase Part 2

HBase and Big Data:

Let's look at *real* big Data – not just a large dataset by importing part of Wikipedia into our HBase wiki table. We will need to script using the HBase APIs in order to stream Wikipedia content into our wiki.

By doing this you will get an idea of how long it actually takes to move BIG data around. There are some performance tricks for making faster import jobs. We can also look at HBase's internals to see how it partitions data into regions, achieving both performance and disaster recovery design goals that are mentioned in our text.

As we saw in part 1 of the lab, individually issuing Put operations with static strings, is inefficient. Fortunately, pasting commands into the shell is not the only way to execute them. When you start the HBase shell from the command line, you can specify the name of a JRuby script to run. HBase will execute that script as though it were entered directly into the shell. The syntax looks like this:

${HBASE_HOME}/bin/hbase shell <your_script> [<optional_arguments> ...]

We are going to use this feature to create a script that will import Wikipedia articles into our wiki table. The WikiMedia Foundation (which oversees Wikipedia, Wictionary, and their other projects) periodically publishes data dumps we can use. These dumps are in the form of enormous XML files.

Building an XML Parser

One technique to parse huge XML infosets is by using the SAX (Simple API for XML) API to create a streaming parser. A streaming parser refers to a programming model in which XML infosets are transmitted and parsed serially at application runtime, often in real time, and often from dynamic sources whose contents are not precisely known beforehand. Moreover, a stream-based parser (as opposed to the DOM (Document Object Model) API) can start generating output immediately, and infoset elements can be discarded and garbage collected immediately after they are used. The caveat when using a streaming parser is that you need to know what processing you want to do before reading the XML document.

A Parser Framework

The basic outline for parsing an XML file in JRuby, record by record, looks like this:

```
import 'javax.xml.stream.XMLStreamConstants'

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

while reader.has_next
        type = reader.next
        if type == XMLStreamConstants::START_ELEMENT
```

```
            tag = reader.local_name
            # do something with tag
        elsif type == XMLStreamConstants::CHARACTERS
            text = reader.text
            # do something with text
        elsif type == XMLStreamConstants::END_ELEMENT
            # same as START_ELEMENT
        end
end
```

What this code does:

First, we produce an XMLStreamReader and connect it to java.lang.System.in, which means it will be reading from standard input.

Next, we set up a while loop, which will continuously pull out tokens from the XML stream until there are none left. Inside the while loop, we process the current token. What to do depends on whether the token is the start of an XML tag (START_ELEMENT), the end of a tag (END_ELEMENT), or the text in between (CHARACTERS).

Incorporating our Parser code into the framework

Now we can combine this basic XML processing framework with our previous exploration of the HTable and Put interfaces to create our import script. Most of it should look familiar, and we'll discuss a few novel parts (denoted with a circled number):

```
require 'time'

import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'javax.xml.stream.XMLStreamConstants'

def jbytes( *args )
args.map { |arg| arg.to_s.to_java_bytes }
end

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

①       document = nil
        buffer = nil
        count = 0

        table = HTable.new( @hbase.configuration, 'wiki' )
```

```
②       table.setAutoFlush( false )

        while reader.has_next
                type = reader.next
③       if type == XMLStreamConstants::START_ELEMENT
                case reader.local_name
                when 'page' then document = {}
                when /title|timestamp|username|comment|text/ then buffer = []
                end
④       elsif type == XMLStreamConstants::CHARACTERS
                buffer << reader.text unless buffer.nil?
⑤       elsif type == XMLStreamConstants::END_ELEMENT
                case reader.local_name
                when /title|timestamp|username|comment|text/
                        document[reader.local_name] = buffer.join
                when 'revision'
                        key = document['title'].to_java_bytes
                        ts = ( Time.parse document['timestamp'] ).to_i
                        p = Put.new( key, ts )
                        p.add( *jbytes( "text", "", document['text'] ) )
                        p.add( *jbytes( "revision", "author", document['username'] ) )
                        p.add( *jbytes( "revision", "comment", document['comment'] ) )
                        table.put( p )
                        count += 1
                        table.flushCommits() if count % 10 == 0
                        if count % 500 == 0
                                puts "#{count} records inserted (#{document['title']})"
                        end
                end
        end
end
table.flushCommits()
exit
```

① The first difference of note is the introduction of a few variables:

- document: Holds the current article and revision data
- buffer: Holds character data for the current field within the document (text, title, author, and so on)
- count: Keeps track of how many articles we've imported so far

② Pay special attention to the use of table.setAutoFlush(false). In HBase, data is *automatically flushed* to disk periodically. This is preferred in most applications. By disabling autoflush in our script, any put operations we execute will be buffered until we call table.flushCommits(). This allows us to batch up writes and execute them when it's convenient for us.

③ Next, let's look at what happens in parsing. If the start tag is a <page>, then reset document to an empty hash. Otherwise, if it's another tag we care about, reset buffer for storing its text.

④ We handle character data by appending it to the buffer.

⑤ For most closing tags, we just stash the buffered contents into the document. If the closing tag is a </revision>, however, we create a new Put instance, fill it with the document's fields, and submit it to the table. After that, we use flushCommits() if we haven't done so in a while, and report progress to standard out (puts).

Adding Compression and Bloom Filters

We're almost ready to run the script; we just have one more bit of housecleaning to do first. The text column family is going to contain big blobs of text content; it would benefit from some compression.

HBase supports two compression algorithms: Gzip (GZ) and Lempel-Ziv-Oberhumer (LZO). The HBase community highly recommends using LZO over Gzip, but we'll use using GZ for the lab because it's already installed. *(The problem with LZO is the implementation's license. While open source, it's not compatible with Apache's licensing philosophy, so LZO is not bundled with HBase. Detailed instructions are available online for installing and configuring LZO support. In practice, if you want high-performance compression, use LZO.)*

A Bloom filter is an effective data structure that efficiently answers the question, "Have I ever seen this thing before?" Originally developed by Burton Howard Bloom in 1970 for use in spell-checking applications, Bloom filters are widely used in data storage applications for determining quickly whether a key exists. HBase supports using Bloom filters to determine whether a particular column exists for a given row key (BLOOMFILTER=>'ROWCOL') or just whether a given row key exists at all (BLOOMFILTER=>'ROW'). Because the number of columns within a column family and the number of rows are both potentially unbounded, Bloom filters offer a fast way of determining whether data exists before incurring an expensive disk read.

Running the Streaming Import Parser

The ruby files for the input parser are available on http://csc570e.uis.edu/csc561 so that you do not have to type the entire script in manually. You should put this script in your /usr/lib/hbase directory

If you have previously shut down your HBase, restart it:

```
cd /usr/lib/hbase/hbase-1.1.3/bin
start-hbase.sh
```

```
student@student-virtual-machine:~$ cd /usr/lib/hbase/hbase-1.1.3/bin
student@student-virtual-machine:/usr/lib/hbase/hbase-1.1.3/bin$ start-hbase.sh
starting master, logging to /usr/lib/hbase/hbase-1.1.3/logs/hbase-student-master
-student-virtual-machine.out
```

Enter the shell in order to enable compression and fast lookups:

disable 'wiki'
alter 'wiki', {NAME => 'text', COMPRESSION => 'GZ', BLOOMFILTER => 'ROW'}
enable 'wiki'

```
student@student-virtual-machine:/usr/lib/hbase/hbase-1.1.3/bin$ hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/hbase/hbase-1.1.3/lib/slf4j-log4j12-1.
7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/lib/hadoop-2.7.2/share/hadoop/common
/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.1.3, r72bc50f5fafeb105b2139e42bbee3d61ca724989, Sat Jan 16 18:29:00 PST
 2016

hbase(main):001:0> disable 'wiki'
0 row(s) in 3.0770 seconds

hbase(main):002:0> alter 'wiki', {NAME=>'text', COMPRESSION => 'GZ', BLOOMFILTER=
>'ROW'}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 2.2960 seconds

hbase(main):003:0> enable 'wiki'
0 row(s) in 1.5180 seconds
```

Exit the shell

Now, we'll run our script.

First, we need to install curl using the following command:

sudo apt-get update - This updates your entire system

sudo apt-get install curl

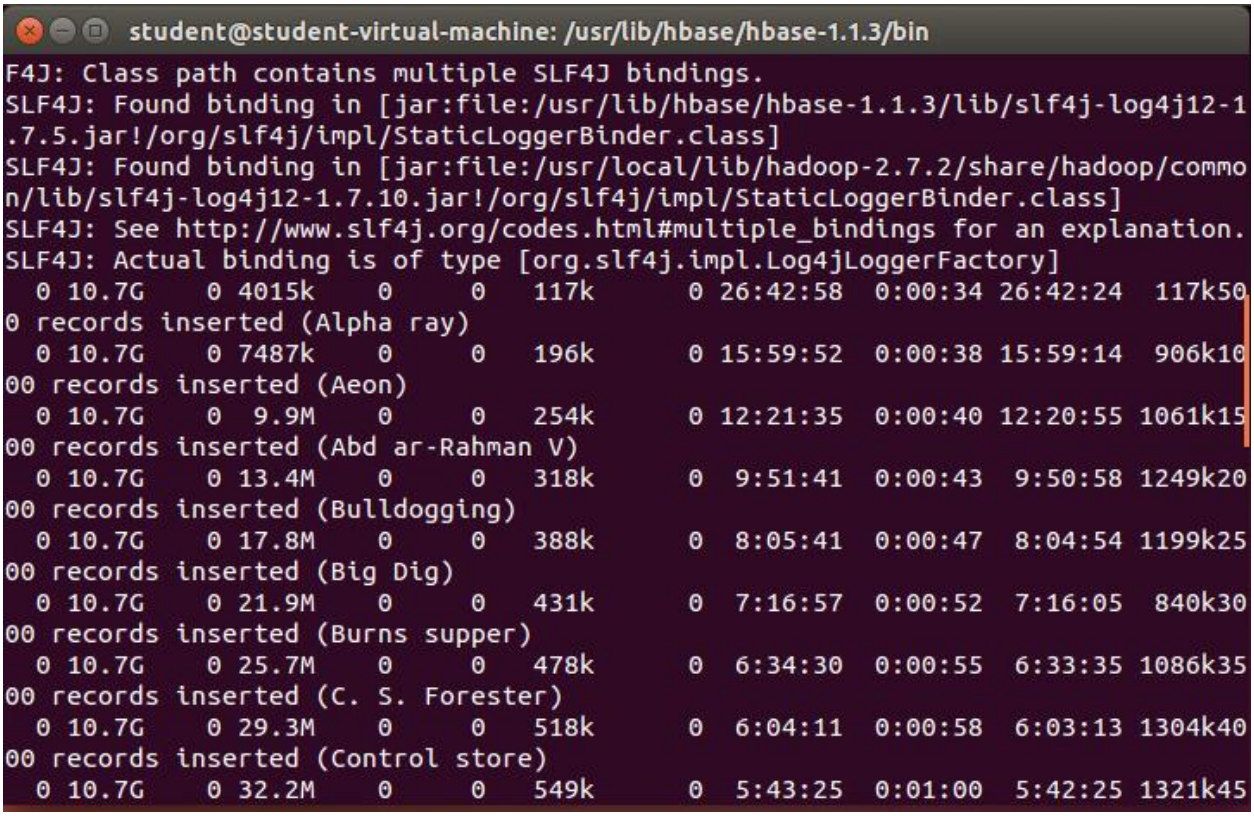
Then download both .rb files from http://csc570e.uis.edu/csc561. Once that is done we will need to move them, so open a new terminal and type the following:

cd ~/Downloads
sudo mv import_wikipedia.rb /usr/lib/hbase/hbase-1.1.3/
sudo mv generate_wiki_links.rb /usr/lib/hbase/hbase-1.1.3/

cd /usr/lib/hbase/hbase-1.1.3/bin

Finally, we can run our script and continue on.
sudo curl https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2 |bzcat| hbase shell import_wikipedia.rb

it should begin streaming input like below:



This script will keep on importing data until you stop it with Ctrl+C. Let it run for 30 seconds-1 minute (enough so that you get into the C's or D's in the entry titles) then stop it. ***Yes, you can run your VM out of memory at this step. If you do, shut it down and redo the lab (since the job is interrupted HBase won't commit this transaction).***

Extracting Information from our Wiki Import

Wiki syntax is filled with links, some of which link internally to other articles and some of which link to external resources. This interlinking contains a wealth of topological data. Our goal is to capture the relationships between articles as directional links, pointing one article to another or receiving a link from another. An internal article link in wikitext looks like this: [[<target name>|<alt text>]], where <target name> is the article to link to, and <alt text> is the alternative text to display (optional).

For example, if the text of the article on Star Wars contains the string "[[Yoda|jedi master]]", we want to store that relationship twice—once as an outgoing link from Star Wars and once as an incoming link to

Yoda. Storing the relationship twice means that it's fast to look up both a page's outgoing links and its incoming links.

To store this additional link data, we'll create a new table. Enter the shell and create a new table:

create 'links', {NAME => 'to', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'},{NAME => 'from', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'}

```
hbase(main):003:0> create 'links',{NAME=>'to', VERSIONS=>1, BLOOMFILTER=>'ROWCOL
'},{NAME=>'from', VERSIONS=>1, BLOOMFILTER=>'ROWCOL'}
```

Constructing the Scanner

With the links table created, we're ready to implement a script that will scan all the rows of the wiki table. Then, for each row, retrieve the wikitext and parse out the links. Finally, for each link found, create incoming and outgoing link table records. The bulk of this script is similar to our last parsing script. Most of the pieces are recycled, I've put notes for the lines that are different.

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'org.apache.hadoop.hbase.client.Scan'
import 'org.apache.hadoop.hbase.util.Bytes'

def jbytes( *args )
        return args.map { |arg| arg.to_s.to_java_bytes }
end
wiki_table = HTable.new( @hbase.configuration, 'wiki' )
links_table = HTable.new( @hbase.configuration, 'links' )
links_table.setAutoFlush( false )

①       scanner = wiki_table.getScanner( Scan.new )
        linkpattern = /\[\[([^\[\]\|\:\#][^\[\]\|:]*)(?:\|([^\[\]\|]+))?\]\]/
        count = 0

        while (result = scanner.next())
②               title = Bytes.toString( result.getRow() )
                text = Bytes.toString( result.getValue( *jbytes( 'text', '' ) ) )
                if text
                        put_to = nil
③                       text.scan(linkpattern) do |target, label|
                                unless put_to
                                        put_to = Put.new( *jbytes( title ) )
                                        put_to.setWriteToWAL( false )
                                end
```

```
                    target.strip!
                    target.capitalize!
                    label = '' unless label
                    label.strip!
                    put_to.add( *jbytes( "to", target, label ) )
                    put_from = Put.new( *jbytes( target ) )
                    put_from.add( *jbytes( "from", title, label ) )
                    put_from.setWriteToWAL( false )
④                   links_table.put( put_from )
                end
⑤       links_table.put( put_to ) if put_to
        links_table.flushCommits()
        count += 1
        puts "#{count} pages processed (#{title})" if count % 500 == 0
        end
links_table.flushCommits()
exit
```

① First, we grab a Scan object, which we'll use to scan through the wiki table.

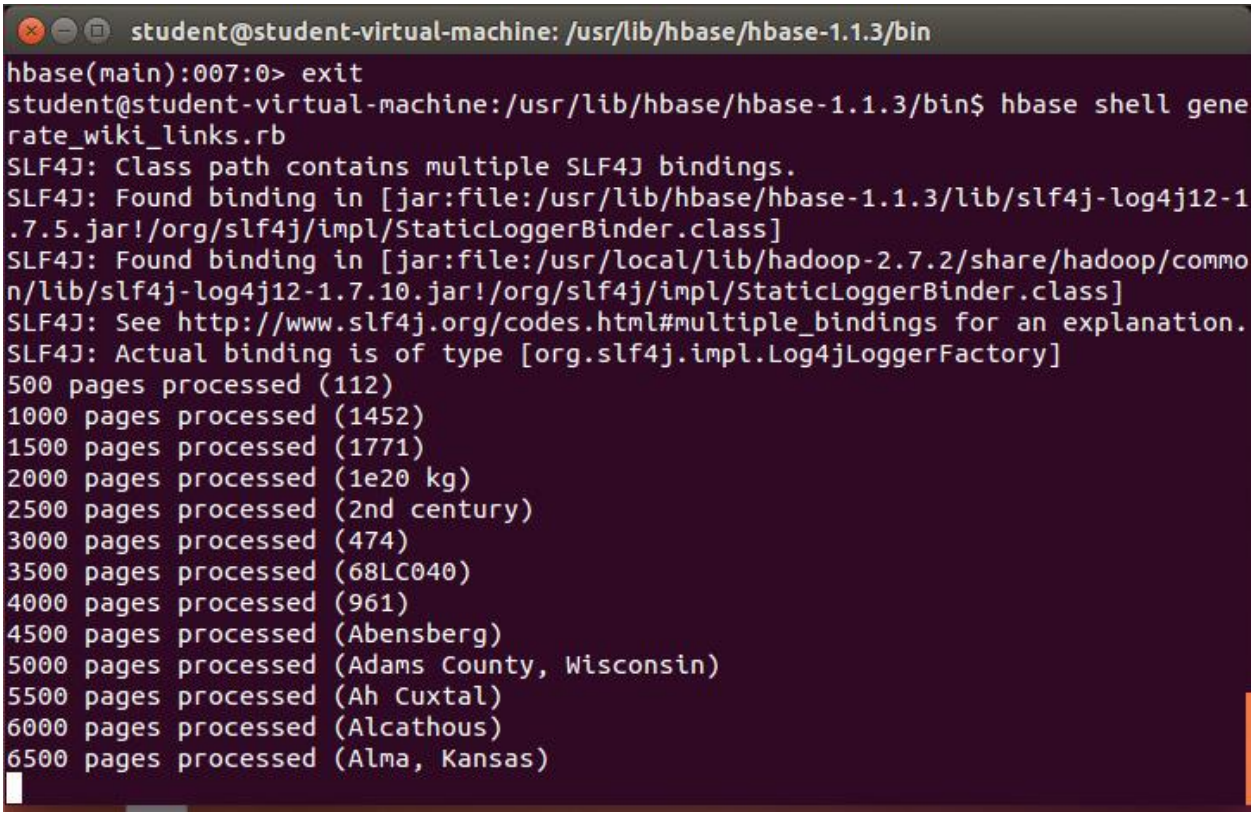
② Extracting row and column data requires some byte wrangling, this is typical when parsing text entries

.

③ Each time the linkpattern appears in the page text, we extract the target article and text of the link and then use those values to add to our Put instances.

④ Using setWriteToWAL(false) for these puts is a judgment call. Since this exercise is for educational purposes and since we could simply rerun the script if anything went wrong, we'll take the speed bonus and accept our fate should the node fail. In a production system, you would insert code for exceptions and node failure

⑤ Finally, we tell the table to execute our accumulated Put operations. It's possible (though unlikely) for an article to contain no links at all, which is the reason for the "if put_to" clause.



As with the previous script, you can let it run as long as you like, even to completion. If you want to stop it, press CTRL+C. ***Yes, you can run your VM out of memory at this step too. If you do, shut it down and redo this portion of the lab. Your data from your wiki import should still be there when you come back.***

Examining our Newly Created Wiki Information

We just created a scanner programmatically to perform a sophisticated task. Now we'll use the shell's scan command to simply dump part of a table's contents to the console. For each link the script finds in a text: blob, it will indiscriminately create both to and from entries in the links table. To see the kinds of links being created, re-enter your shell enter:

scan 'links', STARTROW=>'Admiral Ackbar', ENDROW=>'Admiralty'

```
hbase(main):005:0> scan 'links', STARTROW=>'Admiral Ackbar', ENDROW=>'Admiralty'
ROW                  COLUMN+CELL
 Admiral Canaris     column=to:Wilhelm canaris, timestamp=1460675620474, value=
 Admiral Doenitz     column=to:Karl d\xC3\xB6nitz, timestamp=1460675620474, valu
                     e=
 Admiral Kuznetsov   column=to:Admiral kuznetsov class aircraft carrier, timesta
                     mp=1460675620475, value=''Admiral Kuznetsov'' class aircraf
                     t carriers
 Admiral Kuznetsov   column=to:Nikolay gerasimovich kuznetsov, timestamp=1460675
                     620475, value=
 Admiral Kuznetsov   column=to:Russian aircraft carrier admiral kuznetsov, times
                     tamp=1460675620475, value=Russian aircraft carrier ''Admira
                     l Kuznetsov''
 Admiral farragut aca column=from:Alan Shepard, timestamp=1460675629556, value=
 demy
 Admiral horthy      column=from:1919, timestamp=1460675562546, value=
 Admiral kolhammer   column=from:2021, timestamp=1460675584745, value=Admiral Ph
                     illip Kolhammer
 Admiral kuznetsov cl column=from:Admiral Kuznetsov, timestamp=1460675620475, val
 ass aircraft carrier ue=''Admiral Kuznetsov'' class aircraft carriers
```

Or, if you wanted to just examine the links for a single article, enter:

get ‘links’, ‘admiralty’

```
hbase(main):006:0> get 'links', 'Admiralty'
COLUMN                CELL
 from:1673            timestamp=1460675551377, value=Lord High Admiral
 from:1903            timestamp=1460675560985, value=
 from:Aberdare        timestamp=1460675609329, value=
 from:Admiralty law   timestamp=1460675620489, value=
 from:Advocate        timestamp=1460675622045, value=
 from:Air Ministry    timestamp=1460675626252, value=
 from:Airspeed Ltd.   timestamp=1460675626943, value=
 from:Wes             timestamp=1460675635265, value=
 t Dunbartonshire
 from:Anchor          timestamp=1460675663815, value=
 from:Anne, Queen of  timestamp=1460675686841, value=
 Great Britain
 from:Anson County, N timestamp=1460675688703, value=First Lord of the Admiralty
 orth Carolina
 from:Aston Martin    timestamp=1460675784158, value=
 to:Admiral           timestamp=1460675620480, value=
 to:Admiralty adminis timestamp=1460675620480, value=
 tration
 to:Admiralty arch    timestamp=1460675620480, value=
 to:Admiralty board ( timestamp=1460675620480, value=Admiralty Board
 united kingdom)
 to:Admiralty chart   timestamp=1460675620480, value=
 to:Admiralty house,  timestamp=1460675620480, value=Admiralty House
 london
 to:Air ministry      timestamp=1460675620480, value=
 to:Alexander pope    timestamp=1460675620480, value=
 to:Baroque           timestamp=1460675620480, value=
```


In the wiki table, the rows are very regular with respect to columns. Each row has text:, revision:author, and revision:comment columns. In our new links table there is no such regularity. Each row may have one column or hundreds. And the variety of column names is as diverse as the row keys themselves (titles of Wikipedia articles). HBase is a so-called sparse data store for exactly this reason. To find out just how many rows are now in your table, you can use the count command. (This can take a while to run to completion or lock up the machine if you imported a lot of data!)

hbase> count 'wiki', INTERVAL => 100000, CACHE => 10000

```
hbase(main):007:0> count 'wiki', INTERVAL=>100000, CACHE=>10000
88540 row(s) in 32.6100 seconds

=> 88540
hbase(main):008:0>
```

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

WORK TO SUBMIT

Expanding on the idea of data import, let's build a database containing nutrition facts. Download the MyPyramid Raw Food Data set from http://csc570e.uis.edu/csc561/Food_Display_Table.xml to your VM to work on it. (From the Terminal run the command "wget http://csc570e.uis.edu/csc561/Food_Display_Table.xml")

This data consists of many pairs of <Food_Display_Row> tags. Inside these, each row has a <Food_Code> (integer value), <Display_Name> (string), and other facts about the food in appropriately named tags.

1. Create a new table called foods with a single column family called fact to store the facts. Enable the table to do compression (GZ) and fast lookups (BLOOMFILTER).
2. Create a new JRuby script for importing the food data. Use the SAX parsing style we used earlier for the Wikipedia import script and tailor it for the food data.
3. Pipe the food data into your import script on the command line to populate the table.
4. Finally, using the HBase shell, query the foods table for information about your favorite foods.

Your submission to GitHub will include 2 files:

1. The base.txt file should have your create foods table definition
2. The lab4/lab4b.rb should have the JRuby script for importing the food data (#2 above)

The Food_Display_Table.xml is already in your Github lab4b folder and the continuous integration server is already configured to pipe its data into the import script you are to create for step 2.