Extracted from:

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Databases in Seven Weeks

Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond and Jim R. Wilson

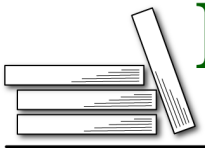
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond
and Jim R. Wilson

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series Editor: Bruce A. Tate
Copy Editor: Nancy Rapoport
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-253-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2018

Day 2: Working with Big Data

With Day 1's table creation and manipulation under our belts, it's time to start adding some serious data to our wiki table. Today, you'll script against the HBase APIs, ultimately streaming Wikipedia content right into our wiki! Along the way, you'll pick up some performance tricks for making faster import jobs. Finally, you'll poke around in HBase's internals to see how it partitions data into regions, achieving a series of both performance and disaster recovery goals.

Importing Data, Invoking Scripts

One common problem people face when trying a new database system is how to migrate data into it. Handcrafting `Put` operations with static strings, as you did in Day 1, is all well and good, but you can do better.

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> ...]
```

Because we're interested specifically in "Big Data," let's create a script for importing Wikipedia articles into our wiki table. The WikiMedia Foundation, which oversees Wikipedia, Wictionary, and other projects, periodically publishes data dumps we can use. These dumps are in the form of enormous XML files. Here's an example record from the English Wikipedia:

```
<page>
  <title>Anarchism</title>
  <id>12</id>
  <revision>
    <id>408067712</id>
    <timestamp>2011-01-15T19:28:25Z</timestamp>
    <contributor>
      <username>RepublicanJacobite</username>
      <id>5223685</id>
    </contributor>
    <comment>Undid revision 408057615 by [[Special:Contributions...</comment>
    <text xml:space="preserve">{{Redirect|Anarchist|the fictional character|
...
[[bat-smg:Anarkėzmos]]
    </text>
  </revision>
</page>
```

Because we have such incredible foresight, the individual items in these XML files contain all the information we've already accounted for in our schema: title (row key), text, timestamp, and author. We ought to be able to write a script to import revisions without too much trouble.

Streaming XML

First things first: We'll need to parse the huge XML files in a streaming fashion, so let's start with that. The basic outline for parsing an XML file in JRuby, record by record, looks like this:

hbase/basic_xml_parsing.rb

```
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
```

Breaking this down, there are a few parts worth mentioning. First, we produce an XMLStreamReader and wire it up 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 happens then depends on whether the token is the start of an XML tag, the end of a tag, or the text in between.

Streaming Wikipedia

Now we can combine this basic XML processing framework with our previous exploration of the HTable and Put interfaces you explored previously. Here is the resultant script. Most of it should look familiar, and we will discuss a few novel parts.

hbase/import_from_wikipedia.rb

```
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)

document = nil
buffer = nil
count = 0

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

while reader.has_next
  type = reader.next
```

```
  if type == XMLStreamConstants::START_ELEMENT

    case reader.local_name
    when 'page' then document = {}
    when /title|timestamp|username|comment|text/ then buffer = []
    end

  elsif type == XMLStreamConstants::CHARACTERS

    buffer << reader.text unless buffer.nil?

  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
```

A few things to note in the preceding snippet:

- Several new variables were introduced:
  - 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 you've imported so far.
- 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 you execute will be buffered until you call table.flushCommits(). This allows you to batch writes together and execute them when it's convenient for you.

- If the start tag is a <page>, then reset document to an empty hash. Otherwise, if it's another tag you care about, reset buffer for storing its text.
- We handle character data by appending it to the buffer.
- For most closing tags, you just stash the buffered contents into the document. If the closing tag is a </revision>, however, you create a new Put instance, fill it with the document's fields, and submit it to the table. After that, you use flushCommits() if you haven't done so in a while and report progress to stdout.

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. Reading those values will take much longer than values like Hello world or Welcome to the wiki! from Day 1. HBase enables us to compress that data to speed up reads:

```
hbase> alter 'wiki', {NAME=>'text', COMPRESSION=>'GZ', BLOOMFILTER=>'ROW'}
0 row(s) in 0.0510 seconds
```

HBase supports two compression algorithms: Gzip (GZ) and Lempel-Ziv-Oberhumer (LZO). The HBase community highly recommends using LZO over Gzip pretty much unilaterally, but here we're using Gzip. Why is that?

The problem with LZO for our purposes here is the implementation's license. While open source, LZO is not compatible with Apache's licensing philosophy, so LZO can't be bundled with HBase. Detailed instructions are available online for installing and configuring LZO support. If you want high-performance compression, use LZO in your own projects.

A *Bloom filter* is a really cool data structure that efficiently answers the question "Have I ever seen this thing before?" and is used to prevent expensive queries that are doomed to fail (that is, to return no results). Originally developed by Burton Howard Bloom in 1970 for use in spell-checking applications, Bloom filters have become popular 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'). 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.

How Do Bloom Filters Work?

Without going too deep into implementation details, a Bloom filter manages a statically sized array of bits initially set to 0. Each time a new blob of data is presented to the filter, some of the bits are flipped to 1. Determining which bits to flip depends on generating a hash from the data and turning that hash into a set of bit positions.

Later, to test whether the filter has been presented with a particular blob in the past, the filter figures out which bits would have to be 1 and checks them. If any are 0, then the filter can unequivocally say "no." If all of the bits are 1, then it reports "yes." Chances are it has been presented with that blob before, but false positives are increasingly likely as more blobs are entered.

This is the trade-off of using a Bloom filter as opposed to a simple hash. A hash will never produce a false positive, but the space needed to store that data is unbounded. Bloom filters use a constant amount of space but will occasionally produce false positives at a predictable rate based on saturation. False positives aren't a huge deal, though; they just mean that the filter *says* a value is likely to be there, but you will eventually find out that it isn't.

Engage!

Now that we've dissected the script a bit and added some powerful capabilities to our table, we're ready to kick off the script. Remember that these files are enormous, so downloading and unzipping them is pretty much out of the question. So, what are we going to do?

Fortunately, through the magic of *nix pipes, we can download, extract, and feed the XML into the script all at once. The command looks like this:

```
$ curl https://url-for-the-data-dump.com | bzcat | \
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Note that you should replace the preceding dummy URL with the URL of a WikiMedia Foundation dump file of some kind.[2] You should use [project]-latest-pages-articles.xml.bz2 for either the English Wikipedia (~12.7 GB)[3] or the English Wiktionary (~566 MB).[4] These files contain all of the most recent revisions of pages in the Main namespace. That is, they omit user pages, discussion pages, and so on.

Plug in the URL and run it! You should start seeing output like this shortly:

2. https://dumps.wikimedia.org/enwiki/latest
3. https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
4. https://dumps.wikimedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2

```
500 records inserted (Ashmore and Cartier Islands)
1000 records inserted (Annealing)
1500 records inserted (Ajanta Caves)
```

The script will happily chug along as long as you let it or until it encounters an error, but you'll probably want to shut it off after a while. When you're ready to kill the script, press Ctrl+C. For now, though, let's leave it running so we can take a peek under the hood and learn about how HBase achieves its horizontal scalability.