

Extracted from:

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

This PDF file contains pages extracted from *Seven Databases in Seven Weeks*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
and Jim R. Wilson

Edited by Jacquelyn Carter



Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

4.3 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, we’ll script against the HBase APIs, ultimately streaming Wikipedia content right into our wiki! Along the way, we’ll pick up some performance tricks for making faster import jobs. Finally, we’ll poke around in HBase’s internals to see how it partitions data into regions, achieving 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, like we did in Day 1, is all well and good, but we 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> ...]
```

Since 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, Wiktionary, 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:Anarkézm0s]]
  </text>
</revision>
</page>
```

Because we were so smart, this contains all the information we've already accounted for in our schema: title (row key), text, timestamp, and author. So, 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 (SAX) 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'll 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, 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. Here's the resultant script. Most of it should look familiar, and we'll 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)
```

```
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`).

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. Let's enable compression and fast lookups:

```

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 GZ.

The problem with LZO is the implementation's license. While open source, it's 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, get LZO.

A Bloom filter is a really cool 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 have become popular in data storage applications for determining quickly whether a key exists. If you're unfamiliar with Bloom filters, they're explained briefly in [How Do Bloom Filters Work?, on page 10](#).

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.

Engage!

Now 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 <dump_url> | bzcata | \
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Note that you should replace <dump_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 (~6GB)³ or the English Wiktionary (~185MB).⁴ These files contain all 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 see output like this (eventually):

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

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.

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

It'll 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.

Introduction to Regions and Monitoring Disk Usage

In HBase, rows are kept in order, sorted by the row key. A region is a chunk of rows, identified by the starting key (inclusive) and ending key (exclusive). Regions never overlap, and each is assigned to a specific region server in the cluster. In our simplistic stand-alone server, there is only one region server, which will always be responsible for all regions. A fully distributed cluster would consist of many region servers.

So, let's take a look at your HBase server's disk usage, which will give us insight into how the data is laid out. You can inspect HBase's disk usage by opening a command prompt to the `hbase.rootdir` location you specified earlier and executing the `du` command. `du` is a standard *nix command-line utility that tells you how much space is used by a directory and its children, recursively. The `-h` option tells `du` to report numbers in human-readable form.

Here's what ours looked like after about 12,000 pages had been inserted and the import was still running:

```

$ du -h
231M  ./logs/localhost.localdomain,38556,1300092965081
231M  ./logs
4.0K  ./META./1028785192/info
12K   ./META./1028785192/.oldlogs
28K   ./META./1028785192
32K   ./META.
12K   ./-ROOT-/70236052/info
12K   ./-ROOT-/70236052/.oldlogs
36K   ./-ROOT-/70236052
40K   ./-ROOT-
72M   ./wiki/517496fecabb7d16af7573fc37257905/text
1.7M  ./wiki/517496fecabb7d16af7573fc37257905/revision
61M   ./wiki/517496fecabb7d16af7573fc37257905/.tmp
12K   ./wiki/517496fecabb7d16af7573fc37257905/.oldlogs
134M  ./wiki/517496fecabb7d16af7573fc37257905
134M  ./wiki
4.0K  ./oldlogs
365M  .

```

This output tells us a lot about how much space HBase is using and how it's allocated. The lines starting with `/wiki` describe the space usage for the wiki table. The long-named subdirectory `517496fecabb7d16af7573fc37257905` represents an individual region (the only region so far). Under that, the directories `/text` and `/revision` correspond to the text and revision column families, respectively. Finally, the last line sums up all these values, telling us that HBase is using 365MB of disk space.

One more thing. The first two lines at the top of output, starting with `./logs`, show us the space used by the write-ahead log (WAL) files. HBase uses write-ahead logging to provide protection against node failures. This is a fairly typical disaster recovery technique. For instance, write-ahead logging in file systems is called *journaling*. In HBase, logs are appended to the WAL before any edit operations (put and increment) are persisted to disk.

For performance reasons, edits are not necessarily written to disk immediately. The system does much better when I/O is buffered and written to disk in chunks. If the *region server* responsible for the affected region were to crash during this limbo period, HBase would use the WAL to determine which operations were successful and take corrective action.

Writing to the WAL is optional and enabled by default. Edit classes like `Put` and `Increment` have a setter method called `setWriteToWAL()` that can be used to exclude the operation from being written to the WAL. Generally you'll want to keep the default option, but in some instances it might make sense to change it. For example, if you're running an import job that you can rerun

any time, like our Wikipedia import script, you might want to take the performance benefit of disabling WAL writes over the disaster recovery protection.

Regional Interrogation

If you let the script run long enough, you'll see HBase split the table into multiple regions. Here's our du output again, after about 150,000 pages have been added:

```
$ du -h
40K      ./logs/localhost.localdomain,55922,1300094776865
44K      ./logs
24K      ./META./1028785192/info
4.0K    ./META./1028785192/recovered.edits
4.0K    ./META./1028785192/.tmp
12K     ./META./1028785192/.oldlogs
56K     ./META./1028785192
60K     ./META.
4.0K    ./corrupt
12K     ./-ROOT-/70236052/info
4.0K    ./-ROOT-/70236052/recovered.edits
4.0K    ./-ROOT-/70236052/.tmp
12K     ./-ROOT-/70236052/.oldlogs
44K     ./-ROOT-/70236052
48K     ./-ROOT-
138M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/text
5.8M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/revision
4.0K    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/.tmp
144M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458
149M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/text
6.5M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/revision
4.0K    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/.tmp
155M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a
145M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/text
6.3M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/revision
4.0K    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/.tmp
151M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987
135M    ./wiki/a79c0f6896c005711cf6a4448775a33b/text
6.0M    ./wiki/a79c0f6896c005711cf6a4448775a33b/revision
4.0K    ./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp
141M    ./wiki/a79c0f6896c005711cf6a4448775a33b
591M    ./wiki
4.0K    ./oldlogs
591M    .
```

The biggest change is that the old region (517496fecabb7d16af7573fc37257905) is now gone, replaced by four new ones. In our stand-alone server, all the regions are served by our singular server, but in a distributed environment, these would be parceled out to the various region servers.

This raises a few questions, such as “How do the region servers know which regions they’re responsible for serving?” and “How can you find which region (and, by extension, which region server) is serving a given row?”

If we drop back into the HBase shell, we can query the `.META.` table to find out more about the current regions. `.META.` is a special table whose sole purpose is to keep track of all the user tables and which region servers are responsible for serving the regions of those tables.

```
hbase> scan '.META.', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

Even for a small number of regions, you should get a lot of output. Here’s a fragment of ours, formatted and truncated for readability:

ROW

```
wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.
```

COLUMN+CELL

```
column=info:server, timestamp=1300333136393, value=localhost.localdomain:3555
column=info:regioninfo, timestamp=1300099734090, value=REGION => {
  NAME => 'wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.',
  STARTKEY => '',
  ENDKEY => 'Demographics of Macedonia',
  ENCODED => a79c0f6896c005711cf6a4448775a33b,
  TABLE => {{...}}
```

ROW

```
wiki,Demographics of Macedonia,1300099733696.0a25ac7e5d0be211b9e890e83e24e458.
```

COLUMN+CELL

```
column=info:server, timestamp=1300333136402, value=localhost.localdomain:35552
column=info:regioninfo, timestamp=1300099734011, value=REGION => {
  NAME => 'wiki,Demographics of Macedonia,1300099733696.0a25...e458.',
  STARTKEY => 'Demographics of Macedonia',
  ENDKEY => 'June 30',
  ENCODED => 0a25ac7e5d0be211b9e890e83e24e458,
  TABLE => {{...}}
```

Both of the regions listed previously are served by the same server, `localhost.localdomain:35552`. The first region starts at the empty string row (`"`) and ends with `'Demographics of Macedonia'`. The second region starts at `'Demographics of Macedonia'` and goes to `'June 30'`.

`STARTKEY` is inclusive, while `ENDKEY` is exclusive. So, if we were looking for the `'Demographics of Macedonia'` row, we’d find it in the second region.

Since rows are kept in sorted order, we can use the information stored in `.META.` to look up the region and server where any given row should be found. But where is the `.META.` table stored?

It turns out that the `.META.` table is split into regions and served by region servers just like any other table would be. To find out which servers have which parts of the `.META.` table, we have to scan `-ROOT-`.

```
hbase> scan '-ROOT-', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }

ROW
.META.,,1
COLUMN+CELL
column=info:server, timestamp=1300333135782, value=localhost.localdomain:35552
column=info:regioninfo, timestamp=1300092965825, value=REGION => {
  NAME => '.META.,,1',
  STARTKEY => '',
  ENDKEY => '',
  ENCODED => 1028785192,
  TABLE => {{...}}
```

The assignment of regions to region servers, including `.META.` regions, is handled by the *master* node, often referred to as `HBaseMaster`. The master server can also be a region server, performing both duties simultaneously.

When a region server fails, the master server steps in and reassigns responsibility for regions previously assigned to the failed node. The new stewards of those regions would look to the WAL to see what, if any, recovery steps are needed. If the master server fails, responsibility defers to any of the other region servers that step up to become the master.

Where's My TABLE Schema?

The TABLE schema has been removed from the example output of regioninfo scans. This reduces clutter, and we'll be talking about performance-tuning options later. If you're dying to see the schema definition for a table, use the describe command. Here's an example:

```
hbase> describe 'wiki'  
hbase> describe '.META.'  
hbase> describe '-ROOT-'
```