Hbase

Apache HBase is made for big jobs. If your data is not measured by many gigabytes, you probably need a smaller tool.

HBase, at first glance, looks a lot like a relational database—but isn’t

The most challenging part of learning HBase isn’t the technology; it’s that many of the words used in HBase are deceptively familiar.

For example, HBase stores data in buckets it calls *tables*, which contain *cells* that appear at the intersection of *rows* and *columns*.

In HBase, tables don’t behave like relations, rows don’t act like records, and columns are completely variable (not enforced by a schema description).

Schema design is still important, since it informs the performance characteristics of the system.

So, why would you use this database?

Aside from scalability, there are a few reasons.

First, HBase has some built-in features that other databases lack, such as versioning, compression, garbage collection (for expired data), and in-memory tables.

Having these features available right out of the box means less code that you have to write when your requirements demand them.

HBase also makes strong consistency guarantees, making it easier to transition from relational databases.

For all of these reasons, HBase really shines as the cornerstone of an online analytics processing system.

While individual operations may be slower than equivalent operations in other databases, scanning through enormous datasets is something HBase takes to with relish.

So, for genuinely big queries, HBase often outpaces other databases.

This also explains why HBase is often employed at big companies to back logging and search systems.

Introducing HBase   
HBase is a *column-oriented (Wide Column)* database that prides itself on consistency and scaling out.

It is based on BigTable, a high-performance, proprietary database developed by Google and described in the 2006 white paper “Bigtable: A Distributed Storage System for Structured Data.”

On the architecture front, HBase is designed to be fault tolerant.

Hardware failures may be uncommon for individual machines, but in a large cluster, node failure is the norm.

By using write-ahead logging (WAL) and distributed configuration, HBase can quickly recover from individual server failures.

Additionally, HBase lives in an ecosystem that has its own complementary benefits.

HBase is built on Hadoop—a sturdy, scalable computing platform that provides a distributed file system and mapreduce capabilities.

Wherever you find HBase, you’ll find Hadoop and other infrastructural components that you can lever in your own applications.

**Apache Hadoop** is an [open-source](http://en.wikipedia.org/wiki/Open_source) [software framework](http://en.wikipedia.org/wiki/Software_framework) for [distributed storage](http://en.wikipedia.org/wiki/Clustered_file_system) and [distributed processing](http://en.wikipedia.org/wiki/Distributed_processing) of [Big Data](http://en.wikipedia.org/wiki/Big_Data) on [clusters](http://en.wikipedia.org/wiki/Computer_cluster) of [commodity hardware](http://en.wikipedia.org/wiki/Commodity_hardware). Its [Hadoop Distributed File System (HDFS)](http://en.wikipedia.org/wiki/Apache_Hadoop#HDFS) splits files into large blocks (default 64MB or 128MB) and distributes the blocks amongst the nodes in the cluster. For processing the data, the Hadoop [Map/Reduce](http://en.wikipedia.org/wiki/MapReduce) ships code (specifically [Jar files](http://en.wikipedia.org/wiki/Jar_files)) to the nodes that have the required data, and the nodes then process the data in parallel. – from wiki

It is actively used and developed by a number of high-profile companies for their “Big Data” problems.

Notably, Facebook chose HBase as a principal component of its new messaging infrastructure announced in November 2010

Twitter uses HBase extensively, ranging from data generation (for applications such as people search) to storing monitoring/performance data also eBay, Meetup, Ning, Yahoo!, and many others.

With all of this activity, new versions of HBase are coming out at a fairly rapid clip.

The current stable version is 1.4.3

CRUD and Table Administration

We’ll get a local instance of HBase running in stand-alone mode, and then we’ll use

the HBase shell to create and alter tables and to insert and modify data using basic commands.

I have built a machine on the cloud with HBase so today we will ‘putty in’ to that

Along the way, we’ll uncover some HBase architectural concepts, such as the relationship between rows, column families, columns, and values in a table.

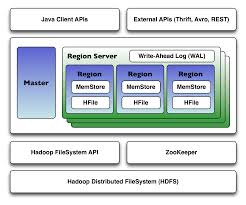
A fully operational, production-quality HBase cluster should really consist of no fewer than five nodes, or so goes the conventional wisdom.

HBase supports three running modes:

• Stand-alone mode is a single machine acting alone.

• Pseudodistributed mode is a single node pretending to be a cluster.

• Fully distributed mode is a cluster of nodes working together.



We’ll be running HBase in stand-alone mode.

Open Putty and add the following details:

Hostname: ubuntu@xx.xx.xx.xx

Click SSH/Auth and in the private key file download the pkk file from the lab folder for today (NoSQL16.pkk) and save it on C:/ and browse to it.

Click Open

When you are logged on I want you to change to be a different user using (your firstinitial and surname)

sudo useradd <tnumber>

sudo su <tnumber>

whoami

*( I will do this*

*To start HBase, open a terminal (command prompt) and run this command:*

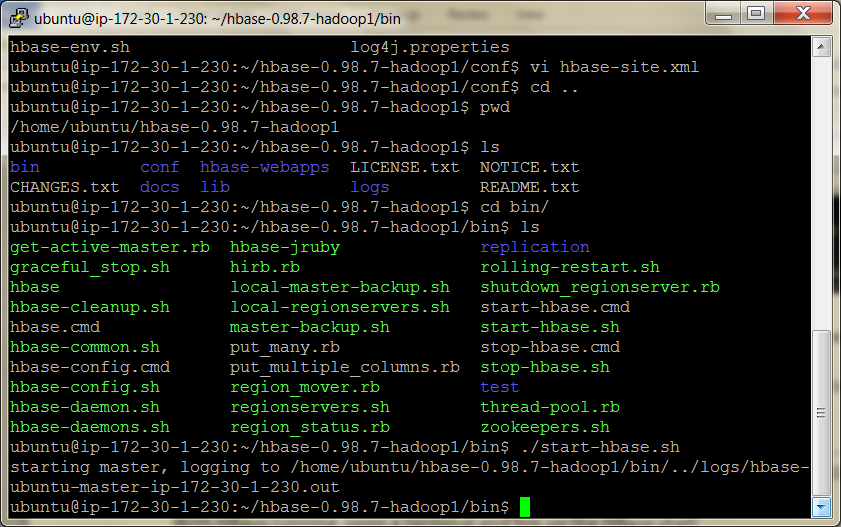
*cd /usr/lib/hbase/bin/*

*sudo ./start-hbase.sh*

*To shut down HBase, use thesudo ./stop-hbase.sh command in the same directory.*

*I will do this)*

*Kill process on 2181 – sudo netstat –tupln*



The HBase Shell

The HBase shell is a JRuby-based command-line program you can use to interact with HBase.

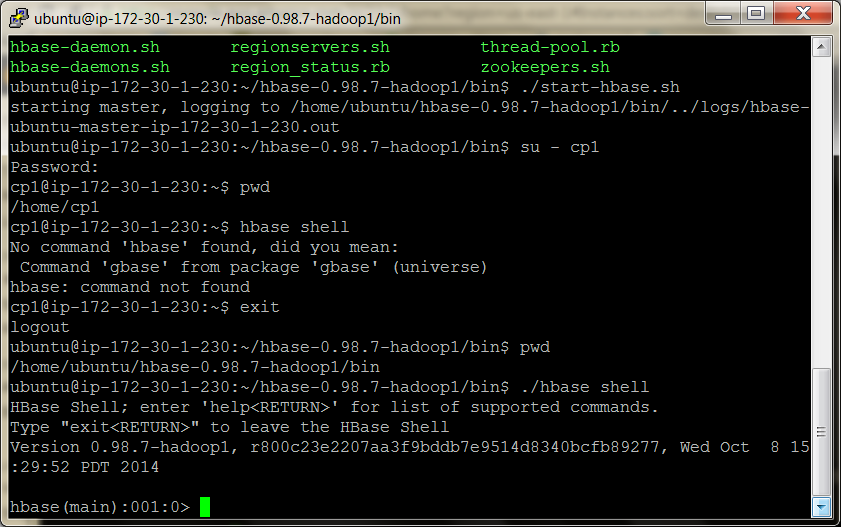
In the shell, you can add and remove tables, alter table schema, add or delete data, and do a bunch of other tasks.

Later we’ll explore other means of connecting to HBase, but for now the shell will be our home.

With HBase running, open a terminal and fire up the HBase shell with the command:

hbase shell

To confirm that it’s working properly, try asking it for version information.



**hbase> version**

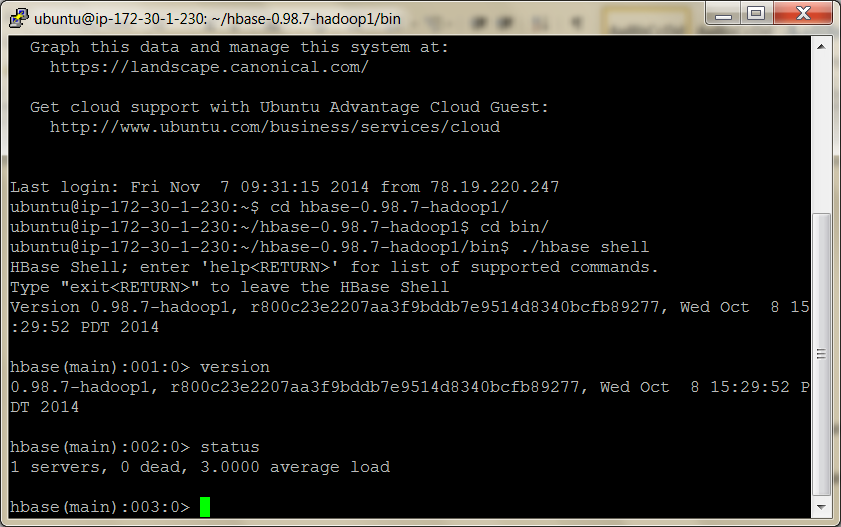
0.98.7, …, Wed Oct 8 15:29:52 PDT 2014

Next, execute the status command to see how your HBase server is holding up.

**hbase> status**

1 servers, 0 dead, 2.0000 average load

 (note if it doesn’t work try sudo netstat -nutlp and sudo kill the process using port 2181)



Creating a Table

A map is a key-value pair, like a hash in Ruby or a hashmap in Java.

A table in HBase is basically a big map. Well, more accurately, it’s a map of maps

In an HBase table, keys are arbitrary strings that each map to a *row* of data.

A row is itself a map, in which keys are called *columns* and values are uninterpreted arrays of bytes.

Columns are grouped into *column families*, so a column’s full name consists of two parts: the column family name and the *column qualifier*.

Often these are concatenated together using a colon (for example, 'family:qualifier').

To illustrate these concepts, take a look at Figure 13, *HBase tables consist of rows, keys, column families, columns, and values*, on page 98.

In this figure, we have a hypothetical table with two column families: color and shape.

The table has two rows—denoted by dashed boxes—identified by their row keys: first and second.

Looking at just the first row, we see that it has three columns in the color column family (with qualifiers red, blue, and yellow) and one column in the shape column family (square).

The combination of row key and column name (including both family and qualifier) creates an address for locating data.

In this example, the tuple first/color:red points us to the value '#F00'.

Now let’s take what we’ve learned about table structure to make a wiki!

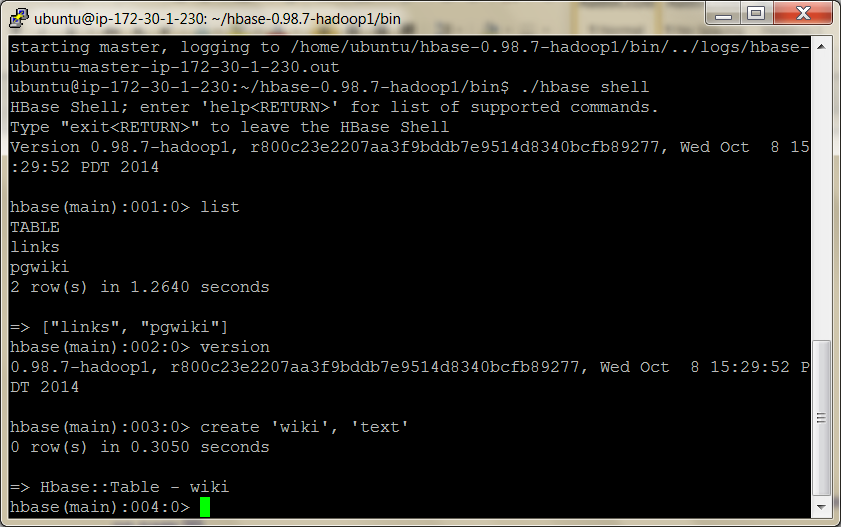
A wiki contains pages, each of which has a unique title string and contains some article text.

Use the create command to make our wiki table:

Use your initials in front of any tables e.g. below I used ‘pgwiki’

**hbase> create 'wiki1611', 'text'**

0 row(s) in 1.2160 seconds



Here, we’re creating a table called wiki with a single column family called text.

The table is currently empty; it has no rows and thus no columns.

Unlike a relational database, in HBase a column is specific to the row that contains it.

When we start adding rows, we’ll add columns to store data at the same time.

Visualizing our table architecture, we arrive at something like Figure 14, *The wiki table has one column family*, on page 99.

By our own convention, we expect each row to have exactly one column within the text family, qualified by the empty string (''””).

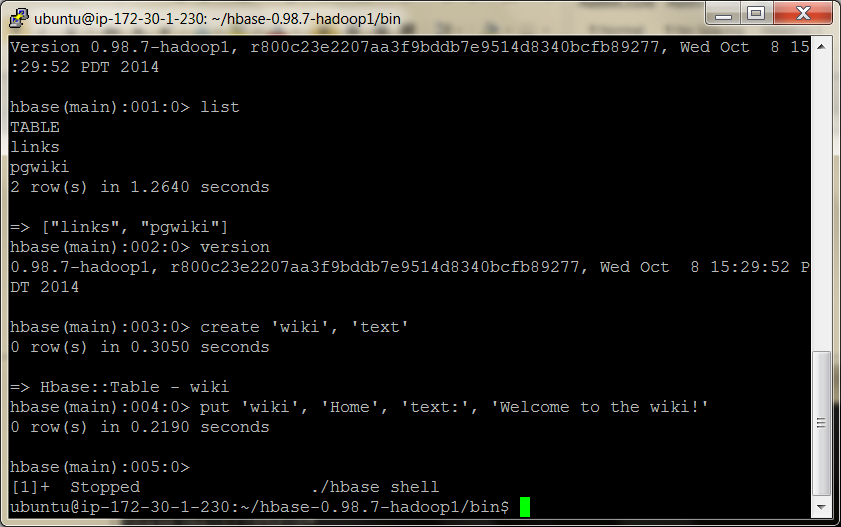
So, the full column name containing the text of a page will be 'text:'.

Of course, for our wiki table to be useful, it’s going to need content. Let’s add some!

Inserting, Updating, and Retrieving Data

Our wiki needs a Home page. To add data to an HBase table, use the put command:

**hbase> put 'wiki1611', 'Home', 'text:', 'Welcome to the wiki!'**



This command inserts a new row into the wiki table with the key 'Home', adding

'Welcome to the wiki!' to the column called 'text:'.

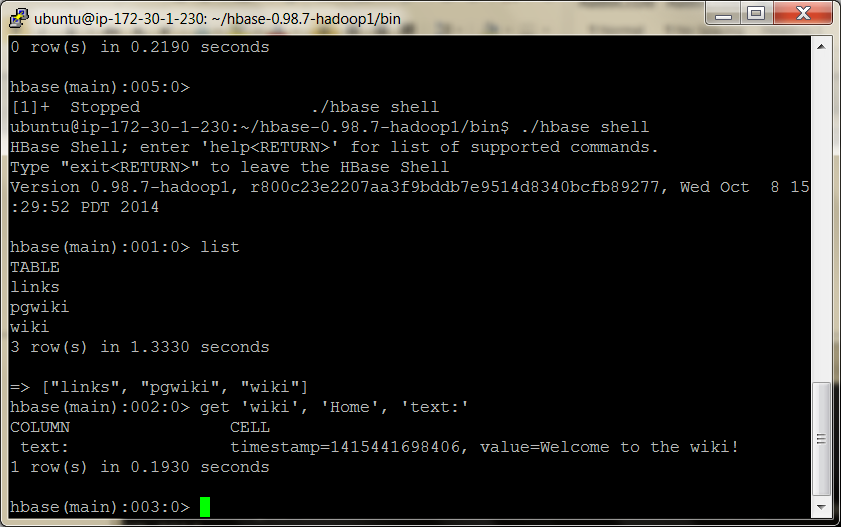
We can query the data for the 'Home' row using get, which requires two parameters: the table name and the row key.

You can optionally specify a list of columns to return.

**hbase> get 'wiki1611', 'Home', 'text:'**

COLUMN CELL

text: timestamp=1295774833226, value=Welcome to the wiki!  
1 row(s) in 0.0590 seconds



Notice the timestamp field in the output. HBase stores an integer timestamp for all data values, representing time in milliseconds since the epoch (00:00:00 UTC on January 1, 1970).

When a new value is written to the same cell, the old value hangs around, indexed by its timestamp.

This is a pretty awesome feature.

Most databases require you to specifically handle historical data yourself, but in HBase, versioning is baked right in!

Put and Get

The put and get commands allow you to specify a timestamp explicitly.

If using milliseconds since the epoch doesn’t strike your fancy, you can specify another integer value of your choice.

If you don’t specify a timestamp, HBase will use the current time when inserting, and it will return the most recent version when reading.

Altering Tables

So far, our wiki schema has pages with titles, text, and an integrated version history but nothing else.

Let’s expand our requirements to include the following:

• In our wiki, a page is uniquely identified by its title.

• A page can have unlimited revisions.

• A revision is identified by its timestamp.

• A revision contains text and optionally a commit comment.

• A revision was made by an author, identified by name.

Visually, our requirements can be sketched, like in Figure 15, *Requirements for a wiki page (including time dimension)*, on page 102.

In this abstract representation of our requirements for a page, we see that each revision has an author, a commit comment, some article text, and a timestamp.

The title of a page is not part of a revision, because it’s the identifier we use to denote revisions belonging to the same page.

Mapping our vision to an HBase table takes a somewhat different form, as illustrated in Figure 16, *Updated wiki table architecture (time dimension not shown)*, on page 102.

Our wiki table uses the title as the row key and will group other page data into two column families called text and revision.

The text column family is the same as before; we expect each row to have exactly one column, qualified by the empty string (''), to hold the article contents.

The job of the revision column family is to hold other revision-specific data, such as the author and commit comment.

Defaults

We created the wiki table with no special options, so all the HBase default values were used.

One such default value is to keep only three VERSIONS of column values, so let’s increase that.

To make schema changes, first we have to take the table offline with the disable command.

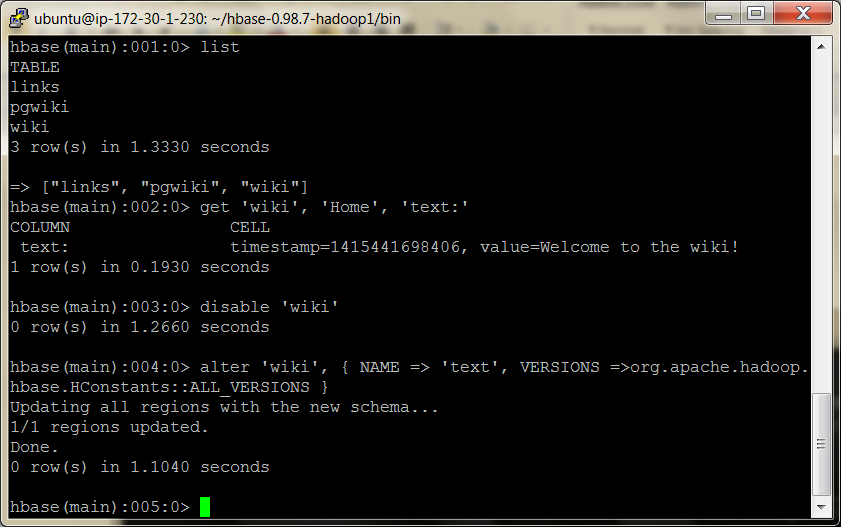
**hbase> disable 'wiki1611'**

0 row(s) in 1.0930 seconds

Now we can modify column family characteristics using the alter command.

**hbase> alter 'wiki1611', { NAME => 'text', VERSIONS =>org.apache.hadoop.hbase.HConstants::ALL\_VERSIONS }**

0 row(s) in 0.0430 seconds



Here, we’re instructing HBase to alter the text column family’s VERSIONS attribute.

Altering a Table

Operations that alter column family characteristics can be very expensive because HBase has to create a new column family with the chosen specifications and then copy all the data over.

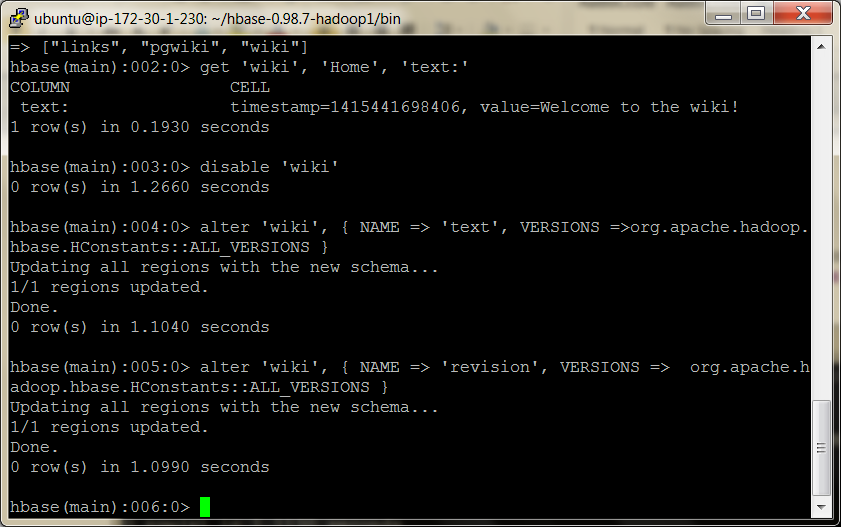
In a production system, this may incur significant downtime

For this reason, settling on column family options up front is a good thing.

With the wiki table still disabled, let’s add the revision column family, again using the alter command:

**hbase> alter 'wiki1611', { NAME => 'revision', VERSIONS =>** **org.apache.hadoop.hbase.HConstants::ALL\_VERSIONS }**

0 row(s) in 0.0660 seconds



Just as before, with the text family, we’re only adding a revision *column family* to the table schema, not individual *columns*.

Though we expect each row to eventually contain a revision:author and revision:comment, it’s up to the client to honor this expectation; it’s not written into any formal schema.

If someone wants to add a revision:foo for a page, HBase won’t stop them.

With these additions in place, let’s reenable our wiki:

**hbase> enable 'wiki1611'**

0 row(s) in 0.0550 seconds

Now that our wiki table has been modified to support our growing requirements list, we can start adding data to columns in the revision column family.

Adding Data Programmatically

As we’ve seen, the HBase shell is great for tasks such as manipulating tables.

Sadly, the shell’s data insertion support isn’t the best.

The put command only allows setting one column value at a time, and in our newly updated schema, we need to add multiple column values simultaneously so they all share the same timestamp.

We’re going to need to start scripting.

The following script can be executed directly in the HBase shell, since the shell is also a JRuby interpreter.

When run, it adds a new version of the text for the Home page, setting the author and comment fields at the same time.

JRuby runs on the Java virtual machine (JVM), giving it access to the HBase Java code.

These examples will not work with non-JVM Ruby.

hbase/put\_multiple\_columns.rb

import *'org.apache.hadoop.hbase.client.HTable'*

import *'org.apache.hadoop.hbase.client.Put'*

**def** jbytes( \*args )

args.map { |arg| arg.to\_s.to\_java\_bytes }

**end**

table = HTable.new( @hbase.configuration, *"wiki511"* )

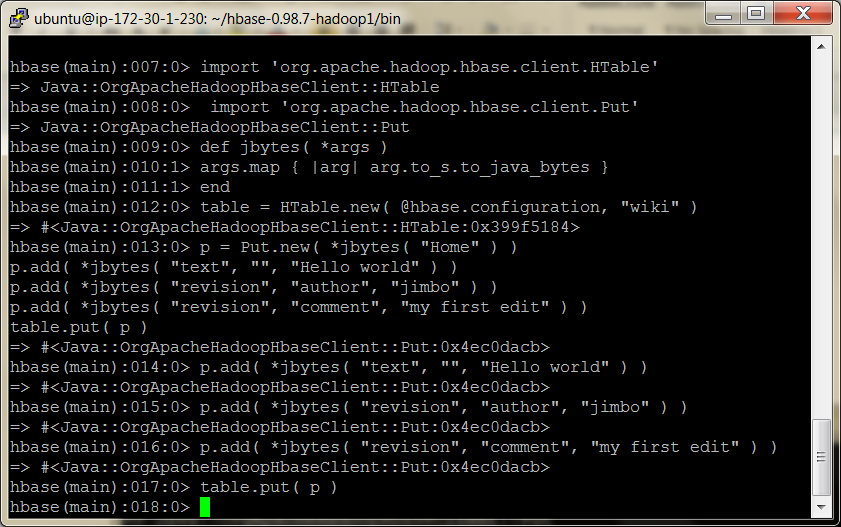
p = Put.new( \*jbytes( *"Home"* ) )

p.add( \*jbytes( *"text"*, *""*, *"Hello world"* ) )

p.add( \*jbytes( *"revision"*, *"author"*, *"peter given"* ) )

p.add( \*jbytes( *"revision"*, *"comment"*, *"my first edit"* ) )

table.put( p )



The import lines bring references to useful HBase classes into the shell.

This saves us from having to write out the full namespace later.

Next, the jbytes() function takes any number of arguments and returns an array converted to Java byte arrays, as the HBase API methods demand.

After that, we create a local variable (table) pointing to our wiki table, using the @hbase administration object for configuration information.

Next we stage a commit operation by creating and preparing a new instance of Put, which takes the row to be modified.

In this case, we’re sticking with the Home page we’ve been working with thus far.

Finally, we add() properties to our Put instance and then call on the table object to execute the put operation we’ve prepared.

The add() method has several forms; in our case, we used the three-argument version: add(column\_family, column\_qualifier, value).

Why Column Families?

You may be tempted to build your whole structure without column families; why not store all of a row’s data in a single column family?

That solution would be simpler to implement.

But there are downsides to avoiding column families, namely, missing out on fine-grained performance tuning.

Each column family’s performance options are configured independently.

These settings affect things such as read and write speed and disk space consumption.

All operations in HBase are atomic at the *row level*.

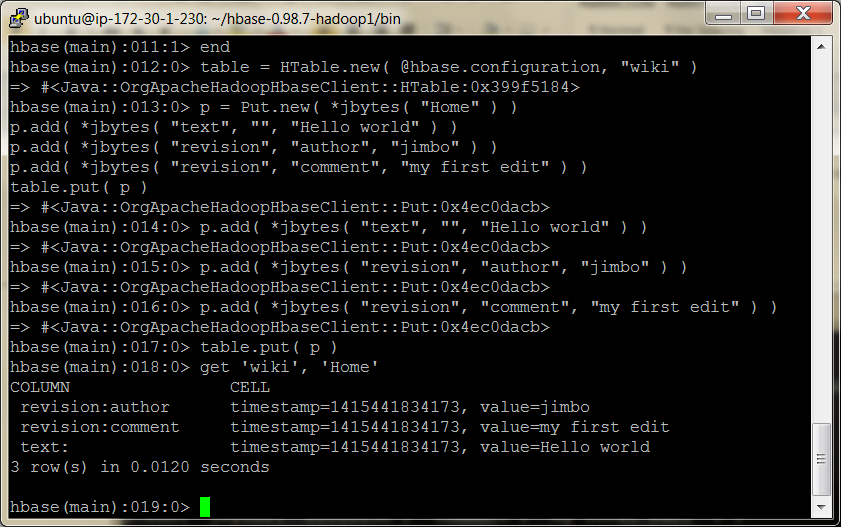
No matter how many columns are affected, the operation will have a consistent view of the particular row being accessed or modified.

This design decision helps clients reason intelligently about the data.

Our put operation affects several columns and doesn’t specify a timestamp, so all column values will have the same timestamp (the current time in milliseconds).

Let’s verify by invoking get.

**hbase> get 'wiki1611', 'Home'**



As you can see, each column value listed previously has the same timestamp.

Designing an HBase schema means making choices about column family options and, just as important, our semantic interpretation of features like timestamps and row keys.

Importing Data, Invoking Scripts

One common problem people face when trying a new database system is how to migrate data into it.

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, 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 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)

document = nil  
buffer = nil  
count = 0

table = HTable.new( @hbase.configuration, *'wiki1611'* )

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

(see book for an explanation of the code)

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.

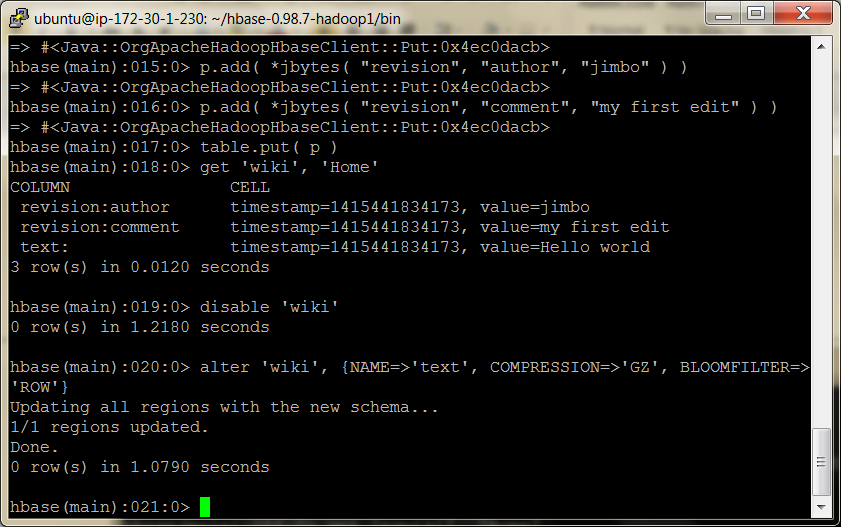
(Note the ruby file is in the bin folder)

hbase>disable ‘wiki1611’

Let’s enable compression and fast lookups:

**hbase> alter 'wiki1611', {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.

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.

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.

(Note enable ‘wiki’)

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> | bzcat | \  
${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.

You should use [project]-latest-pages-articles.xml.bz2 for either the English Wikipedia (~11GB)3 or the English Wiktionary (~485MB).

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! (note I had to type in the –k as pasting it won’t work in mobaXterm, worked ok on pasting in putty)

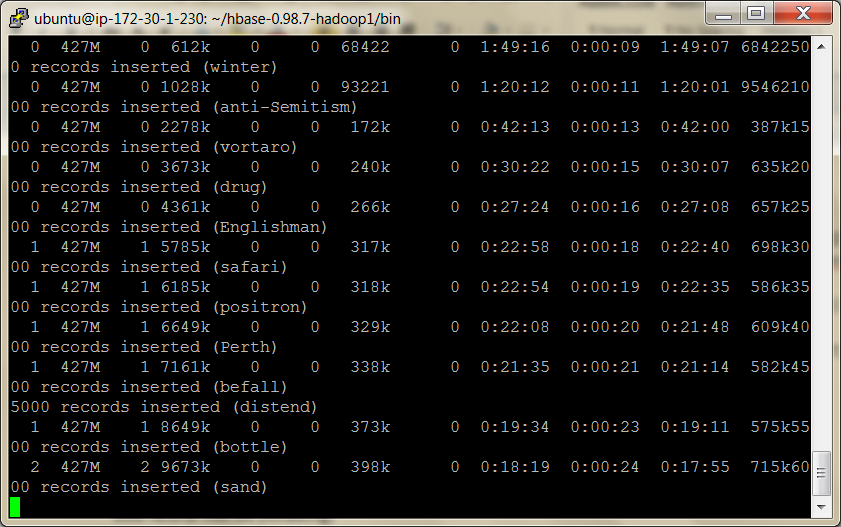
curl –k https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2 | bzcat | hbase shell ./import\_from\_wikipedia.rb

or try this one which should be smaller (MBs rather than GBs)

curl –k https://dumps.wikimedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2 | bzcat | hbase shell ./import\_from\_wikipedia.rb

You should see output like this (eventually):

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



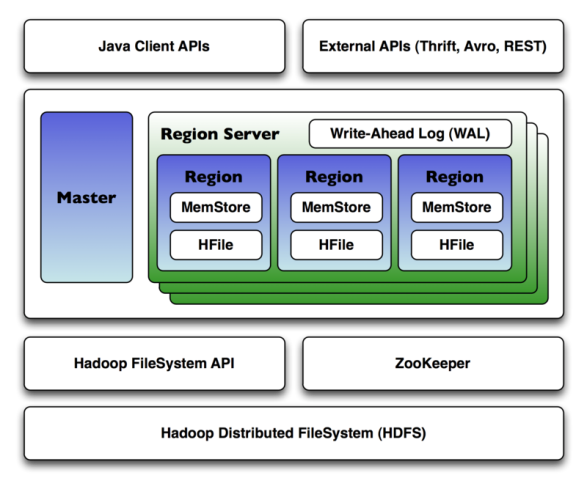
HERE 5/11/19

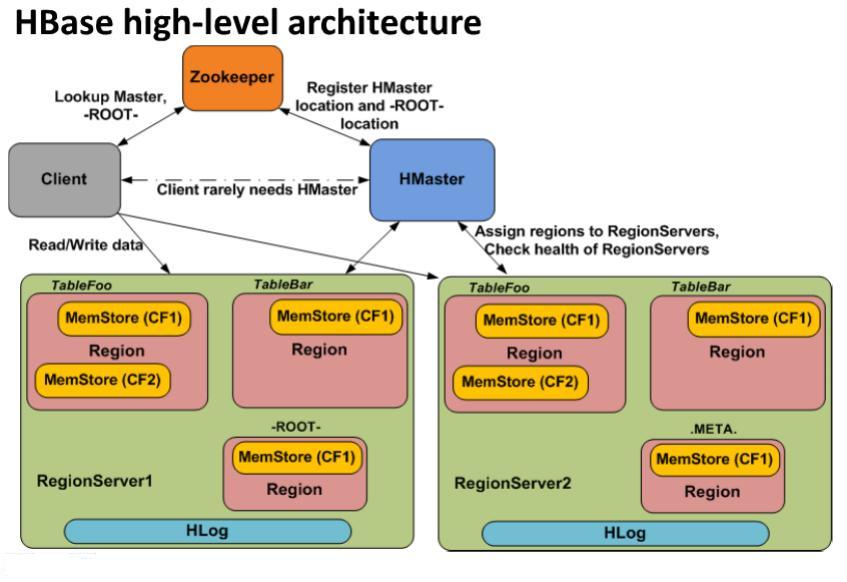
Introduction to Regions and Monitoring Disk Usage

In HBase, rows are kept in order, sorted by the row key.

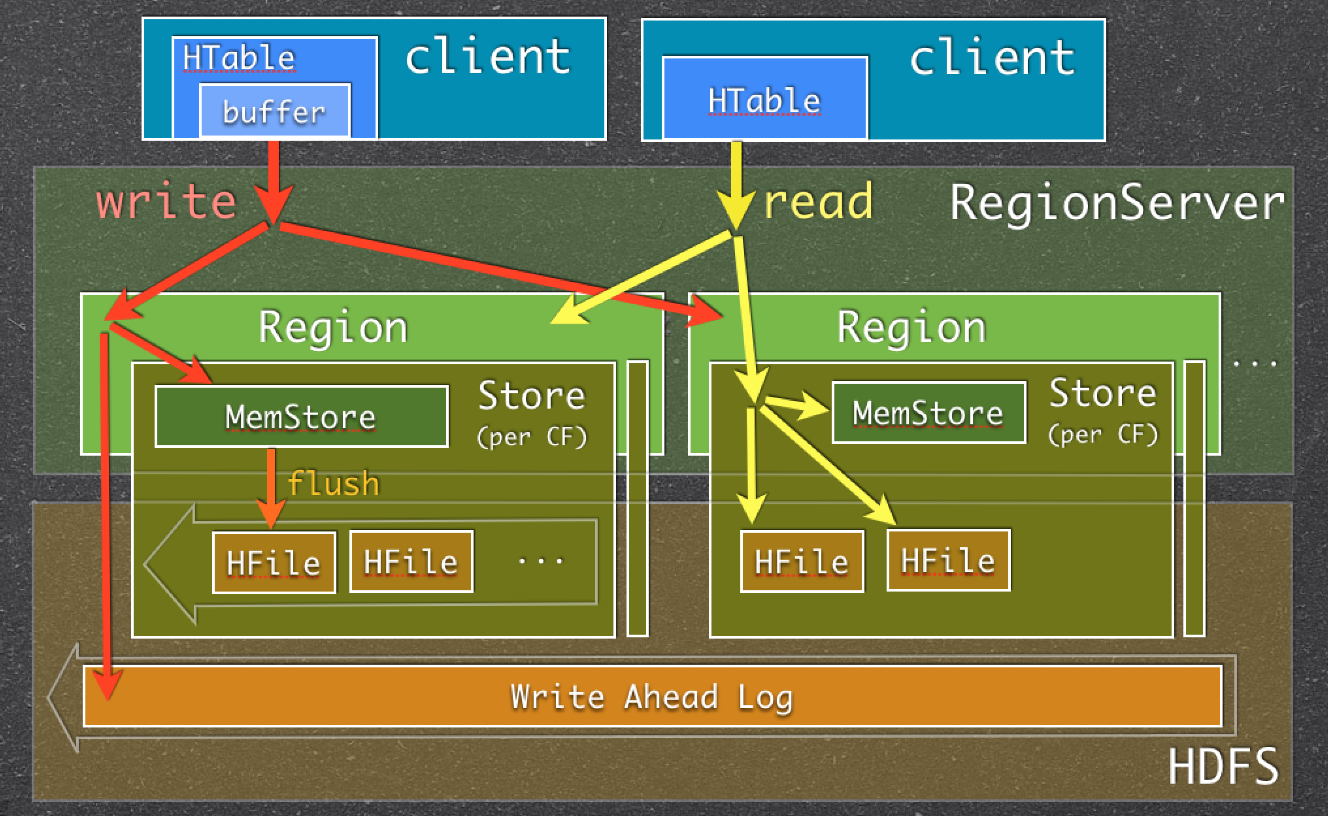
A region is a chunk of rows, identified by the starting key (inclusive) and ending key (exclusive).

Regions (Slaves) never overlap, and each is assigned to a specific region server in the cluster.





When RegionServer (RS) receives write request, it directs the request to specific Region. Each Region stores set of rows. Rows data can be separated in multiple column families (CFs). Data of particular CF is stored in HStore which consists of Memstore and a set of HFiles. Memstore is kept in RS main memory, while HFiles are written to HDFS. When write request is processed, data is first written into the Memstore. Then, when certain thresholds are met (obviously, main memory is well-limited) Memstore data gets flushed into HFile.

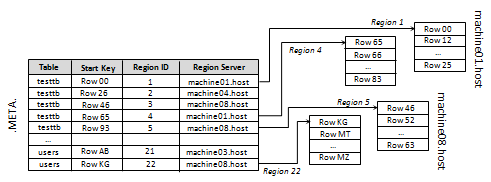


**Apache ZooKeeper** is a software project of the [Apache Software Foundation](http://en.wikipedia.org/wiki/Apache_Software_Foundation), providing an [open source](http://en.wikipedia.org/wiki/Open_source) distributed configuration service, synchronization service, and naming registry for large [distributed systems](http://en.wikipedia.org/wiki/Distributed_systems) - wiki

Clients connect to ZooKeeper to get the latest state. The HBaseMaster role is to make sure this list is correct (i.e. assign regions to regionservers on startup, failures etc.). Clients will contact the HBaseMaster only for admin purposes e.g. creating a table, changing its structure etc.

The mapping of Regions to Region Servers is kept in a system table called META. By reading META, you can identify which region is responsible for your key. This means that for read and write operations, the master is not involved at all, and clients can go directly to the Region Server responsible to serve the requested data.

To put or get a row clients don’t have to contact the master, clients can contact directly  the Region Server that handles the specified row. In the case of a scan clients can contact directly the set of Region Servers responsible for handling the set of keys.



To identify the Region Server, the client does a query on the META table. META is a system table, used to keep track of regions. It contains the server name and a region identifier comprised of a table name and the start row-key. By looking at the start-key and the next region start-key clients are able to identify the range of rows contained in a a particular region.

The client keeps a cache for the region locations. This avoids the need for clients to hit the META table every time an operation on the same region is issued. In case of a region split or move to another Region Server (due to balancing, or assignment policies) the client will receive an exception as response and the cache will be refreshed by fetching the updated information from the META table.

Client API - Master and Regions responsibilities

The HBase java client API is composed of two main interfaces.

* **HBaseAdmin**: allows interaction with the “table schema" by creating/deleting/modifying tables, and it allows interaction with the cluster by assigning/unassigning regions, merging regions together, calling for a flush, and so on.  This interface communicates with the Master.
* **HTable**: allows the client to manipulate the data of a specified table, by using get, put, delete and all the other data operations.  This interface communicates directly with the Region Servers responsible for handling the requested set of keys.

Those two interfaces have separate responsibilities: HBaseAdmin is only used to execute admin operations and communicate with the Master while the HTable is used to manipulate data and communicate with the Regions.

In case of HBase REST the client sends REST request to the REST server which holds an HBase client internally

As we’ve seen, having a Master/Slave architecture does not mean that each operation goes through the master. To read and write data the HBase client, in fact, goes directly to the specific Region Server responsible to handle the row keys for all the data operations (*HTable*). The Master is used by the client only for table creation, modification and deletion operations (*HBaseAdmin*).

See: <https://blogs.apache.org/hbase/entry/hbase_who_needs_a_master>

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:

(note look in /tmp/hbase-root/hbase)

**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

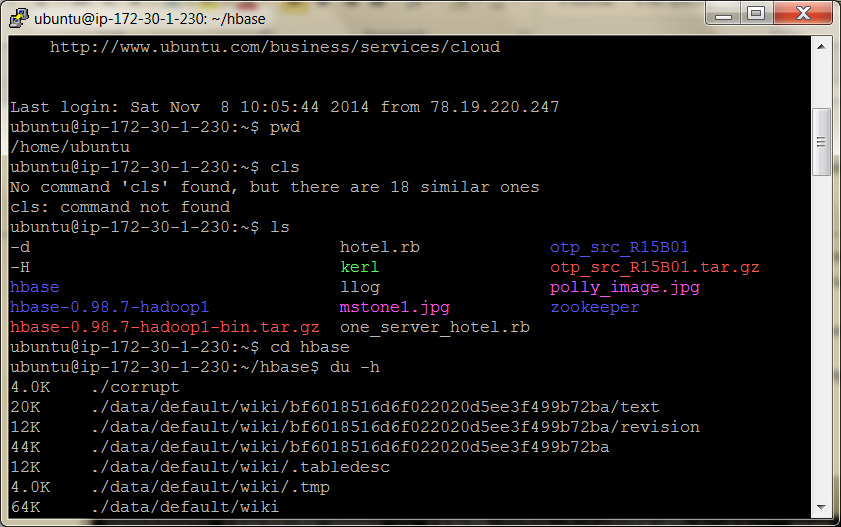
61M ./wiki/517496fecabb7d16af7573fc37257905/.tmp

12K ./wiki/517496fecabb7d16af7573fc37257905/.oldlogs 134M ./wiki/517496fecabb7d16af7573fc37257905

134M ./wiki

4.0K ./.oldlogs

365M .



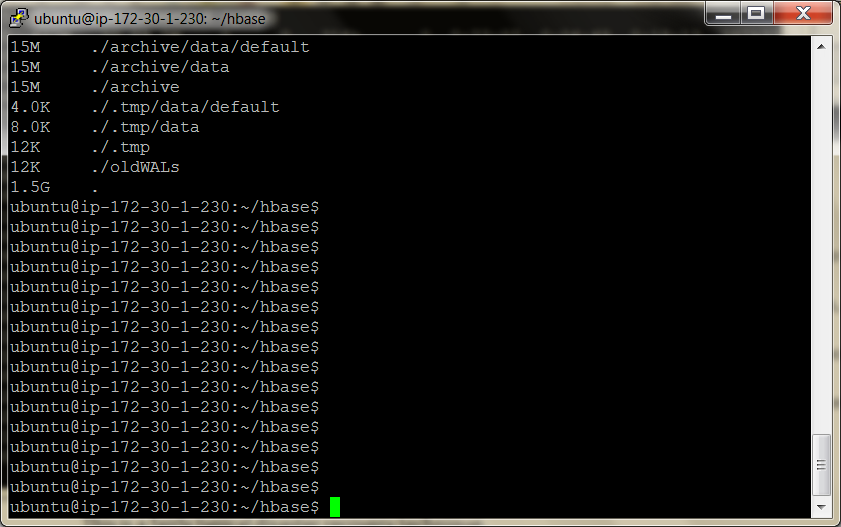
This output tells us a lot about how much space HBase is using and how it’s allocated.

The lines starting with /wiki describe the space usage for the wiki table.

The long-named subdirectory bf60185……..ba 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 1.5GB of disk space.



One more thing. The /WAL folder, show us the space used by the write-ahead log (WAL) files.

HBase uses writeahead 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.

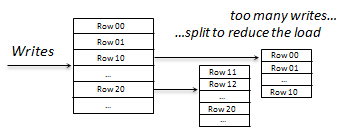
If the *region server* responsible for the affected region were to crash during this limbo period, HBase would use the WAL to determine which operations were successful and take corrective action.

Writing to the WAL is optional and enabled by default.

Regional Interrogation

If you let the script run long enough, you’ll see HBase split the table into multiple regions.

Initially, there is only one region for a table.  When regions become too large after adding more rows, the region is split into two at the middle key, creating two roughly equal halves.



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

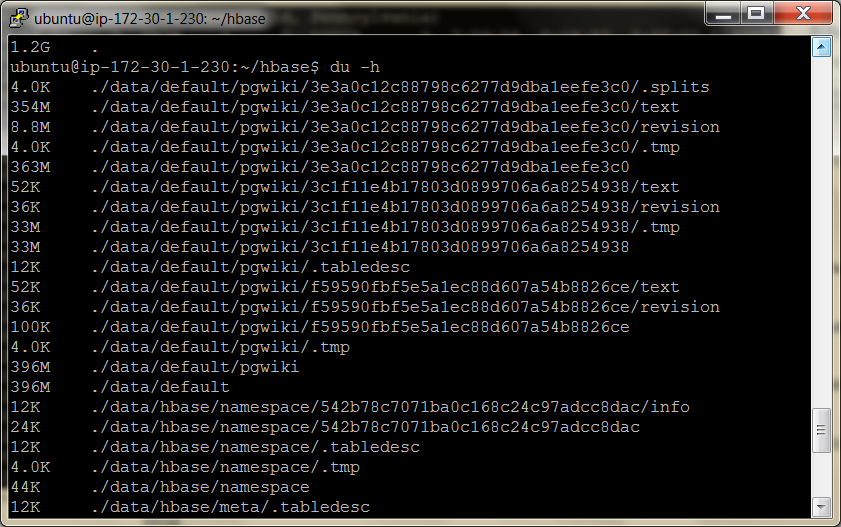
6.0M ./wiki/a79c0f6896c005711cf6a4448775a33b/revision 4.0K ./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp

141M ./wiki/a79c0f6896c005711cf6a4448775a33b

591M ./wiki

4.0K ./.oldlogs

591M .



The biggest change is that the old region (517496fecabb7d16af7573fc37257905) is now gone, replaced by two new ones.

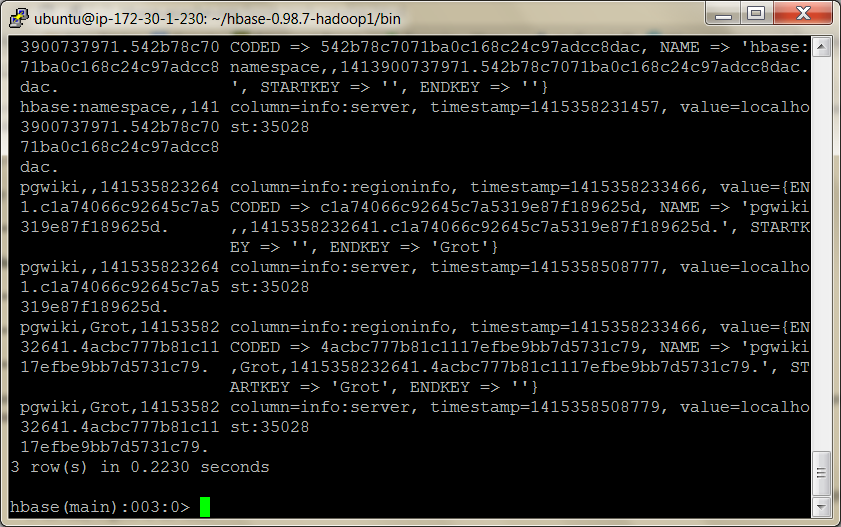
In our stand-alone server, all the regions are served by our singular server, but in a distributed environment, these would be parcelled out to the various region servers.

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 are served by the same server

The first region starts at the empty string row ('') and ends with 'Demographics of Macedonia'.

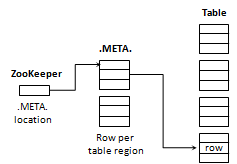
The second region starts at 'Demographics of Macedonia' and goes to 'June 30'.

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.



Since META is a table like the others, the client has to identify on which server META is located. The META locations are stored in a ZooKeeper node on assignment by the Master, and the client reads directly the node to get the address of the Region Server that contains META.

We can also see which servers have which parts of the .META. table above.

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.

<https://hbase.apache.org/0.94/replication.html>

Developing a Thrifty HBase Application

So far, we’ve been using the HBase shell, but HBase supports a number of protocols for client connectivity.

The following is a full list:

Name Connection Method Production Ready

Shell Direct Yes

Java API Direct Yes

Thrift Binary protocol Yes

REST HTTP Yes

Avro Binary protocol No (still experimental)

(A **binary protocol** is a [protocol](http://en.wikipedia.org/wiki/Network_protocol) which is intended or expected to be read by a machine rather than a human being, as opposed to a [plain text protocol](http://en.wikipedia.org/wiki/Text-based_protocol) such as [IRC](http://en.wikipedia.org/wiki/IRC),[SMTP](http://en.wikipedia.org/wiki/SMTP), or [HTTP](http://en.wikipedia.org/wiki/HTTP). Binary protocols have the advantage of terseness, which translates into speed of transmission and interpretation.

There has always been tension between two software development camps that believe new protocols should preferably be text based or binary, respectively. In recent years, with the ready availability of **network bandwidth** and mass storage, the text based camp has been gaining significant ground - [XML](http://en.wikipedia.org/wiki/XML) and [JSON](http://en.wikipedia.org/wiki/JSON)-based systems are nearly ubiquitous.)  - wiki

In the previous table, the connection method describes whether the protocol makes Java calls directly, shuttles data over HTTP, or moves data using a compact binary format.

All of them are production-grade, except for [Avro](http://avro.apache.org/docs/current/), which is relatively new and should be treated as experimental.

Of all these options, [Thrift](https://thrift.apache.org/static/files/thrift-20070401.pdf) is probably the most popular for developing client applications

A mature binary protocol with little overhead, Thrift was originally developed and open sourced by Facebook, later to become an Apache Incubator project.

Installing Thrift

Like many things in the database realm, working with Thrift requires a little setup.

To connect to our HBase server via Thrift, we’ll need to do the following:

1. Have HBase run the Thrift service.
2. Install the Thrift command-line tool.
3. Install libraries for your chosen client language
4. Generate HBase model files for your language.
5. Create and run a client application.

We’ll start by running the Thrift service, which is pretty easy.

Start the daemon from the command line like this:

background

From ${HBASE\_HOME}/bin/

sudo ./hbase-daemon.sh start thrift -b 127.0.0.1

(or foreground; hbase thrift start -p <port> --infoport <infoport>

)

(port is optional and defaults to 9090)

Also to start REST service

Foreground

hbase rest start -p <port> --infoport <infoport>

(port is optional and defaults to 8080)

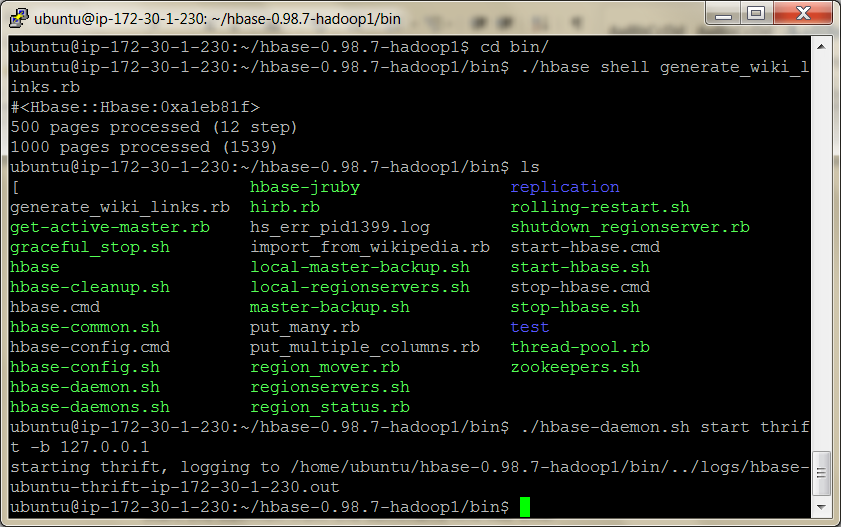
Or background:

From

bin/

sudo ./hbase-daemon.sh start rest -p <port>

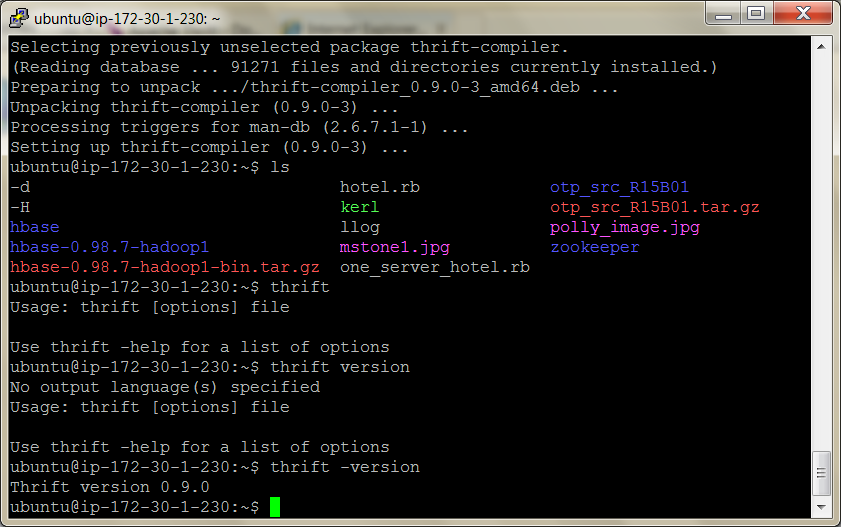
 See manual for using REST: https://hbase.apache.org/book.html#\_rest



Next, you’ll need to install the Thrift command-line tool

To test whether you have this installed correctly, call it on the command line with the -version flag. You should see something like this:

**$ thrift -version** Thrift version 0.6.0



For the client language, we’ll use Ruby, although the steps are similar for other languages.

Install the Thrift Ruby gem on the command line like so:

>sudo apt-get install ruby-dev

 >sudo  gem install thrift

Building a Client Application

Our program will connect to HBase over Thrift and then list any tables it finds along with their column families.

These would be the first steps toward building an administrative interface for HBase.

Unlike our previous examples, this script is meant to be run by good old normal Ruby, not JRuby.

It could be suitable for inclusion in a Ruby-based web application, for example.

Key this into a new text file (we called ours thrift\_example.rb):

hbase/thrift\_example.rb

$:.push('./gen-rb')

require 'thrift'

require 'hbase'

socket = Thrift::Socket.new( 'localhost', 9090 )

transport = Thrift::BufferedTransport.new( socket )

protocol = Thrift::BinaryProtocol.new( transport )

client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol )

transport.open()

client.getTableNames().sort.each do |table|

puts "#{table}"

client.getColumnDescriptors( table ).each do |col, desc|

puts " #{desc.name}"

puts " maxVersions: #{desc.maxVersions}"

puts " compression: #{desc.compression}"

puts " bloomFilterType: #{desc.bloomFilterType}"

end

end

transport.close()

In the previous code, the first thing we do is make sure Ruby can find the model files by adding gen-rb to the path and including thrift and hbase.

After that, we create a connection to the Thrift server and wire it up to an HBase client instance.

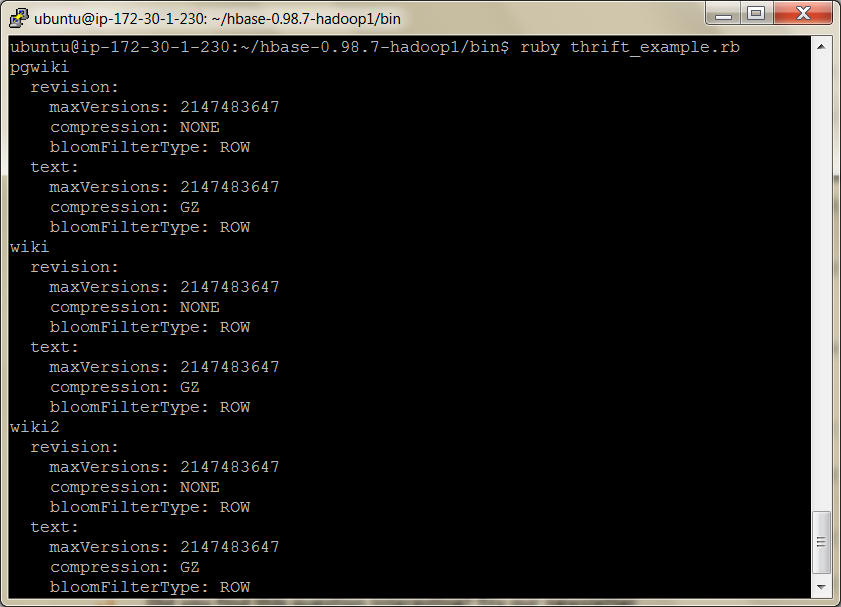
The client object will be our means for communicating with HBase.

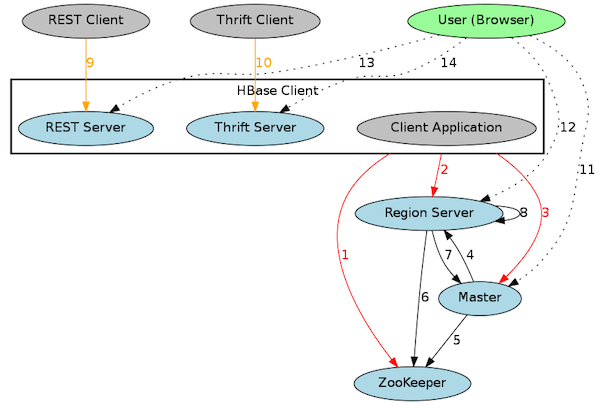
After opening the transport, we iterate over the tables brought back by getTable- Names().

For each table, we iterate over the list of column families returned by getColumnDescriptors() and output some properties to standard output.

Now, let’s run the program on the command line.

Your output should look similar since we’re connecting to the local HBase server we started with earlier.

$> ruby thrift\_example.rb  




 Wrap-Up

HBase is a juxtaposition of simplicity and complexity.

The data storage model is pretty straightforward, with a few built-in schema constraints.

It doesn’t help, though, that many terms are overloaded with baggage from the relational world (for example, words like *table* and *column*).

Most of HBase schema design is deciding on the performance characteristics of your tables and columns.

HBase’s Strengths

Noteworthy features of HBase include a robust scale-out architecture and built-in versioning and compression capabilities.

HBase’s built-in versioning capability can be a compelling feature for certain use cases.

Keeping the version history of wiki pages is a crucial feature for policing and maintenance, for instance.

By choosing HBase, we don’t have to take any special steps to implement page history—we get it for free.

On the performance front, HBase is meant to scale out.

If you have huge amounts of data, measured in many gigabytes or terabytes, HBase may be for you.

HBase is rack-aware, replicating data within and between datacenter racks so that node failures can be handled gracefully and quickly.

The HBase community is pretty awesome. There’s almost always somebody on the IRC channel or mailing lists ready to help with questions and get you pointed in the right direction.

Although a number of high-profile companies use HBase for their projects, there is no corporate HBase service provider.

This means the people of the HBase community do it for the love of the project and the common good.

HBase’s Weaknesses

Although HBase is designed to scale out, it doesn’t scale down.

The HBase community seems to agree that five nodes is the minimum number you’ll want to use.

Because it’s designed to be big, it can also be harder to administrate

Solving small problems isn’t what HBase is about, and nonexpert documentation is tough to come by, which steepens the learning curve.

Additionally, HBase is almost never deployed alone.

Rather, it’s part of an ecosystem of scale-ready pieces, called Hadoop.

These include YARN (an implementation of Google’s MapReduce), the Hadoop distributed file system (HDFS), and Zookeeper (a configuration service that aids internode coordination), Pig and Hive (to do SQL like queries of the data)

This ecosystem is both a strength and a weakness; it simultaneously affords a great deal of architectural sturdiness but also encumbers the administrator with the burden of maintaining it.

One noteworthy characteristic of HBase is that it doesn’t offer any sorting or indexing capabilities aside from the row keys.

Rows are kept in sorted order by their row keys, but no such sorting is done on any other field, such as column names and values.

So, if you want to find rows by something other than their key, you need to scan the table or maintain your own index.

Another missing concept is datatypes.

All field values in HBase are treated as uninterpreted arrays of bytes.

There is no distinction between, say, an integer value, a string, and a date.

They’re all bytes to HBase, so it’s up to your application to interpret the bytes.

HBase on CAP

With respect to CAP, HBase is decidedly CP.

HBase makes strong consistency guarantees.

If a client succeeds in writing a value, other clients will receive the updated value on the next request.

Some databases, like Riak, allow you to tweak the CAP equation on a per-operation basis.

Not so with HBase.

In the face of reasonable amounts of partitioning—for example, a node failing— HBase will remain available, shunting the responsibility off to other nodes in the cluster.

However, in the pathological example, where only one node is left alive, HBase has no choice but to refuse requests.

Parting Thoughts

The terminology can be deceptively reassuring, and the installation and configuration are not for the faint of heart.

On the plus side, some of the features HBase offers, such as versioning and compression, are quite unique.

These aspects can make HBase quite appealing for solving certain problems.

And of course, it scales out to many nodes of commodity hardware quite well.

See the document ‘*introduction to hbase schema design’*

 Some useful commands in Hbase

<https://akbarahmed.com/2012/08/13/hbase-command-line-tutorial/>

list - to see all tables

scan ‘table’ – all columns

get ‘table’, ‘row1’