Big Data – Exercises

Fall 2021 – Week 5 – ETH Zurich

Wide Column Stores - HBase

This exercise will consist of five main parts:

- Hands-on practice with your own HBase cluster running in Docker
- Architecture of HBase
- Bloom filter
- Log-structured merge-tree (LSM tree, optional)

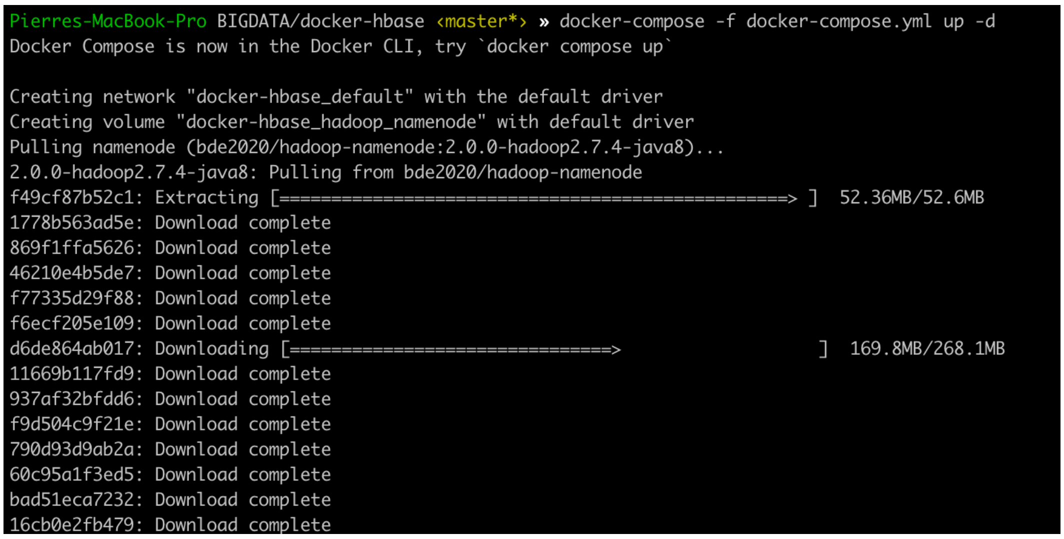
Exercise 1 — Creating and using an HBase cluster

It's time to touch HBase! You will create, fill with data, and query an HBase cluster running on Azure.

Do the following to set up an HBase cluster using Docker:

0. Please first get the folder from Moodle containing all the necessary files that configure Docker. Please also open Docker, you should see that docker is active with green on the UI of Docker desktop.

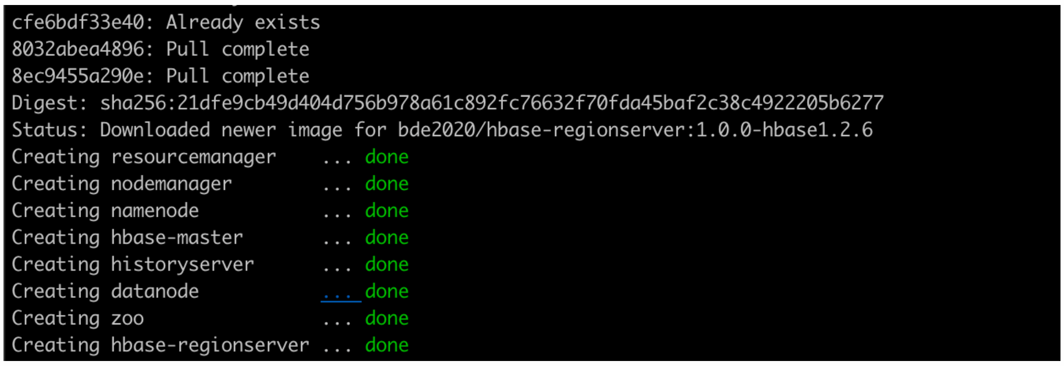
1. In the command line, navigate into your folder using 'cd' and instantiate the cluster by running: docker-compose -f docker-compose.yml up -d



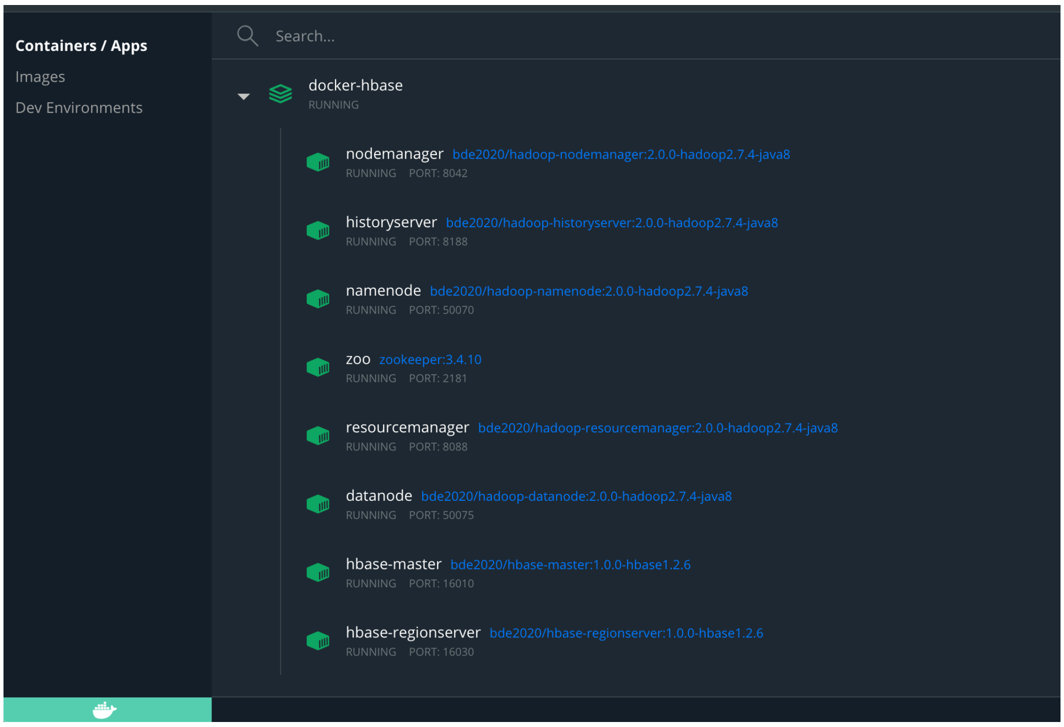
```
Pierres-MacBook-Pro BIGDATA/docker-hbase <master*> » docker-compose -f docker-compose.yml up -d
Docker Compose is now in the Docker CLI, try `docker compose up`

Creating network "docker-hbase_default" with the default driver
Creating volume "docker-hbase_hadoop_namenode" with default driver
Pulling namenode (bde2020/hadoop-namenode:2.0.0-hadoop2.7.4-java8)...
2.0.0-hadoop2.7.4-java8: Pulling from bde2020/hadoop-namenode
f49cf87b52c1: Extracting [==================================================> ]  52.36MB/52.6MB
1778b563ad5e: Download complete
869f1ffa5626: Download complete
46210e4b5de7: Download complete
f77335d29f88: Download complete
f6ecf205e109: Download complete
d6de864ab017: Downloading [===============================>                   ]  169.8MB/268.1MB
11669b117fd9: Download complete
937af32bfdd6: Download complete
f9d504c9f21e: Download complete
790d93d9ab2a: Download complete
60c95a1f3ed5: Download complete
bad51eca7232: Download complete
16cb0e2fb479: Download complete
```

After a few minutes you should get this output in the command line and the UI will display each container's status:



```
cfe6bdf33e40: Already exists
8032abea4896: Pull complete
8ec9455a290e: Pull complete
Digest: sha256:21dfe9cb49d404d756b978a61c892fc76632f70fda45baf2c38c4922205b6277
Status: Downloaded newer image for bde2020/hbase-regionserver:1.0.0-hbase1.2.6
Creating resourcemanager    ... done
Creating nodemanager        ... done
Creating namenode           ... done
Creating hbase-master       ... done
Creating historyserver      ... done
Creating datanode           ... done
Creating zoo                ... done
Creating hbase-regionserver ... done
```



docker-hbase
RUNNING

nodemanager bde2020/hadoop-nodemanager:2.0.0-hadoop2.7.4-java8
RUNNING PORT: 8042

historyserver bde2020/hadoop-historyserver:2.0.0-hadoop2.7.4-java8
RUNNING PORT: 8188

namenode bde2020/hadoop-namenode:2.0.0-hadoop2.7.4-java8
RUNNING PORT: 50070

zoo zookeeper:3.4.10
RUNNING PORT: 2181

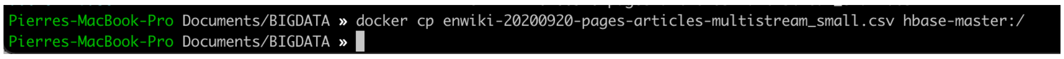
resourcemanager bde2020/hadoop-resourcemanager:2.0.0-hadoop2.7.4-java8
RUNNING PORT: 8088

datanode bde2020/hadoop-datanode:2.0.0-hadoop2.7.4-java8
RUNNING PORT: 50075

hbase-master bde2020/hbase-master:1.0.0-hbase1.2.6
RUNNING PORT: 16010

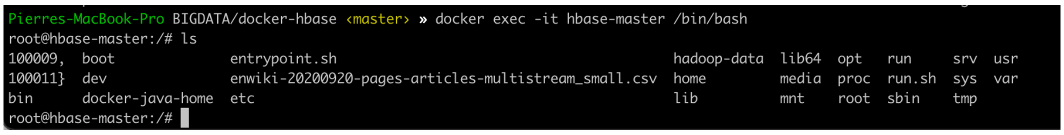
hbase-regionserver bde2020/hbase-regionserver:1.0.0-hbase1.2.6
RUNNING PORT: 16030

2. You will then copy the csv data file from your local system to the docker container (the hbase-master) where hbase is running. This will be useful in Exercise 2.



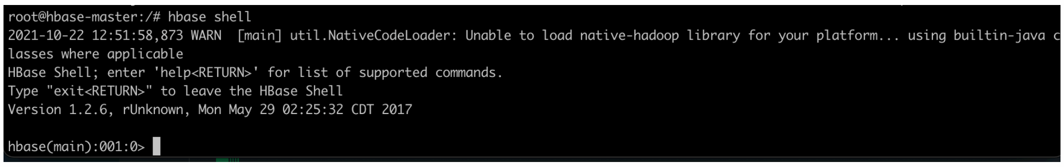
```
Pierres-MacBook-Pro Documents/BIGDATA » docker cp enwiki-20200920-pages-articles-multistream_small.csv hbase-master:/
Pierres-MacBook-Pro Documents/BIGDATA »
```

3. Access to your container's bash by running the following command, once in the bash, you can list your files and check the presence of the csv data file:



```
Pierres-MacBook-Pro BIGDATA/docker-hbase ‹master› » docker exec -it hbase-master /bin/bash
root@hbase-master:/# ls
100009,  boot              entrypoint.sh                                      hadoop-data  lib64  opt   run     srv  usr
100011}  dev               enwiki-20200920-pages-articles-multistream_small.csv  home         media  proc  run.sh  sys  var
bin      docker-java-home  etc                                                lib          mnt    root  sbin    tmp
root@hbase-master:/#
```

4. We will use the .csv file later to populate our database. For now let's explore the basics of HBase in this playground. To start, run 'hbase shell' in the container's bash:



```
root@hbase-master:/# hbase shell
2021-10-22 12:51:58,873 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java c
lasses where applicable
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.6, rUnknown, Mon May 29 02:25:32 CDT 2017

hbase(main):001:0>
```

Interact with your HBase cluster using the shell

In this task we will go through some basic HBase commands, in preparation for the exercise after the next one, where we will import a dataset from a csv file and run queries against it.

Open the HBase shell by running the following command:

```
hbase shell
```

Let's say we want to create an HBase table that will store sentences adhering to the structure subject-verb-object (e.g., "I eat mangoes", "She writes books") in different languages. Here is a schema that we may use:

Table name = `sentences`

- Column family: `words`
  - column: `subject`
  - column: `verb`
  - column: `object`
- Column family: `info`
  - column: `language`

With the following command we can create such a table (a description of HBase shell commands is available here):

```
create 'sentences', 'words', 'info'
```

You can see the schema of the table with this command:

```
describe 'sentences'
```

Let's insert some sentences into our table. We will put data cell by cell with the command `put <table>, <rowId>, <columnFamily:columnQualifier>, <value>`:

```
put 'sentences', 'row1', 'words:subject', 'I'
```

```
put 'sentences', 'row1', 'words:verb', 'drink'
```

```
put 'sentences', 'row1', 'words:object', 'coffee'
```

Now, let's try to query this sentence from the table:

```
get 'sentences', 'row1'
```

You should see output similar to this:

```
COLUMN                                    CELL
 words:object                             timestamp=1602785046682, value=coffee
 words:subject                            timestamp=1602785045625, value=I
 words:verb                               timestamp=1602785045849, value=drink
3 row(s) in 0.0910 seconds
```

As you can see, HBase shell returns data as key-value pairs rather than as rows literally. You may also notice that the lines are lexicographically sorted by the key, which is why "subject" appears after "object" in the list.

I don't know about you, but I like tea more than coffee, so let me update our sentence...

`put 'sentences', 'row1', 'words:object', 'tea'`

As you can see, we are using the same `put` command to *update* a cell. But remember that HBase does not actually update cells in place—it just inserts new versions instead. If you now run the query again, you will see the new data:

`get 'sentences', 'row1'`

returns:

```
COLUMN                                    CELL
 words:object                             timestamp=1602785160890, value=tea
 words:subject                            timestamp=1602785045625, value=I
 words:verb                               timestamp=1602785045849, value=drink
3 row(s) in 0.0200 seconds
```

We actually wanted to store sentences in different languages, so let's first set the language for the existing one:

`put 'sentences', 'row1', 'info:language', 'English'`

Note that we are now inserting a value into a different column family but for the same row. Verify with a `get` that this took effect.

Now, let's add a sentence in another language (note that we are using another rowID now—`row2`):

`put 'sentences', 'row2', 'words:subject', 'Ich'`

`put 'sentences', 'row2', 'words:verb', 'trinke'`

`put 'sentences', 'row2', 'words:object', 'Wasser'`

`put 'sentences', 'row2', 'info:language', 'Deutsch'`

Let's check that we indeed have 2 rows now:

`count 'sentences'`

Now, let's query all rows from the table:

`scan 'sentences'`

This, indeed, returns all two rows, in key-value format as before.

Of course, you can also scan by column family for column:

`scan 'sentences', {COLUMNS => 'info'}`

`scan 'sentences', {COLUMNS => 'info:language'}`

You can also scan by row ranges (note min incl., max excl.):

```
scan 'sentences', {STARTROW=>'row1', ENDROW=>'row3'}
```

It is, of course, possible to do some filtering in queries:

```
scan 'sentences', {FILTER => "ValueFilter(=, 'binary:English')"}
```

```
scan 'sentences', {COLUMNS => 'words:subject', FILTER => "ValueFilter(=, 'substring:I')"}
```

```
scan 'sentences', {COLUMNS => 'words:object', ROWPREFIXFILTER => 'row'}
```

What if we want to store a sentence that also contains an adjective, in addition to the subject, verb, and object? This is not a problem with HBase, because we can create new columns inside *existing* column families on the fly:

```
put 'sentences', 'row3', 'words:subject', 'Grandma'
```

```
put 'sentences', 'row3', 'words:verb', 'bakes'
```

```
put 'sentences', 'row3', 'words:adjective', 'delicious'
```

```
put 'sentences', 'row3', 'words:object', 'cakes'
```

This row now has more columns in the `words` column family than others:

```
get 'sentences', 'row3'
```

We can also add new columns to existing rows:

```
put 'sentences', 'row1', 'words:adjective', 'hot'
```

```
get 'sentences', 'row1'
```

Let's see what happens if you update a value in an existing column:

```
put 'sentences', 'row1', 'words:adjective', 'cold'
```

You should notice that the time stamp of the column `words:adjective` has been updated.

When you are done with the queries, simply type `exit` to quit the hbase shell.

Note: to drop a table in HBase, first `disable <table>` then `drop <table>`.

Important: if you do not plan to do the next section right now, please delete your cluster and just recreate it when you need it again.

Exercise 2 — The Wikipedia dataset

In this task we will see how HBase will handle a large dataset, as well as learn about the filters and caching in HBase.

The provided dataset comprises approximately the meta data of articles from the English Wikipedia. You will see the following variables in the csv file:

variable	meaning	Sample value
`page_id`	the page id in the enwiki data dump	1000108
`page_title`	the page id in the enwiki data dump	AEG Z.6
`page_ns`	page namespace	0
`revision_id`	the id of revision to the article	16782282
`timestamp`	the time a contributor makes a revision	2004-09-19T23:44:33Z
`contributor_id`	the id of contributor	8817
`contributor_name`	the name of contributor	Rlandmann
`bytes`	bytes in revision	21

We will use the `wiki_small` data (about 85MB in csv) in this assignment because it takes shorter time (about 5-10 minutes) to load into HBase.

Based on the descriptives about the variables above, we can categorize the variables into some about a page and other about an author/contributor.
Let us create the schemas in HBase now with two column families `page` and `author`

```
hbase shell
```

```
create 'wiki_small', 'page', 'author'
```

After the table is created, we need to exit the HBase shell and return back to the container's bash:

```
exit
```

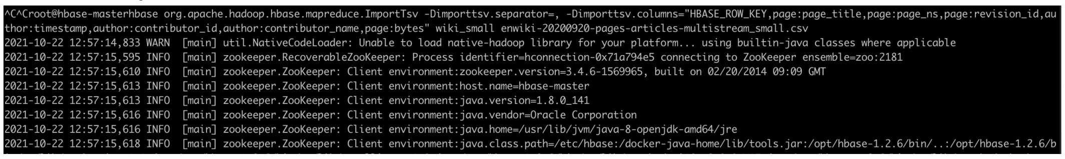
Now we need to populate both tables with data. We will use the ImportTsv utility of HBase.

Populate the table `wiki_small` by running the following command. We need to specify which column in the csv maps to which column in the HBase table. Note that we make `page_id` into the `HBASE_ROW_KEY`. Note that these commands print a lot of messages, but they are mostly informational with an occasional non-critical warning; unless something goes wrong, of course :) The commands will also report some "Bad Lines", but you can safely ignore this—some lines may contain illegal characters and be dropped, but most of the data is in good shape.

`hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=, -Dimporttsv.columns="HBASE_ROW_KEY,page:page_title,page:page_ns,page:revision_id,author:timestamp,author:contributor_id,author:contributor_name,page:bytes" wiki_small enwiki-20200920-pages-articles-multistream_small.csv`

Note here how we specify the mappings between the csv columns and the column family:column in the HBase table.

Recall that you should run this command into the container's bash.



```
^C^Croot@hbase-masterhbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=, -Dimporttsv.columns="HBASE_ROW_KEY,page:page_title,page:page_ns,page:revision_id,au
thor:timestamp,author:contributor_id,author:contributor_name,page:bytes" wiki_small enwiki-20200920-pages-articles-multistream_small.csv
2021-10-22 12:57:14,833 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2021-10-22 12:57:15,595 INFO  [main] zookeeper.RecoverableZooKeeper: Process identifier=hconnection-0x71a794e5 connecting to ZooKeeper ensemble=zoo:2181
2021-10-22 12:57:15,610 INFO  [main] zookeeper.ZooKeeper: Client environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT
2021-10-22 12:57:15,613 INFO  [main] zookeeper.ZooKeeper: Client environment:host.name=hbase-master
2021-10-22 12:57:15,613 INFO  [main] zookeeper.ZooKeeper: Client environment:java.version=1.8.0_141
2021-10-22 12:57:15,616 INFO  [main] zookeeper.ZooKeeper: Client environment:java.vendor=Oracle Corporation
2021-10-22 12:57:15,616 INFO  [main] zookeeper.ZooKeeper: Client environment:java.home=/usr/lib/jvm/java-8-openjdk-amd64/jre
2021-10-22 12:57:15,618 INFO  [main] zookeeper.ZooKeeper: Client environment:java.class.path=/etc/hbase:/docker-java-home/lib/tools.jar:/opt/hbase-1.2.6/bin/..:/opt/hbase-1.2.6/b
```

You can count how many rows there are using this command from your head node's shell:
`hbase org.apache.hadoop.hbase.mapreduce.RowCounter 'wiki_small'`

Now let's go into HBase shell again and run some queries against the `wiki_small` table. We will look at some of the filters listed by HBase if you run `show_filters` in an HBase shell, e.g., `PrefixFilter(), ValueFilter(), SingleColumnValueFilter()`.

Tasks to do with the table `wiki_small`

1. How does HBase index the row keys?
   We choose `page_id` in the original table to be the row keys in the HBase table. Run these two queries and what do you observe? What can we say about row key indexing based their results?

   ```
   scan 'wiki_small', {STARTROW=>'100009', ENDROW=>'100011'}
   ```

   ```
   scan 'wiki_small', {STARTROW=>'100015', ENDROW=>'100016'}
   ```

2. Write the following queries:
3. Select all article titles and author names where the row name starts with '`1977`'
4. Select all article titles and author names where the author contains the substring '`tom`'.
5. Write the following queries:
6. Return the number of articles from 2017.
7. Return the number of articles that contain the word `Attacks` on them. Please discuss different possibilities to formulate this query.
8. Execute your queries more than once and observe the query execution times
9. What are the advantages and disadvantages of pure row stores?

0. What are the advantages and disadvantages of pure column stores?

.1. What are the advantages and disadvantages of wide column stores?

2. What are the advantages and disadvantages of denormalization?

Exercise 3 — Architecture of HBase

In the previous tasks, we have seen HBase in action. Let us now take a look at the internal architecture of HBase. You may want to consult the lecture slides when solving these tasks.

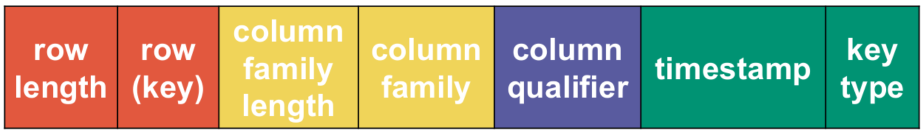
In this exercise you will see how a RegionServer in HBase would execute a query.

Imagine that we have an HBase table called '`phrases`', which has the following schema:

- Column family: `words`
  - column: A
  - column: B
  - column: C
  - (potentially also columns D, E, F, etc.)

Thus, the table has only one column family. Each column in this family holds one word.

Recall from the lecture slides that keys in HBase have the following structure:



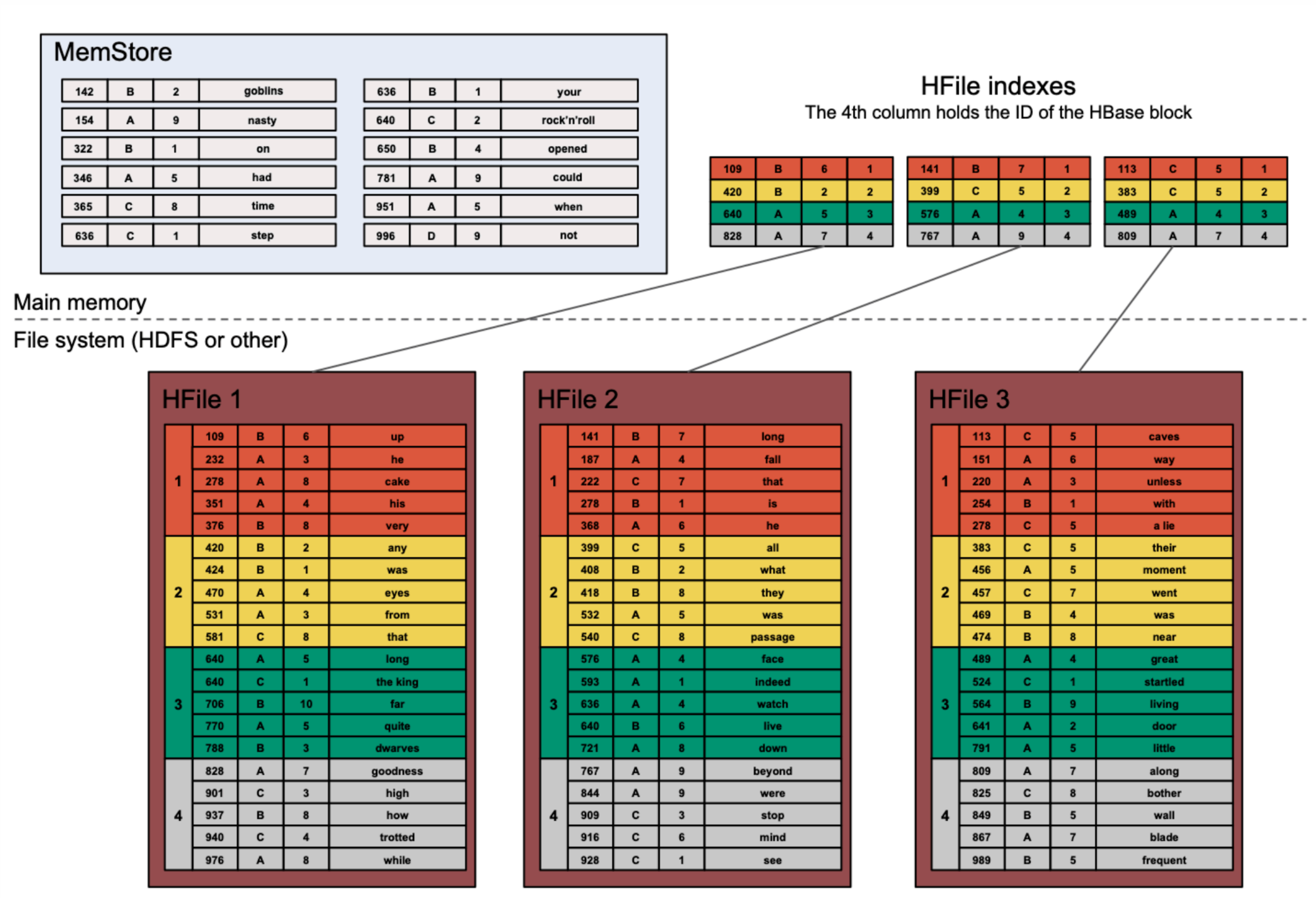
We need make certain simplifications to the format of keys to avoid excessive clutter in this exercise. Since the table in this exercise has only one column family, we will omit it from the key and will only specify the column name (A,B,C, ...). We will also omit the length fields and the "key type" field. The timestamp field in this exercise will contain integers from 1 to 10, where in reality it would contain the number of milliseconds since an event in the long past. Thus, keys as will be used in this exercise consist of three fileds: row, column, timestamp.

Tasks to do

State which Key-Value pairs will be returned by each of the following queries, given in HBase shell syntax which you have already seen in the first exercise. Assume that the HBase instance is configured to return only the latest version of a cell.

1. `get 'phrases', '278'`
2. `get 'phrases', '636'`
3. `get 'phrases', '593'`
4. `get 'phrases', '640'`
5. `get 'phrases', '443'`

To answer this question, use the diagram below, which represents the state of a RegionServer responsible for the row region in the range of row IDs 100–999, which is the region into which all these queries happen to fall.


MemStore
HFile indexes
The 4th column holds the ID of the HBase block
Main memory
File system (HDFS or other)
HFile 1
HFile 2
HFile 3

MemStore

142	B	2	goblins
154	A	9	nasty
322	B	1	on
346	A	5	had
365	C	8	time
636	C	1	step
636	B	1	your
640	C	2	rock'n'roll
650	B	4	opened
781	A	9	could
951	A	5	when
996	D	9	not

HFile indexes

The 4th column holds the ID of the HBase block

109	B	6	1
420	B	2	2
640	A	5	3
828	A	7	4

141	B	7	1
399	C	5	2
576	A	4	3
767	A	9	4

113	C	5	1
383	C	5	2
489	A	4	3
809	A	7	4

HFile 1

1	109	B	6	up
	232	A	3	he
	278	A	8	cake
	351	A	4	his
	376	B	8	very
2	420	B	2	any
	424	B	1	was
	470	A	4	eyes
	531	A	3	from
	581	C	8	that
3	640	A	5	long
	640	C	1	the king
	706	B	10	far
	770	A	5	quite
	788	B	3	dwarves
4	828	A	7	goodness
	901	C	3	high
	937	B	8	how
	940	C	4	trotted
	976	A	8	while

HFile 2

1	141	B	7	long
	187	A	4	fall
	222	C	7	that
	278	B	1	is
	368	A	6	he
2	399	C	5	all
	408	B	2	what
	418	B	8	they
	532	A	5	was
	540	C	8	passage
3	576	A	4	face
	593	A	1	indeed
	636	A	4	watch
	640	B	6	live
	721	A	8	down
4	767	A	9	beyond
	844	A	9	were
	909	C	3	stop
	916	C	6	mind
	928	C	1	see

HFile 3

1	113	C	5	caves
	151	A	6	way
	220	A	3	unless
	254	B	1	with
	278	C	5	a lie
2	383	C	5	their
	456	A	5	moment
	457	C	7	went
	469	B	4	was
	474	B	8	near
3	489	A	4	great
	524	C	1	startled
	564	B	9	living
	641	A	2	door
	791	A	5	little
4	809	A	7	along
	825	C	8	bother
	849	B	5	wall
	867	A	7	blade
	989	B	5	frequent

You can format your answer for this exercise as follows

1. get 'phrases', 'row_id'

Row	Column	Timestamp	Value	Where it came from (which HFile)

Task 3.2 — Bloom filters

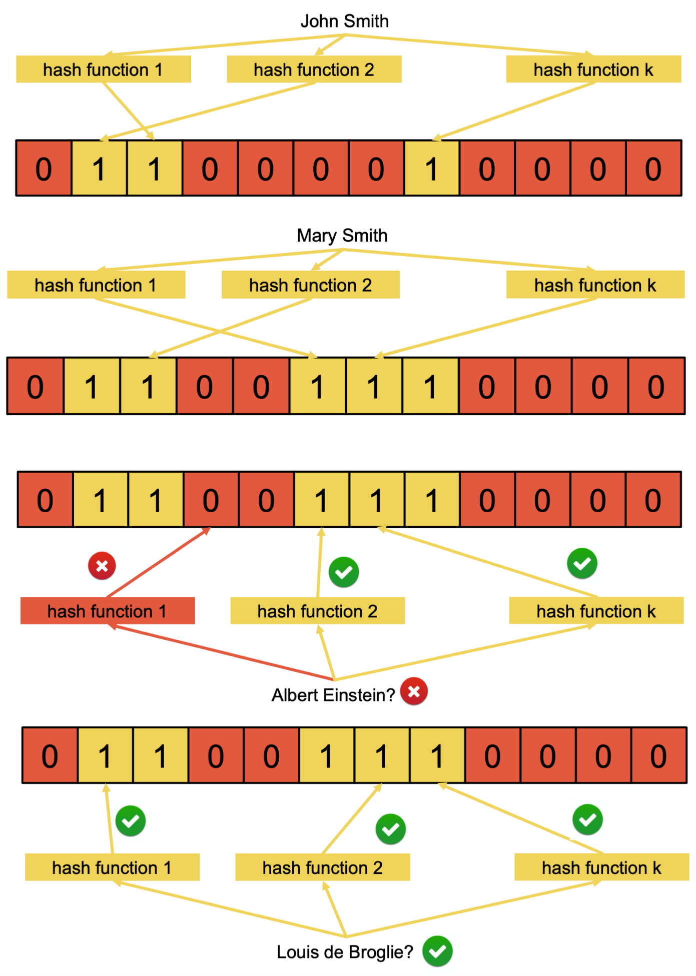
Let's start with summarizing what we have in memory when working with HBase: MemStore, LRU (Least Recently Used) BlockCache, Indices of HFiles, and Bloom Filters. We did not have time to cover it in the lecture, but bloom filters are actually very crucial in avoiding disk reads if we can guarantee that a key is **not** in an HFile. Bloom filters allow us to very quickly determine whether an element belongs to a set.

Bloom filters are a data structure used to speed up queries, useful in the case in which it's likely that the value we are looking doesn't exist in the collection we are querying. Their main component is a bit array with all values initially set to 0. When a new element is inserted in the collection, its value is first run through a certain number of (fixed) hash functions, and the locations in the bit array corresponding to the outputs of these functions are set to 1.

This means that when we query for a certain value, if the value has previously been inserted in the collection then all the locations corresponding to the hash function outputs will certainly already have been set to 1. On the contrary, if the element hasn't been previously inserted, then the locations may or may not have already been set to 1 by other elements.

Then, if prior to accessing the collection we run our queried value through the hash functions, check the locations corresponding to the outputs, and find any of them to be 0, we are guaranteed that the element is not present in the collection (No False Negatives), and we don't have to waste time looking. If the corresponding locations are all set to 1, the element may or may not be present in the collection (Possibility of False Positives), but in the worst case we're just wasting time.

Inspect the following examples. Say we have hash functions that map the input `John Smith` and `Mary Smith` to the bit array `011001110000`. When we have a new input `Albert Einstein` which is mapped by the same hash functions to the bit array `011001110000`. This clearly does not correspond to the bit array produced by the previous two inputs. Hence, we can say that `Albert Einstein` is not in the set which `John Smith` and `Mary Smith` belong to (denoted as `{the Smiths}` for short). However, another input `Louis de Broglie` whose bit array after hashing is `011001110000` is then a false positive for the set `{the Smiths}`.



John Smith
hash function 1
hash function 2
hash function k
0 1 1 0 0 0 0 1 0 0 0 0
Mary Smith
hash function 1
hash function 2
hash function k
0 1 1 0 0 1 1 1 0 0 0 0
0 1 1 0 0 1 1 1 0 0 0 0
hash function 1
hash function 2
hash function k
Albert Einstein?
0 1 1 0 0 1 1 1 0 0 0 0
hash function 1
hash function 2
hash function k
Louis de Broglie?

As you have seen in the task above, HBase has to check all HFiles, along with the MemStore, when looking for a particular key. As an optimisation, Bloom filters are used to avoid checking an HFile if possible. Before looking inside a particular HFile, HBase first checks the requested key against the Bloom filter associated with that HFile. If it says that the key does not exist, the file is not read.

In this task we will look at how Bloom filters work. We will use a list of words to populate a Bloom filter and we will then query it.

Bloom filter requires several hash functions. To keep things easily computable by a human, we will define the following three hash functions for the purpose of this exercise:

1. Given an English word x, the value of the first hash function, $hash_1(x)$, is equal to the *first letter of the word*. E.g.: $hash_1$(`"federal"`) = `"f"`
2. Given an English word x, the value of the second hash function, $hash_2(x)$, is equal to the *second letter of the word*. E.g.: $hash_2$(`"federal"`) = `"e"`
3. Given an English word x, the value of the third hash function, $hash_3(x)$, is equal to the *third letter of the word*. E.g.: $hash_3$(`"federal"`) = `"d"`

A Bloom filter starts with a bit array which has value "0" recorded for each possible output value of all three hash functions (or, for example, modulo the size of the bit array, if the output range of the hash functions is too large). When we *add* an element to a Bloom filter, we compute the three values of the three hash functions and set those locations in the Bloom filter to "1". For example, if we add "`federal`" to the Bloom filter using the three hash functions that we have defined above, we get the following:

			1	1	1																				
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Here, only values "1" are displayed to avoid cluttering the view; thus, if a cell is empty, it is assumed to hold a "0".

First, populate the following table with the outputs of these hash functions (double-click the table to edit it and hit Ctrl+Enter to exit the editing mode; you are also free to do this task in some other tool, of course):

Word	$hash_1$	$hash_2$	$hash_3$
round			
sword			
past			
pale			
nothing			
darkness			
water			
feet			
thin			
passage			
corner			

Now, *add* each word from the list into the following Bloom filter (you can also double-click to edit it; you can double-click the Bloom filter populated with "federal" above to see an example of a filled-in filter):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

For each word from the following list, state whether this Bloom filter reports it as belonging to the set or not (skip filling-in the hash columns, if you want):

Word	$hash_1$	$hash_2$	$hash_3$	**The Bloom filter says the word belongs to the set: (yes/no)**
sword				
sound				
psychic				
pale				
book				
deaf				
truss				

Which of the words that were flagged by the Bloom filter as belonging to the set are actually **not** in the set (a *false positive* outcome)?

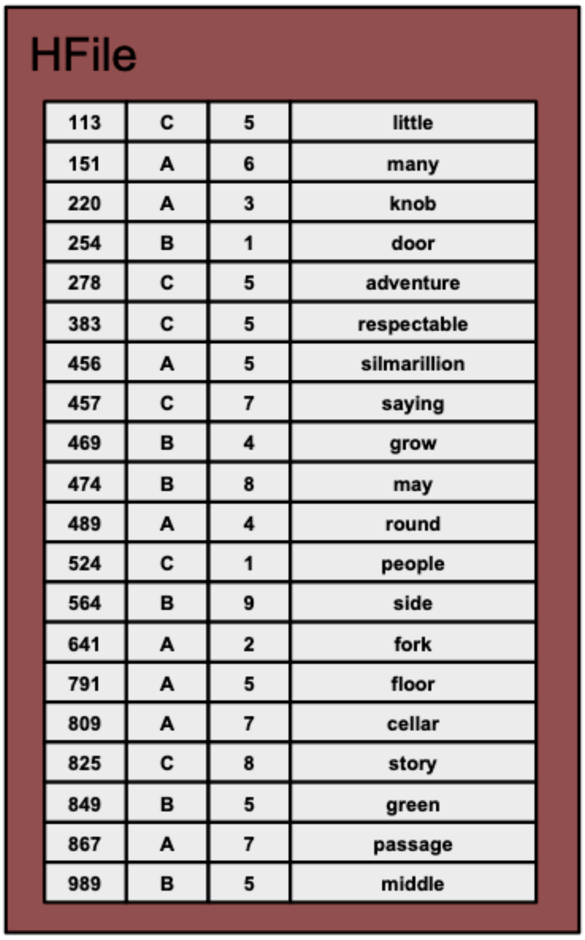
Which of the words that were flagged by the Bloom filter as **not** belonging to the set actually **do belong** to the set (a *false negative* outcome)?

Task 3.3 — Building an HFile index

As discussed in Task 3.2, HBase uses Bloom filters, stored in the metadata of each HFile, in order to discard all get requests which query data not stored in the HFile. However when performing a get for which the bloom filter returns positive, the RegionServer needs to check its MemStore and all HFiles for the existence of the requested key. In order to avoid scanning HFiles entirely, HBase uses index structures to quickly skip to the position of the *HBase block* which may hold the requested key. Note HBase block is not to be confused with HDFS block and the underlying file system block, see here for a good discussion. HBase blocks come in 4 varieties: DATA, META, INDEX, and BLOOM.

By default, each *HBase block* is 64KB (configurable) in size and always contains whole key-value pairs, so, if a block needs more than 64KB to avoid splitting a key-value pair, it will just grow.

In this task, you will be building the index of an HFile. **For the purpose of this exercise**, assume that each HBase block is 40 bytes long, and each character in keys and values is worth 1 byte: for example, the first key-value pair in the diagram below is worth $3 + 1 + 1 + 6 = 11$ bytes. Below this diagram you will find a table for you to fill in.



HFile

113	C	5	little
151	A	6	many
220	A	3	knob
254	B	1	door
278	C	5	adventure
383	C	5	respectable
456	A	5	silmarillion
457	C	7	saying
469	B	4	grow
474	B	8	may
489	A	4	round
524	C	1	people
564	B	9	side
641	A	2	fork
791	A	5	floor
809	A	7	cellar
825	C	8	story
849	B	5	green
867	A	7	passage
989	B	5	middle

Based on the contents of the HFile above, you need to populate the index, following the approach described in the lecture slides. Use the following table (again, you can edit it by double-clicking). Use as many or as few rows as you need.

RowId	Column	Version

Exercise 4 — Thinking about the schema (optional)

A very important schema design question in HBase is the choice of the row key.

Imagine that you have a dataset containing:

- addresses of websites (URLs), potentially of all websites available online
- for each URL: the country in which the owner of the website is registered
- for each URL and for each country in the world: the number of visits to that URL from that country during the last month

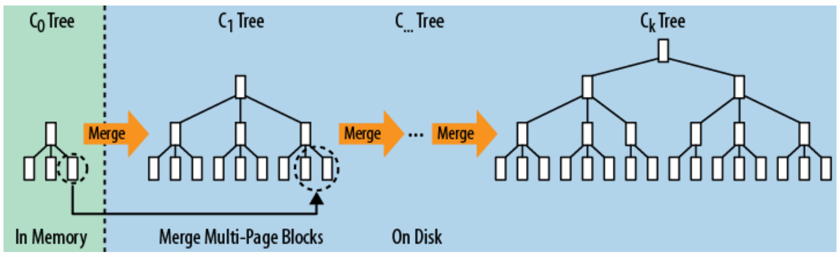
You plan to store this dataset in HBase. For each of the following queries, state what you think is the best choice for the row key:

1. Given a particular URL, count the total number of visits
2. Given a particular country, find the URL that is visited the most by the users from that country
3. Among all URLs whose owners are registered in a particular country, find the most visited one.

Exercise 5 — Log-structured mergetree (LSM tree) (optional)

We learn in the lecture that LSM tree is highly efficient in applications using wide column storage such as HBase, Cassandra, BigTable, LevelDB, where insertions in memory happen quite often. As opposed to B+-tree which has a time complexity of *O(log n)* when inserting new elements, *n* being the total number of elements in the tree, LSM tree has *O(1)* for inserting, which is a constant cost. In the reading of this week, you can find more on this topic.

The following figure is from the HBase Guide book where we see how a multipage block is merged from the in-memory tree into the next on-disk tree. Trees in the store files are arranged similar to B-trees. Merging writes out a new block with the combined result. Eventually, the trees are merged into the larger blocks.



Inserting data into LSM Tree:

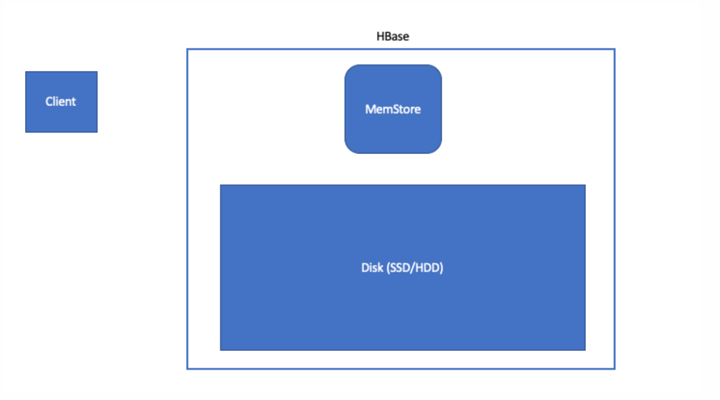
1. When a write comes, it is inserted in the memory-resident MemStore.
2. When the size of the MemStore exceeds a certain threshold, it's flushed to the disk.
3. As MemStore is already sorted, creating a new HFile segment from it is efficient enough.
4. Old HFiles are periodically compacted together to save disk space and reduce fragmentation of data.

Reading data from LSM Tree:

1. A given key is first looked up in the MemStore.
2. Then using a hash index it's searched in one or more HFiles depending upon the status of the compaction.

Deletes are a special case of update wherein a delete marker is stored and is used during the lookup to skip "deleted" keys. When the pages are rewritten asynchronously, the delete markers and the key they mask are eventually dropped.

We will now walk through a concrete exercise to understand how LSM tree works in HBase. Image we have a client who is constantly writing into HBase. The client also occasionally reads and deletes key value pairs in HBase. MemStore and disk are two storages we examine in this assignment.



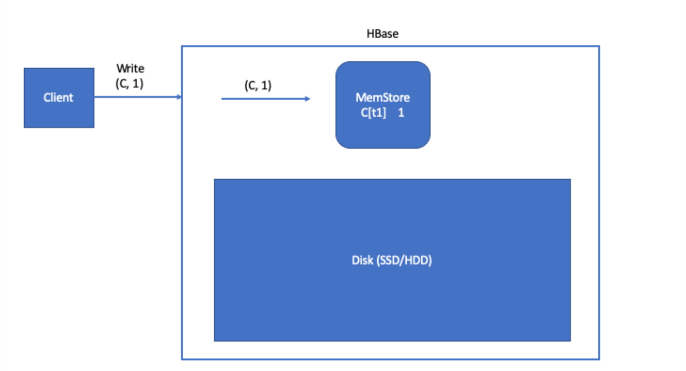
The client requests the following operations from HBase in sequence:

1. writing the following key value pairs into HBase: `(C,1), (B,2), (A,9), (A,109), (G,8), (D,67), (Z,0)`;
2. reading the value of key `A`;
3. deleting the key `Z`;
4. writing the following key value pairs: `(S,100), (Z1,900), (A1,9), (A01,1), (A11,1)`.

In the meanwhile, in HBase flush and compaction are conducted to optimize transfer.

Please draw the processes of

1. how the key value pairs are stored in HBase? To simplify the actual key in HBase, we use key[t] to denote the key. E.g., when the client writes the key value pair `(C,1)` into HBase, it is first stored in MemStore with the key value `C[t1] 1`.



2. how does HBase flush and compact the HFiles? Let us set the threshold of flush to three, i.e., when the key value pairs in MemStore have reached three, HBase will flush them to disk (from C_0 to C_1). Let us also set the threshold of compaction to three, i.e., if there exist two HFiles, each with three key value pairs, we have to compact them into a bigger HFile (from C_1 to C_2). The same rule applies for bigger HFiles: whenever there is of factor two some HFile at the level of C_{k-1}, compact them to the level C_k.