Big Data Exercises (Fall 2023) – Week 5 –

ETH Zurich

Wide Column Stores - HBase

This exercise will consist of four 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

Big Data Week 5

1. HBase & HFile organization
   - Example questions
2. Bloom filters
   - Derive and understand $\mathbb{P}(FP) = \left(1 - e^{-kn/m}\right)^k$
3. Log-structured merge (LSM) trees
   - Example problems

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:

   A. Select all article titles where the row name starts with '`1977`'
   B. Select all author names where the author contains the substring '`tom`'.
   C. Execute your queries on Question 2.A more than once and observe the query execution times.

3. Write the following queries:

   A. Return the number of articles from 2017.
   B. Return the number of articles that contain the word `Attacks` on them (case-insensitive). Please discuss different possibilities to formulate this query.

4. How can we use `SingleColumnValueFilter` to filter multiple columns at the same time? Here is the syntax: `scan '<table_name>', {FILTER=>"SingleColumnValueFilter('<column_family>','<column_qua` `<comp_operator>,'condition:<qualifier_value>') AND SingleColumnValueFilter('<column_family>','<column_qualifier>', <comp_operator>,'condition:<qualifier_value>')",COLUMNS=> ['column_family']}`. Write a query to return all the articles published in 2018, whose article title contains `Attacks` (case-insensitive).

5. What are the advantages and disadvantages of pure row stores?

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

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

8. What are the advantages and disadvantages of denormalization?

Solutions

1. `scan 'wiki_small', {STARTROW=>'100009', ENDROW=>'100011'}` returns five keys `1000092, 1000102, 1000104, 1000106, 1000108`, `scan 'wiki_small', {STARTROW=>'100015', ENDROW=>'100016'}` returns seven keys `100015, 1000151, 1000153, 1000155, 1000156, 1000158, 1000159`. Rows are sorted lexicographically by row key. You may

have to pad keys to get the sorting order you really want (without padding 10 is lexicographically smaller than 2).

2. The two queries:

   A. All article titles where the row name starts with '1977'. Two ways of formulating the queries: (1) `scan 'wiki_small', {COLUMNS => 'page:page_title', ROWPREFIXFILTER => '1977'}` OR (2) `scan 'wiki_small', {COLUMNS => 'page:page_title', FILTER => "PrefixFilter('1977')"}`.

   B. All author names where the author name contains the substring `tom`: `scan 'wiki_small', {COLUMNS => 'author:contributor_name', FILTER => "ValueFilter(=, 'substring:tom')"}`

   C. Execution times in Question 2.A.

      a. Queries with `ROWPREFIXFILTER` should be much quicker than `PrefixFilter` for Question 2.A, because the filter is applied to the row key rather than to the contents of columns.

      b. Subsequent invocations of the same command take less time due to caching.

3. The queries counting the rows.

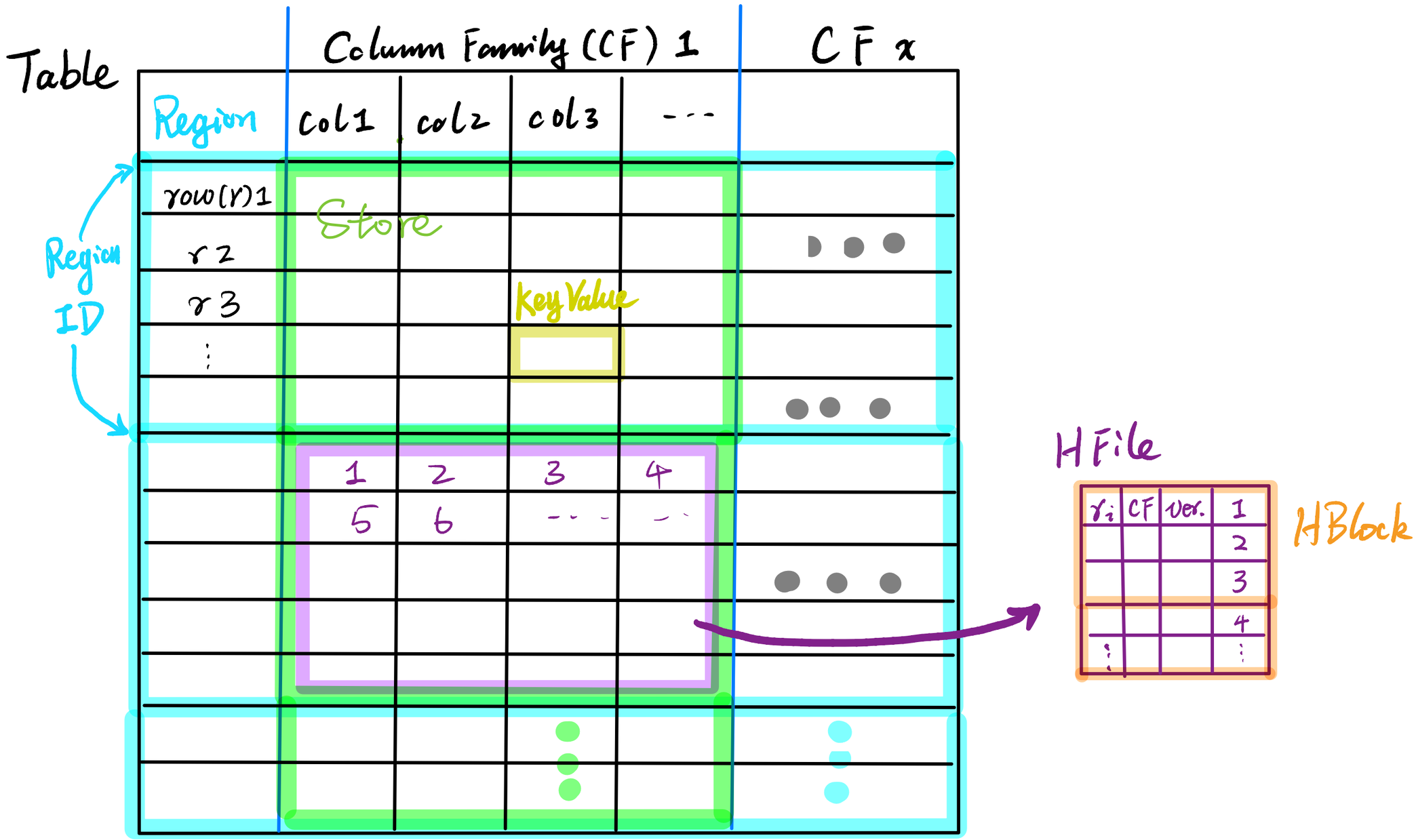
   A. Number of articles from 2017. First, see if we manage to get the articles from 2017 using the following query, limit the return rows to five. `scan 'wiki_small', {COLUMNS =>'author:timestamp', FILTER => "ValueFilter(=, 'substring:2017')", LIMIT=>5}` Then run `scan 'wiki_small', {COLUMNS =>'author:timestamp', FILTER => "ValueFilter(=, 'substring:2017')"}` which gives the returned row count **83647**.

   B. Return the number of articles that contain the word `Attacks` on them. Run this query `scan 'wiki_small', {COLUMNS =>'page:page_title', FILTER => "ValueFilter(=, 'substring:Attacks')"}` which returns 96 rows. Another way of formulating the query is `scan 'wiki_small', {FILTER => " SingleColumnValueFilter ('page', 'page_title', =, 'substring:Attacks') "}`

4. Multiple filters on different columns. Write a query to return all the articles titles published in 2017, whose article title contains `Attacks` (case-insensitive). `scan 'wiki_small', {FILTER=>"SingleColumnValueFilter('page','page_title',=,'substrir AND SingleColumnValueFilter('author','timestamp',=,'substring:2018')' ['page:page_title', 'author:timestamp']}`

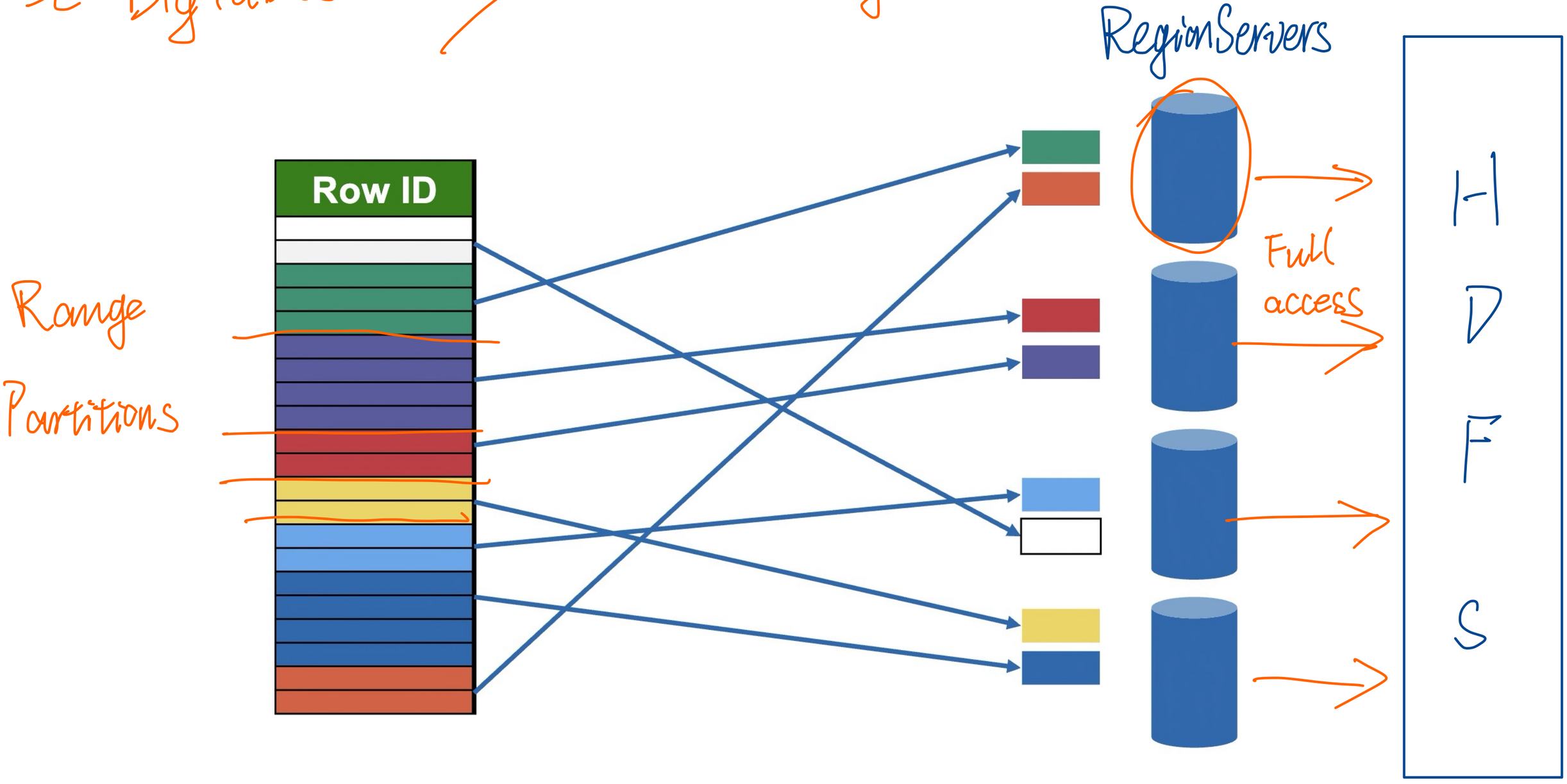
5. **Pure row store:**

HBase Physical Storage



HBase Architecture

≈ BigTable (+ ~~GFS~~ → Colossus) — Google



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

8. 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.

Task 3.1 — Inside a RegionServer

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

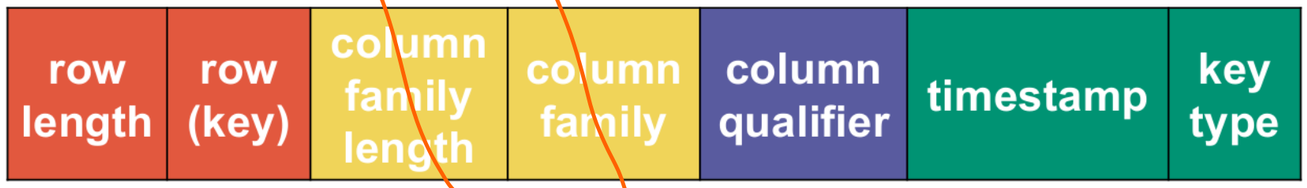
Imagine that we have an HBase table called '`phrases`', which has the following schema:

- 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 fields: 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.

Every where

1

MemStore

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

2

HFile indexes

The 4th column holds the ID of the HBase block

109	B	6	1	141	B	7	1	113	C	5	1
420	B	2	2	399	C	5	2	383	C	5	2
640	A	5	3	576	A	4	3	489	A	4	3
828	A	7	4	767	A	9	4	809	A	7	4

Main memory

File system (HDFS or other)

640

443 Bad

fseek(offset)

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	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

file

You can format your answer for this exercise as follows

```
get 'phrases', 'row_id'
```

Row	Column	Timestamp	Value	Where it came from (which HFile)
640	A	5	long	HF 1
640	B	6	live	HF 2
640	C	2	rock'n'roll	MS

Bloom Filters

Why typically $k > 1$?

Eg. cash collision rate $\rho = 1/2$

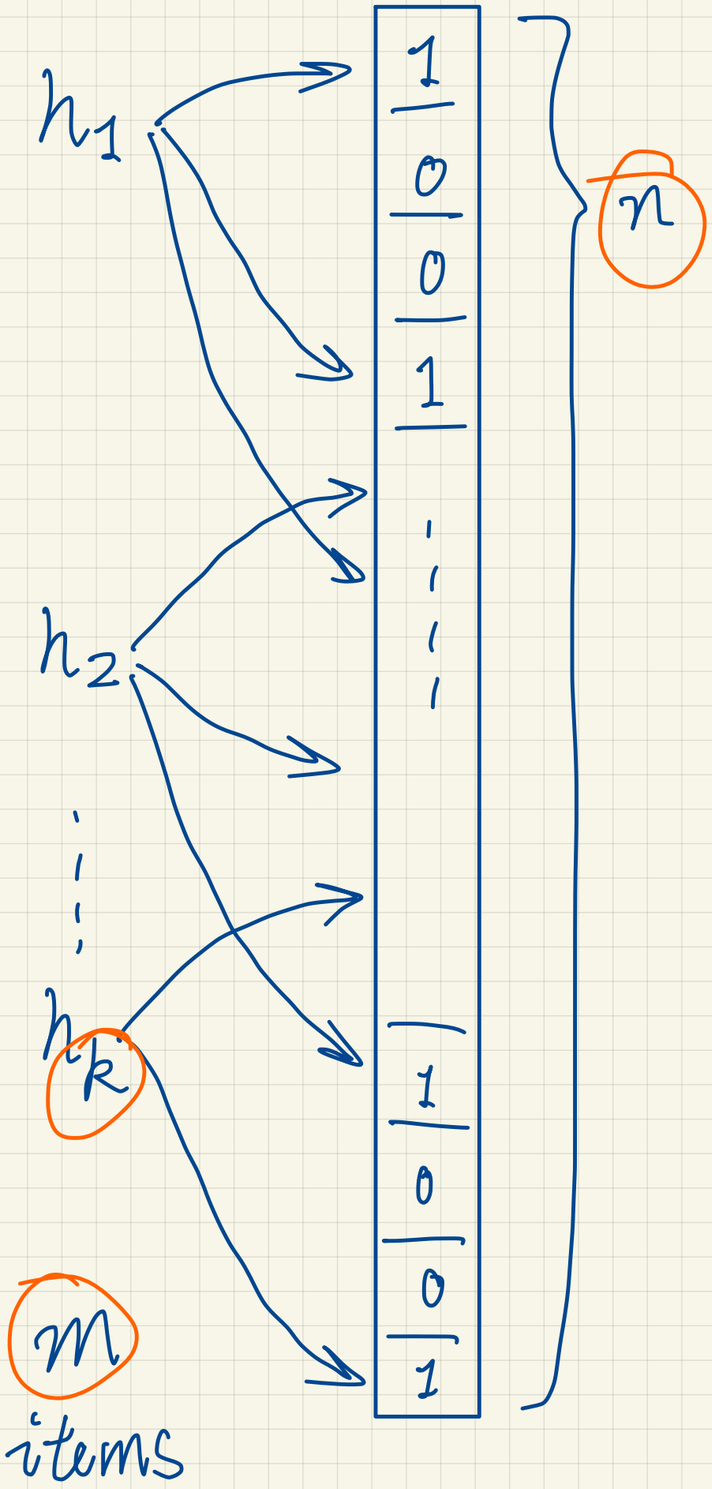
$\Rightarrow$ 10 collisions : $\rho^{10} < 0.1\%$

$\Rightarrow$ Driving down ρ exponentiontially!

① How to calculate FP

② Simplification of ① with e

③ How to choose k with fixed m, n



① Calculate FP

- Probability of a bit being 0 after m items have been inserted?

$$p_0 = \left(1 - \frac{1}{n}\right)^{m*k}$$ ⇝ total # of times a bit could've been set to 1.

$p_1 = 1 - p_0$: Prob of 1 bit being set by ANY of k funcs

- Probability of a FP case $\Leftrightarrow \mathbb{P}$(ALL k hash funcs set a bit to 1.)

$$p_1^{\,k} = (1 - p_0)^k = \left(1 - (1 - 1/n)^{mk}\right)^k = \mathbb{P}(FP)$$

⇓

Hard to consistently analyse.

⇓

Simplify

② Simplification of P(FP) with e

- $(1-\frac{1}{n})^{mk} = e^{[\ln(1-\frac{1}{n})^{mk}]}$

$$= e^{mk * \ln(1-\frac{1}{n})} \quad (\star)$$

$n = 1000$
(1K entries)

$$-\frac{1}{1K} - \frac{1}{2M} - \frac{1}{3B} \cdots$$ small

$\Rightarrow \ln(1-\frac{1}{n}) \approx -\frac{1}{n}$

$(\star) \approx e^{-\frac{1}{n} * mk}$

$\ln(1+x)$ where $x = -1/n$

$\Rightarrow$ Approximate using Taylor expansion

$\Rightarrow (x) - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$

(Mercator series)

$\Rightarrow P(FP) \approx (1 - e^{-\frac{mk}{n}})^k$

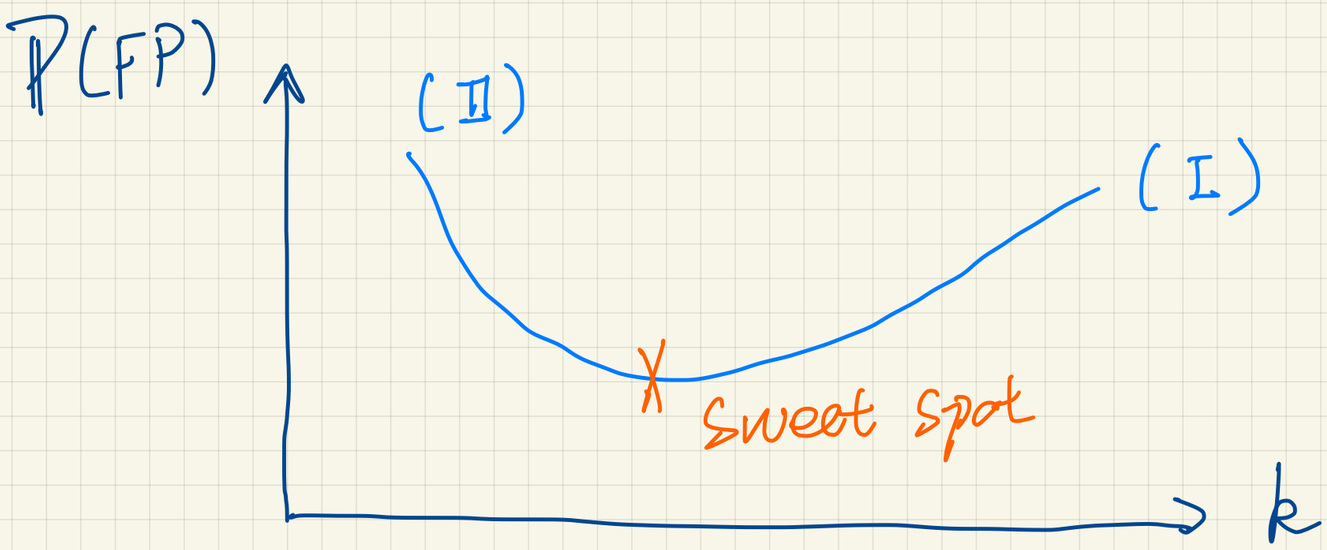
③ Optimize for k

$$\mathbb{P}(FP) \approx \underbrace{\left(1 - e^{-\frac{m}{n}k}\right)}_{p_1}{}^{k}$$

$\frac{m}{n}$ = # items per slot ⇓ fix ☰

(I) Too many hash funcs (large k): $(p_1 \approx 1)^k \rightarrow$ high $\mathbb{P}(FP)$

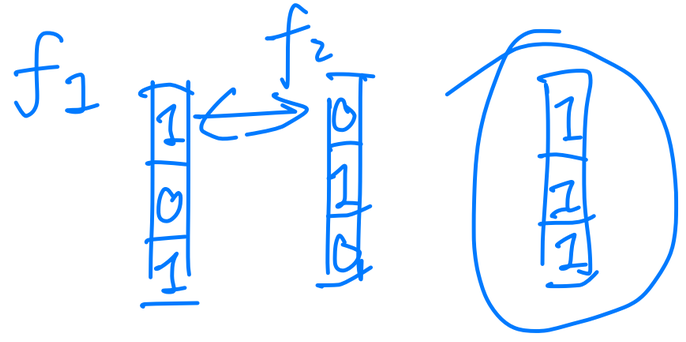
(II) Too few hash funcs (small k): $p_1{}^{\cancel{k}} \rightarrow$ No exponential effect $\rightarrow$ high $\mathbb{P}(FP)$



$$\frac{\partial \mathbb{P}(FP)}{\partial k} = 0$$

$$\Rightarrow k^* = \ln 2 * \frac{n}{m}$$

$\frac{n}{m}$ = bits/item



Task 3.2 — Bloom filters

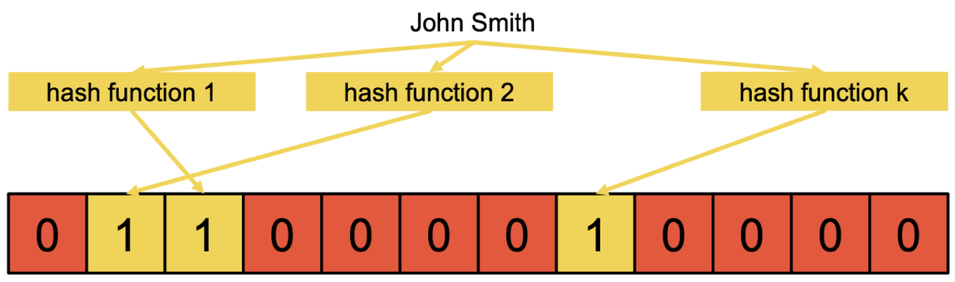
Let's start with summarizing what we have in memory when working with HBase: MemStore, LRU (Least Recently Used) BlockCache, Indices of HFiles, and Bloom filters. We did not have time to cover Bloom filters in the lecture, but they are actually very crucial in avoiding disk reads if we can guarantee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

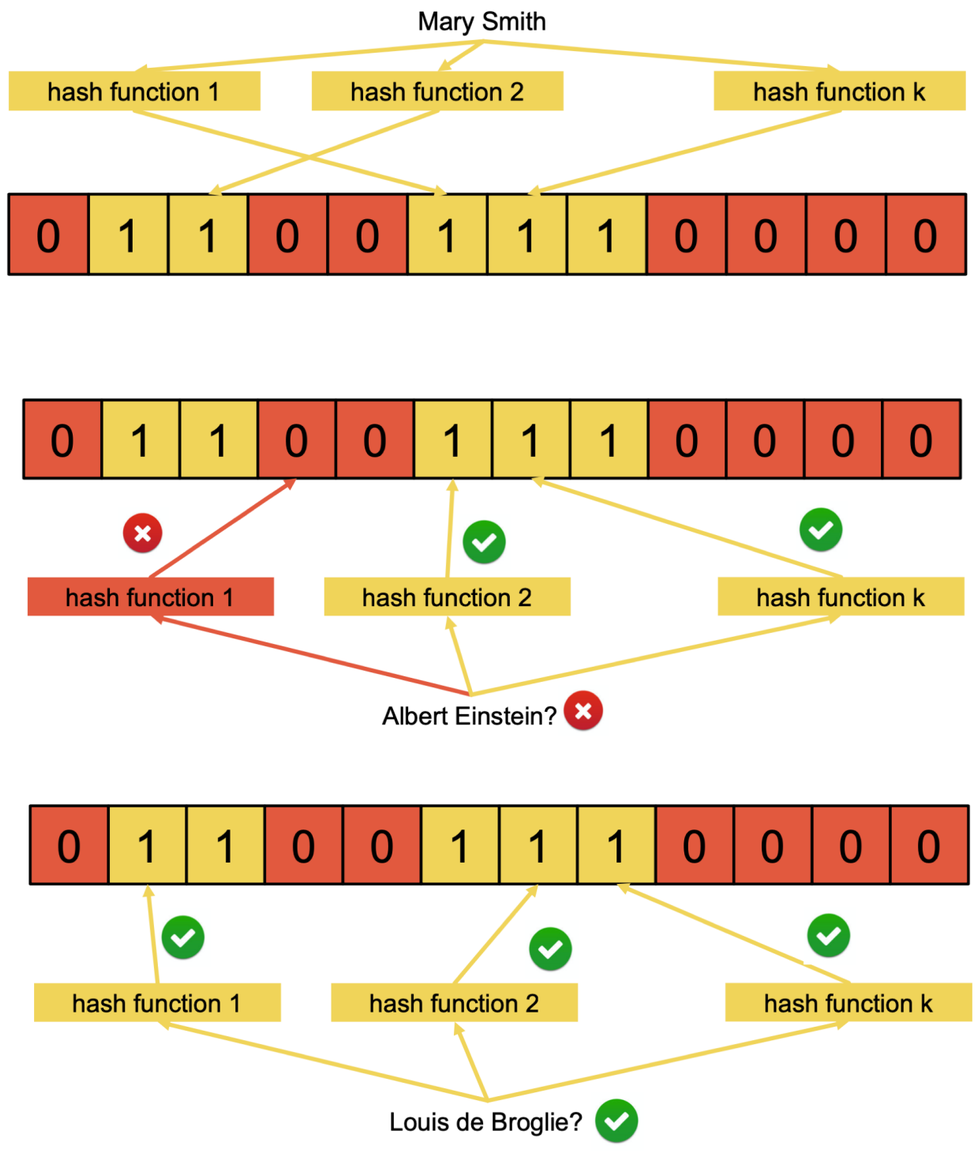
Bloom filters are a data structure used to speed up queries, useful in the case in which it's likely that the value we are looking doesn't exist in the collection we are querying. Their main component is a bit array with all values initially set to 0. When a new element is inserted in the collection, 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 `000101100000`. 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 `010000110000` is then a false positive for the set `{the Smiths}`.





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:

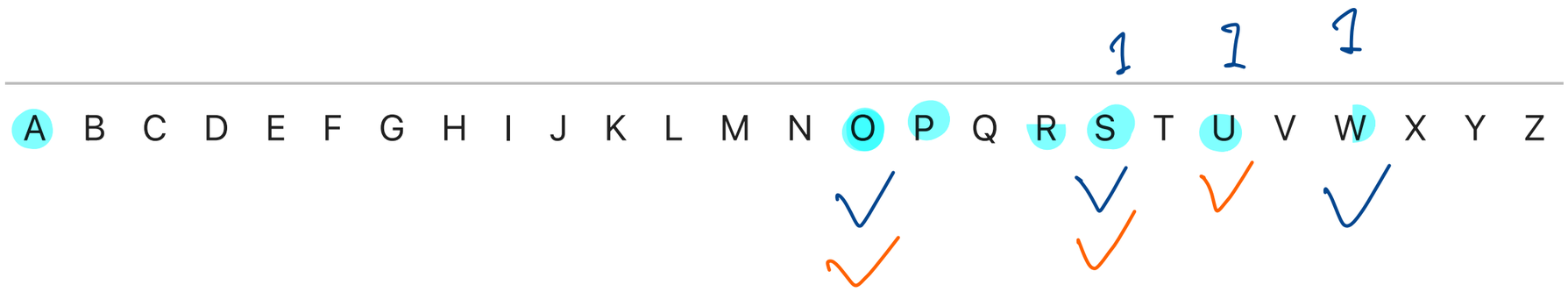
Here, only values "1" are displayed to avoid cluttering the view; thus, if a cell is empty, it is assumed to hold a "0".

Here is what we will do next.
Step 1. Create a bloom filter that for a collection of words (Collection A).
Step 2. Given this bloom filter, check if a set of new words (Collection B) are members of Collection A.

Step 1. 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): [**Collection A**]

Word	hash1	hash2	hash3
round	R	O	U
sword	S	W	O
past	P	A	S
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): [**Bloom Filter for Collection A**]



2. 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): [**Collection B**]

Word	hash1	hash2	hash3	The Bloom filter says the word belongs to the set: (yes/no)
sword	S	W	O	
sound	S	O	U	FP
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)?

FN X Never ever FN = 0

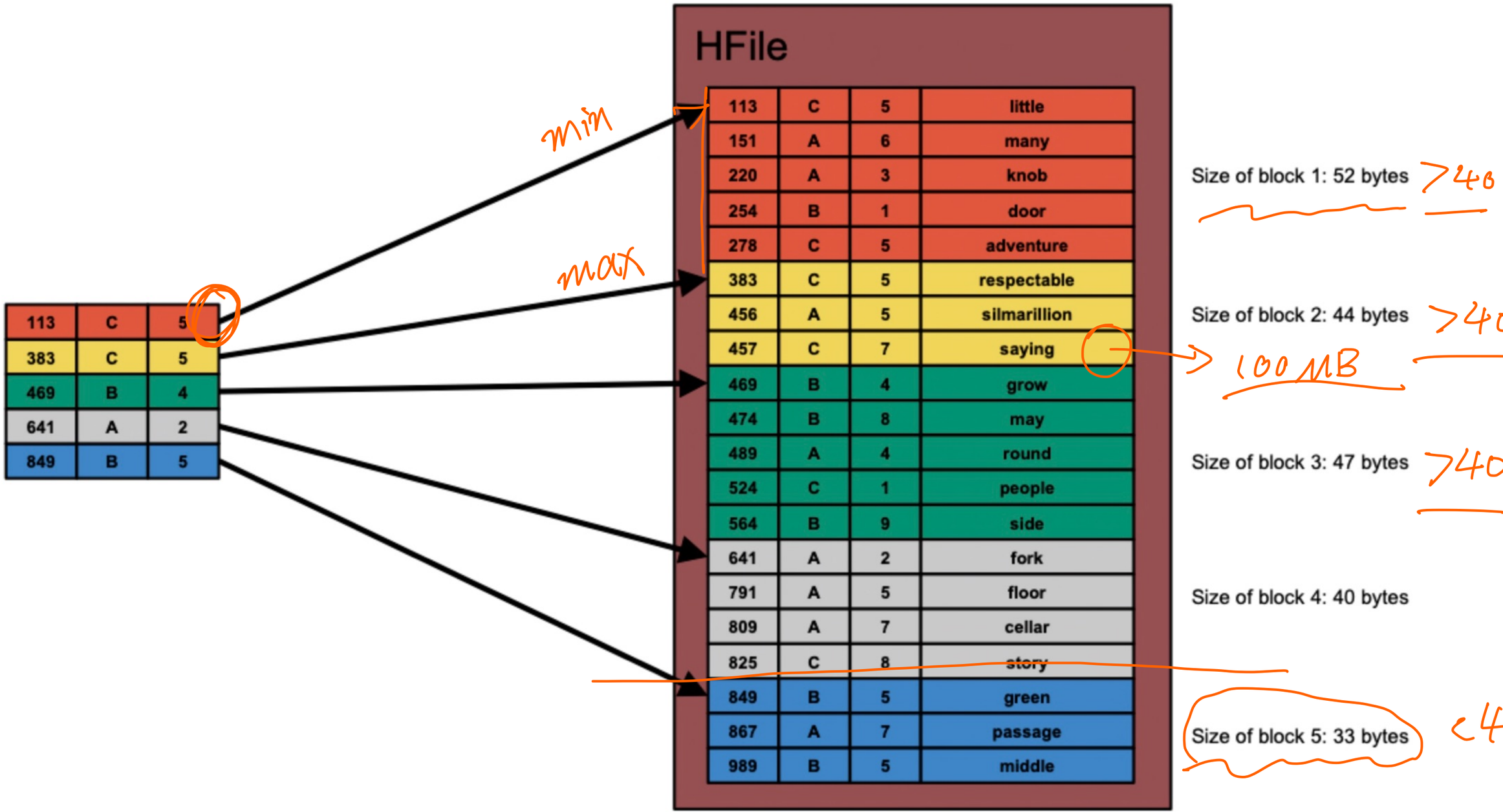
Task 3.3 — Building an HFile index

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.

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.



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

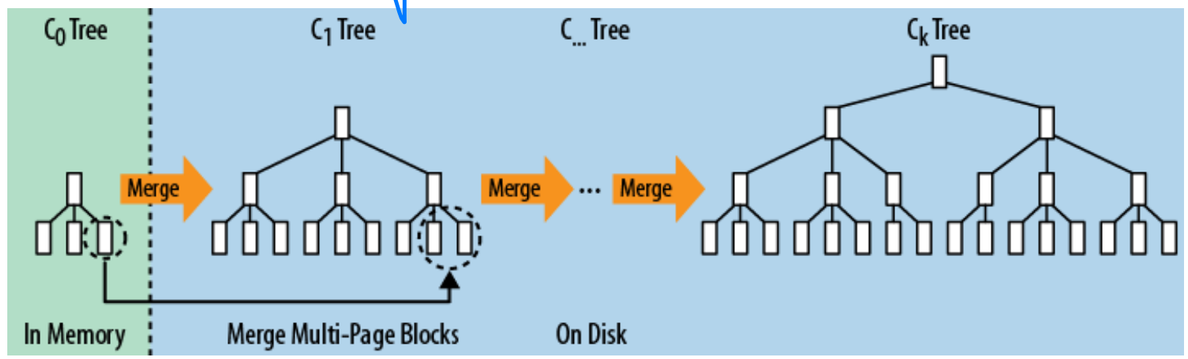
2. Given a particular country, find the URL that is visited the most by the users from that country
3. Among all URLs whose owners are registered in a particular country, find the most visited one.

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

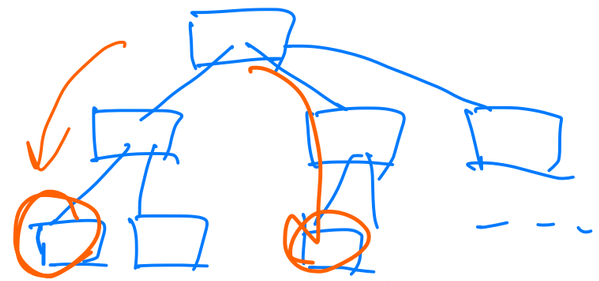
We learn in the lecture that LSM tree is highly efficient in applications using wide column storage such as HBase, Cassandra, BigTable, LevelDB, where insertions in memory happen quite often. As opposed to B+-tree which has a time complexity of *O(log n)* when inserting new elements, *n* being the total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

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.

Backbone of NoSQL



SQL



+ Optimized for reads

– Bad at injecting data

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.

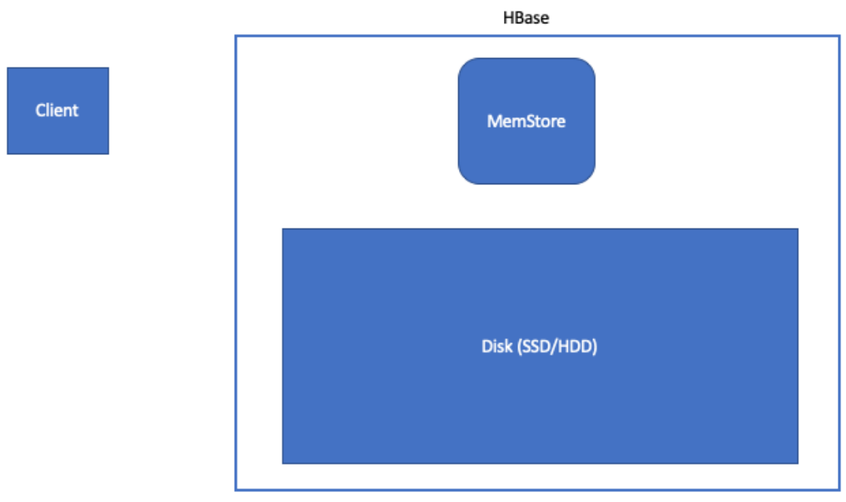
Reading data from LSM Tree:

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:

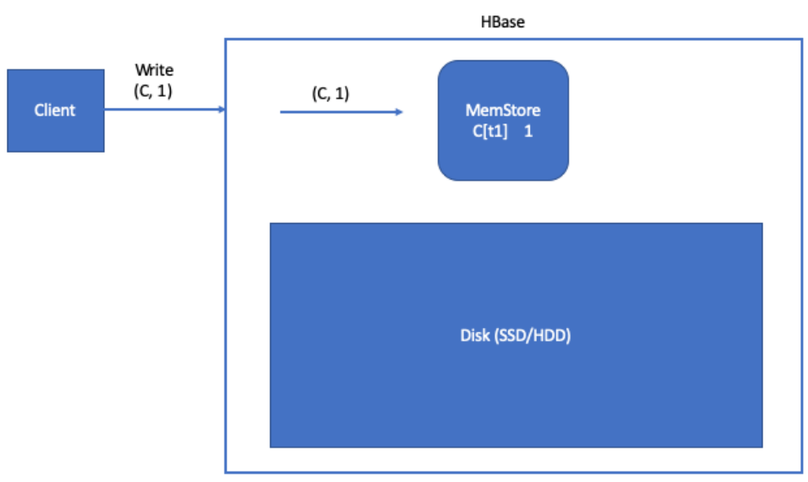
1. writing the following key value pairs into HBase: `(C,1), (B,2), (A,9), (A,109), (G,8), (D,67), (Z,0)`;
2. reading the value of key `A`;
3. deleting the key `Z`;

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.

In [ ]: