6.1
Bloom Filters

Reading:
[Kl], chapter 3
[Pe], chapter 7

LSM Storage (Log-Structured-Merge TREE Storage)

Fast and efficient write operations:

- keys are sorted in memory (Red-Black-Tree)
- only sequential writes to disk (Red-Black-Tree flushed to disk into Sorted String Tables as soon as tree reaches a certain size)
- SSTables are immutable (no random writes to disk, no writes-in-place on disk)
- Updates are treated as inserts - unless key is still in memtable (no random writes to disk, no writes-in-place on disk)
- Deletions are treated as (specific) inserts (no random writes to disk, no writes-in-place on disk)

Downside

- obsolete key-value pairs accumulate quickly (becasue of the duplicate keys in different files)
  → a lot of data garbage in the files
- read operations quickly deteriorate and slow down sharply (because possibly for one read all files have to be searched)

→ Regular compaction is needed in order to get rid of obsolete key-value pairs we need to do compaction

→ Optimization for read operations needed for one read we would need to search all segment files so we need to find optimization for this problem

eva.knirsch@kiu.edu.ge

Compaction

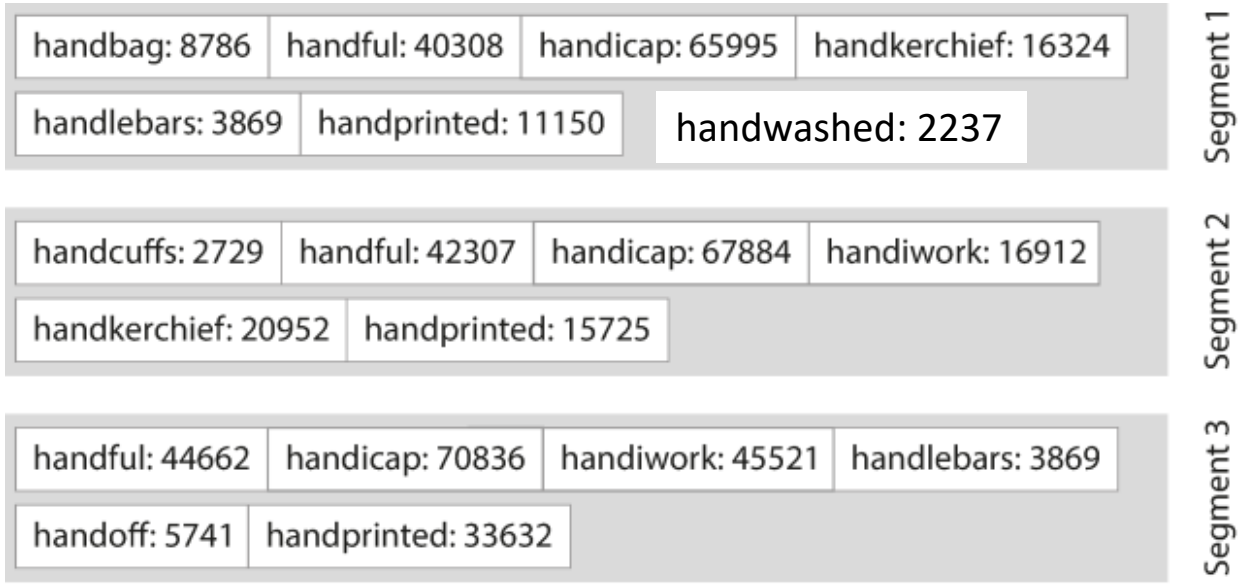
achieves:

remove obsolete key-value pairs --> reduce disk access

sort keys of overlapping keys in segments (so basically when we have several segments which are sorted in their ways but not among each other, we need to get one segment where everything from each segment will be sorted)

eva.knirsch@kiu.edu.ge

LSM-Storage: Read Operations



get(key = handwashed)

How is this read executed?

- load (1st page of) most recent file (segment n) from disk and check smallest key to see whether key COULD in in the file.
  - If yes → search **whole segment n** for key → if found: return key-value pair
  - If no or if key not found in segment n,
- load (1st page of) segment n-1 from disk and check whether key could be in file
  - If yes → search **whole segment n-1** for key → if found: return key-value pair
  - If no or if key not found in segment n-1,
- load (1st page of) segment n-2 from disk and check whether key could be in file.
- ..... continue until key is found or all segments are searched.

so first we check if handwashed could be in segment 3 where smallest key is handful. since handwashed>handful, it can be in segment 3 so we search for it in whole segment 3. Since our key isn't in segment 3, we move on to segment 2, handwashed>handcuffes, so it could be in this segment and we search for our key in this segment. It is not found there so we move on to segment 1. Since handwashed>handbag, it could be in segment 1 and we search for it in this segment. It is found in segment 1 and handwashed: 2237 is returned.

so in the worst case we have to go through all the files and load them completely in memory

Example taken out of [KI], p. 76

eva.knirsch@kiu.edu.ge

LSM Storage Read Operations: Sparse Indexes

The read process outlined is extremely inefficient.

→ for each segment a sparse index is kept in memory.

→ typically: the smallest key of each page of the file is kept in memory

What is a sparse index?

subset of the full index

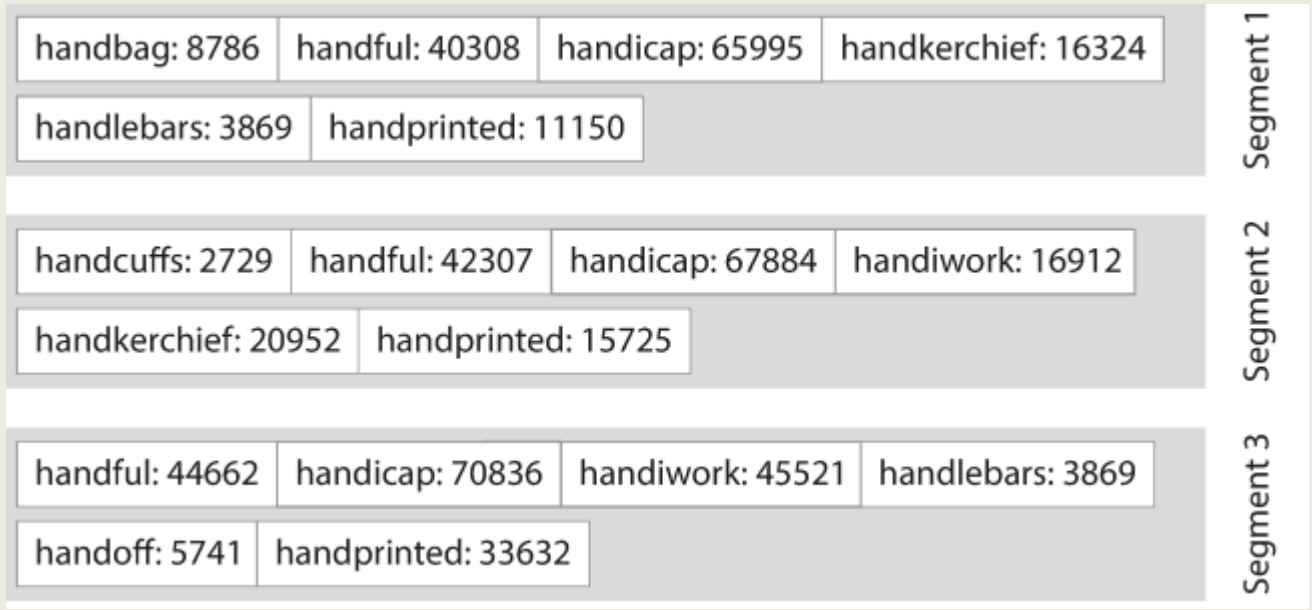
for btrees we have full index which store all keys, because we need to know where each tuple is located.

What optimization is achieved using sparse indexes for each segment?

instead of loading the whole file, only 1 page needs to be loaded.

eva.knirsch@kiu.edu.ge

LSM-Storage: Searches / Read Operations



get(key = handwashed)

How is this read executed?

first we look at segment 3 and find if this key can be there and on which page. It can be on page 1 bacause handprinted<hanwashed<handwritten. So we only need to load page 1. Not found, so we move on to segment 2, it can be on page 2, we load this page and we find handwashed.

Index for Segment 1

key	page
handbag	0
handprinted	1
handsome	2
handwritten	3
...	

Index for Segment 2

key	page
handcuffs	0
handprinted	1
handwashed	2
handyman	3
...	

Index for Segment 3

key	page
handful	0
handprinted	1
handwritten	2
handzipped	3
...	

Example adapted from [KI], p. 76

eva.knirsch@kiu.edu.ge

LSM-Storage: Searches / Read Operations

get(key = handbag)
How is this read executed?

Index for Segment 1

key	page
handbag	0
handprinted	1
handsome	2
handwritten	3
...	

Index for Segment 2

key	page
handcuffs	0
handprinted	1
handwashed	2
handyman	3
...	

Index for Segment 3

key	page
handful	0
handprinted	1
handwritten	2
handzipped	3
...	

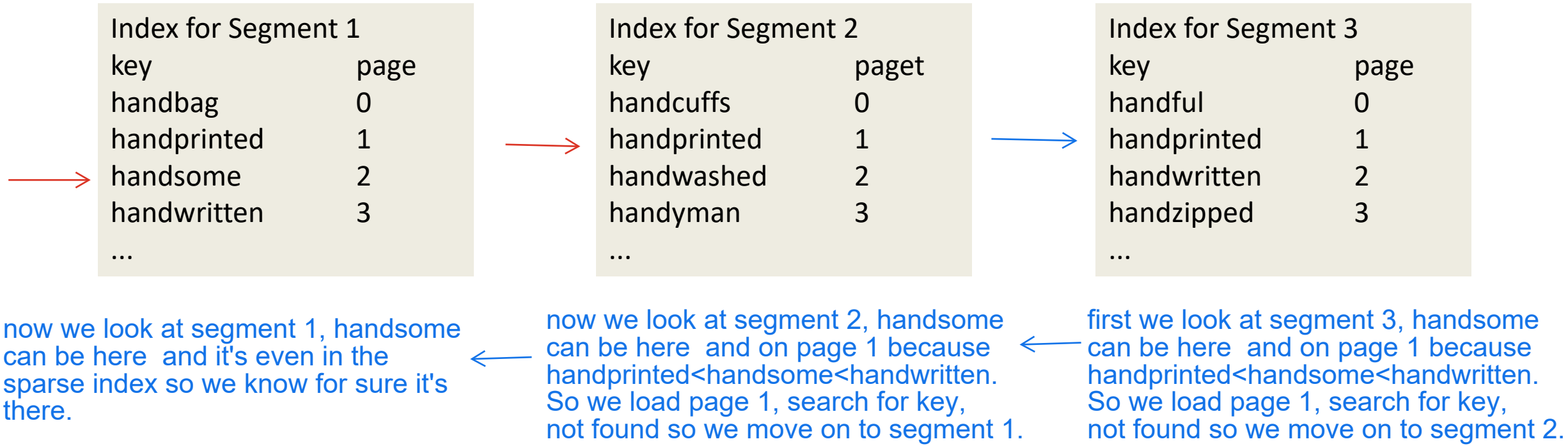
lastly we look at segment 1, smallest key here is handbag so we know that this key is here and we load page 0.

then we look at segment 2 and handbag also can't be here since handbag<handcuffs

first we look at segment 3, handbag can't be here because handbag<hanful

LSM-Storage: Searches / Read Operations

get(key = handsome)
How is this read executed?



Read Operations using Bloom Filters

- The sparse index tells whether a key COULD be on a certain page of a segment file.
  → it still forces the system to load all pages on which the key COULD be.
- If the system knew first that a key is definitely NOT on a page this would minimze costly disk access considerably.
  → this is what a Bloom filter does

Index for Segment 1

key	page
handbag	0
handprinted	1
handsome	2
handwritten	3
...	

Index for Segment 2

key	page
handcuffs	0
handprinted	1
handwashed	2
handyman	3
...	

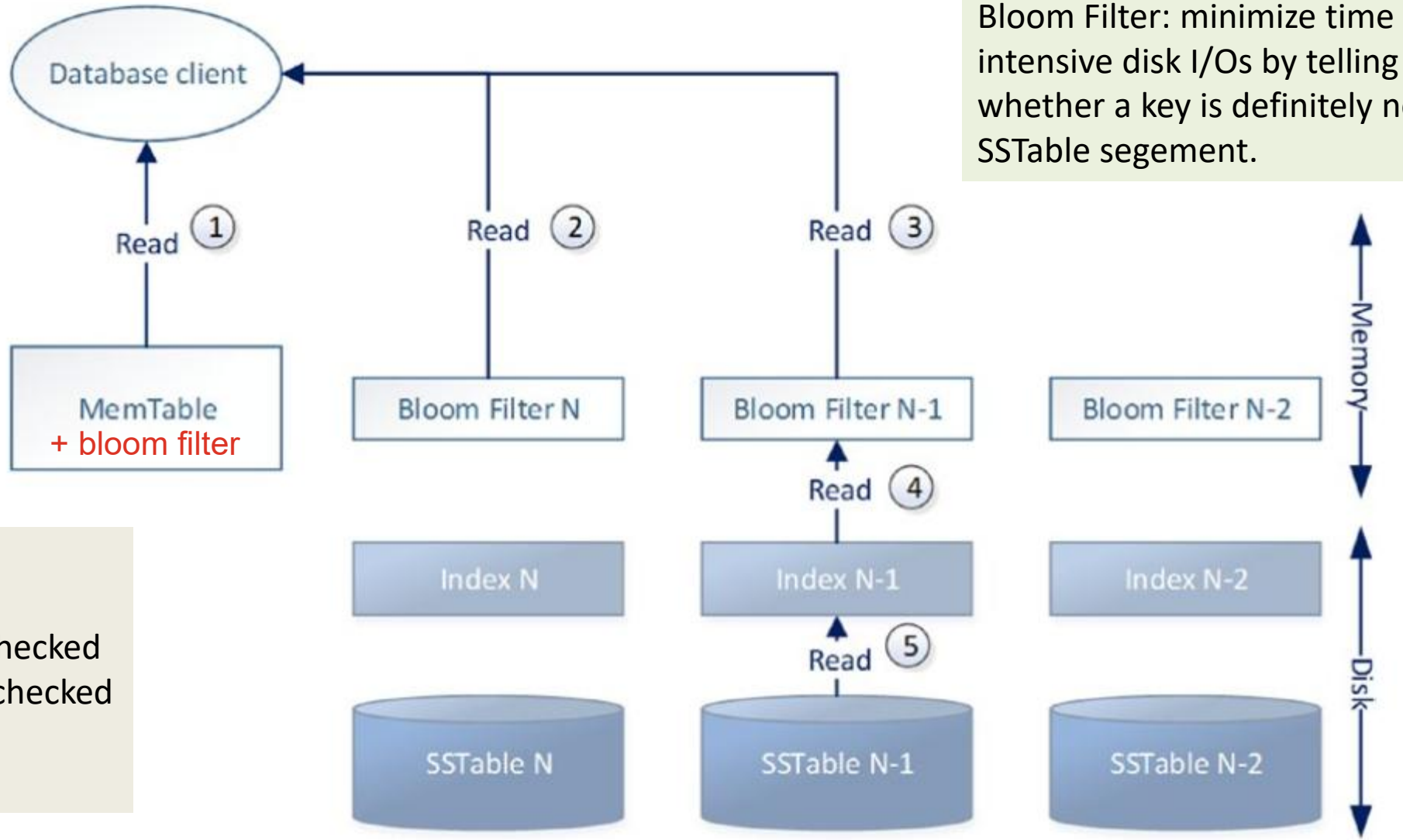
Index for Segment 3

key	page
handful	0
handprinted	1
handwritten	2
handzipped	3
...	

- Provided that the key 'handsome' is only in segment 1, a Bloom filter would reduce disk access from 3 disk accesses to one disk access.

eva.knirsch@kiu.edu.ge

Bloom Filter: minimize time intensive disk I/Os by telling whether a key is definitely not in a SSTable segement.



Read Process:

1. Bloom filter is checked
2. Sparse index is checked
3. Disk access

[Ha], p161

Figure 10-12. *Log-structured merge tree reads (Cassandra terminology)*

eva.knirsch@kiu.edu.ge

LSM-Storage: Bloom Filters

- The search can be optimized if the system can exclude from the beginning all segments in which the key is definitely not present (even though the index says that it could be present in the segment).

- This is exactly what a Bloom filter does: A Bloom filter is a data structure that can tell if an element is definitely not in a set. It cannot tell whether a key is psotively in a set or not.

- A Bloom filter in LSM can tell whether a key is definitely not in a set.

    - **True negative:** means that the Bloom filter returns correctly that **$k \notin S$.**
    - **False negative:** would mean that the Bloom filter returns **$k \notin S$ when actually $k \in S$.** ✗
      False negatives do not happen with Bloom filters.
    - **True positive:** means that the Bloom filter correctly returns $k \in S$
    - **False positive:** means that the Bloom filter wrongly returns $k \in S$

When the system uses a Bloom filter, this is checked first, then the sparse indexes.

eva.knirsch@kiu.edu.ge

Bloom Filter

A bloom filter performs a membership test: Is a certain element (key-value pair) member in a certain set (segment)?

The bloom filter is yet another data structure in the LSM-tree storage model. So, it is expected to be fast and space-efficient (resides in memory).

The bloom filter is a probabalistic data structure. It can tell that

- a certain key is definitely NOT a member in a specific data set or
- a certain key may be a member in a specific data set.

The probabalistic nature (and with such potentially false positive results of the membership test) is accepted in order to achieve speed and space-efficiency.

The Bloom filter performs the membership test using hash functions.

eva.knirsch@kiu.edu.ge

Boom Filter Hash Functions

- A hash function converts variable-length keys to fixed-length hash values.
- Hash values should be evenly distributed over the keyspace.
- The hash function should prevent or minimize collisions.

Bloom filter hash function do NOT do collision management. They just need to be fast. Collisions are accepted in order to achieve speed and space efficiency.

A bloom filter for one data set (in our case: one segment) consists of

1. one bit array storing the hash values
2. k hash functions
    1. need to be fast
    2. are independent → can be computed in parallel
    3. need to spread evenly over keyspace (which is the bit array)
    4. k is a constant → insertions into the bloom filter have constant time complexity O(1)

eva.knirsch@kiu.edu.ge

Boom Filter Example

- Our bitarray has 20 bits, initially always all set to 0

  B = 0000 0000 0000 0000 0000

- We use 3 hash functions:

  h1(x) = x mod 20
  h2(x) = 3x mod 20
  h3(x) = 7x mod 20

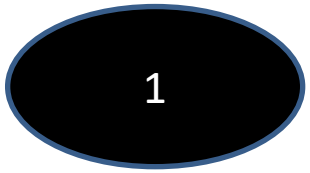
eva.knirsch@kiu.edu.ge

Inserts

Our segment has 3 keys:

1: some value
12: some value
7: some value

Remember that integer keys may not be best solution for your key-value store (but good for our simple example)



B = 0000 0000 0000 0000 0000

h1(1) = 1 mod 20 = 1
h2(1) = 3 mod 20 = 3
h3(1) = 7 mod 20 = 7

system flips corresponding bits from 0 to 1.

B = 0101 0001 0000 0000 0000

so we flipped 0s on indexes: 1,3,7

- Our example bit array has 20 positions.
- The hash values spread - "randomly", evenly - over the bit array.
- Each hash value "flips" the bit from 0 to 1.

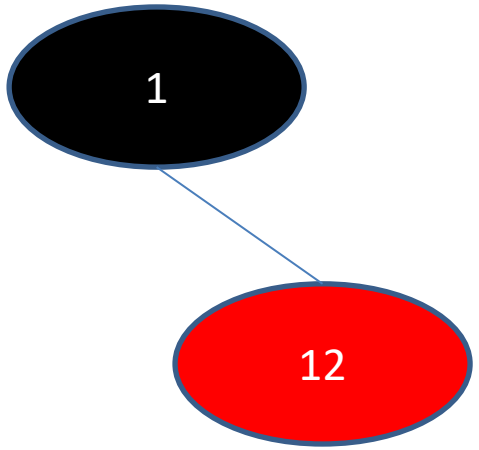
eva.knirsch@kiu.edu.ge

Inserts

1: some value

12: some value

7: some value



B = 0101 0001 0000 0000 0000

h1(12) = 12 mod 20 = 12
h2(12) = 36 mod 20 = 16
h3(12) = 84 mod 20 = 4

B = 0101 0001 0000 0000 0000

What does B look like aftre inserting 12?

we are flipping on indexes: 12,16,4

B = 0101 1001 0000 1000 1000

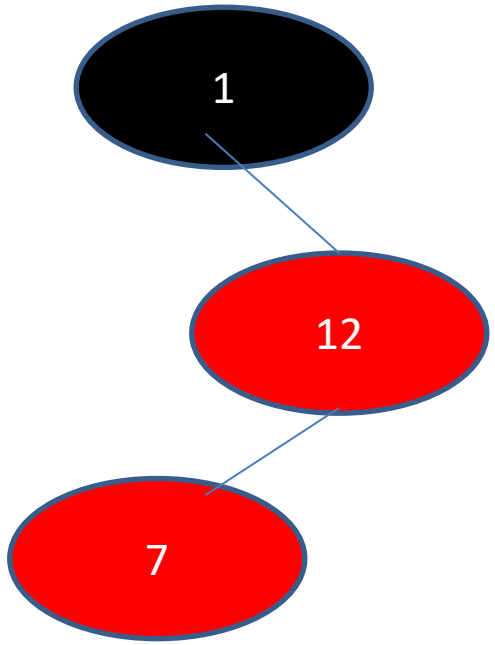
eva.knirsch@kiu.edu.ge

Inserts

1: some value

12: some value

7: some value



h1(12) = 12 mod 20 = 12
h2(12) = 36 mod 20 = 16
h3(12) = 84 mod 20 = 4

B = 0101 1001 0000 1000 1000

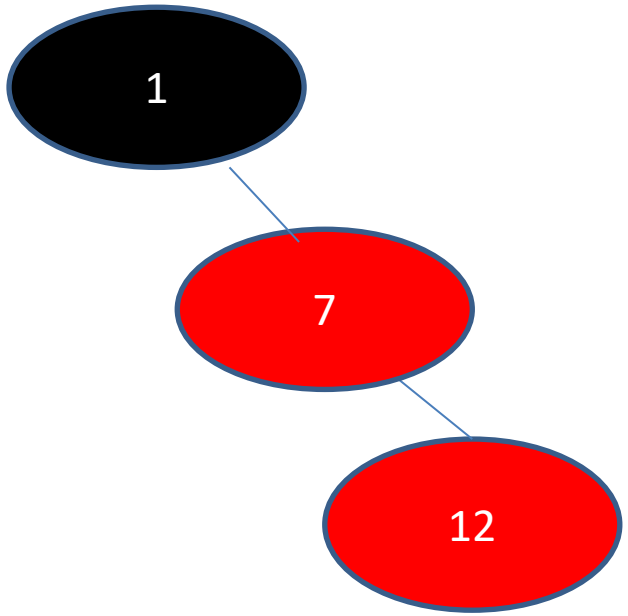
eva.knirsch@kiu.edu.ge

Inserts

1: some value

12: some value

7: some value



h1(12) = 12 mod 20 = 12
h2(12) = 36 mod 20 = 16
h3(12) = 84 mod 20 = 4

B = 0101 1001 0000 1000 1000

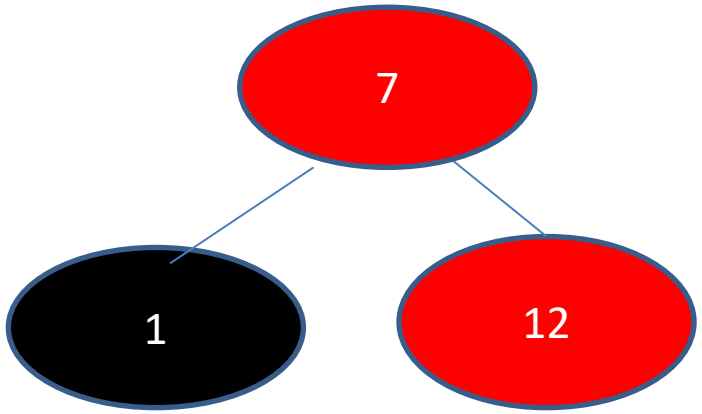
eva.knirsch@kiu.edu.ge

Inserts

1: some value

12: some value

7: some value



h1(12) = 12 mod 20 = 12
h2(12) = 36 mod 20 = 16
h3(12) = 84 mod 20 = 4

B = 0101 1001 0000 1000 1000

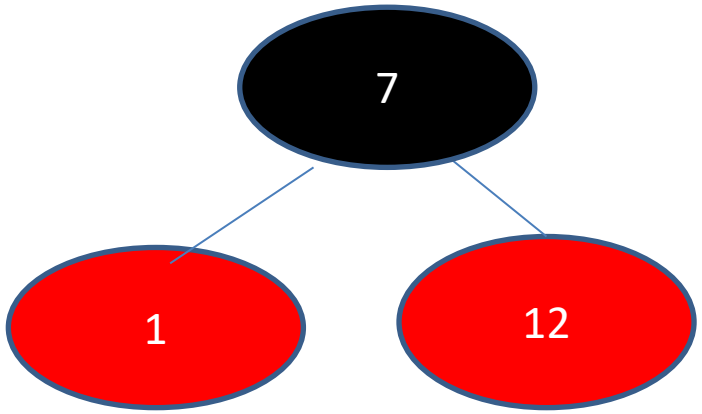
eva.knirsch@kiu.edu.ge

Inserts

1: some value

12: some value

7: some value



B = 0101 1001 0000 1000 1000

h1(7) = 7 mod 20 = 7
h2(7) = 21 mod 20 = 1
h3(7) = 49 mod 20 = 9

B = 0101 1001 0000 1000 1000

What does B look like aftre inserting 7?

we have to flip on indexes: 7,1,9. If it's already flipped then we remain that, so we never flip from 1 to 0.

B = 0101 1001 0100 1000 1000

- Memtable is flushed to SortedStringTable on disk.
- Memtable in memory is discarded.
  (Again: Rememer that typically a memtable has a fixed size, between 1 – 4 MB – and not only 3 keys.)
- Bloom filter is kept in memory.

basically if it's already 1 we are rewriting with another 1.

collision

eva.knirsch@kiu.edu.ge

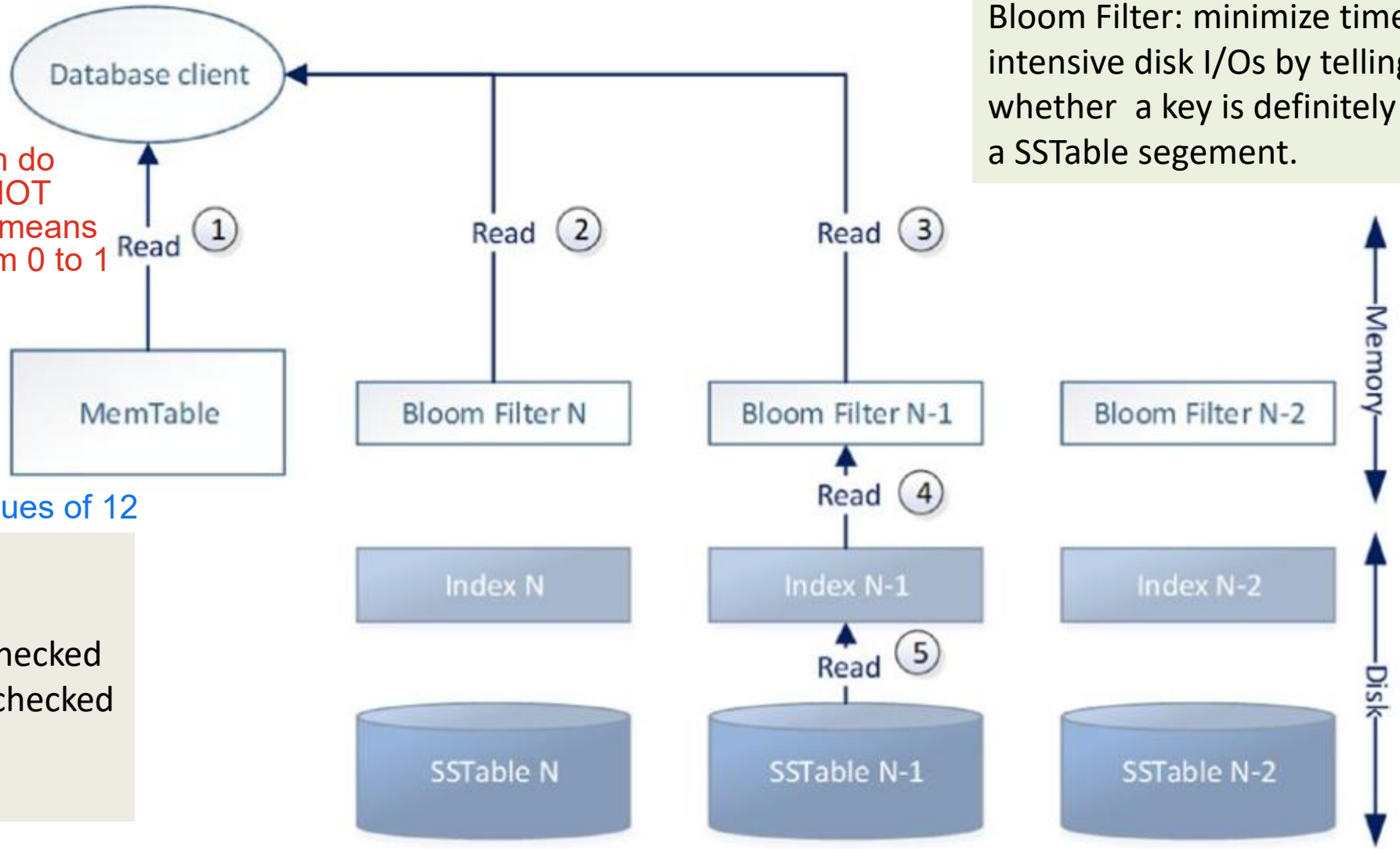
Bloom Filter: minimize time intensive disk I/Os by telling whether a key is definitely not in a SSTable segement.

with bloom filter we can do insertions and reads, NOT delete because delete means that we need to flip from 0 to 1 which is not possible.

example of deletion:
12 something
--> 12 tombstone
--> bloom filte: hash values of 12

Read Process:

1. Bloom filter is checked
2. Sparse index is checked
3. Disk access



[Ha], p161

Figure 10-12. *Log-structured merge tree reads (Cassandra terminology)*

each segment has its own bit array

eva.knirsch@kiu.edu.ge

Bloom Filter Search

B = 0101 1001 0100 1000 1000

get(key=2)

h1(2) = 2 mod 20 = 2 → bit = 0
h2(2) = 6 mod 20 = 6 → bit = 0
h3(2) = 14 mod 20 = 14 → bit = 0

Key = 2 is NOT in the set (segment)

since bits on indexes: 2,6,14 are all zero then this key is inot in the segment

eva.knirsch@kiu.edu.ge

Bloom Filter Search

B = 0101 1001 0100 1000 1000

get(key=4)

h1(4) = 4 mod 20 = 4 → bit = 1
h2(4) = 12 mod 20 = 12 → bit = 1
h3(4) = 28 mod 20 = 8 → bit = 0

If at least one bit = 0, the key is NOT in the set
Key = 4 is NOT in the set (segment)

eva.knirsch@kiu.edu.ge

Bloom Filter Search

B = 0101 1001 0100 1000 1000

get(key=12)

h1(12) = 12 mod 20 = 12 → bit = 1
h2(12) = 36 mod 20 = 16 → bit = 1
h3(12) = 84 mod 20 = 4 → bit = 1

Key = 12 MAY BE in the set (segment)

since all the bits on indesxes: 12,16,4 are 1s then this key may be in segment.

How would the read process for get(key=12) continue?

so this tells us what segment are we looking at and then we would need to go to sparse index of that segment.

eva.knirsch@kiu.edu.ge

Bloom Filter Search

B = 0101 1001 0100 1000 1000

get(key=9)

h1(9) = 9 mod 20 = 9 → bit = 1
h2(9) = 21 mod 20 = 1 → bit = 1
h3() = 63 mod 20 = 3 → bit = 1

Key = 9 MAY BE in the set (segment).

But we know that it is a false positive.

False positives are the result of collisions.

in order to get fewer colissions we can increase our bit array.

How would the read process for get(key=9) continue?

what segment are we looking at--> sparse index of that segment

eva.knirsch@kiu.edu.ge

Bloom Filter Use Cases

1. Key-Value / Wide-Column Databases:
   Use Bloom filters to reduce costly disk access for read operations.

2. Routing (deciding where to forward)

3. Check against a "blacklist"

    1. Chrome uses bloom filters to check against malicious URLs
    2. Can be used exlude objects from being recommended
    3. Police / Border control checks against a wanted list

Some of the use cases can tolerate a considerable false-positive rate. A false positive in a database search slows down the search but does not any other harm besides making the read inefficient.
→ Bloom filters in Key-Value databases are typically implemented with a false positive rate between 10 – 15 %.

Other use cases require a lower false positve rate. How can this be achieved? Which parameters do we need to adjust

we need to extend bit array to reduce false positives, but hash function will get slower.

eva.knirsch@kiu.edu.ge

Bloom Filter False_Positive_Rate (fp_rate) Calculation

Required: Number of elements in set (segment)

Desired False-Positive Rate (ε)	bits per element	number of hash functions
10%	4.8	3
1%	9.6	7
0.1%	14.4	10
0.01%	19.2	13

For formula and mathematical foundation, cp., for example, wikipedia article.

eva.knirsch@kiu.edu.ge

Cassandra Bloom Filter Tuning

https://cassandra.apache.org/doc/5.0/cassandra/reference/cql-commands/create-table.html

Setting the desired fp_rate is part of the create command in Cassandra:
"
Default: bloom_filter_fp_chance = '0.01' (1%)

False-positive probability for SSTable bloom filter. When a client requests data, the bloom filter checks if the row exists before executing disk I/O. Values range from 0 to 1.0, where: 0 is the minimum value use to enable the largest possible bloom filter (uses the most memory) and 1.0 is the maximum value disabling the bloom filter.
Recommended setting: 0.1. A higher value yields diminishing returns.
"

Cassandra documentation says that Casandra adjusts the length of the bit array according to the desired fp_rate. It does not tell about adjusting the number of used hash functions.

eva.knirsch@kiu.edu.ge

Rocks DB Bloom Filter Tuning

https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter/343f24d064bcb48deb8140305e7329d380dd8741

"

In RocksDB, each SST file has a corresponding Bloom filter. It is created when the SST file is written to storage, and is stored as part of the associated SST file. Bloom filters are constructed for files in all levels in the same way.
Bloom filters may only be created from a set of keys - there is no operation to combine Bloom filters. When we combine two SST files, a new Bloom filter is created from the keys of the new file.
When we open an SST file, the corresponding Bloom filter is also opened and loaded in memory. When the SST file is closed, the Bloom filter is removed from memory.
"

RocksDB offers a "bits_per_key" setting to set the desired fp_rate. Documentation does not say anything about setting number of hash functions.

eva.knirsch@kiu.edu.ge

Variations of Bloom Filters

- Many variations of Bloom filters exist.
- For Postgres there is also an extension bloom filter
- The Postgres bloom filter is also based on hash functions but the concept and implementation are different.

eva.knirsch@kiu.edu.ge