BIG DATA MANAGEMENT

CE/CZ4123

KEY-VALUE STORE
LSM-TREE BASICS

Siqiang Luo

Assistant Professor

PREPARATION

- In previous lectures, we introduced the basic concepts of NoSQL databases
  - Key-Value Store/Key-Value Database
  - Wide-column database
  - Document database
  - Graph database

- In the following lectures, we will introduce the mechanism of Key-Value Store, which is the most fundamental NoSQL database

BASIC CONCEPT

- **Data model**
  - Key-Value Store stores the data in key-value format
  - Every key corresponds to a value
- **Functions**: It supports four functions
  - **Get**: Given a key, search the value indexed by the key
  - **Range-Get**: Given a key range, search all the values indexed by any key within the range
  - **Put** a new key-value pair
  - **Delete** a key-value pair
- **Data engine**: The log-structured merge tree (LSM-tree)

GET AND RANGE-GET

- Suppose a key-value store has the data {(1, value1), (2, value2), (3, value3)}
  - **Get**(1) → value1
  - **Get**(2) → value2
  - **Get**(5) → {}

  - **Range-Get**([2,3]) → {value2, value3}
  - **Range-Get**([3,5]) → {value3}
  - **Range-Get**([4,5]) → {}

PUT AND DELETE

- Suppose a key-value store has the data {(1, value1), (2, value2), (3, value3)}, then logically
  - After **Put**(4,value4), we have {(1, value1), (2, value2), (3, value3), (4, value4)} in the key-value store
  - After **Delete**(1), we have {(2, value2), (3, value3), (4, value4)} in the key-value store
  - After **Put**(3, value5), we have {(2, value2), (3, value5), (4, value4)} in the key-value store

THINK…

Suppose we need to store 10 billion key-value pairs in the database, what data structure you will use?

FIRST IDEA

An array with ten billion entries

Size: 10^{10} * 100 Bytes=1000GB

Stored in memory? Stored in disk?

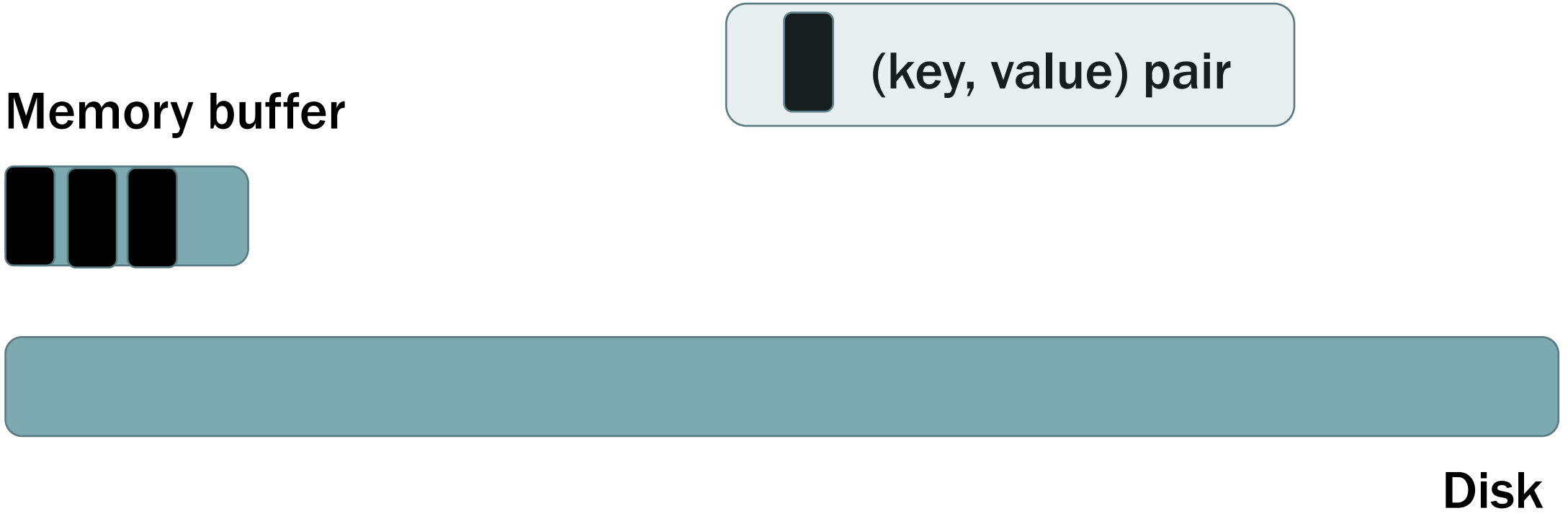
FIRST IDEA

An array with ten billion entries

Too large to be stored in main memory
Main memory does not guarantee persistency

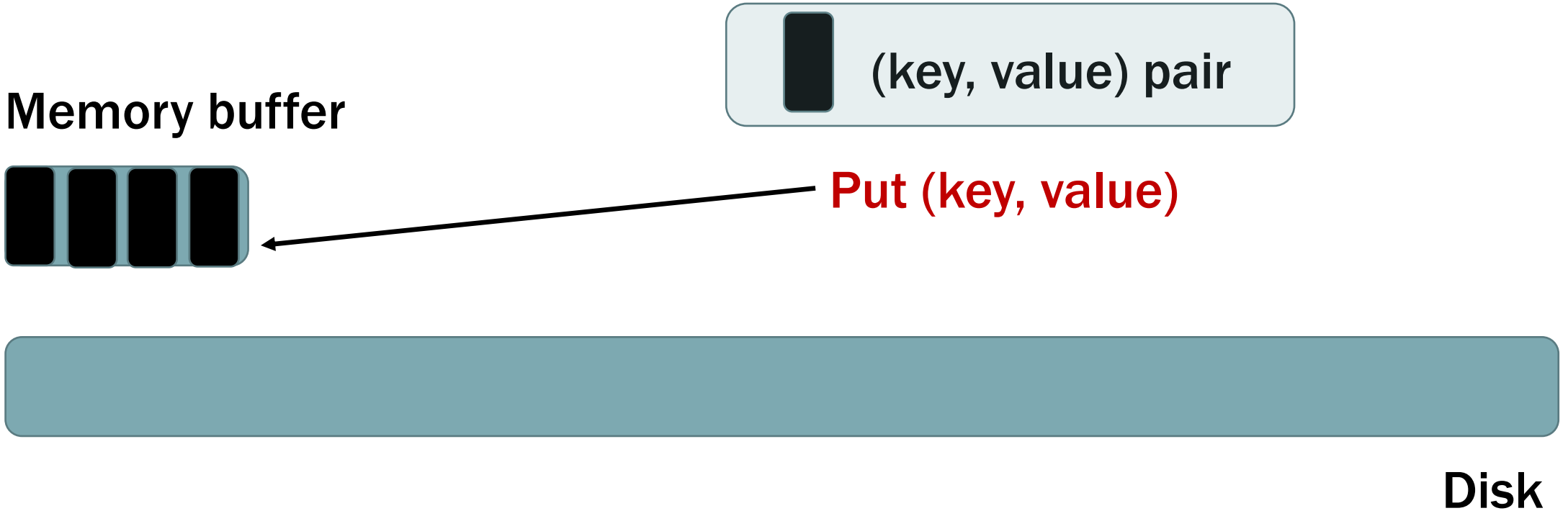
Disk is too slow to access, put and delete operations are costly.

SECOND IDEA



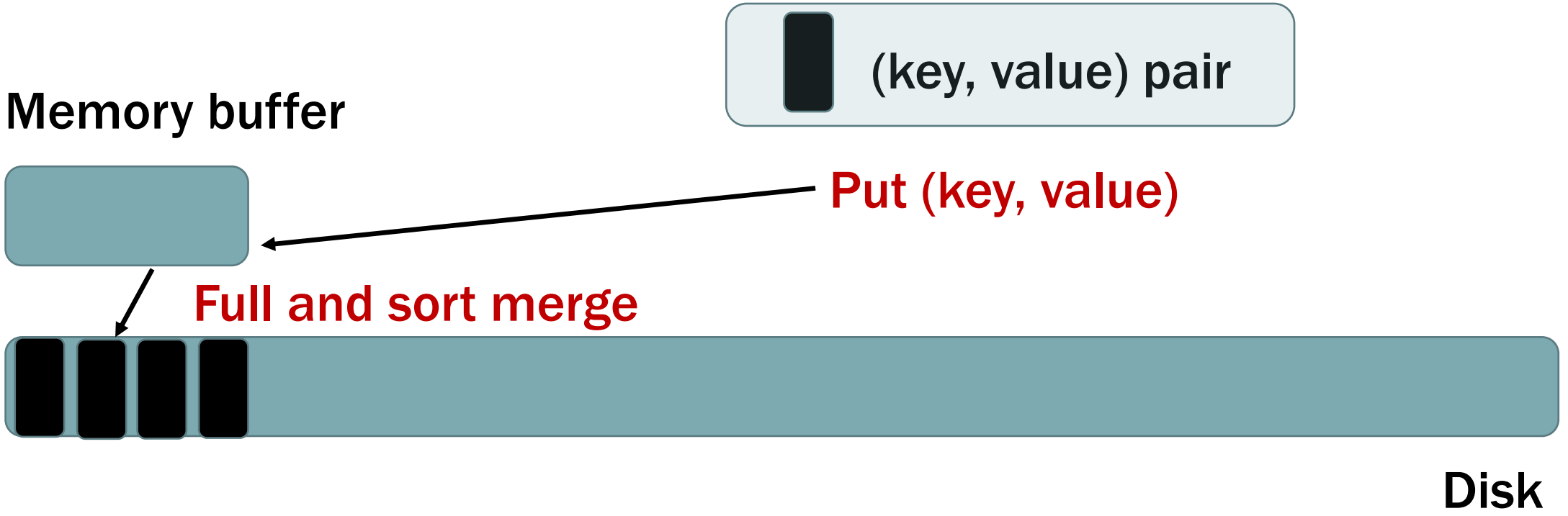
When handling Put(key,value), first insert the key-value pair into the main memory; when memory buffer is full, put all the buffer as a “run” into the disk.

SECOND IDEA



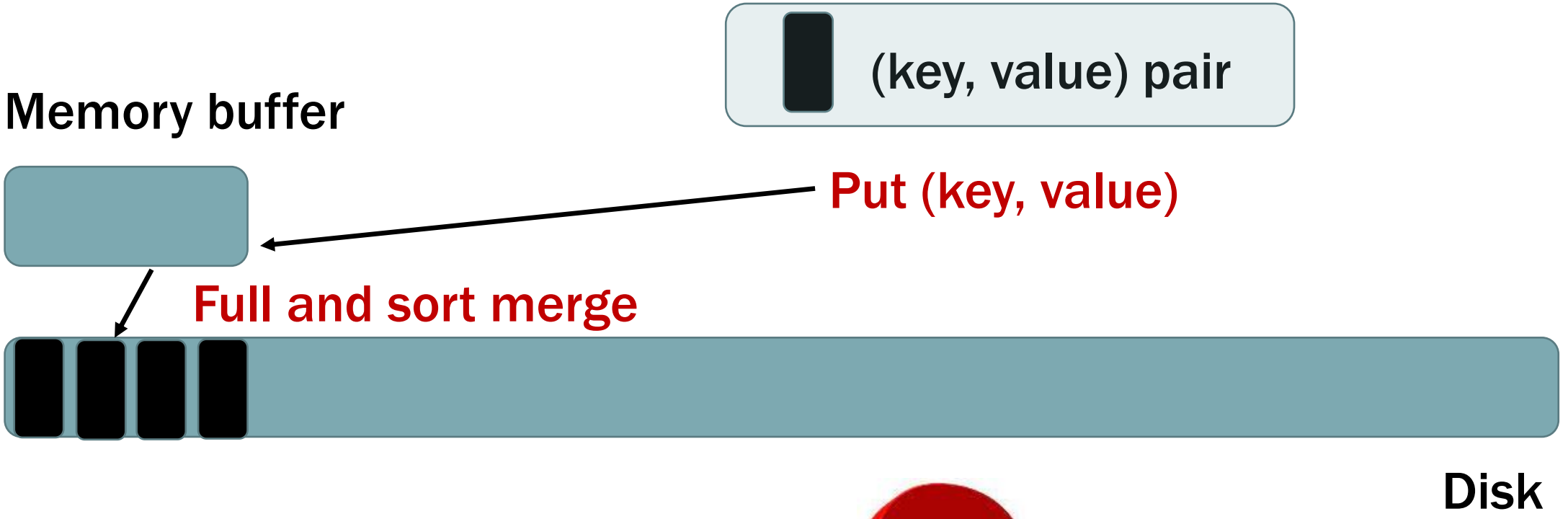
When handling Put(key,value), first insert the key-value pair into the main memory; when memory buffer is full, put all the buffer as a “sorted run” into the disk.

SECOND IDEA



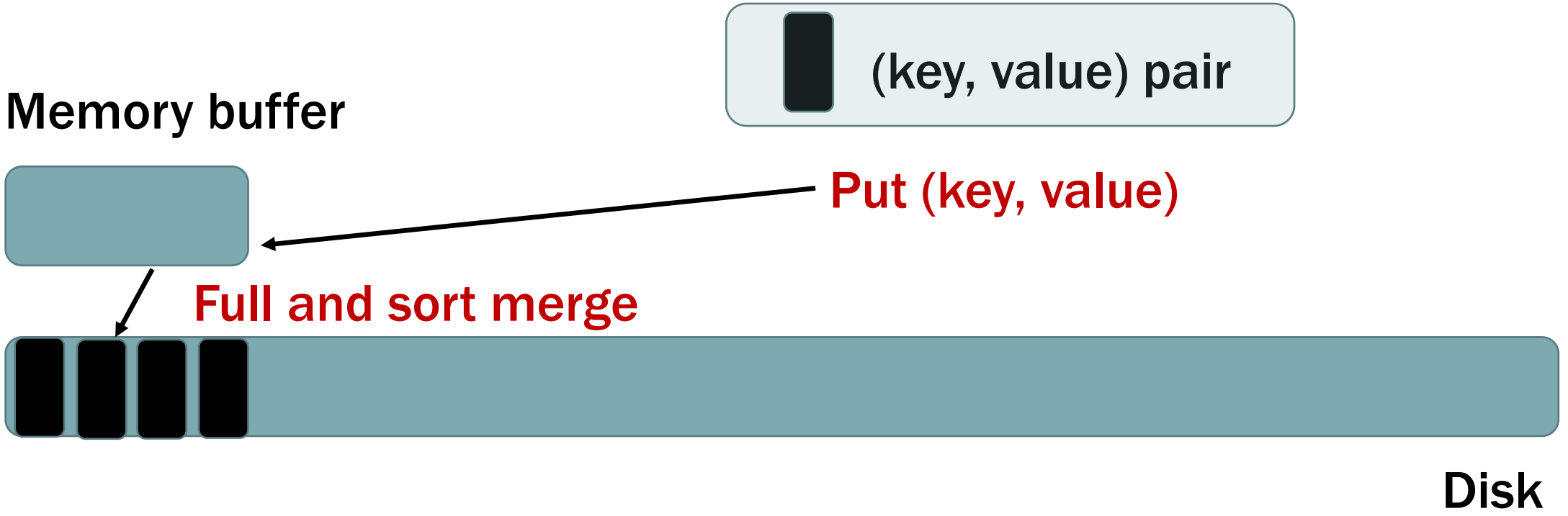
When handling Put(key,value), first insert the key-value pair into the main memory; when memory buffer is full, put all the buffer as a “sorted run” into the disk.

SECOND IDEA



Any problems with this design?

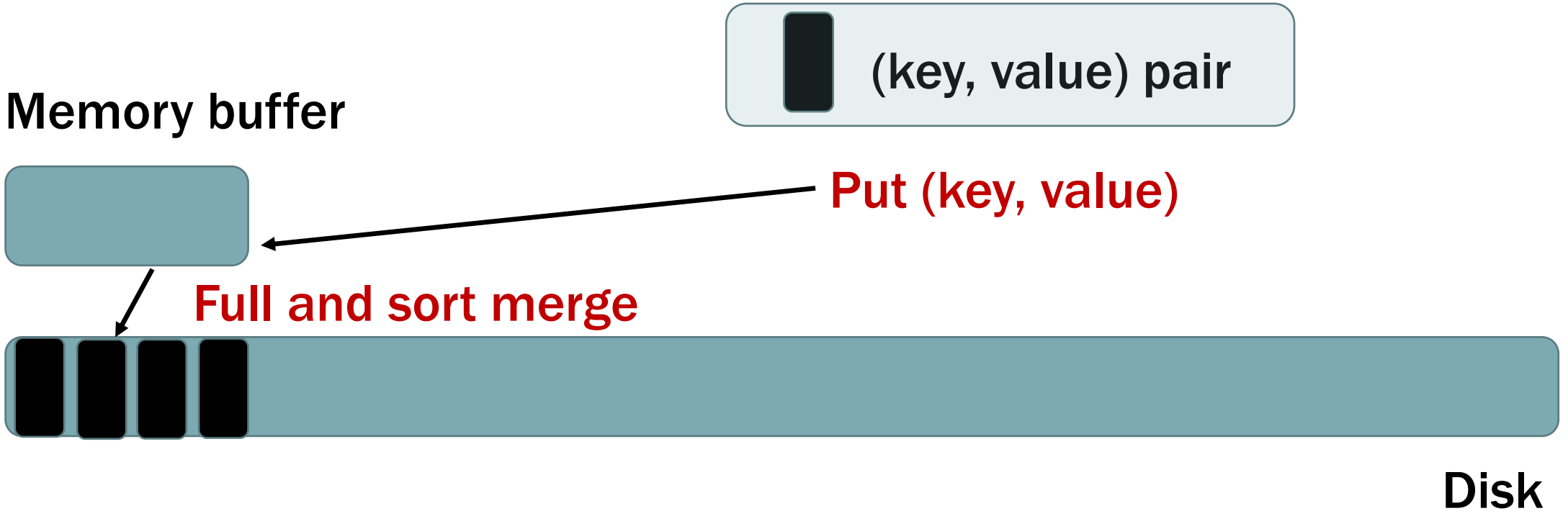
SECOND IDEA



Any problems with this design?

High cost for GET function!

SECOND IDEA

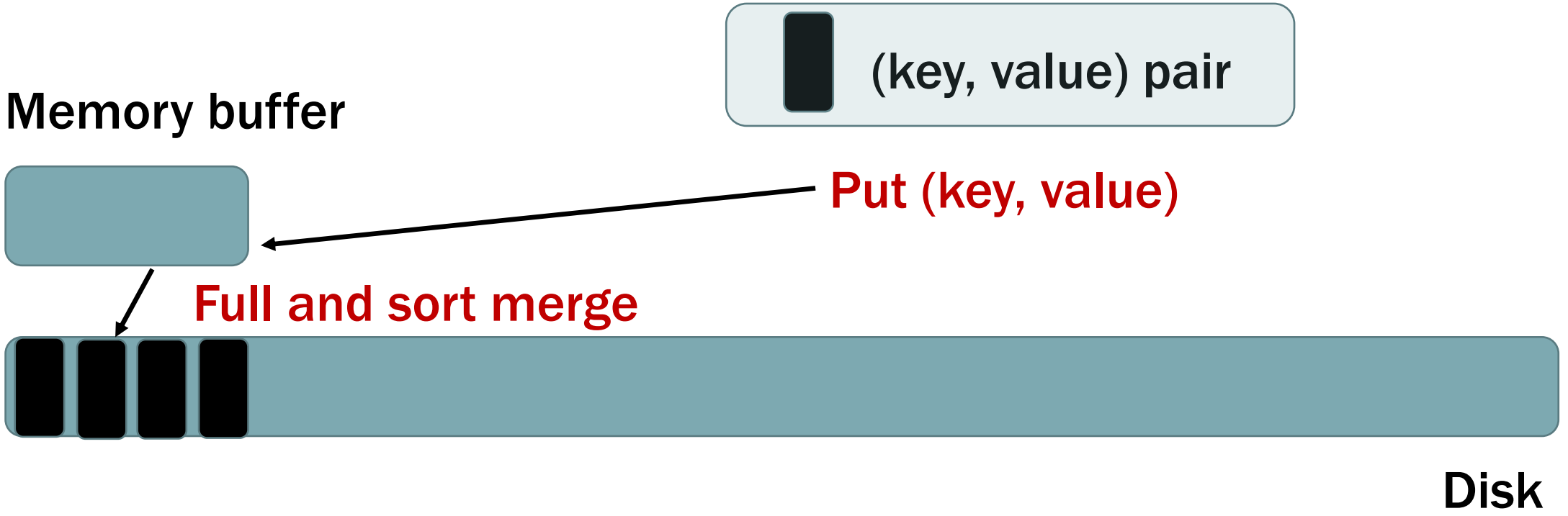


GET(key) consists of two steps

1) Search the key in the main memory buffer; if the key exists in the buffer, directly return the value;

2) Search the key in the disk.

SECOND IDEA

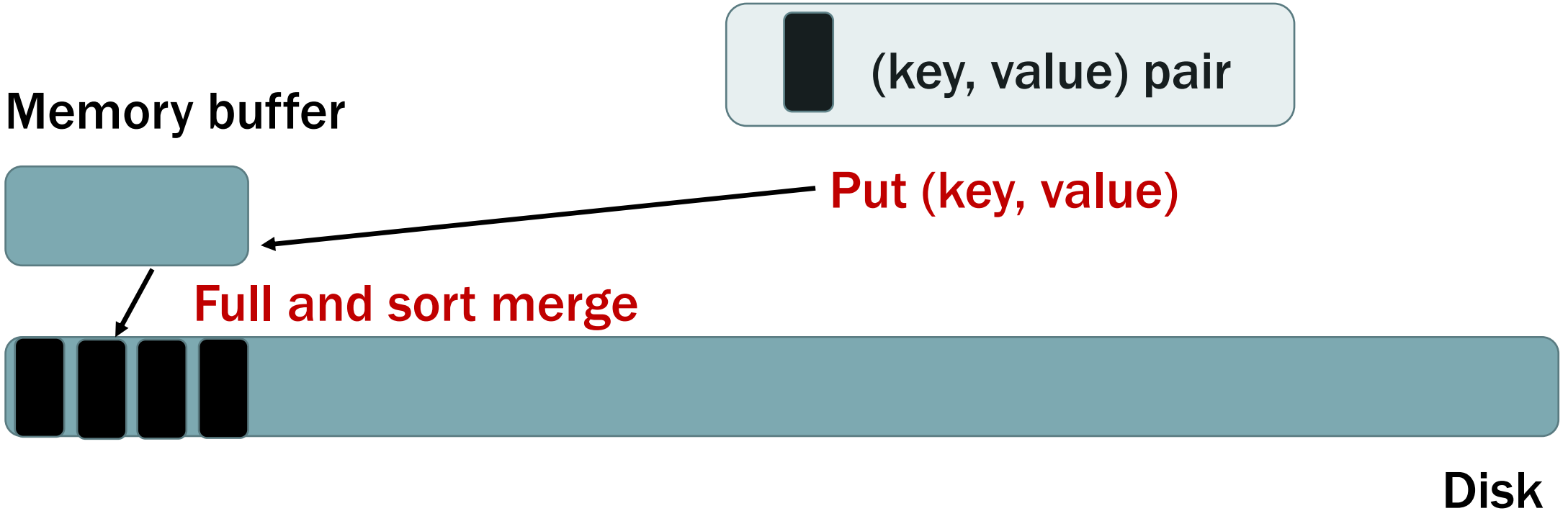


GET(key) consists of two steps

1) Search the key in the main memory buffer; if the key exists in the buffer, directly return the value;

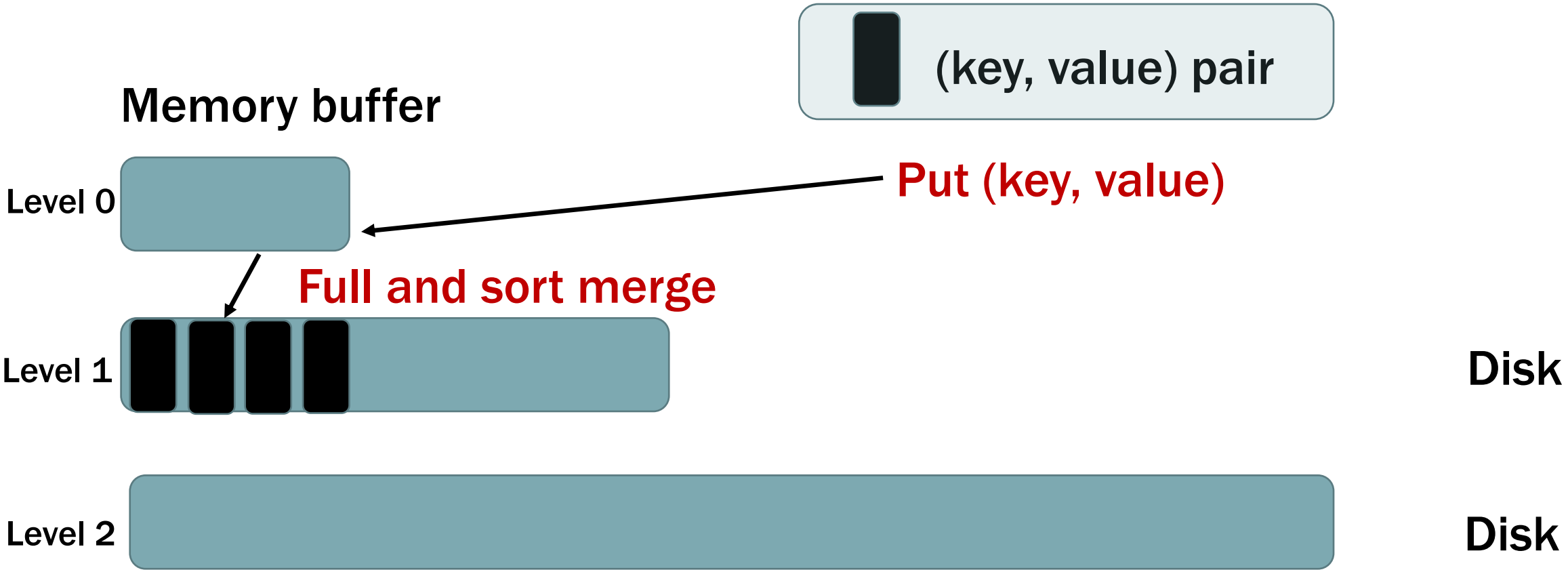
2) Search the key in the disk. ← Very costly

SECOND IDEA



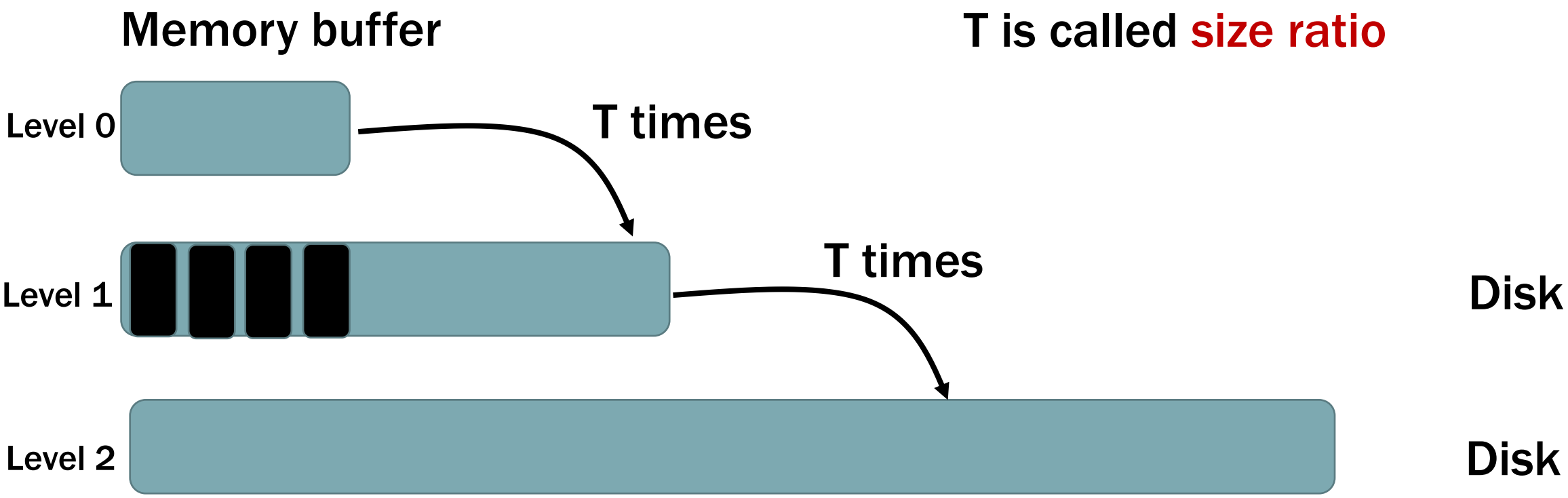
Put(key, value) is also costly

THIRD IDEA: BASIC LSM-TREES

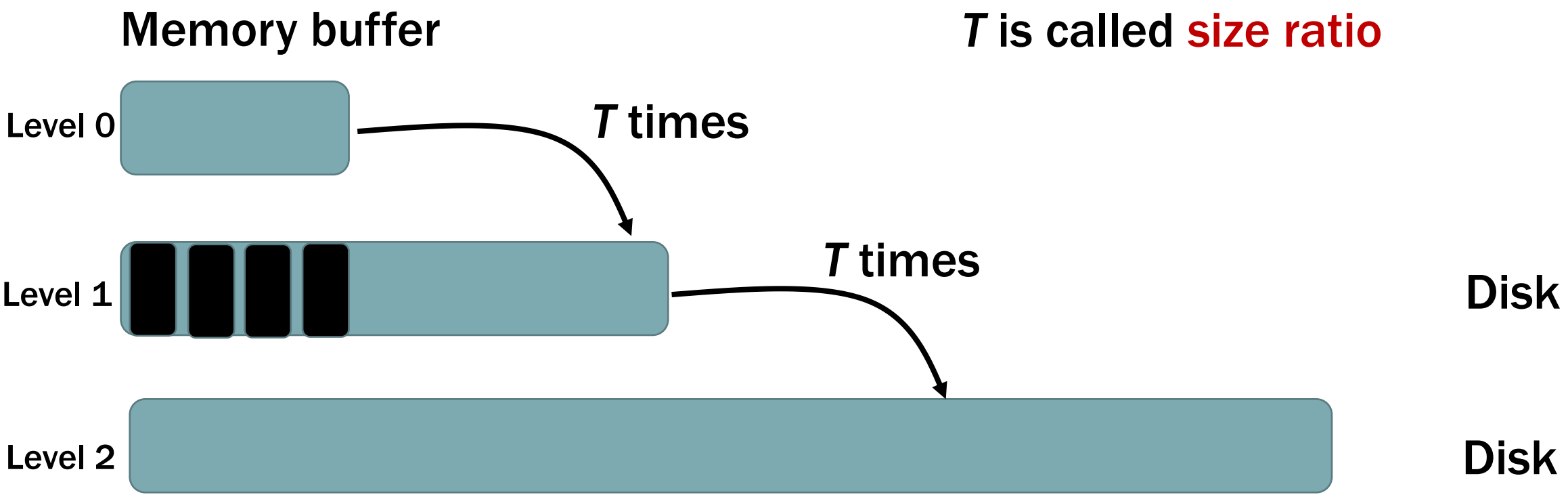


Two levels to multiple levels

BASIC LSM-TREES



BASIC LSM-TREES

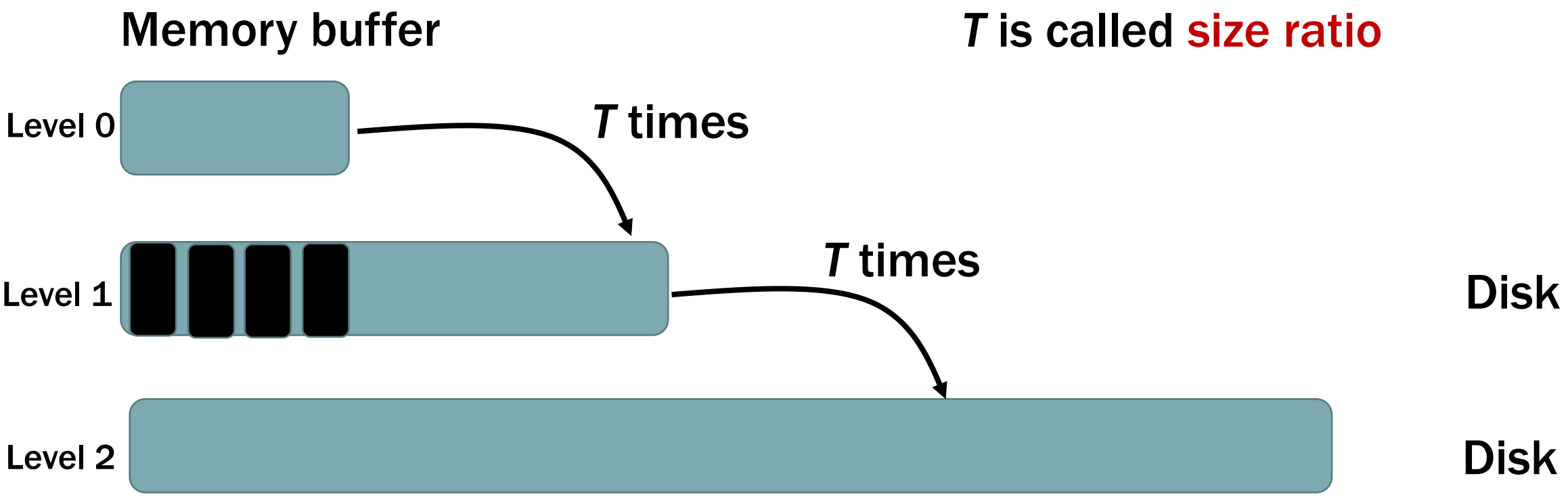


Let *L[i]* be the *i*-th level

L[i] capacity = $S \times T^i$

Mem buffer capacity (i.e., *L[0]* capacity) = *S* (key-value pairs)

BASIC LSM-TREES

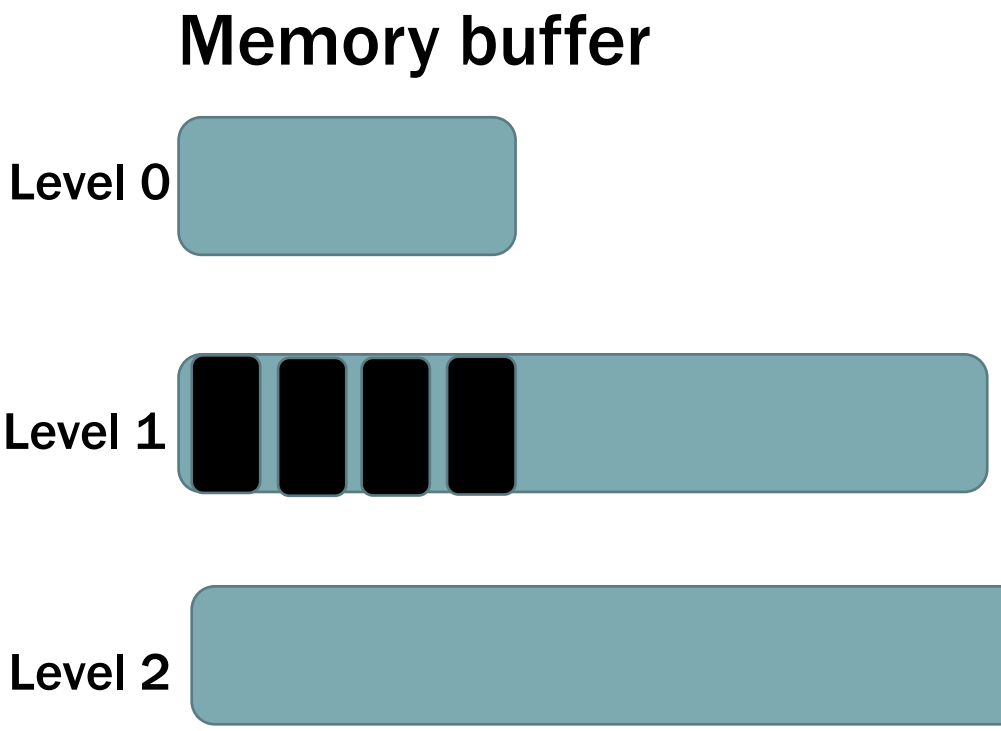


T is called size ratio

Capacity is different from the actual size. *L[i]* capacity may be different from the number of actually stored (key,value) pairs in *L[i]*

L[i] capacity = $S \times T^i$

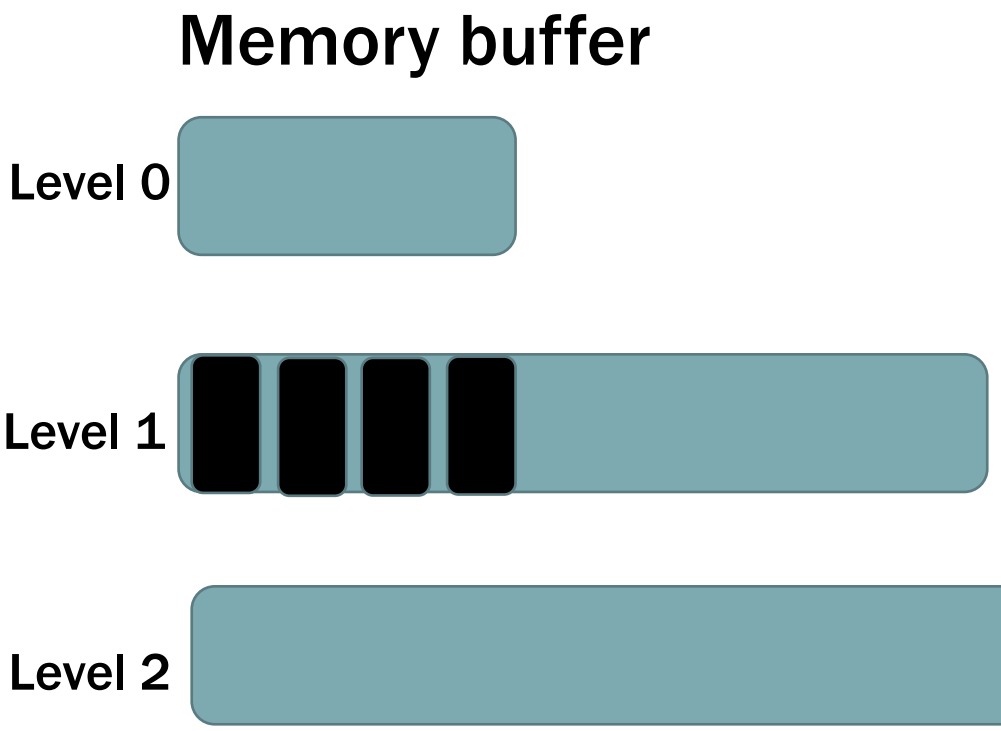
BASIC LSM-TREES




```
void Put(Key k, Value v)
{
    L[0].insert(k,v);
    if(L[0].size==S){
        SortMerge(0);
    }
}
```

```
void SortMerge(Level i)
{
    if (L[i+1] not exists)
        create L[i+1];
    L[i+1].SortMergeWith(L[i]);
    L[i].clear();
    if(L[i+1]>S*(T^(i+1)-T^i )
        SortMerge(i+1);
}
```

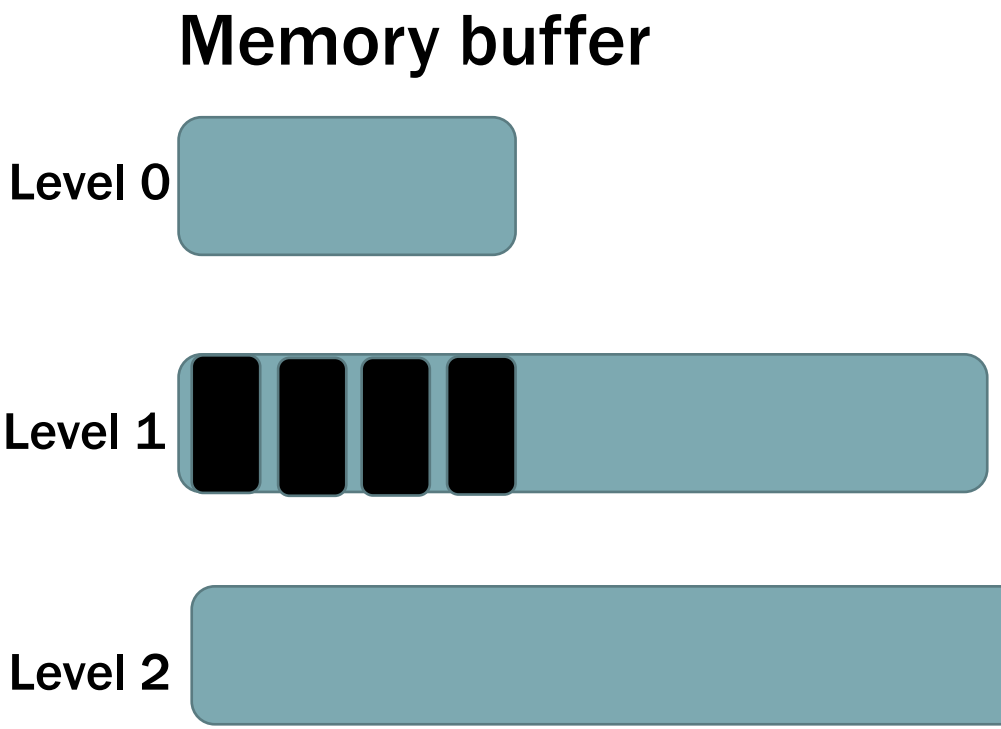
BASIC LSM-TREES




```
void Put(Key k, Value v)
{
    L[0].insert(k,v);
    if(L[0].size==S){
        SortMerge(0);
    }
}
```

```
void SortMerge(Level i)
{
    if (L[i+1] not exists)
        create L[i+1];
    L[i+1].SortMergeWith(L[i]);
    L[i].clear();
    if(L[i+1]>S*(T^(i+1)-T^i))
        SortMerge(i+1);
}
```

BASIC LSM-TREES

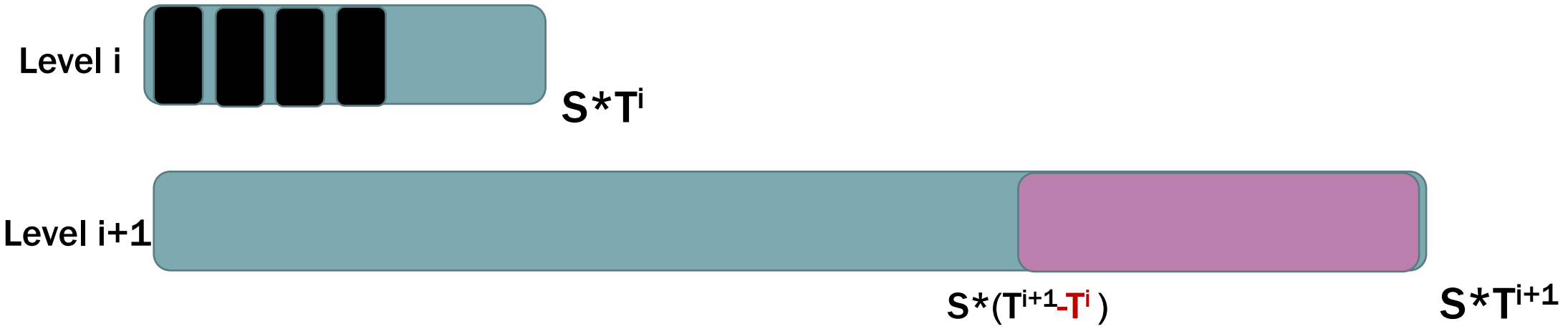



```
void Put(Key k, Value v)
{
    L[0].insert(k,v);
    if(L[0].size==S){
        SortMerge(0);
    }
}
```

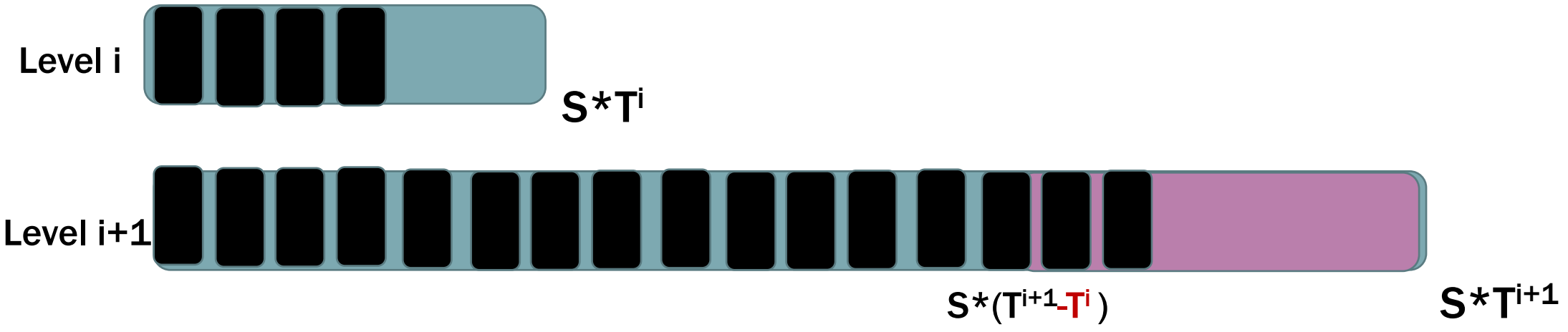
```
void SortMerge(Level i)
{
    if (L[i+1] not exists)
        create L[i+1];
    L[i+1].SortMergeWith(L[i]);
    L[i].clear();
    if(L[i+1]>S*(T^(i+1)-T^i))
        SortMerge(i+1);
}
```

The actual number of (key,value) pairs after merging *L[i]* into *L[i+1]* may be less than actual size(*L[i]*)+actual size(*L[i+1]*)

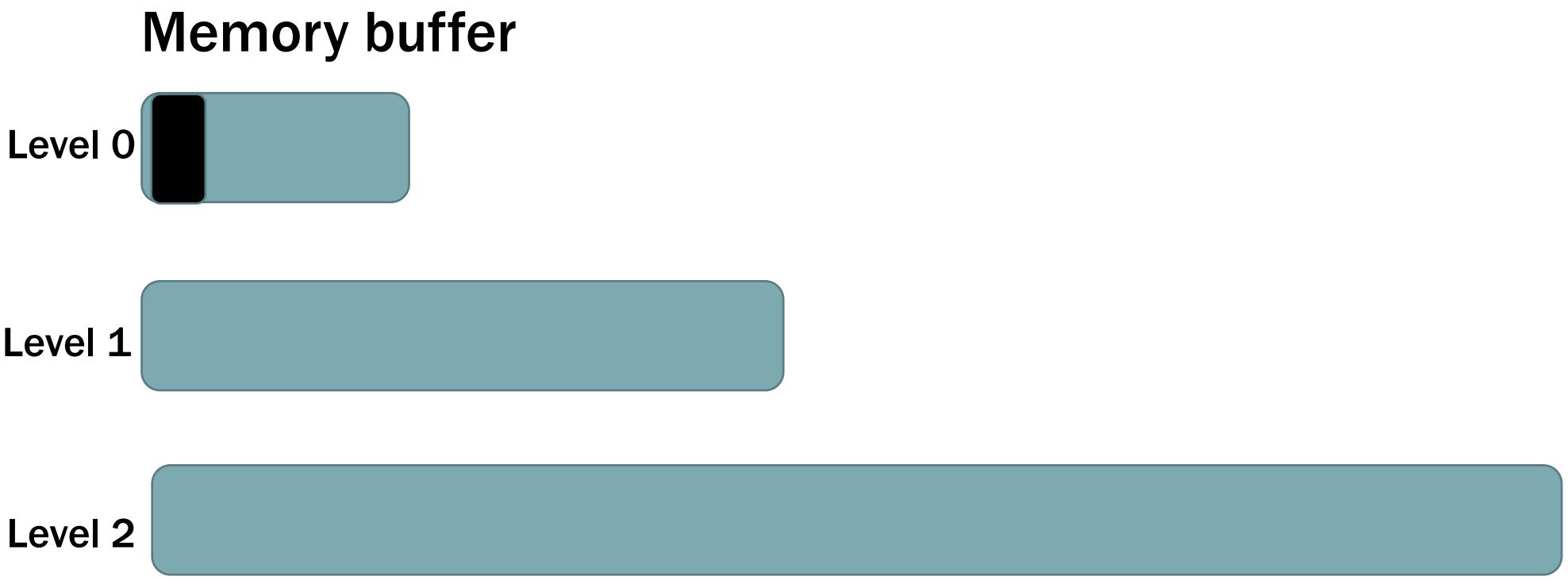
MERGE CONDITION



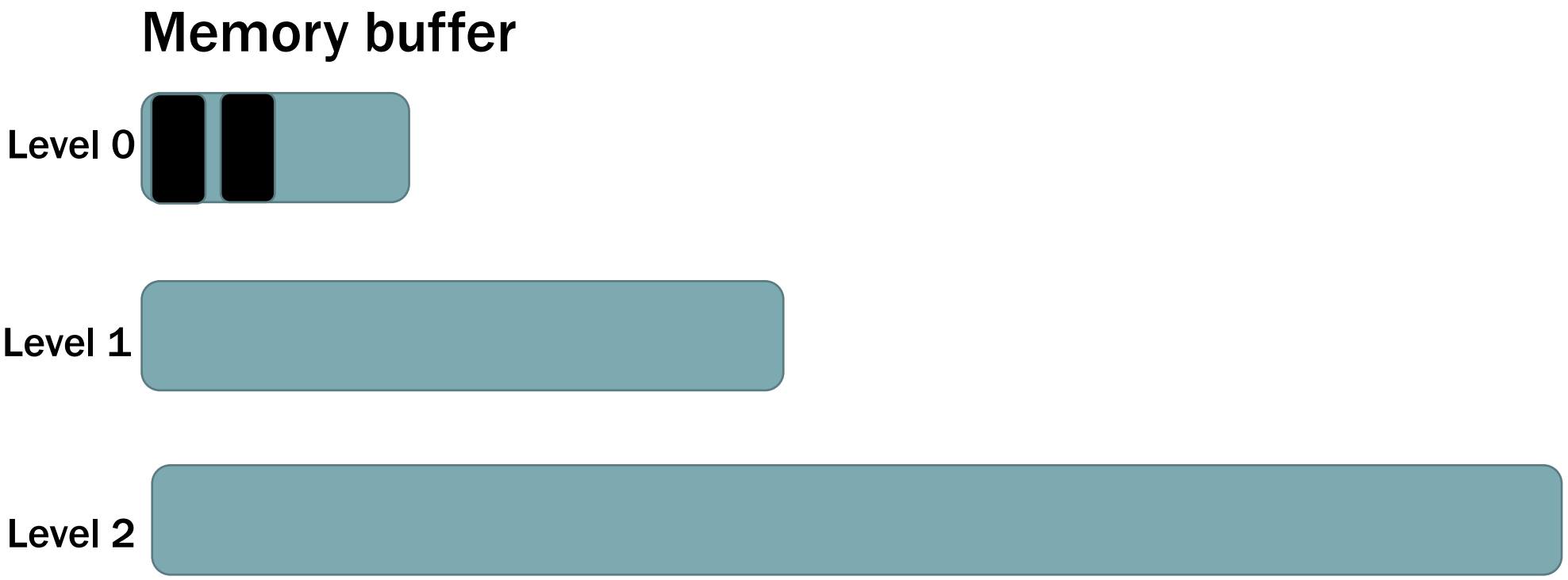
MERGE CONDITION



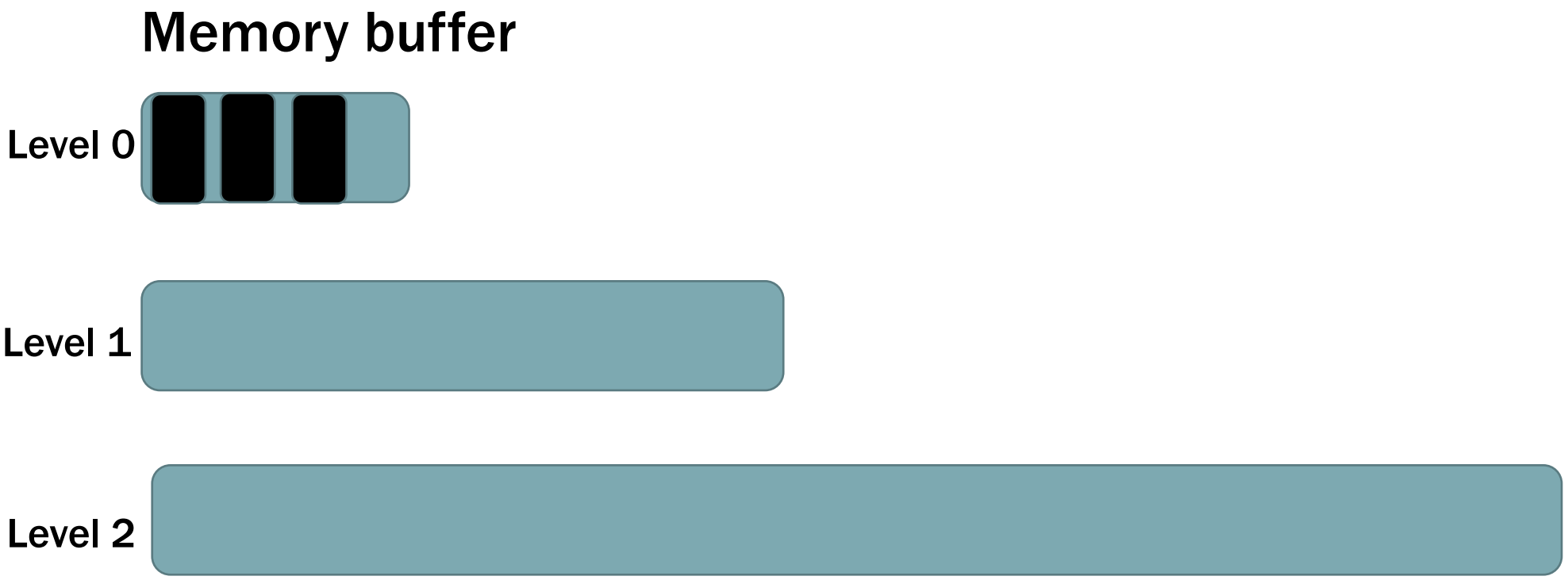
EXAMPLE



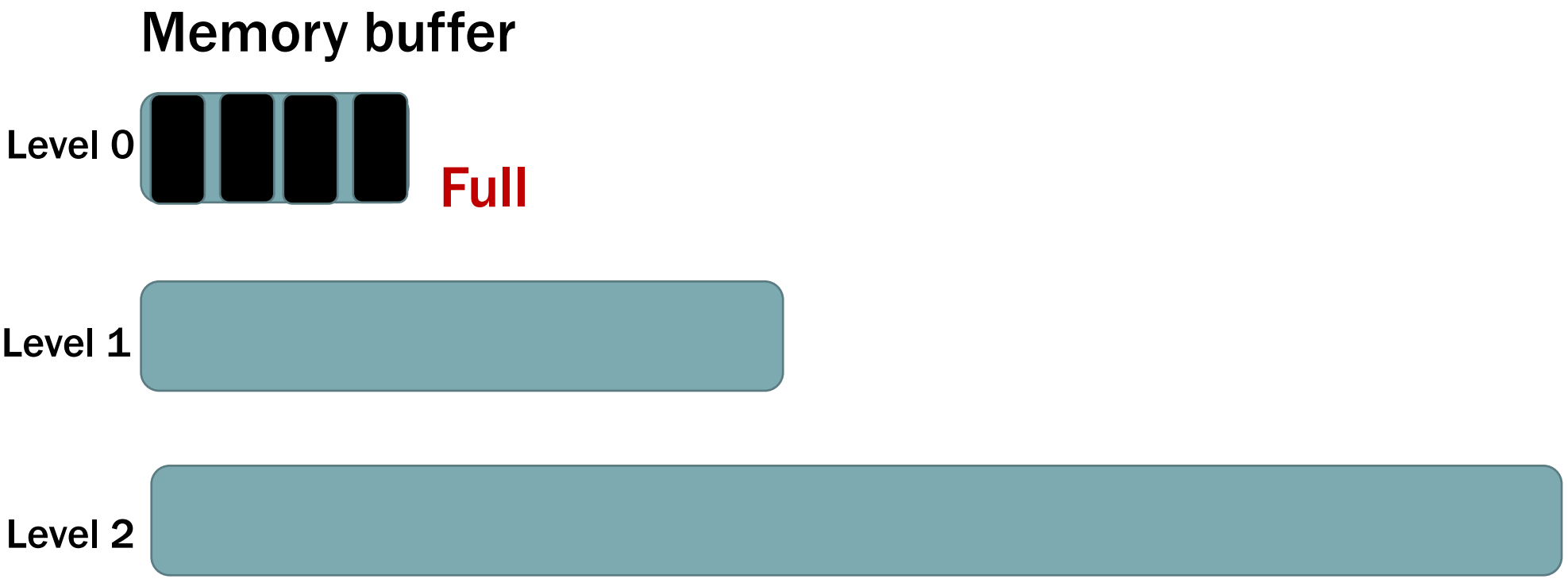
EXAMPLE



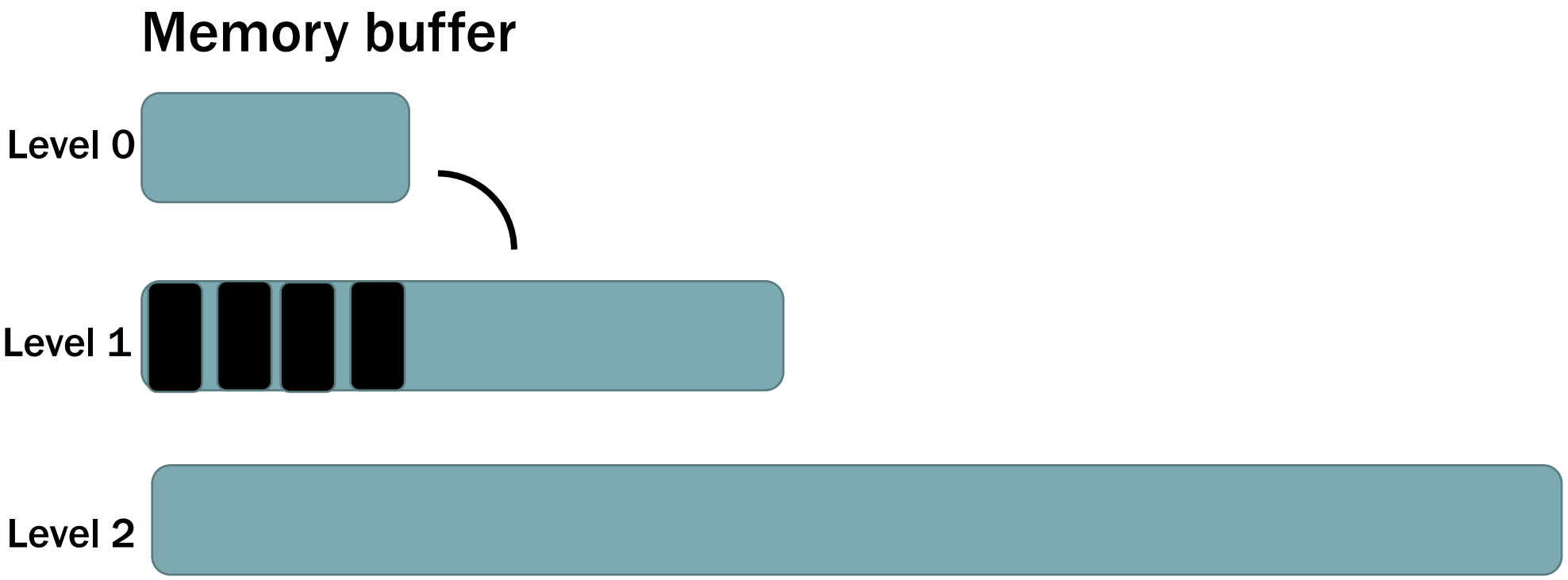
EXAMPLE



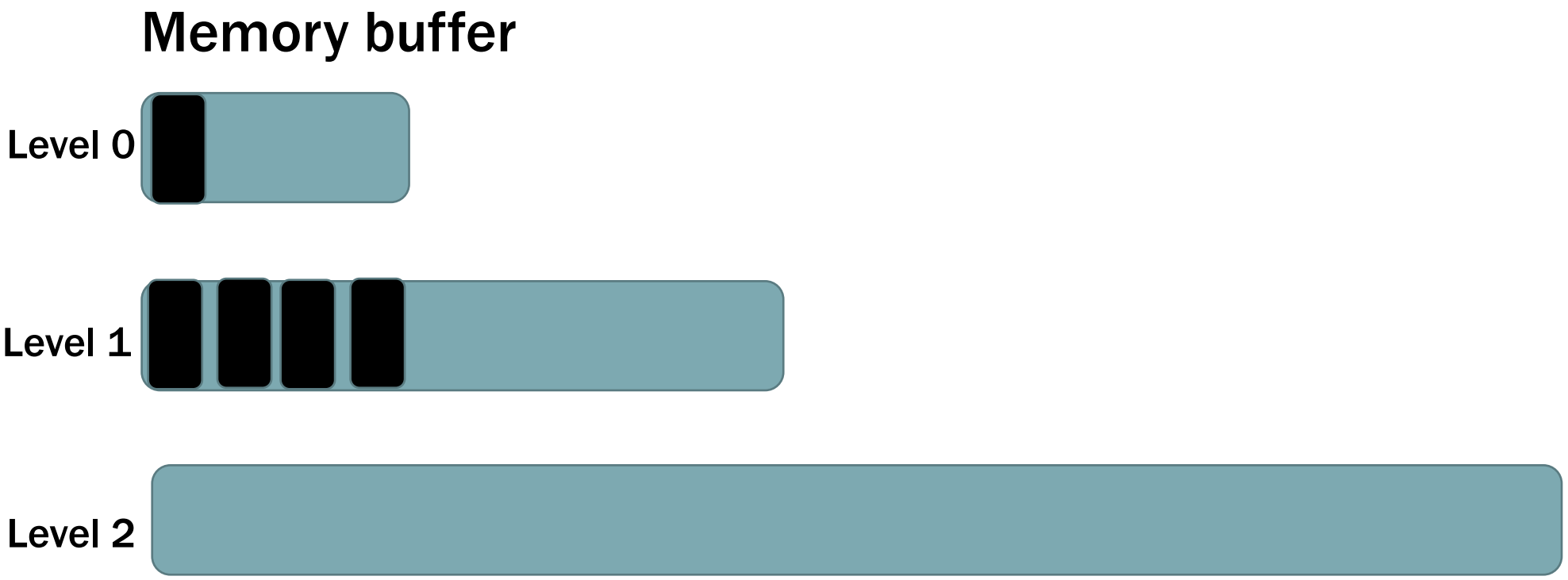
EXAMPLE



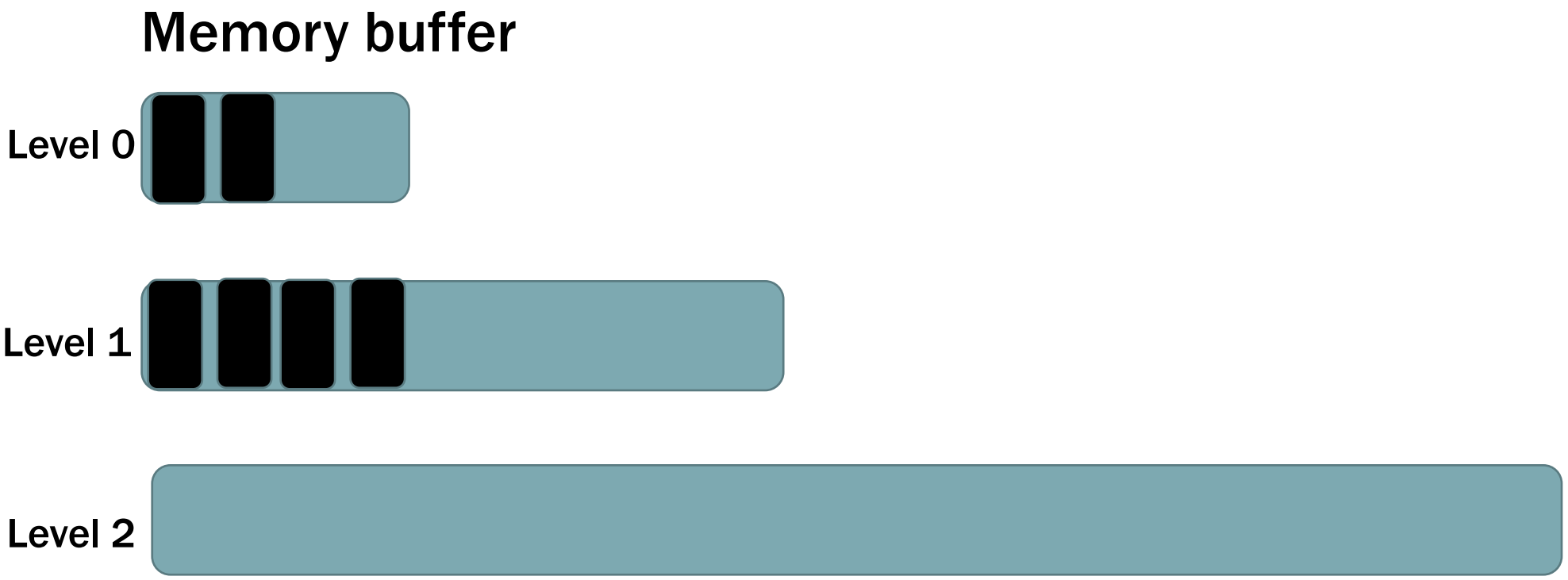
EXAMPLE



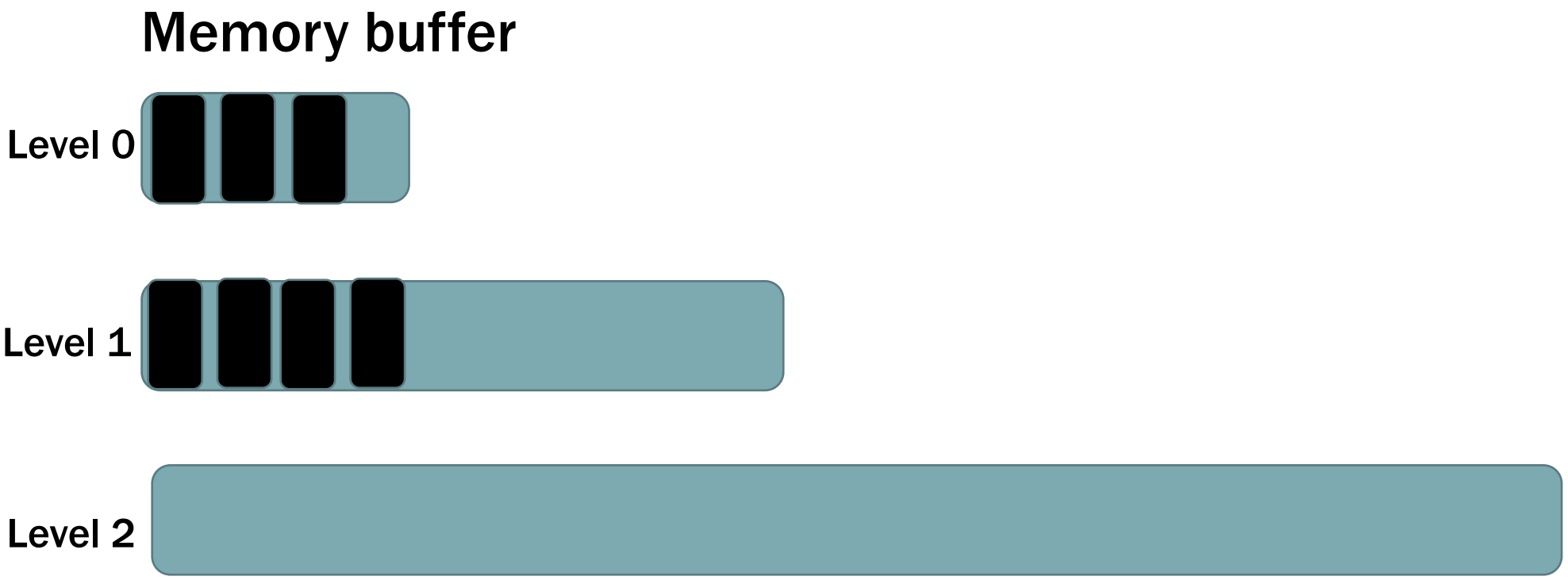
EXAMPLE



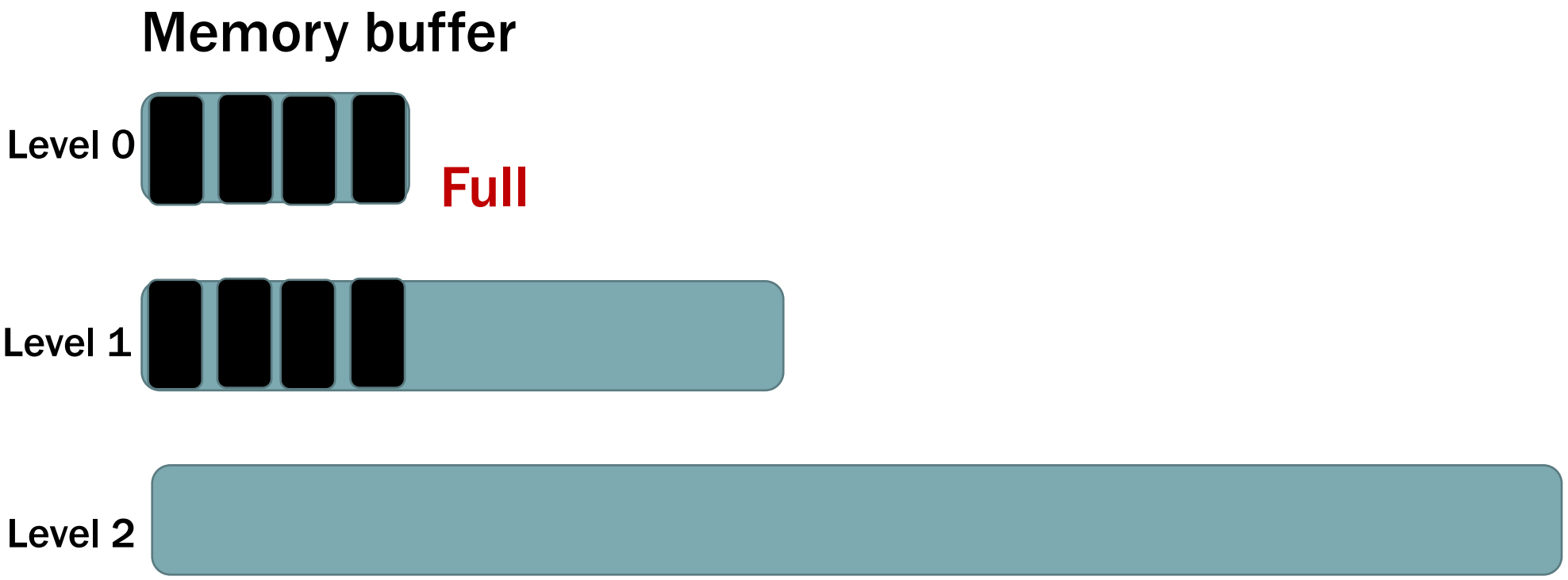
EXAMPLE



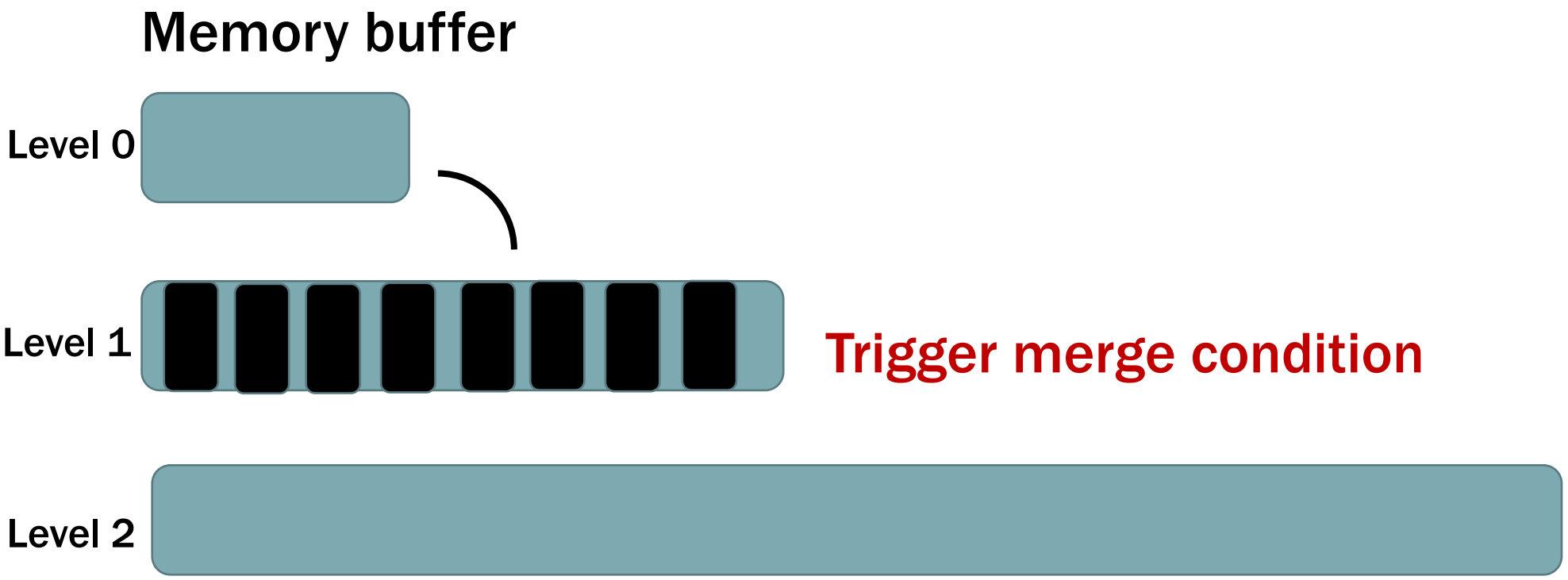
EXAMPLE



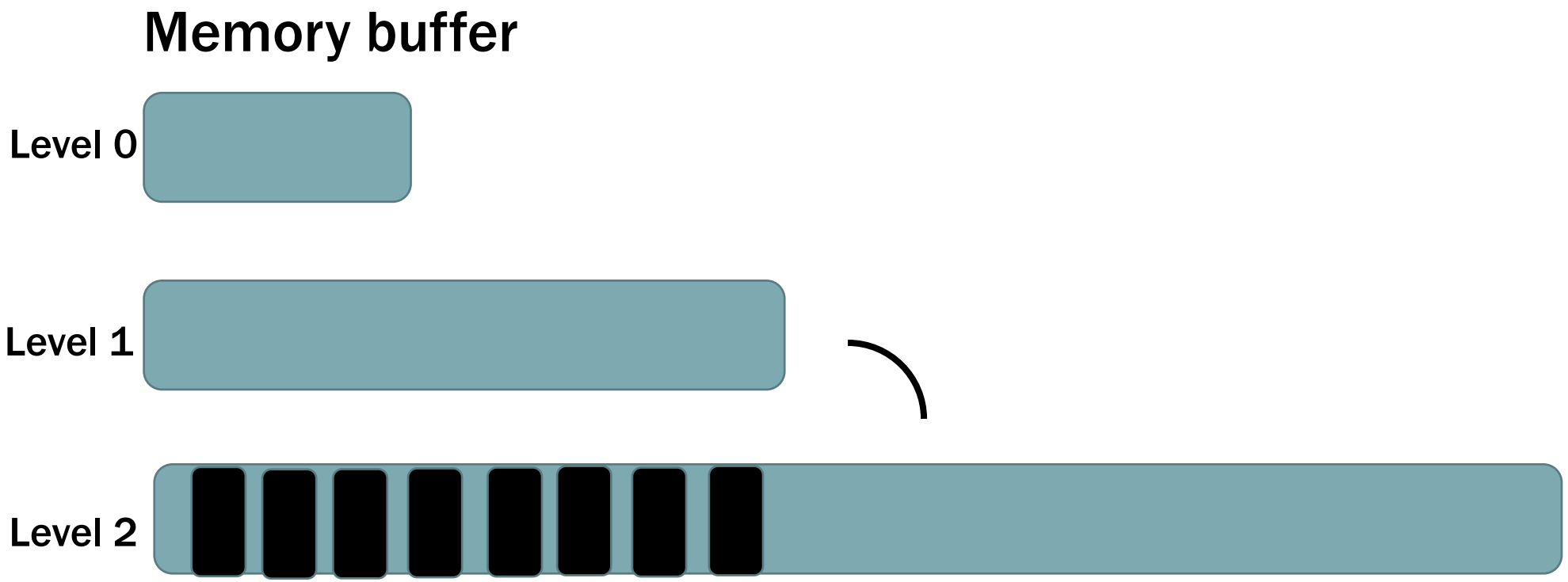
EXAMPLE



EXAMPLE

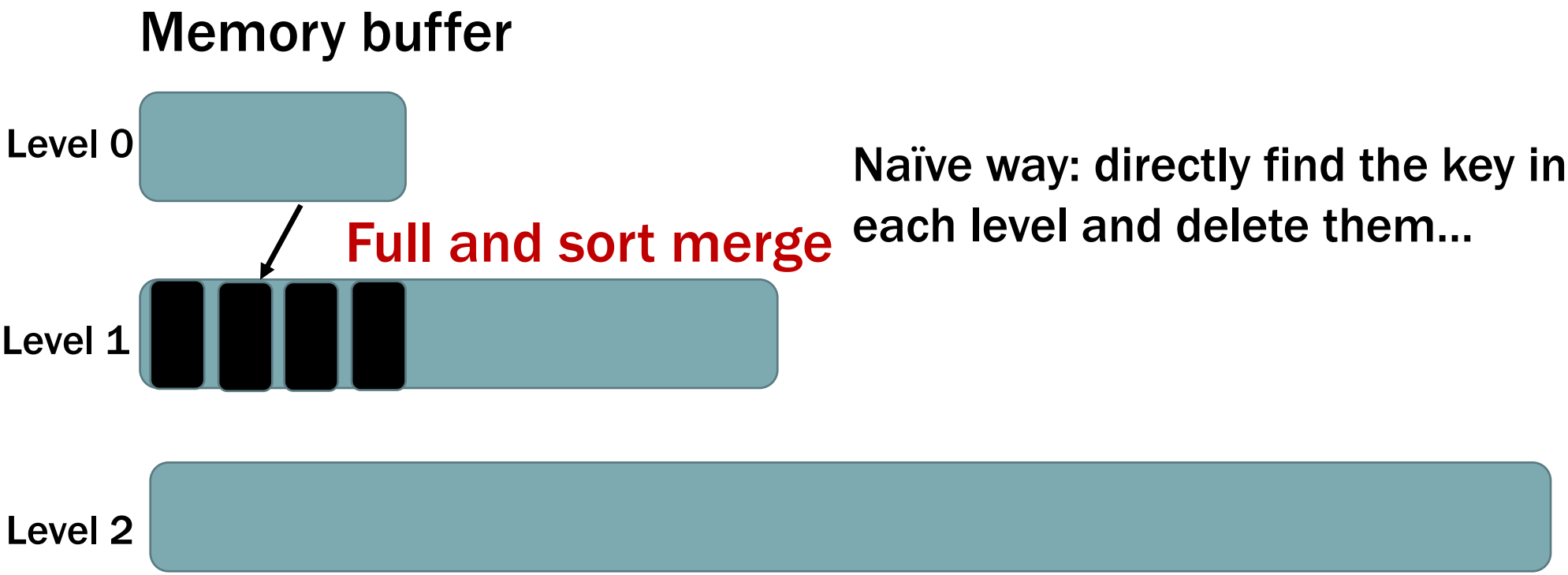

Memory buffer
Level 0
Level 1
Level 2
Trigger merge condition

EXAMPLE



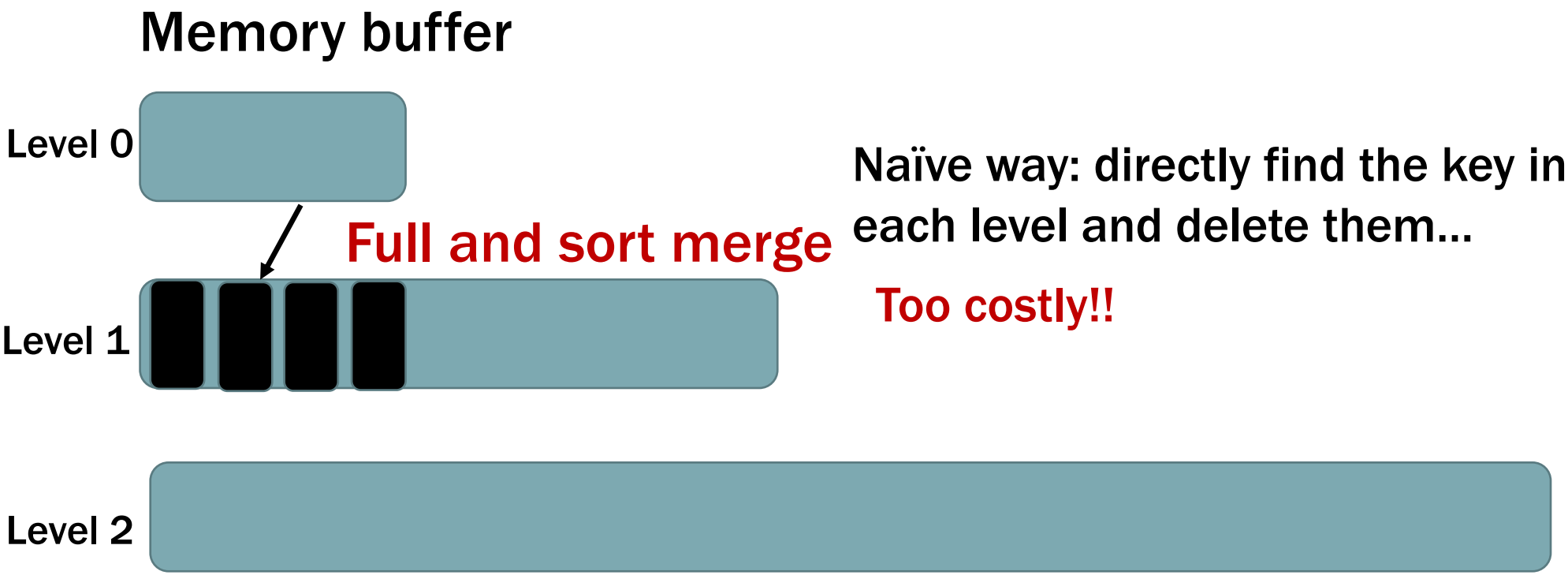
How about deleting a key?

DELETE?



Naïve way: directly find the key in each level and delete them…

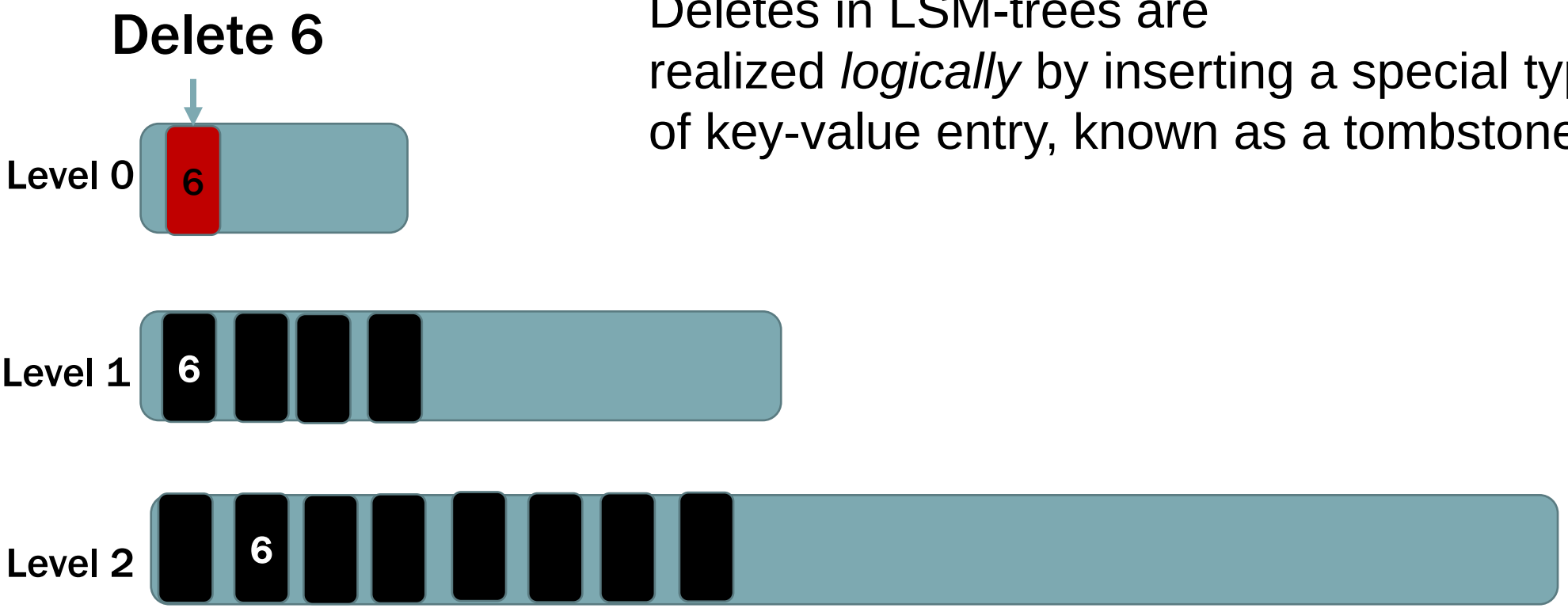
DELETE?



Naïve way: directly find the key in each level and delete them…

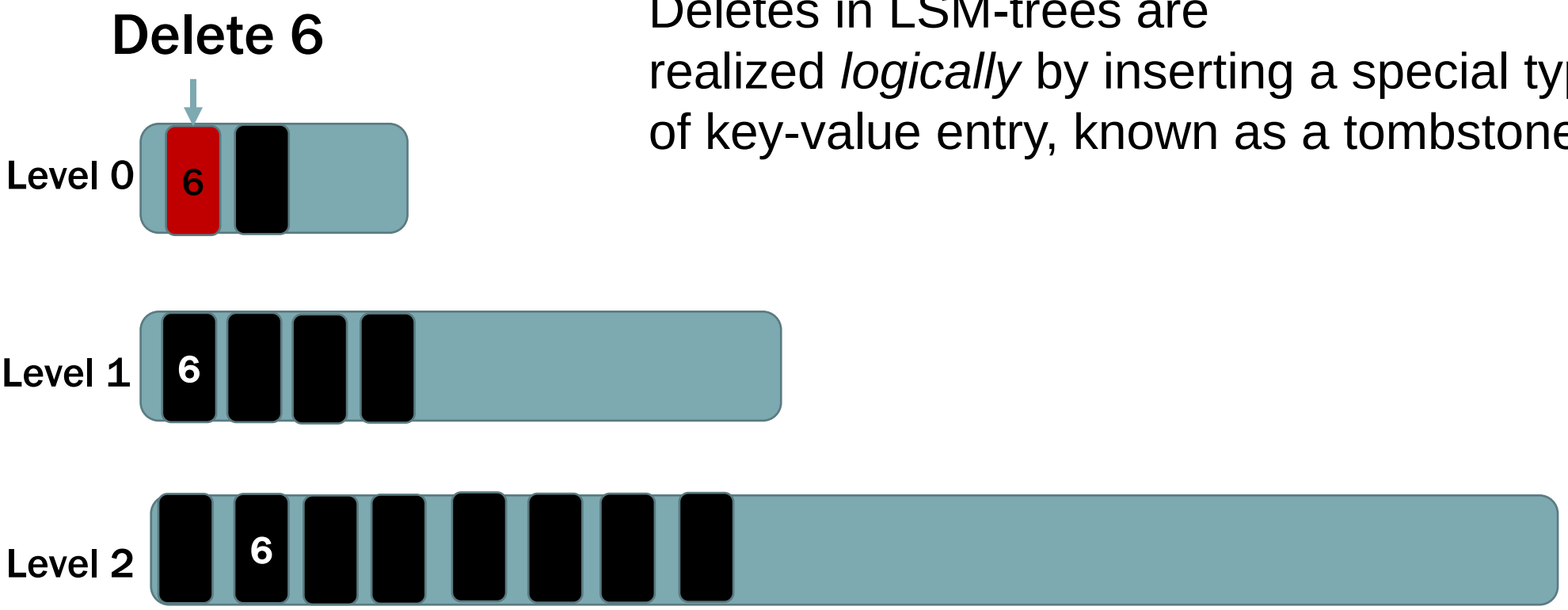
Too costly!!

DELETE--TOMBSTONE



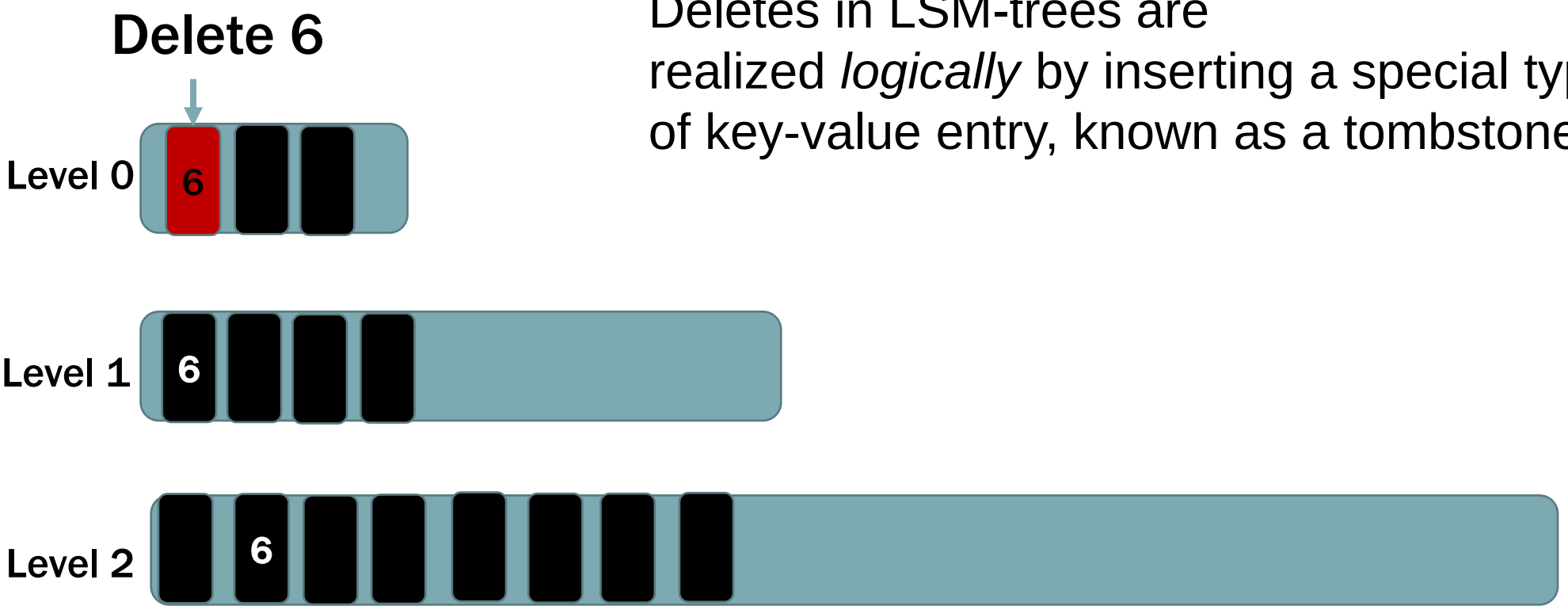
Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

DELETE--TOMBSTONE



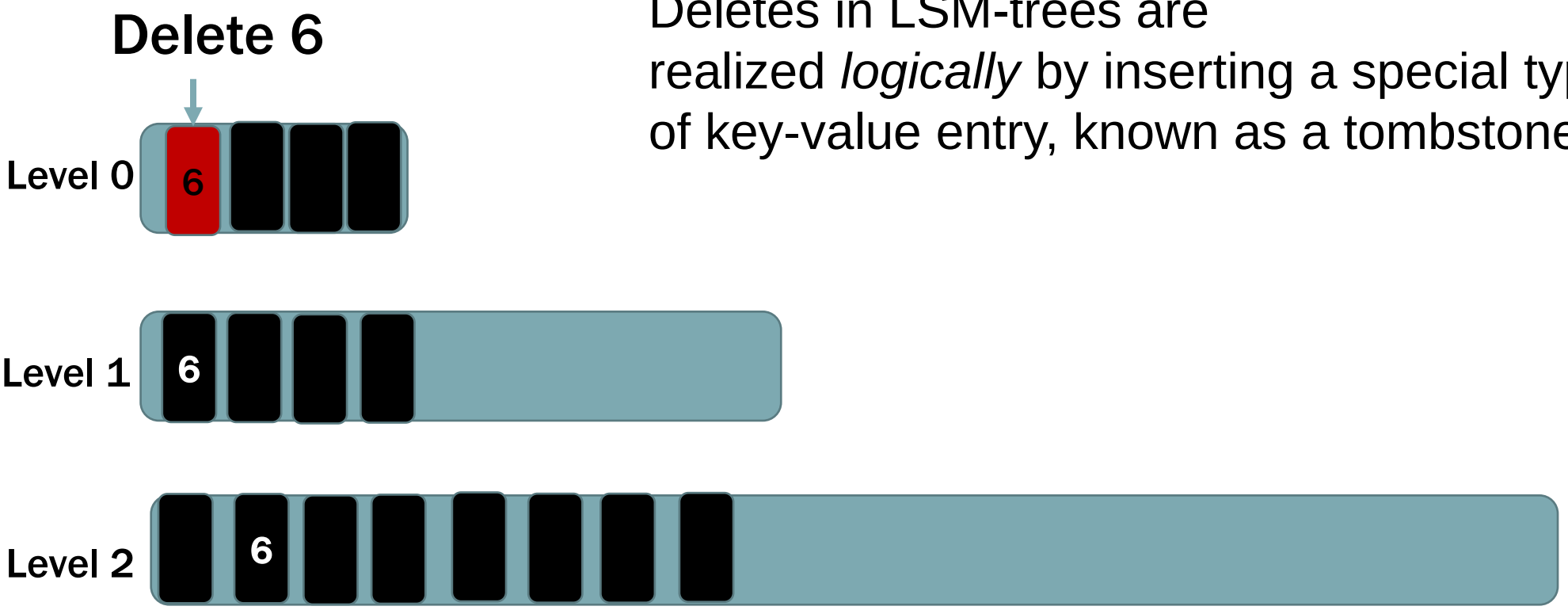
Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

DELETE--TOMBSTONE



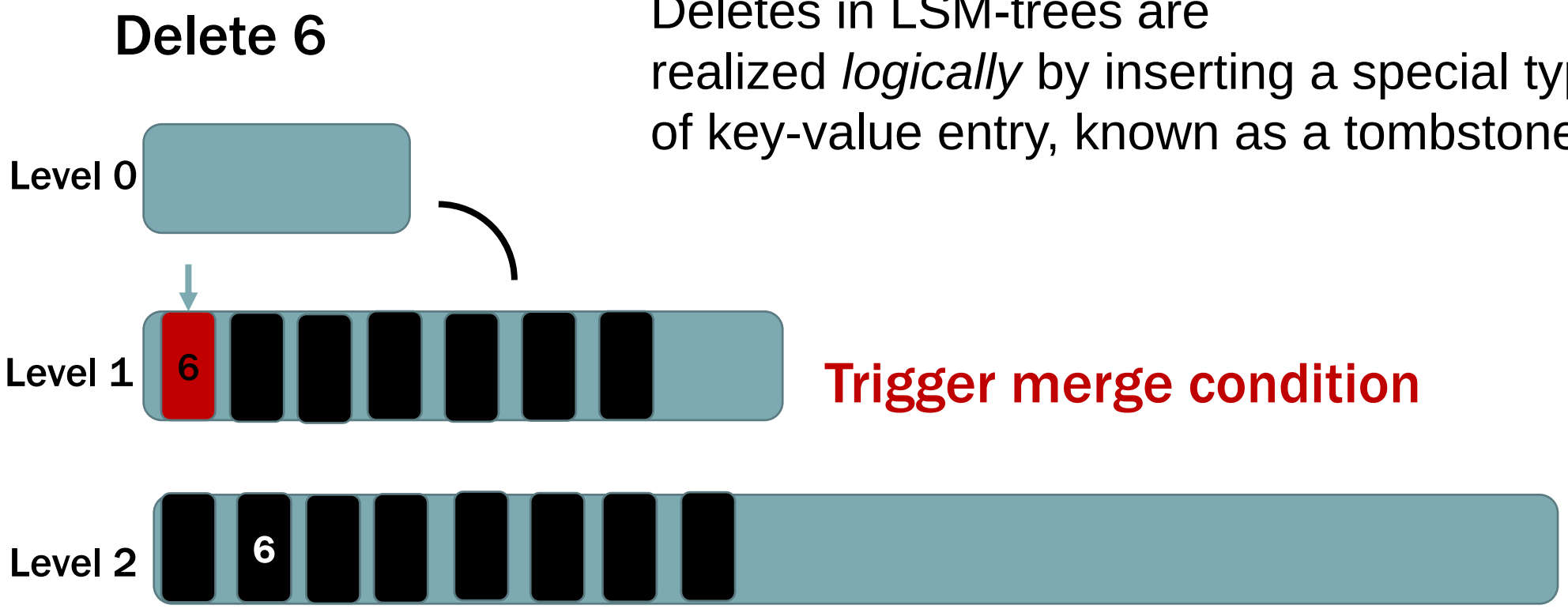
Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

DELETE--TOMBSTONE



Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

DELETE--TOMBSTONE



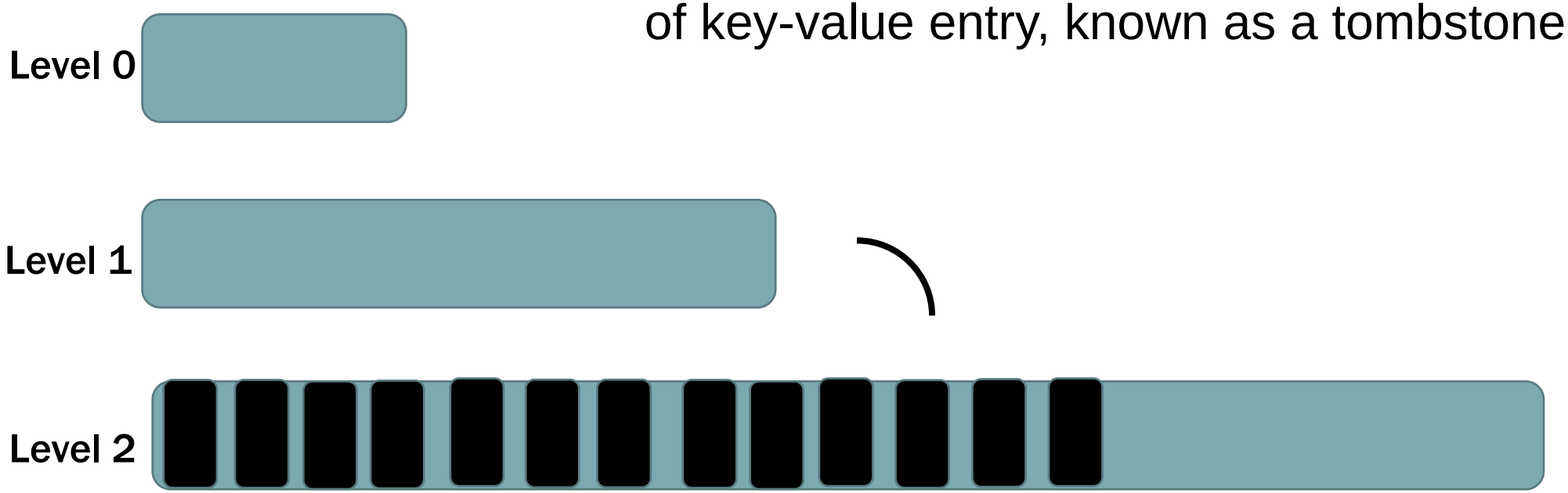
Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

Trigger merge condition

DELETE--TOMBSTONE

Delete 6

Deletes in LSM-trees are realized *logically* by inserting a special type of key-value entry, known as a tombstone.

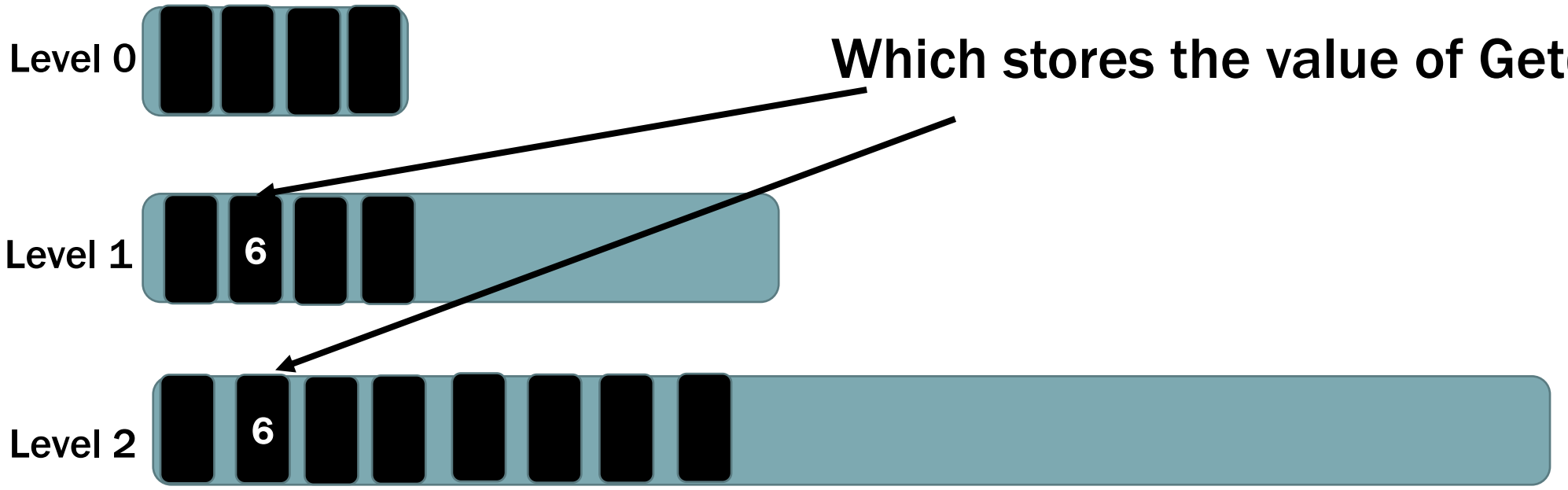


In the last level, the tombstone will be deleted

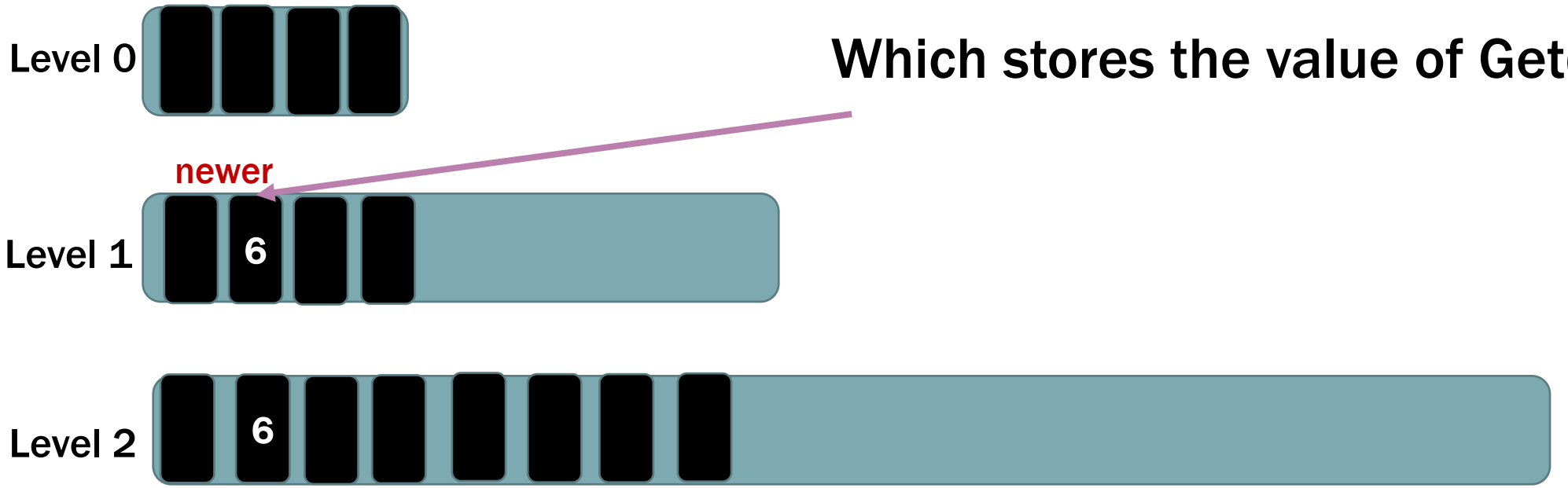
"Red 6" and "black 6" cancel out

How to implement Get(K)?

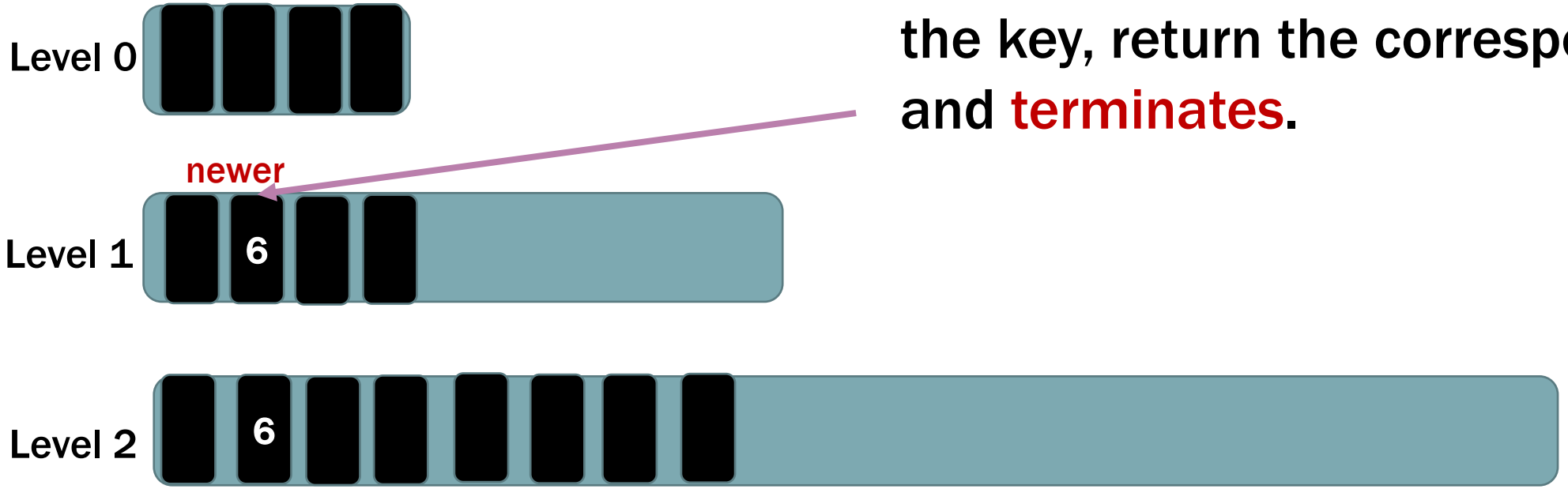
Consider Get(6)



Consider Get(6)

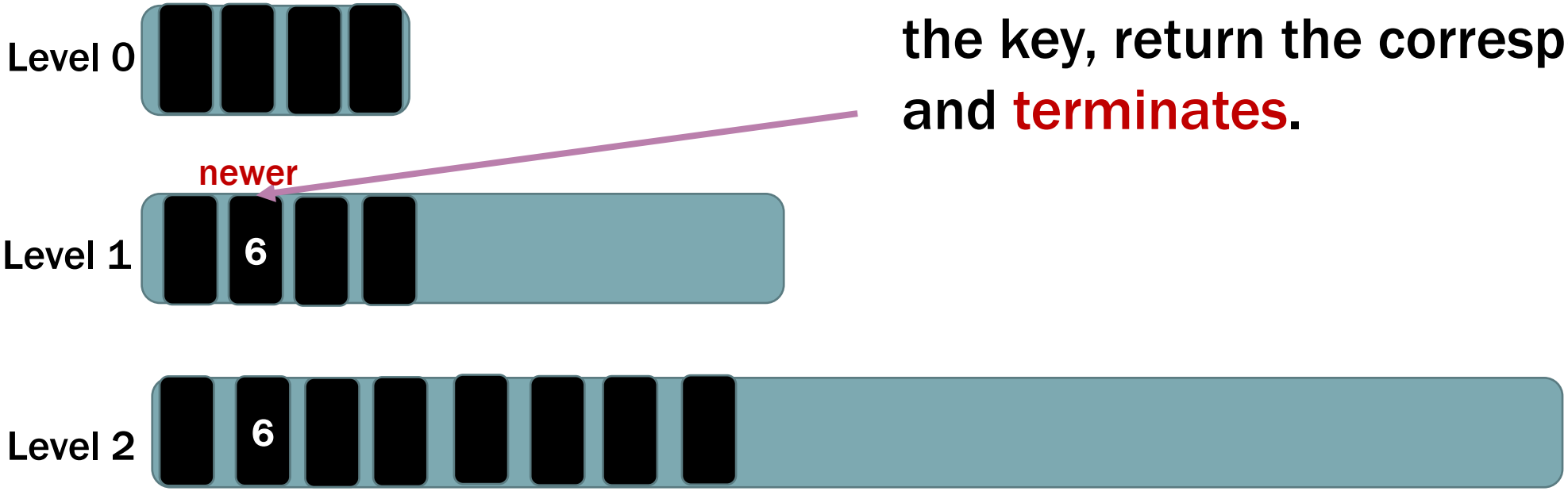


Consider Get(6)



In an L-level LSM-tree,
check from Level 0 to Level L, until we find the key, return the corresponding value, and terminates.

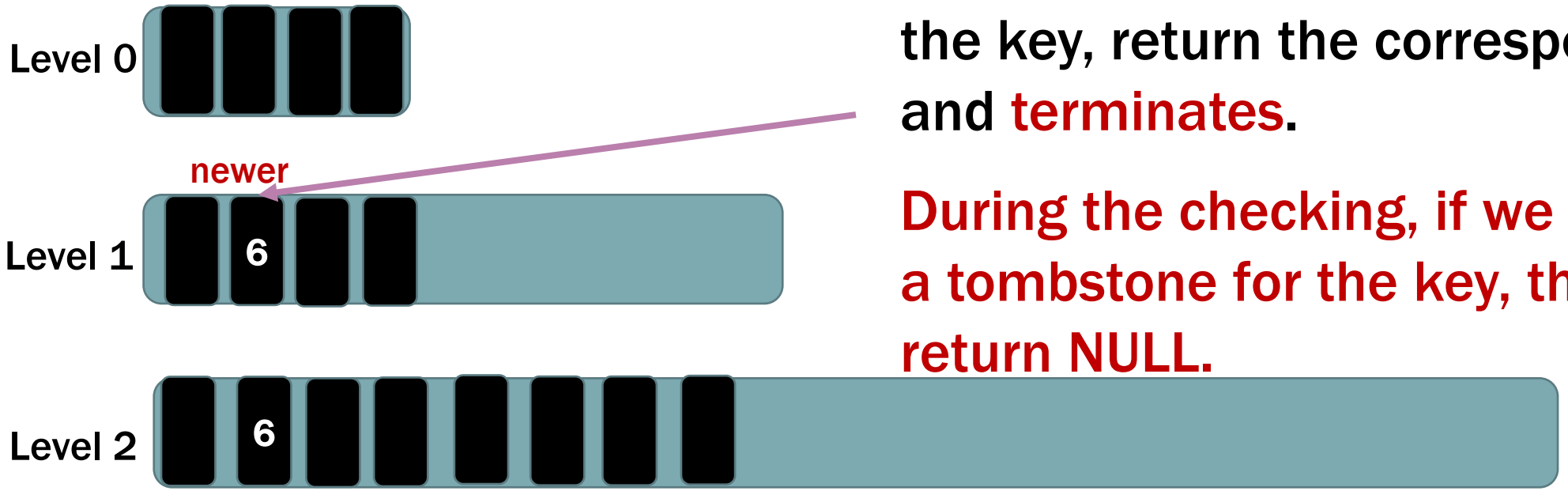
Consider Get(6)



In an L-level LSM-tree,
check from Level 0 to Level L, until we find the key, return the corresponding value, and terminates.

In this case, we do not need to check Level 2 at all.

Consider Get(6)

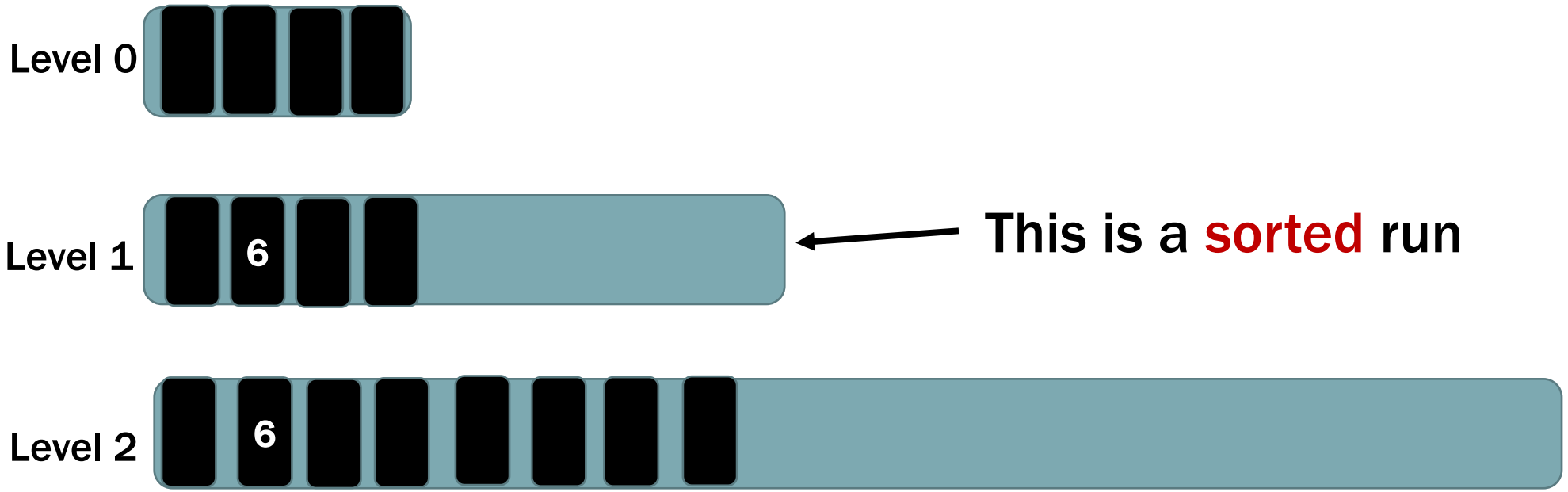


In an L-level LSM-tree,
check from Level 0 to Level L, until we find the key, return the corresponding value, and terminates.

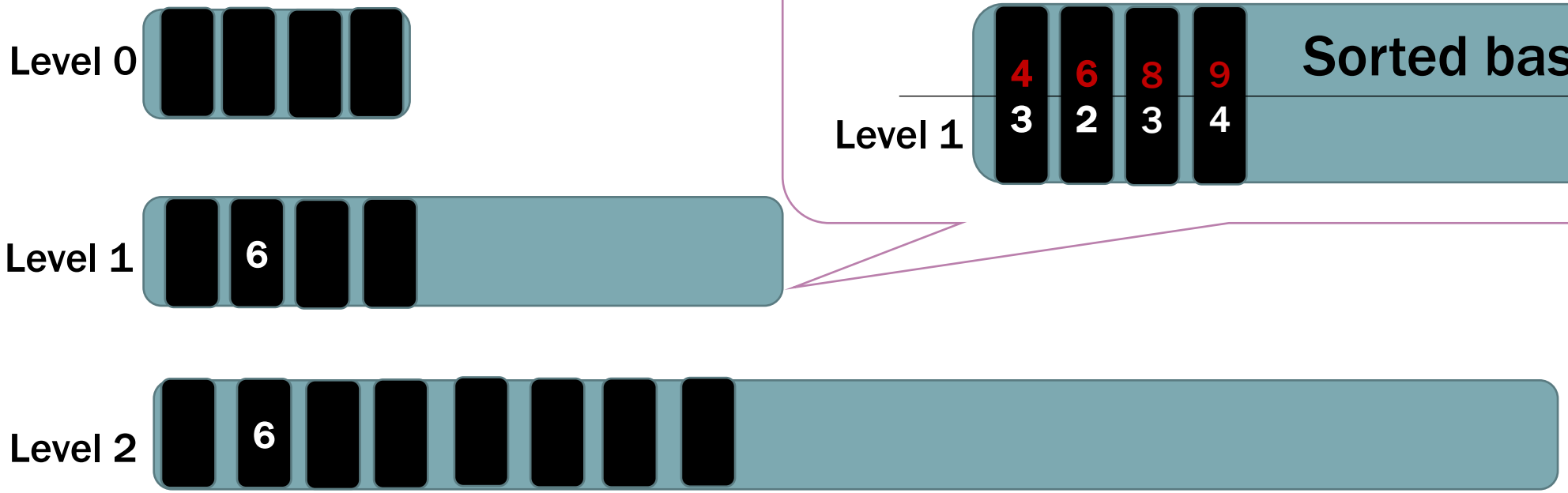
During the checking, if we find a tombstone for the key, then return NULL.

Consider Get(6)

How to check each level on disk (starting from Level 1)?

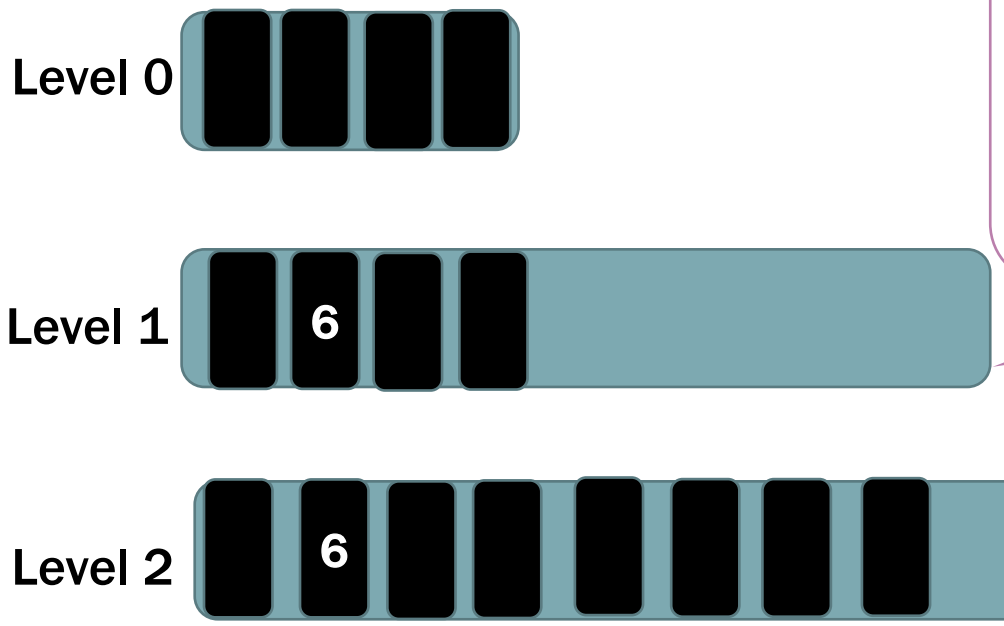


Consider Get(6)

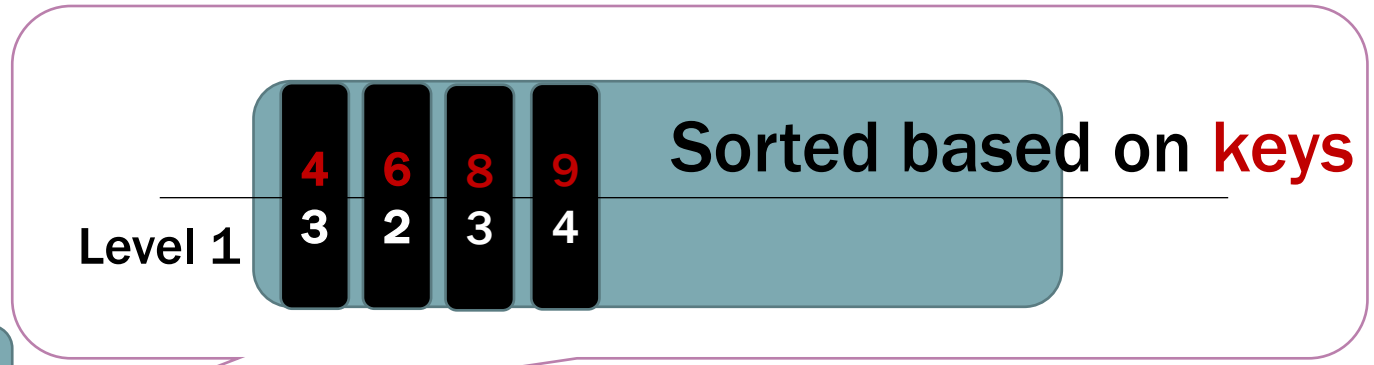


How to check each level?

Consider Get(6)



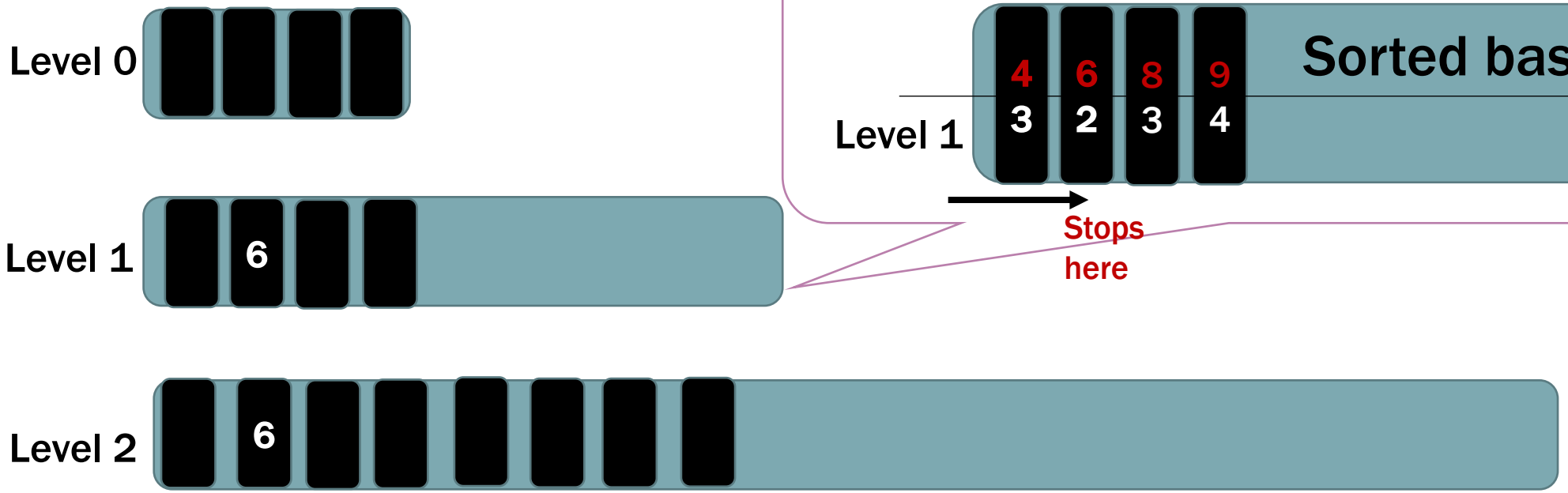
How to check each level?



First idea:

Searching from left to right, and stops when we find 6

Consider Get(6)

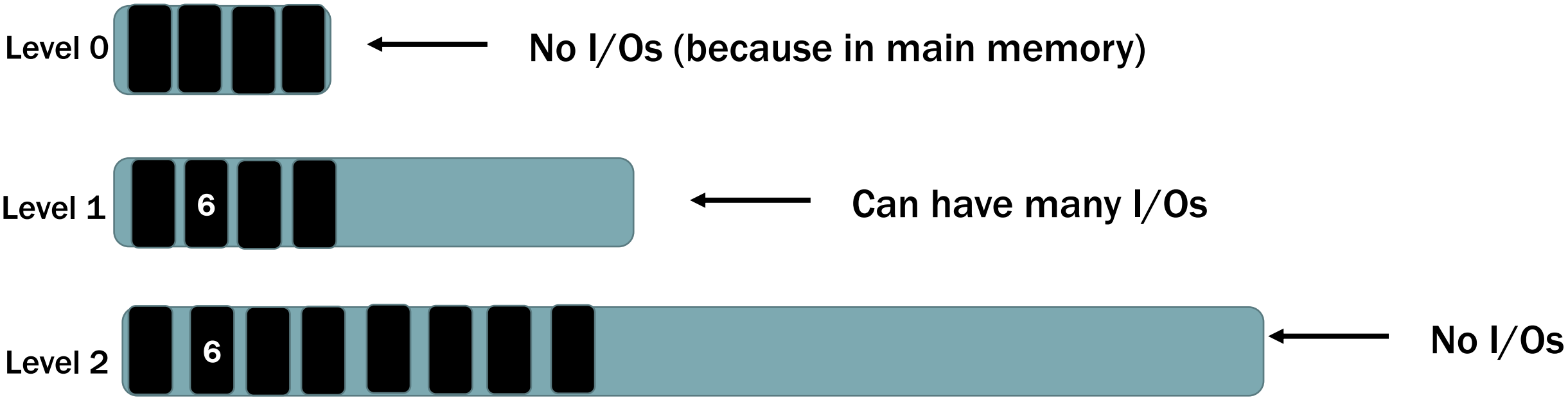


How to check each level?

THE COST FOR GET(6)

Consider Get(6)

How to check each level?

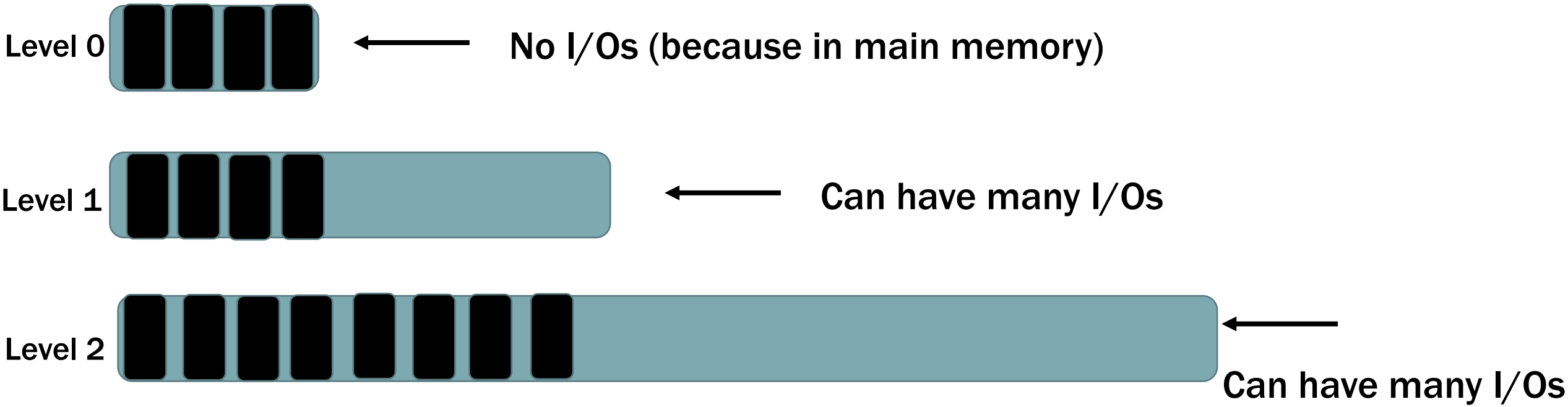


I/O cost: Number of disk page reads or writes

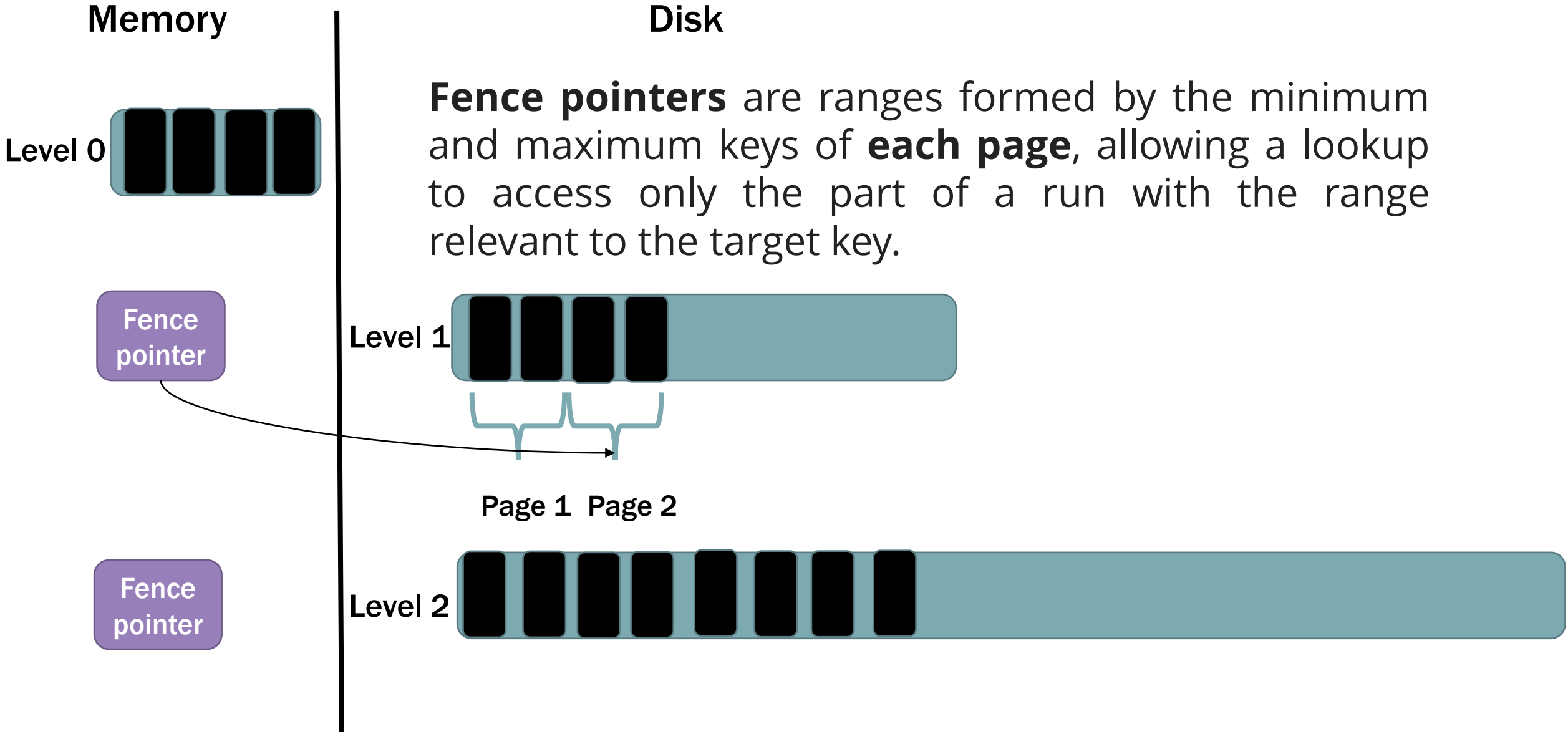
SOME OTHER POSSIBLE CASES (FOR OTHER KEYS)

Consider Get(6)

How to check each level?

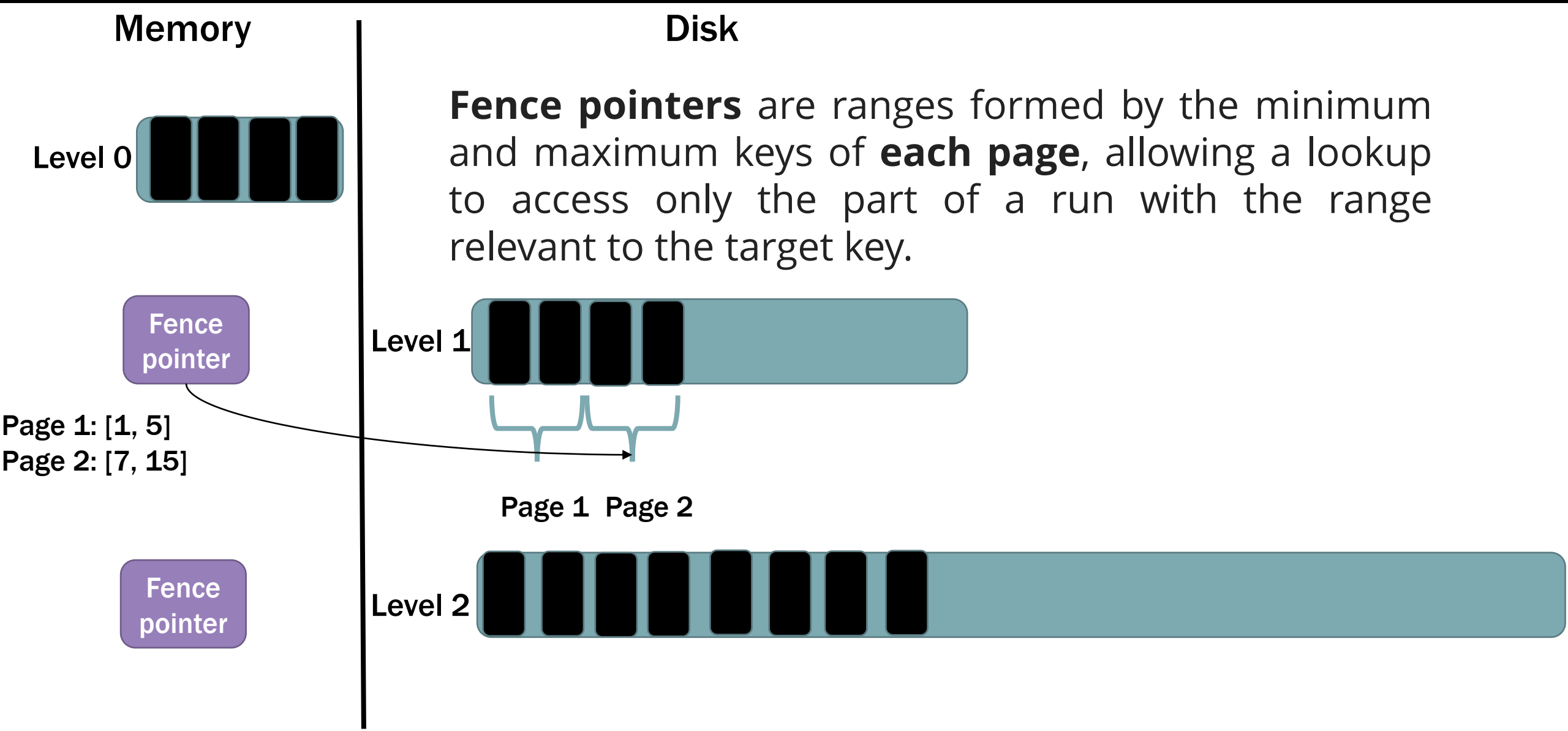


OPTIMIZATION – FENCE POINTERS



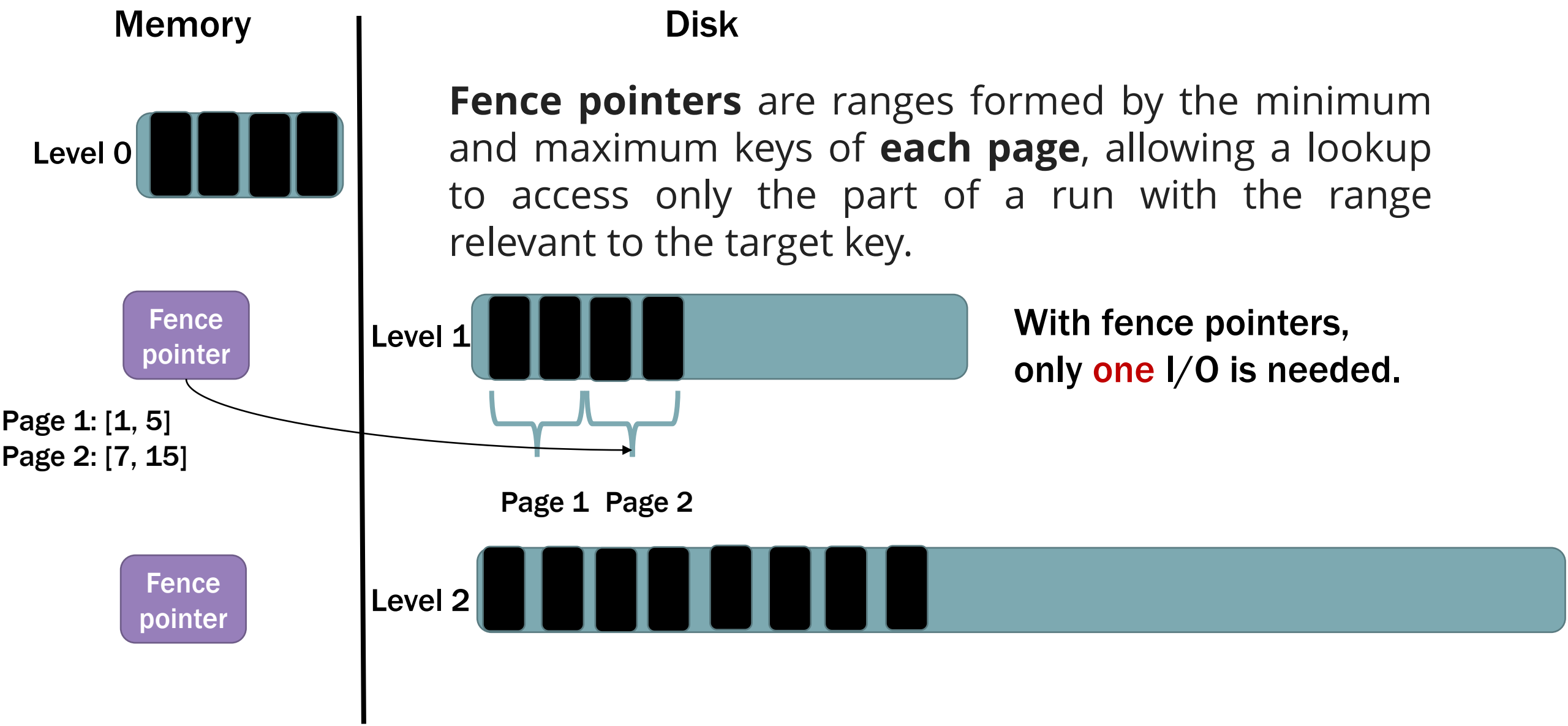
Fence pointers are ranges formed by the minimum and maximum keys of **each page**, allowing a lookup to access only the part of a run with the range relevant to the target key.

OPTIMIZATION – FENCE POINTERS



Fence pointers are ranges formed by the minimum and maximum keys of **each page**, allowing a lookup to access only the part of a run with the range relevant to the target key.

OPTIMIZATION – FENCE POINTERS



Fence pointers are ranges formed by the minimum and maximum keys of **each page**, allowing a lookup to access only the part of a run with the range relevant to the target key.

With fence pointers, only one I/O is needed.

OPTIMIZATION – BLOOM FILTERS