BIG DATA MANAGEMENT

CE/CZ4123

KEY-VALUE STORE LSM-TREE BASICS

Siqiang Luo

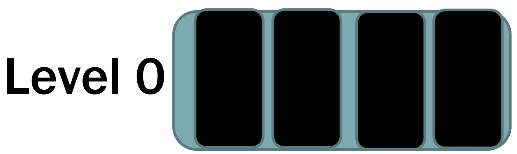
Assistant Professor

IN PREVIOUS LECTURES

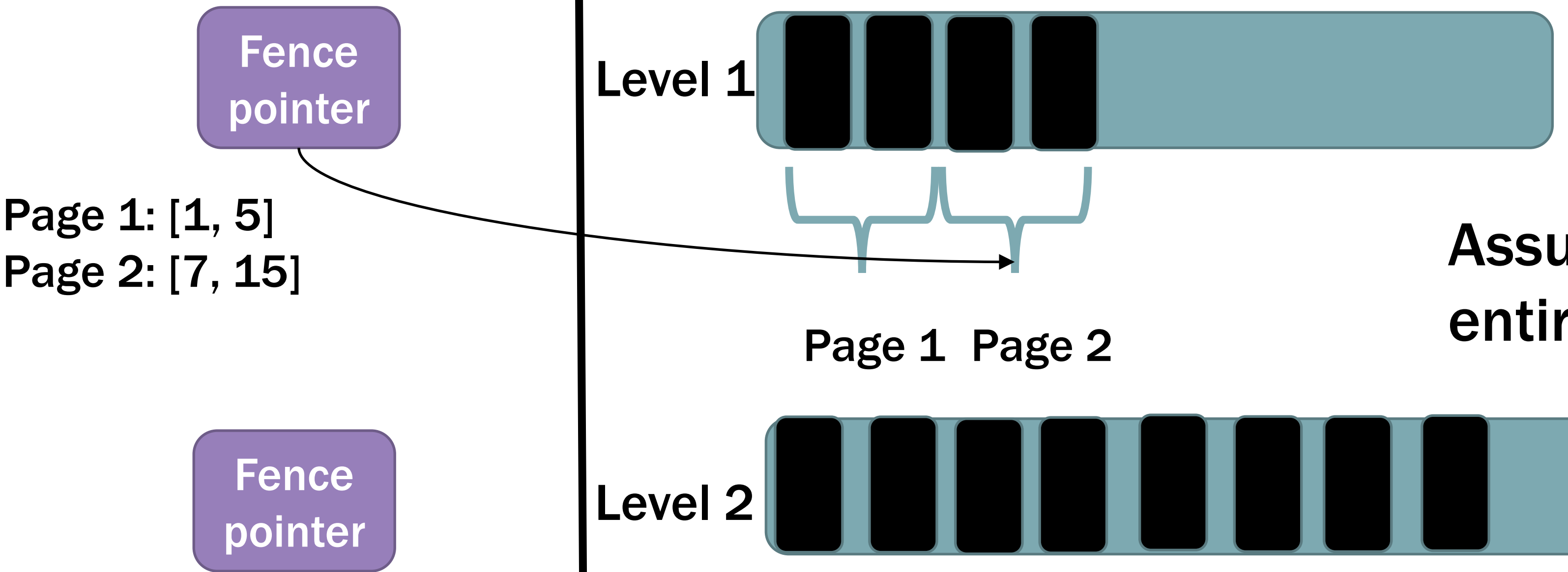
- We discuss the basic structure of an LSM-tree
- We introduce the main functions of an LSM-tree
  - (e.g., Get(), Put(), Delete())
- We introduce
  - (e.g., fence pointers, Bloom filters)

OPTIMIZATION – FENCE POINTERS

Memory | **Disk**



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.

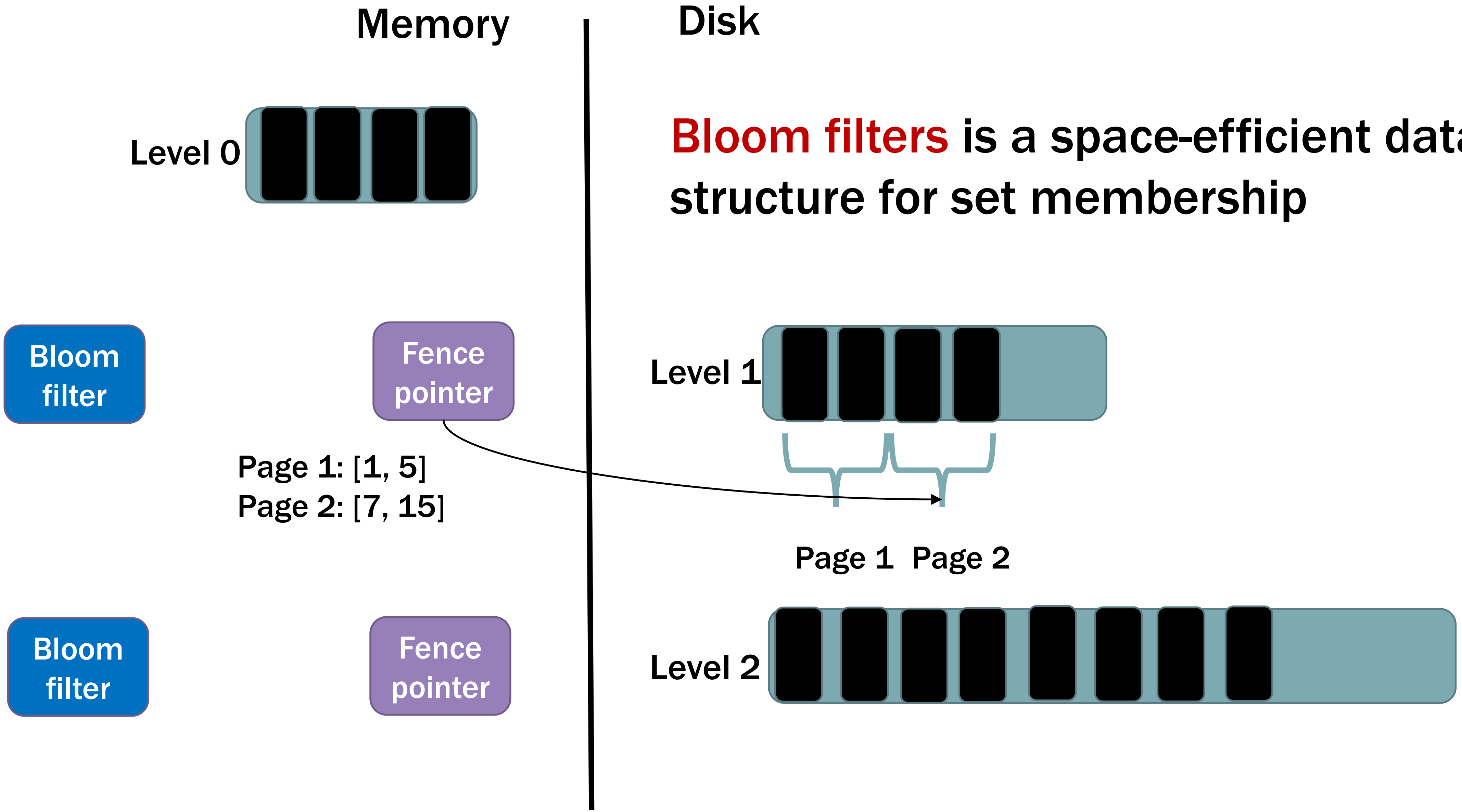


With fence pointers, only one I/O is needed.

Assume that a key-value pair is always entirely stored within a disk page.

OPTIMIZATION – BLOOM FILTERS

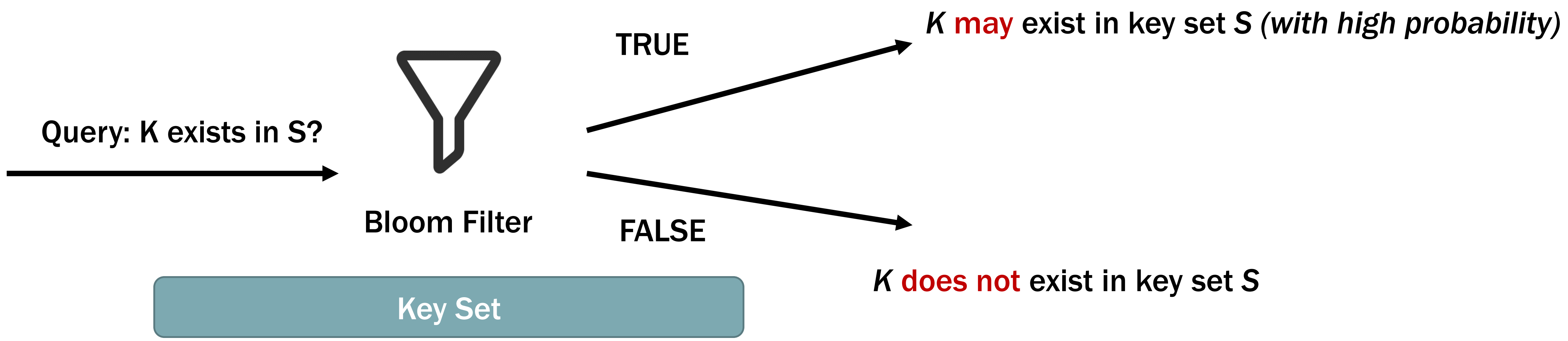
Bloom filters is a space-efficient data structure for set membership



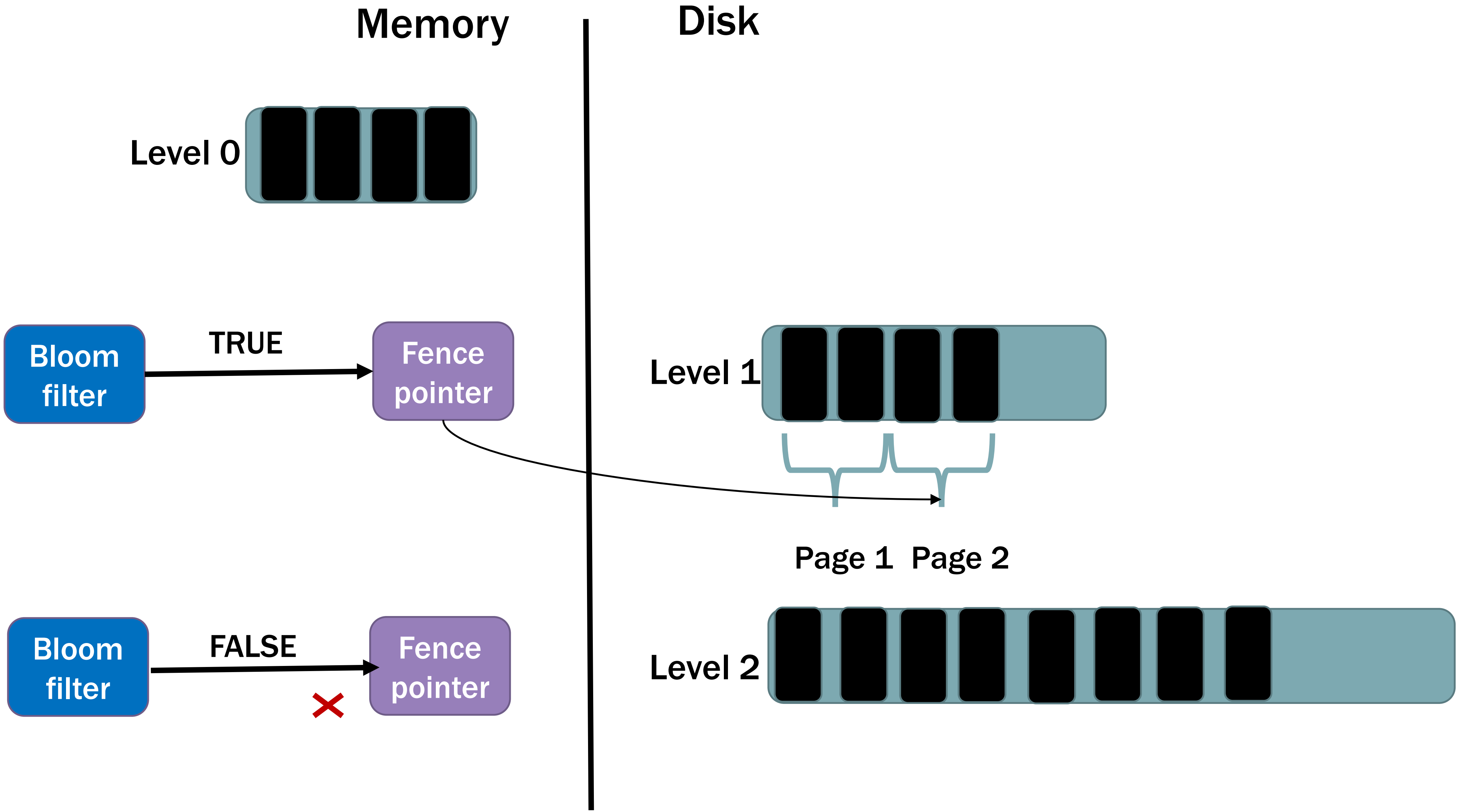
BLOOM FILTER

1. Stored in main memory
2. Built on a set *S* of keys
3. Given a key *K*, the Bloom filter answers TRUE or FALSE for key *K*
4. If it answers FALSE, it means the key *K* does not exist in key set *S*
5. If it answers TRUE, it means the key *K* may exist in key set *S*, and it is still possible that the key *K* does not exist in key set *S*.
6. FPR (False-Positive Rate) is the probability that the filter returns TRUE for a key *K*, but actually *K* does not exist in set *S*. We usually use *P* to denote FPR. Clearly, *P* is in [0, 1] (e.g., *P*=0.3)

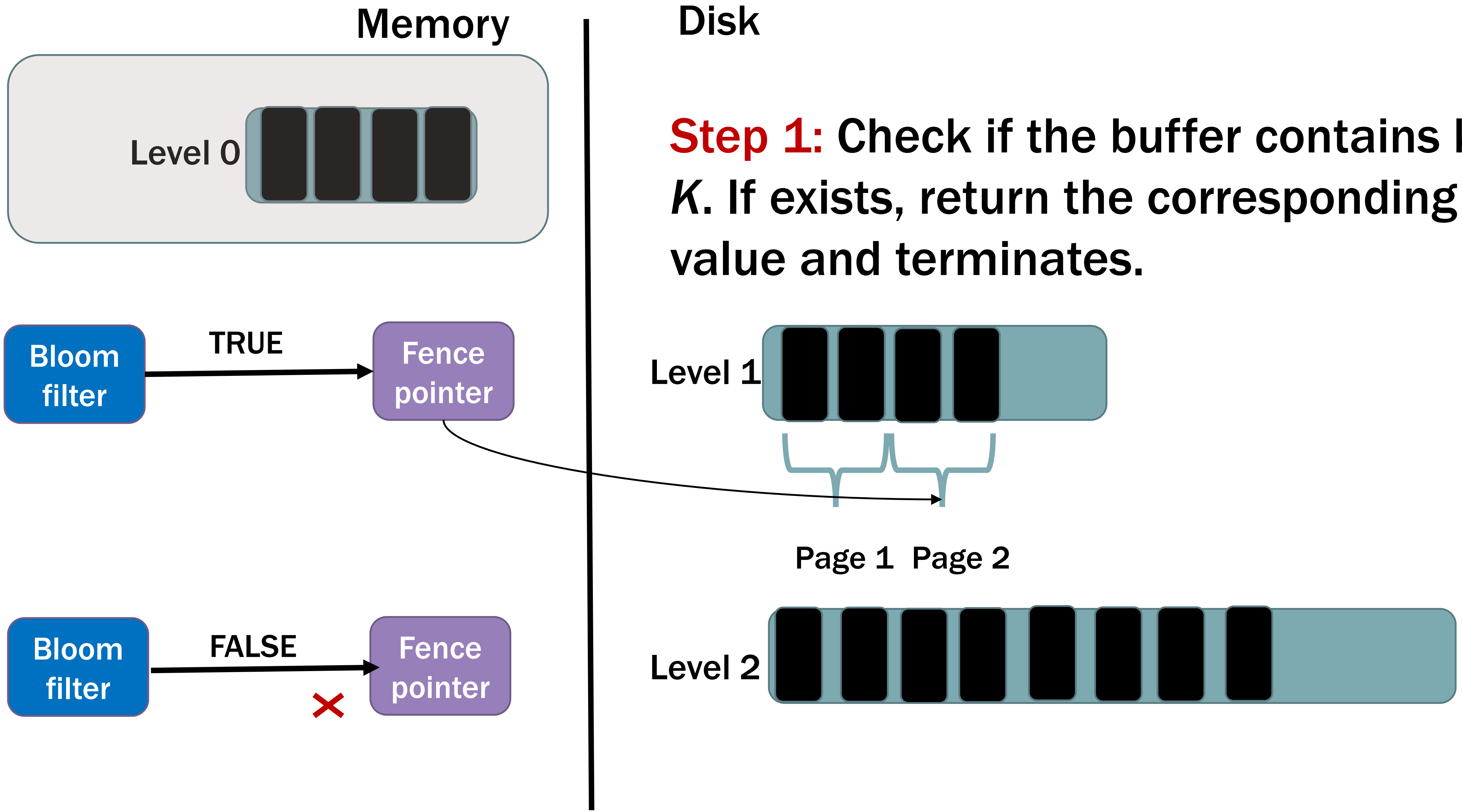
ILLUSTRATION OF BLOOM FILTER



WHY BLOOM FILTER IS HELPFUL?

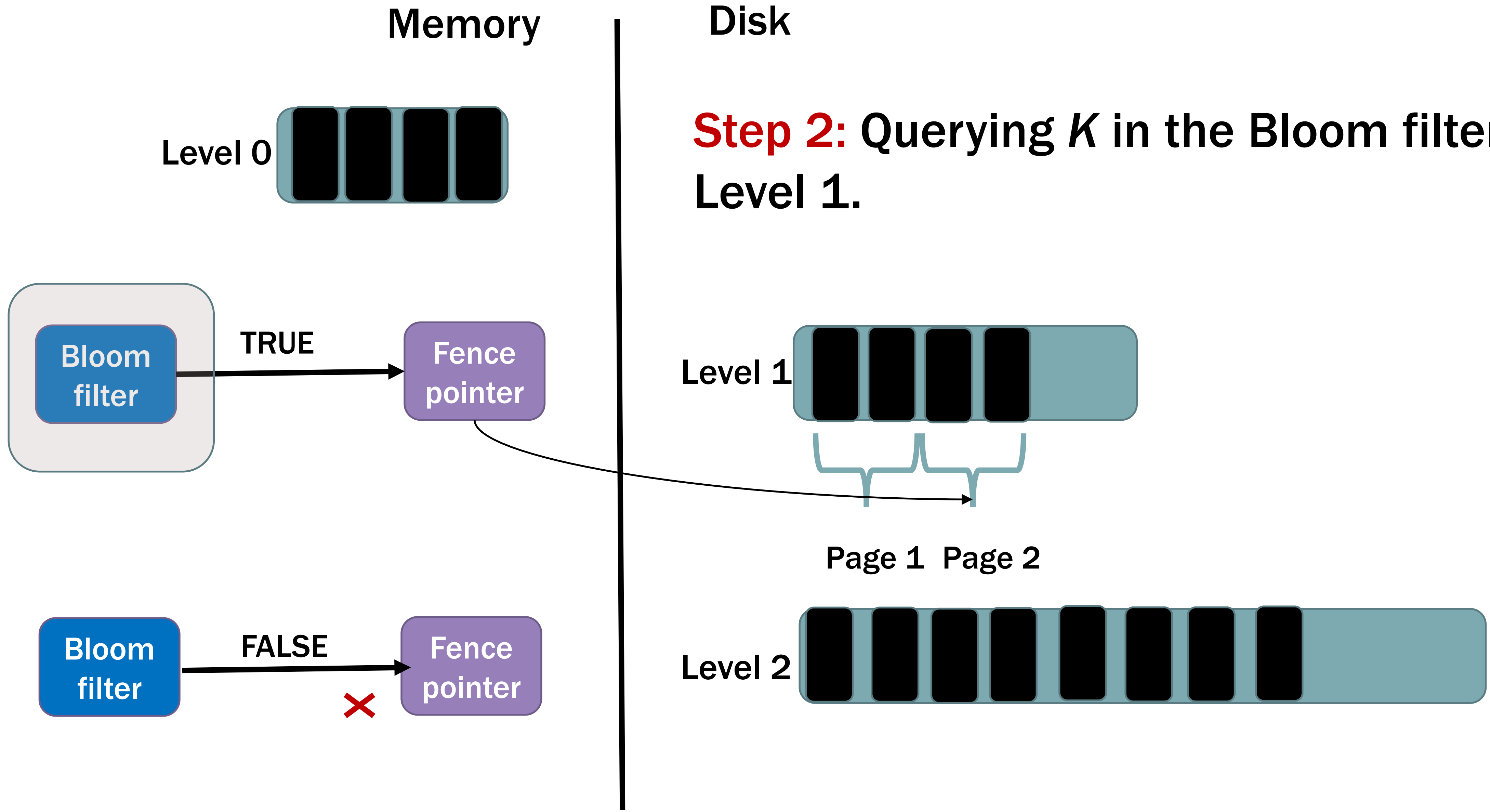


THE PROCEDURE OF GET(K) – WITH FENCE POINTERS AND BLOOM FILTERS



Step 1: Check if the buffer contains key *K*. If exists, return the corresponding value and terminates.

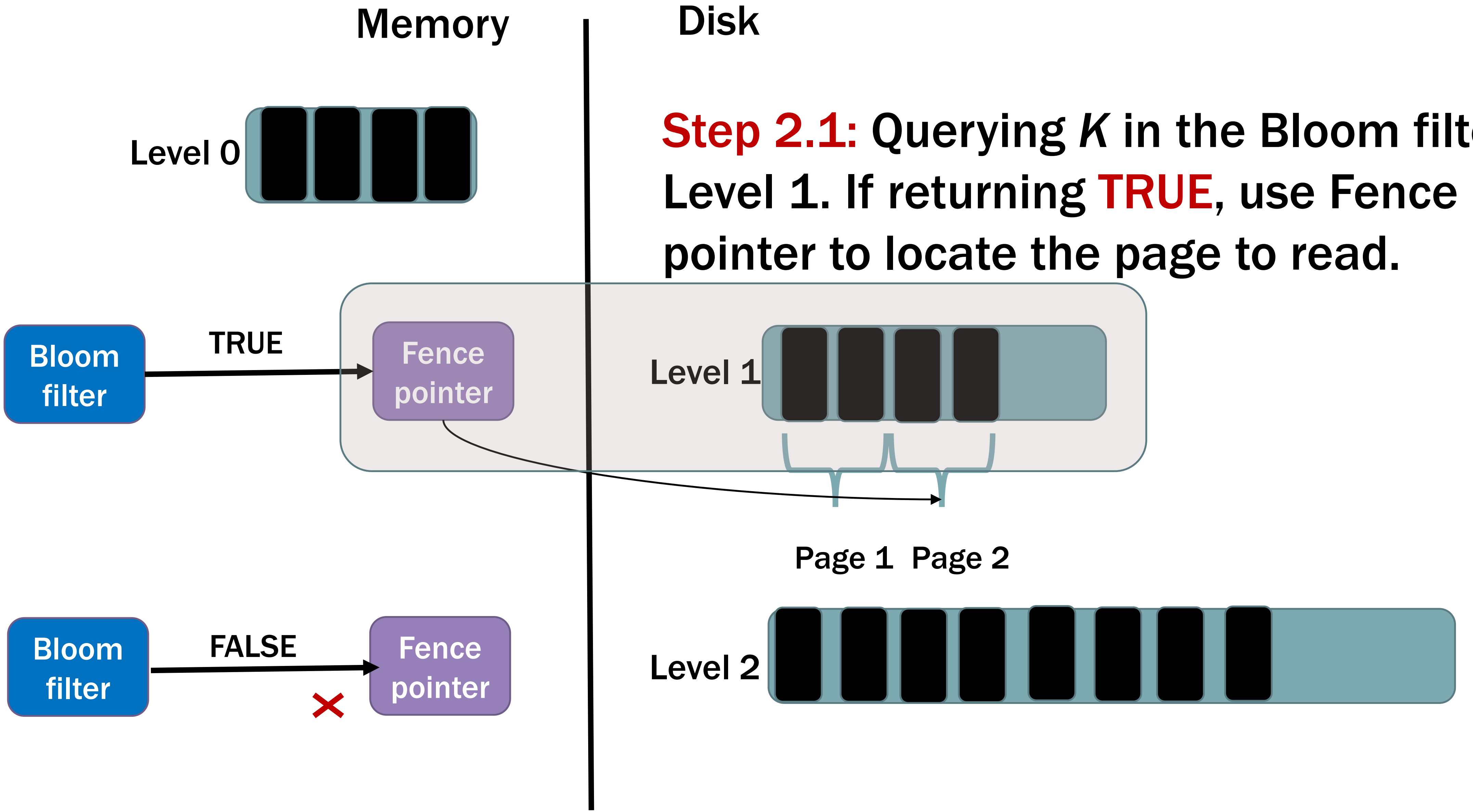
THE PROCEDURE OF GET(K) – WITH FENCE POINTERS AND BLOOM FILTERS



Step 2: Querying *K* in the Bloom filter of Level 1.

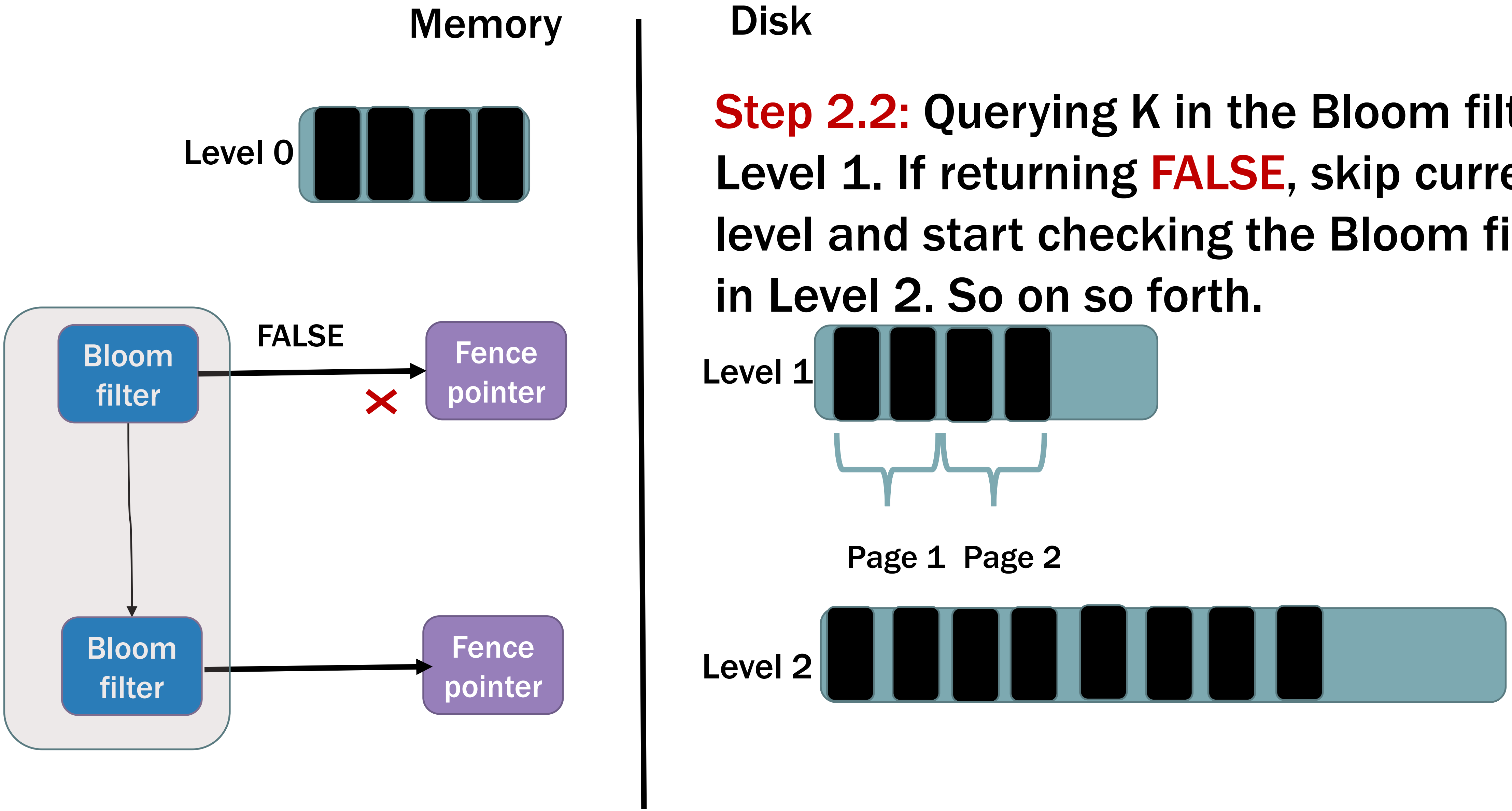
THE PROCEDURE OF GET(K) – WITH FENCE POINTERS AND BLOOM FILTERS

Step 2.1: Querying *K* in the Bloom filter of Level 1. If returning **TRUE**, use Fence pointer to locate the page to read.

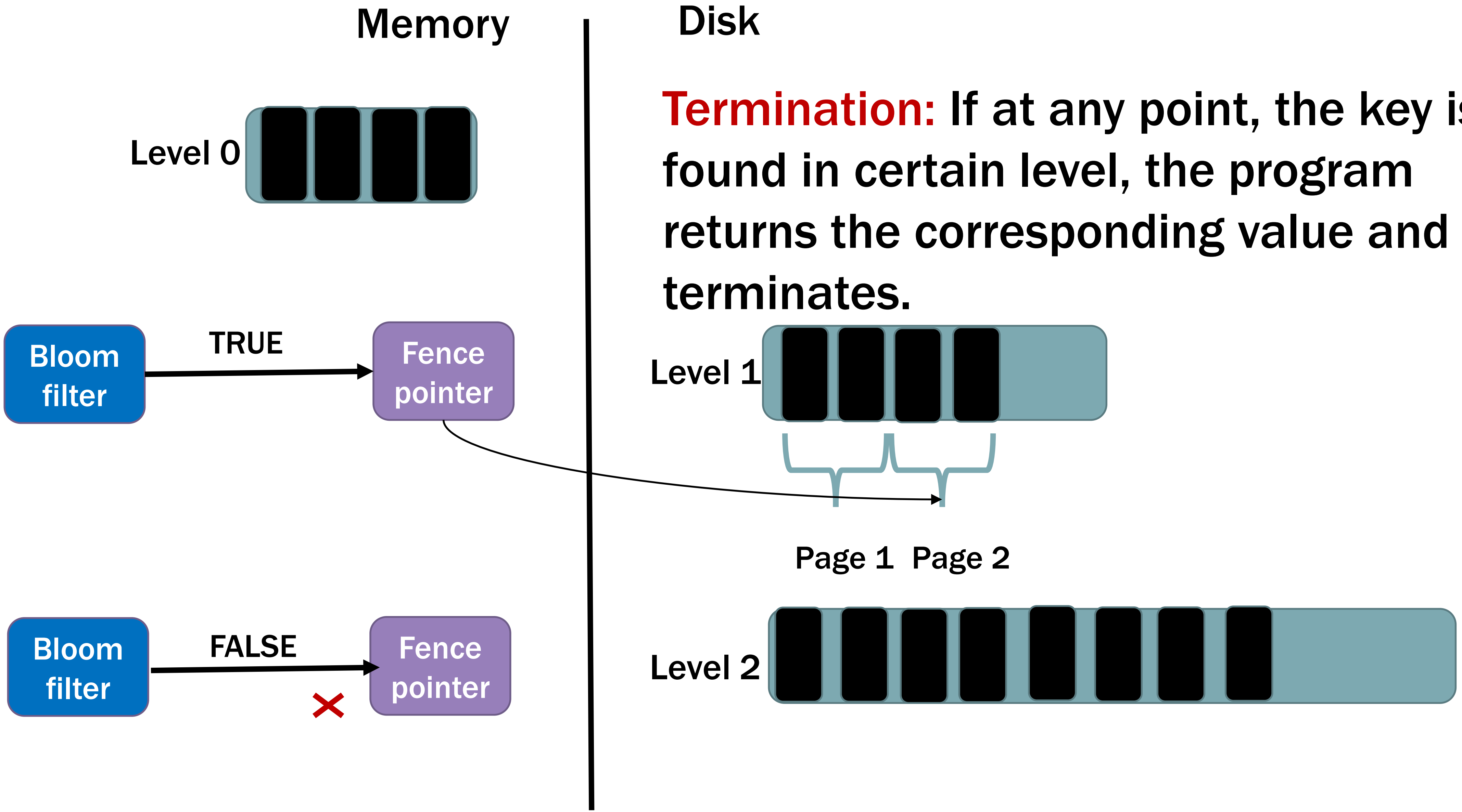


THE PROCEDURE OF GET(K) – WITH FENCE POINTERS AND BLOOM FILTERS

Step 2.2: Querying K in the Bloom filter of Level 1. If returning **FALSE**, skip current level and start checking the Bloom filter in Level 2. So on so forth.



THE PROCEDURE OF GET(K) – WITH FENCE POINTERS AND BLOOM FILTERS



SUMMARY

1. Check buffer level (Level 0)
2. Starting from Level 1, always check its Bloom filter first, and then its Fence pointer.
3. If Bloom filter returns FALSE, then do not access the fence pointer and directly go to the next level for checking.
4. If Bloom filter returns TRUE, then access the fence pointer to fetch the corresponding page in that level. If the key is found, return the value and terminate the program; otherwise go to the next level for checking.
5. If the key has not been found in any level, return NULL (or empty set) and terminate the program.

INTERNAL DESIGNS OF BLOOM FILTER

The main purpose of the filter:

- Built on a set of keys S={K_1, K_2, …, K_n}, where each key is from a large universe U.
- If the Bloom filter returns FALSE, it is 100% correct that the key is NOT in S. Also, with HIGH probability, it can tell if a key is in S.
- Space efficient: on average, each key only occupies a few bits in Bloom filter.
- Inserting a new element into the Bloom filter should be fast
- Querying a key from the Bloom filter should be fast

THE CORE OF BLOOM FILTER – HASH FUNCTION

A hash function h maps a uniformly at random chosen key $x \in U$ to an integer from $R_m=[0, m-1]$ such that each element in R_m is mapped with equal probability.

A Bloom filter is a bit vector B of m bits, with k independent hash functions $h_1, \ldots, h_k$

- Initially all the m bits are 0.
- Insert x into S: compute $h_1(x), \ldots, h_k(x)$ and set $B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)] = 1$.
- Query if $x \in S$: Compute $h_1(x), \ldots, h_k(x)$. If $B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)]=1$, then answer TRUE, else answer FALSE.

EXAMPLE

- m=5, k=2
- $h_1(x)$ = x mod 5
- $h_2(x)$ = $(2x+3)$ mod 5
- Initially B[0]=B[1]=B[2]=B[3]=B[4]=B[5]=0
- Then insert 9 and 11

	$h_1(x)$	$h_2(x)$	B				
Initialize:			0	0	0	0	0
Insert 9:	4	1	0	1	0	0	1
Insert 11:	1	0	1	1	0	0	1

EXAMPLE

Now query 15 and 16

	$h_1(x)$	$h_2(x)$	Answer
Query 15:	0	3	No, not in B (correct answer)
Query 16:	1	0	Yes, in B (wrong answer: false positive)

ANALYSIS

- When m/n is fixed (m/n is often called bits-per-key), the optimal k is $\ln 2 \times (m/n)$ (See here for proof if you are interested)

- Then, the optimal FPR is about $0.6185^{m/n}$

- So, larger m means small FPR (approaches to 0); smaller m means higher FPR (approaches to 1).

SUMMARY OF LSM-TREE

- ❑ Multi-level structured
- ❑ Out-of-place updates (delete acts as a special insert)
- ❑ Fence pointers → reduce I/Os of Get()
- ❑ Bloom filters → reduce I/Os of Get()

CLARIFICATIONS

In practice, fence pointers can be implemented in different ways.

Option 1: containing the min/max of each page;

Option 2: containing only the min (or max) of each page;

For analysis purpose, we reasonably assume that when using Fence pointers, it always incurs one I/O.

VARIANTS OF LSM-TREE

- Leveled LSM-tree (or Leveling LSM-tree)
  - The LSM-tree with leveling merge policy.
- Tiered LSM-tree (or Tiering LSM-tree)
  - The LSM-tree with tiering merge policy.
- Until now, the LSM-tree we introduce belongs to Leveling LSM-tree.
- Next, we introduce what is Tiering LSM-tree

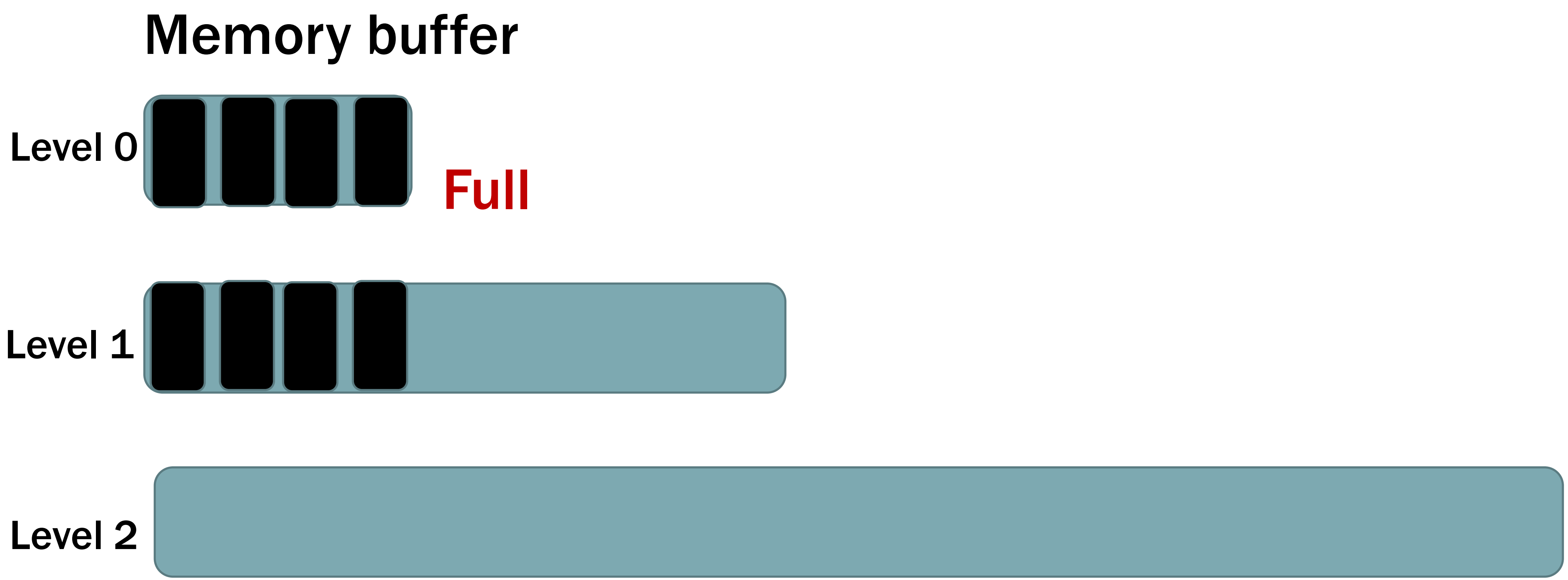
LEVELING LSM-TREE VS TIERING LSM-TREE

- The difference lies in the way of merging a full level to its next level

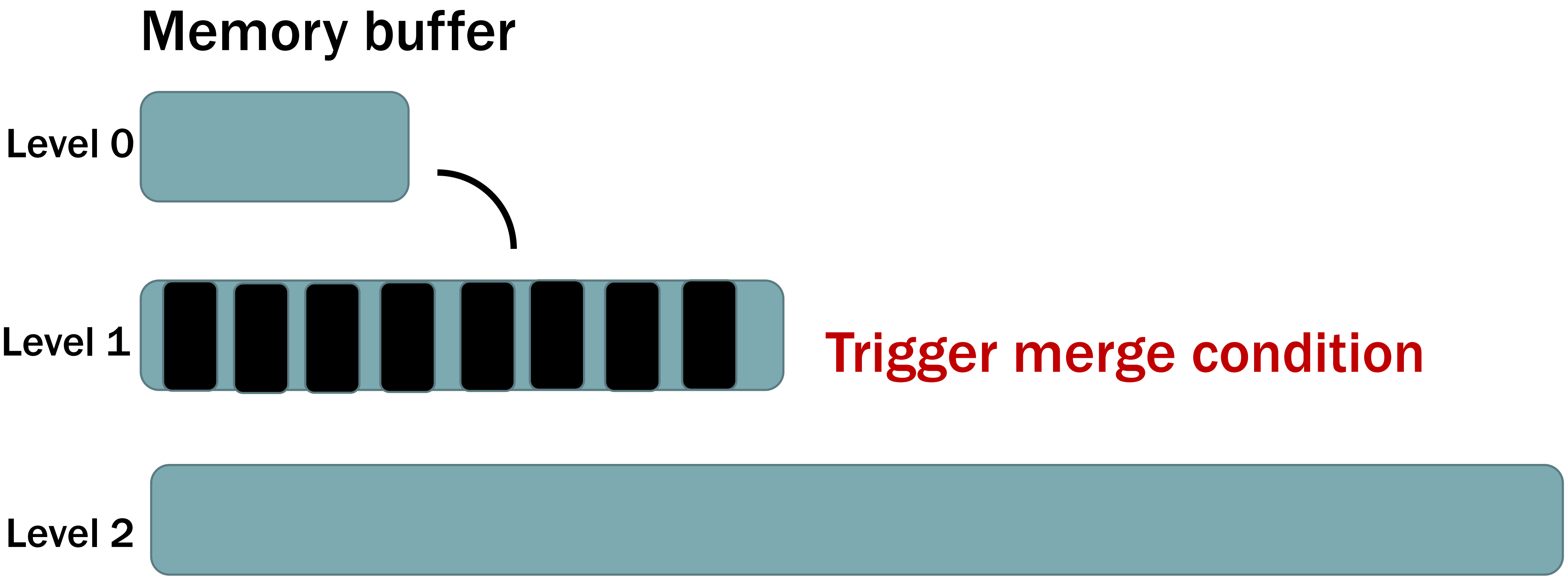
- Leveling LSM-tree:
  - When a level needs to be merged, always sort-merge to the next level

- Tiering LSM-tree:
  - When a level needs to be merged, will not be sort-merged with the next level, but sort themselves and put it as a *tier* stored in the next level

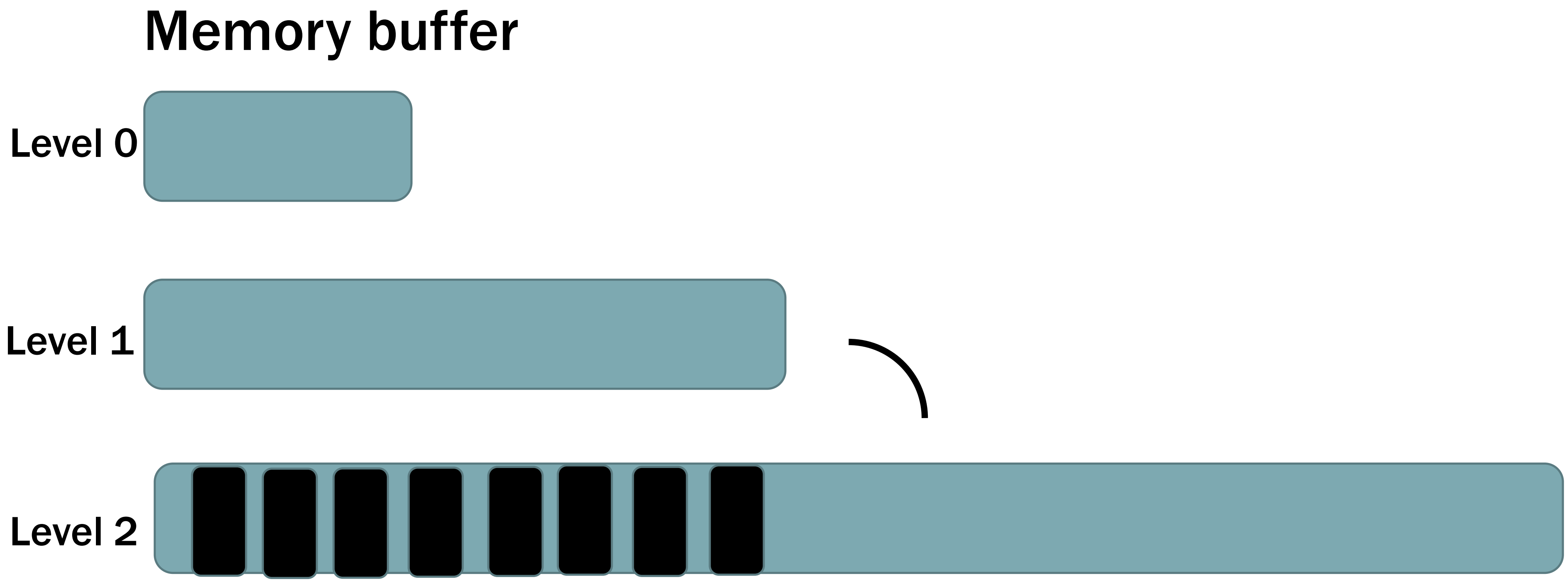
LEVELING LSM-TREE EXAMPLE



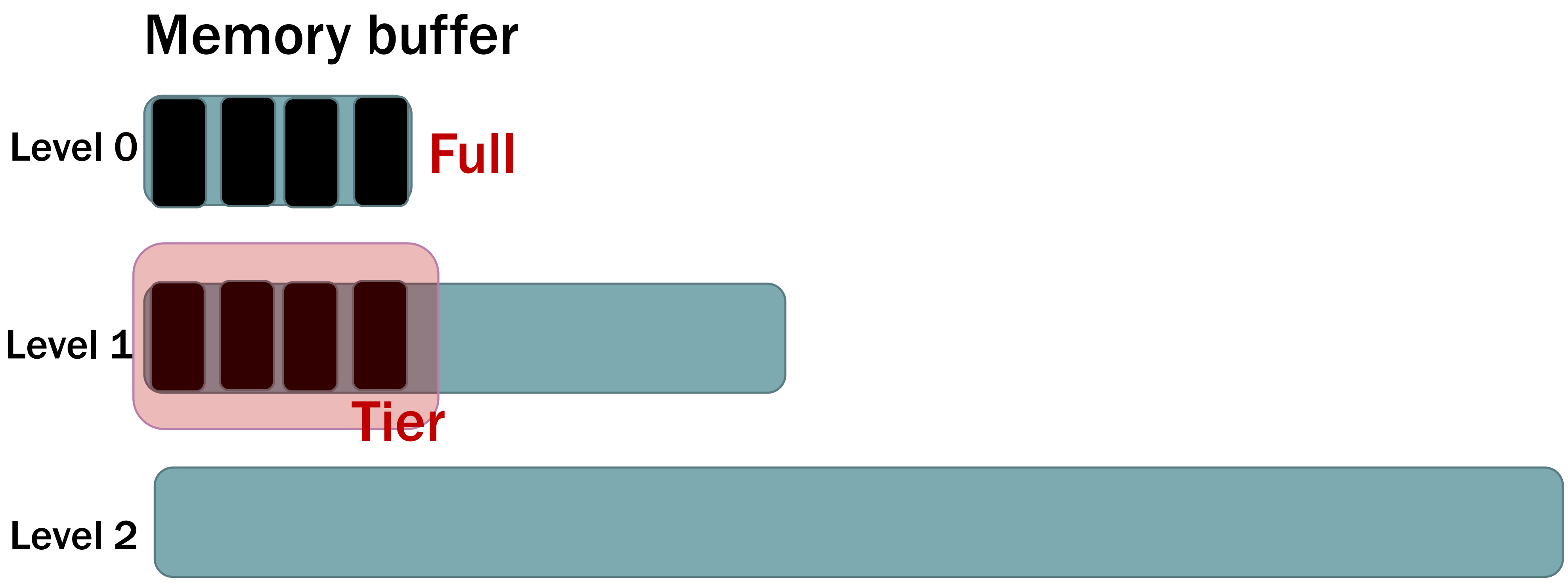
LEVELING LSM-TREE EXAMPLE



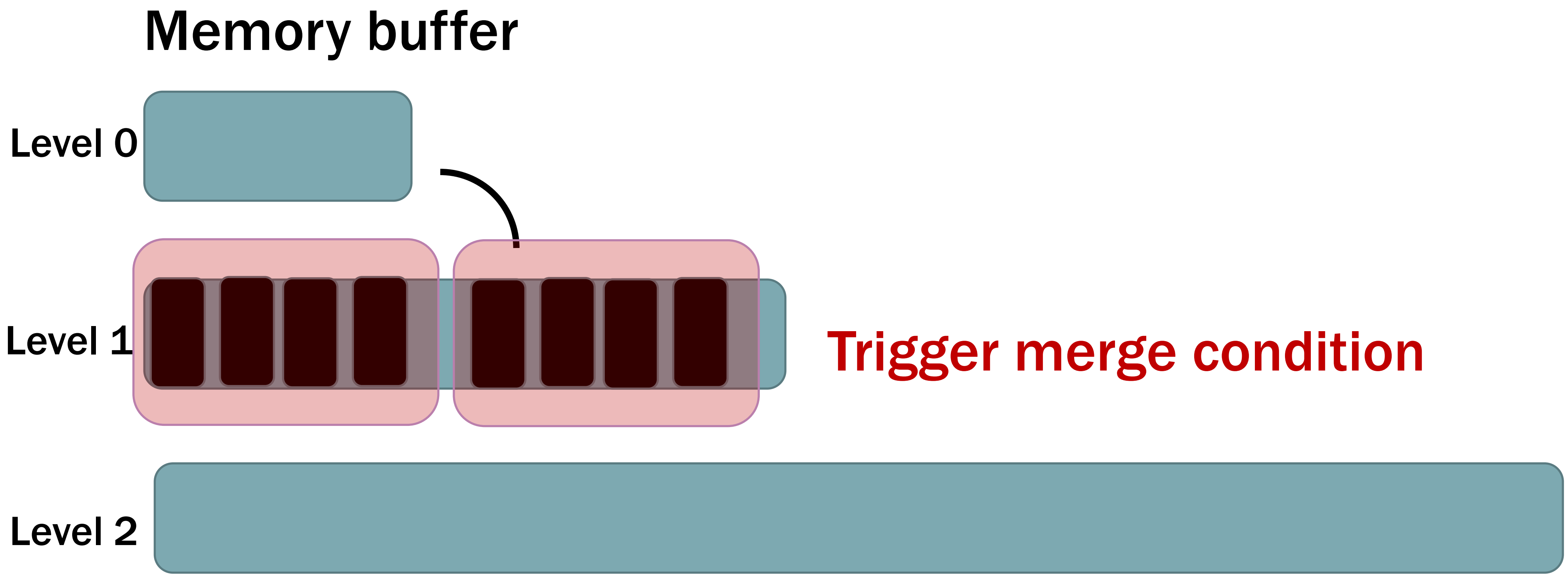
LEVELING LSM-TREE EXAMPLE



TIERING LSM-TREE EXAMPLE

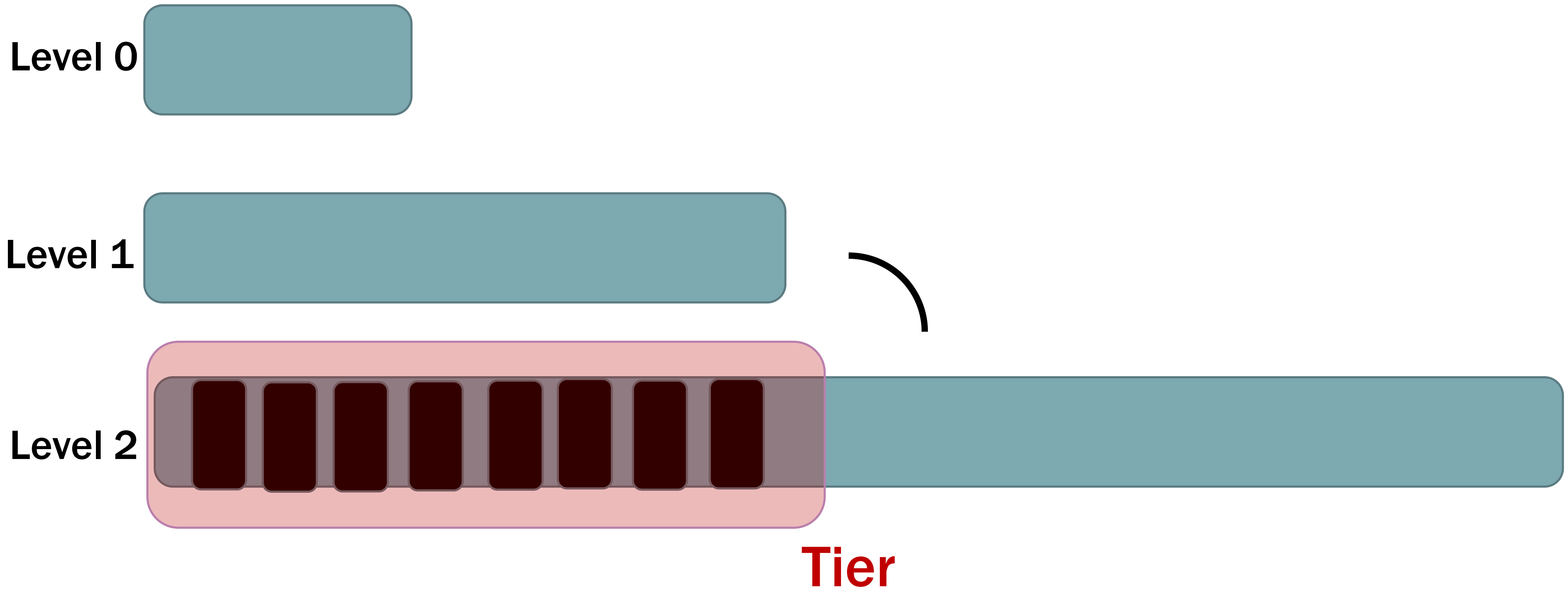


TIERING LSM-TREE EXAMPLE



TIERING LSM-TREE EXAMPLE

Memory buffer

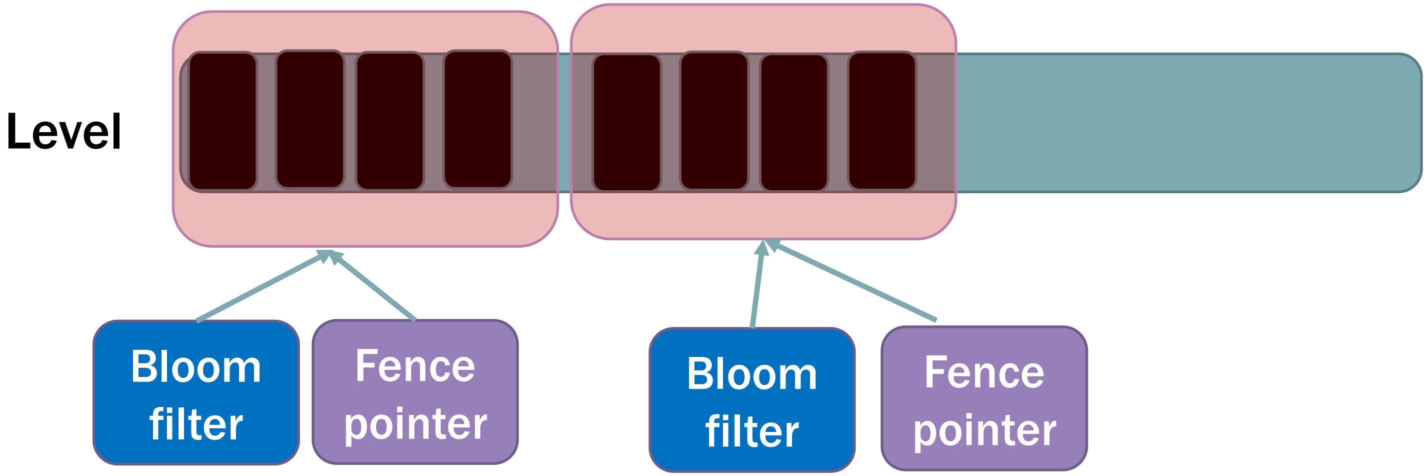


SOME PROPERTIES

- A tier in Level i roughly has the size of the capacity of Level (i-1)
- Usually, in tiering LSM-tree, starting from Level 1, the merge is triggered when there are *T* tiers in it, where *T* is the size ratio.
- Question: In each level of tiering LSM-tree, is a key unique?

FENCE POINTERS AND BLOOM FILTERS IN TIERING LSM-TREES

Bloom filter and fence pointers are built for each tier of each disk level (except Level 0)



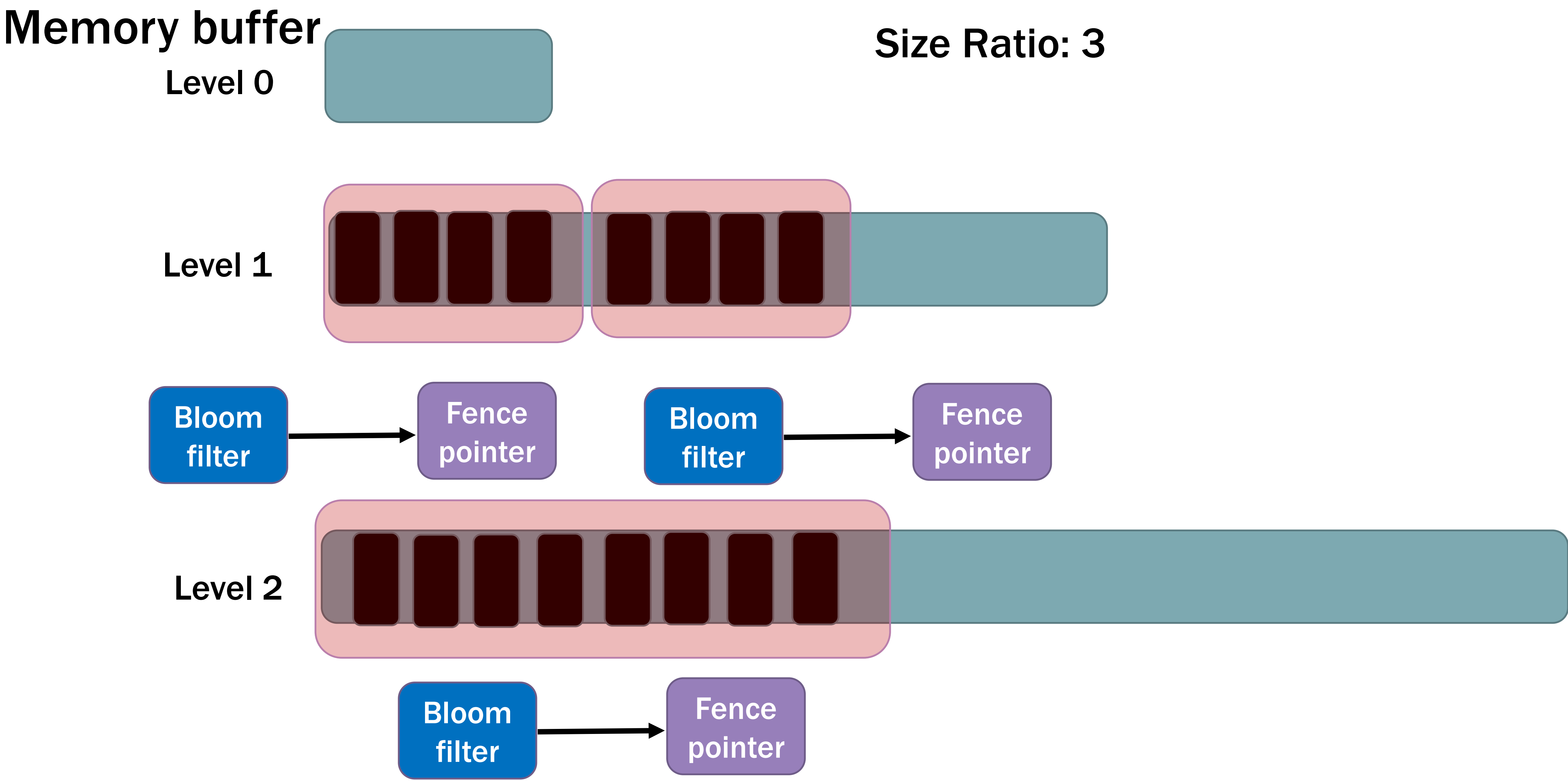
PROS AND CONS OF TIERING LSM-TREE

- Advantages:
  - Avoid costly sort merges
  - Put/Delete is faster

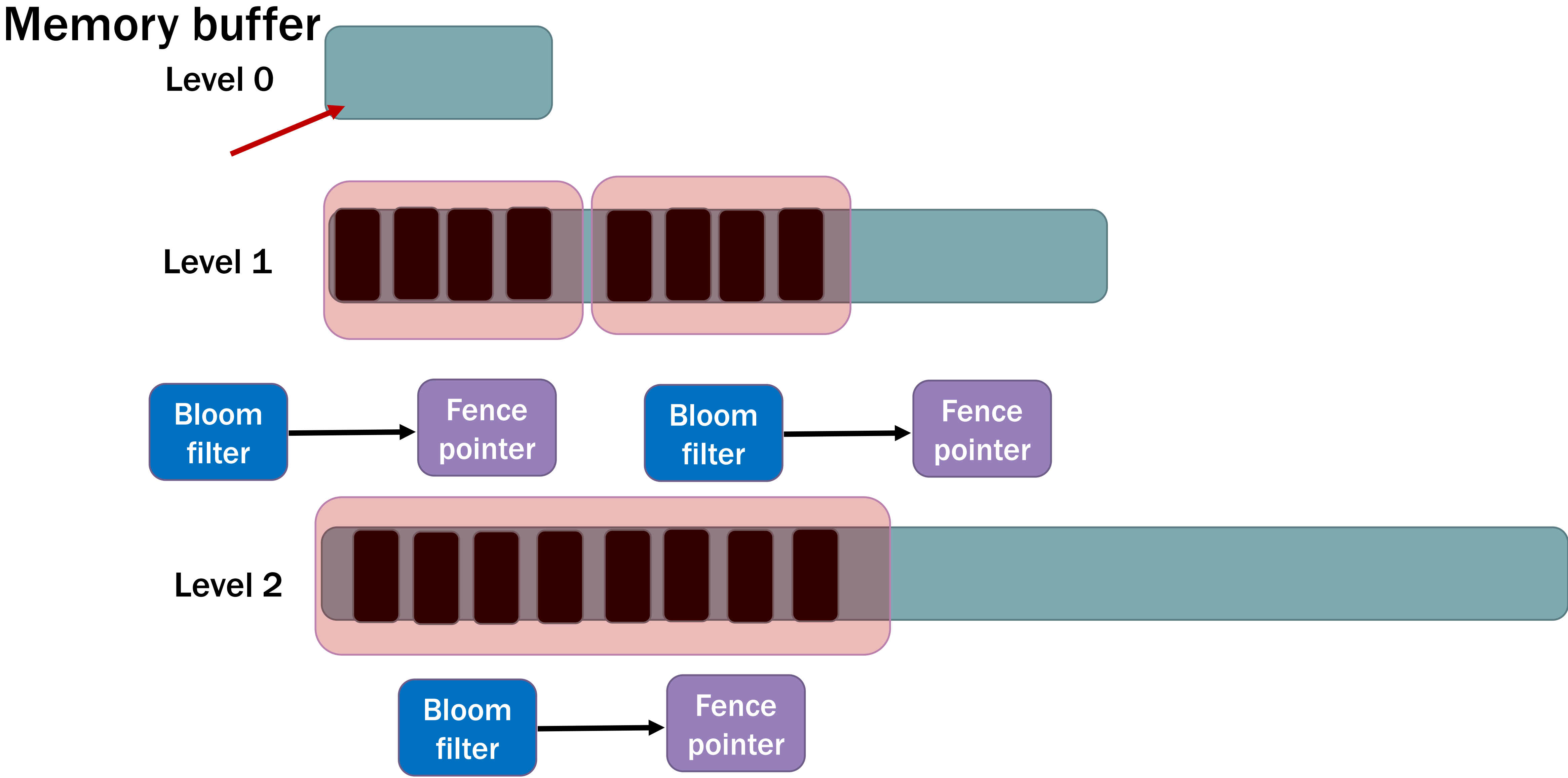
- Disadvantages:
  - Get is slower

- Summary
  - Leveling LSM-tree: faster data reads, slower data writes
  - Tiering LSM-trees: slower data reads, faster data writes

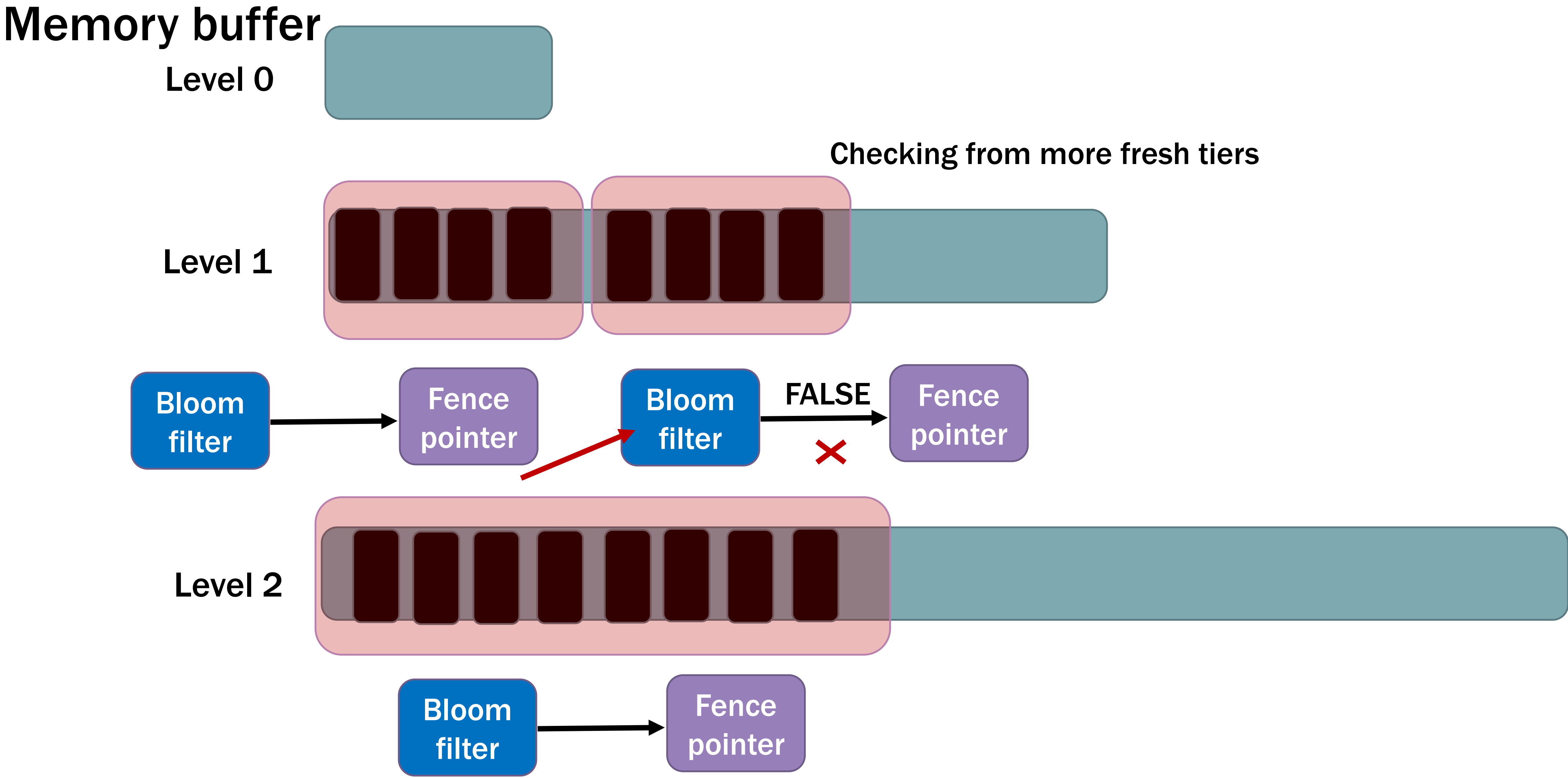
PERFORMING GET(K) IN TIERING LSM-TREE



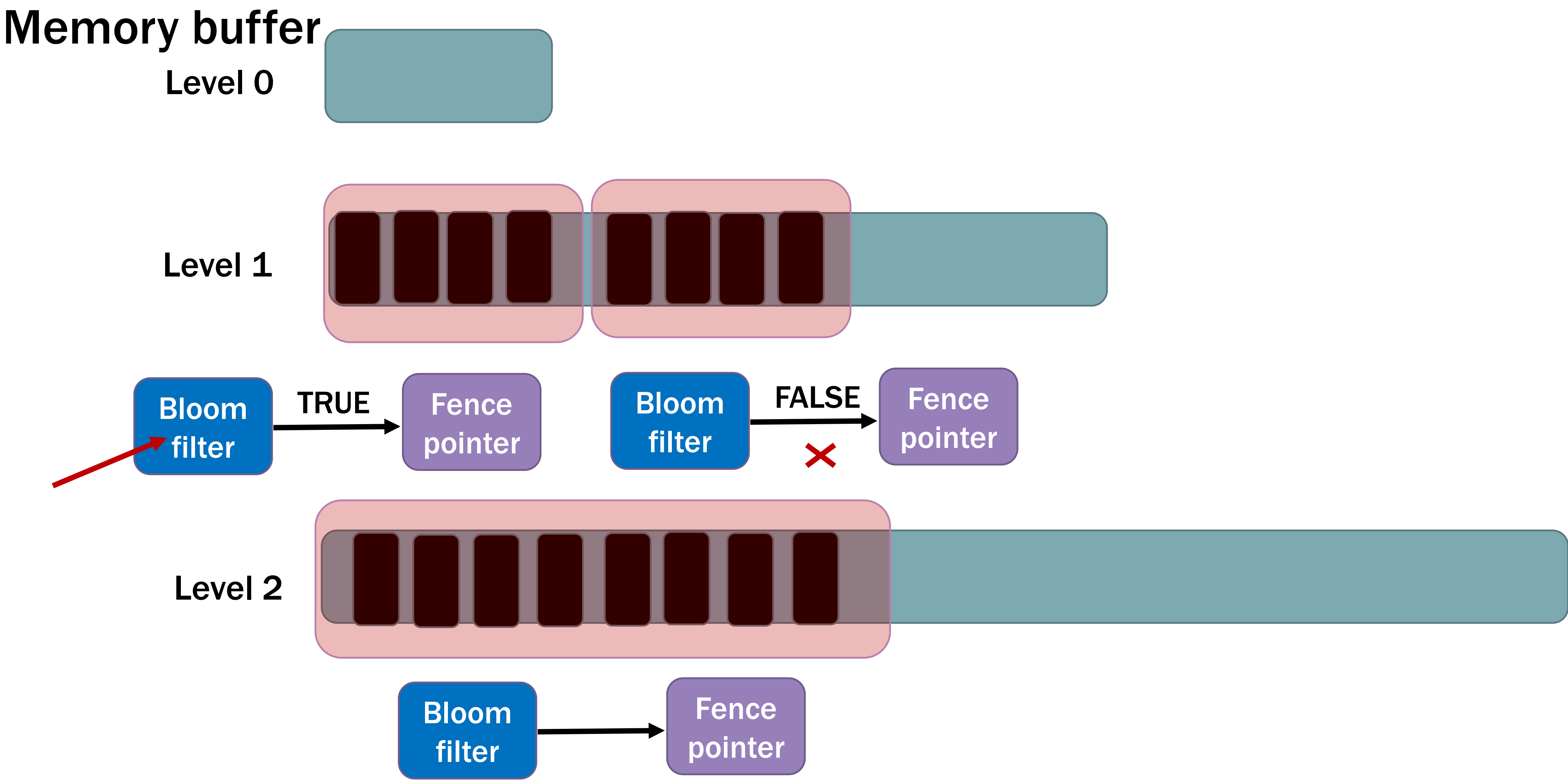
PERFORMING GET(K) IN TIERING LSM-TREE



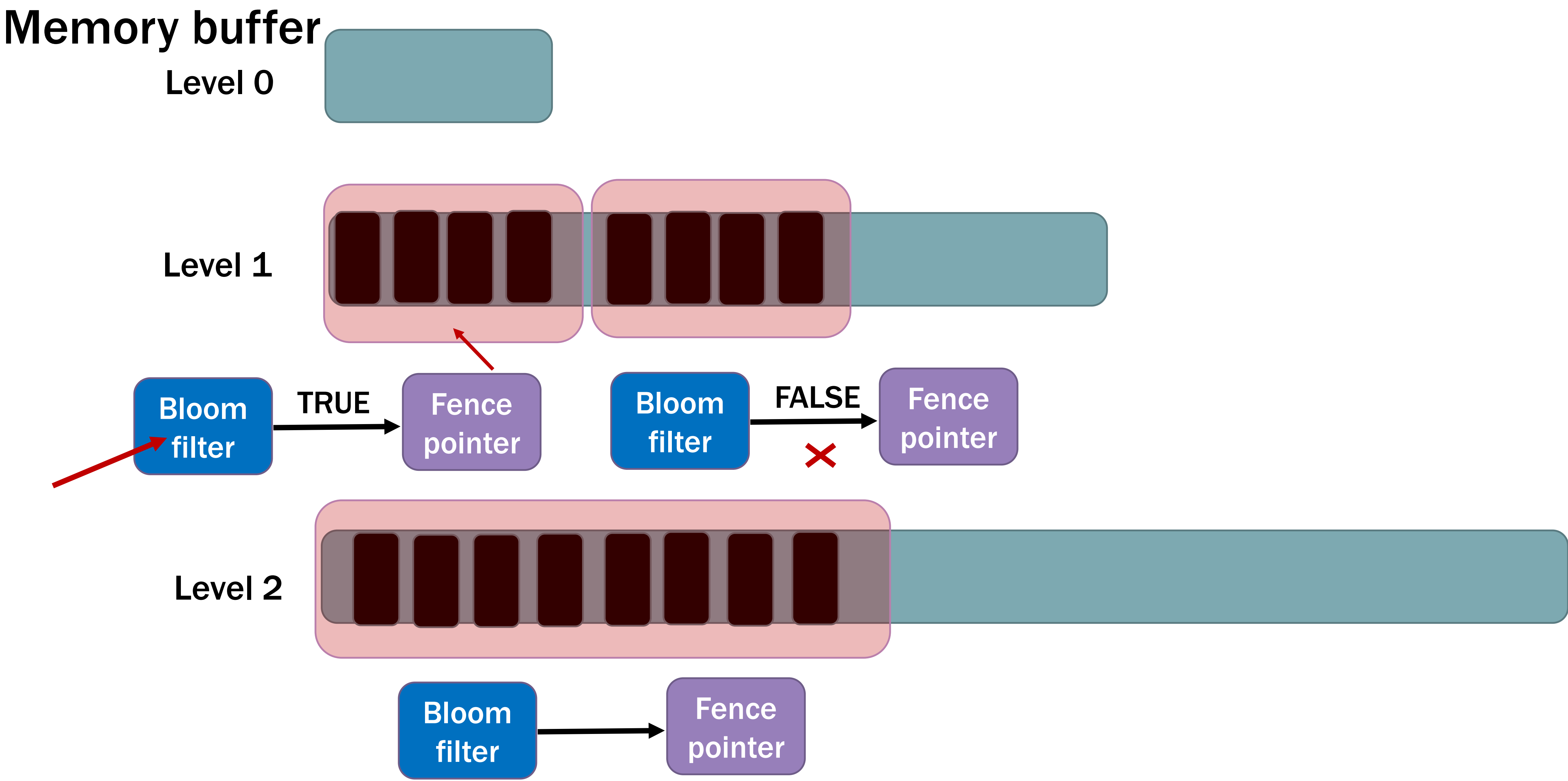
PERFORMING GET(K) IN TIERING LSM-TREE



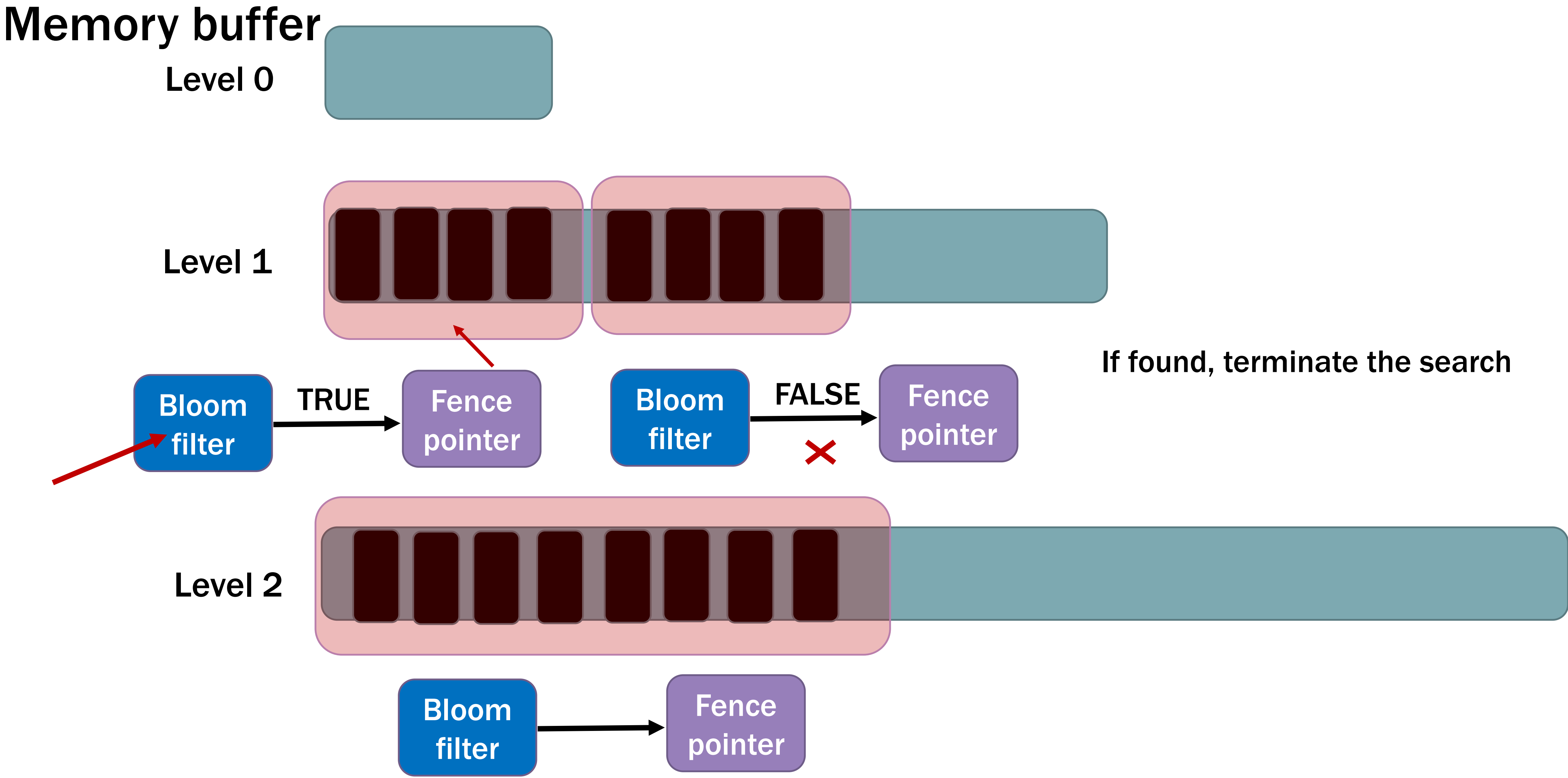
PERFORMING GET(K) IN TIERING LSM-TREE



PERFORMING GET(K) IN TIERING LSM-TREE



PERFORMING GET(K) IN TIERING LSM-TREE



If found, terminate the search

PERFORMING GET(K) IN TIERING LSM-TREE