SLM-DB: Single Level Merge Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, Young-ri Choi

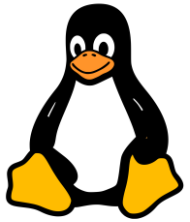
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

1398
성균관대학교
SUNG KYUN KWAN UNIVERSITY

Outline

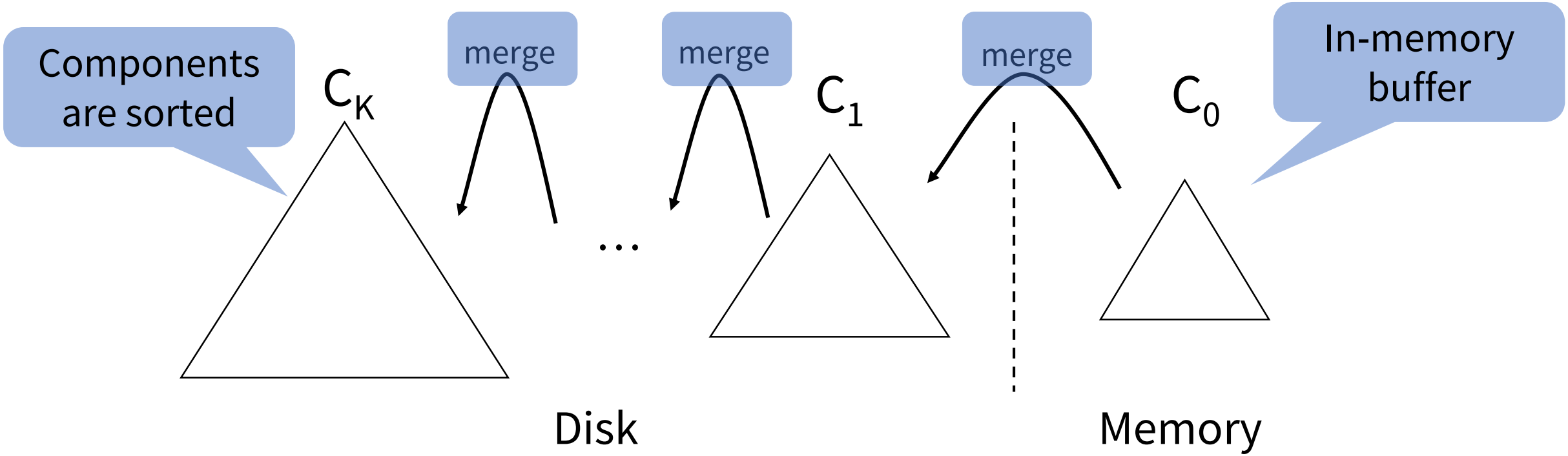
- Background
- Contributions
- Architecture
- Evaluation
- Conclusion

Key-Value Databases

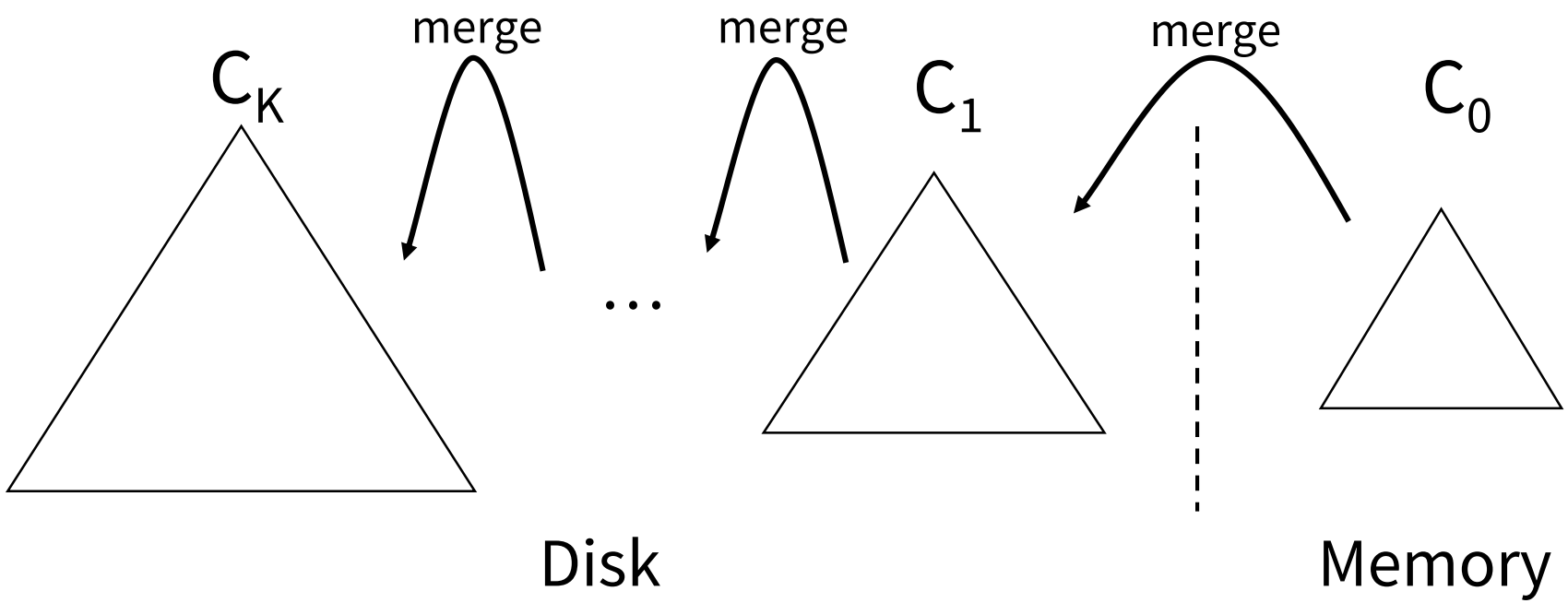
Key	Value
“100”	{[Green, Word, Gates]}
“html_doc”	<html><head>….. </body></html>
“linux_logo”	

Log-Structured Merge (LSM) Tree

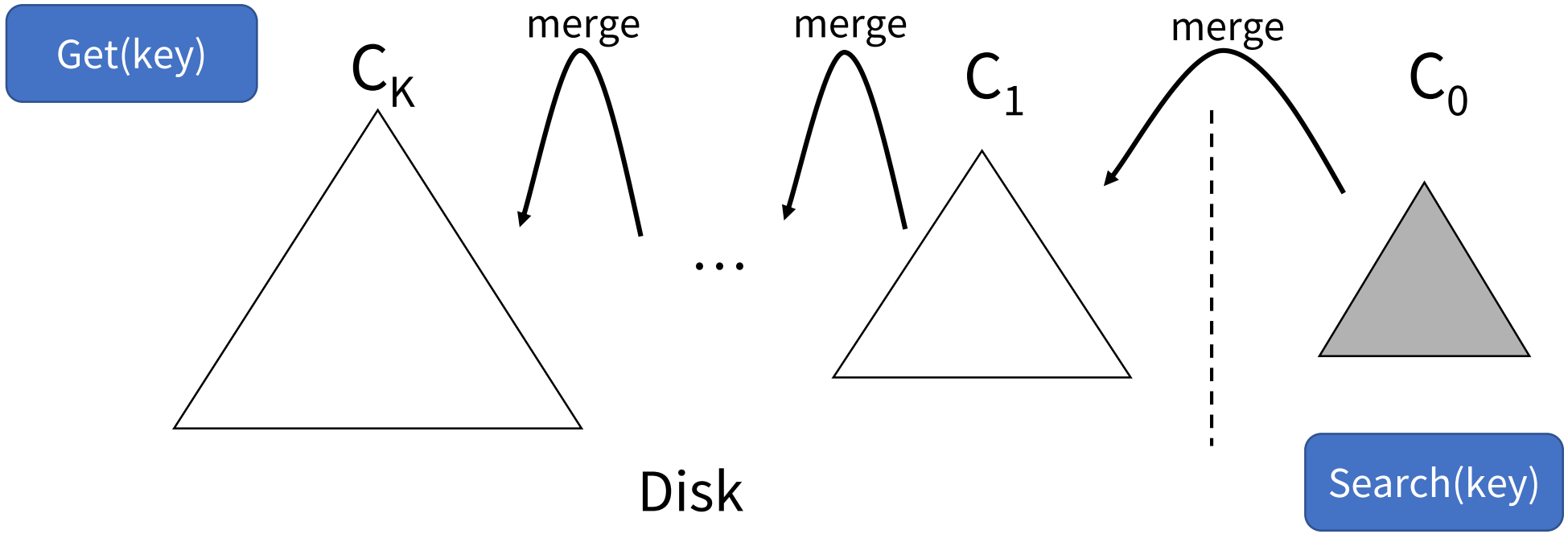
- Optimized for heavy write application usage
- Designed for slow hard drives



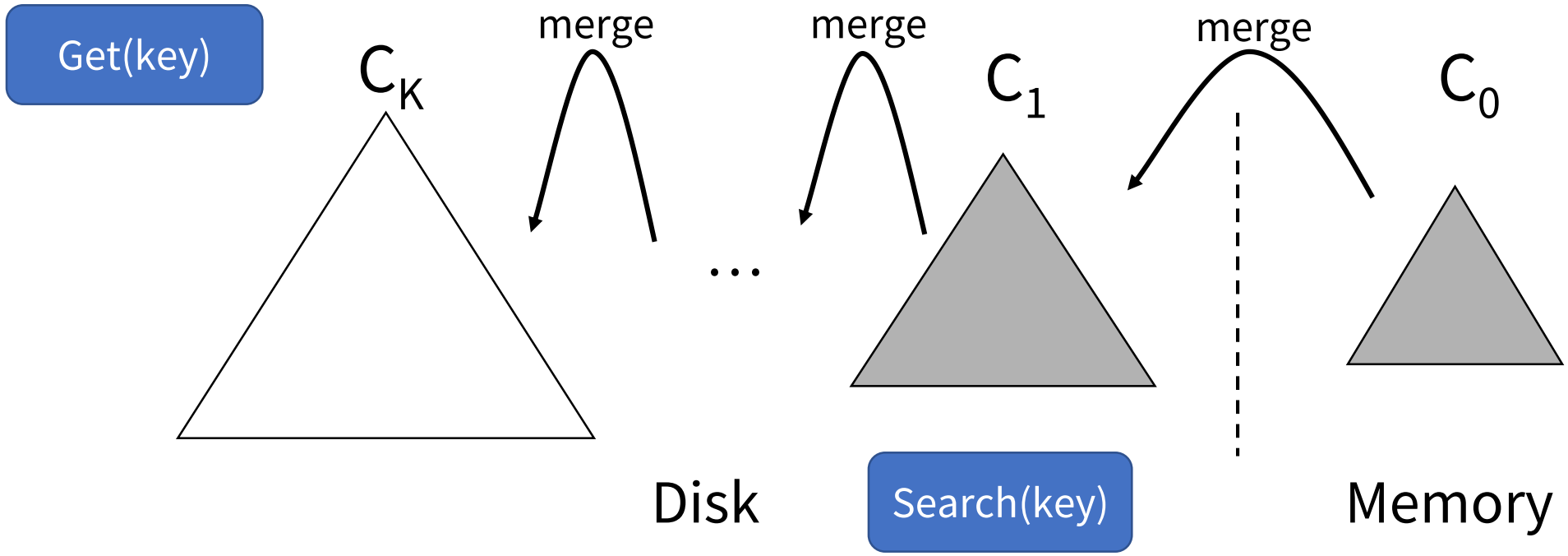
LSM-tree: disadvantages



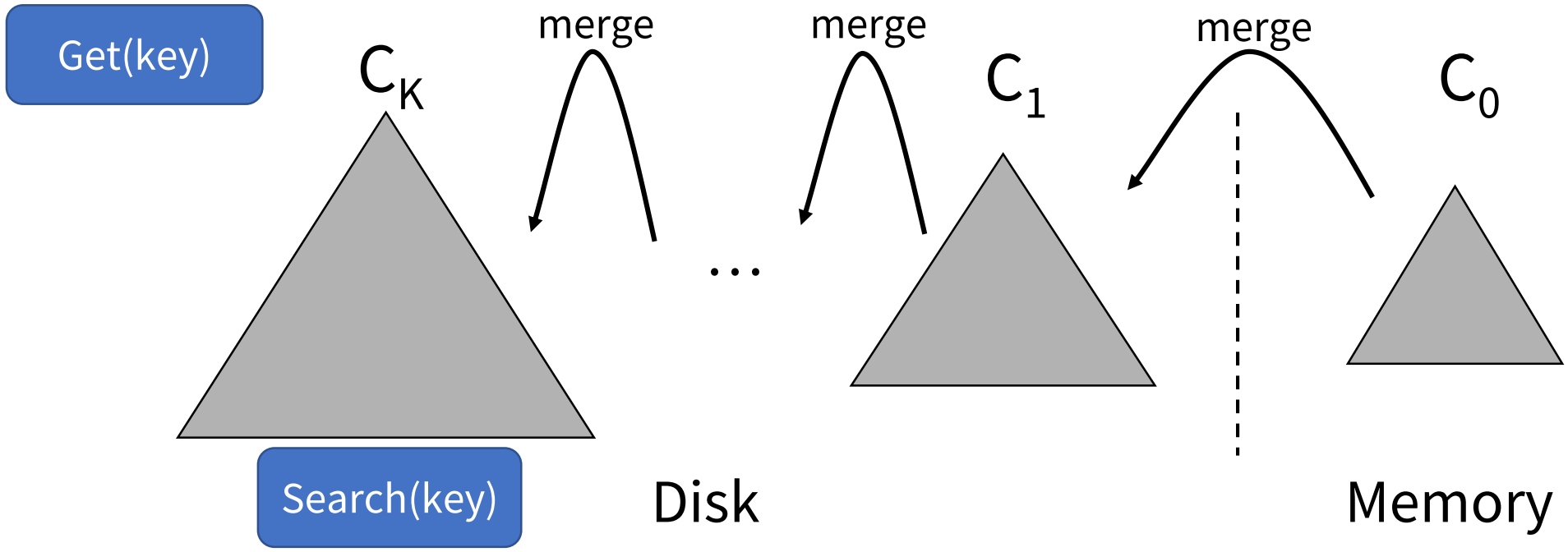
LSM-tree: disadvantages



LSM-tree: disadvantages

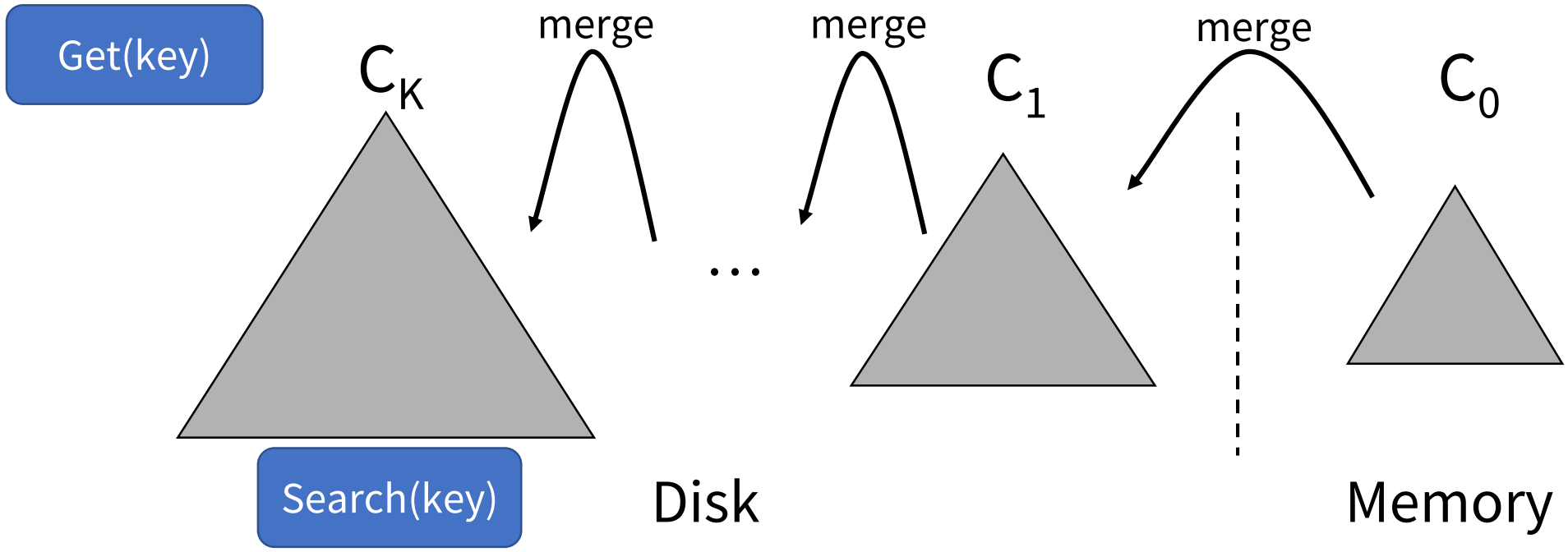


LSM-tree: disadvantages



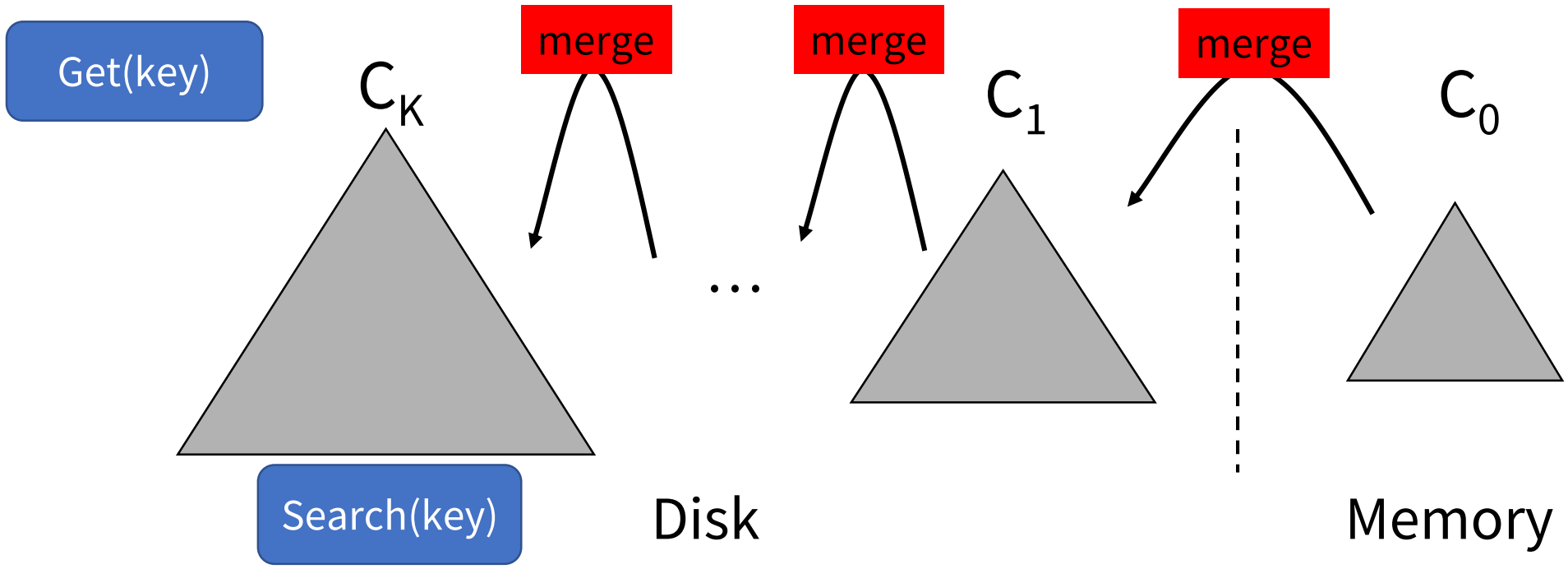
LSM-tree: disadvantages

- Large overhead to locate needed data



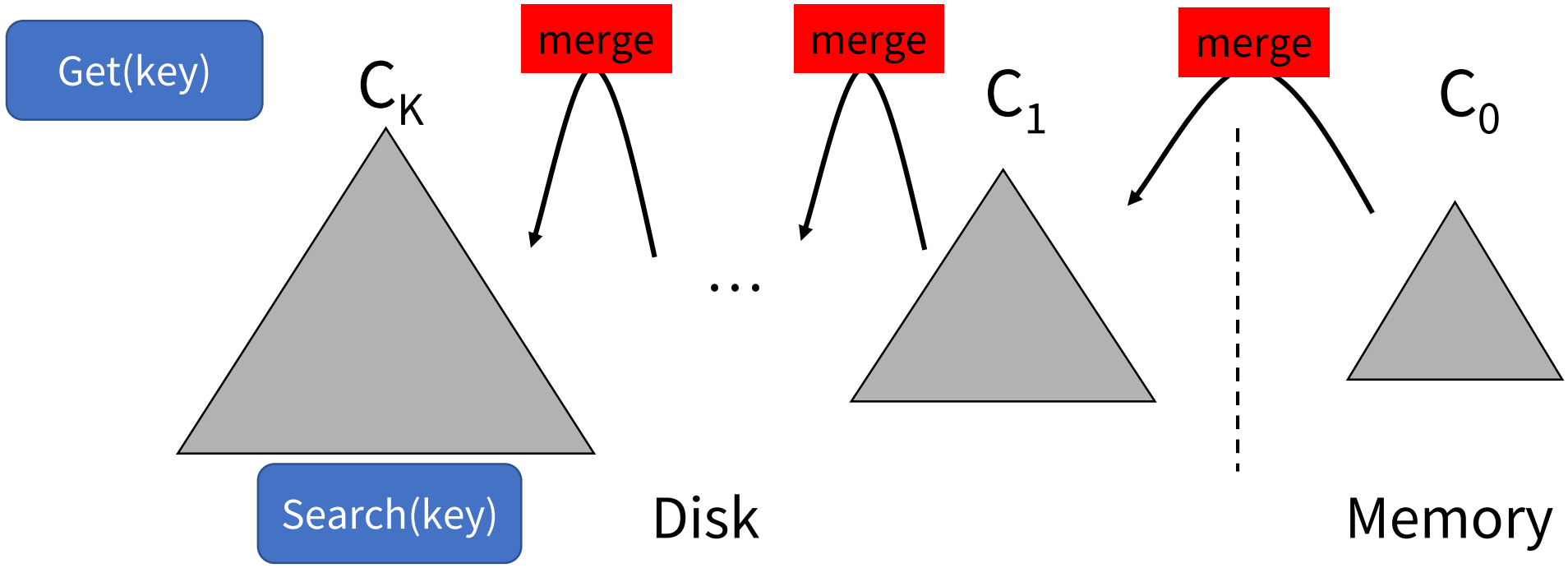
LSM-tree: disadvantages

- Large overhead to locate needed data

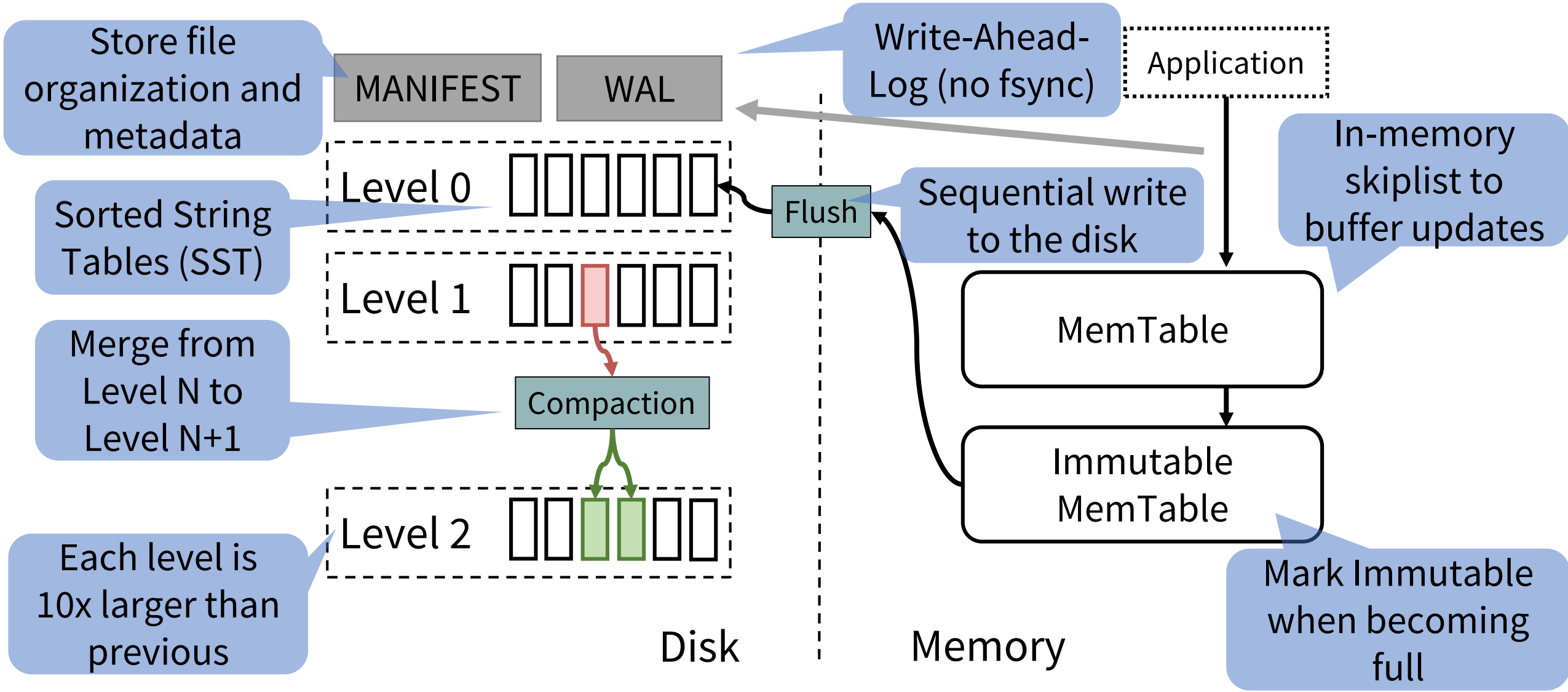


LSM-tree: disadvantages

- Large overhead to locate needed data
- High disk write amplification



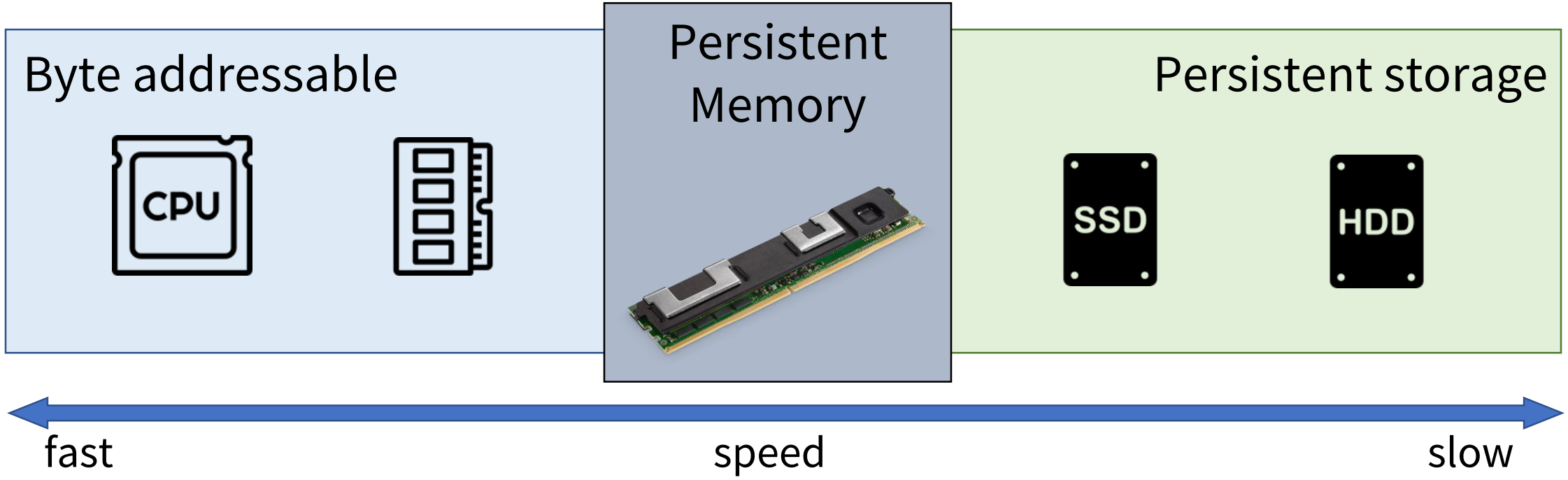
State-of-the-art LSM-tree: LevelDB



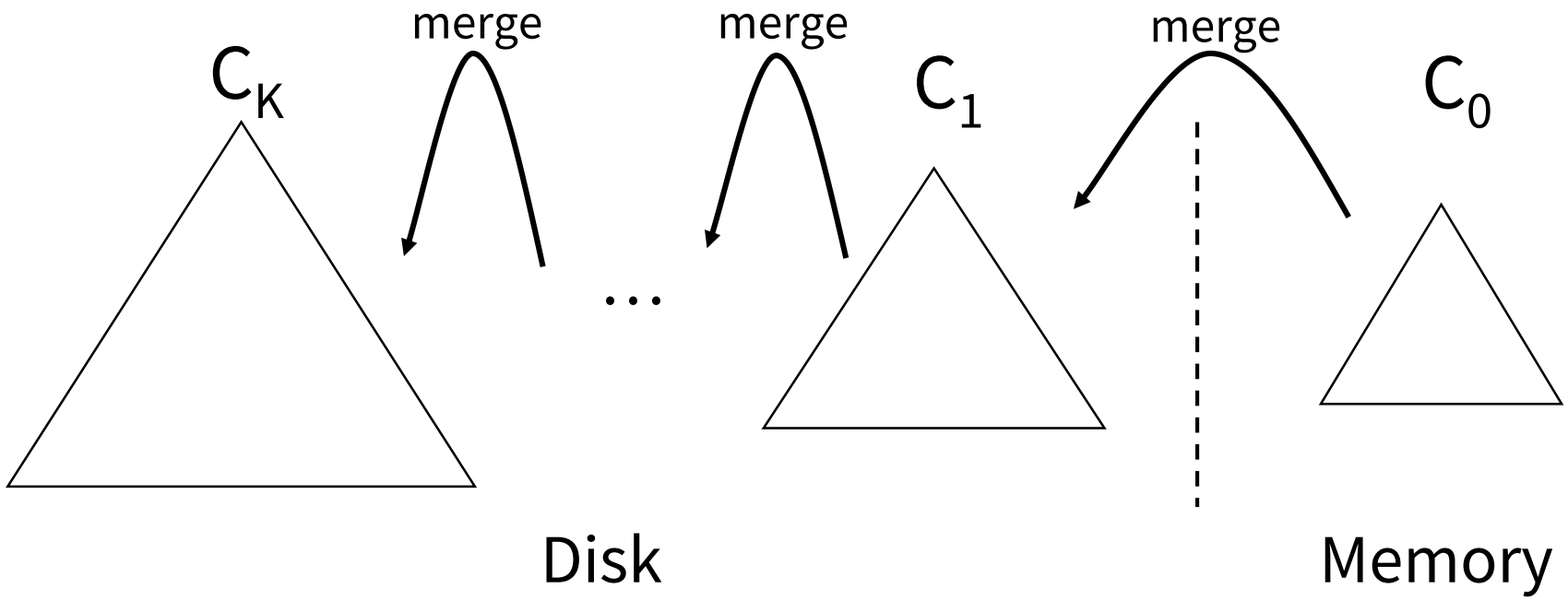
LSM-tree optimizations

- Improve parallelism:
  - RocksDB (Facebook)
  - HyperLevelDB
- Reduce write amplification:
  - PebblesDB (SOSP ‘17)
- Optimize for hardware(SSD):
  - VT-tree (FAST ‘13)
  - WiscKey (FAST ‘16)

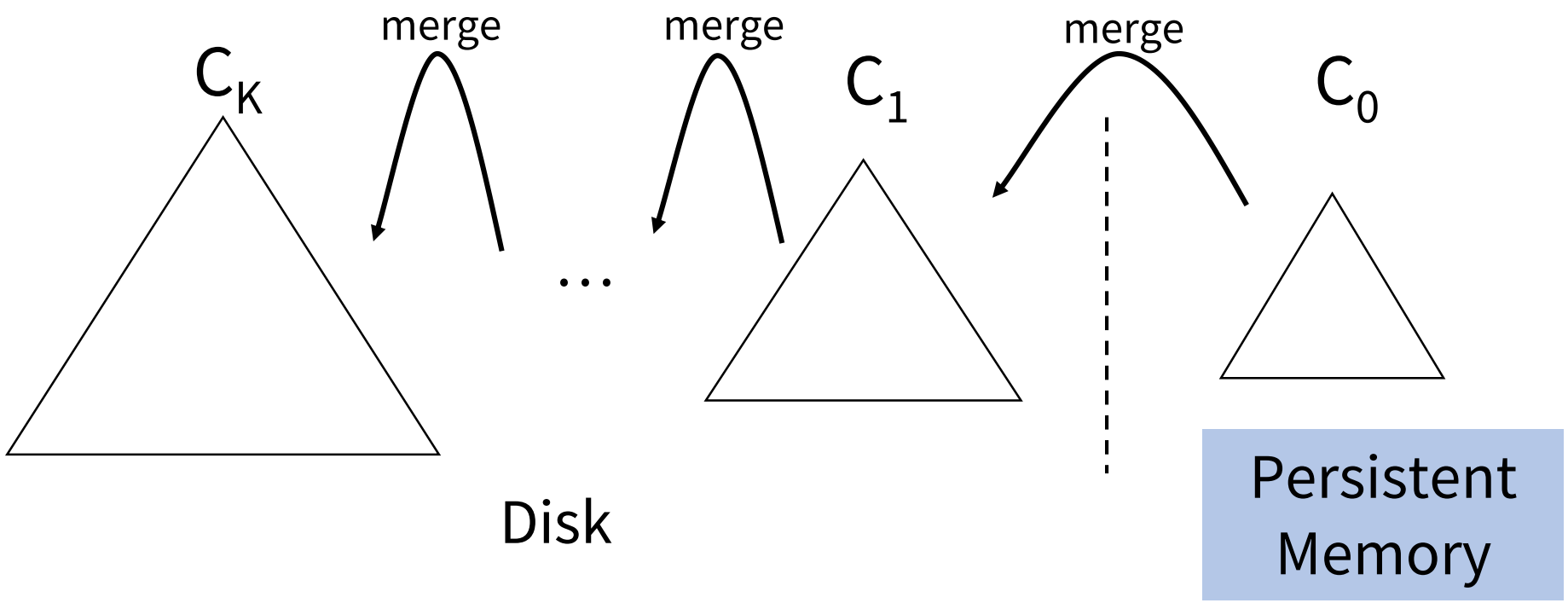
New era


Byte addressable
CPU
Persistent Memory
Persistent storage
SSD
HDD
fast
speed
slow

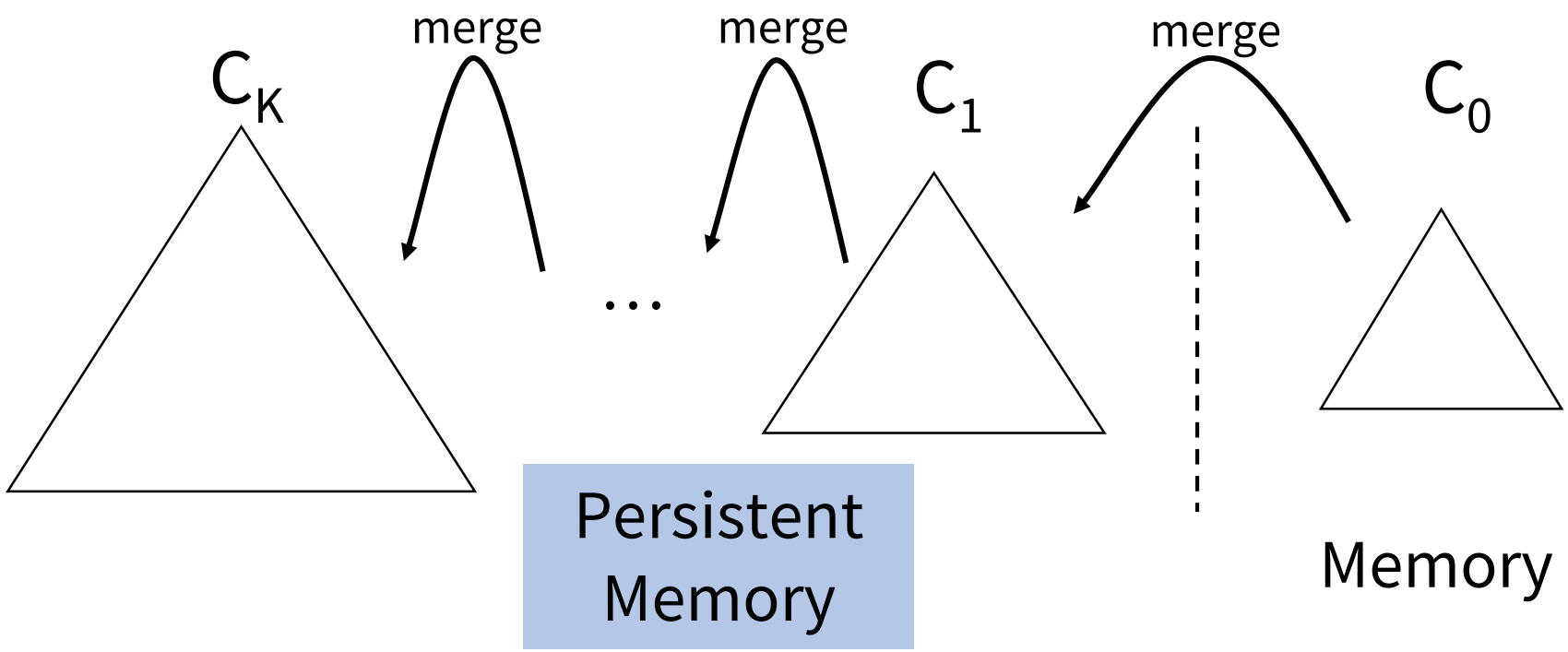
Simple approach



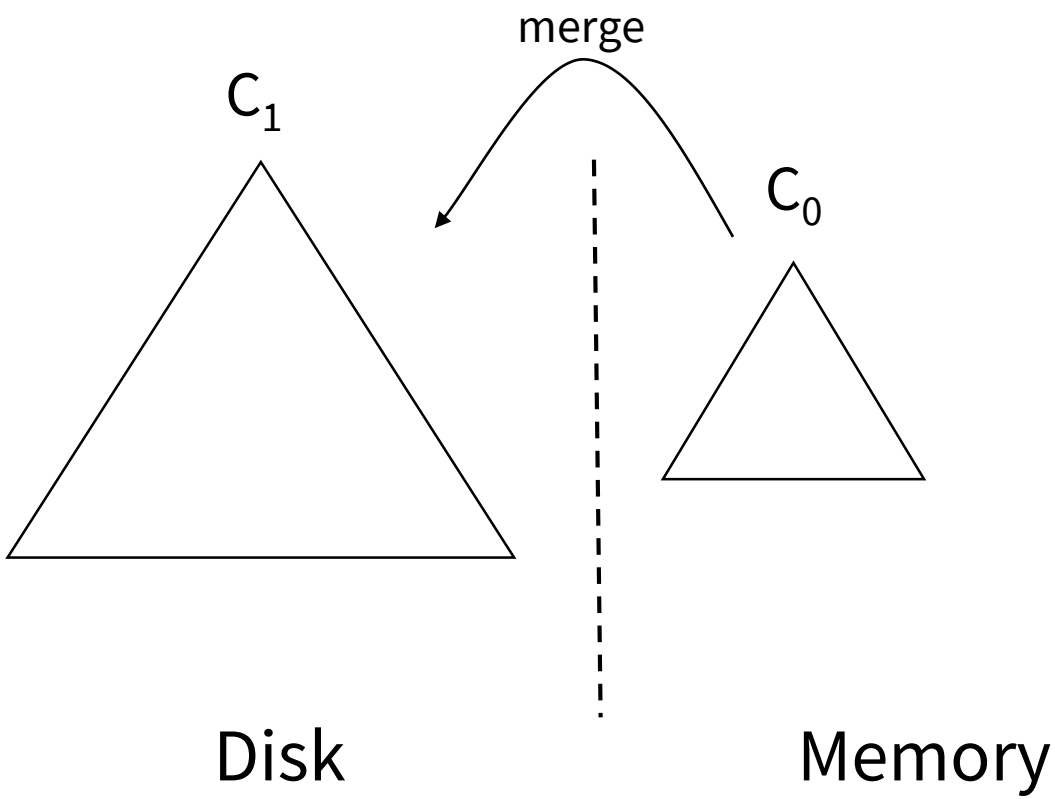
Simple approach



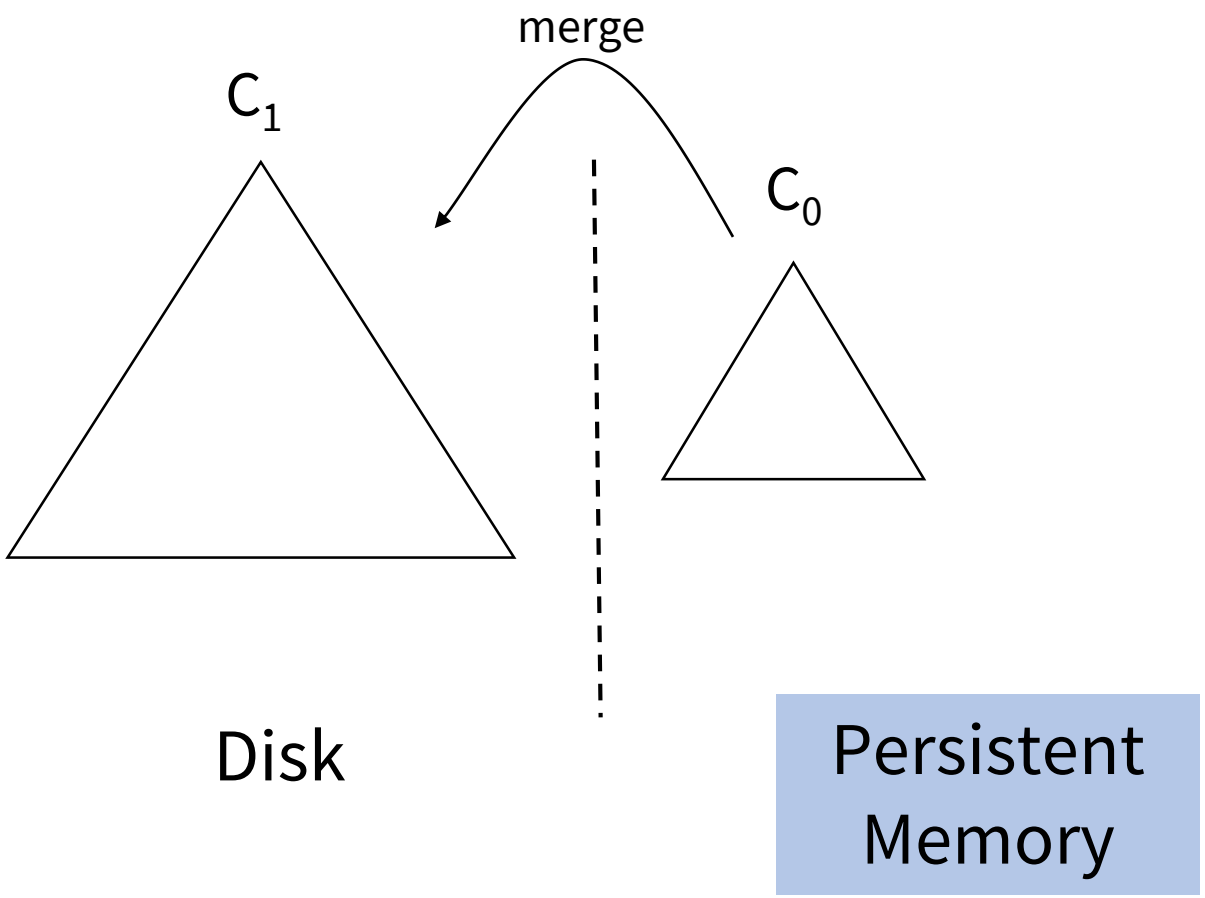
Simple approach



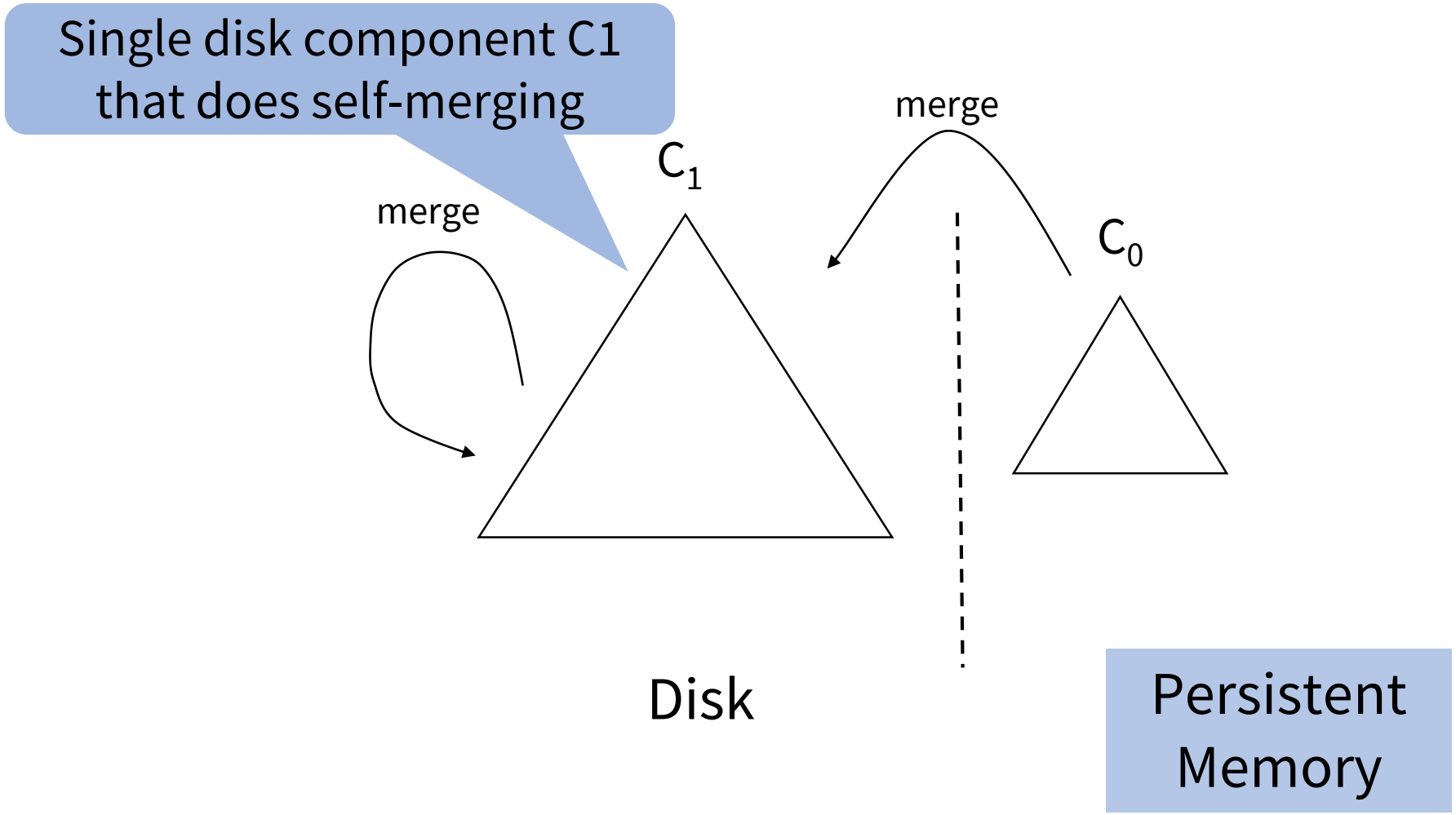
Our approach



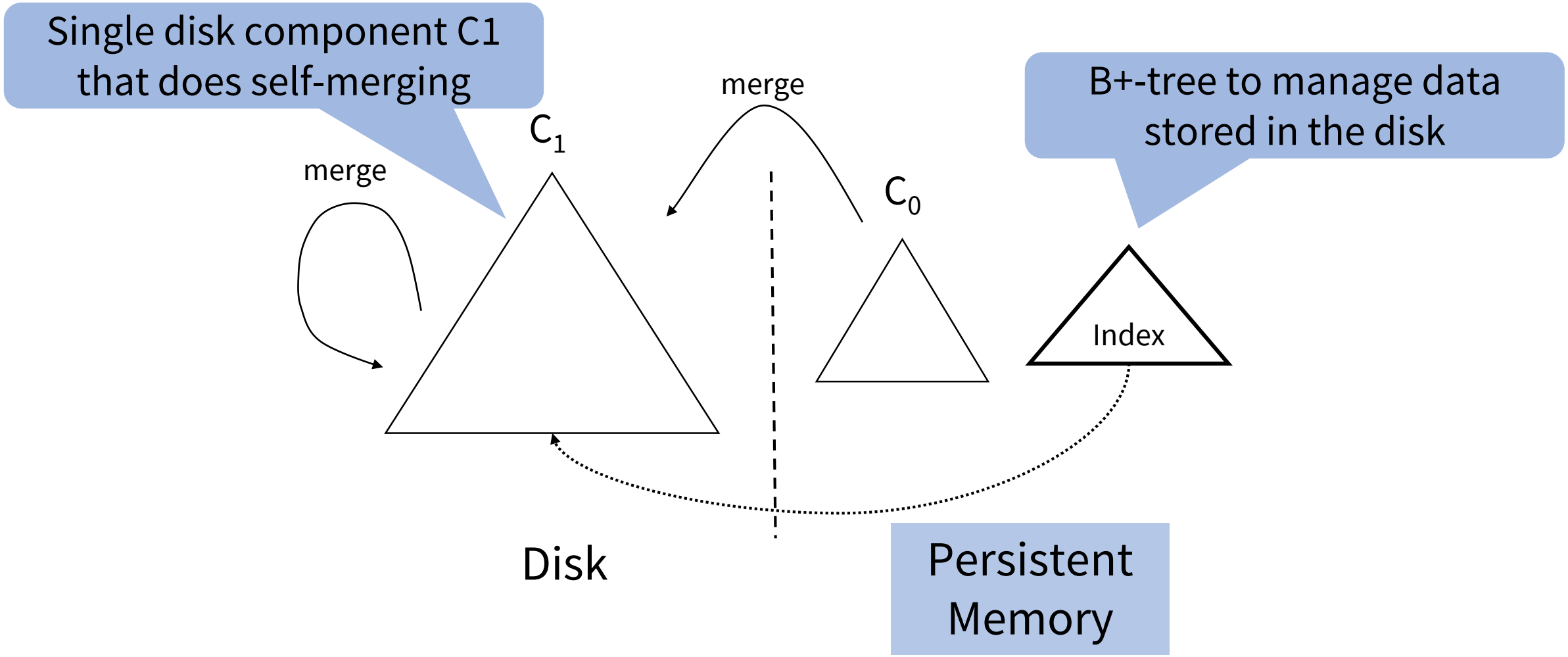
Our approach



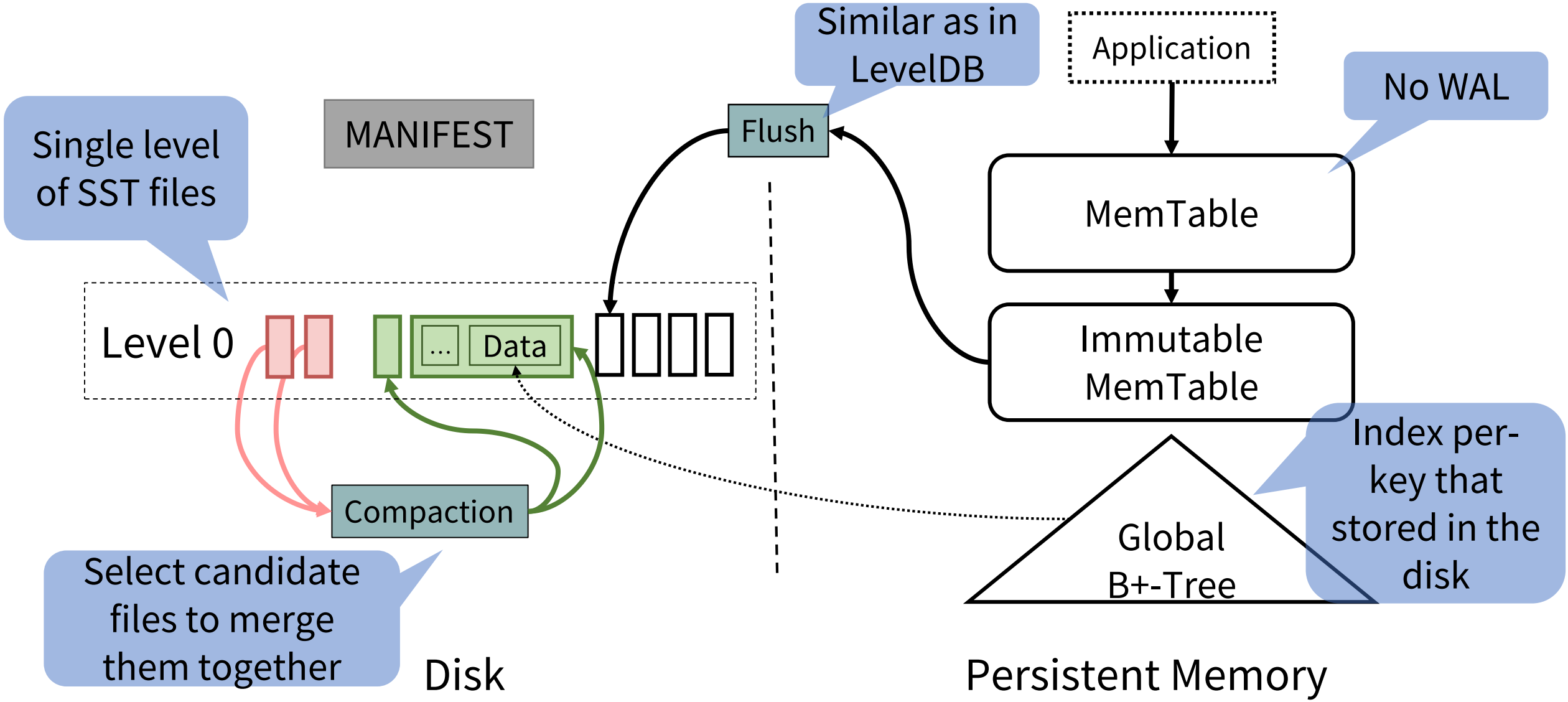
Our approach



Our approach



Single-Level Merge DB (SLM-DB)



Contributions

Persistent MemTable

No Write-Ahead Logging (WAL)

Stronger consistency compared to LevelDB

Persistent B+-tree Index

Per-key index for fast search

No multi-leveled merge structure

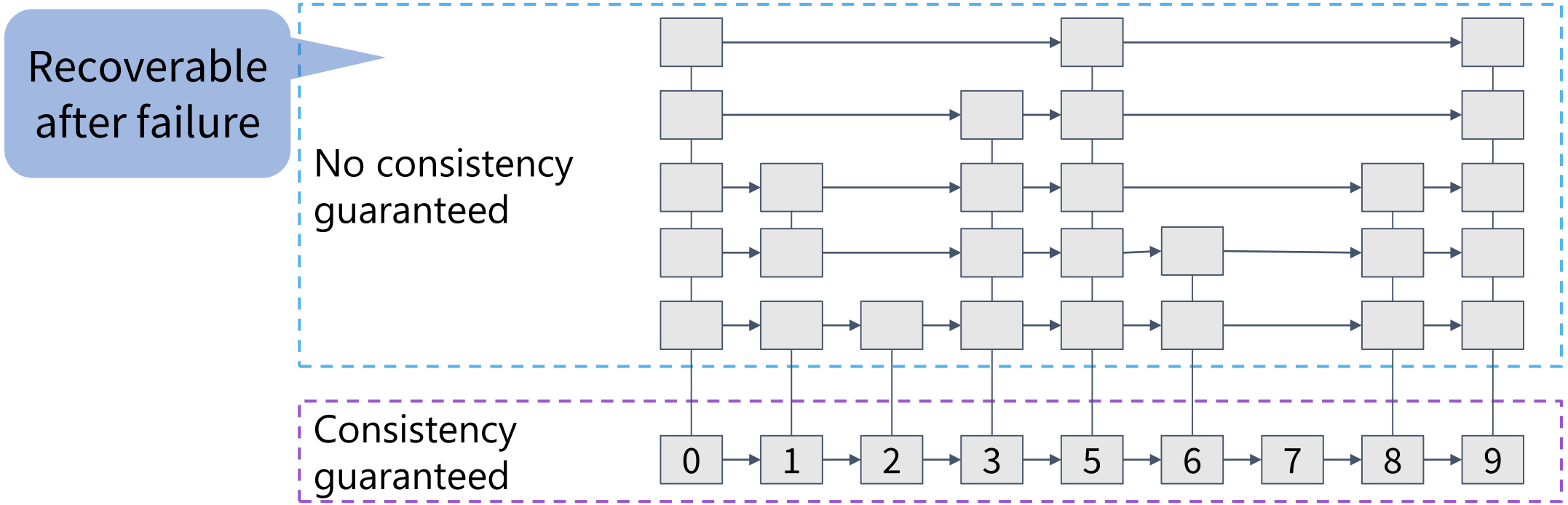
Selective Compaction

Live-key ratio of a Sorted-String Table

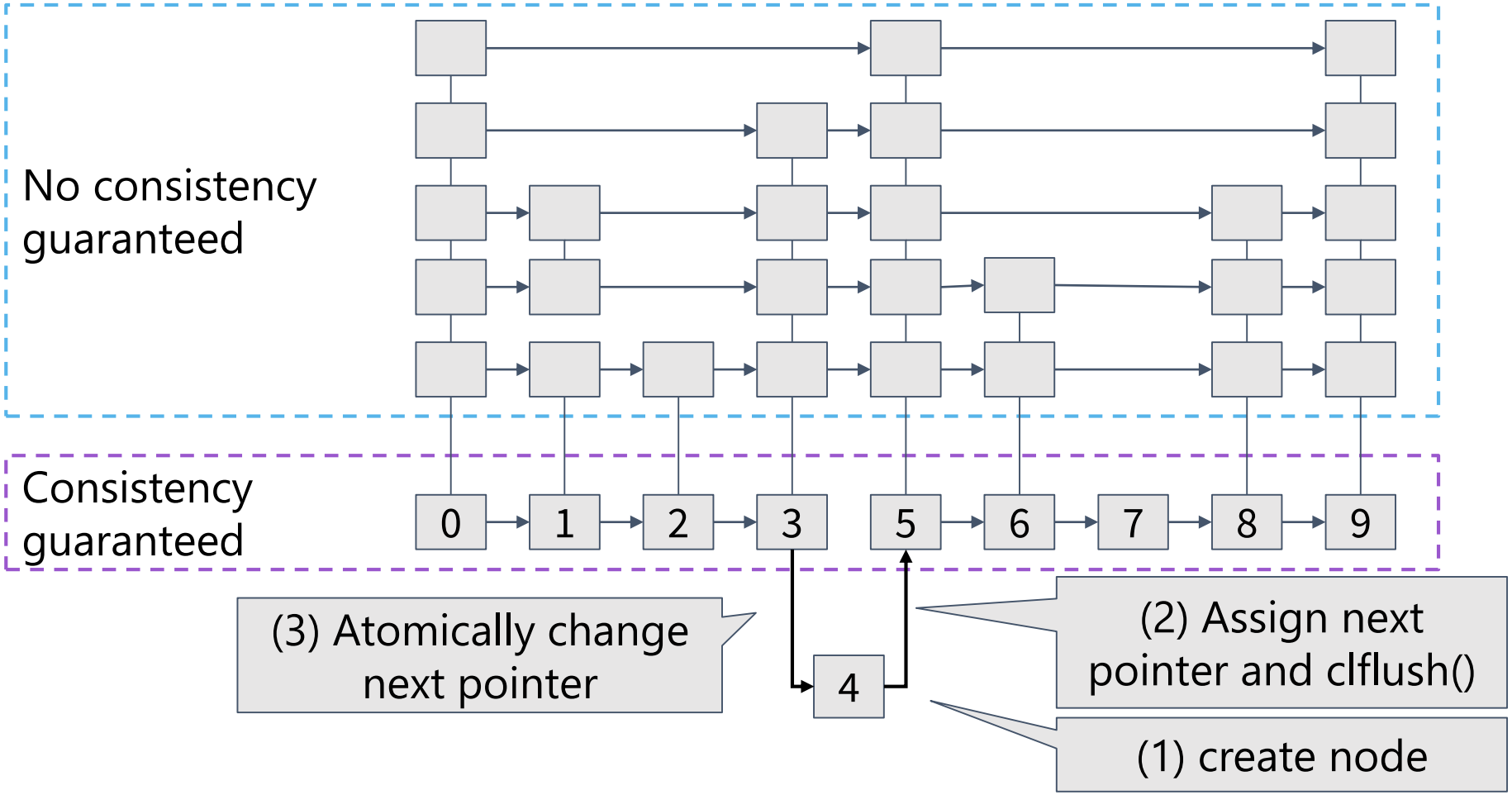
Leaf node scan in the B+-tree

Degree of sequentiality per range query

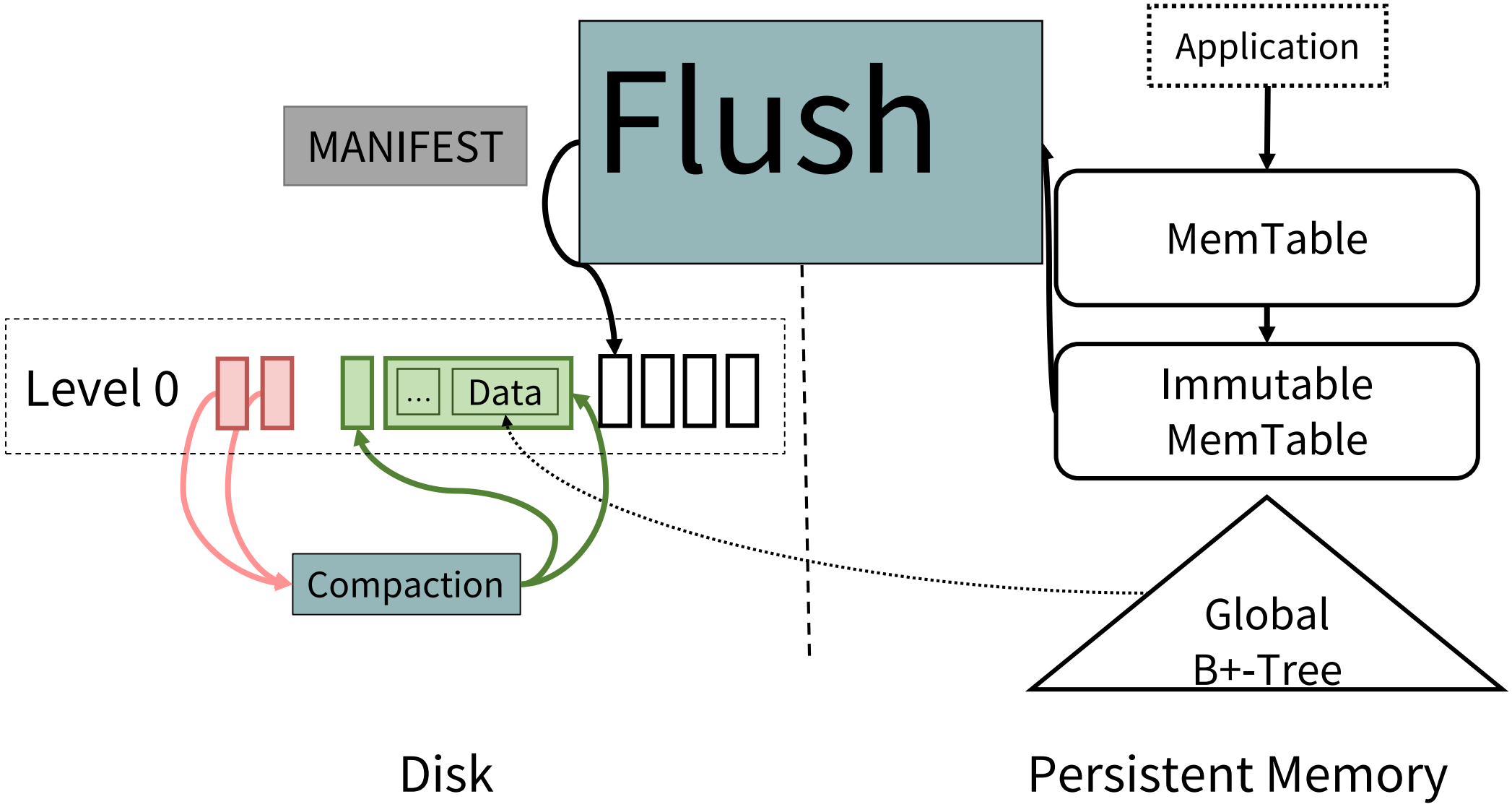
Persistent MemTable



Insert into Persistent MemTable

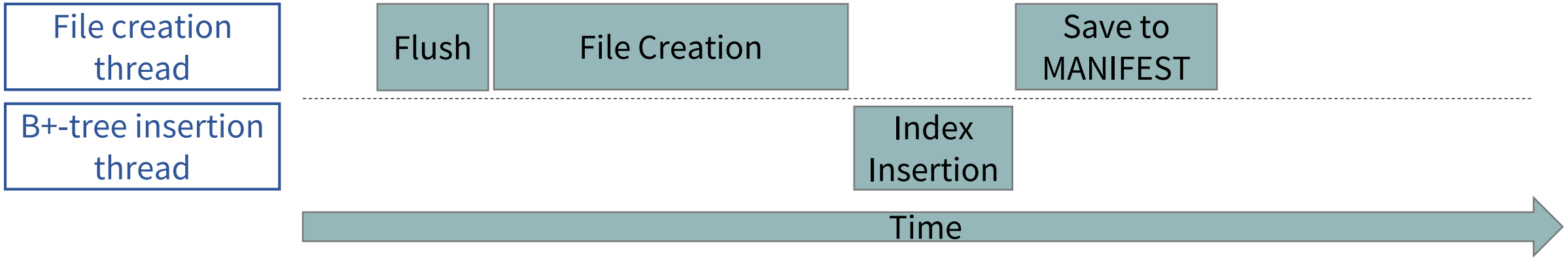


Single-Level Merge DB

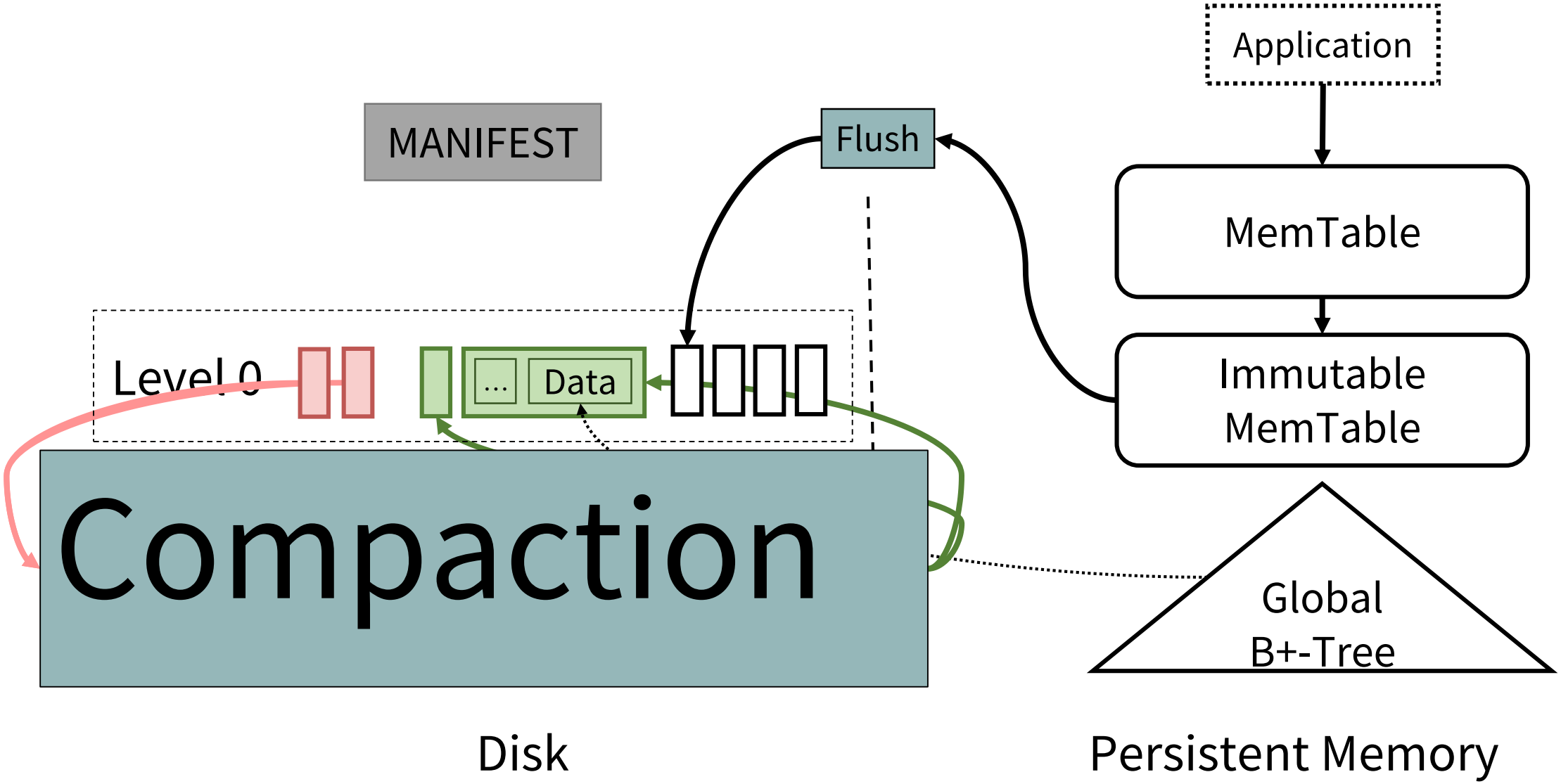


Flush

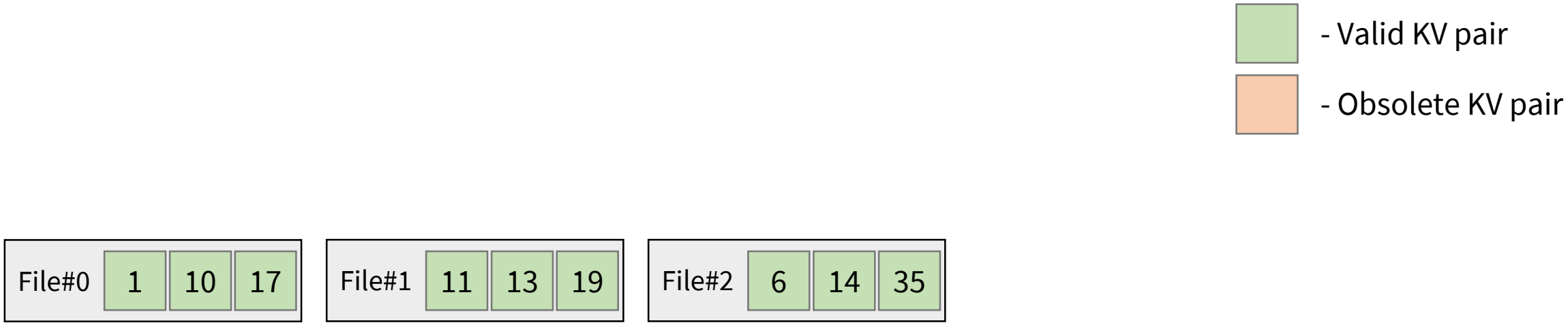
- Key-Index insertion into B+-tree happens during Immutable Memtable Flush to disk
- FAST-FAIR B+-tree (Hwang et al., FAST ’18)



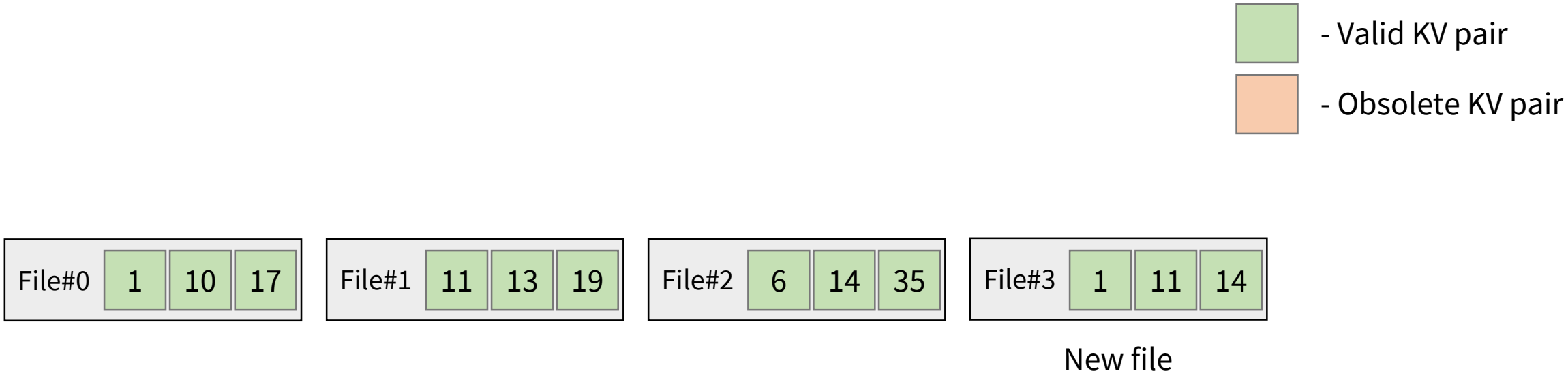
Single-Level Merge DB



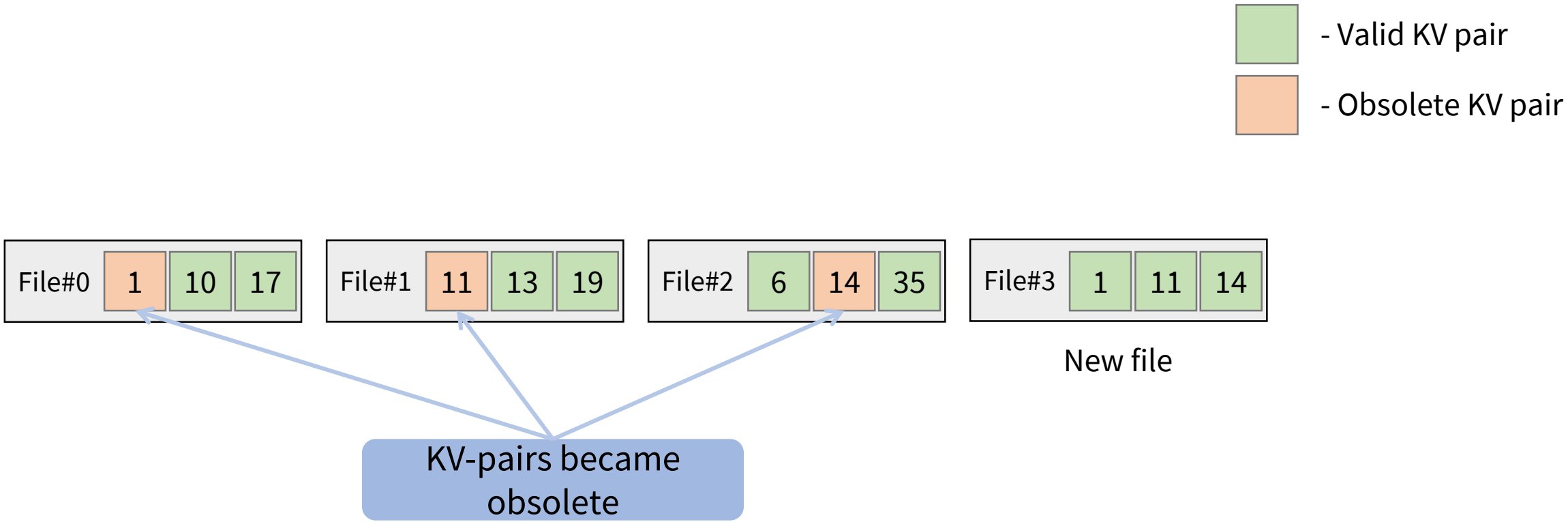
Why we need **Compaction**?



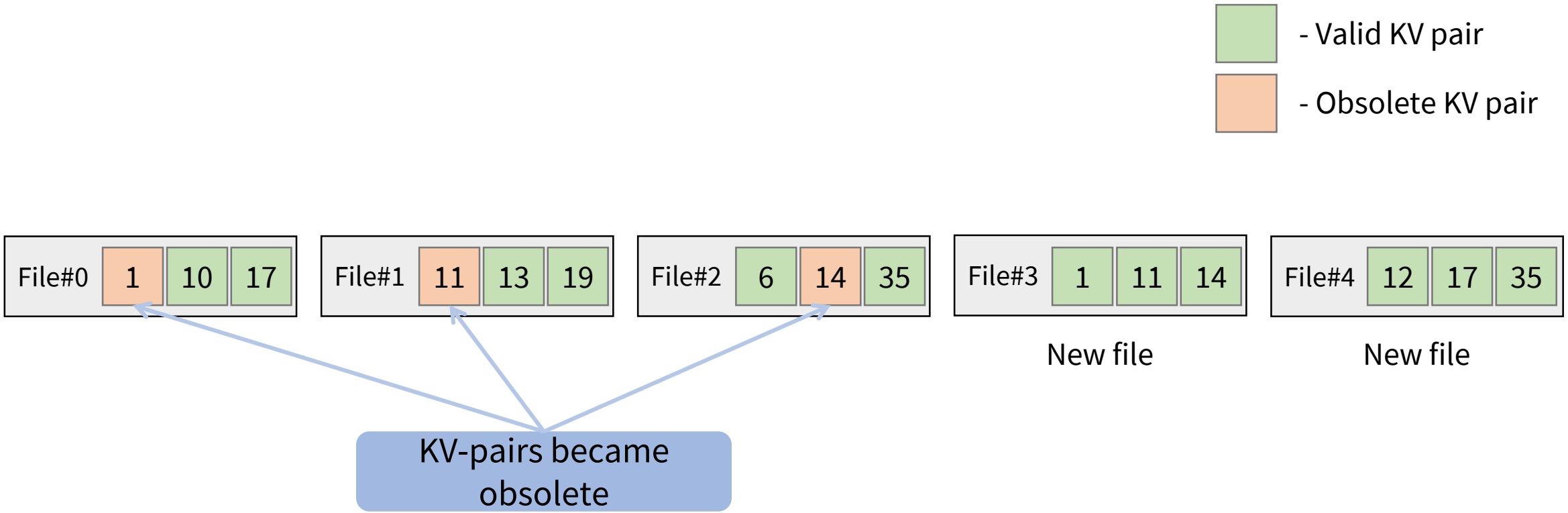
Why we need **Compaction**?



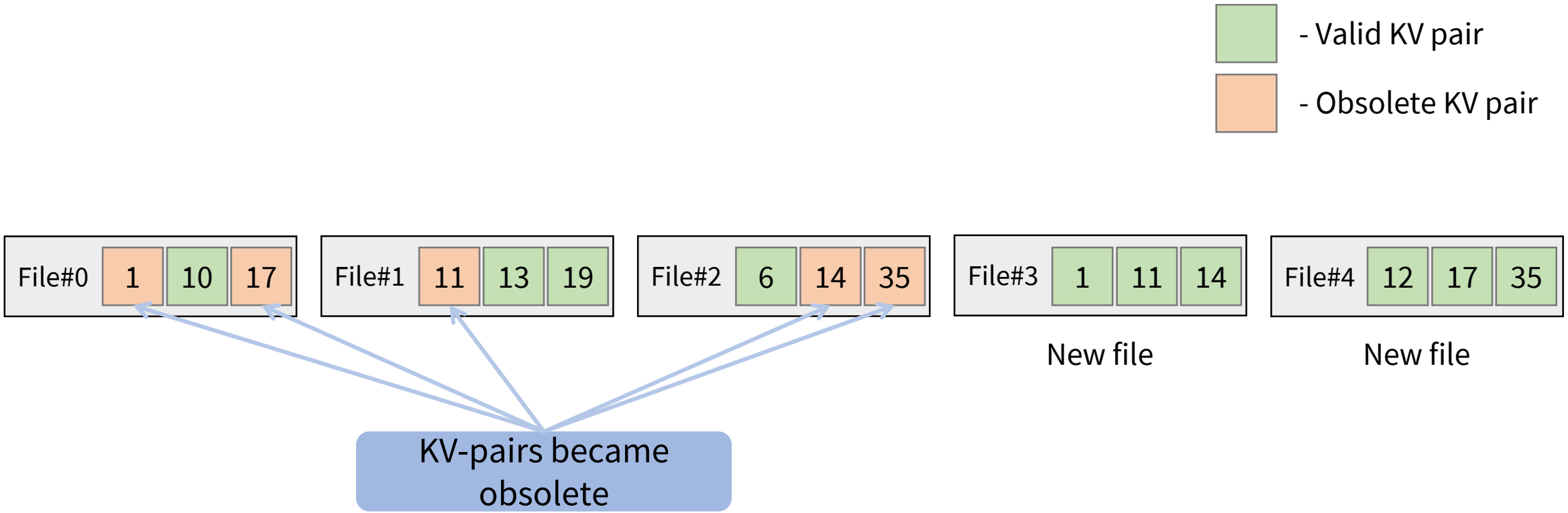
Why we need **Compaction**?



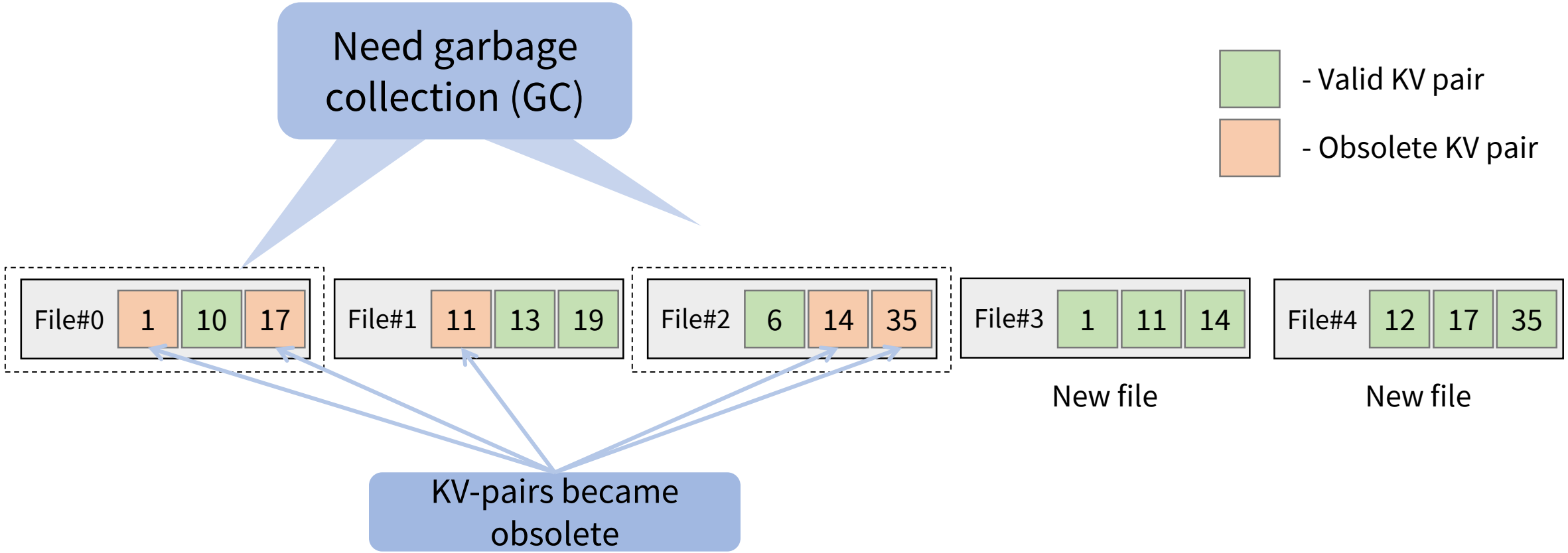
Why we need **Compaction**?



Why we need **Compaction**?

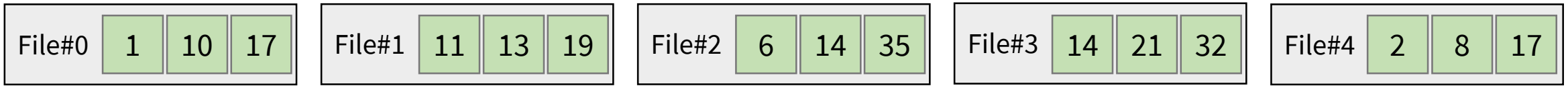


Why we need **Compaction**?



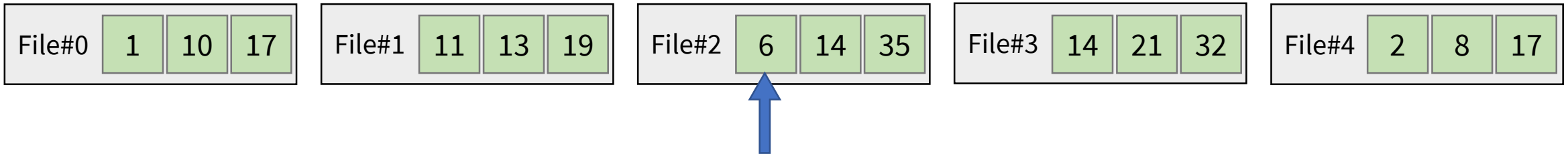
Why else?

RangeQuery(5, 12)



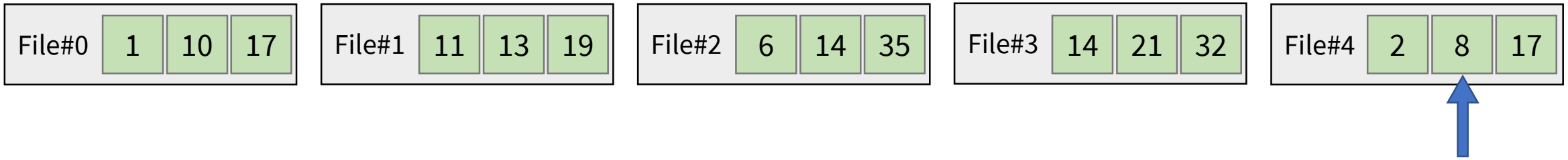
Why else?

RangeQuery(5, 12)



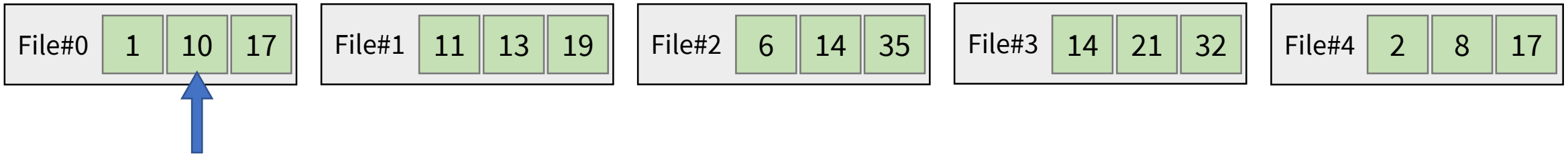
Why else?

RangeQuery(5, 12)



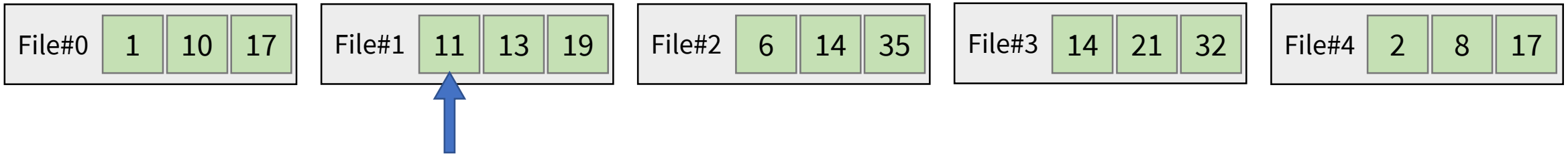
Why else?

RangeQuery(5, 12)



Why else?

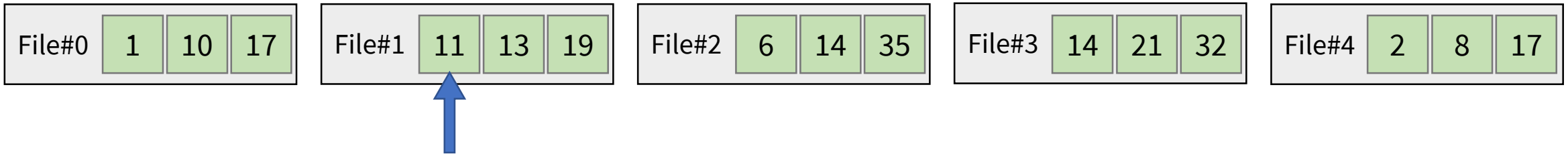
RangeQuery(5, 12)



Why else?

RangeQuery(5, 12)

Need to improve sequentiality

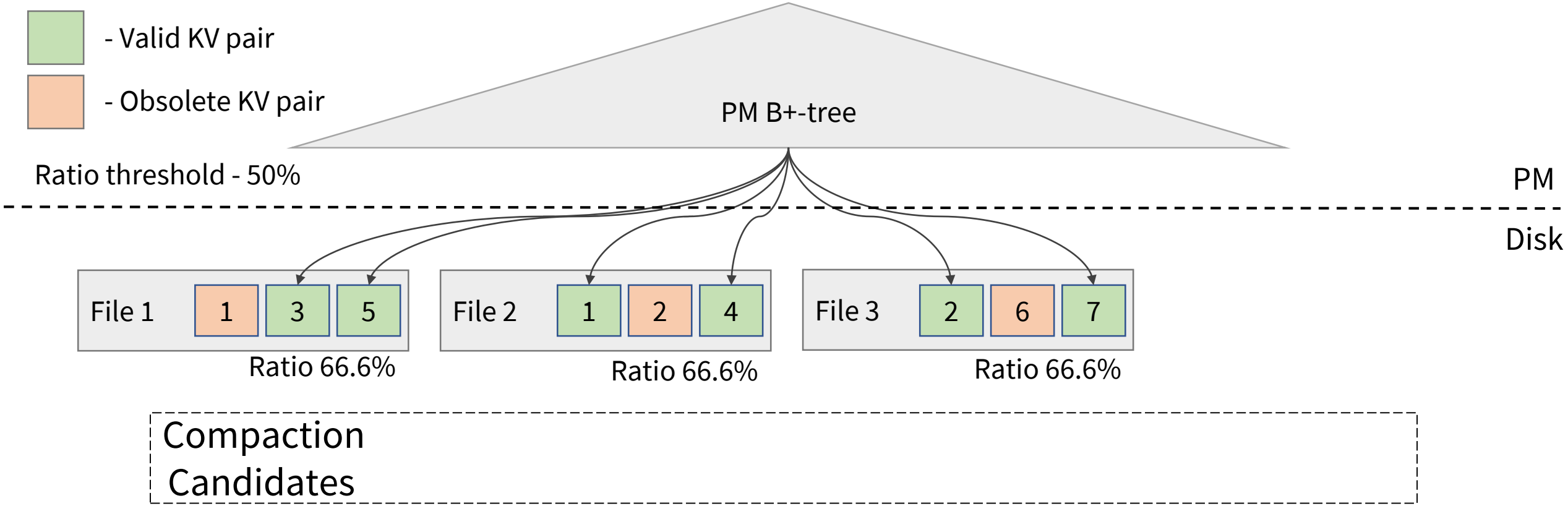


Selective compaction

- Selectively pick SSTable files
- Make those files as compaction candidates
- Merge together most overlapping compaction candidates
- Selection schemes for compaction candidates:
  - Live-key ratio selection of an SSTable (for GC)
  - Leaf node scans in the B+-tree (for sequentiality) [see paper]
  - Degree of sequentiality per range query (for sequentiality) [see paper]

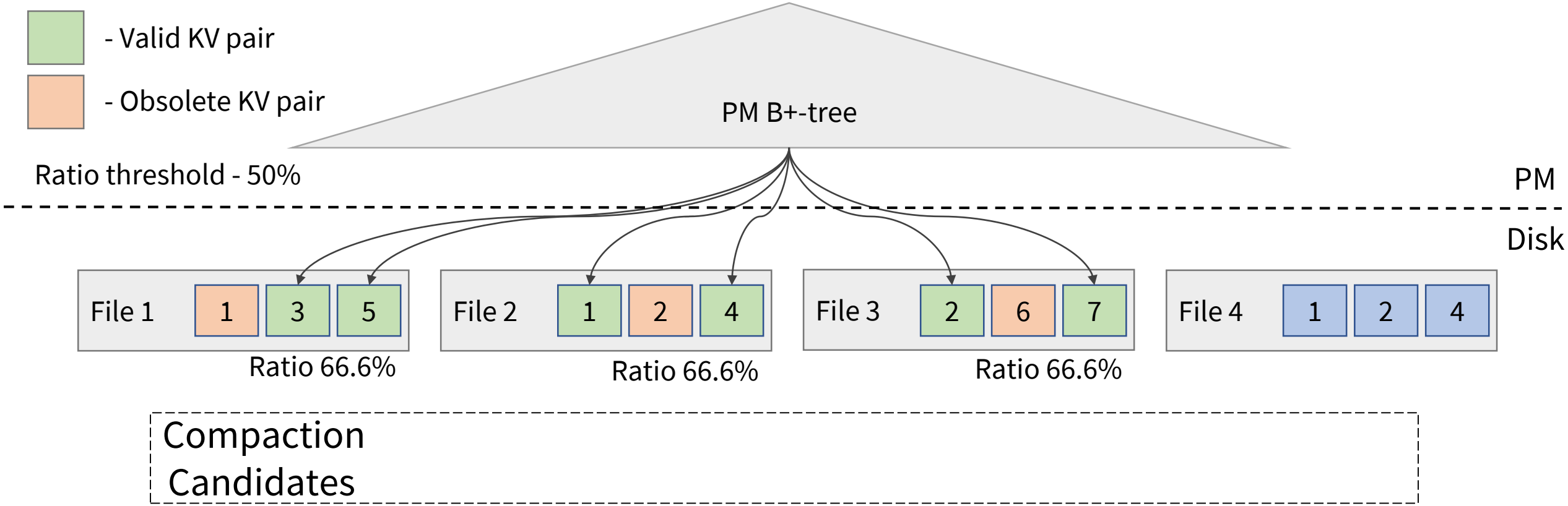
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



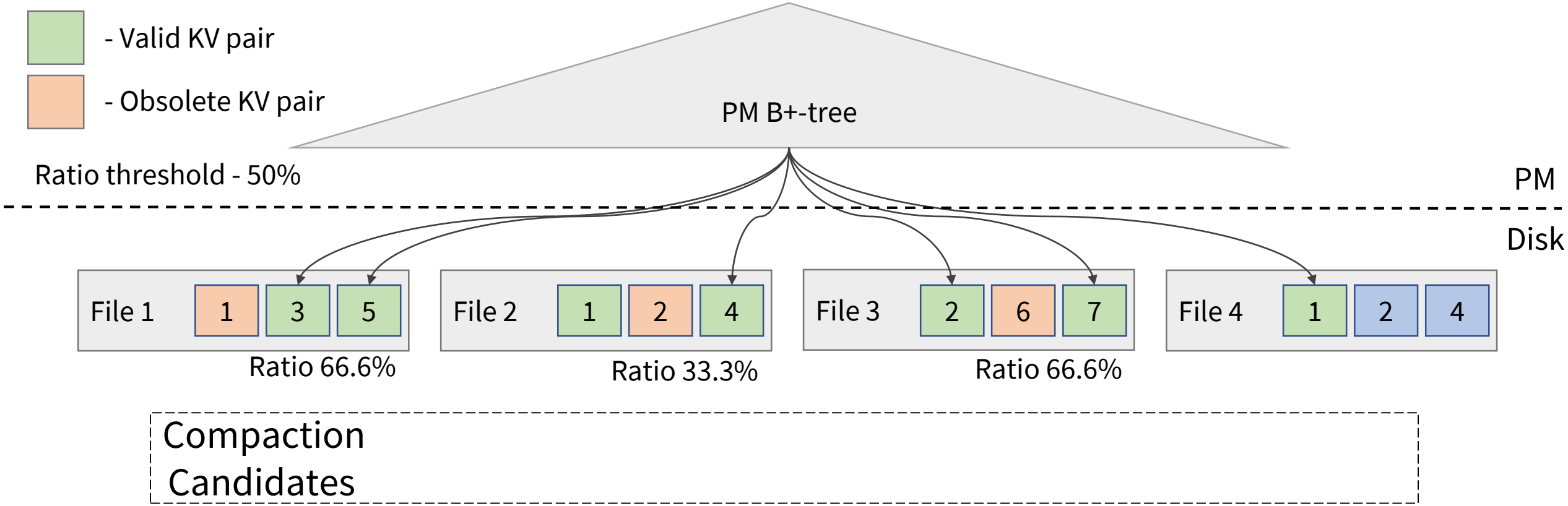
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



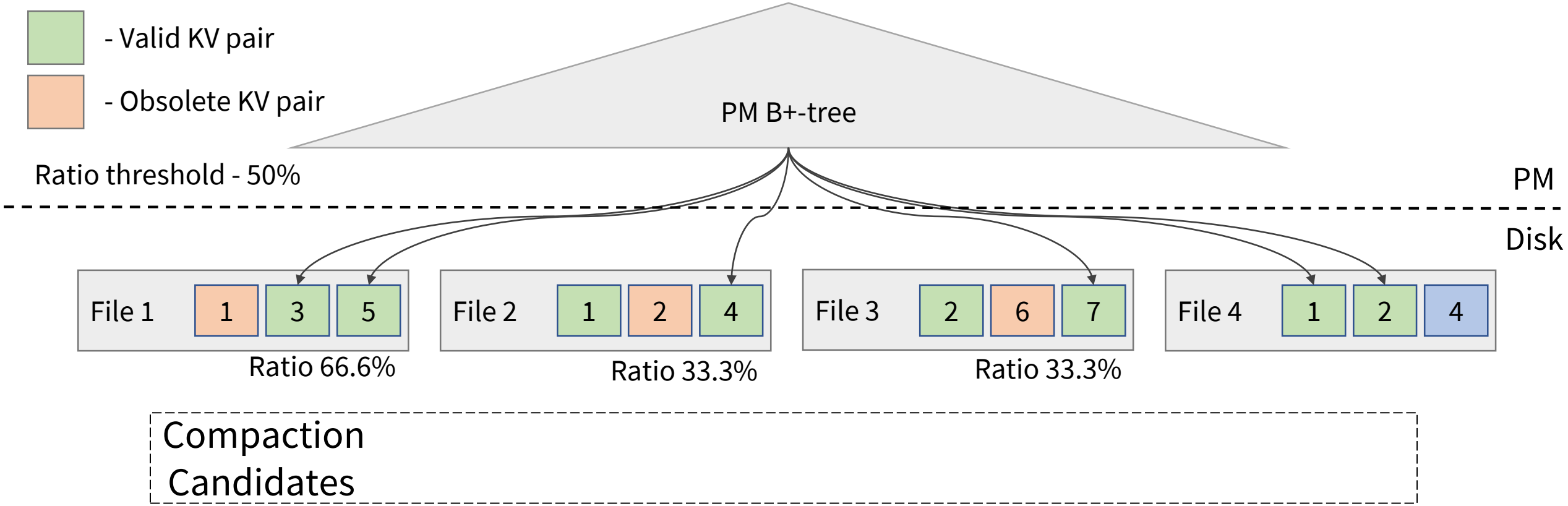
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



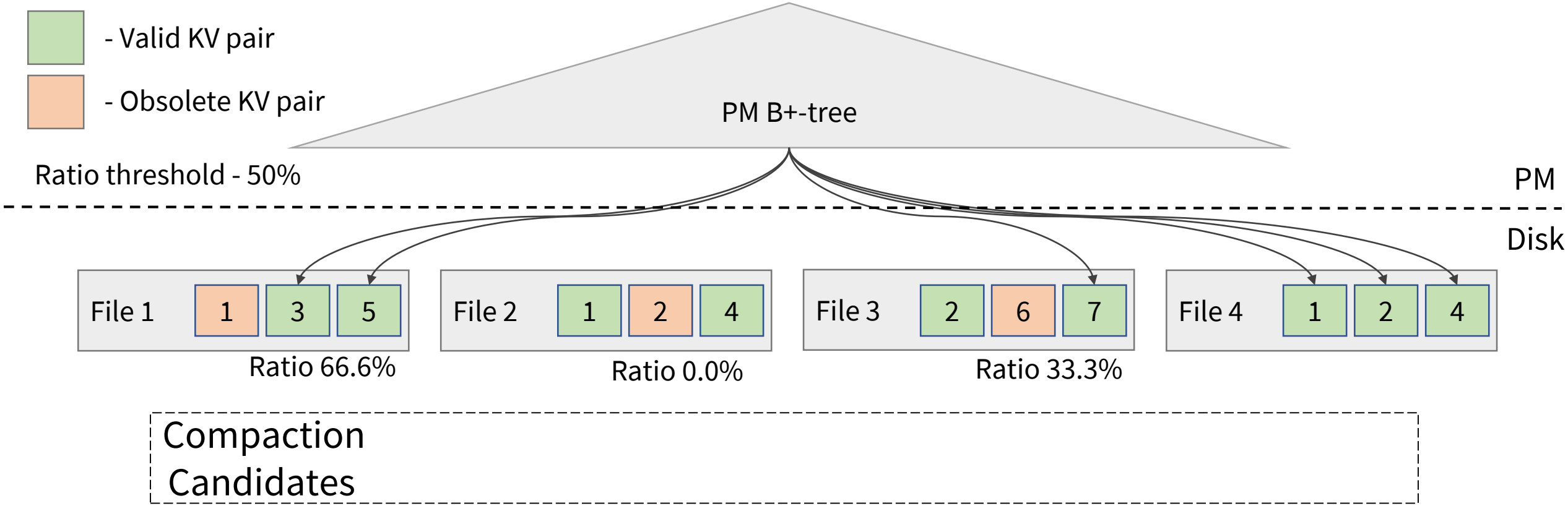
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



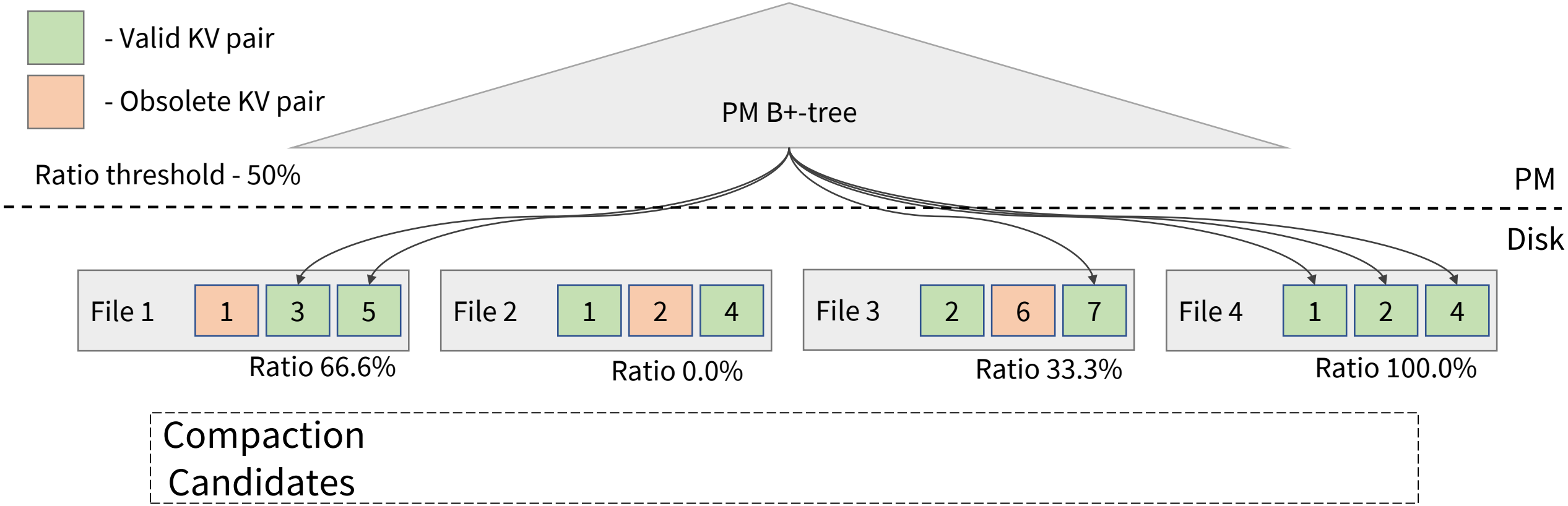
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



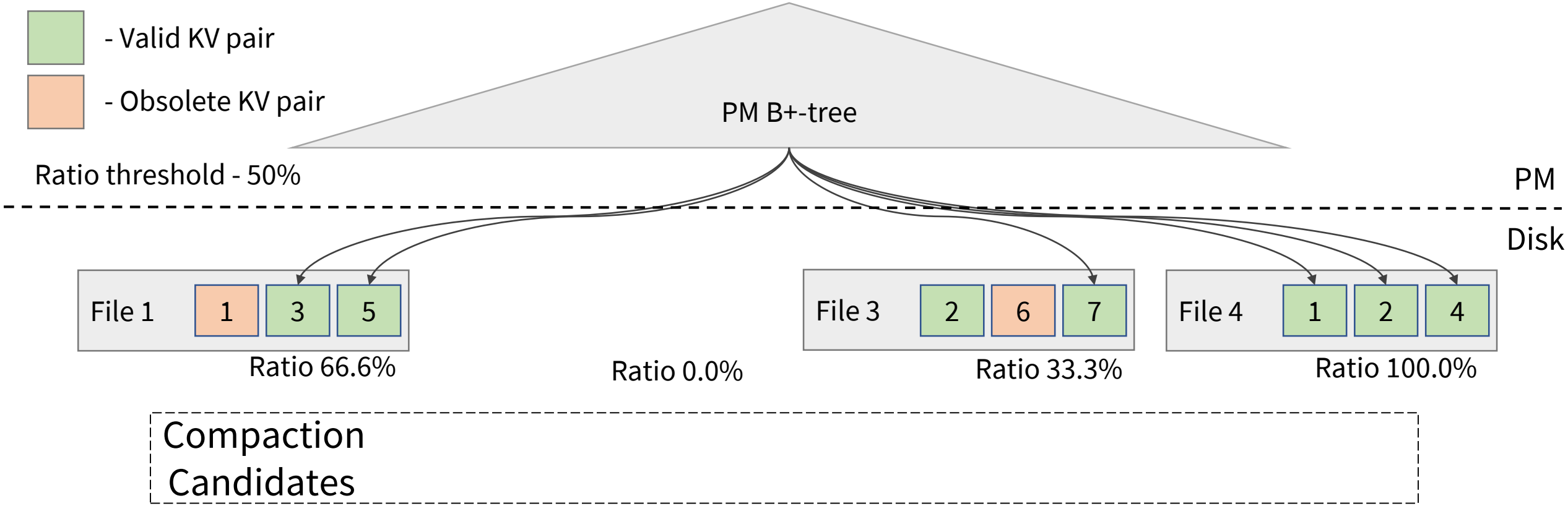
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



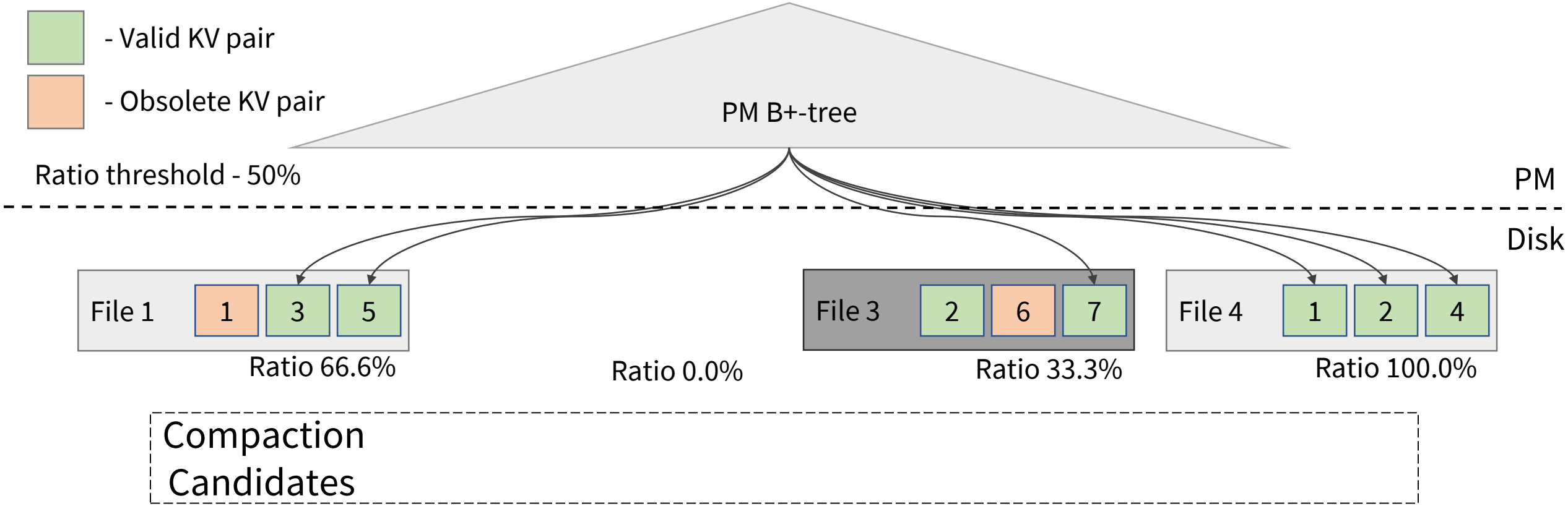
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



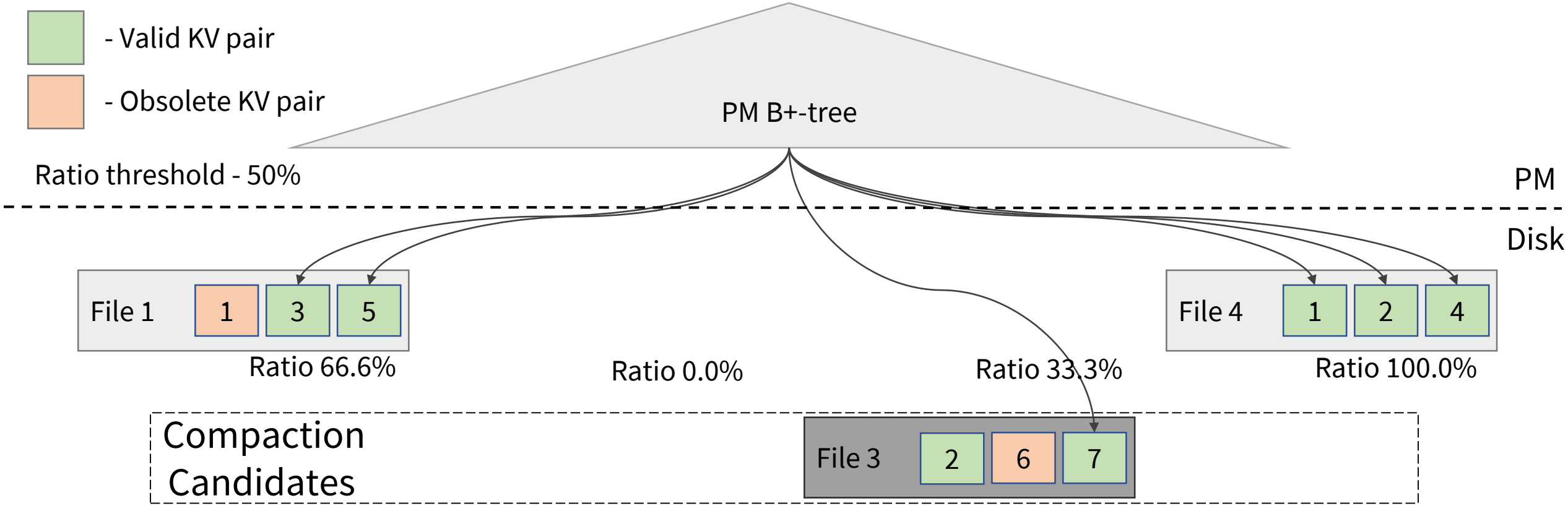
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



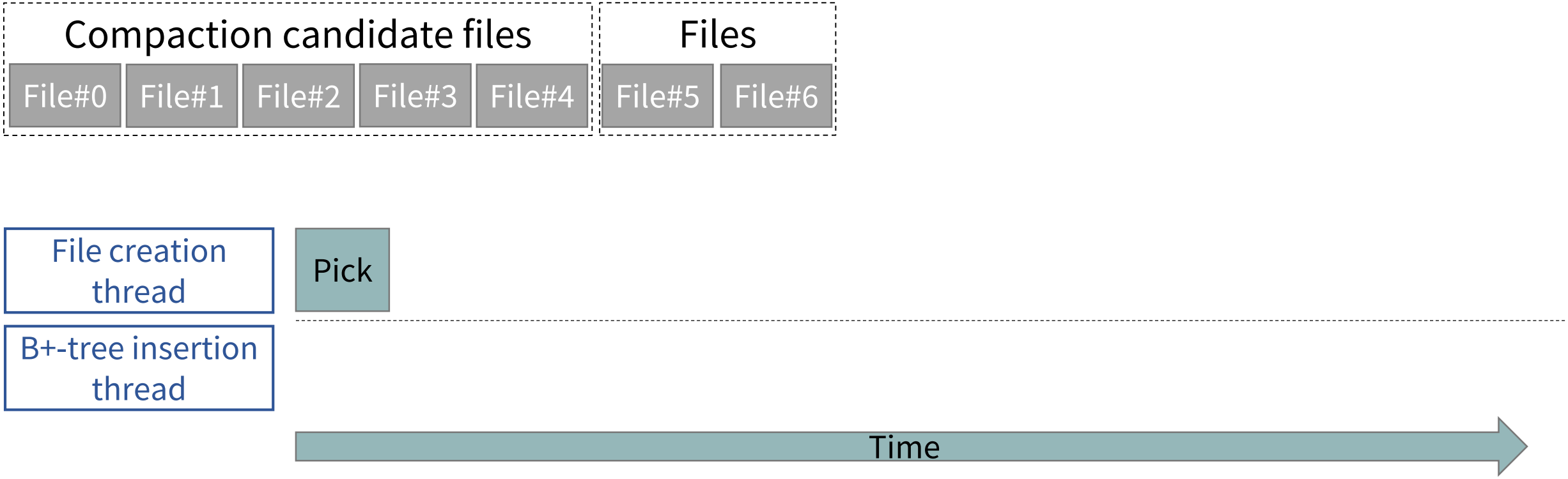
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates



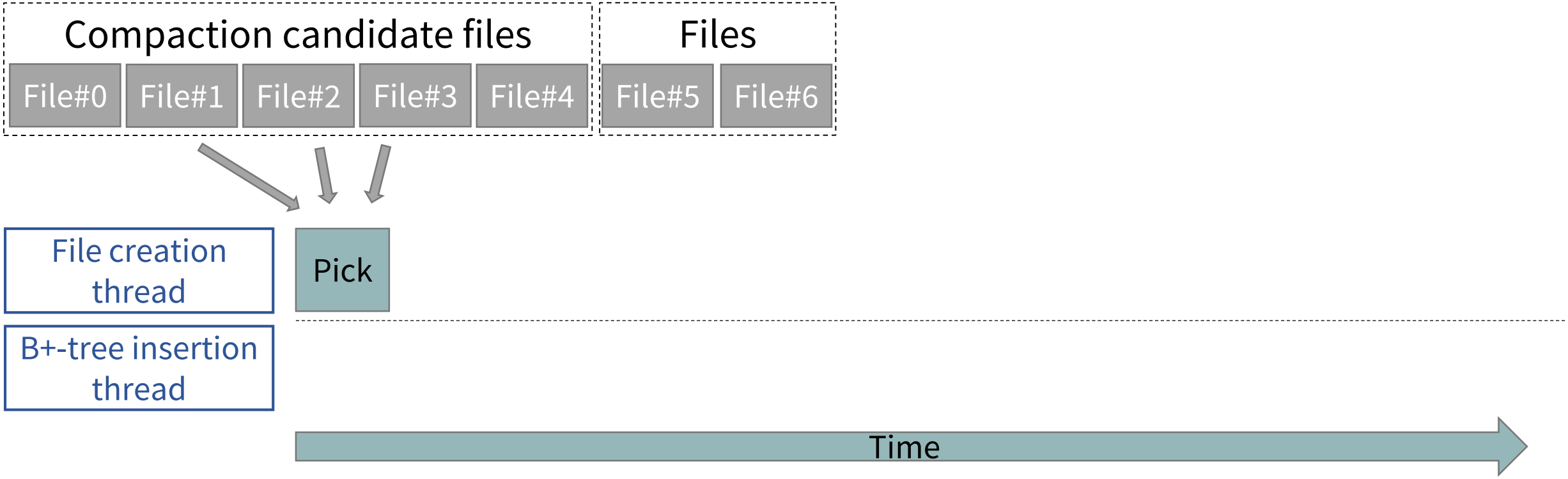
Compaction

- Compaction triggered when there are too many compaction candidate files



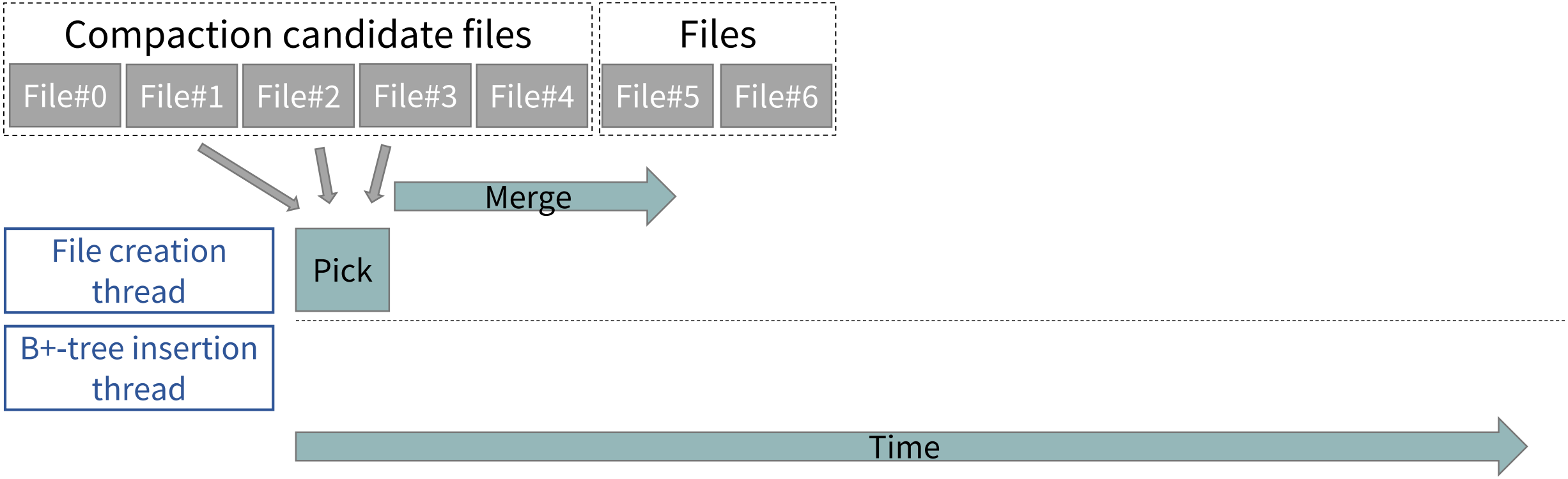
Compaction

- Compaction triggered when there are too many compaction candidate files



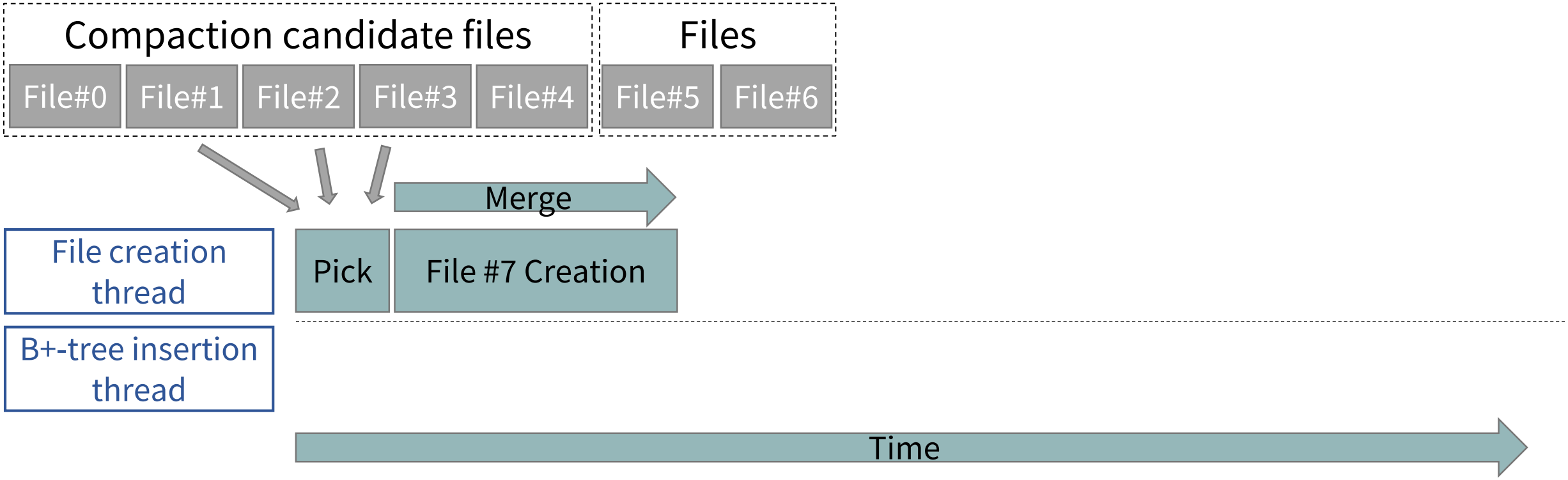
Compaction

- Compaction triggered when there are too many compaction candidate files



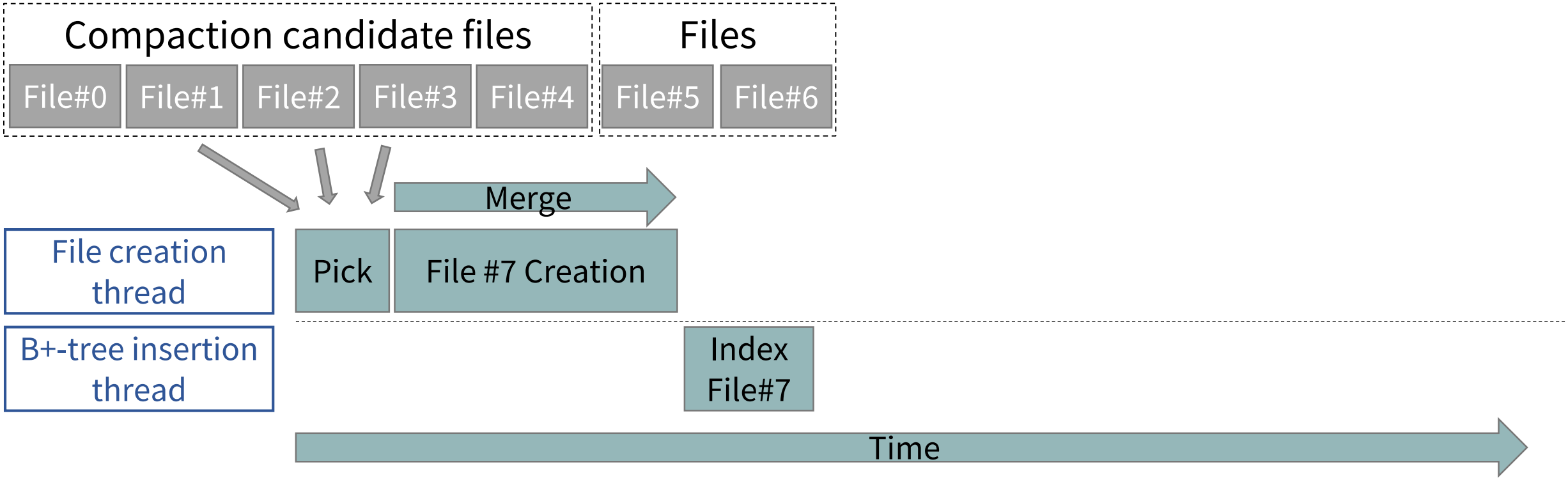
Compaction

- Compaction triggered when there are too many compaction candidate files



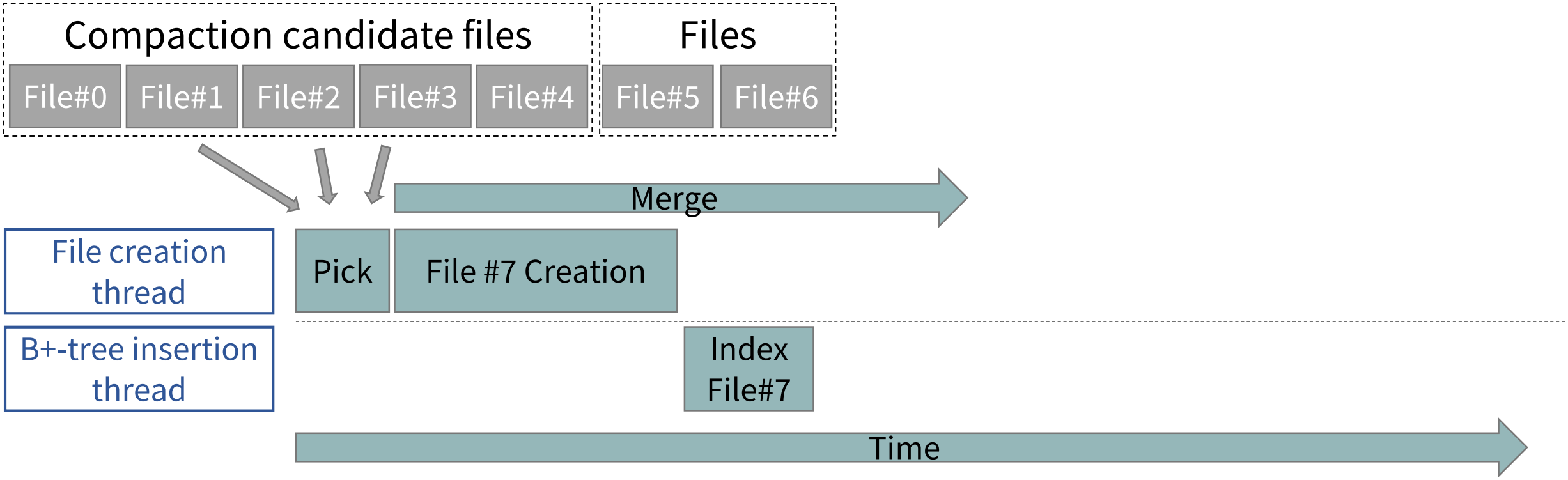
Compaction

- Compaction triggered when there are too many compaction candidate files



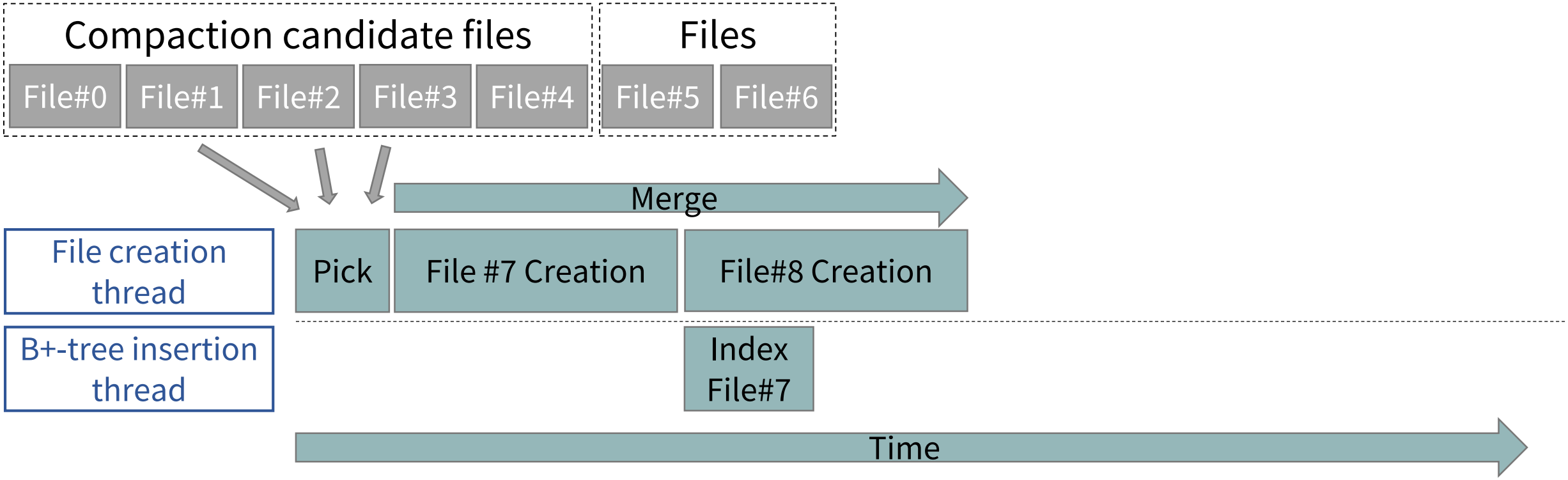
Compaction

- Compaction triggered when there are too many compaction candidate files



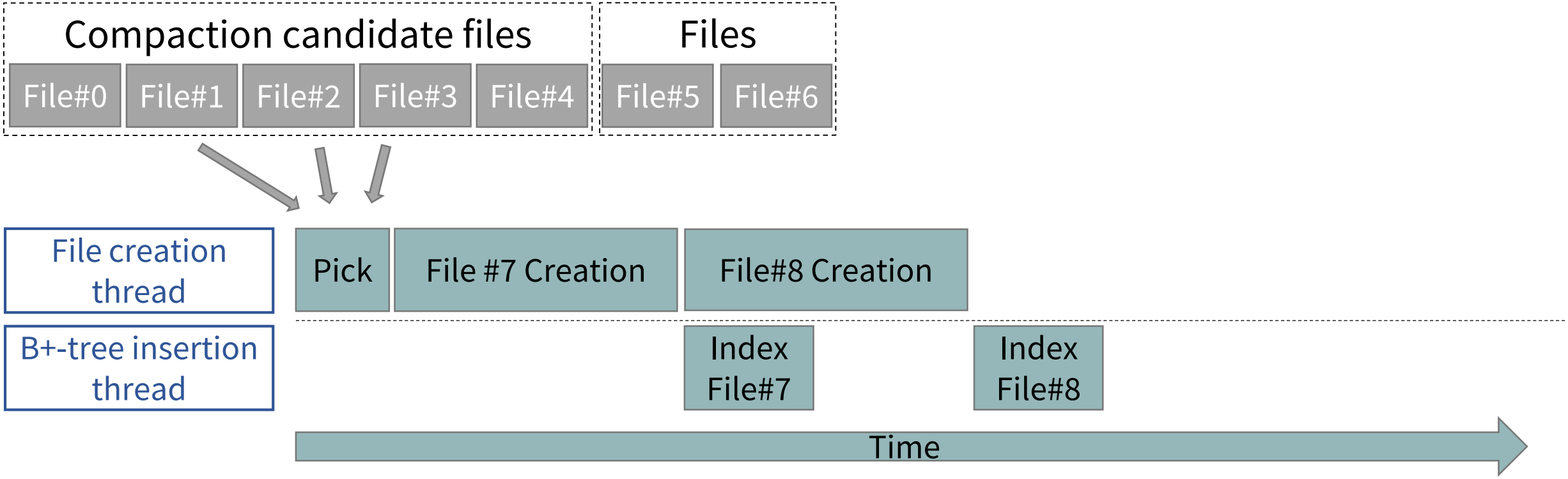
Compaction

- Compaction triggered when there are too many compaction candidate files



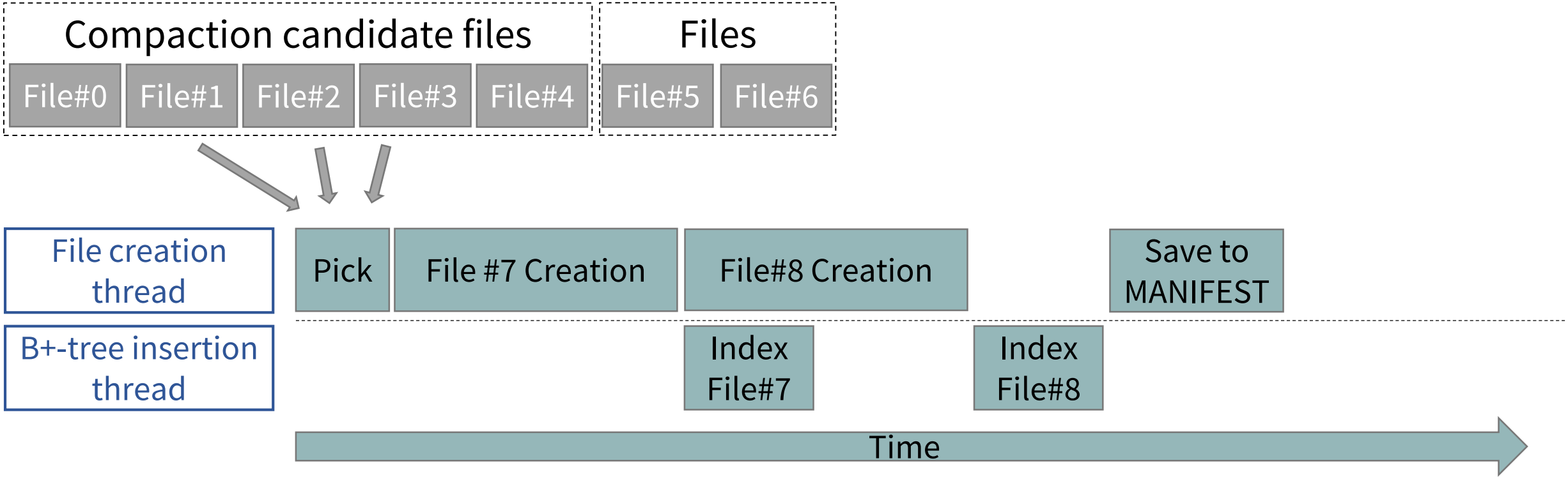
Compaction

- Compaction triggered when there are too many compaction candidate files



Compaction

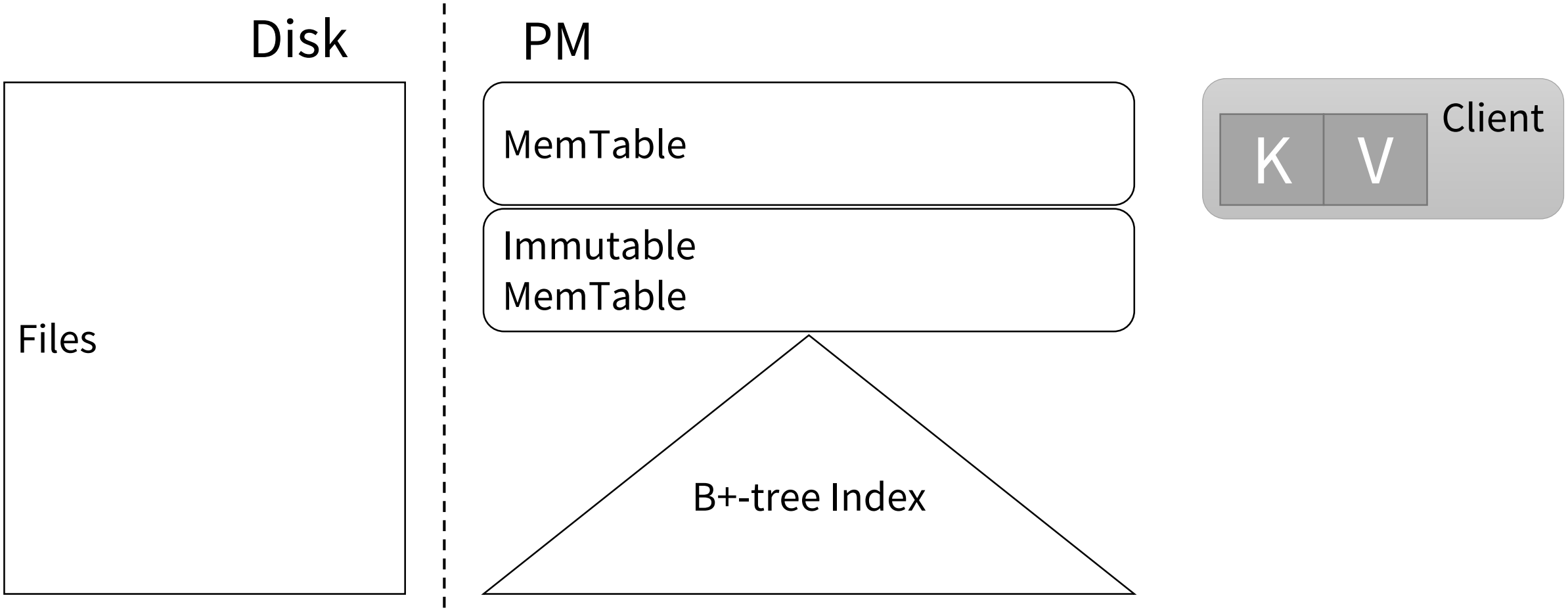
- Compaction triggered when there are too many compaction candidate files



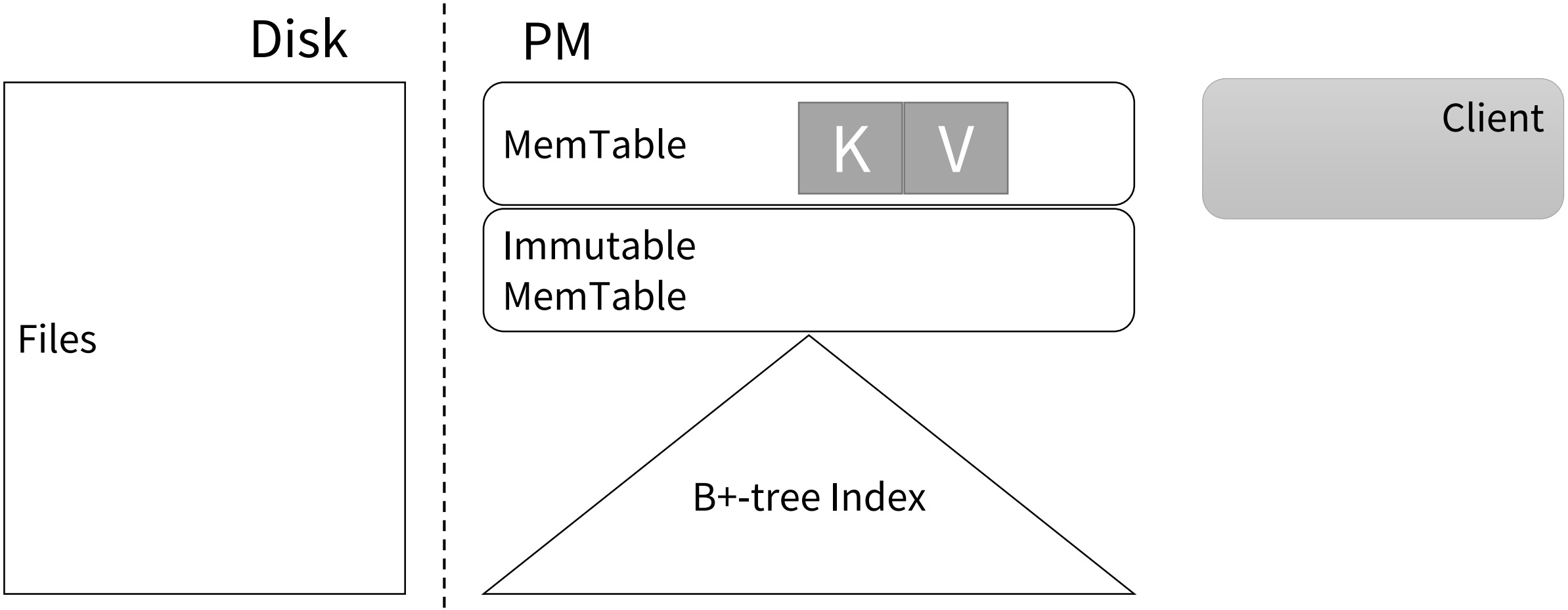
General operations

- Put
- Put if exists/Put if not exists
- Get
- Scan

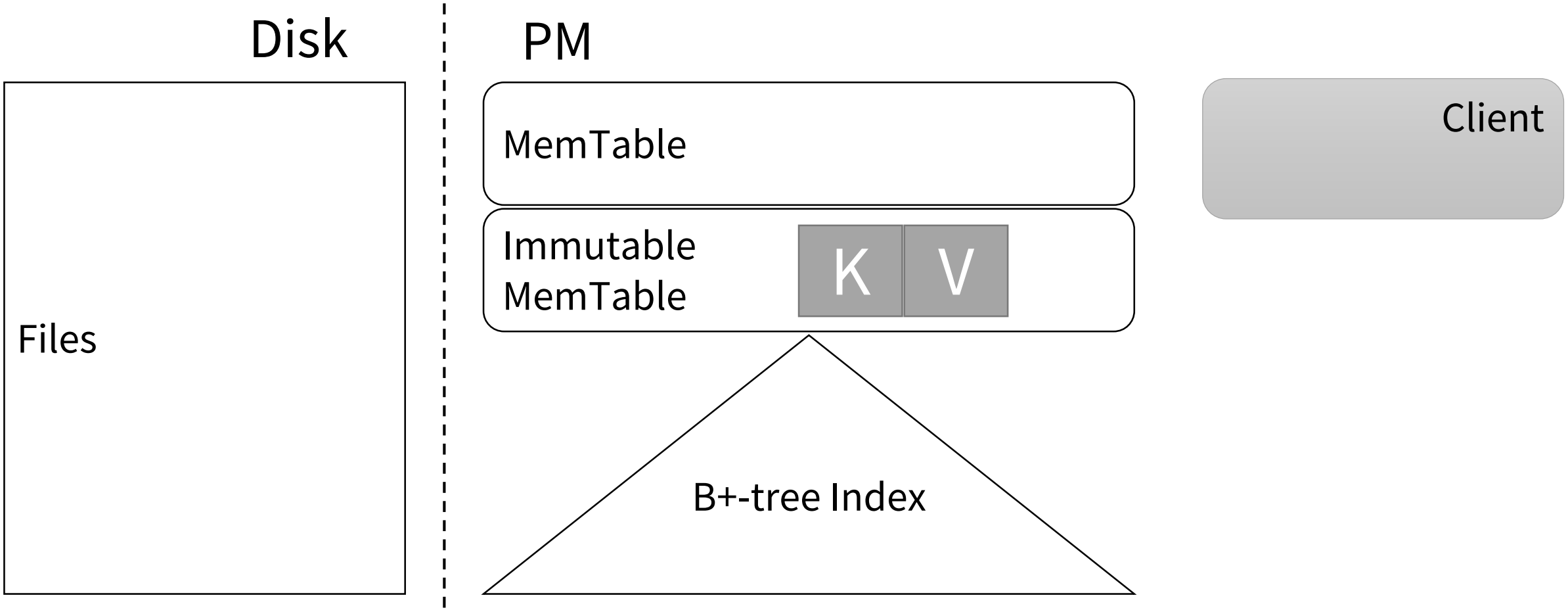
Put(key, value)



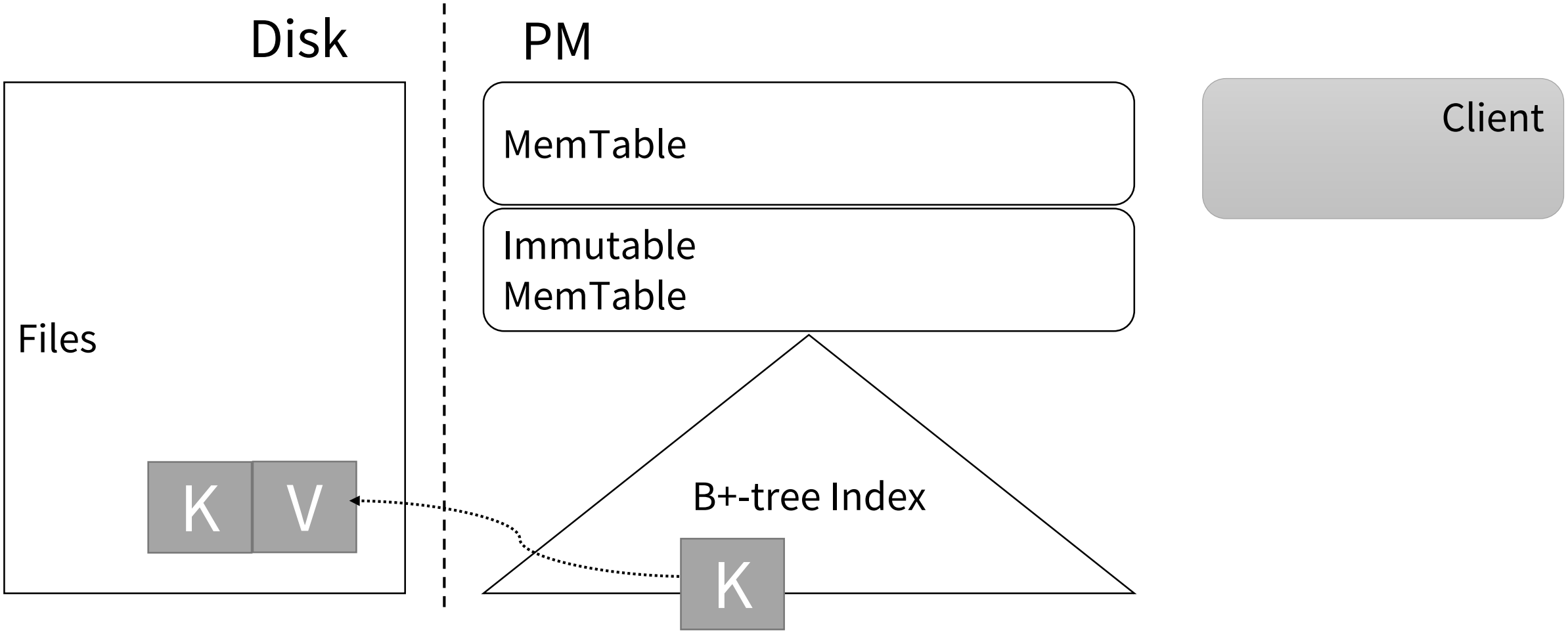
Put(key, value)



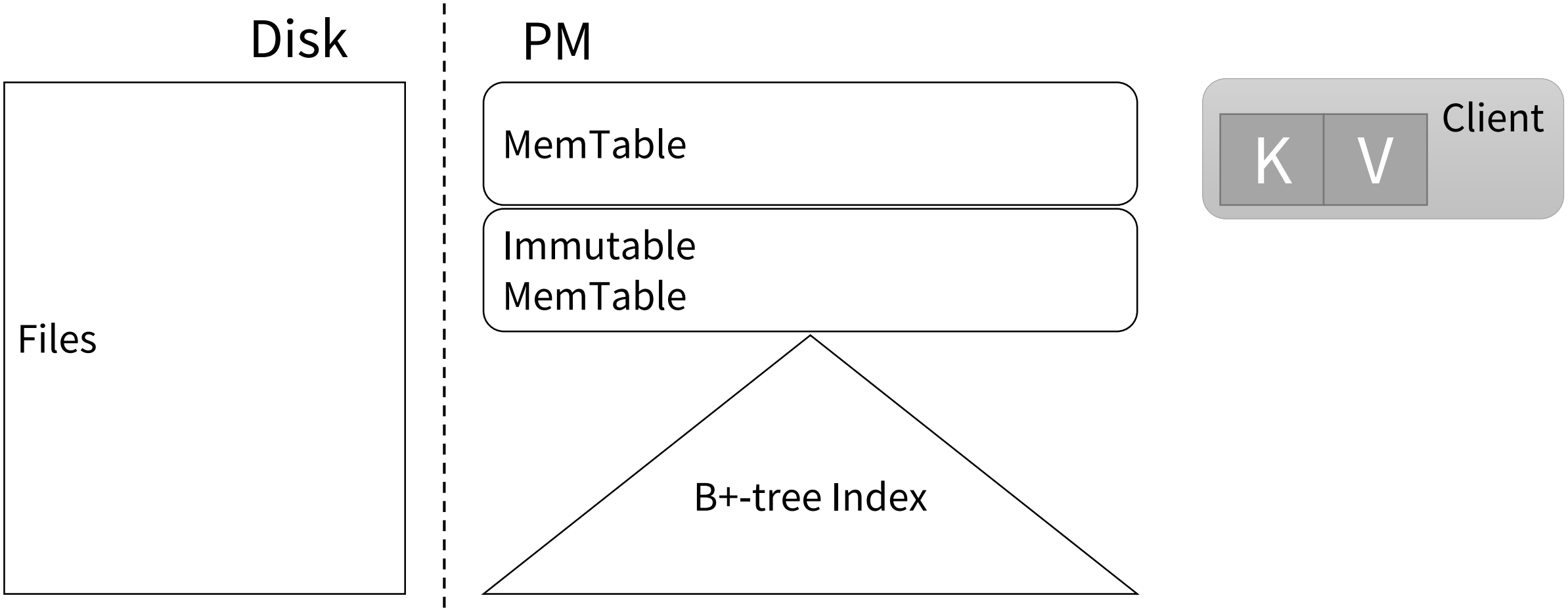
Put(key, value)



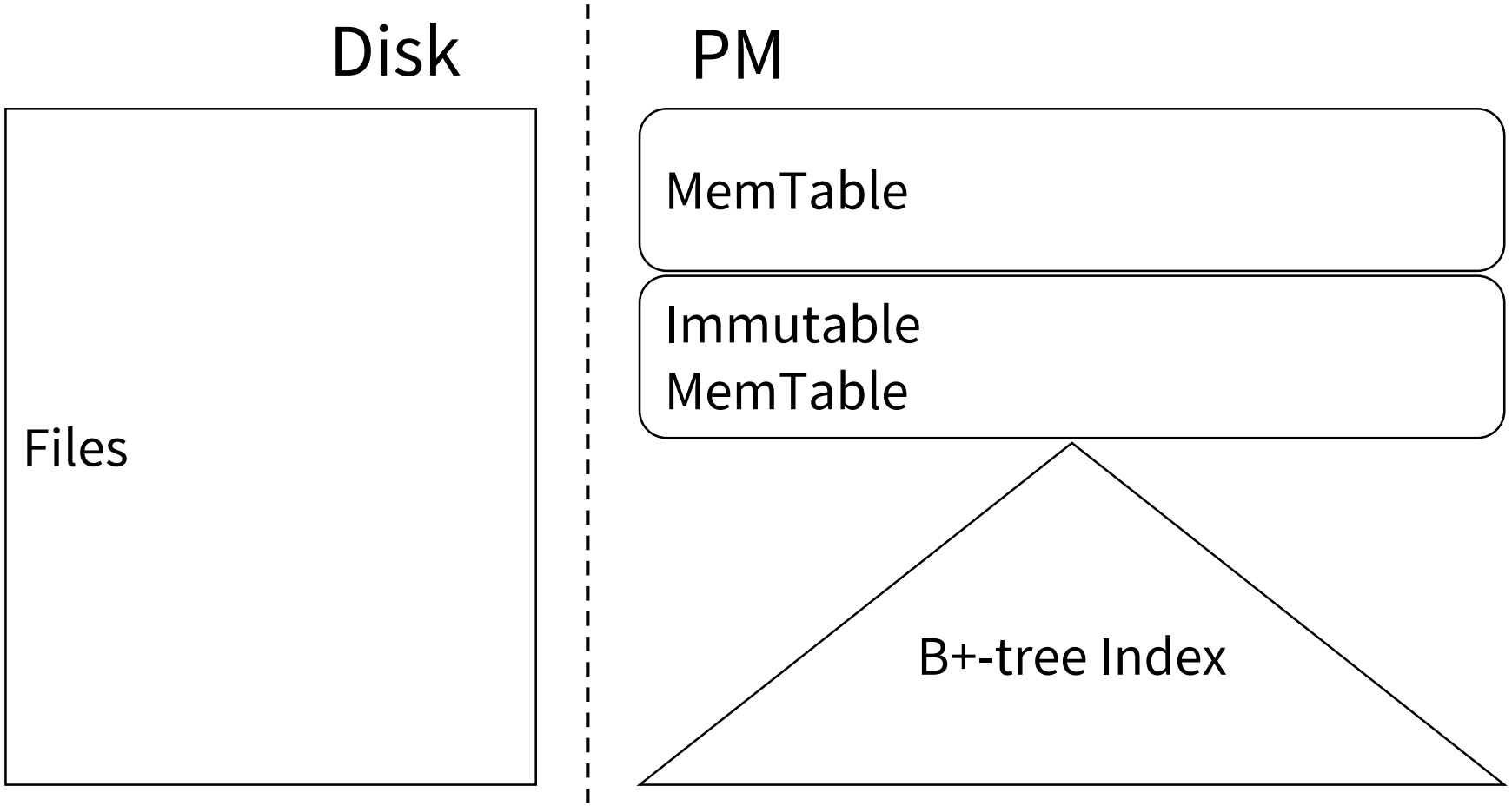
Put(key, value)

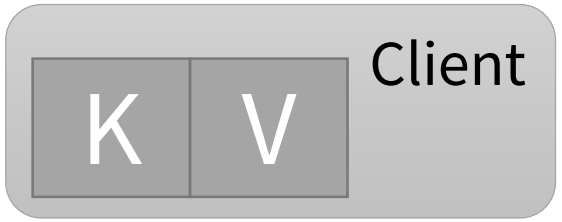


Put(key, value) if exists/if not exists



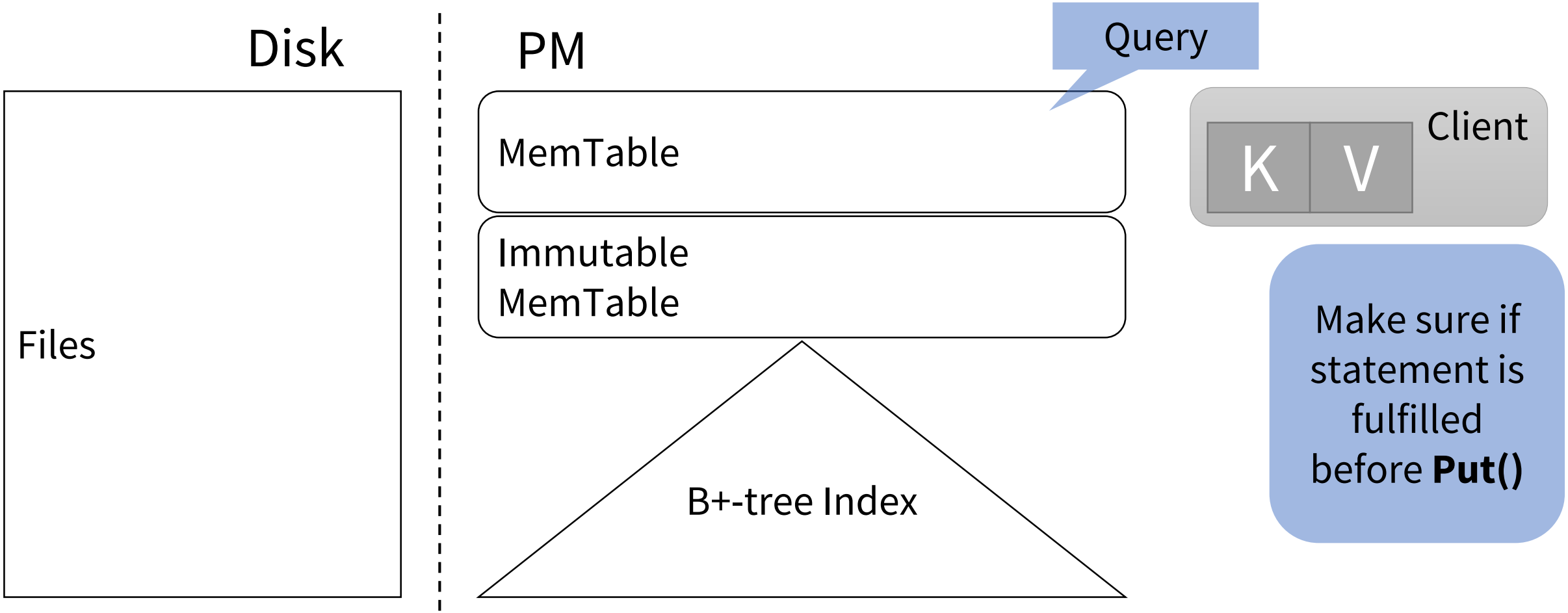
Put(key, value) if exists/if not exists



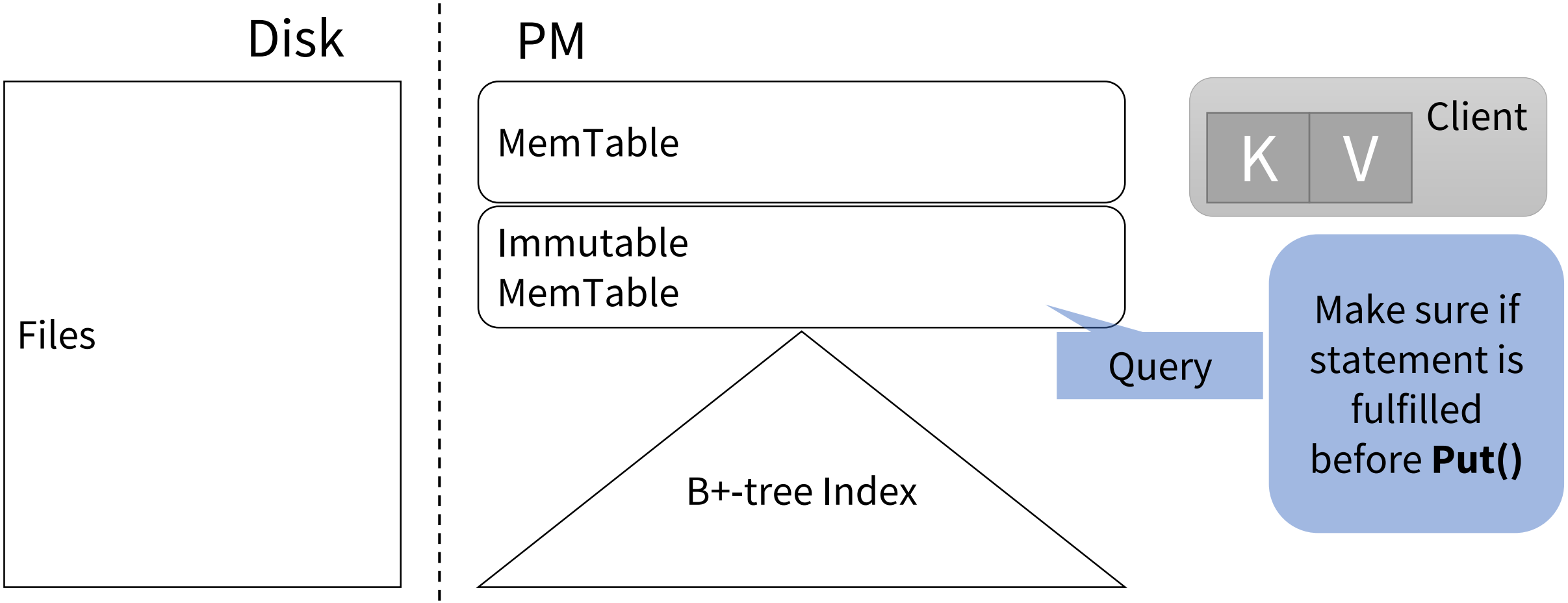


Make sure if statement is fulfilled before **Put()**

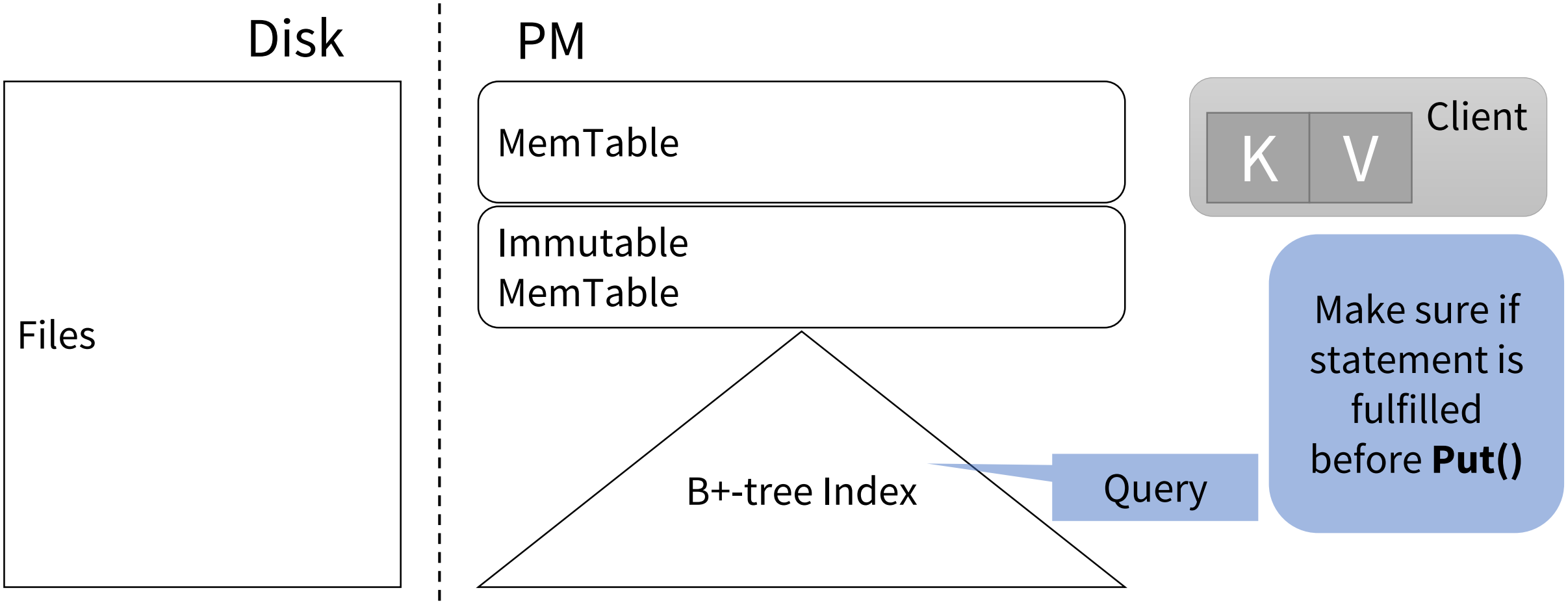
Put(key, value) if exists/if not exists



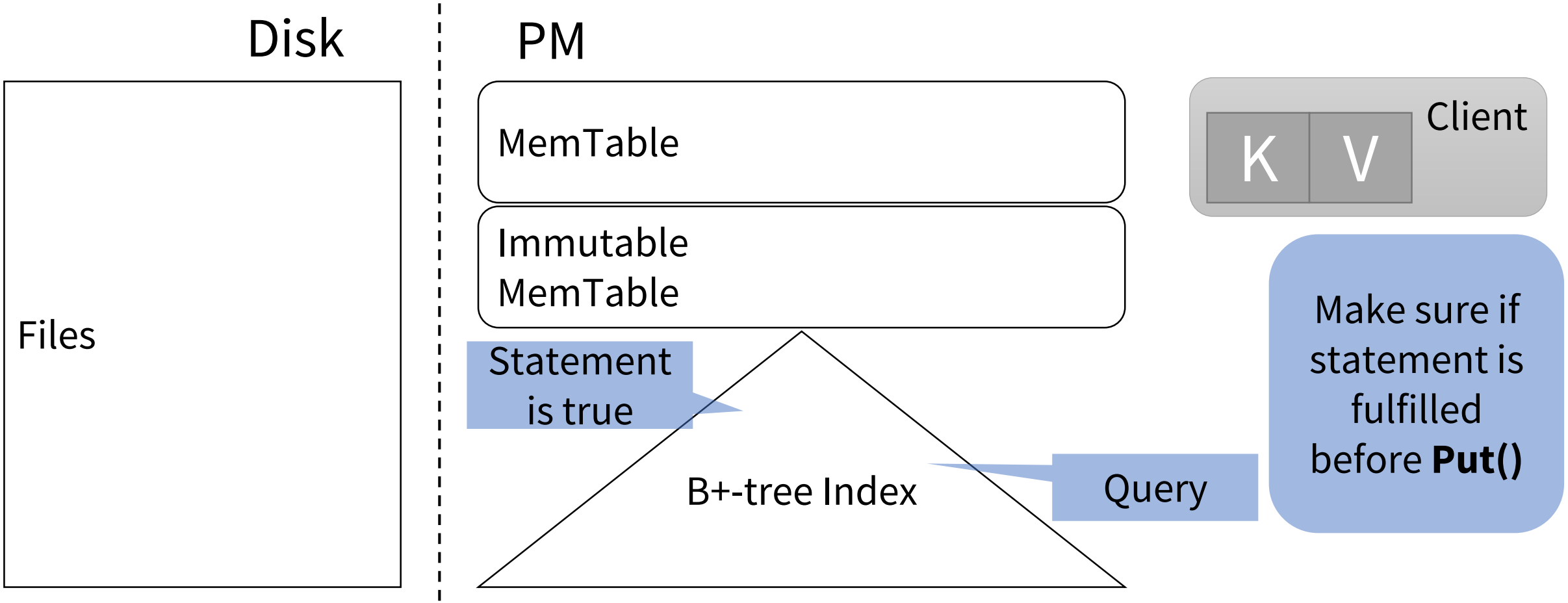
Put(key, value) if exists/if not exists



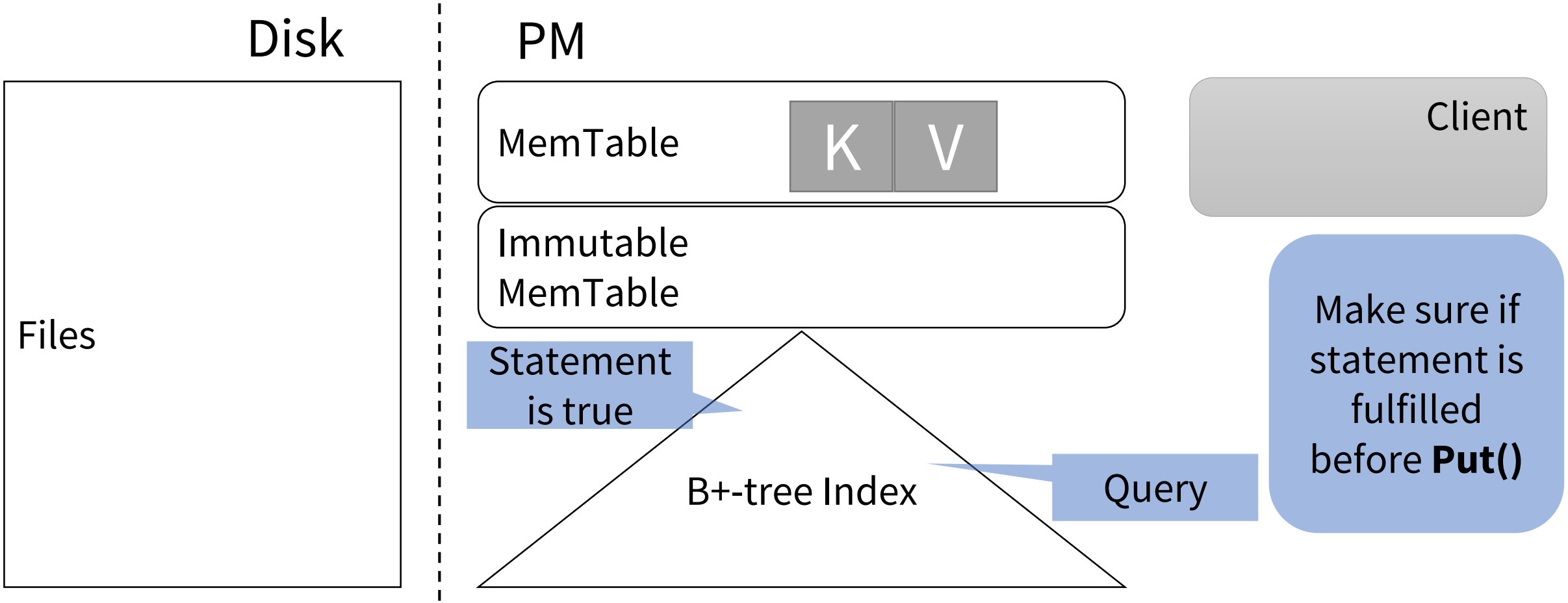
Put(key, value) if exists/if not exists



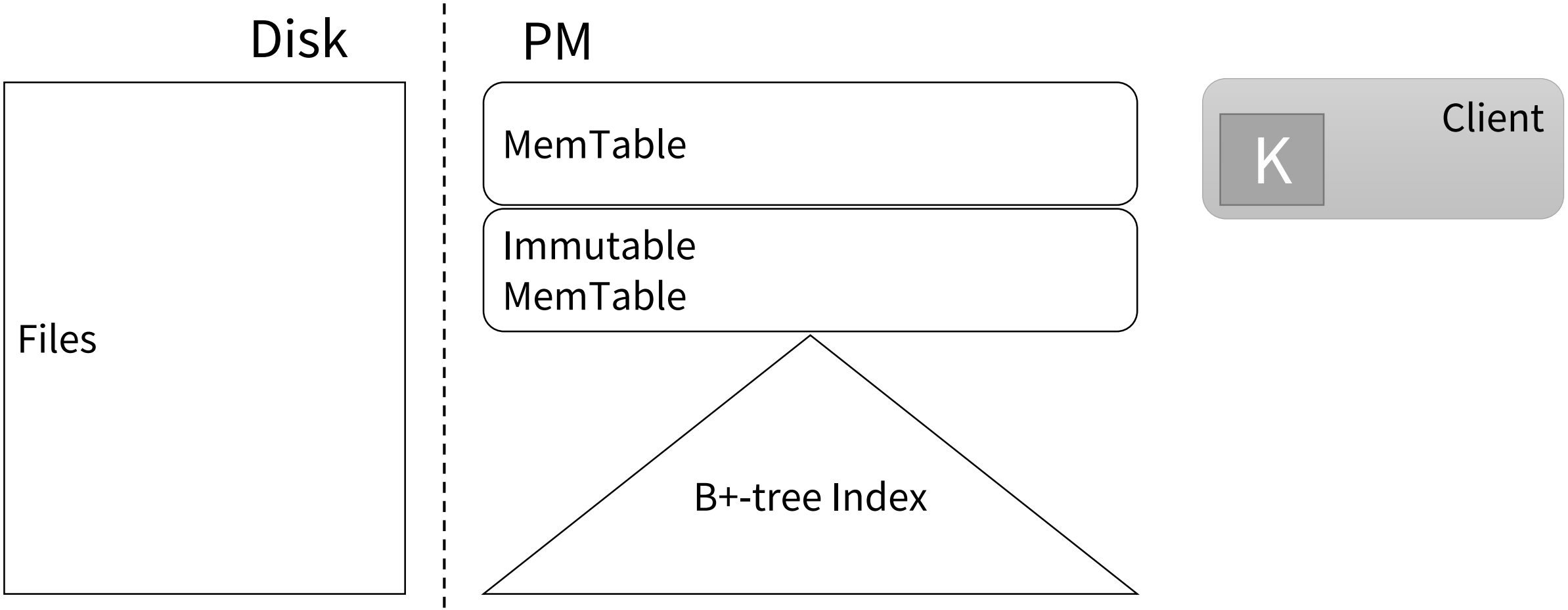
Put(key, value) if exists/if not exists



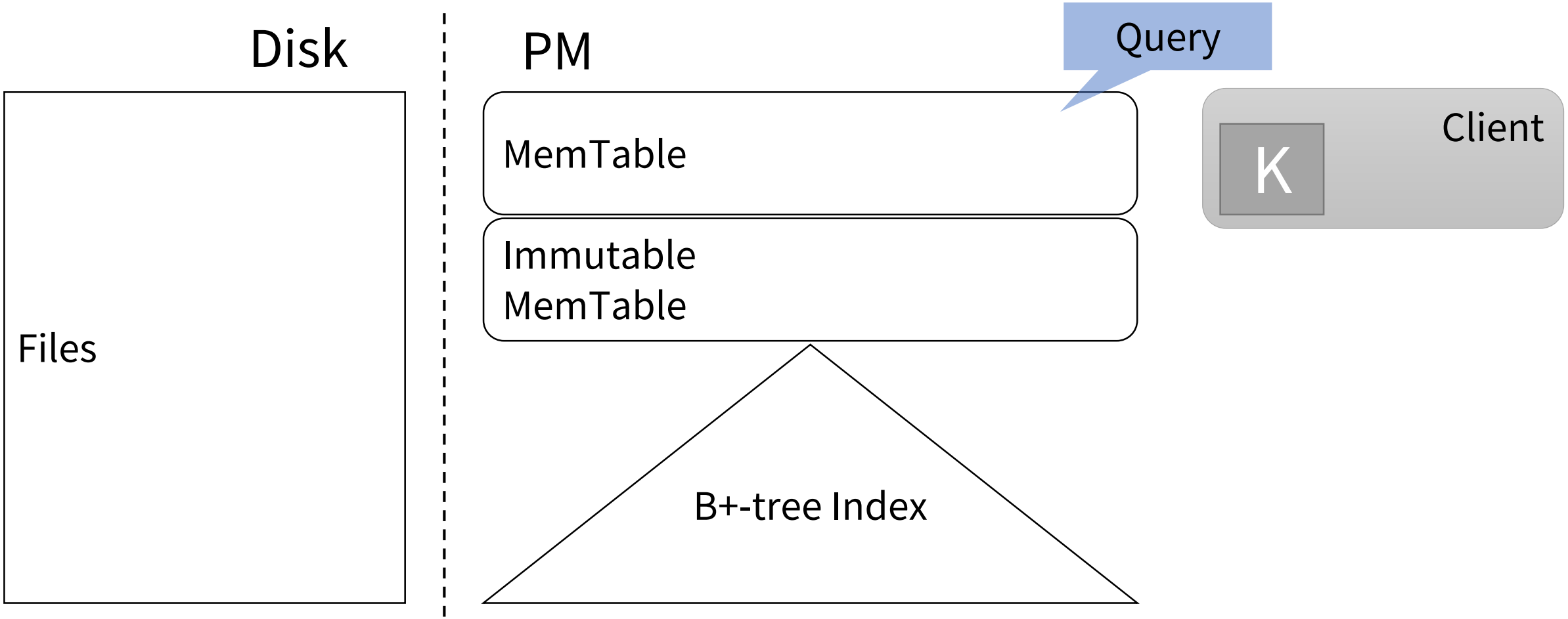
Put(key, value) if exists/if not exists



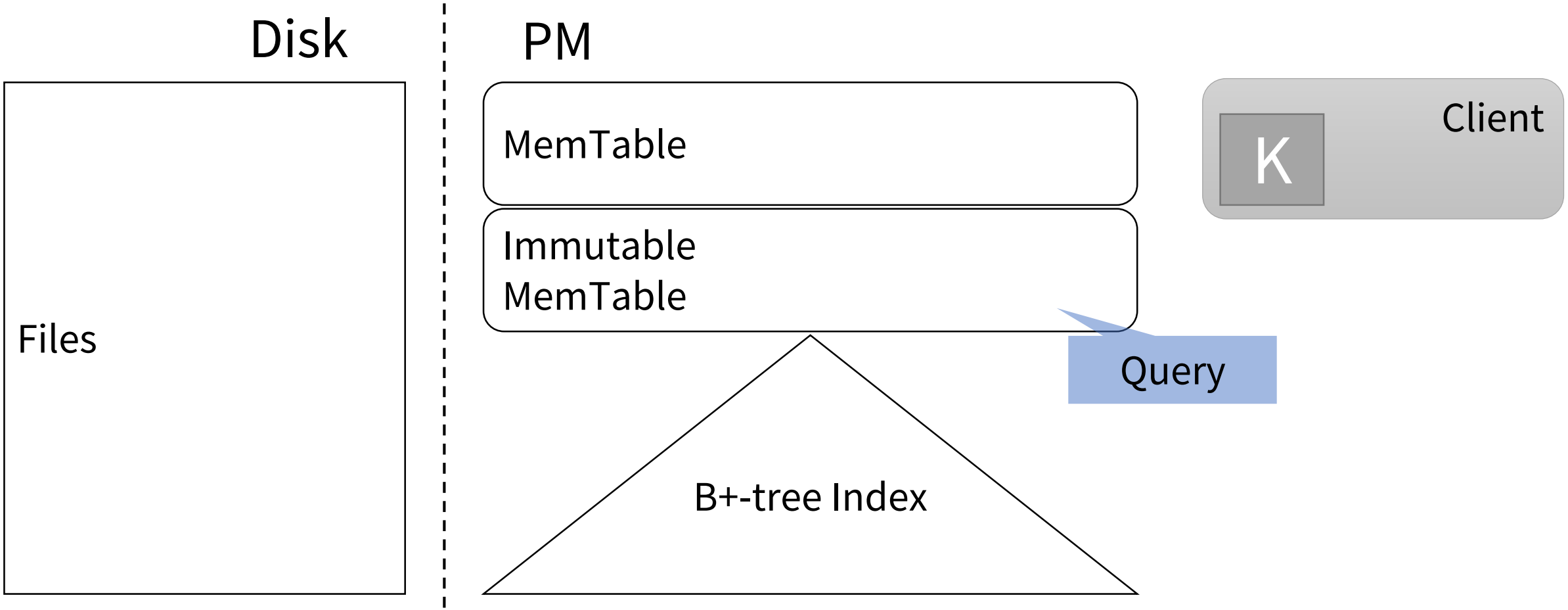
Get(key)


Disk
PM
Files
MemTable
Immutable
MemTable
B+-tree Index
Client
K

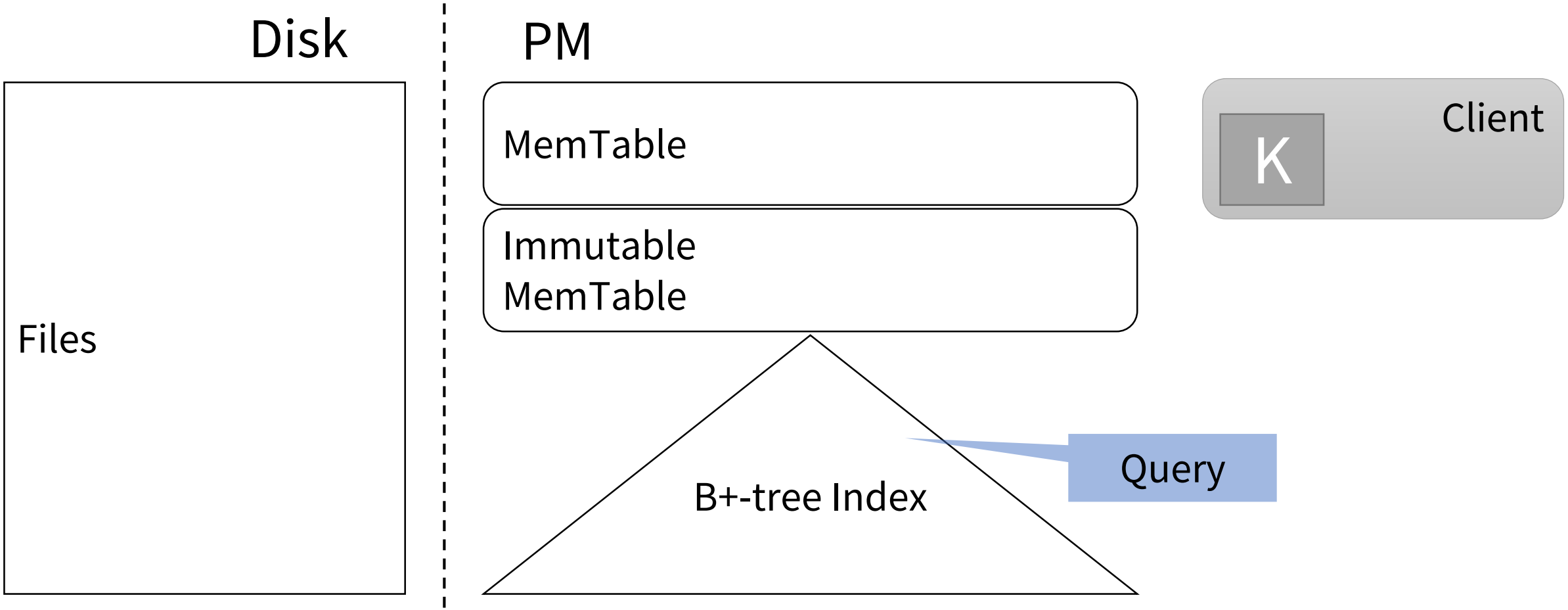
Get(key)


Disk
PM
Query
Client
K
MemTable
Immutable
MemTable
Files
B+-tree Index

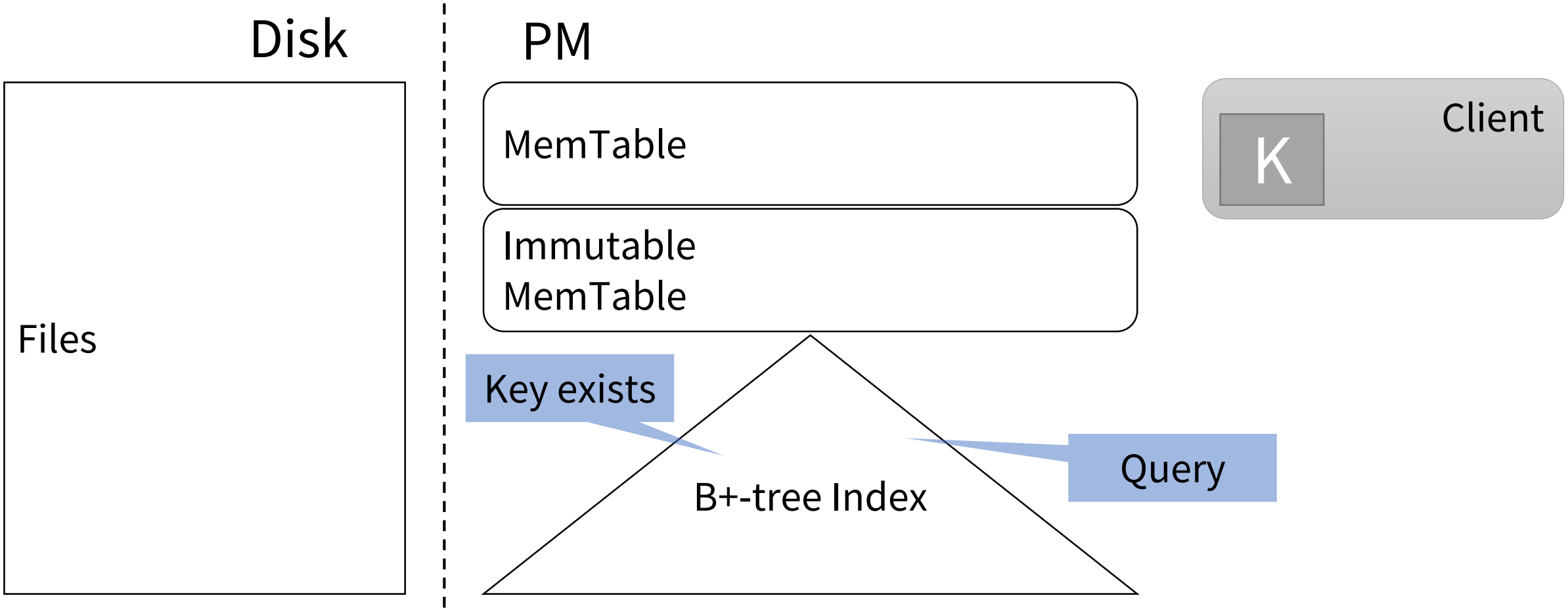
Get(key)


Disk
PM
Files
MemTable
Immutable
MemTable
B+-tree Index
Query
K
Client

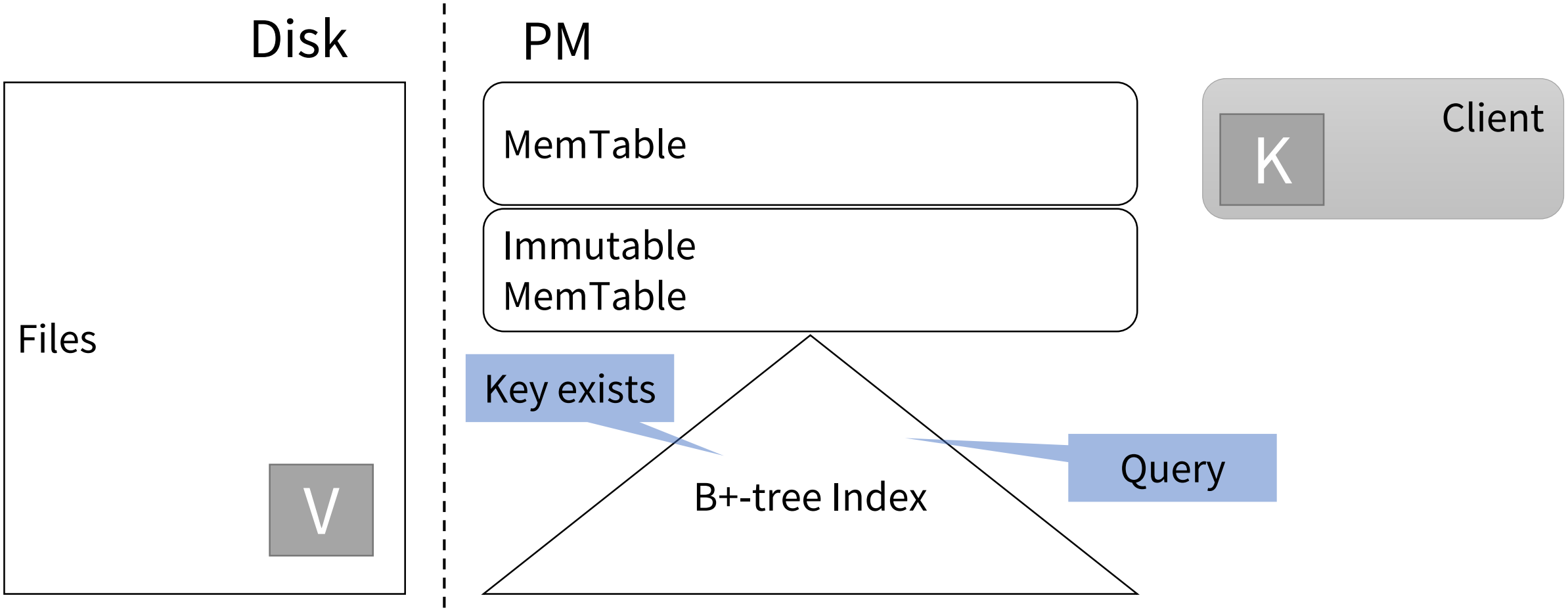
Get(key)


Disk
PM
Files
MemTable
Immutable
MemTable
B+-tree Index
Query
K
Client

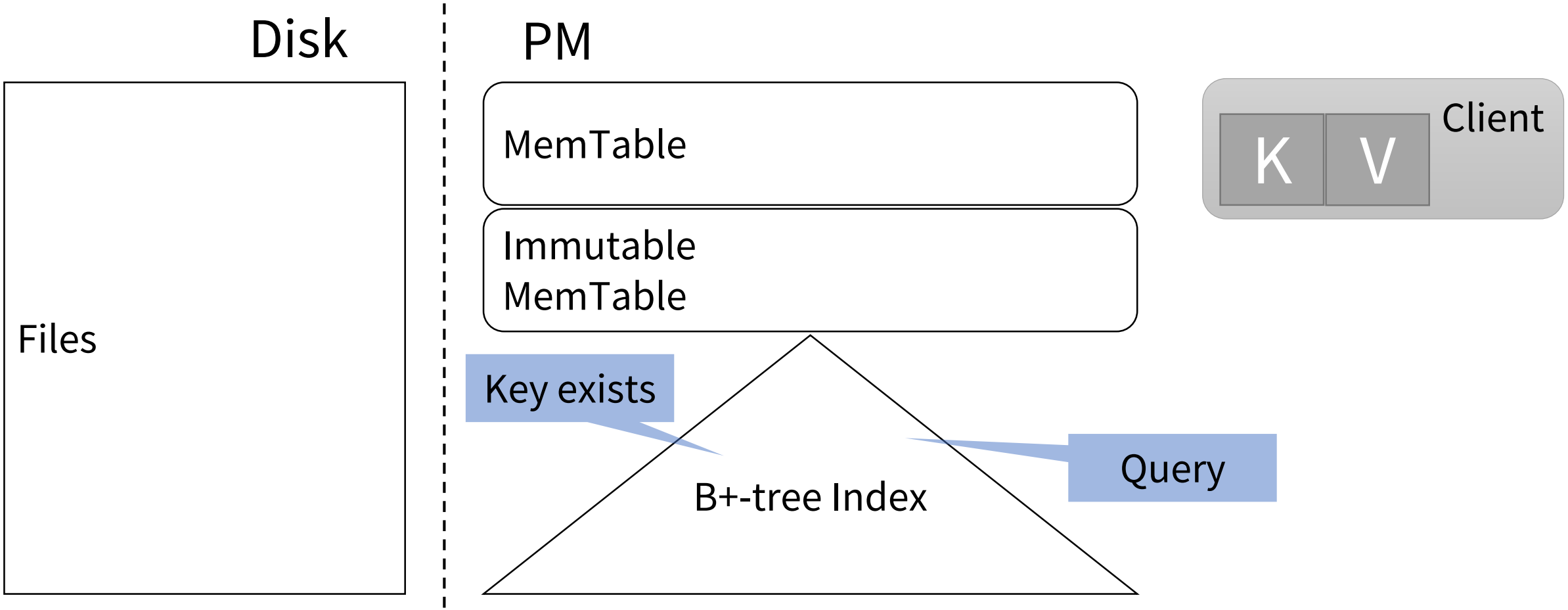
Get(key)


Disk
PM
Files
MemTable
Immutable
MemTable
Client
K
Key exists
Query
B+-tree Index

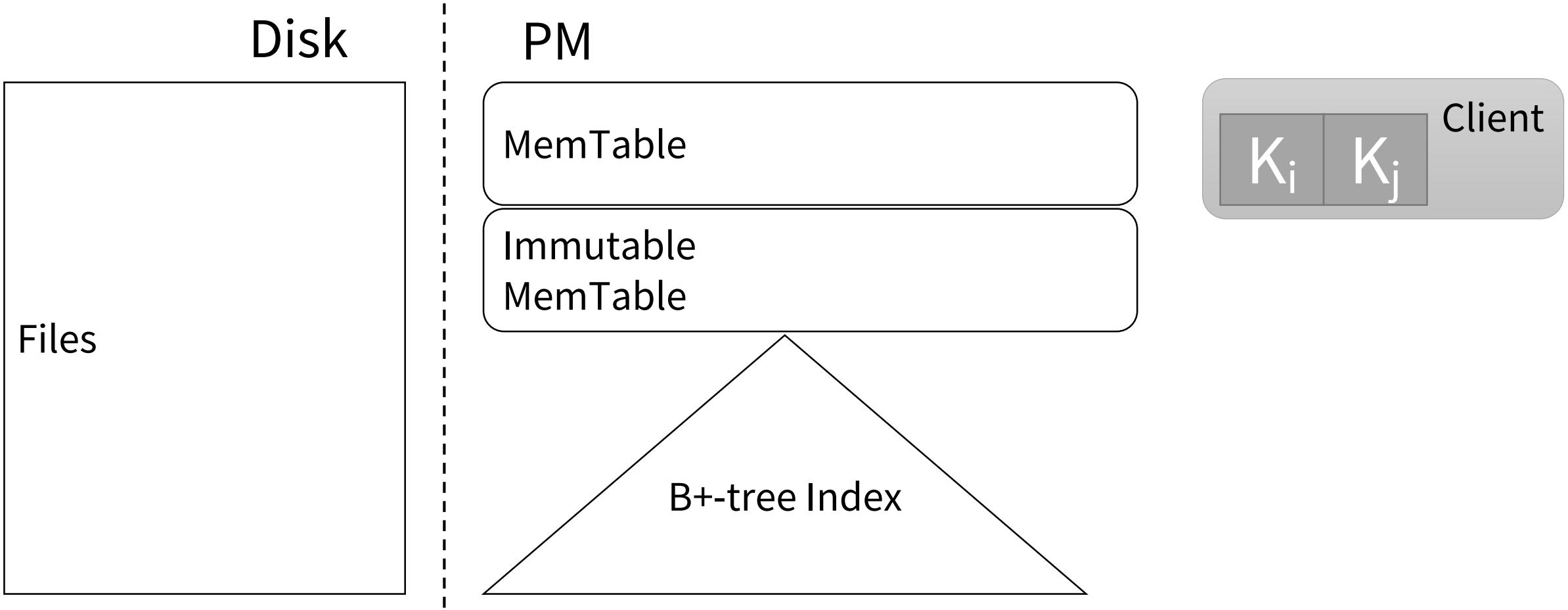
Get(key)


Disk
PM
Files
V
MemTable
Immutable
MemTable
Key exists
B+-tree Index
Query
Client
K

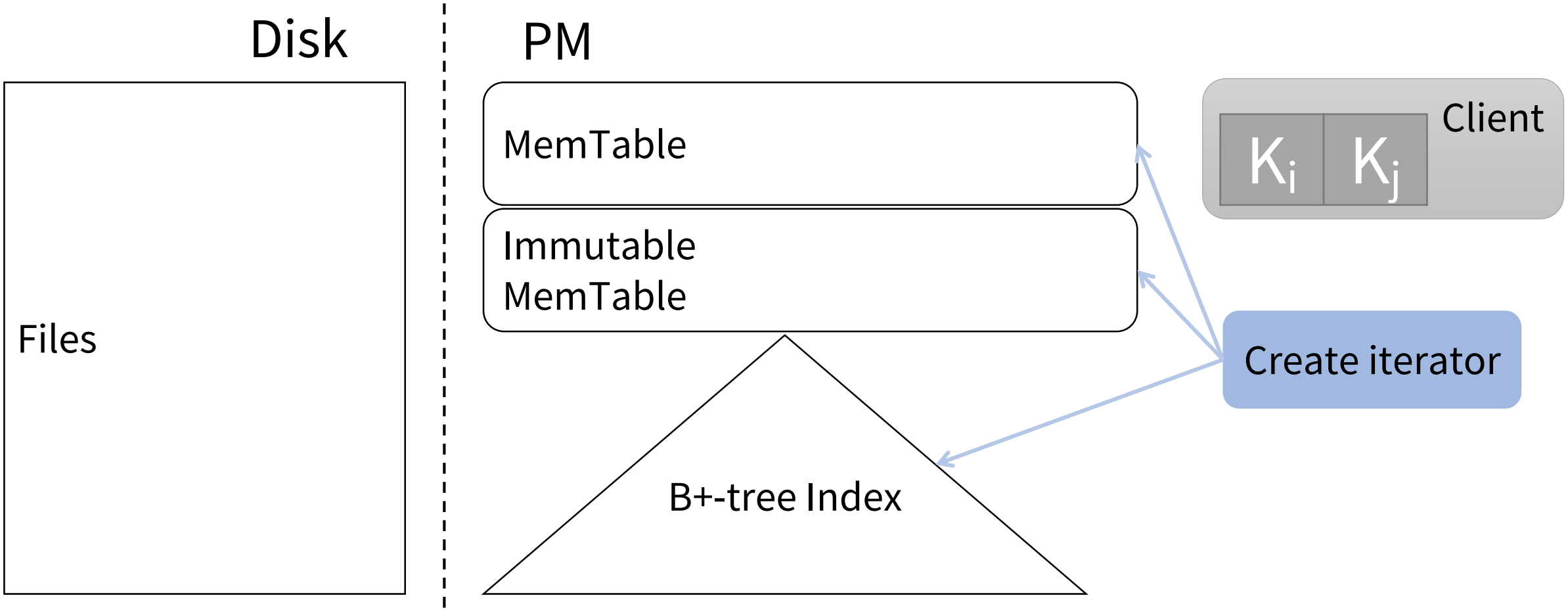
Get(key)


Disk
PM
Files
MemTable
Immutable
MemTable
Key exists
B+-tree Index
Query
K
V
Client

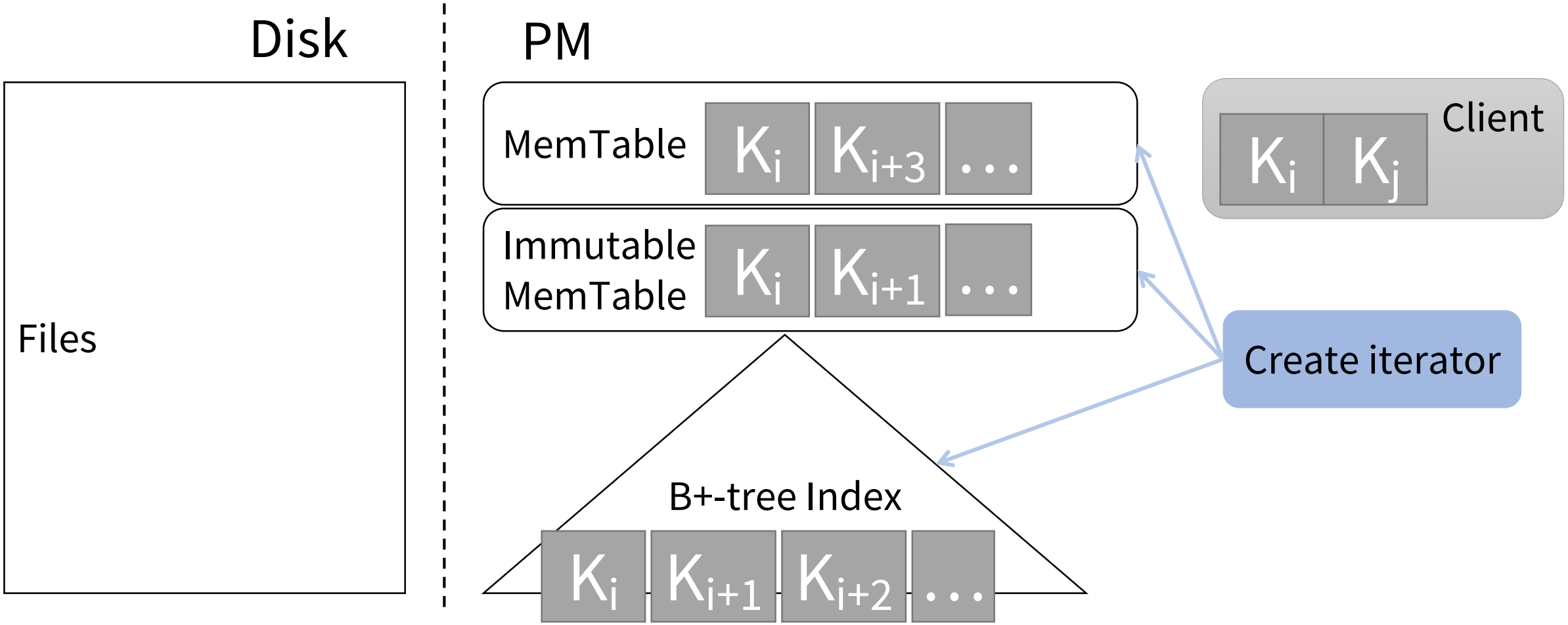
Scan(key_i, key_j)



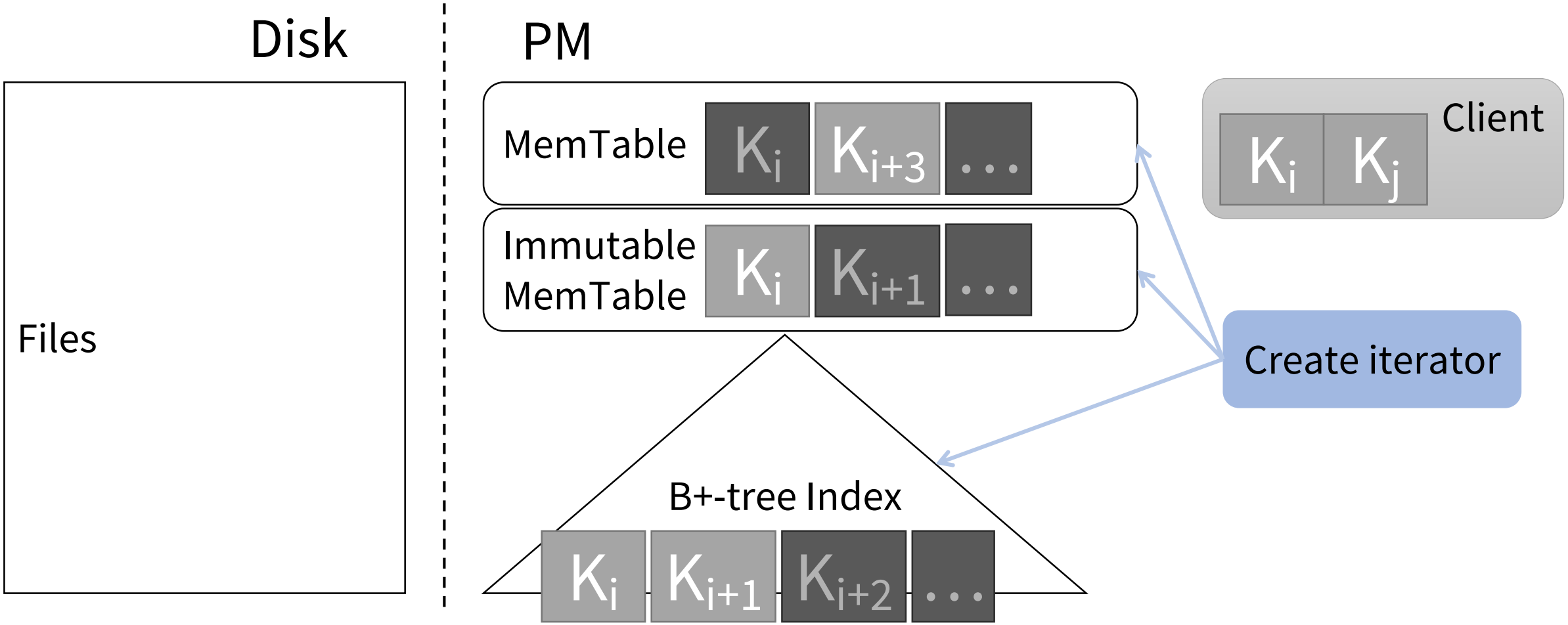
Scan(key_i, key_j)



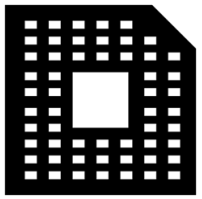
Scan(key_i, key_j)



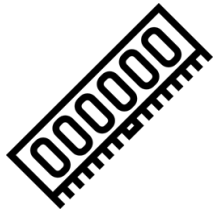
Scan(key_i, key_j)



Evaluation



Intel Xeon
E5-2640 v3



DRAM: **4GB**
Emulated PM: **7GB**

Intel SSD
DC S3520

Ubuntu 18.04
Kernel 4.15

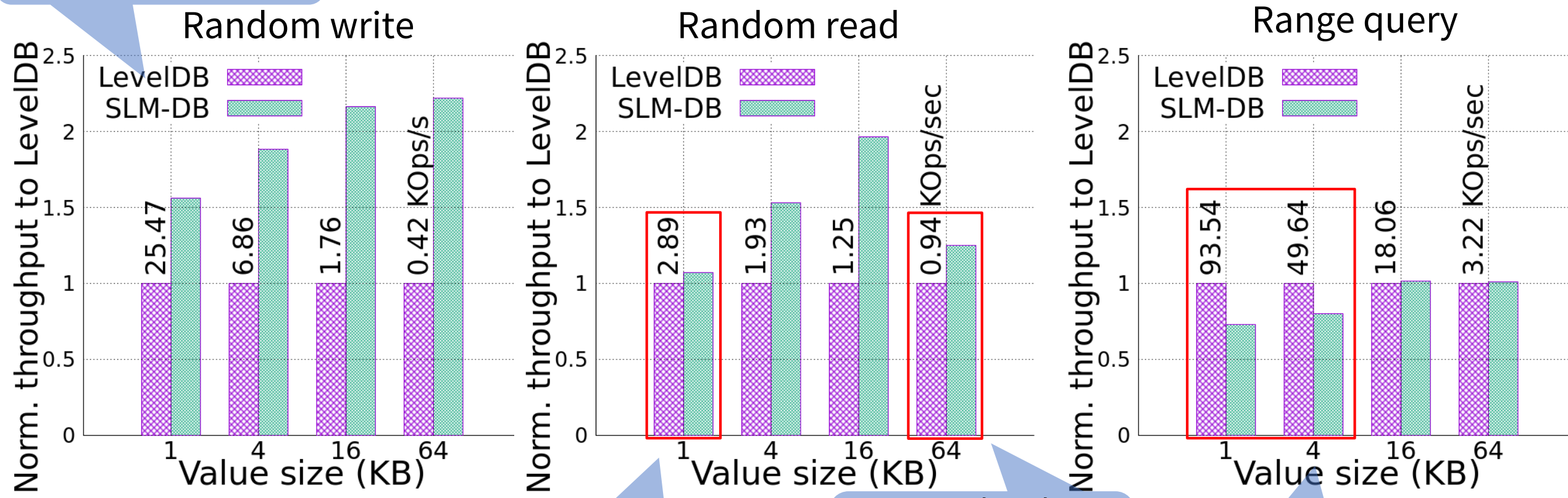
DB: **8GB**/20GB
Memtable: **64MB**

- PM write latency 500ns (5x of DRAM write latency)
- PM read latency & bandwidth same same as DRAM's
- Intel's PMDK used to control PM pool

db_bench microbenchmark

Steady performance increase

Range size = 100



Low file locating overhead

Overhead amortized from large value size

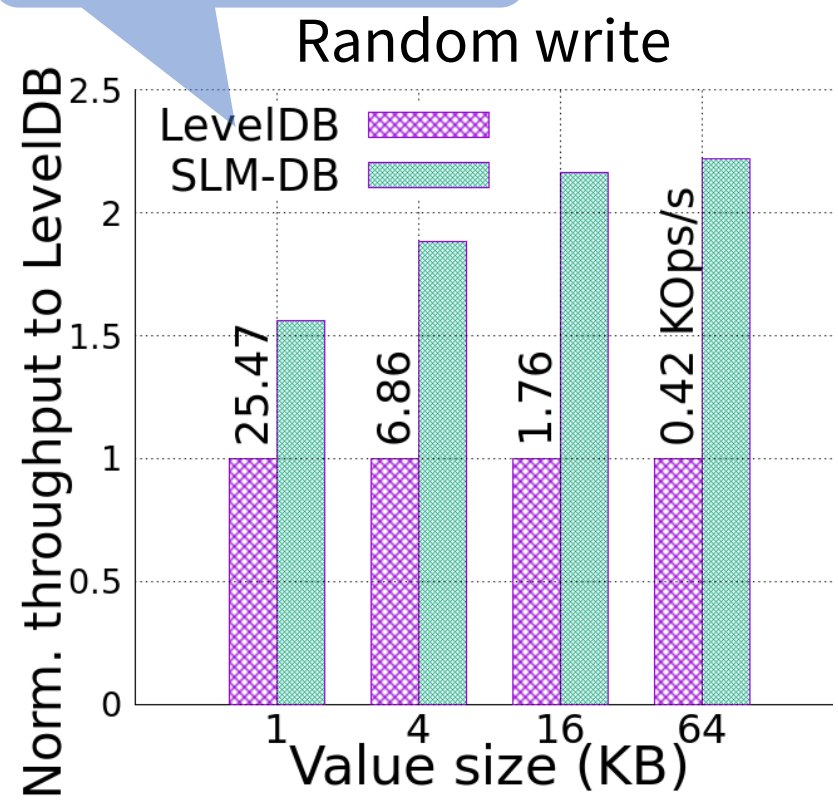
Low sequentiality

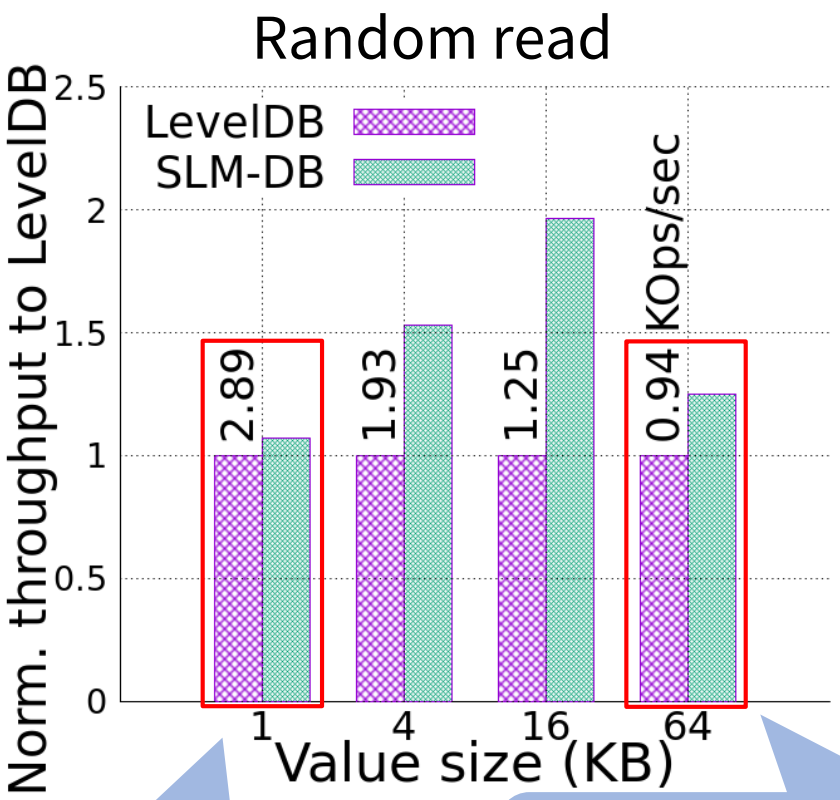
db_bench microbenchmark

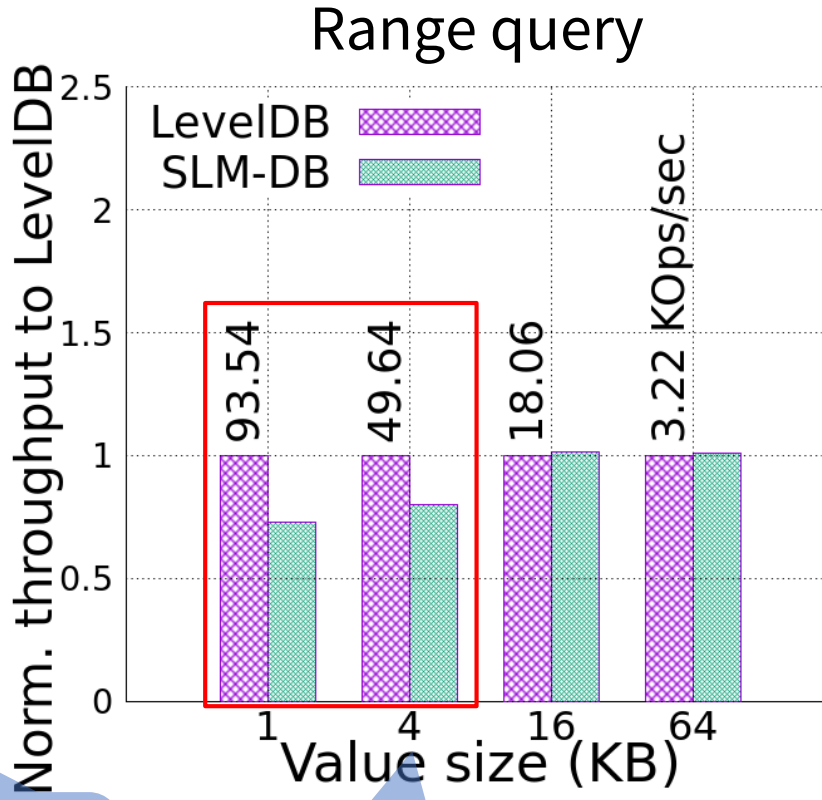
- **~2.56x** less disk write amplification
  - Max 700MB used in PM

Steady performance increase

Range size = 100







Low file locating overhead

Overhead amortized from large value size

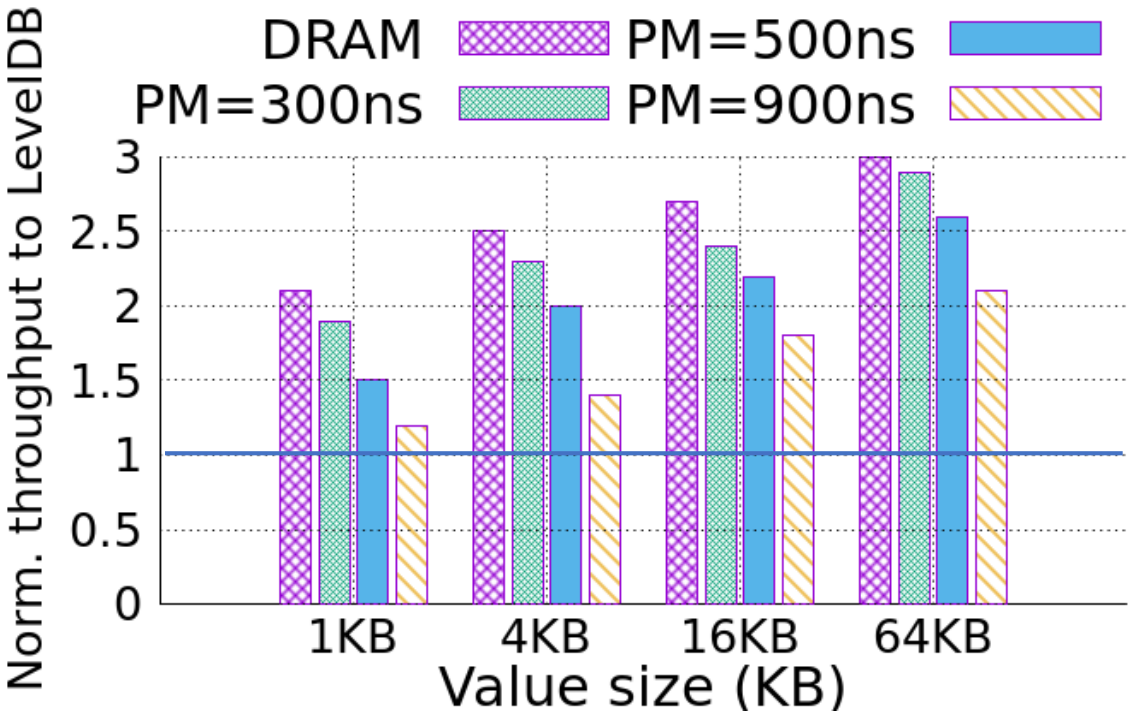
Low sequentiality

PM sensitivity

db_bench

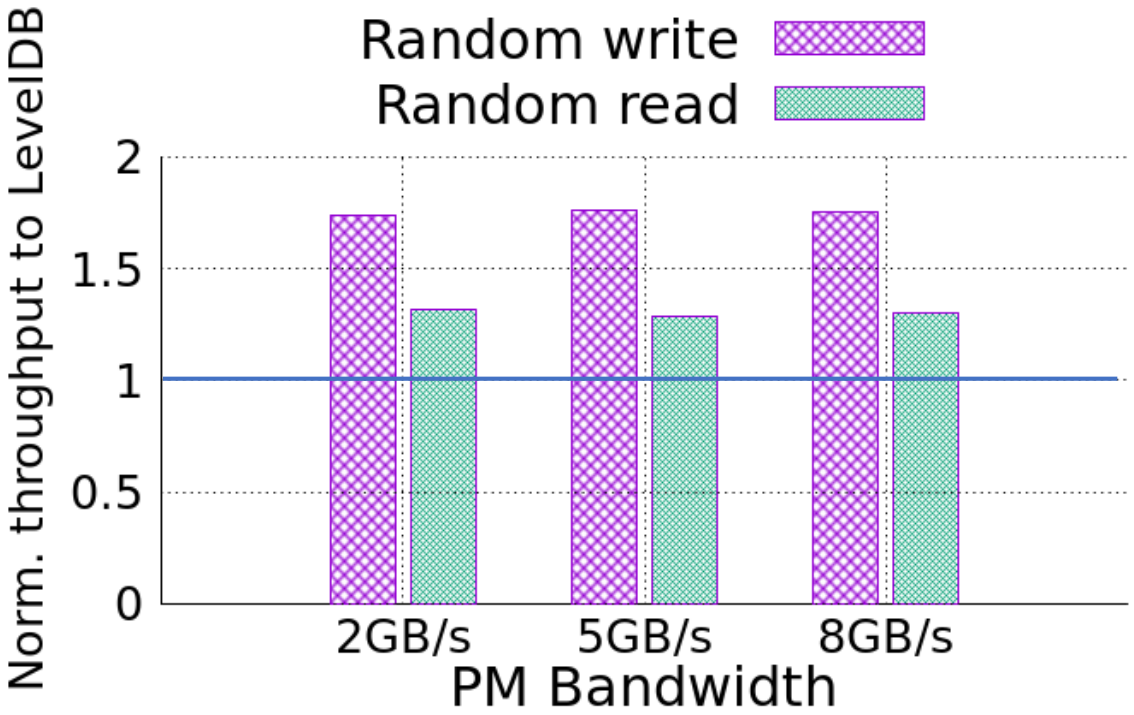
PM write latency sensitivity

Random write benchmark



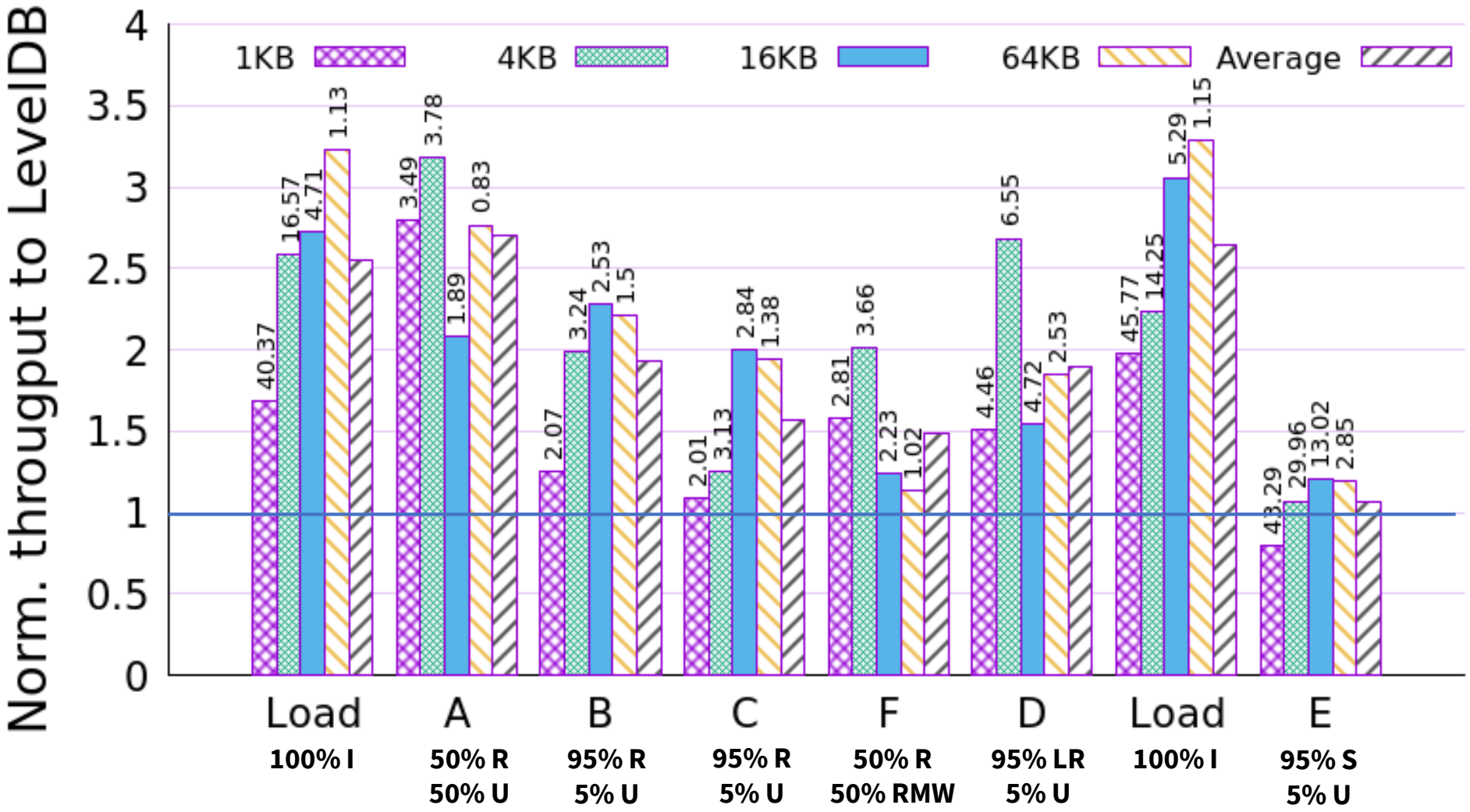
Emulated by inserting cpu pause after clfush()

PM bandwidth sensitivity

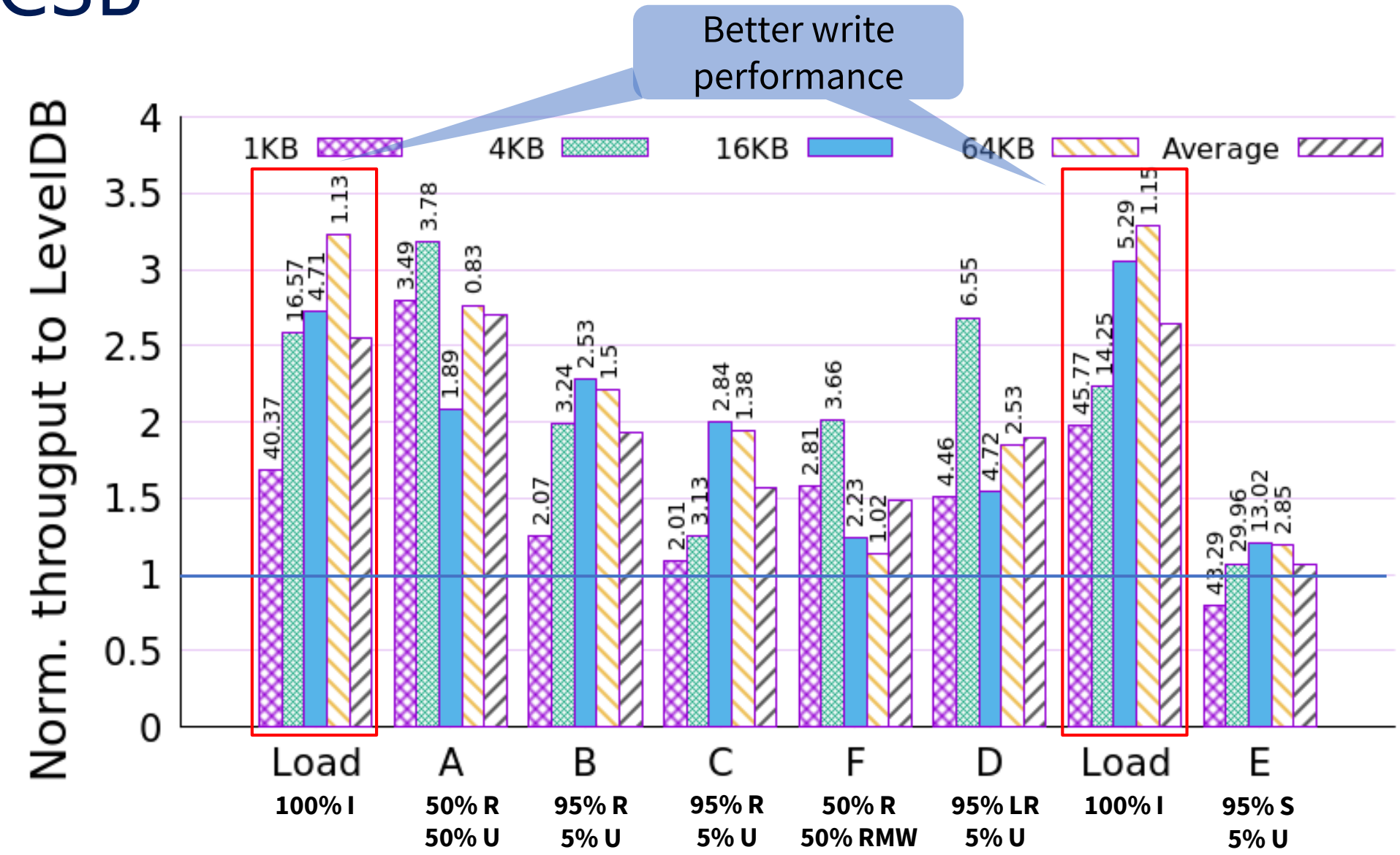


Emulated using Thermal Throttling

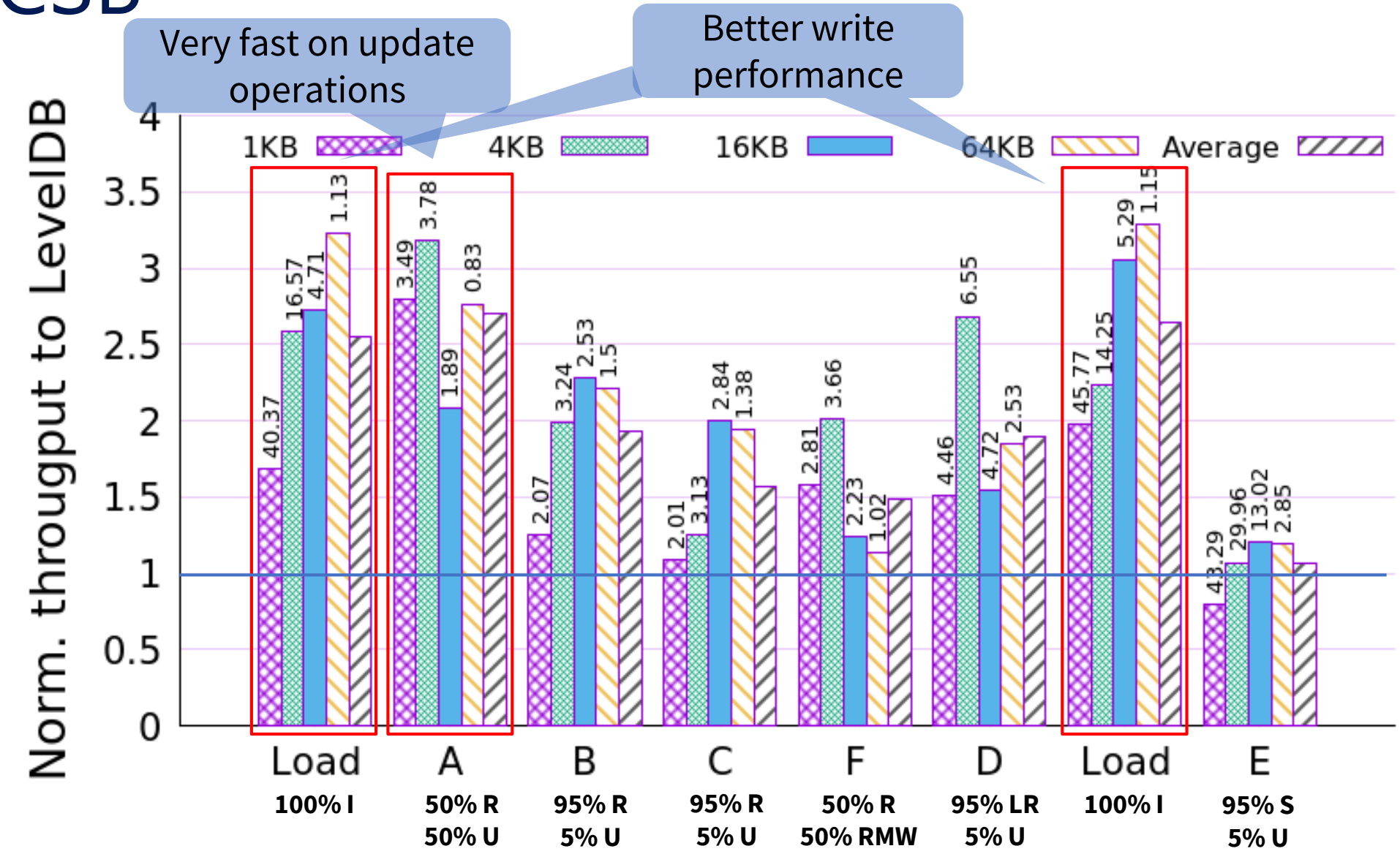
YCSB


Norm. througput to LevelDB
4
3.5
3
2.5
2
1.5
1
0.5
0
1KB
4KB
16KB
64KB
Average
40.37
16.57
4.71
1.13
3.49
3.78
1.89
0.83
2.07
3.24
2.53
1.5
2.01
3.13
2.84
1.38
2.81
3.66
2.23
1.02
4.46
6.55
4.72
2.53
45.77
14.25
5.29
1.15
43.29
29.96
13.02
2.85
Load
100% I
A
50% R
50% U
B
95% R
5% U
C
95% R
5% U
F
50% R
50% RMW
D
95% LR
5% U
Load
100% I
E
95% S
5% U

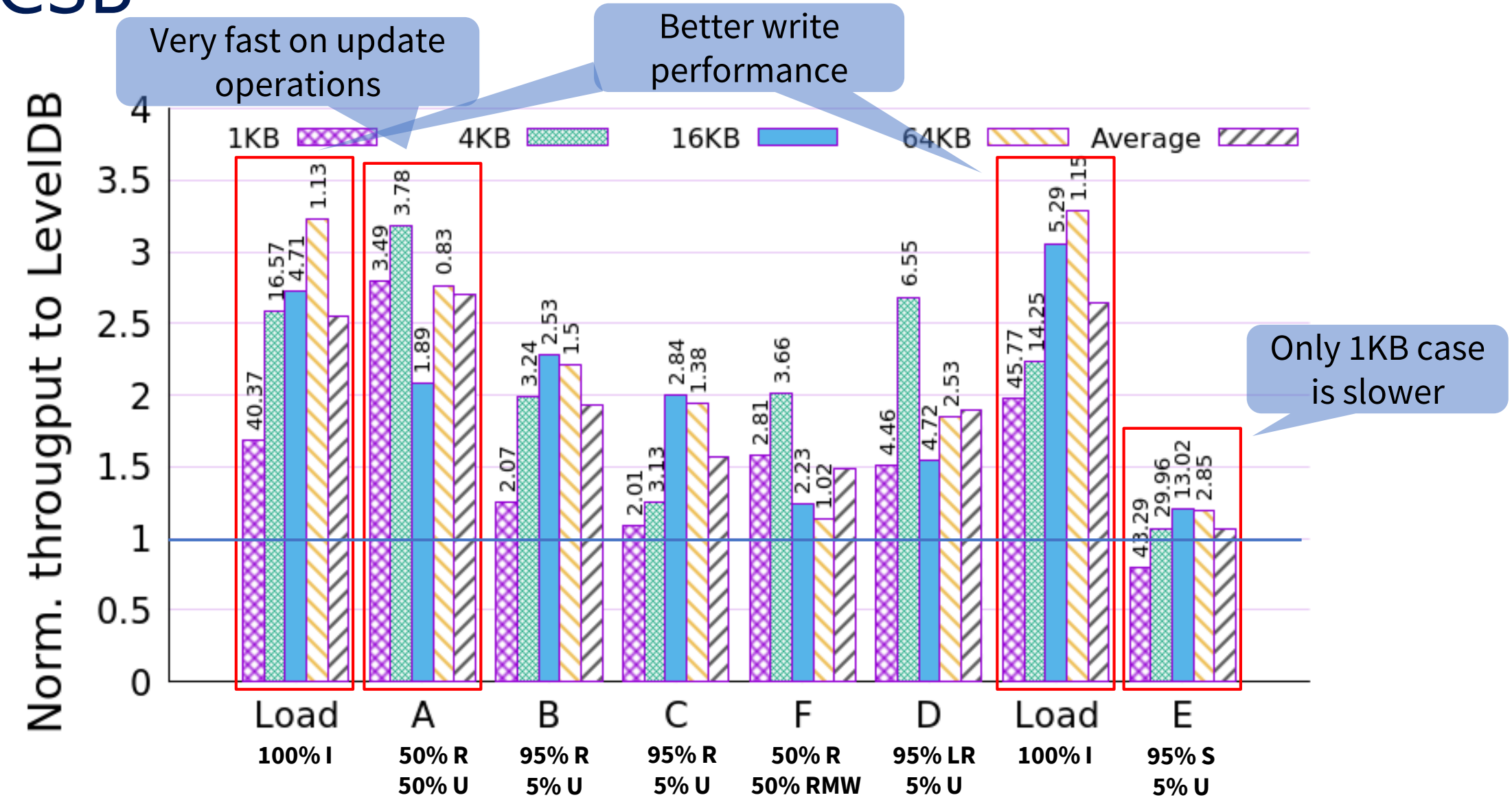
YCSB


Better write performance
Norm. througput to LevelDB
4
3.5
3
2.5
2
1.5
1
0.5
0
1KB
4KB
16KB
64KB
Average
40.37
16.57
4.71
1.13
3.49
3.78
1.89
0.83
2.07
3.24
2.53
1.5
2.01
3.13
2.84
1.38
2.81
3.66
2.23
1.02
4.46
6.55
4.72
2.53
45.77
14.25
5.29
1.15
43.29
29.96
13.02
2.85
Load
100% I
A
50% R
50% U
B
95% R
5% U
C
95% R
5% U
F
50% R
50% RMW
D
95% LR
5% U
Load
100% I
E
95% S
5% U

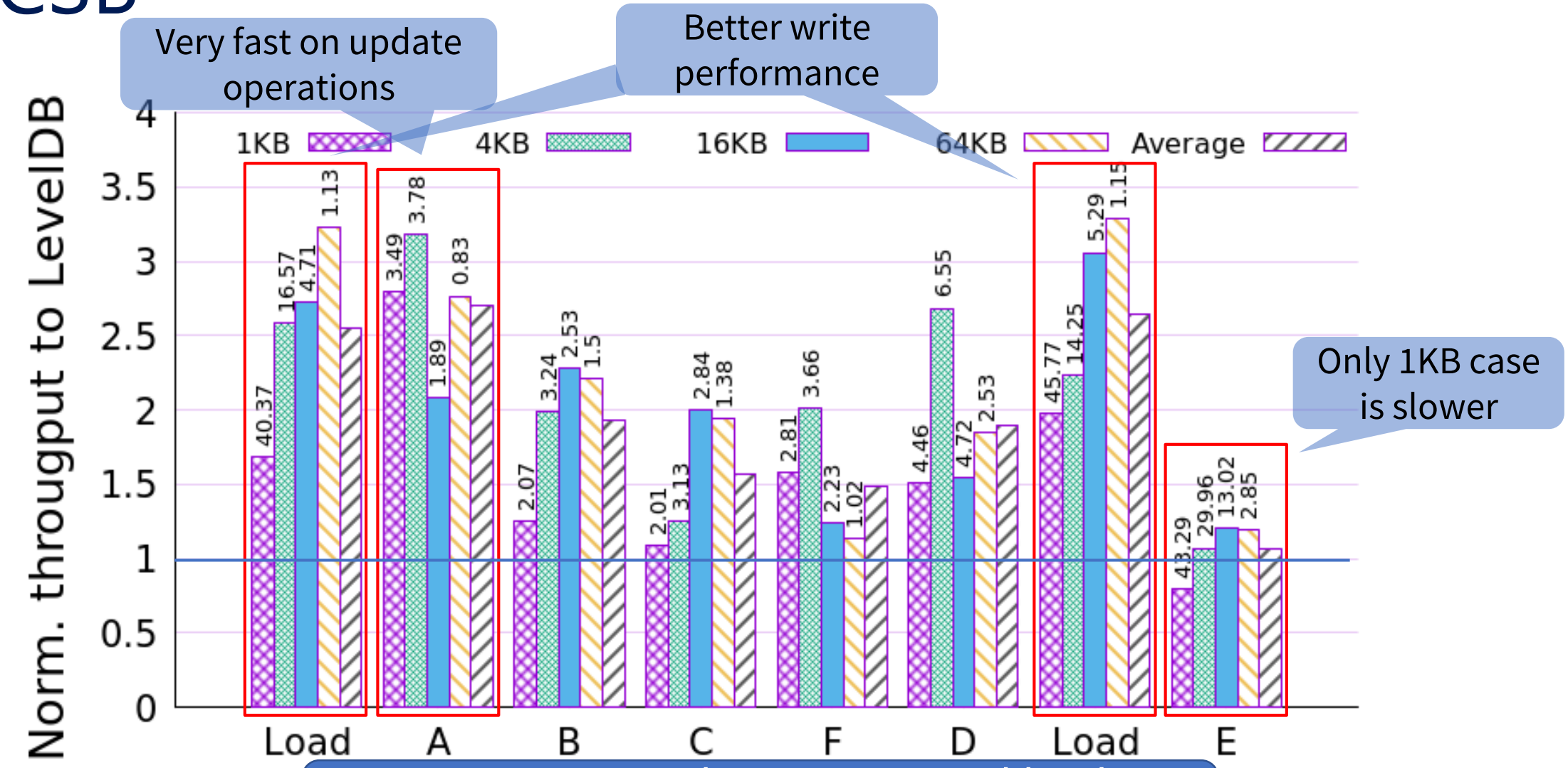
YCSB


Very fast on update operations
Better write performance
Norm. througput to LevelDB
1KB
4KB
16KB
64KB
Average
4
3.5
3
2.5
2
1.5
1
0.5
0
40.37
16.57
4.71
1.13
3.49
3.78
1.89
0.83
2.07
3.24
2.53
1.5
2.01
3.13
2.84
1.38
2.81
3.66
2.23
1.02
4.46
6.55
4.72
2.53
45.77
14.25
5.29
1.15
43.29
29.96
13.02
2.85
Load
100% I
A
50% R
50% U
B
95% R
5% U
C
95% R
5% U
F
50% R
50% RMW
D
95% LR
5% U
Load
100% I
E
95% S
5% U

YCSB


Very fast on update operations
Better write performance
Only 1KB case is slower
Norm. througput to LevelDB
1KB
4KB
16KB
64KB
Average
0
0.5
1
1.5
2
2.5
3
3.5
4
40.37
16.57
4.71
1.13
3.49
3.78
1.89
0.83
2.07
3.24
2.53
1.5
2.01
3.13
2.84
1.38
2.81
3.66
2.23
1.02
4.46
6.55
4.72
2.53
45.77
14.25
5.29
1.15
43.29
29.96
13.02
2.85
Load
100% I
A
50% R
50% U
B
95% R
5% U
C
95% R
5% U
F
50% R
50% RMW
D
95% LR
5% U
Load
100% I
E
95% S
5% U

YCSB



- On average, beats every workload
- Up to **7.7x** less disk write amplification

Conclusion

- Novel design of Key-Value stores with Persistent Memory
- High write/read performance compared to LevelDB
- Comparable scan performance
- Low write amplification
- Near-optimal read amplification

Thanks!

Questions?

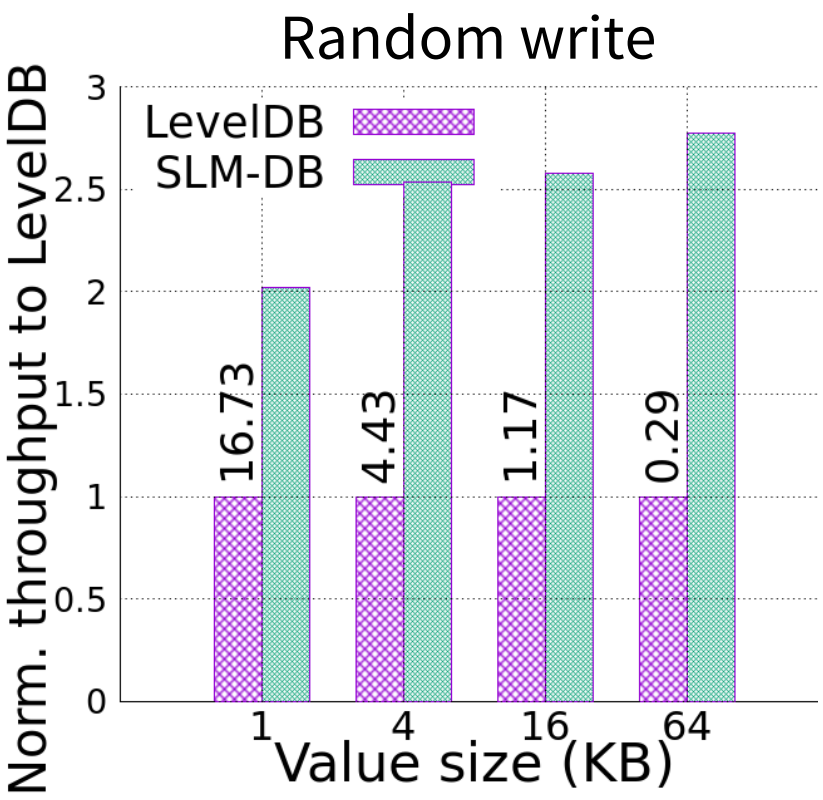
SLM-DB: Single Level Merge Key-Value Store with Persistent Memory

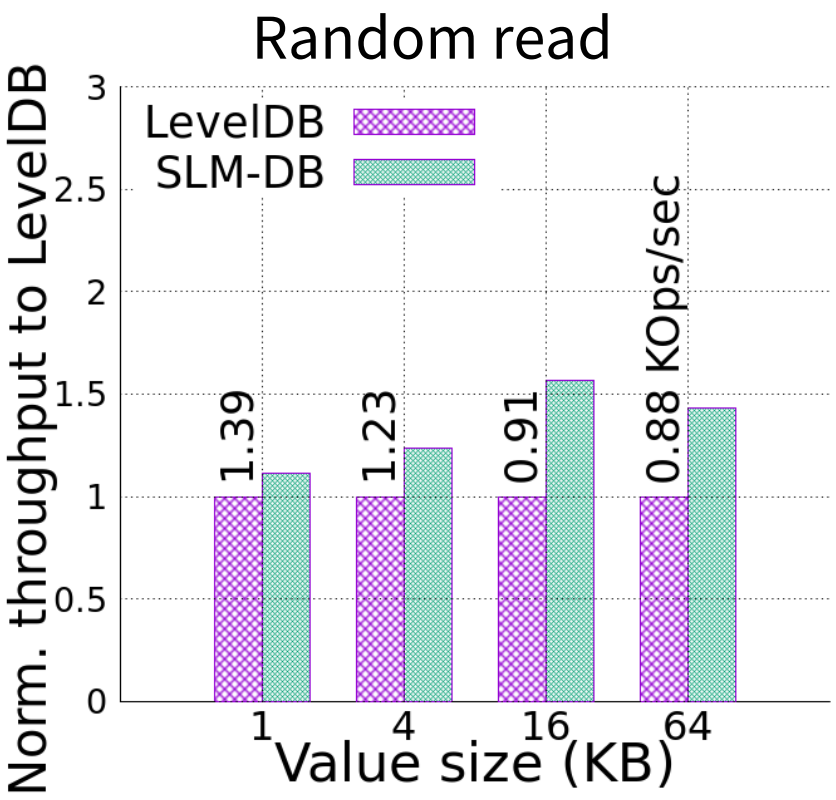
Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, Young-ri Choi

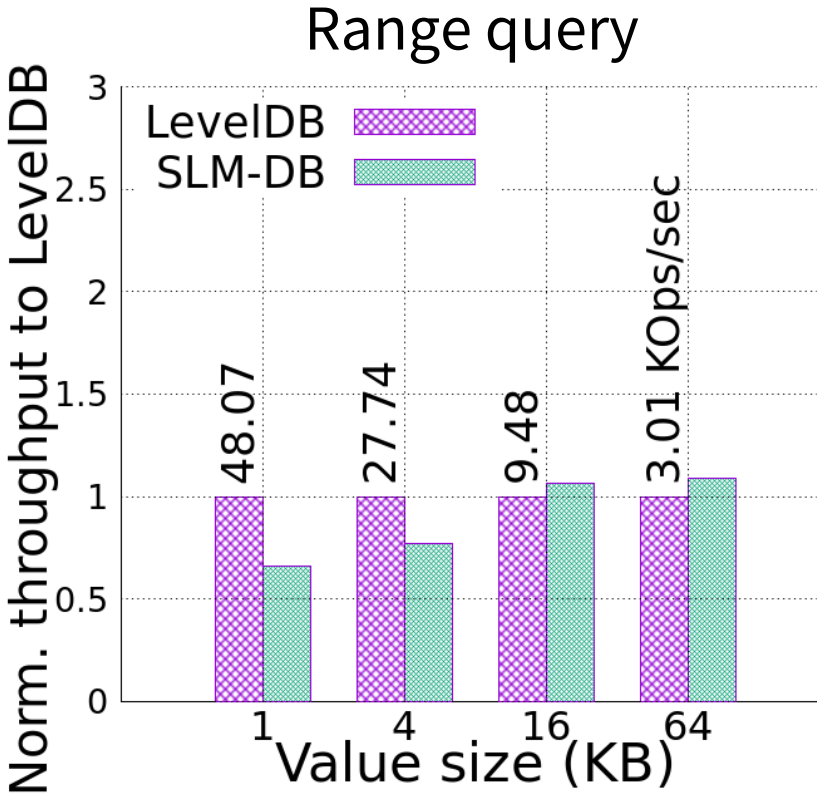
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

1398
성균관대학교
SUNG KYUN KWAN UNIVERSITY

db_bench microbenchmark (20GB)

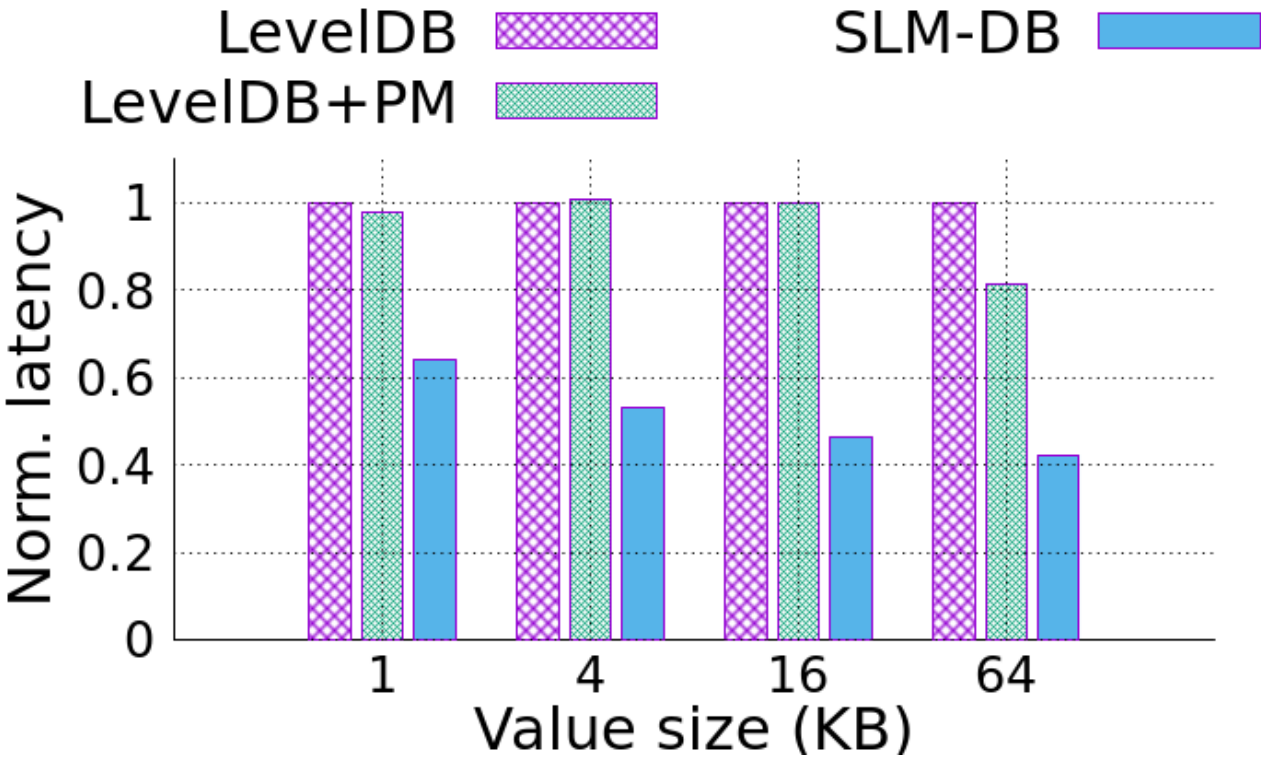




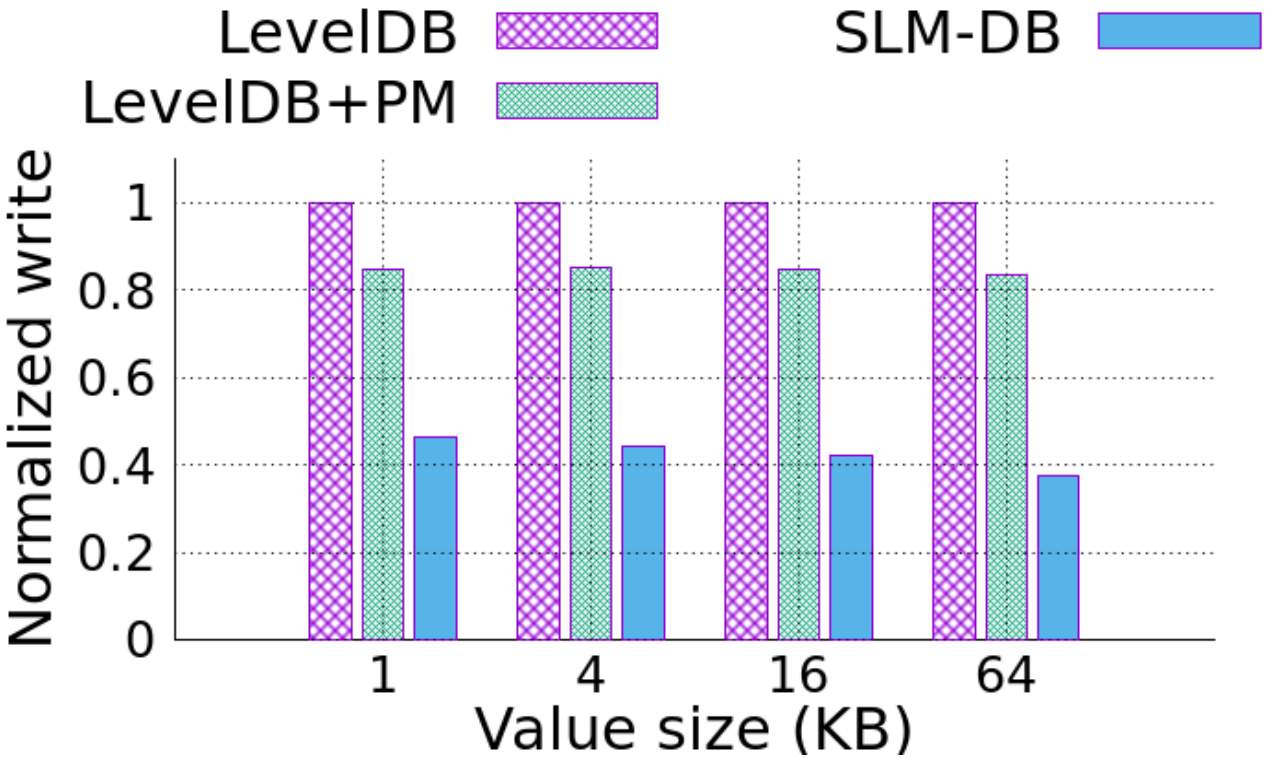


Effect of persistent MemTable

Random write performance

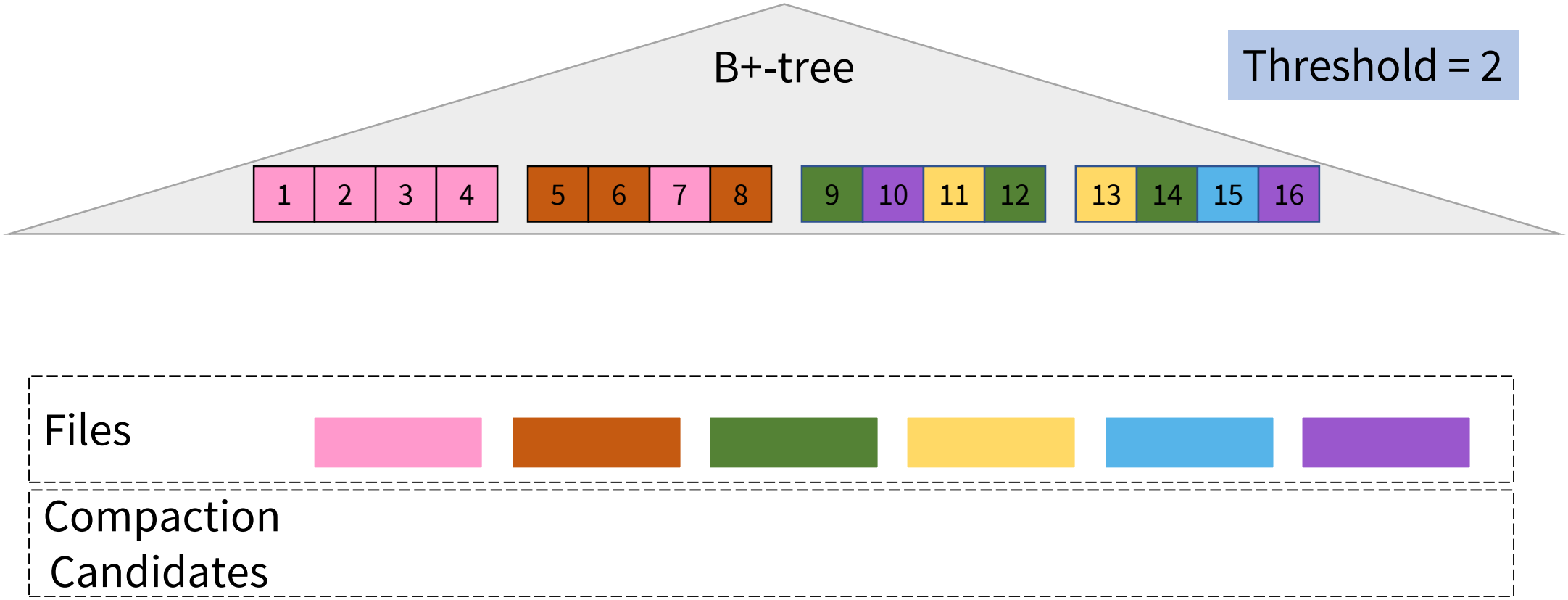


Total disk write



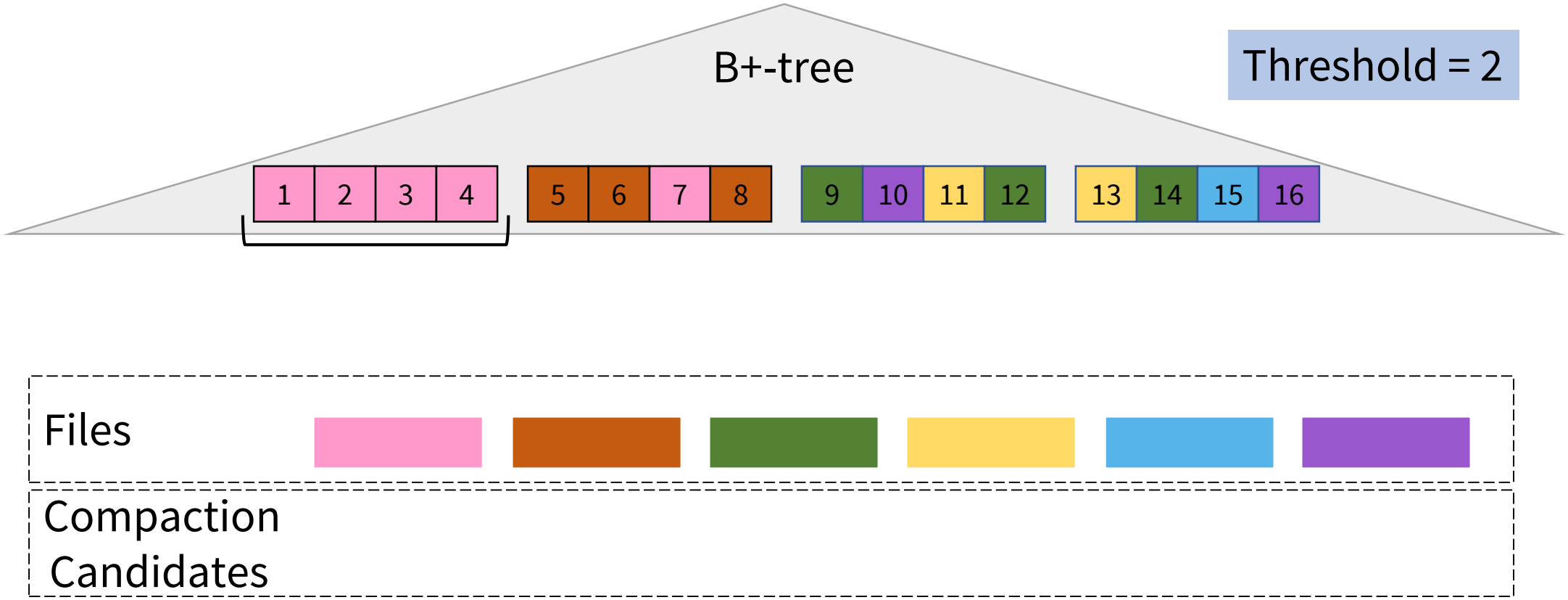
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



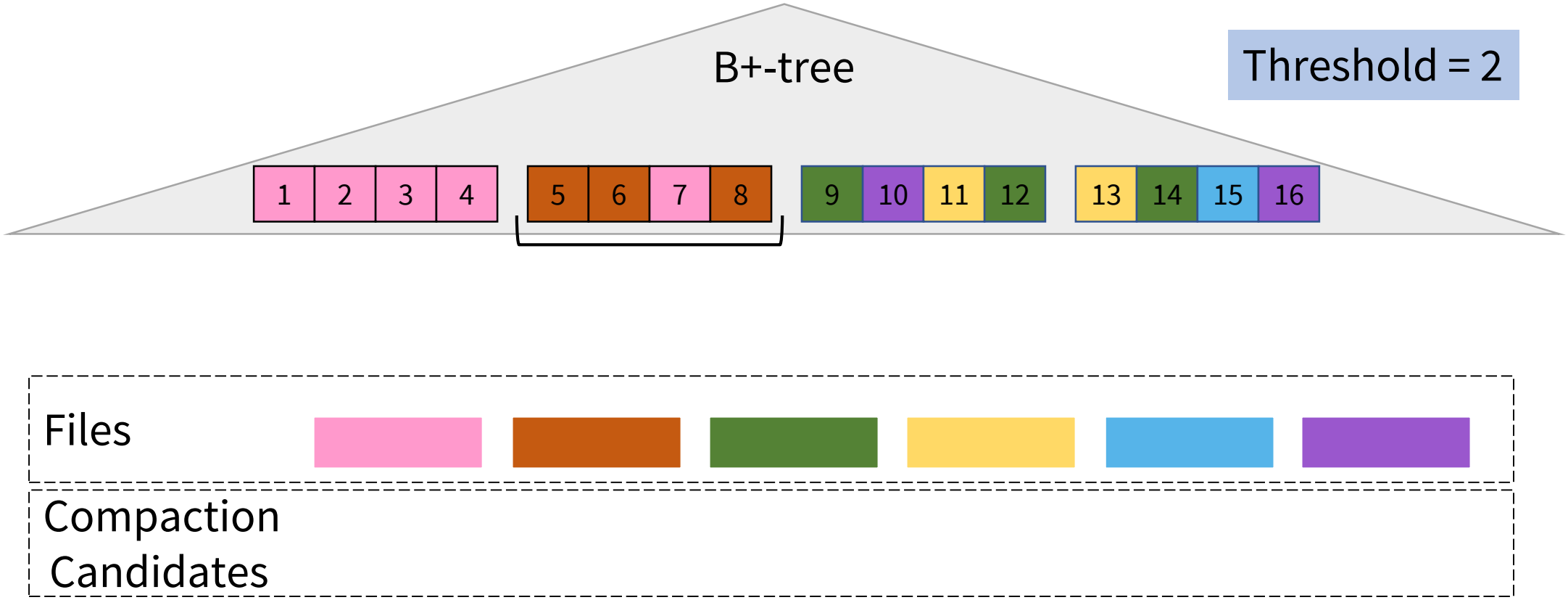
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



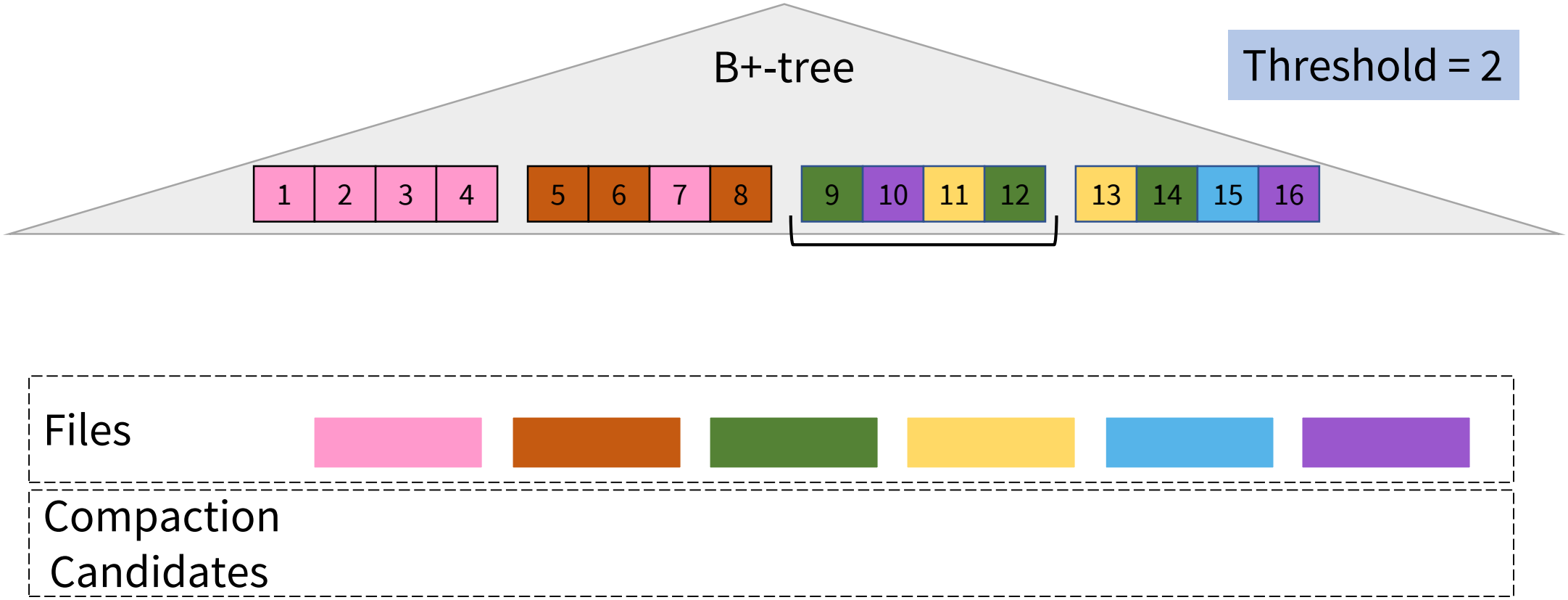
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



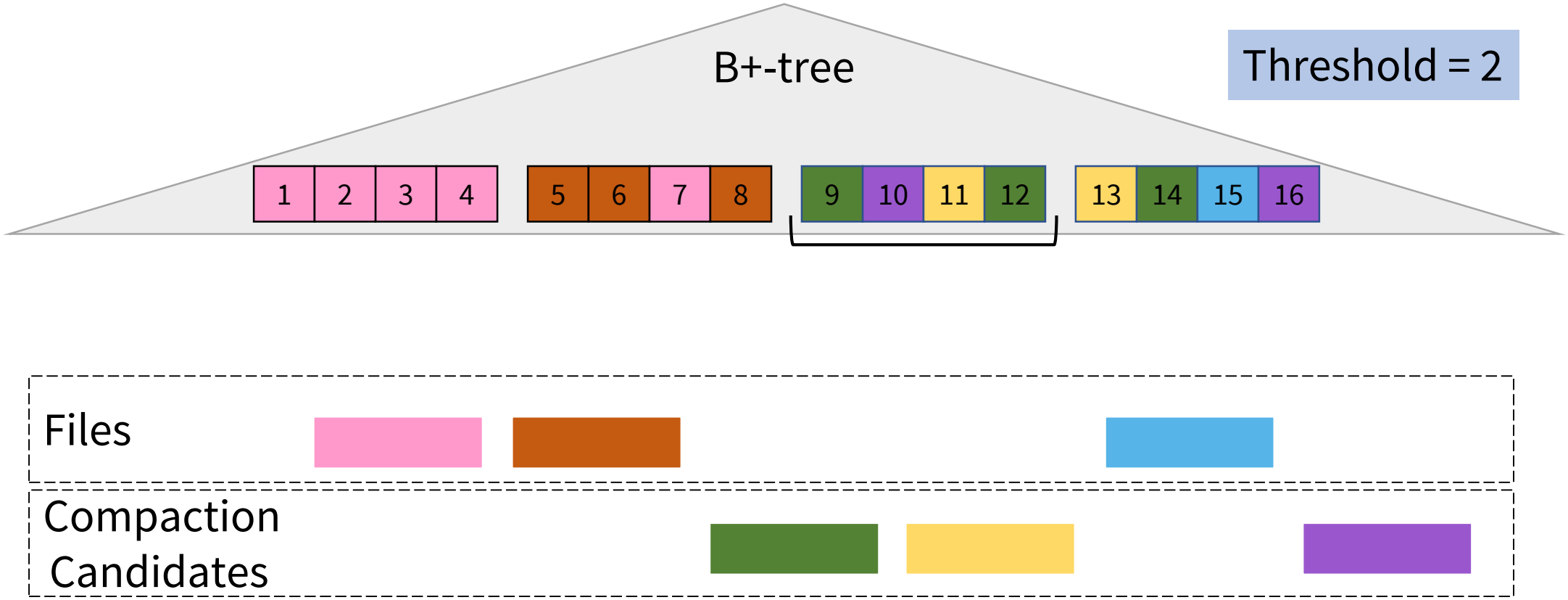
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



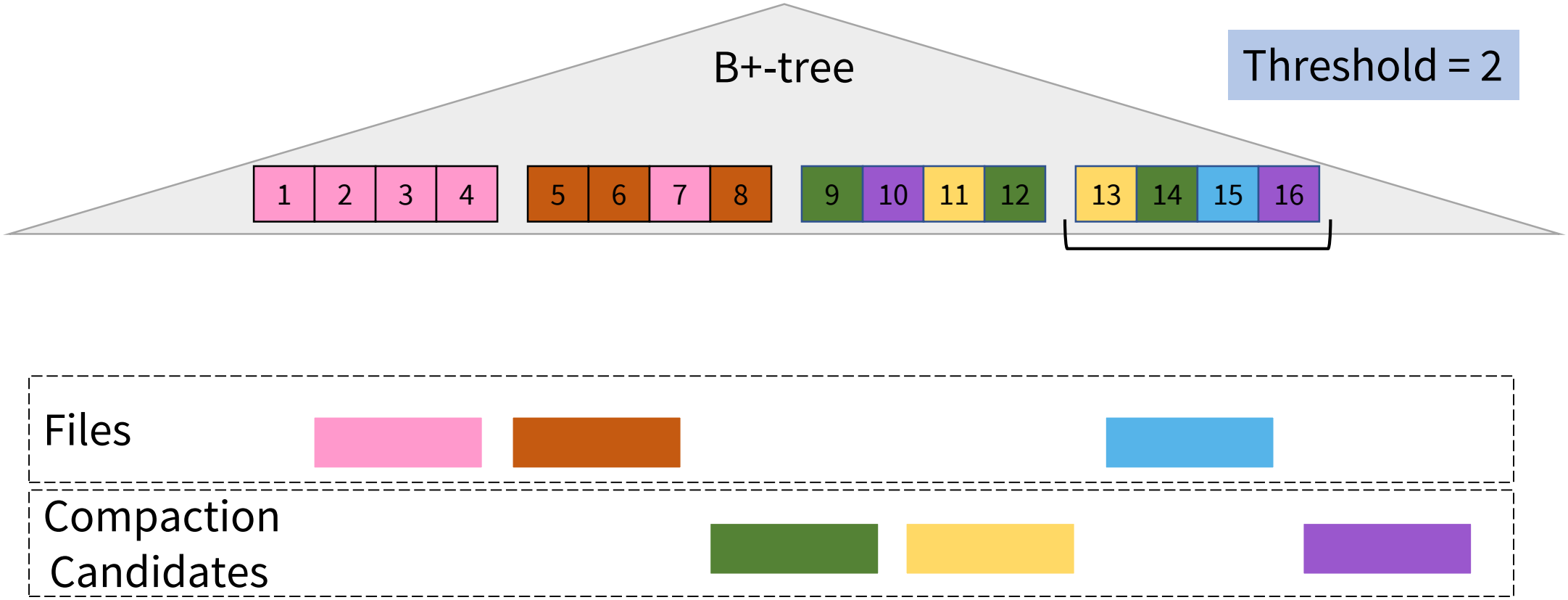
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



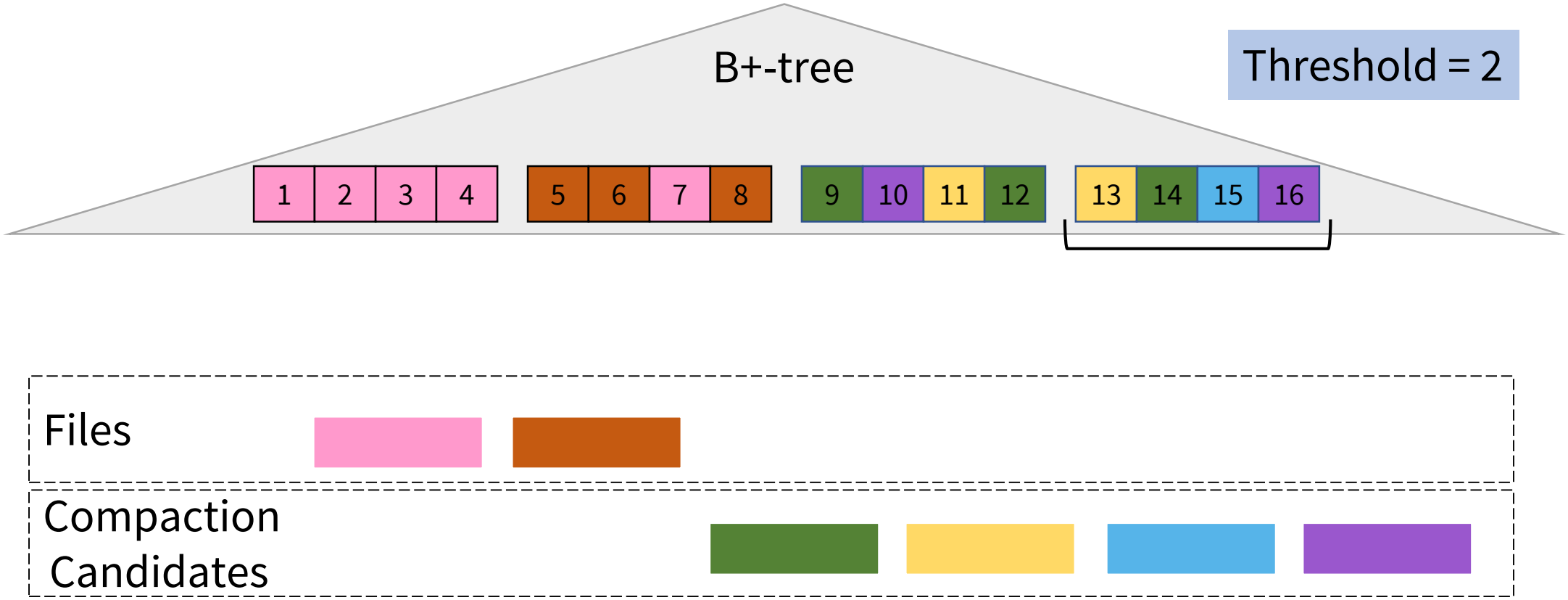
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



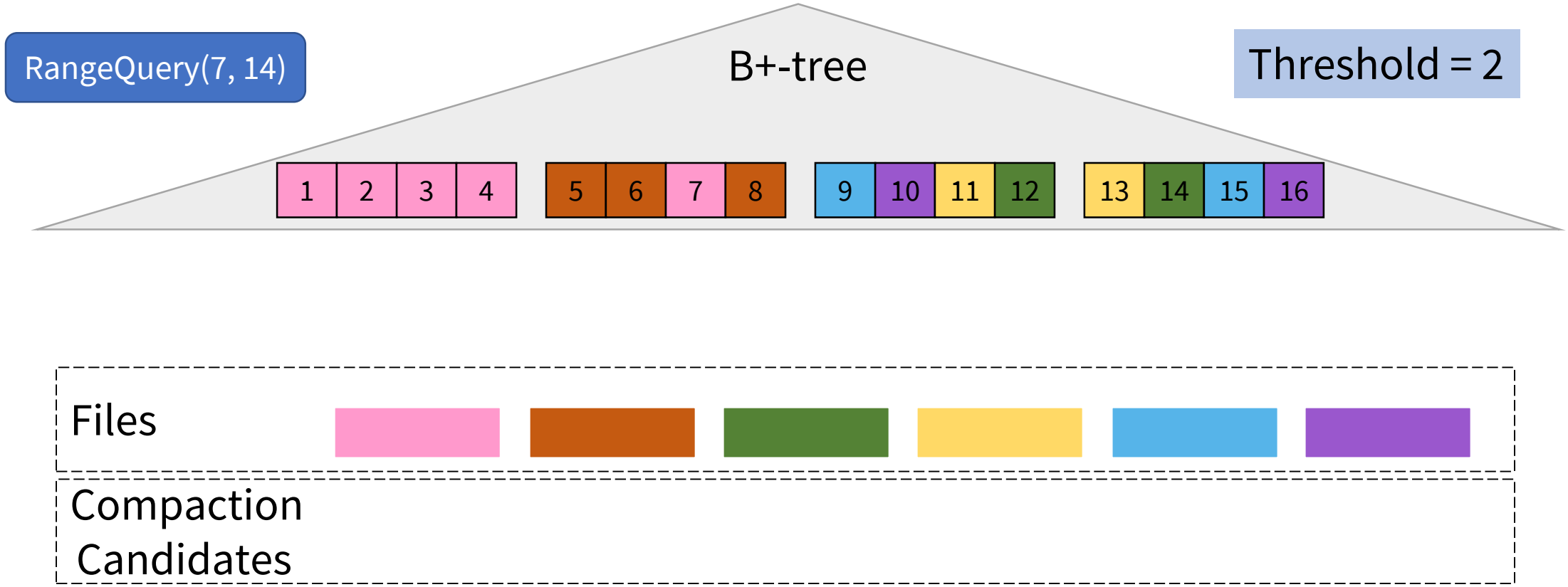
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates



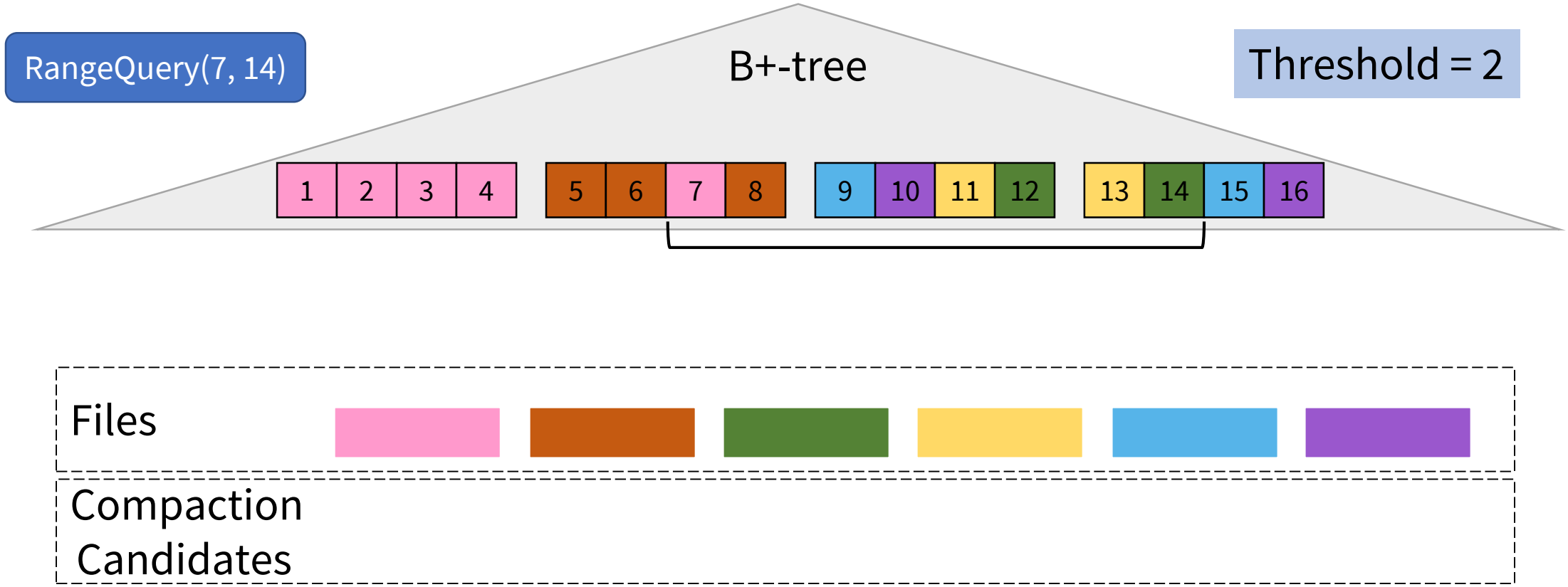
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates



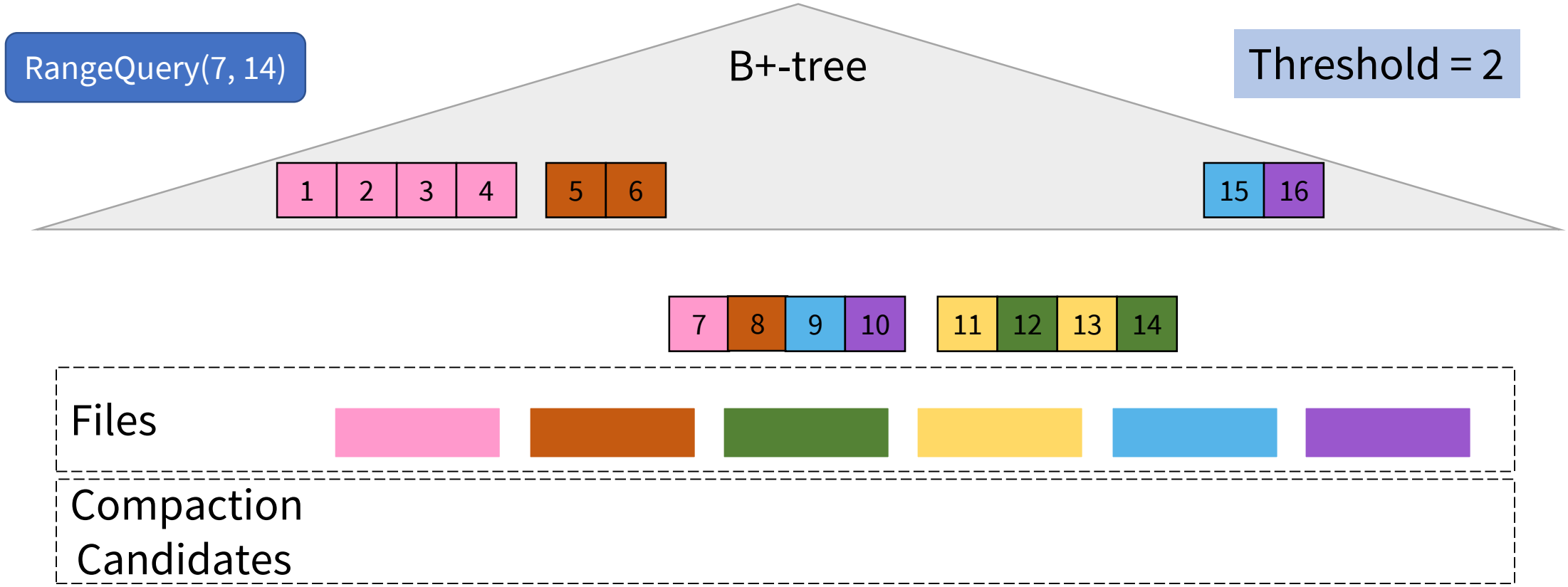
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates



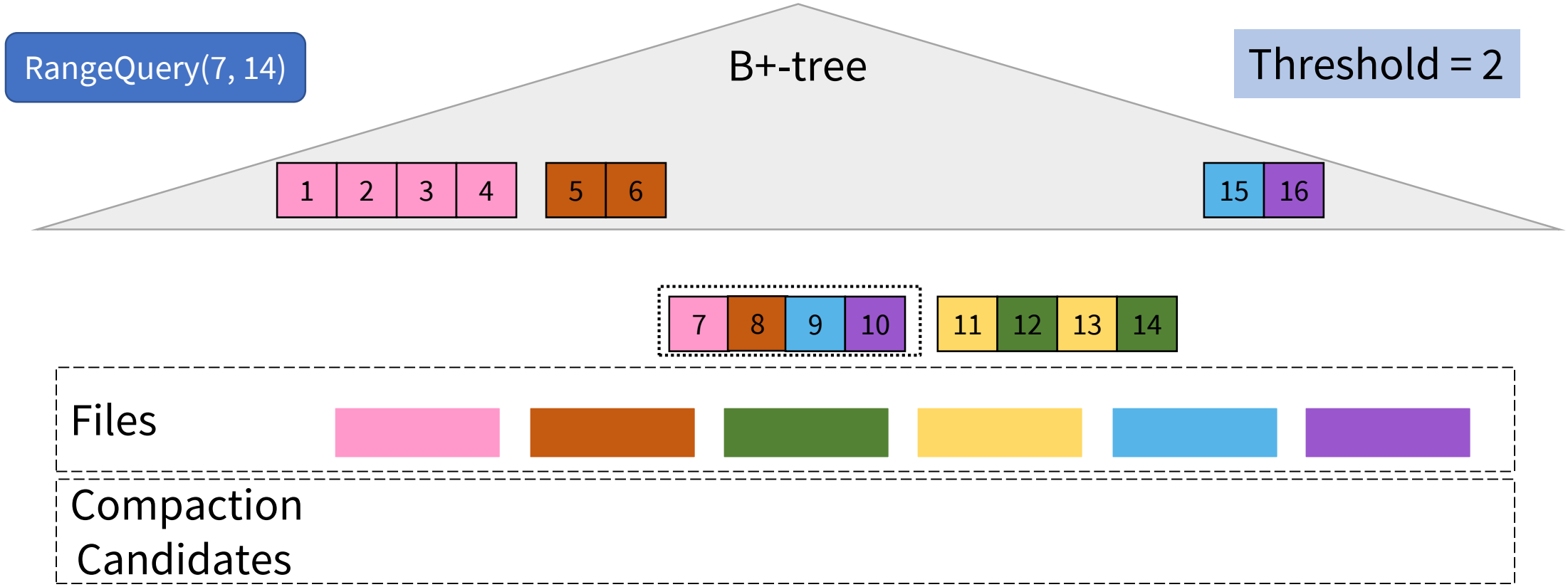
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates



Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates



Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates