PebblesDB: Building Key-Value Stores using Fragmented Log Structured Merge Trees

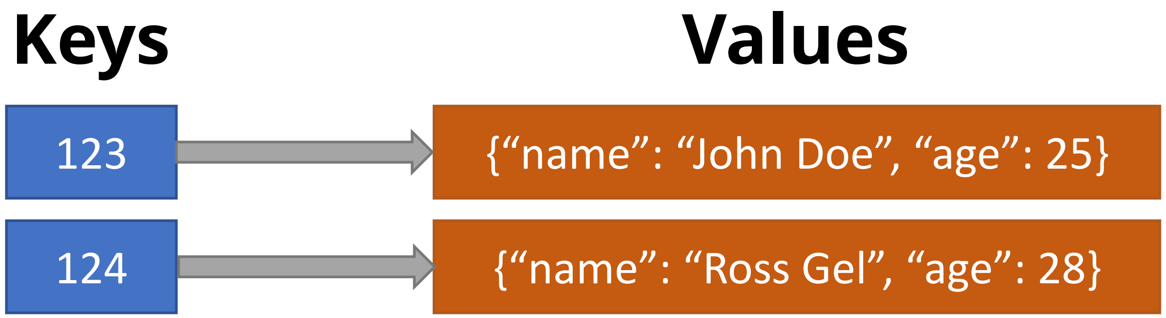
Pandian Raju[1], Rohan Kadekodi[1], Vijay Chidambaram[1,2], Ittai Abraham[2]
[1]The University of Texas at Austin
[2]VMware Research

vmware®

TEXAS
The University of Texas at Austin

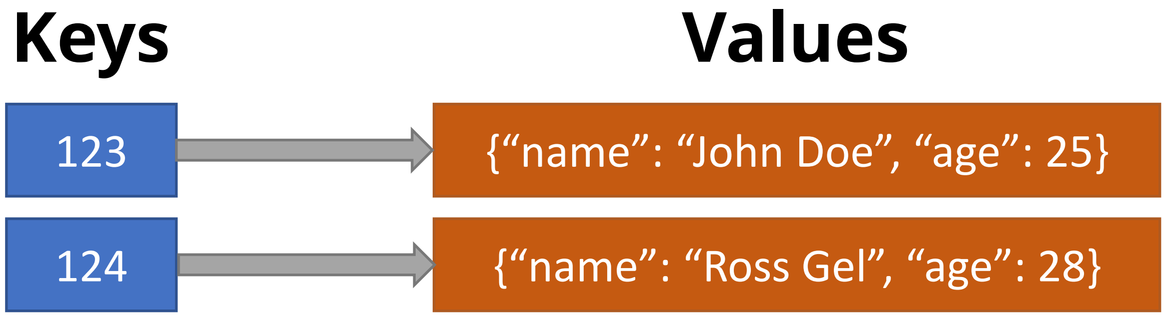
What is a key-value store?

- Store any arbitrary value for a given key



What is a key-value store?

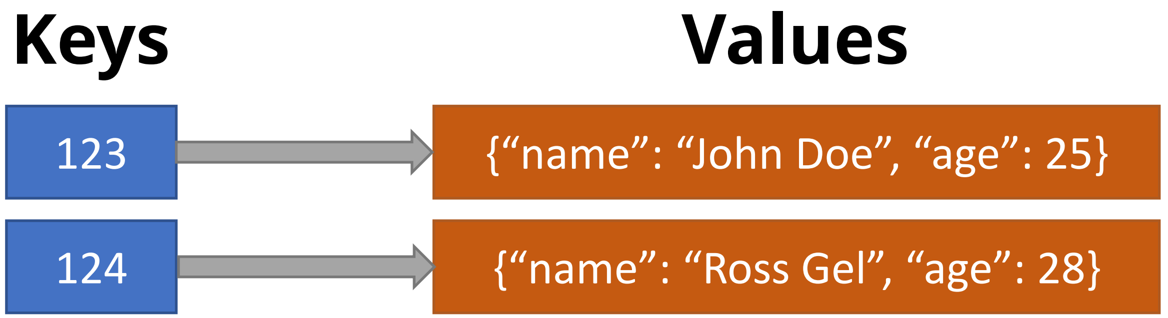
- Store any arbitrary value for a given key



- **Insertions**:
- **Point lookups**:
- **Range Queries**:

What is a key-value store?

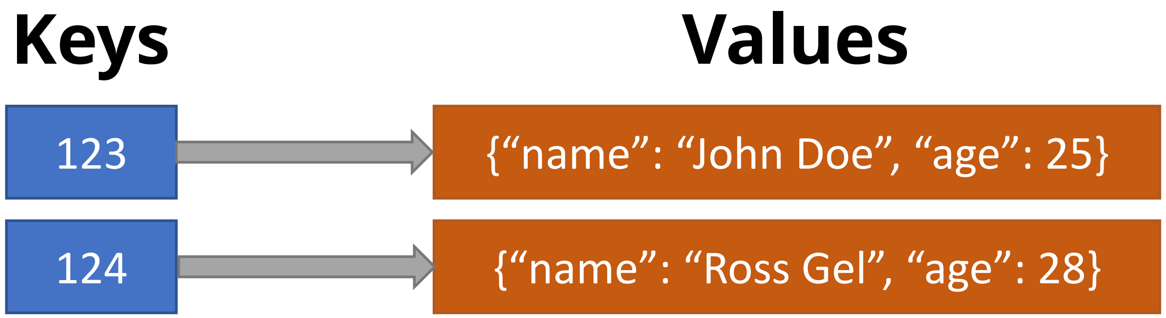
- Store any arbitrary value for a given key



- **Insertions**: put(key, value)
- **Point lookups**:
- **Range Queries**:

What is a key-value store?

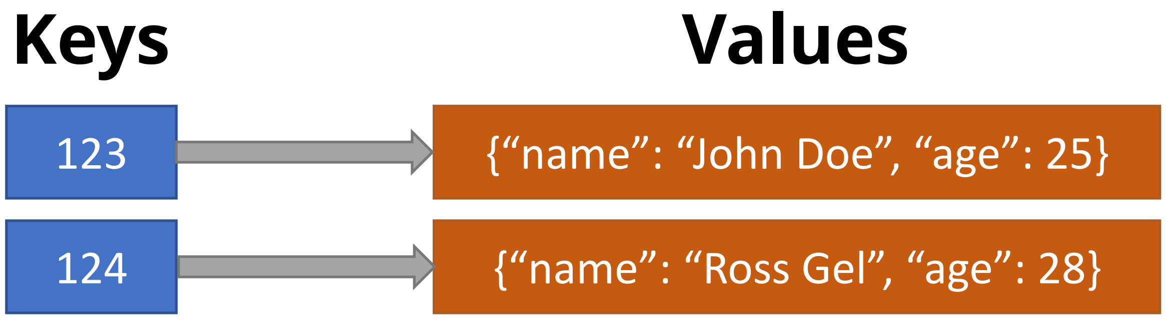
- Store any arbitrary value for a given key



- **Insertions**: put(key, value)
- **Point lookups**: get(key)
- **Range Queries**:

What is a key-value store?

- Store any arbitrary value for a given key



- **Insertions**: put(key, value)
- **Point lookups**: get(key)
- **Range Queries**: get_range(key1, key2)

Key-Value Stores - widely used

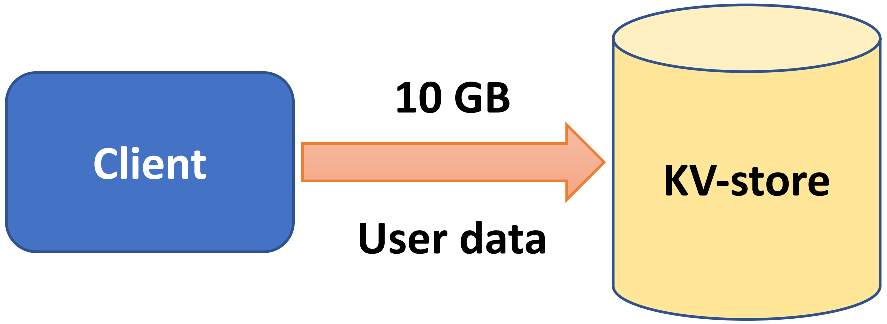
- Google's BigTable powers Search, Analytics, Maps and Gmail
- Facebook's RocksDB is used as storage engine in production systems of many companies

Write-optimized data structures

- Log Structured Merge Tree (LSM) is a write-optimized data structure used in key-value stores
- Provides high write throughput with good read throughput, but suffers high write amplification

Write-optimized data structures

- Log Structured Merge Tree (LSM) is a write-optimized data structure used in key-value stores
- Provides high write throughput with good read throughput, but suffers high write amplification
- Write amplification - Ratio of amount of write IO to amount of user data

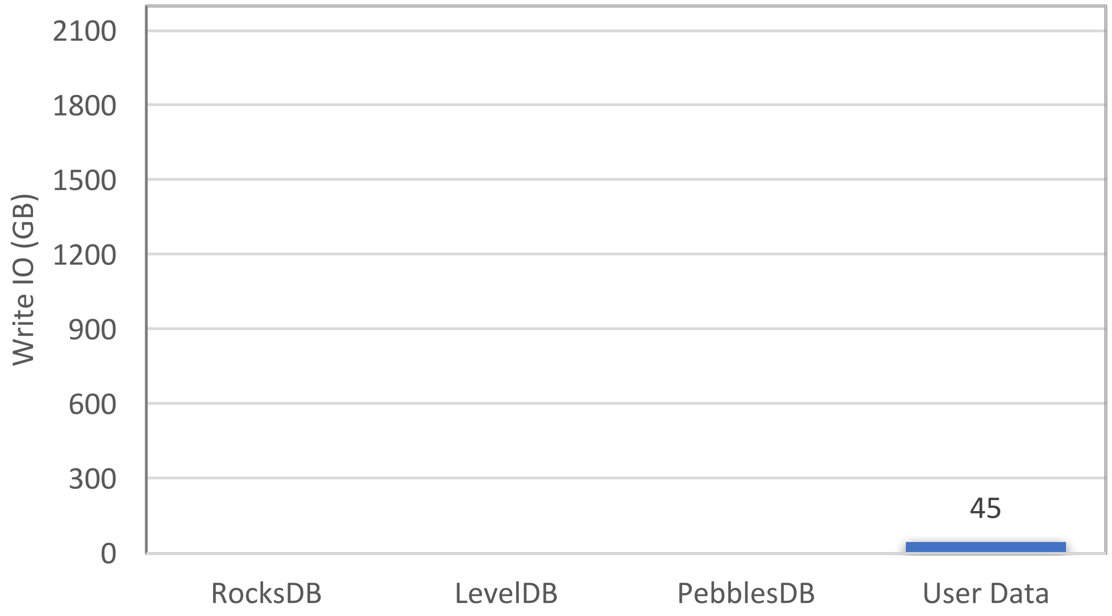


If total write I/O is 200 GB

Write amplification = 20

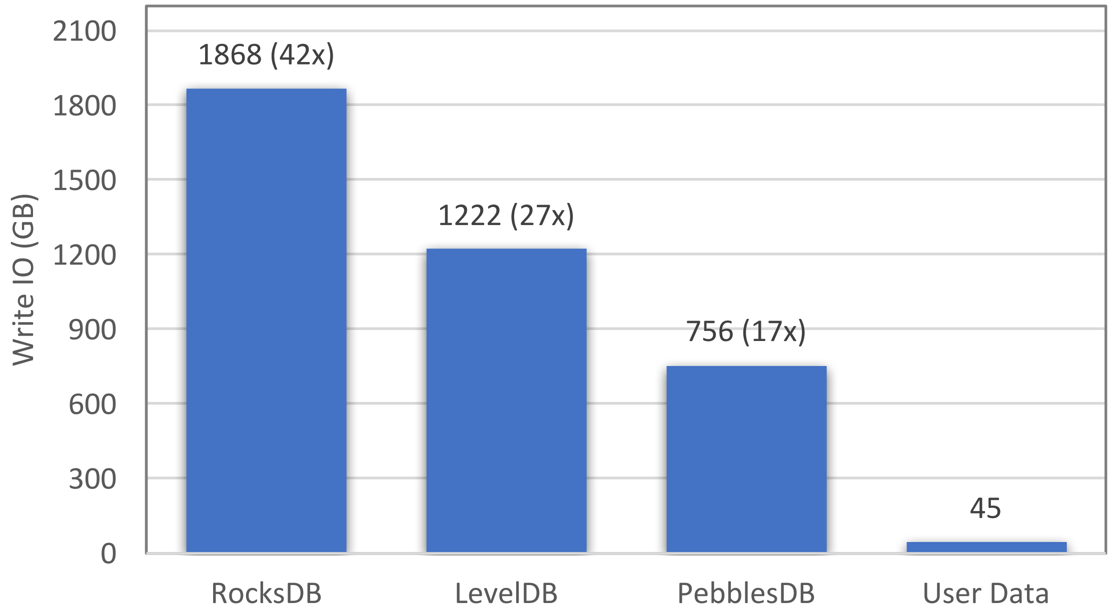
Write amplification in LSM based KV stores

- Inserted 500M key-value pairs
- Key: 16 bytes, Value: 128 bytes
- Total user data: ~45 GB



Write amplification in LSM based KV stores

- Inserted 500M key-value pairs
- Key: 16 bytes, Value: 128 bytes
- Total user data: ~45 GB



Why is write amplification bad?

- Reduces the write throughput
- Flash devices wear out after limited write cycles

(Intel SSD DC P4600 – can last ~5 years assuming ~5 TB write per day)

RocksDB can write ~500 GB of user data per day to a SSD to last 1.25 years

Data source: https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4600-series/dc-p4600-1-6tb-2-5inch-3d1.html

PebblesDB

High performance write-optimized key-value store

Built using new data structure
Fragmented Log-Structured Merge Tree

Achieves 3-6.7x higher write throughput and 2.4-3x lesser write amplification compared to RocksDB

Gets the highest write throughput and least write amplification as a backend store to MongoDB

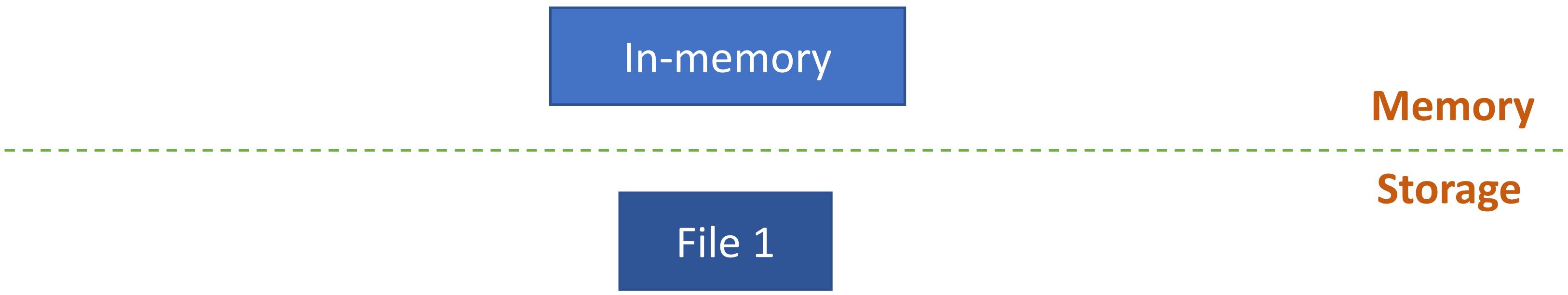
Outline

- Log-Structured Merge Tree (LSM)
- Fragmented Log-Structured Merge Tree (FLSM)
- Building PebblesDB using FLSM
- Evaluation
- Conclusion

Outline

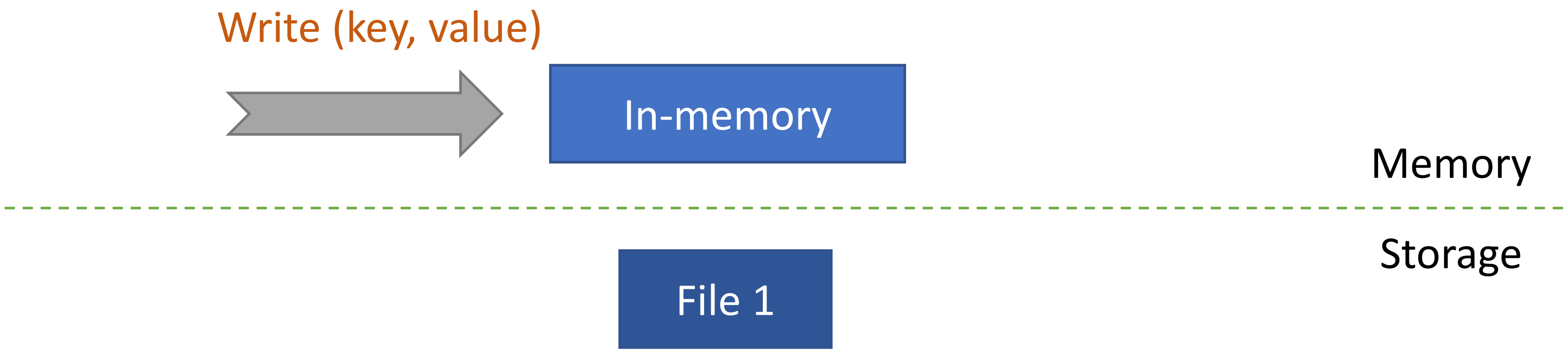
- **Log-Structured Merge Tree (LSM)**
- Fragmented Log-Structured Merge Tree (FLSM)
- Building PebblesDB using FLSM
- Evaluation
- Conclusion

Log Structured Merge Tree (LSM)



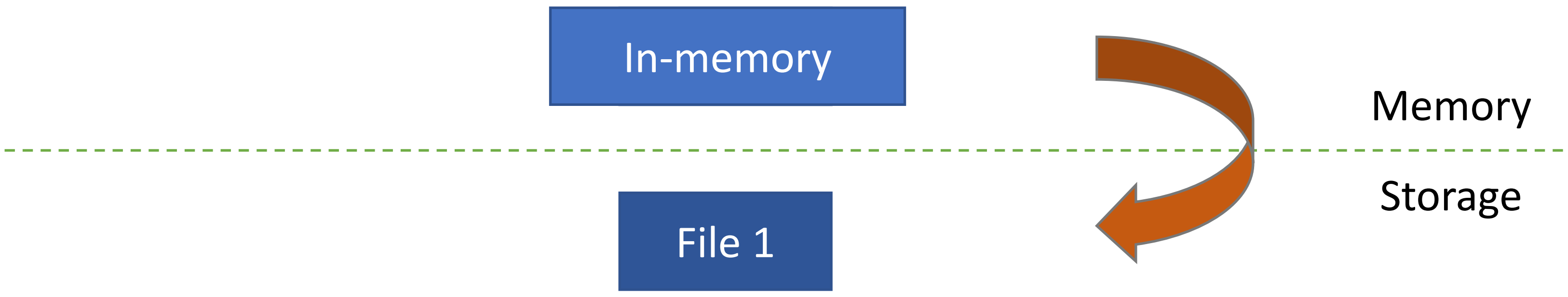
Data is stored both in memory and storage

Log Structured Merge Tree (LSM)



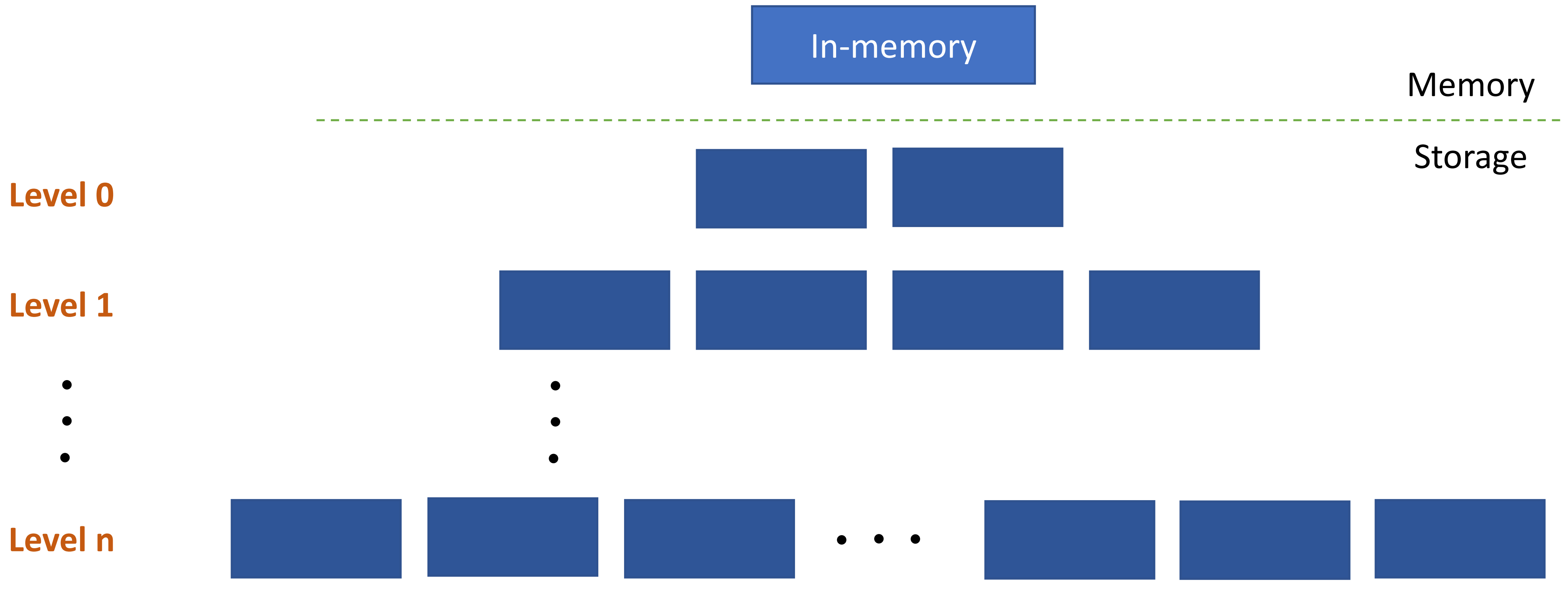
Writes are directly put to memory

Log Structured Merge Tree (LSM)



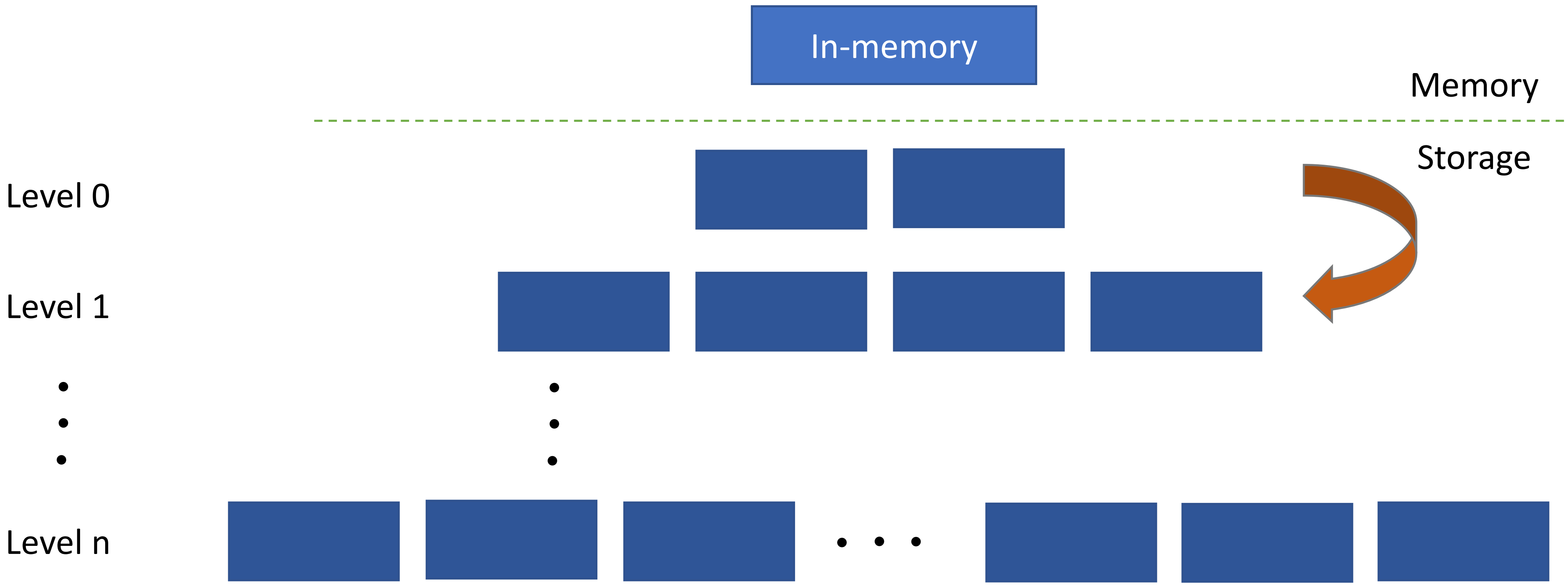
In-memory data is periodically written as files to storage (sequential I/O)

Log Structured Merge Tree (LSM)



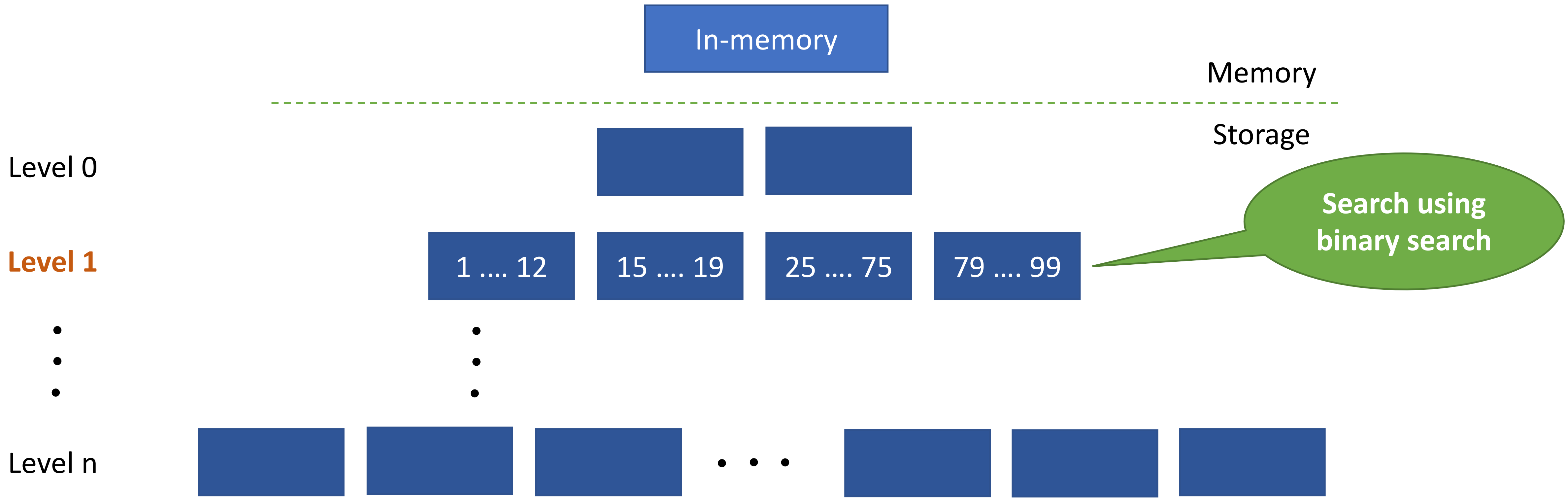
Files on storage are logically arranged in different levels

Log Structured Merge Tree (LSM)



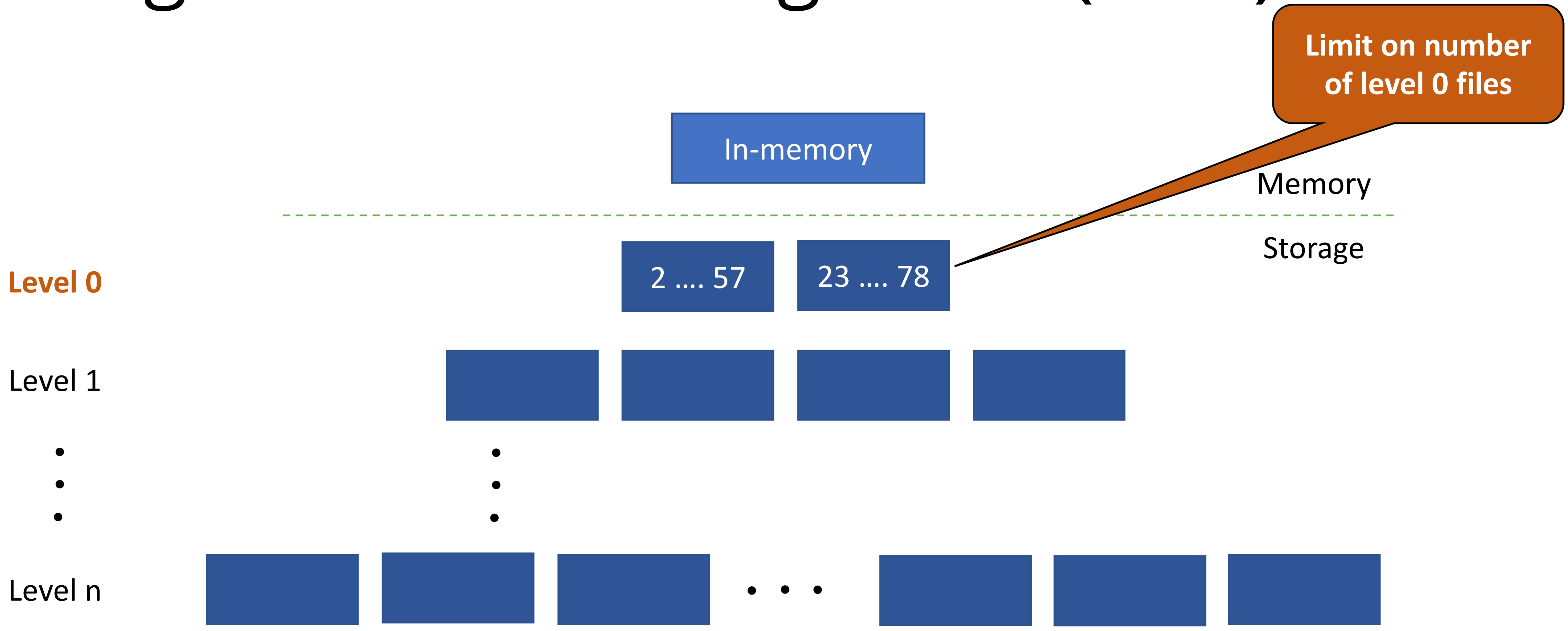
Compaction pushes data to higher numbered levels

Log Structured Merge Tree (LSM)



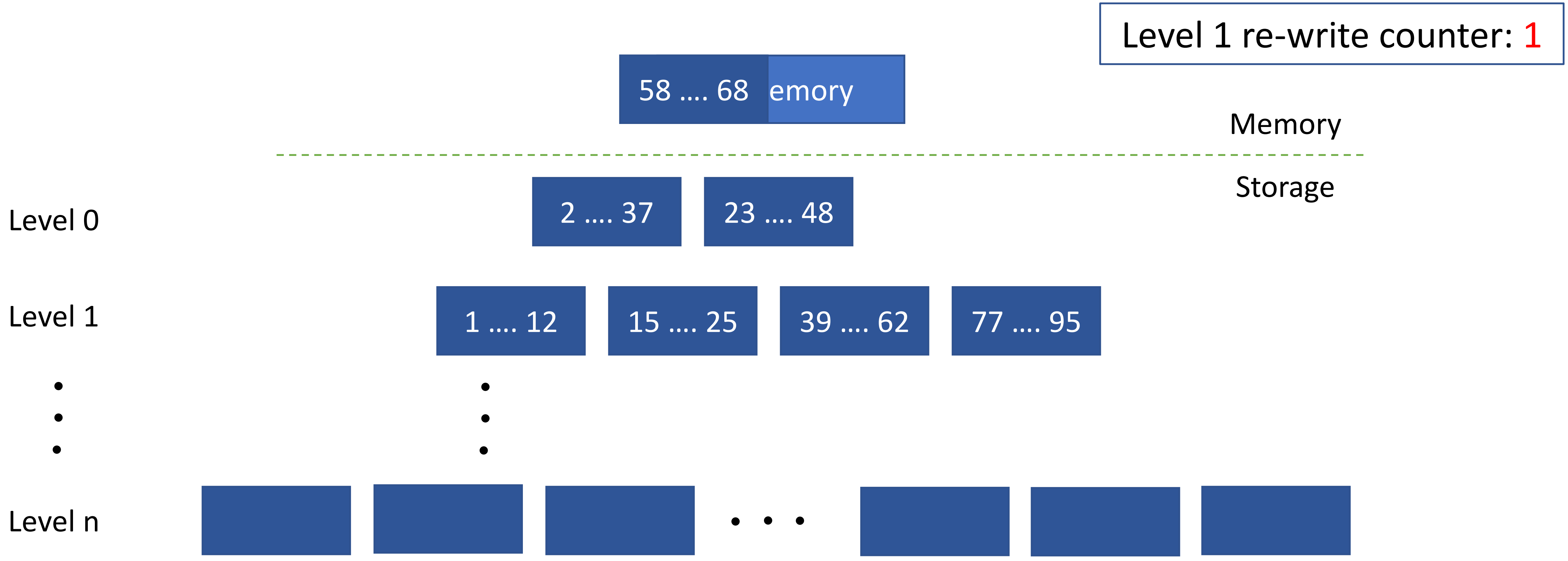
Files are sorted and have non-overlapping key ranges

Log Structured Merge Tree (LSM)



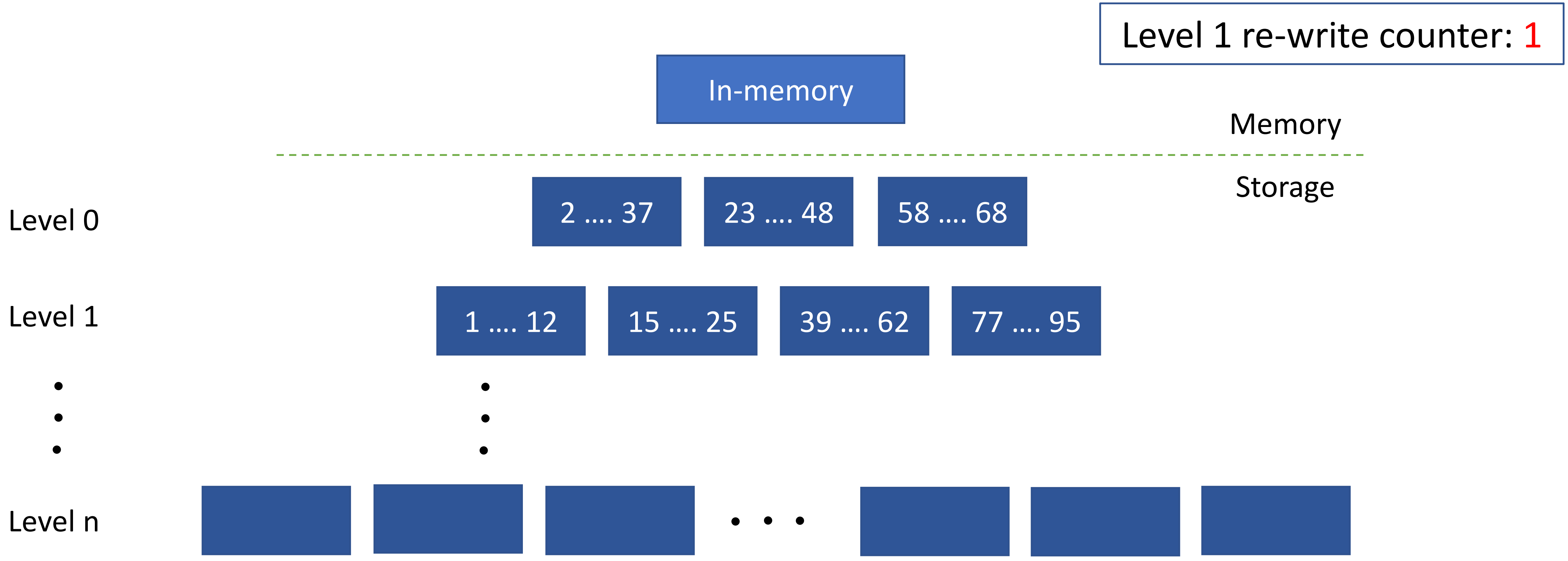
Level 0 can have files with overlapping (but sorted) key ranges

Write amplification: Illustration



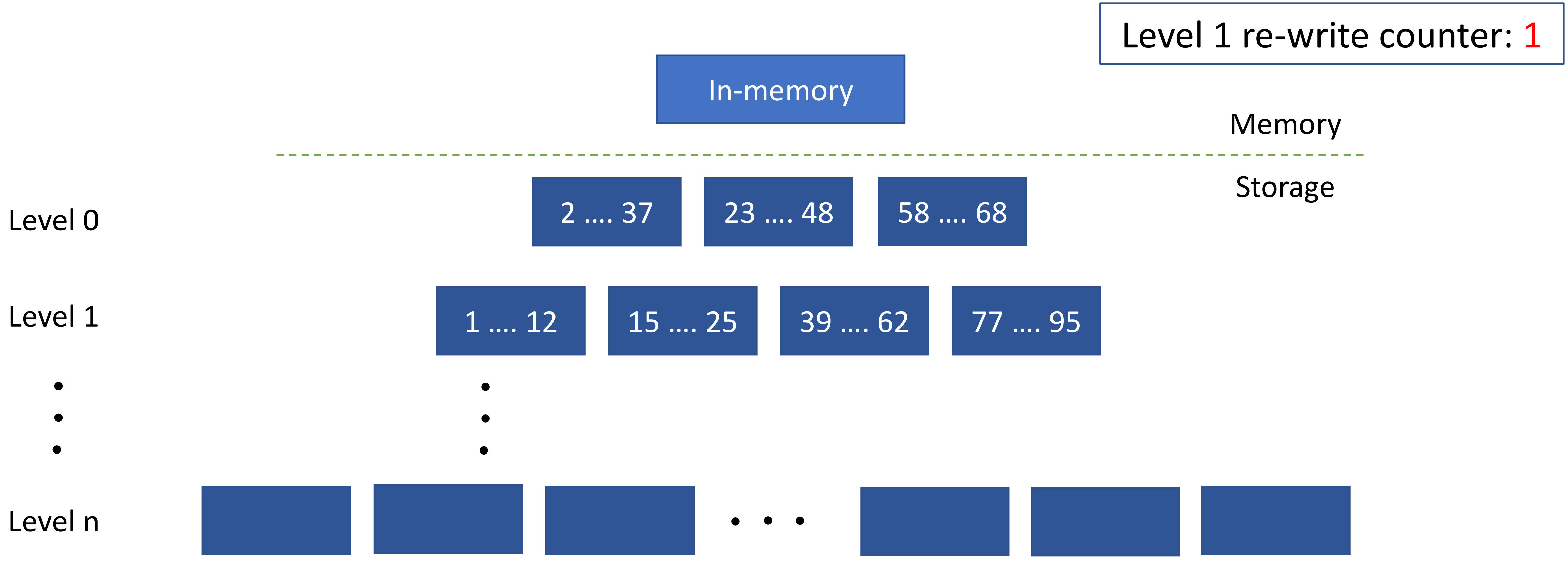
Max files in level 0 is configured to be 2

Write amplification: Illustration



Level 0 has 3 files (> 2), which triggers a compaction

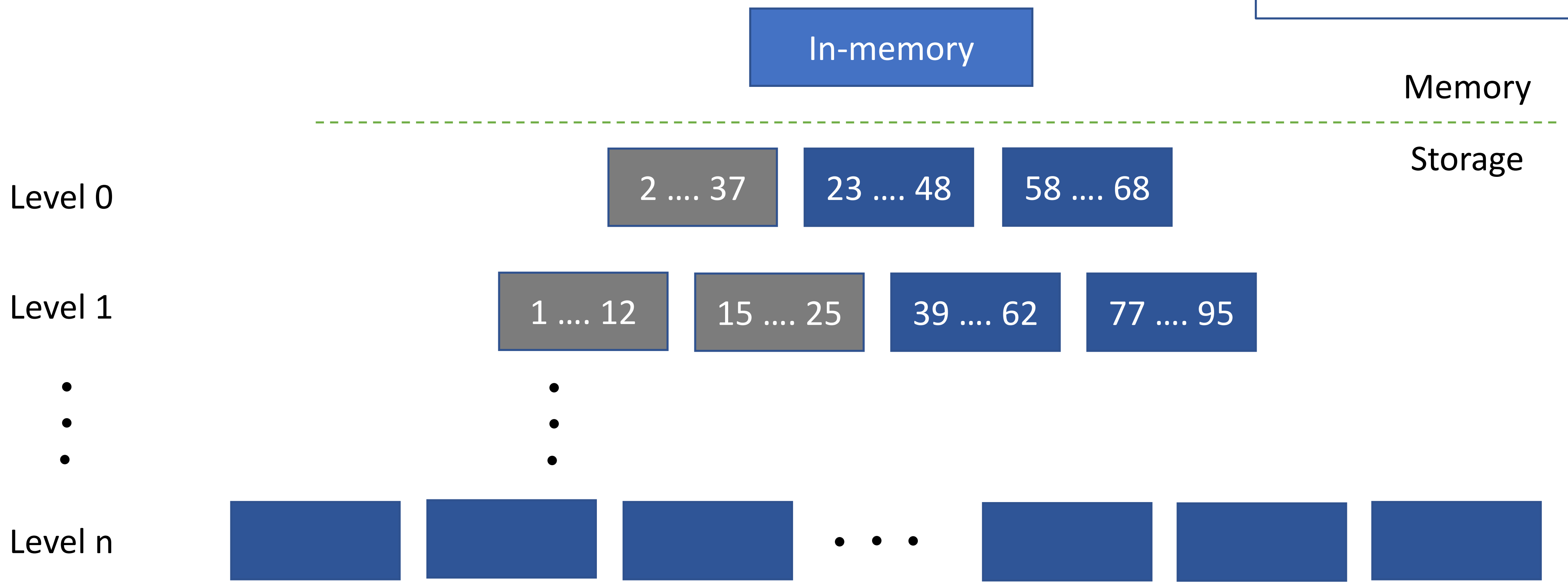
Write amplification: Illustration



* Files are immutable * Sorted non-overlapping files

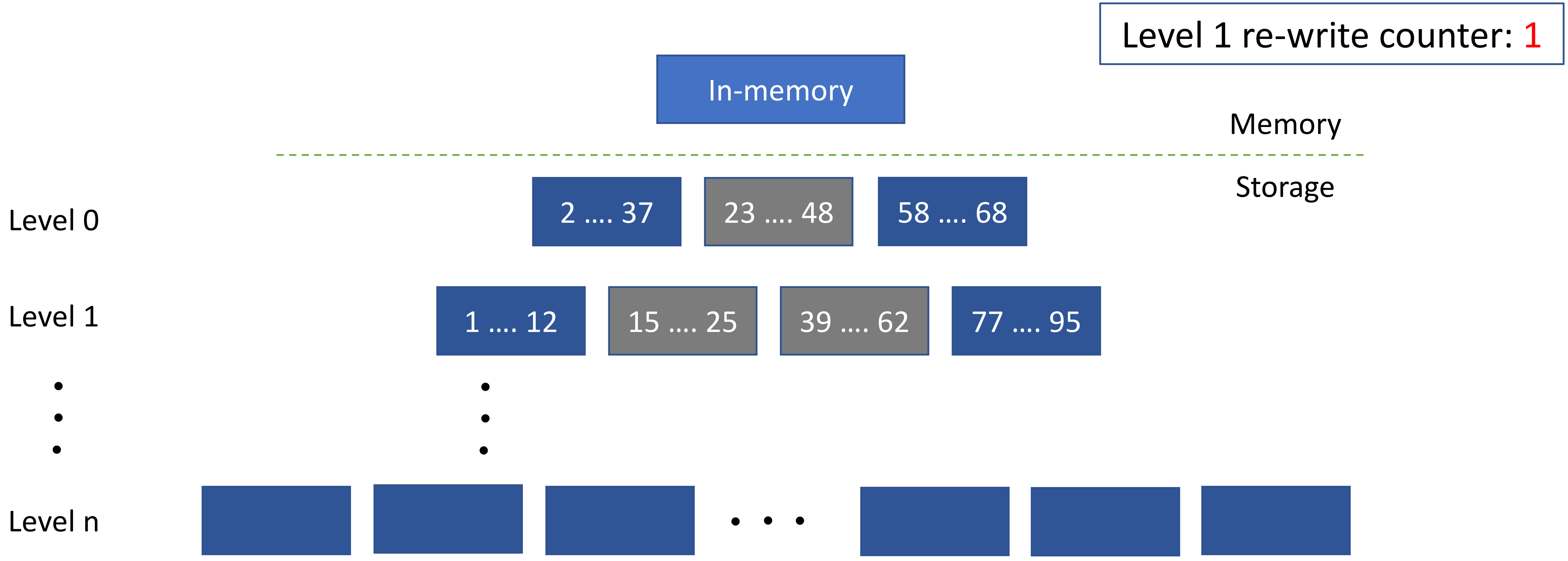
Write amplification: Illustration

Level 1 re-write counter: 1



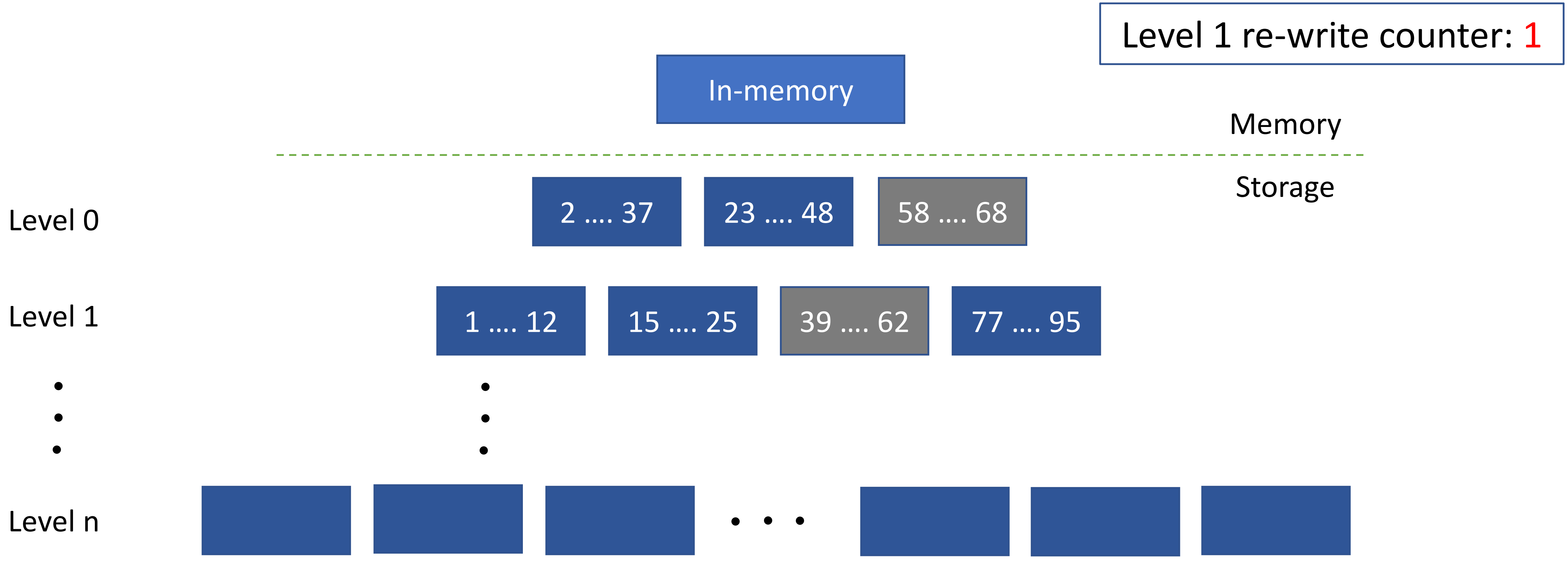
Set of overlapping files between levels 0 and 1

Write amplification: Illustration



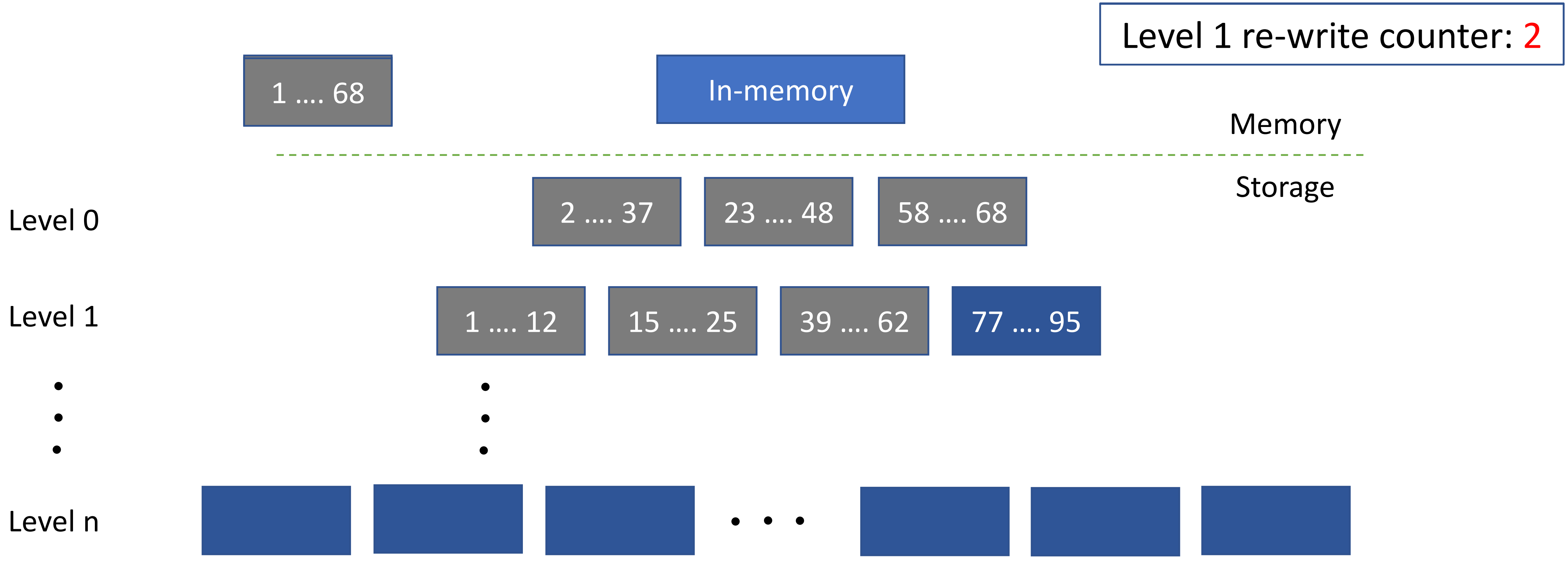
Set of overlapping files between levels 0 and 1

Write amplification: Illustration



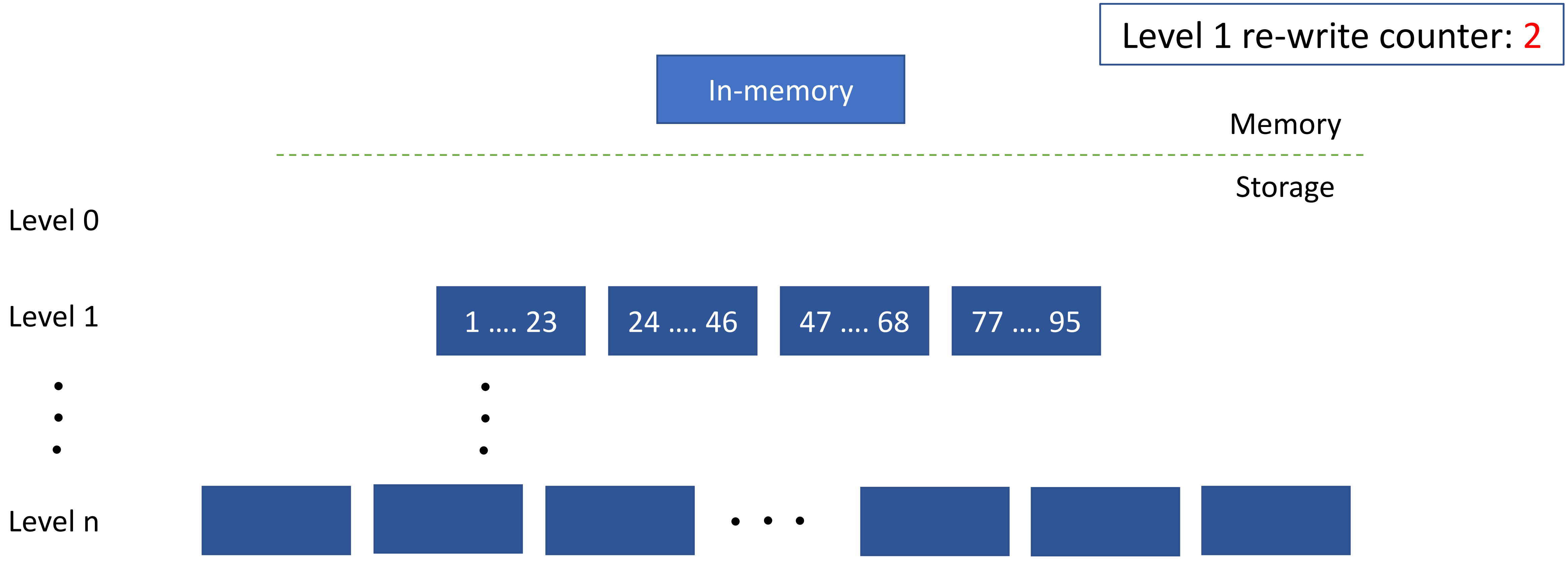
Set of overlapping files between levels 0 and 1

Write amplification: Illustration



Compacting level 0 with level 1

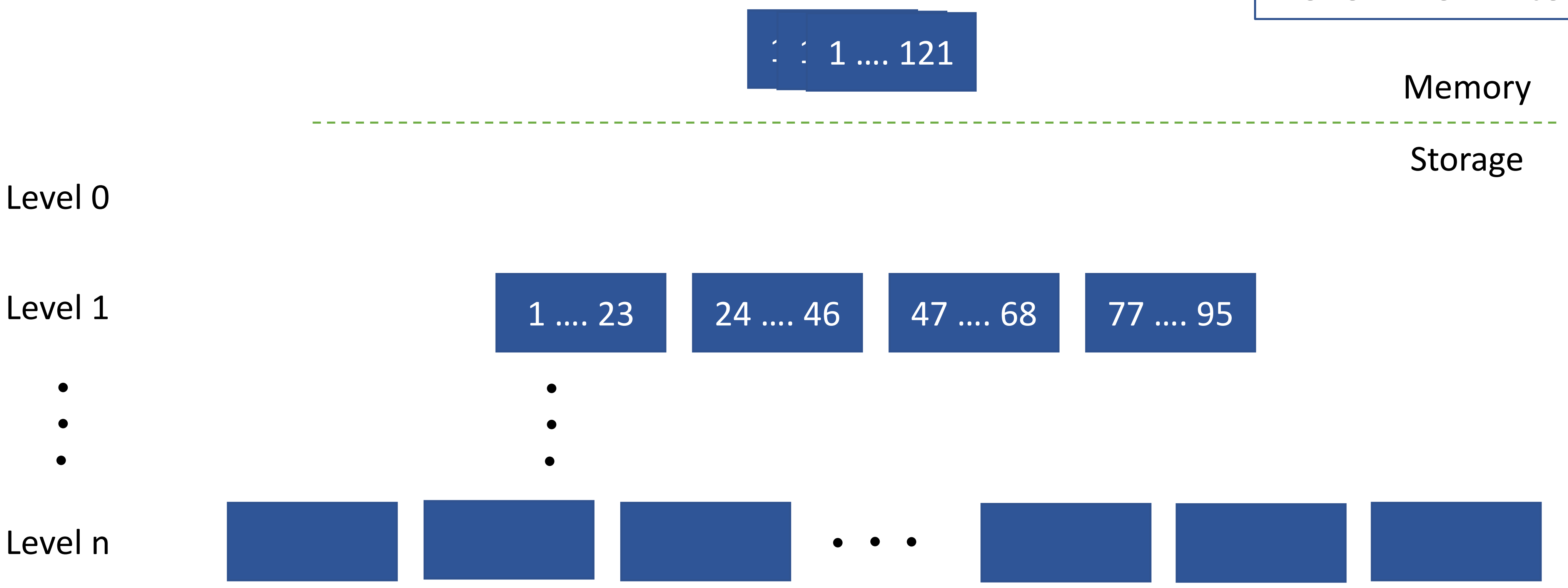
Write amplification: Illustration



Level 0 is compacted

Write amplification: Illustration

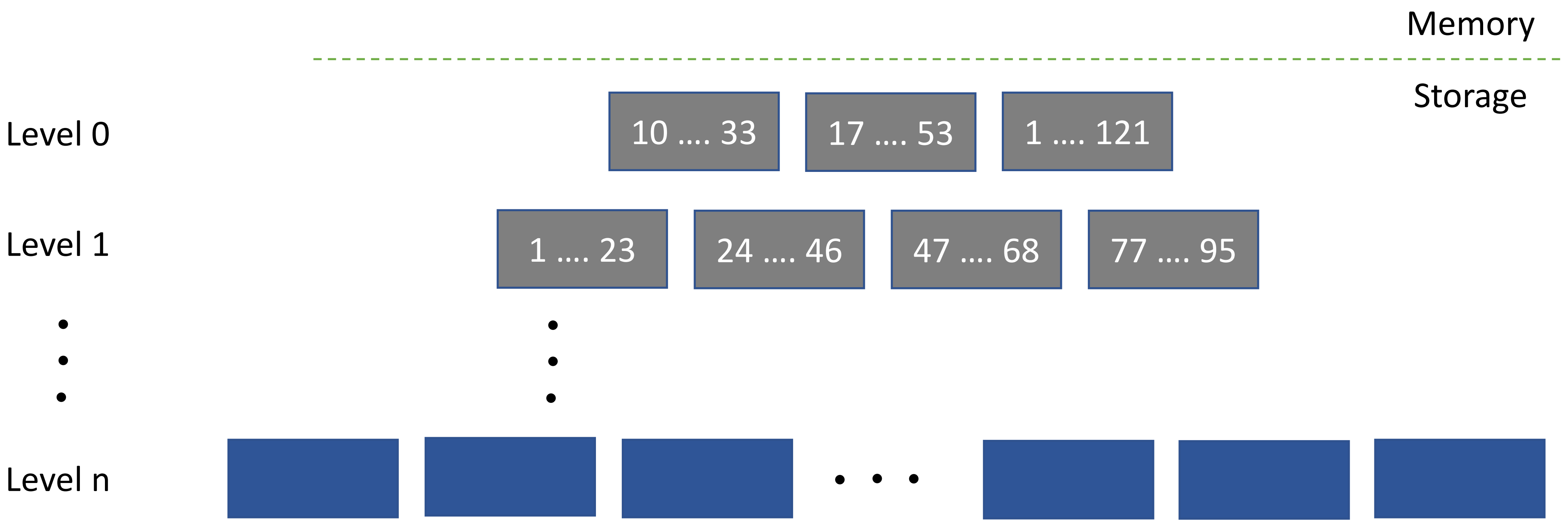
Level 1 re-write counter: 2



Data is being flushed as level 0 files after some Write operations

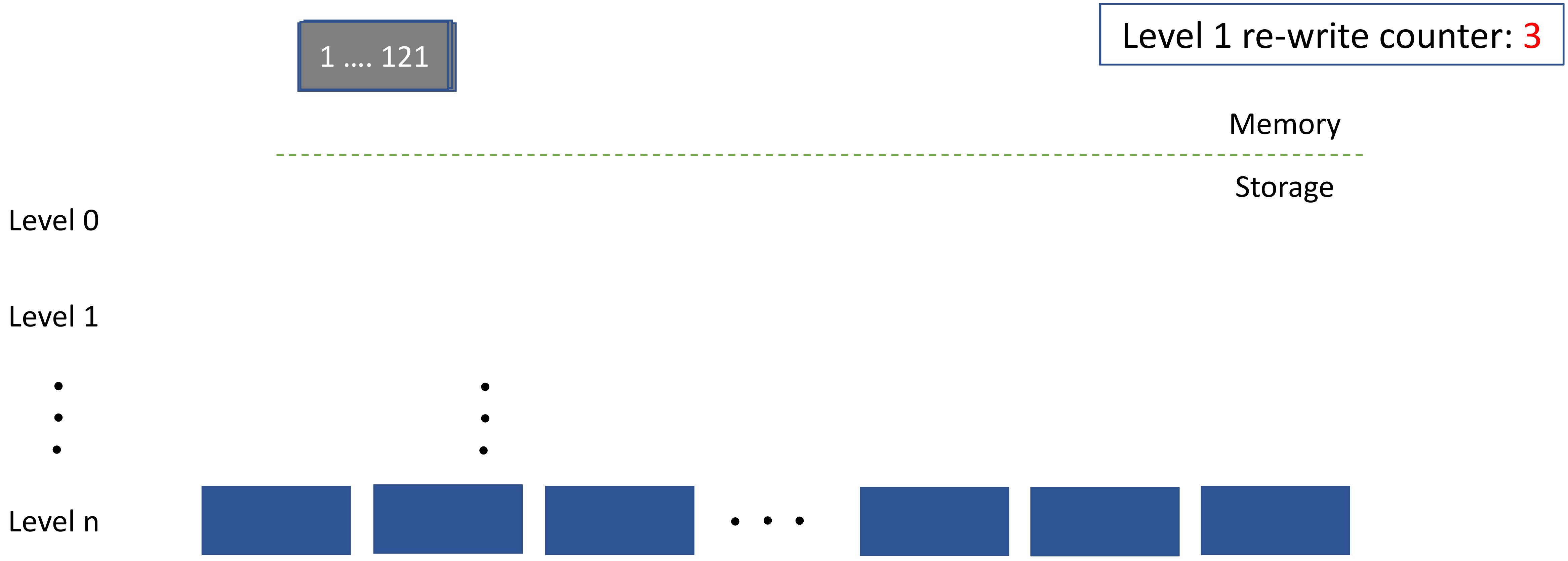
Write amplification: Illustration

Level 1 re-write counter: 2



Compacting level 0 with level 1

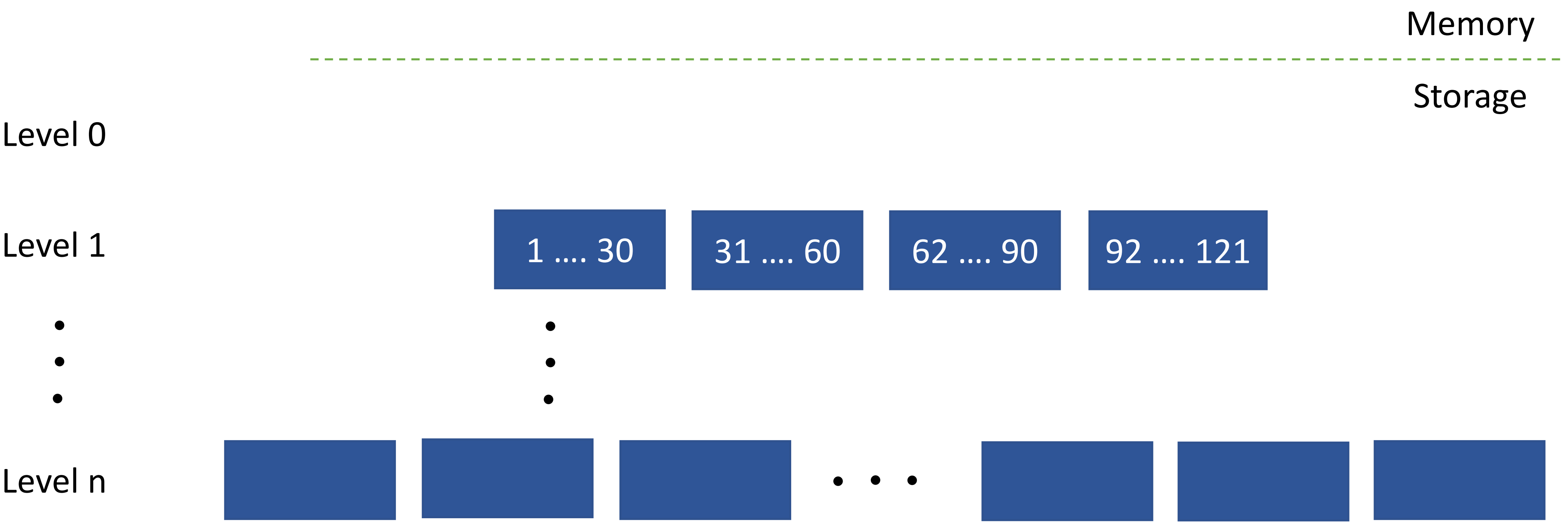
Write amplification: Illustration



Compacting level 0 with level 1

Write amplification: Illustration

Level 1 re-write counter: 3



Existing data is re-written to the same level (1) 3 times

Root cause of write amplification

Rewriting data to the same level
multiple times

To maintain sorted non-overlapping
files in each level

Outline

- Log-Structured Merge Tree (LSM)
- **Fragmented Log-Structured Merge Tree (FLSM)**
- Building PebblesDB using FLSM
- Evaluation
- Conclusion

Naïve approach to reduce write amplification

- Just append the file to the end of next level
- Many (possibly all) overlapping files within a level

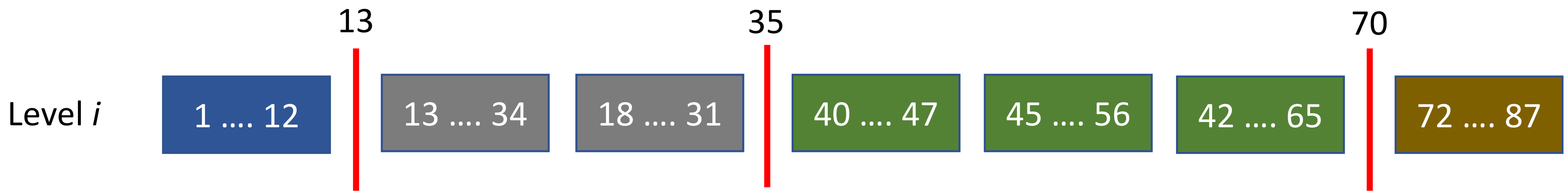
Level *i*

(all files have overlapping key ranges)

- Affects the read performance

Partially sorted levels

- **Hybrid** between all non-overlapping files and all overlapping files
- Inspired from **Skip-List** data structure
- Concrete boundaries (**guards**) to group together overlapping files



(files of same color can have overlapping key ranges)

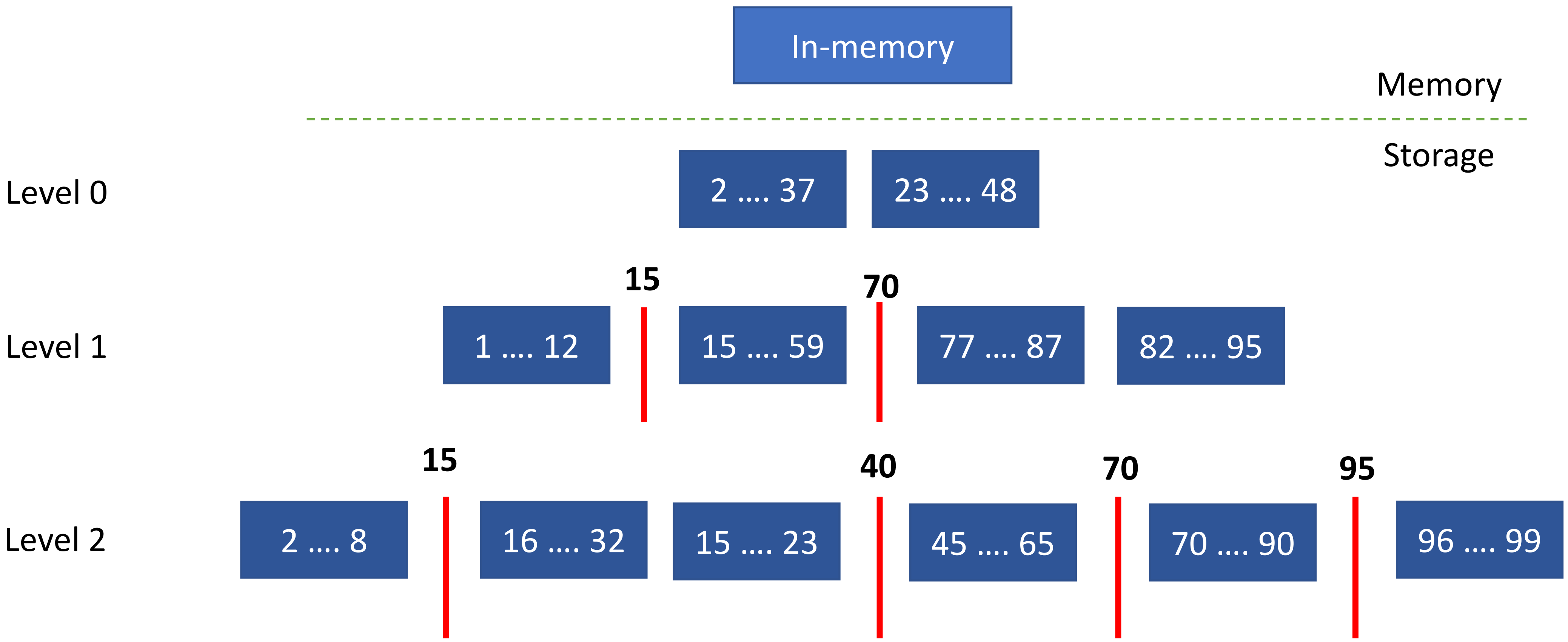
Fragmented Log-Structured Merge Tree

Novel modification of LSM data structure

Uses guards to maintain partially sorted levels

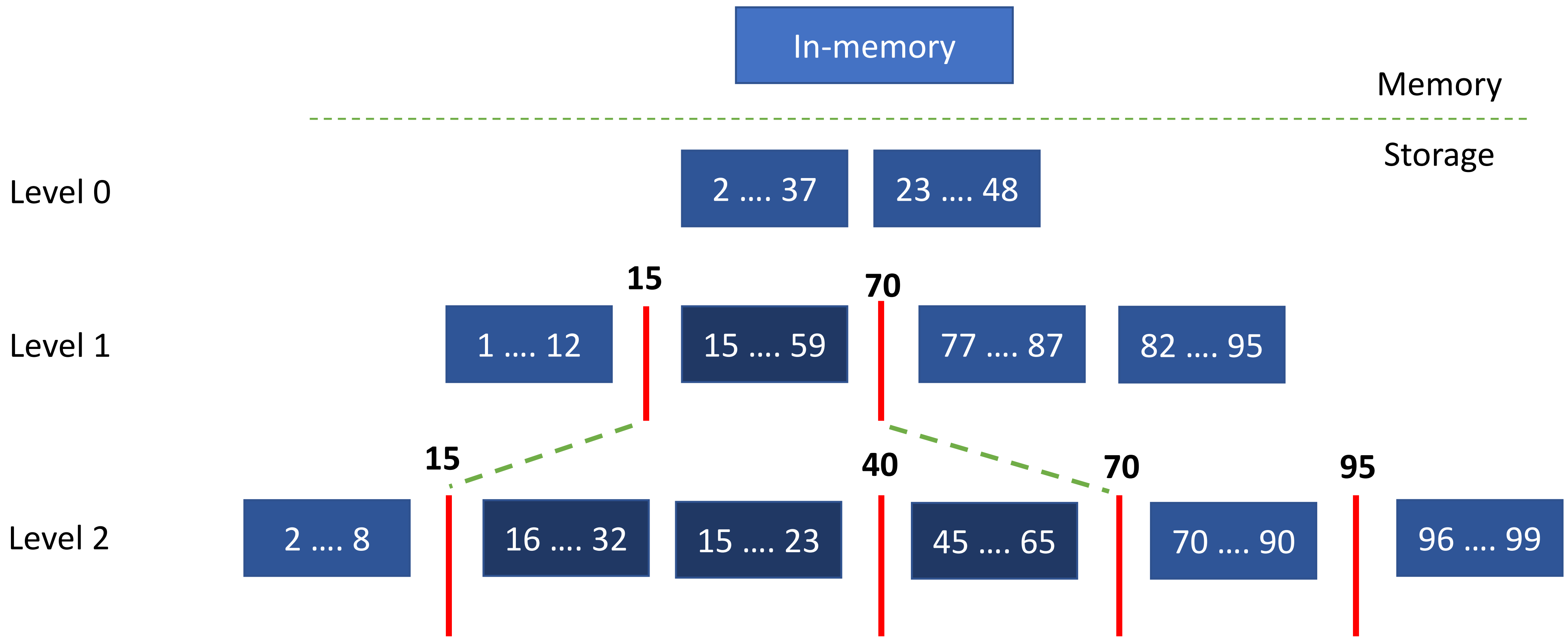
Writes data only once per level in most cases

FLSM structure



Note how files are logically grouped within guards

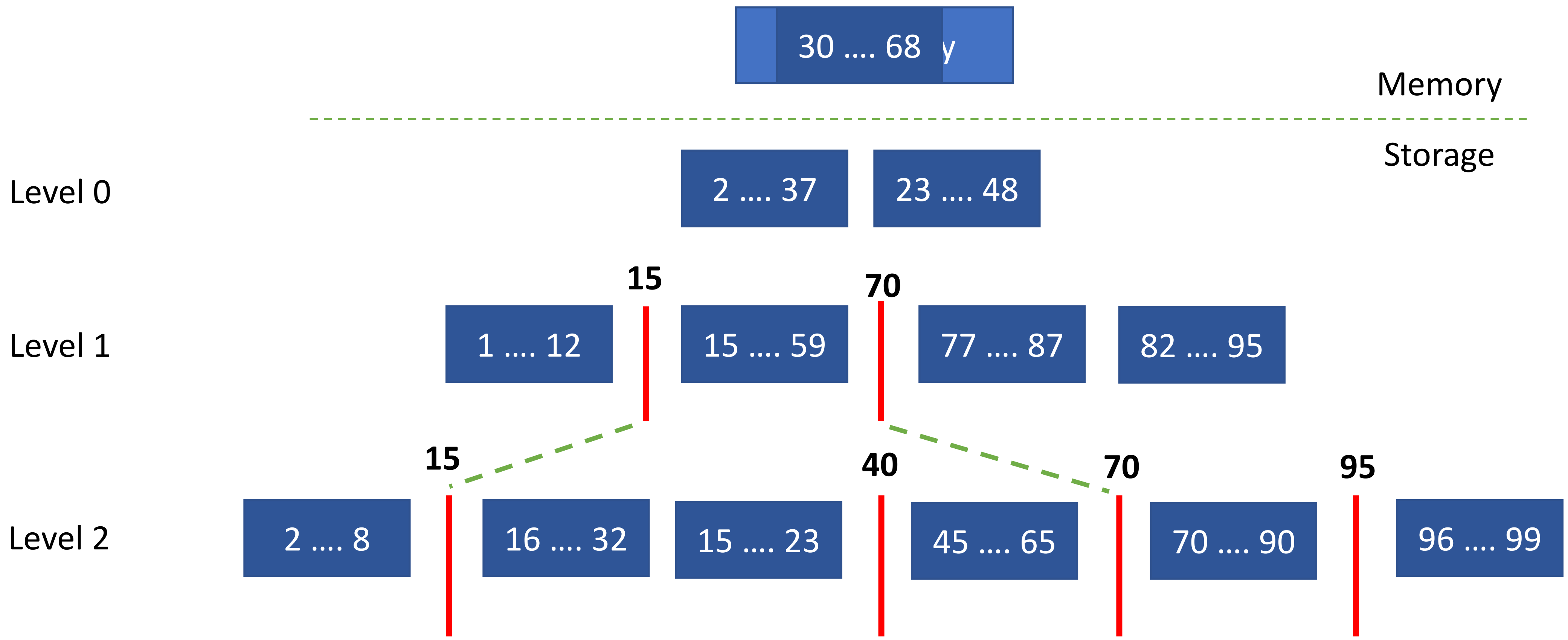
FLSM structure



Guards get more fine grained deeper into the tree

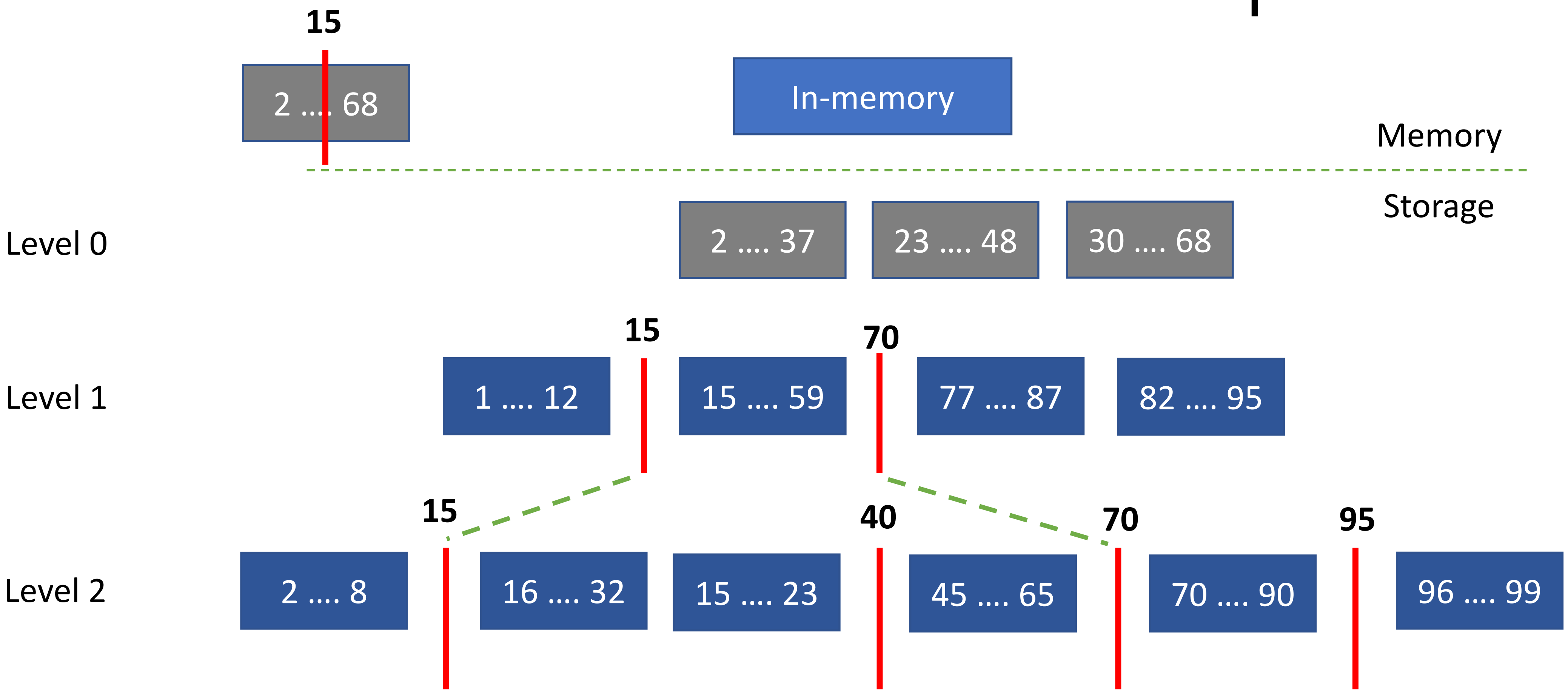
How does FLSM reduce write amplification?

How does FLSM reduce write amplification?



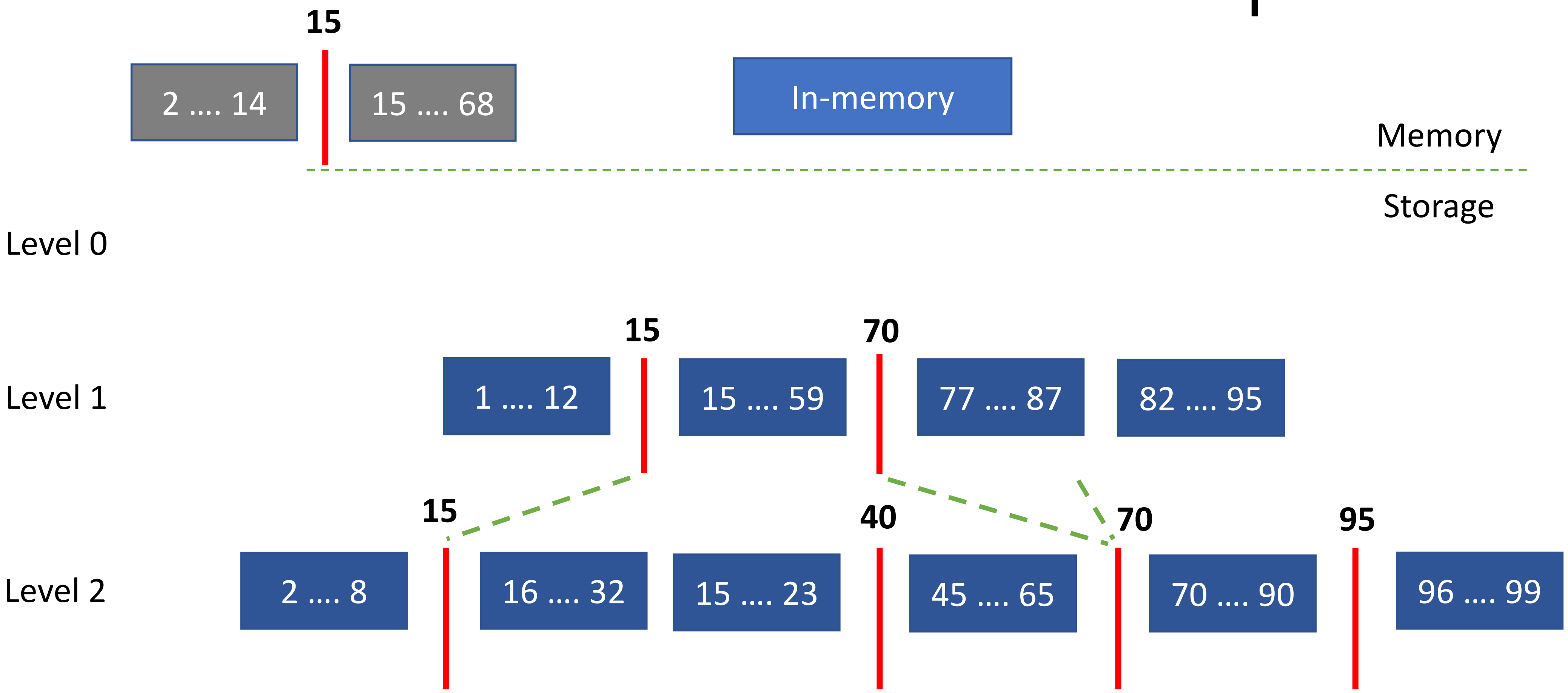
Max files in level 0 is configured to be 2

How does FLSM reduce write amplification?



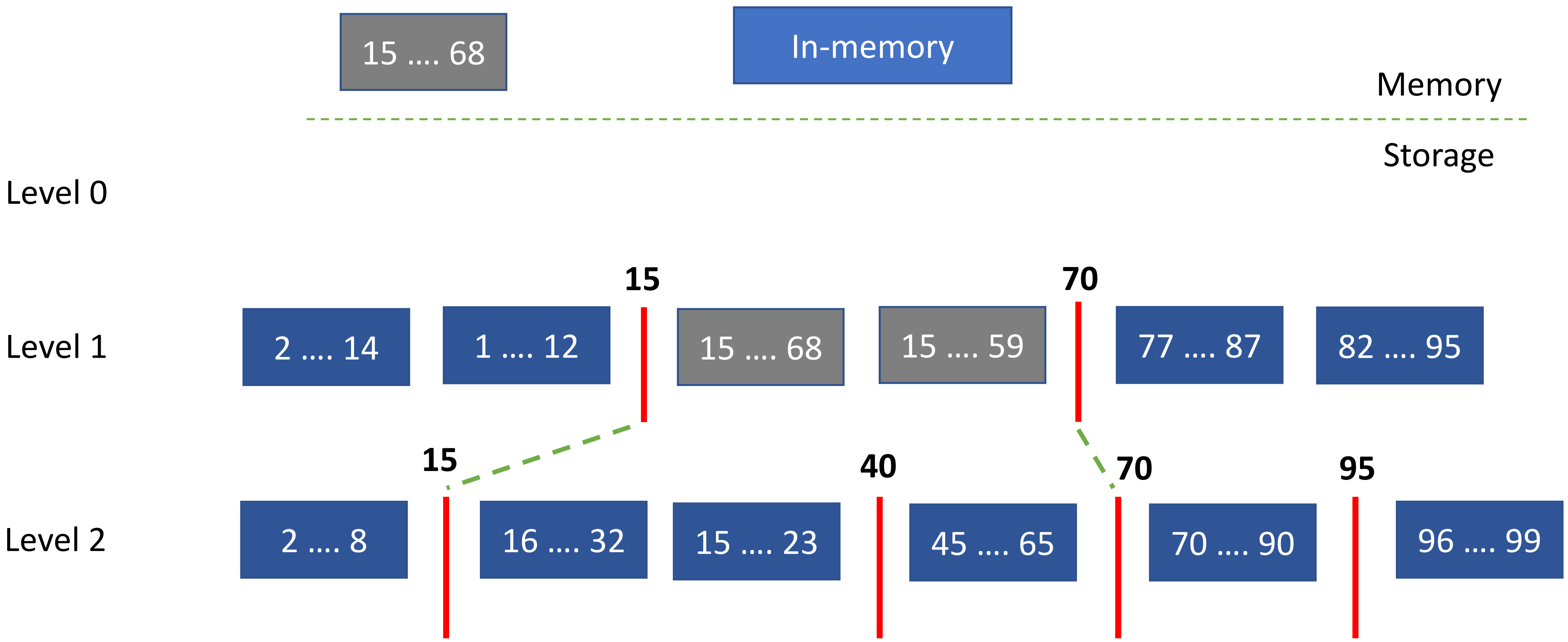
Compacting level 0

How does FLSM reduce write amplification?



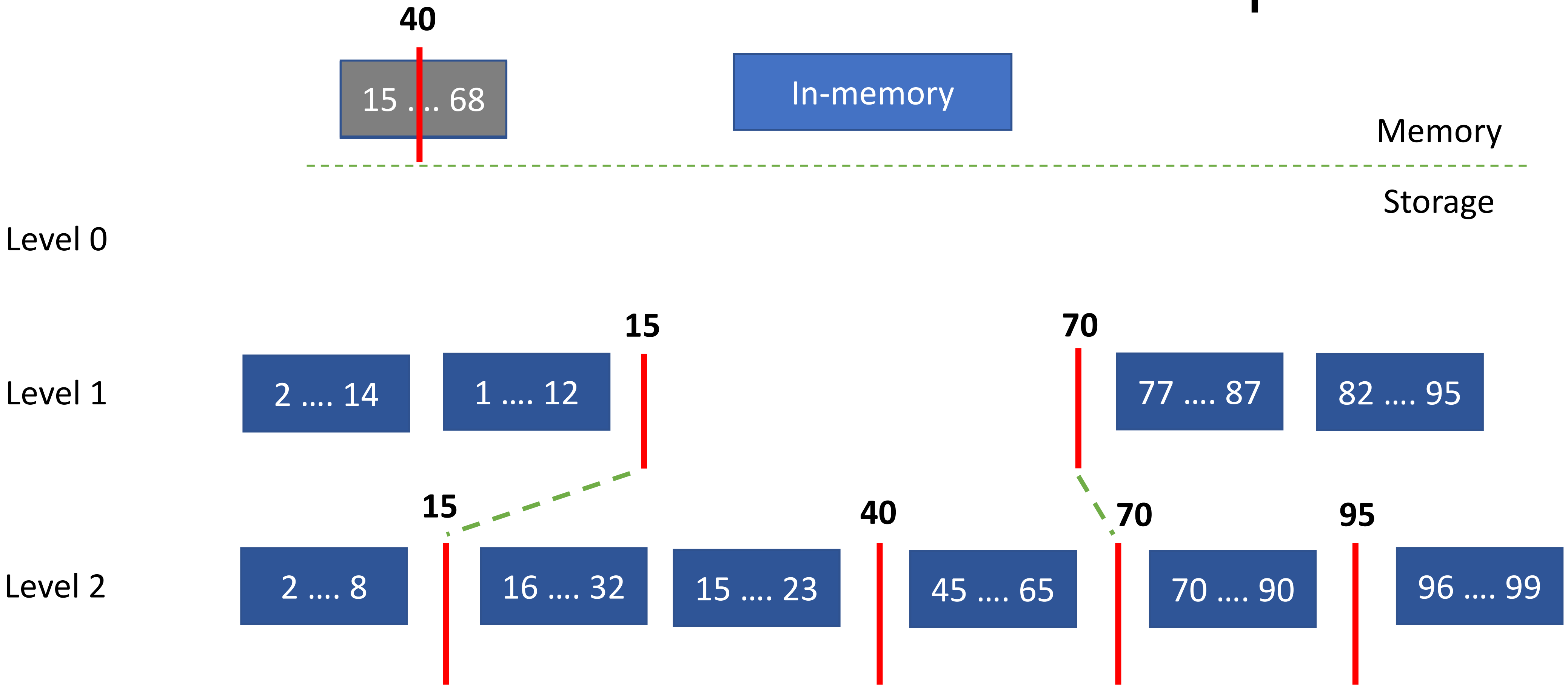
Fragmented files are just appended to next level

How does FLSM reduce write amplification?



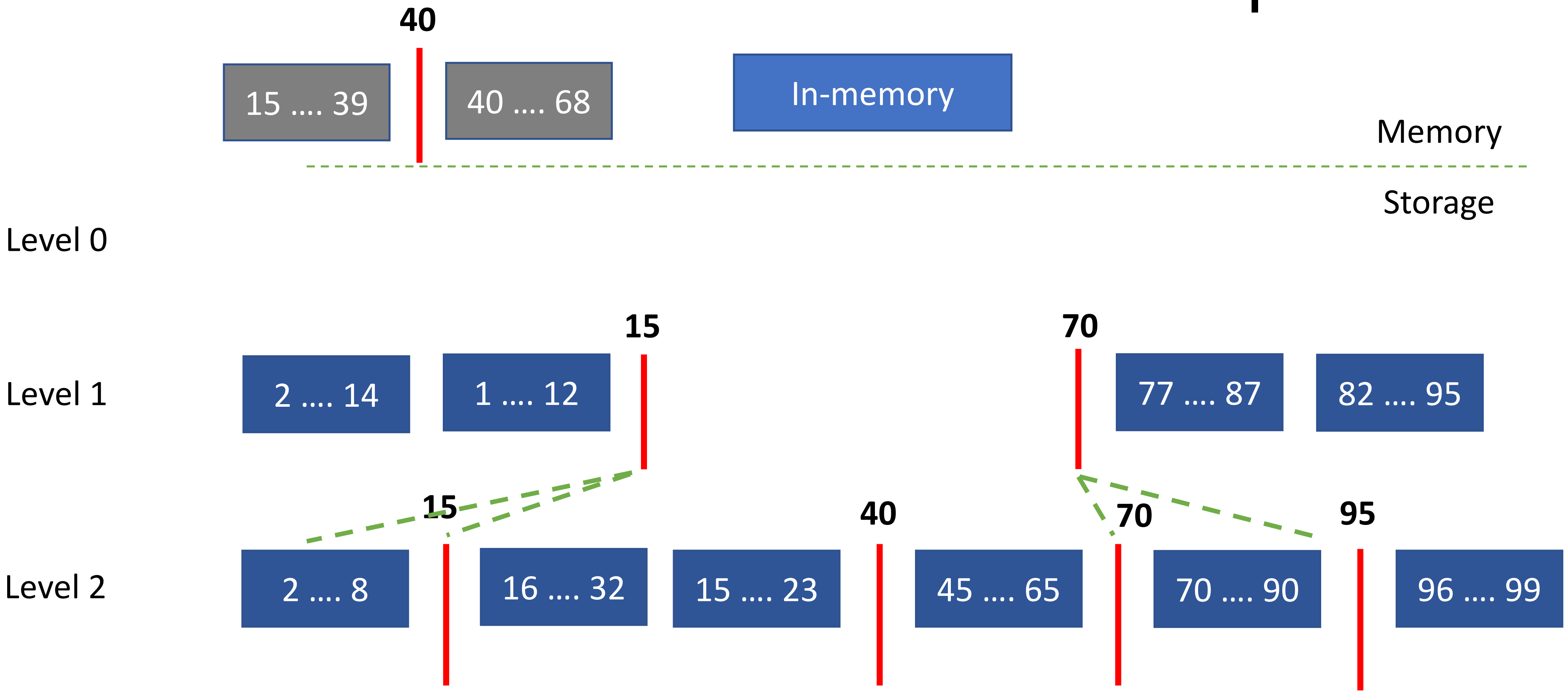
Guard 15 in Level 1 is to be compacted

How does FLSM reduce write amplification?



Files are combined, sorted and fragmented

How does FLSM reduce write amplification?



Fragmented files are just appended to next level

How does FLSM reduce write amplification?

FLSM doesn't re-write data to the same level in most cases

How does FLSM maintain read performance?

FLSM maintains partially sorted levels to efficiently reduce the search space

Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data

Selecting Guards

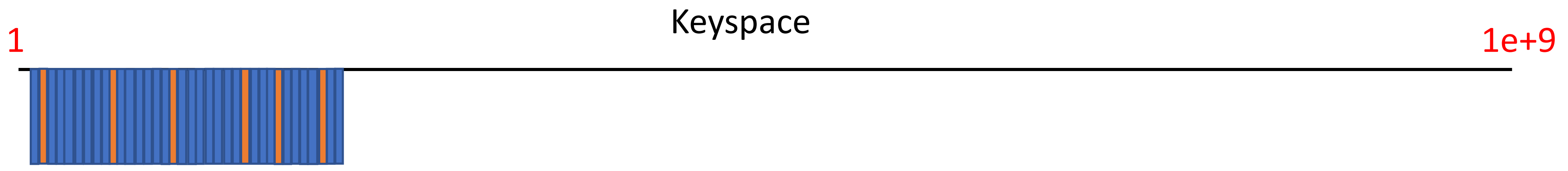
- Guards are chosen randomly and dynamically
- Dependent on the distribution of data

Keyspace

1 1e+9

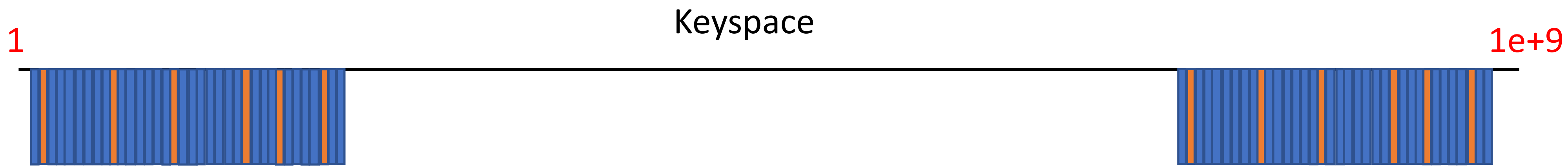
Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data

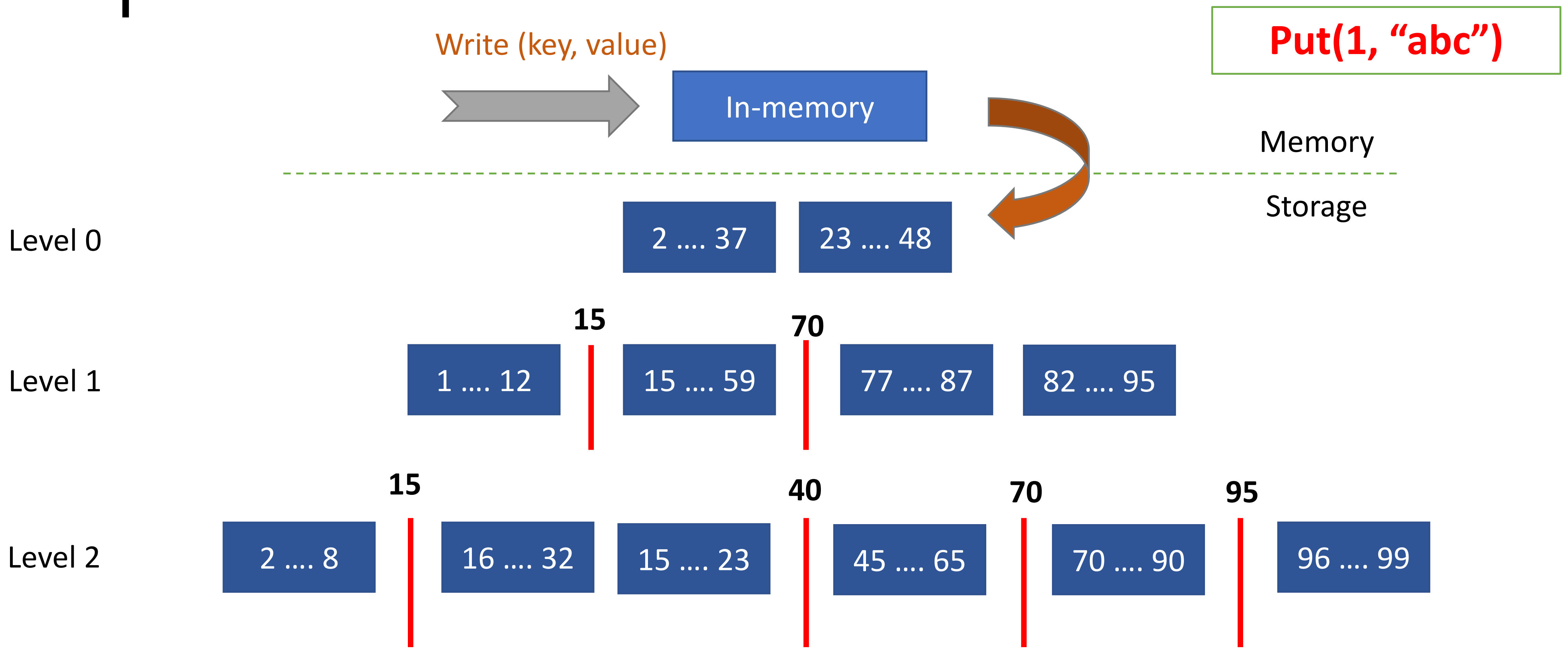


Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data

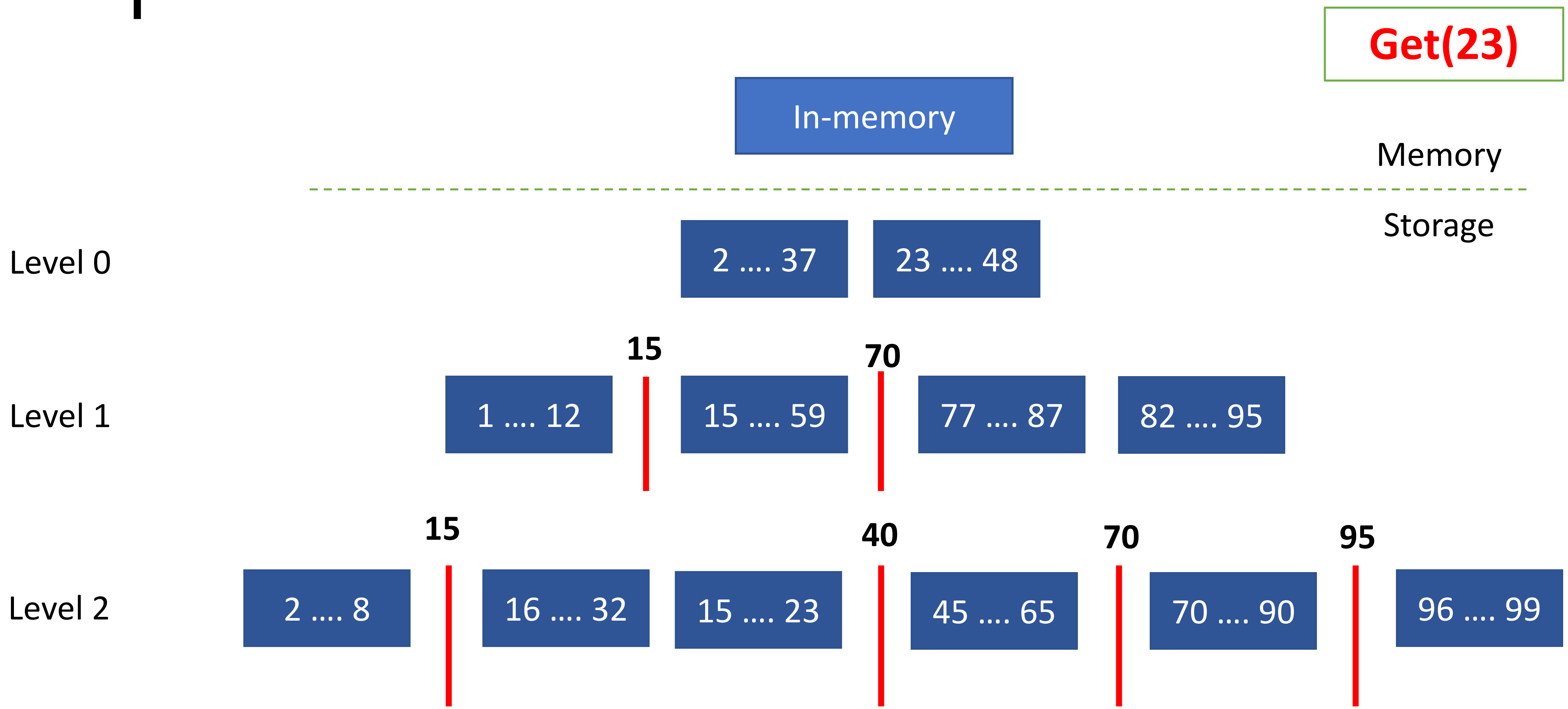


Operations: Write



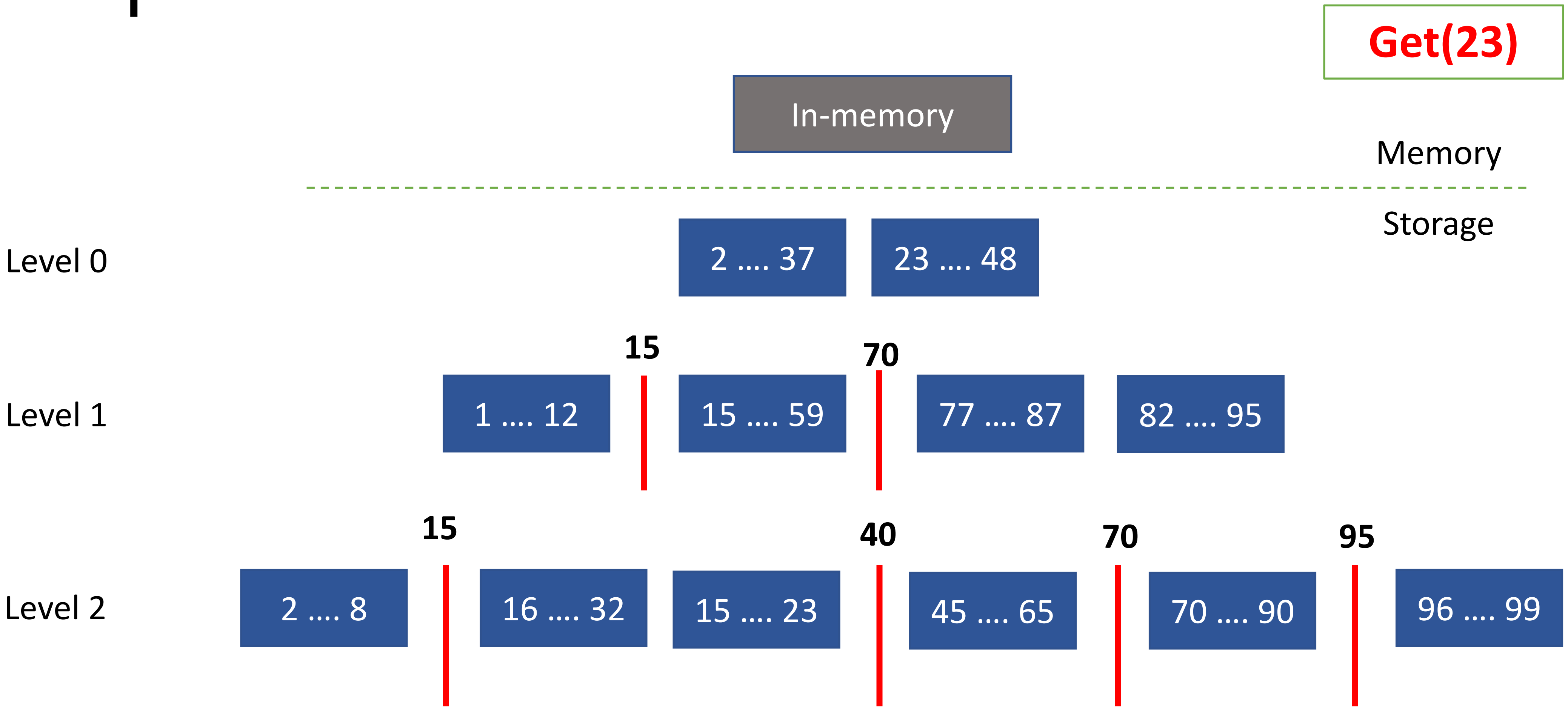
FLSM structure

Operations: Get



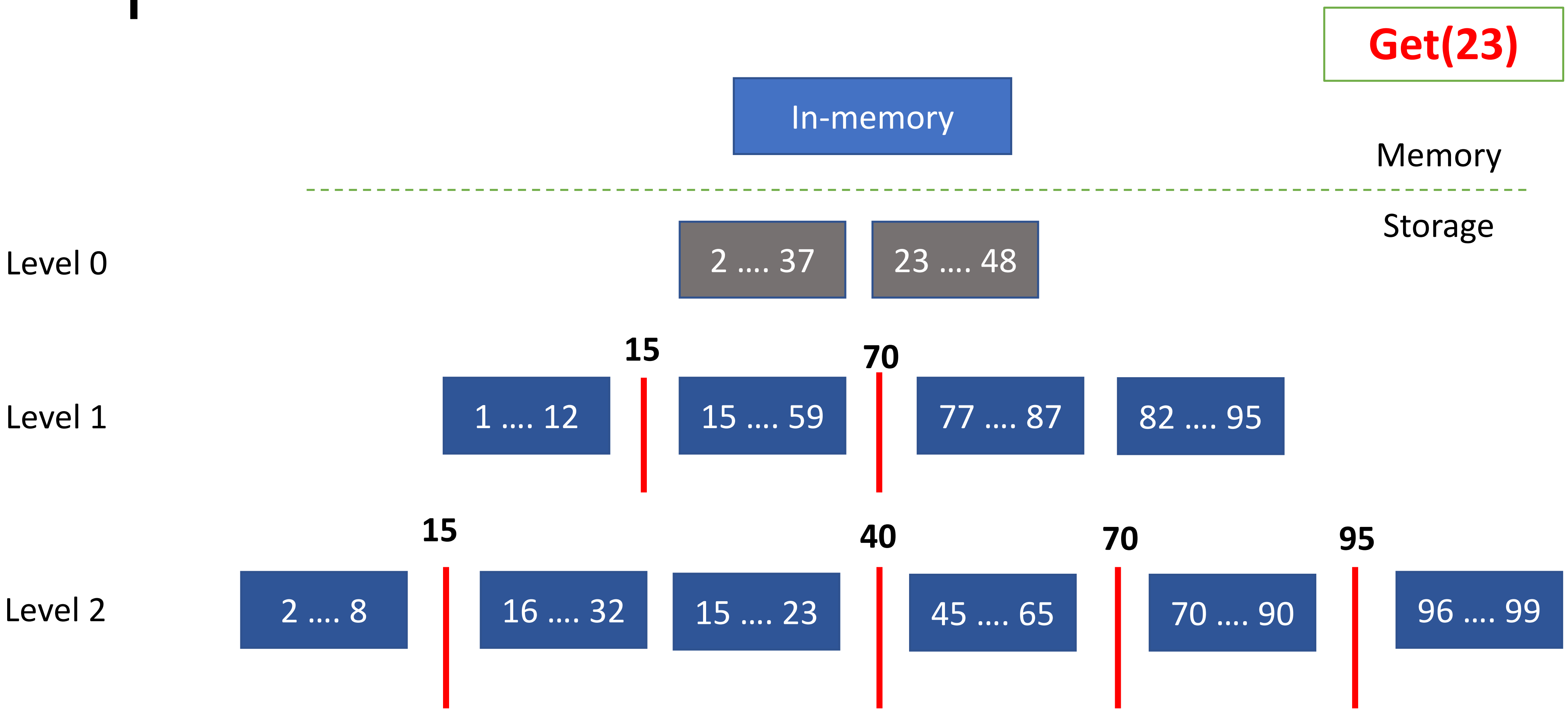
FLSM structure

Operations: Get



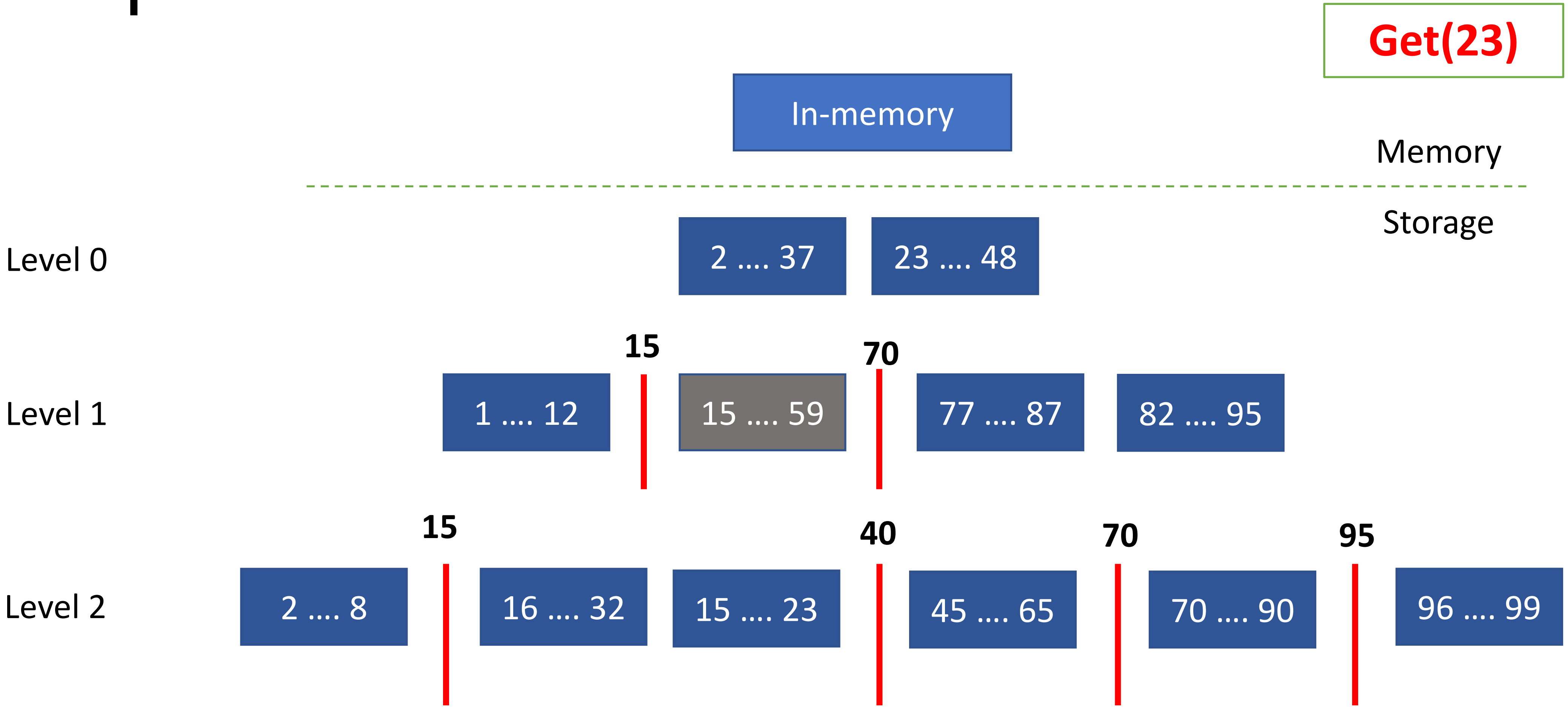
Search level by level starting from memory

Operations: Get



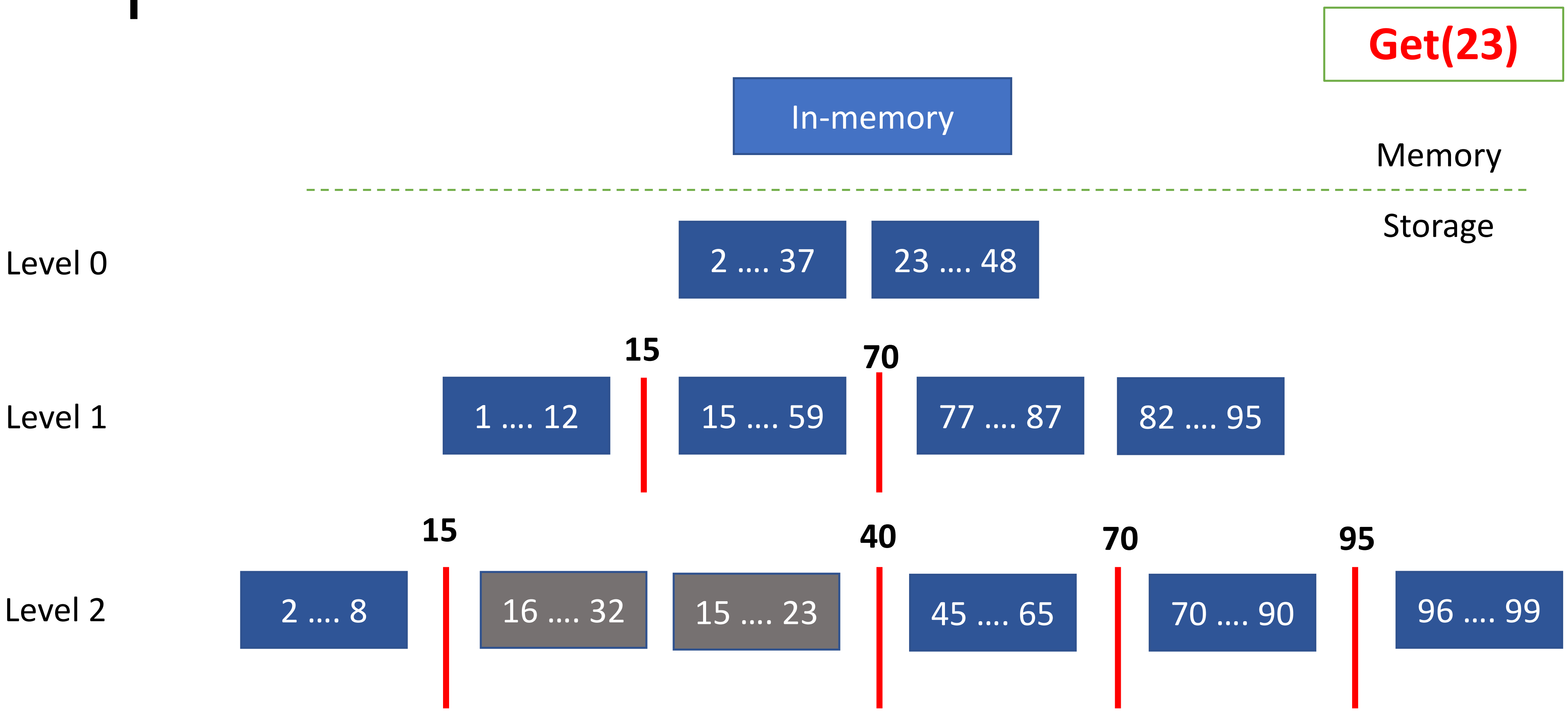
All level 0 files need to be searched

Operations: Get



Level 1: File under guard 15 is searched

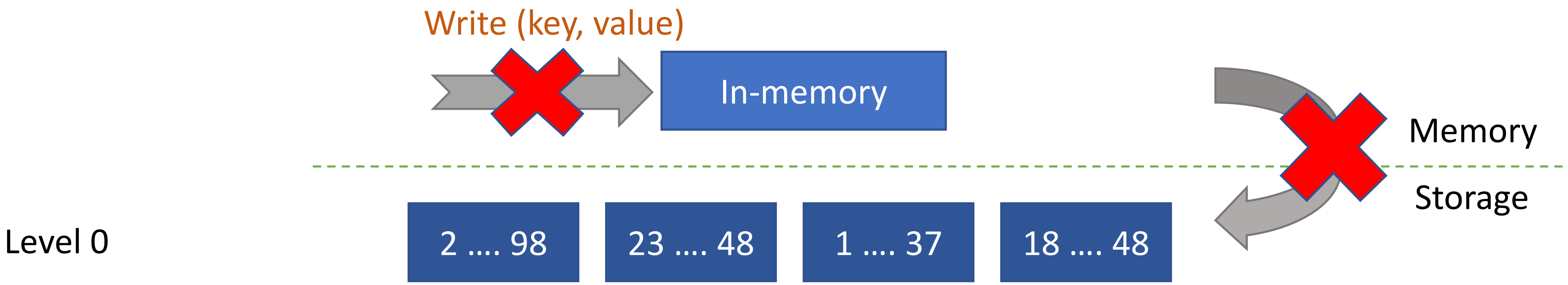
Operations: Get



Level 2: Both the files under guard 15 are searched

High write throughput in FLSM

- Compaction from memory to level 0 is stalled
- Writes to memory is also stalled



If rate of insertion is higher than rate of compaction, write throughput depends on the rate of compaction

High write throughput in FLSM

- Compaction from memory to level 0 is stalled
- Writes to memory is also stalled

FLSM has faster compaction because of lesser I/O and hence higher write throughput

If rate of insertion is higher than rate of compaction, write throughput depends on the rate of compaction

Challenges in FLSM

- Every read/range query operation needs to examine multiple files per level
- For example, if every guard has 5 files, read latency is increased by 5x (assuming no cache hits)

Trade-off between write I/O and read performance

Outline

- Log-Structured Merge Tree (LSM)
- Fragmented Log-Structured Merge Tree (FLSM)
- **Building PebblesDB using FLSM**
- Evaluation
- Conclusion

PebblesDB

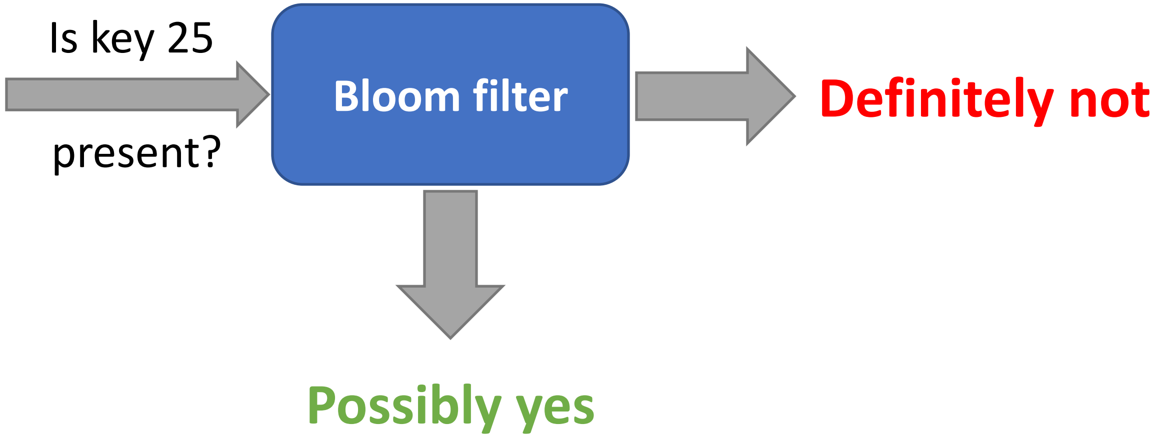
- Built by modifying **HyperLevelDB** (±9100 LOC) to use FLSM
- HyperLevelDB, built over LevelDB, to provide improved parallelism and compaction
- API compatible with LevelDB, but not with RocksDB

Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter

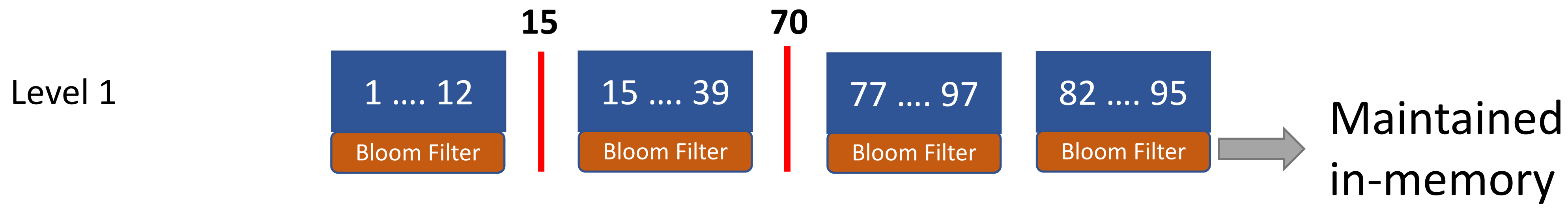
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter



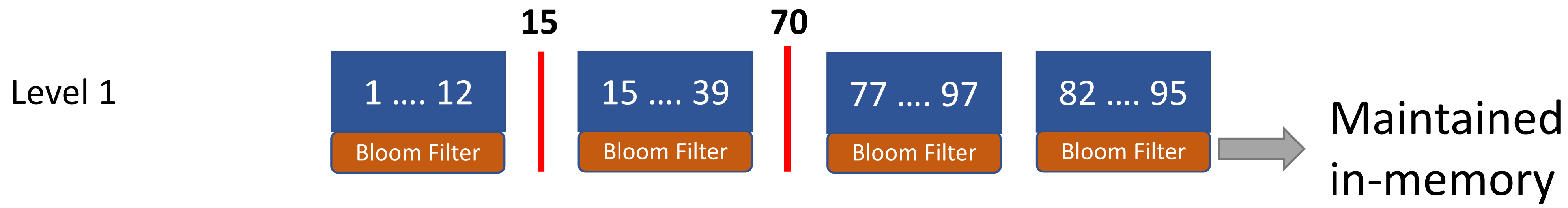
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter



Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter



PebblesDB reads same number of files as any LSM based store

Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter
- Range query performance is improved using parallel threads and better compaction

Outline

- Log-Structured Merge Tree (LSM)
- Fragmented Log-Structured Merge Tree (FLSM)
- Building PebblesDB using FLSM
- **Evaluation**
- Conclusion

Evaluation

Micro-benchmarks

Real world workloads - YCSB

Crash recovery

Small dataset

Low memory

CPU and memory usage

NoSQL applications

Aged file system

Evaluation

Micro-benchmarks

Real world workloads - YCSB

Crash recovery

Small dataset

Low memory

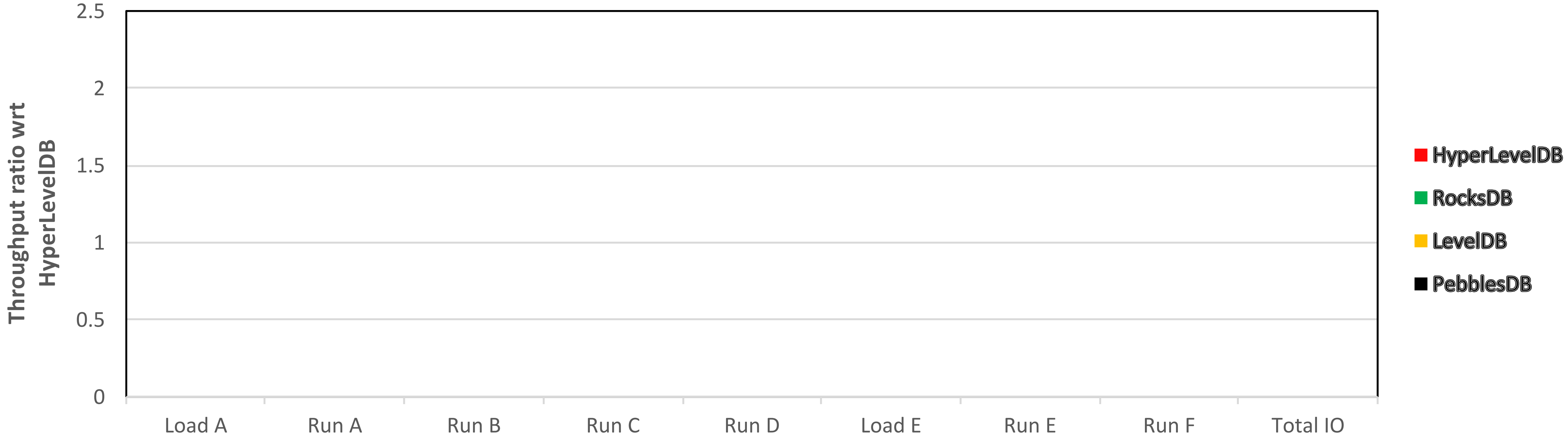
CPU and memory usage

NoSQL applications

Aged file system

Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

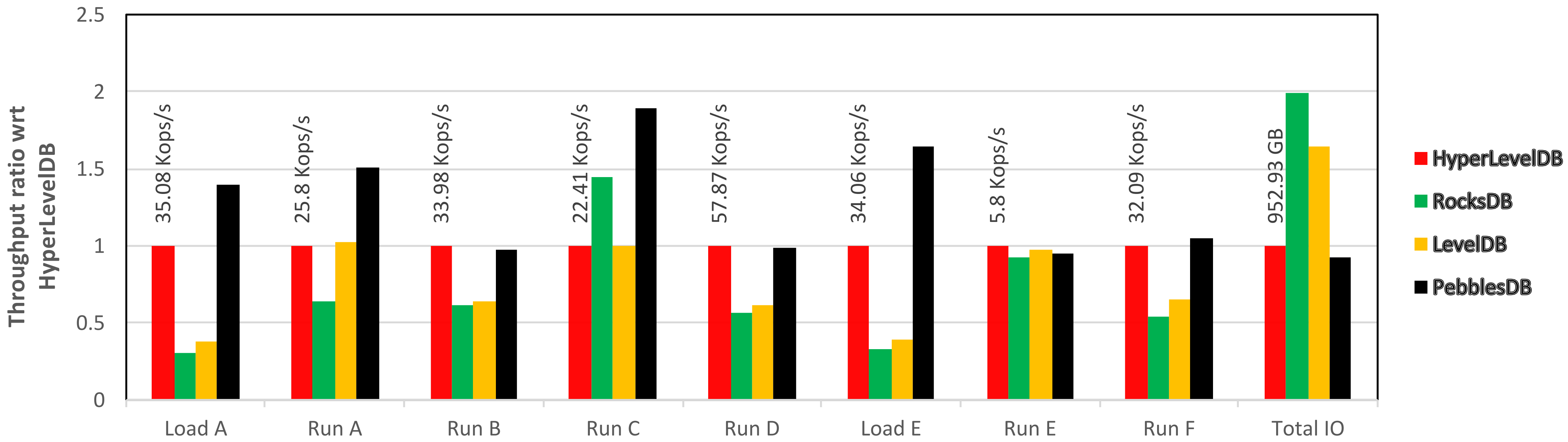


Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads

Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

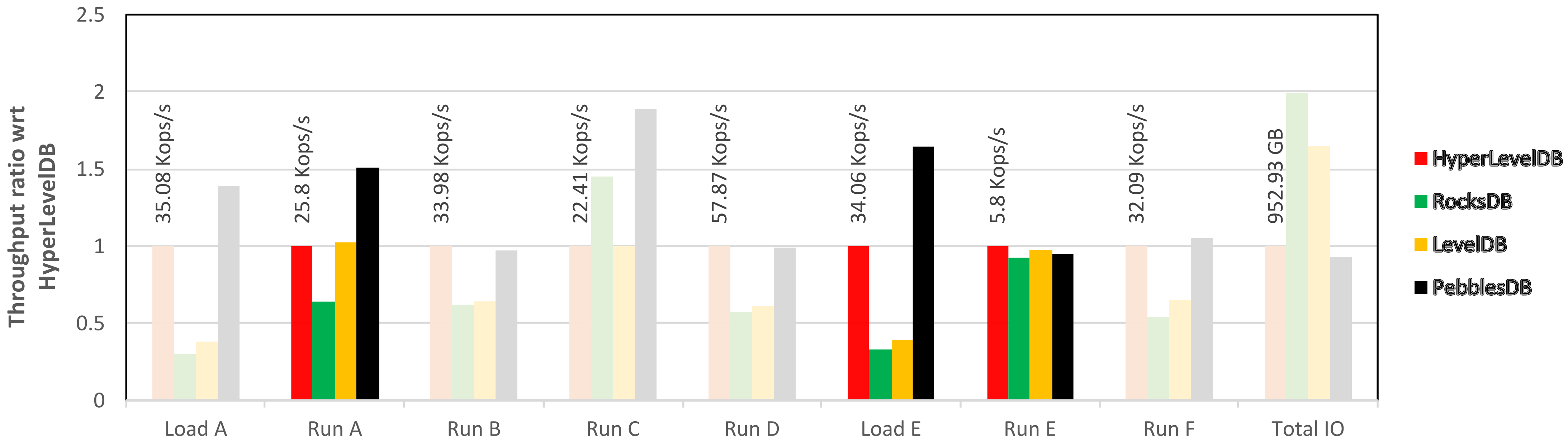
- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

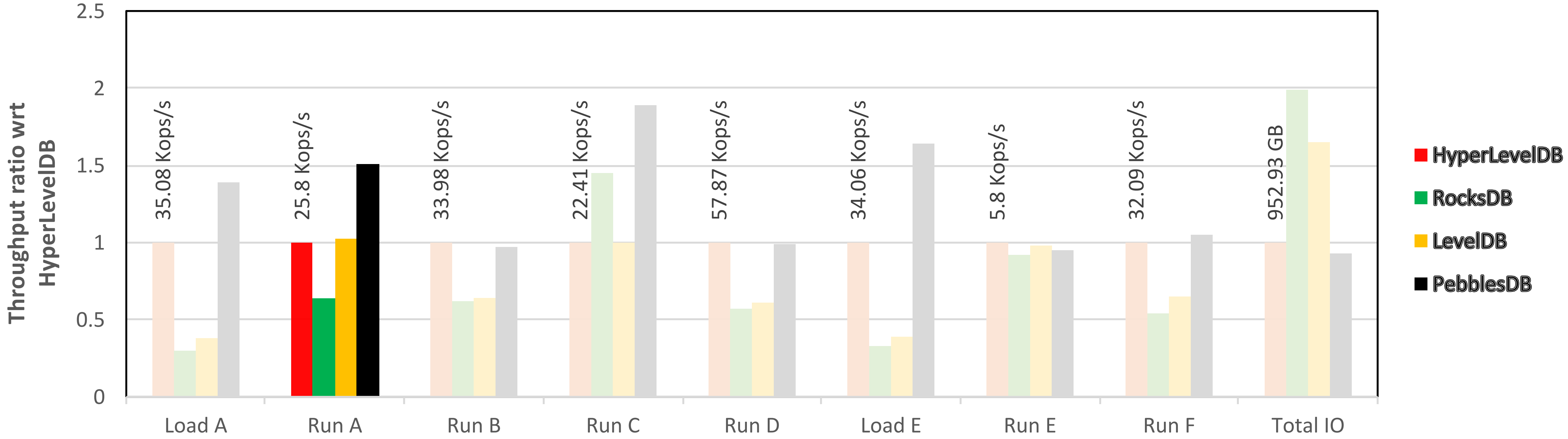
- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB



Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

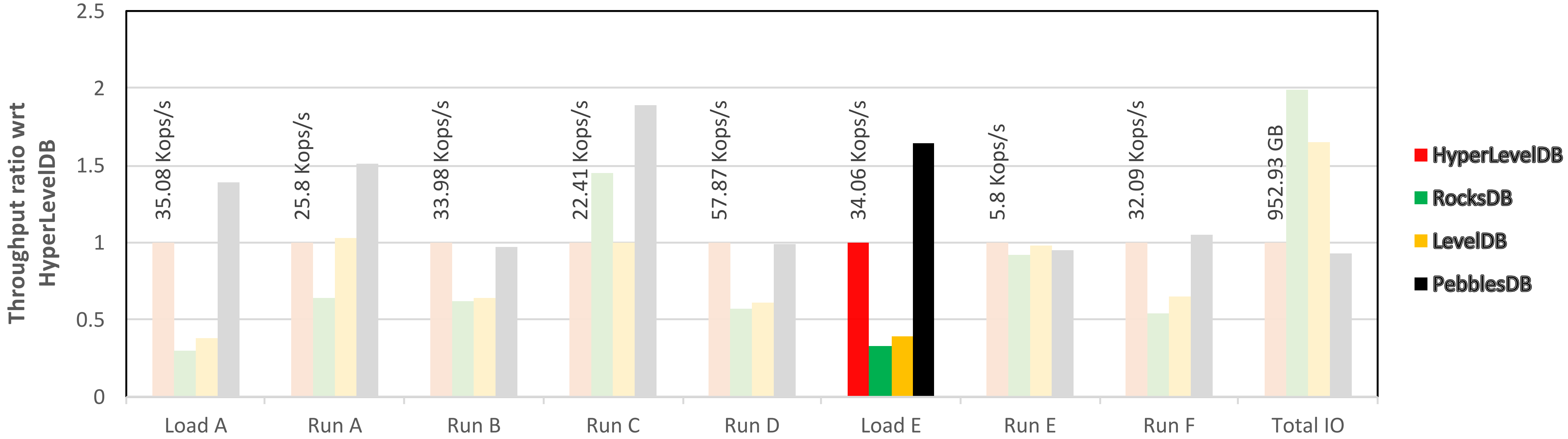
- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB



Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

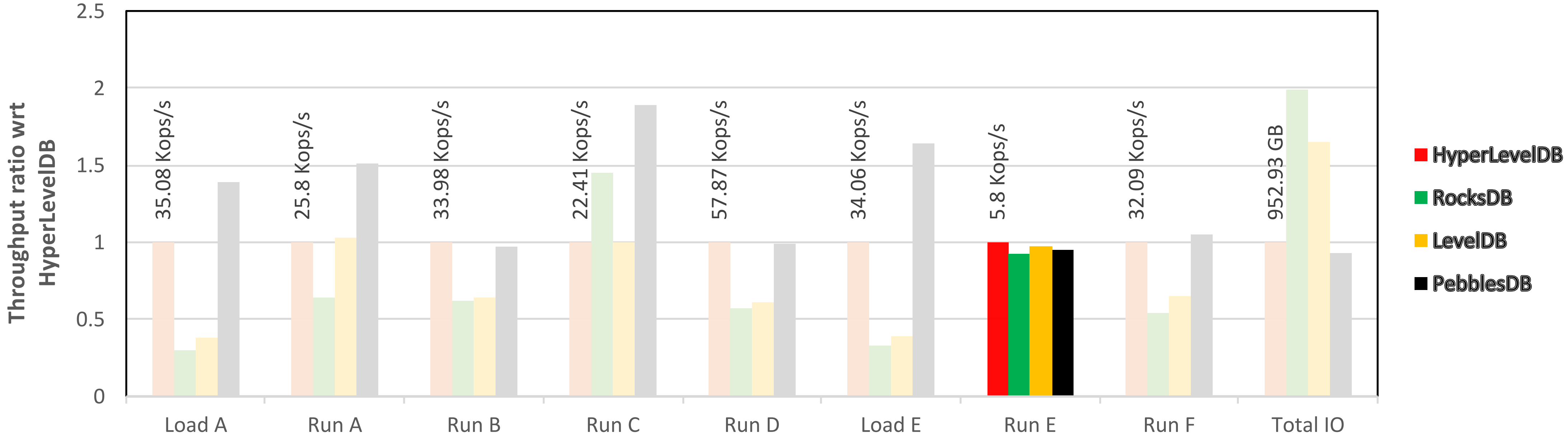
- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB



Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

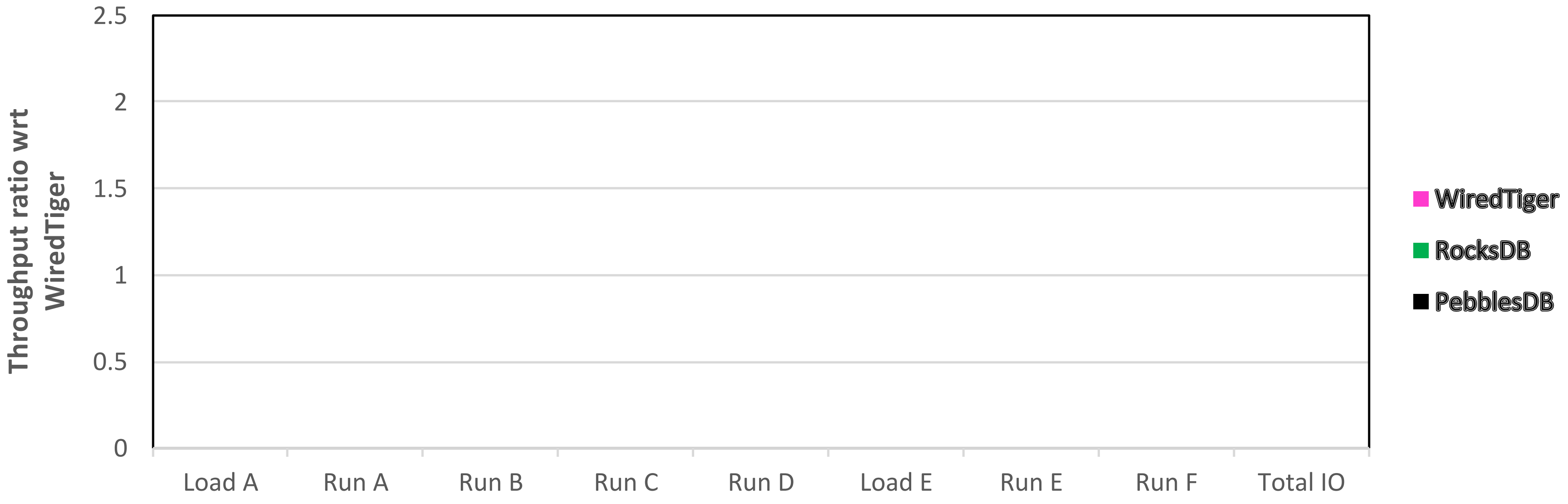
- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB



Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

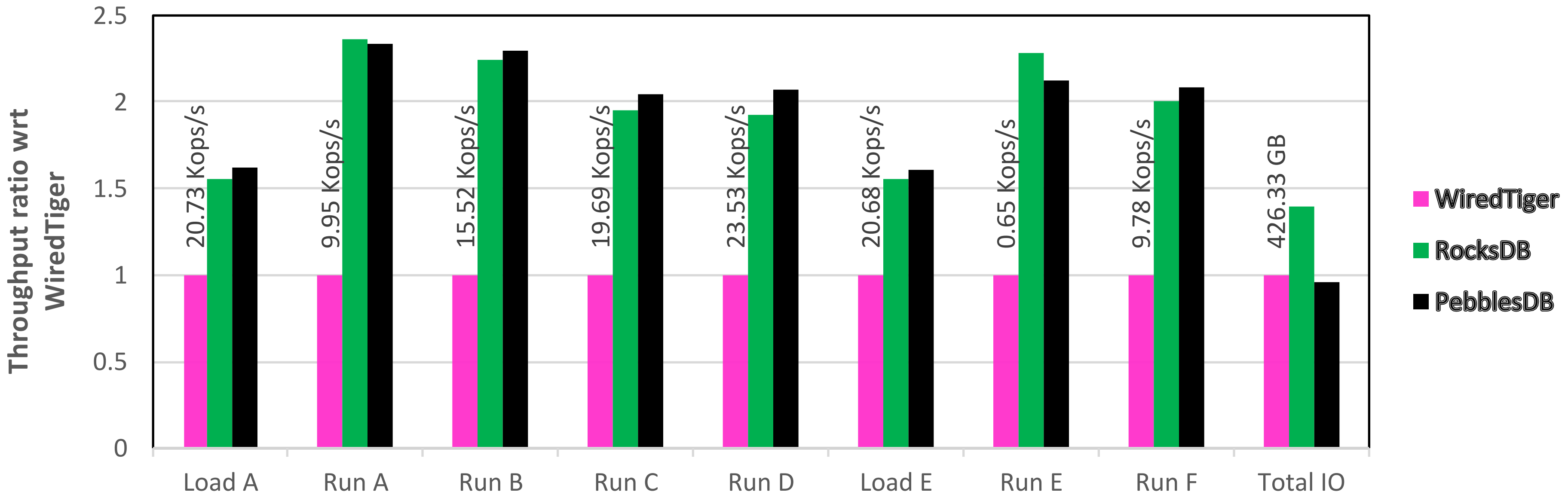
- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

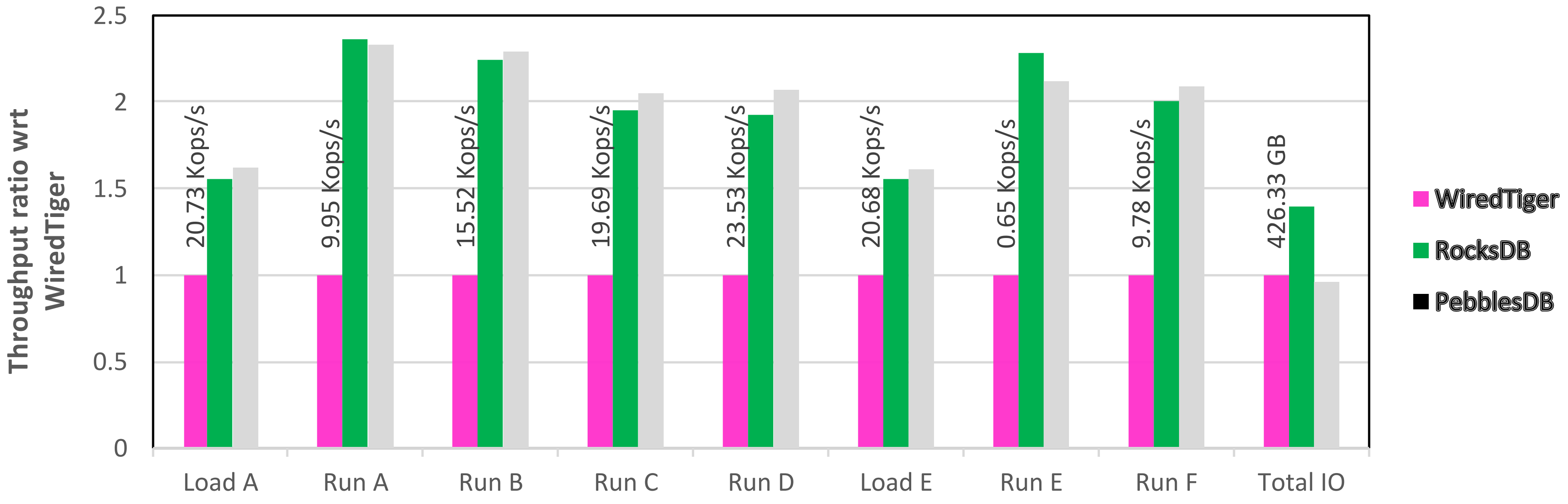
- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

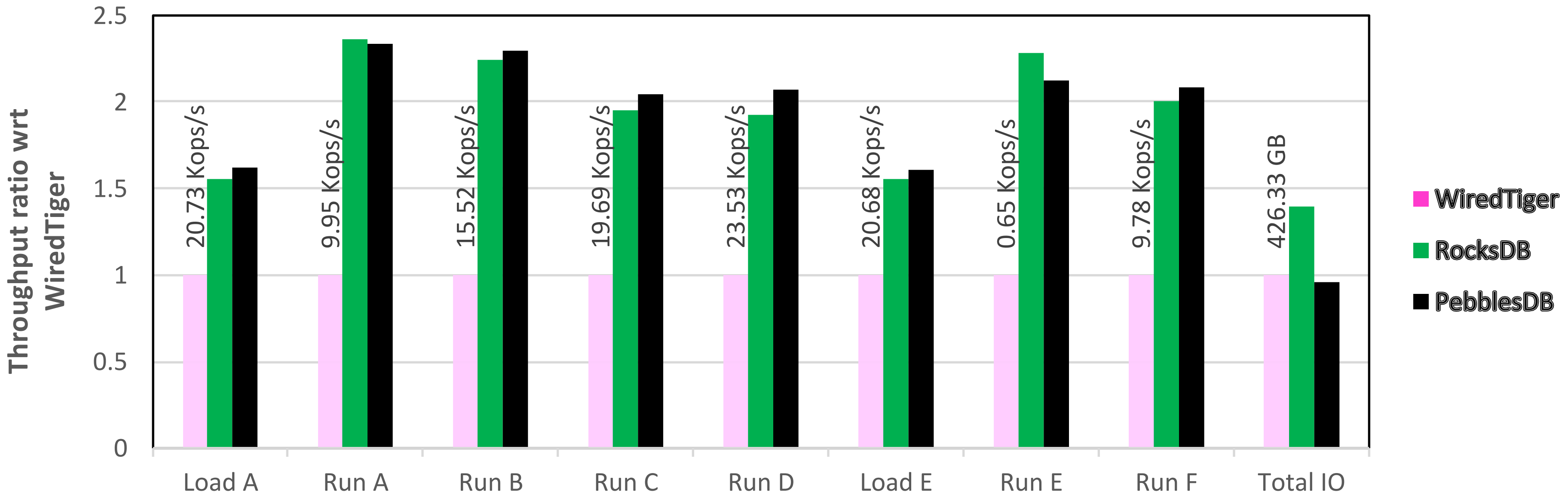
- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

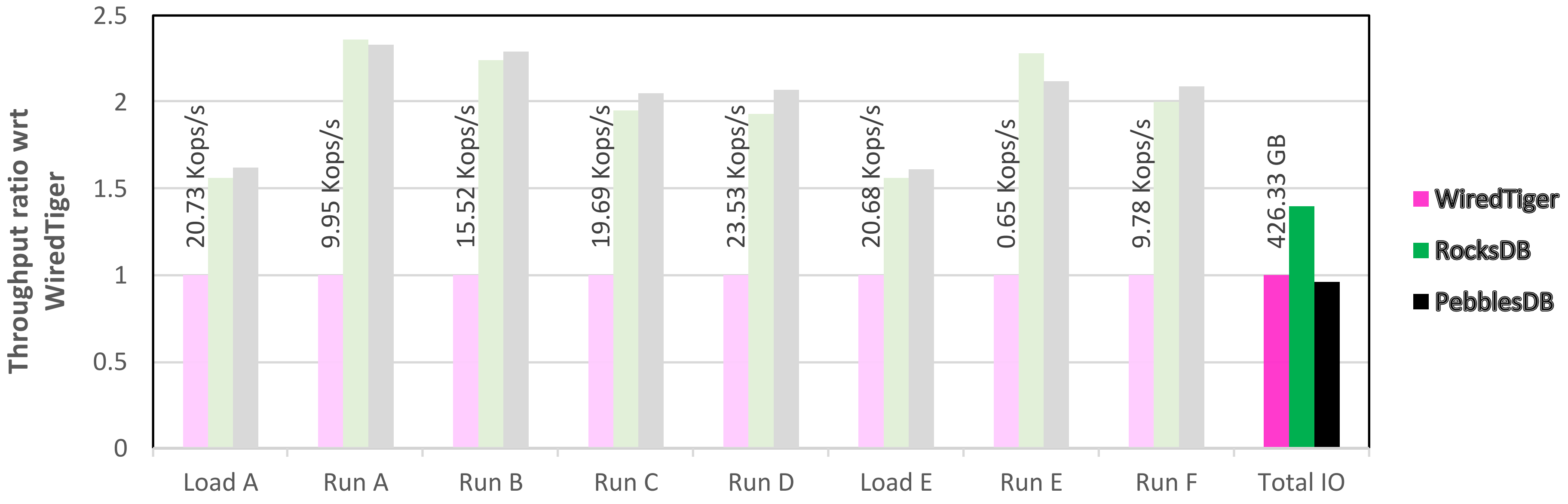
- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

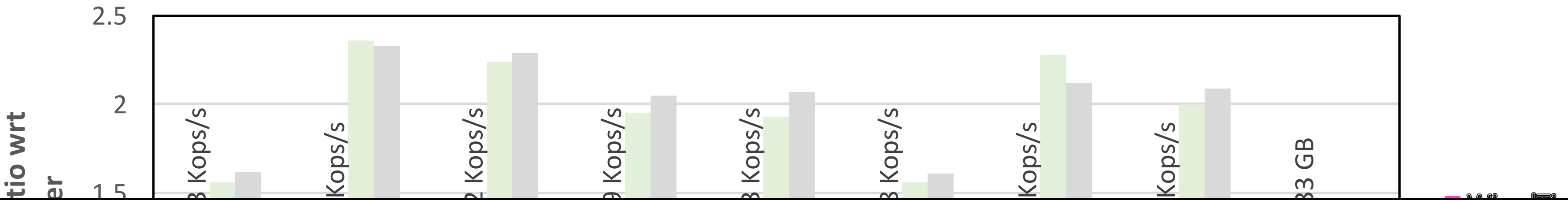
- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



PebblesDB combines low write IO of WiredTiger with high performance of RocksDB

Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

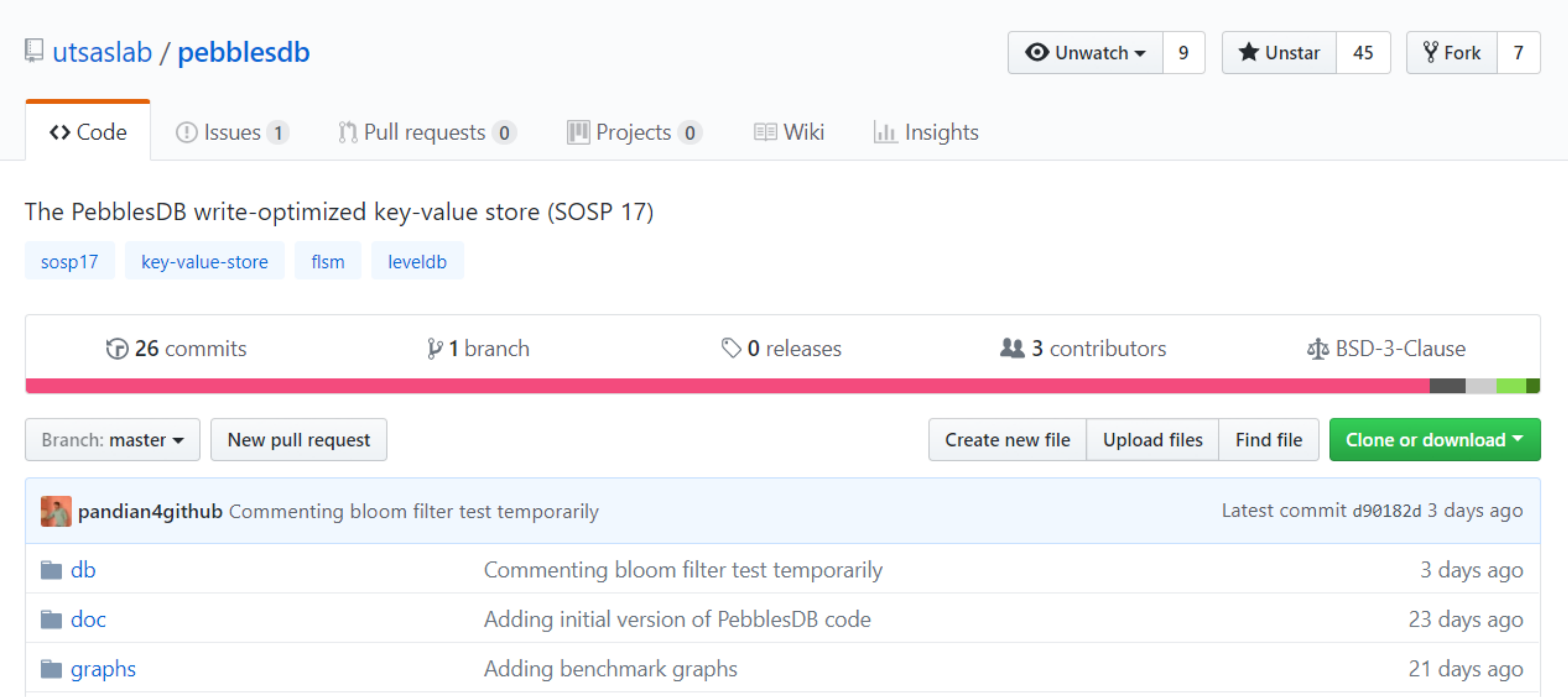
Outline

- Log-Structured Merge Tree (LSM)
- Fragmented Log-Structured Merge Tree (FLSM)
- Building PebblesDB using FLSM
- Evaluation
- **Conclusion**

Conclusion

- PebblesDB: key-value store built on Fragmented Log-Structured Merge Trees
  - Increases write throughput and reduces write IO at the same time
  - Obtains 6X the write throughput of RocksDB
- As key-value stores become more widely used, there have been several attempts to optimize them
- PebblesDB combines algorithmic innovation (the FLSM data structure) with careful systems building

https://github.com/utsaslab/pebblesdb

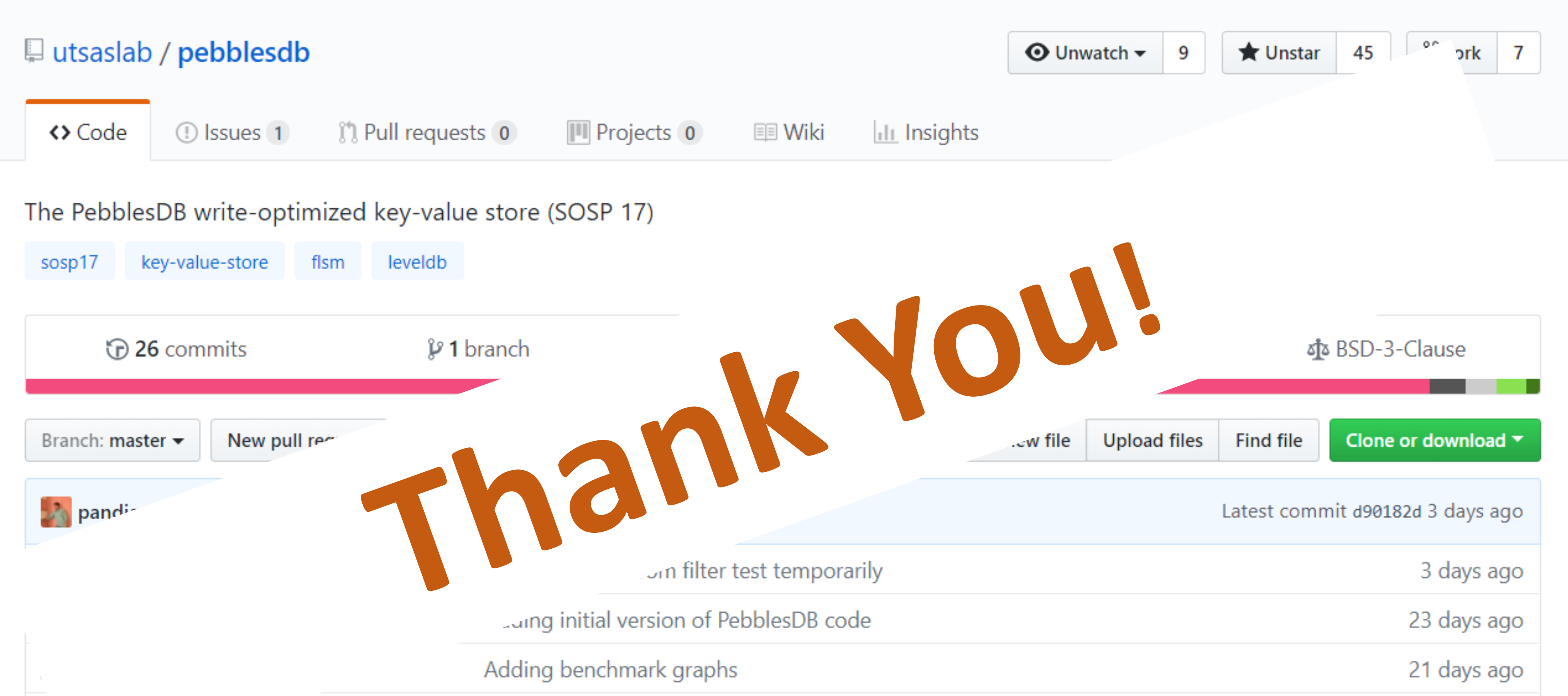


vmware®

Systems and Storage Lab
UT Austin

TEXAS
The University of Texas at Austin

https://github.com/utsaslab/pebblesdb



Thank You!

vmware

Systems and Storage Lab
UT Austin

TEXAS
The University of Texas at Austin

Backup slides

Operations: Seek

- **Seek(target):** Returns the smallest key in the database which is >= target
- Used for range queries (for example, return all entries between 5 and 18)

Level 0	–	1, 2, 100, 1000
Level 1	–	1, 5, 10, 2000
Level 2	–	5, 300, 500

Get(1)

Operations: Seek

- **Seek(target):** Returns the smallest key in the database which is >= target
- Used for range queries (for example, return all entries between 5 and 18)

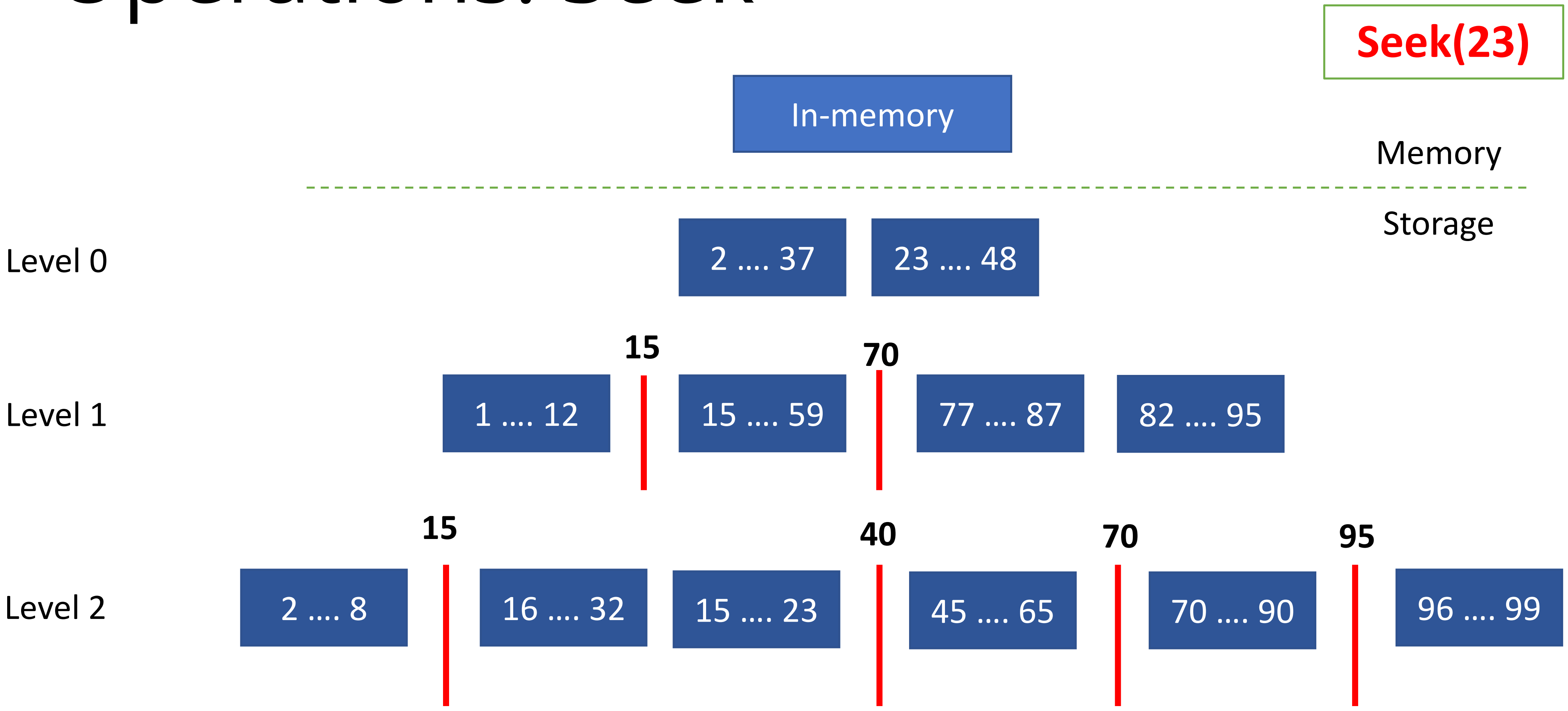
Level 0	–	1, 2, 100, 1000
Level 1	–	1, 5, 10, 2000
Level 2	–	5, 300, 500

Seek(200)

Operations: Seek

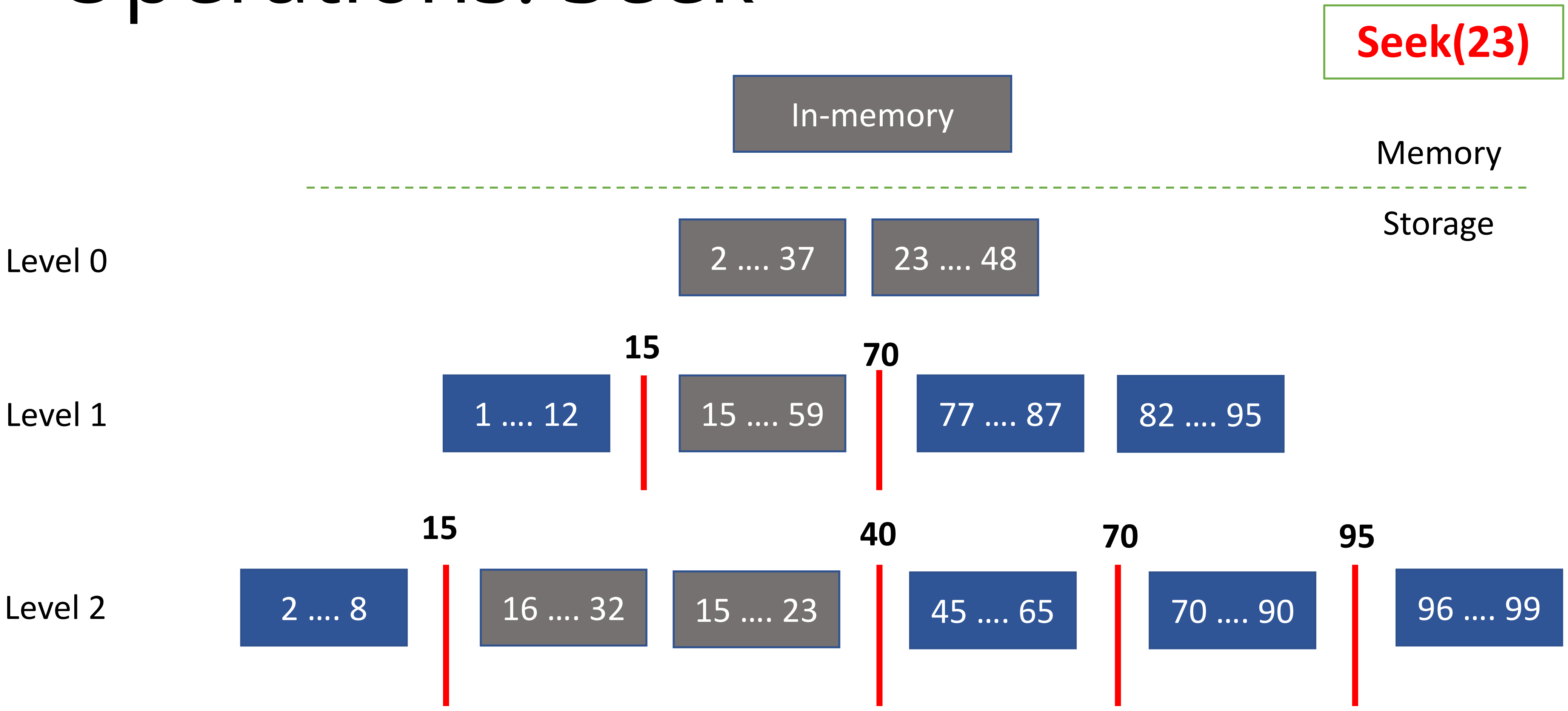
- **Seek(target):** Returns the smallest key in the database which is >= target
- Used for range queries (for example, return all entries between 5 and 18)

Operations: Seek



FLSM structure

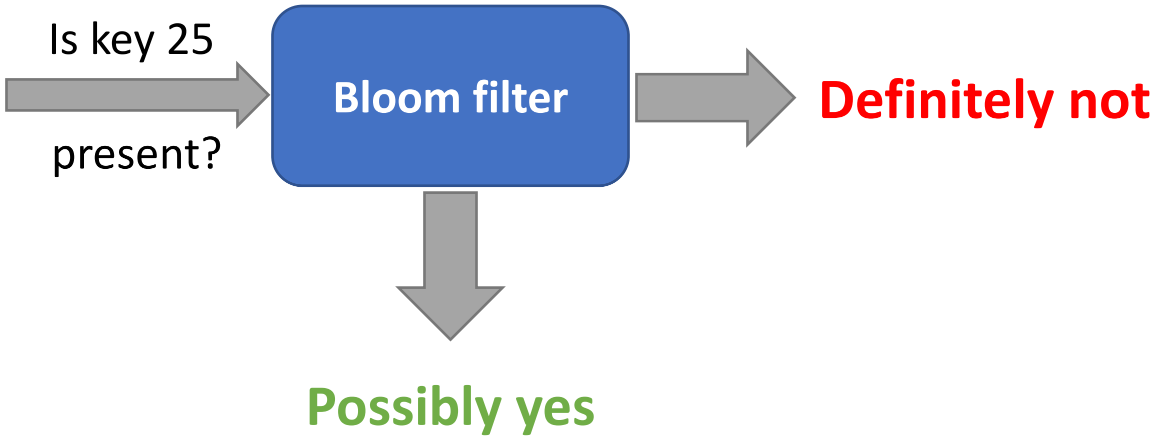
Operations: Seek



All levels and memtable need to be searched

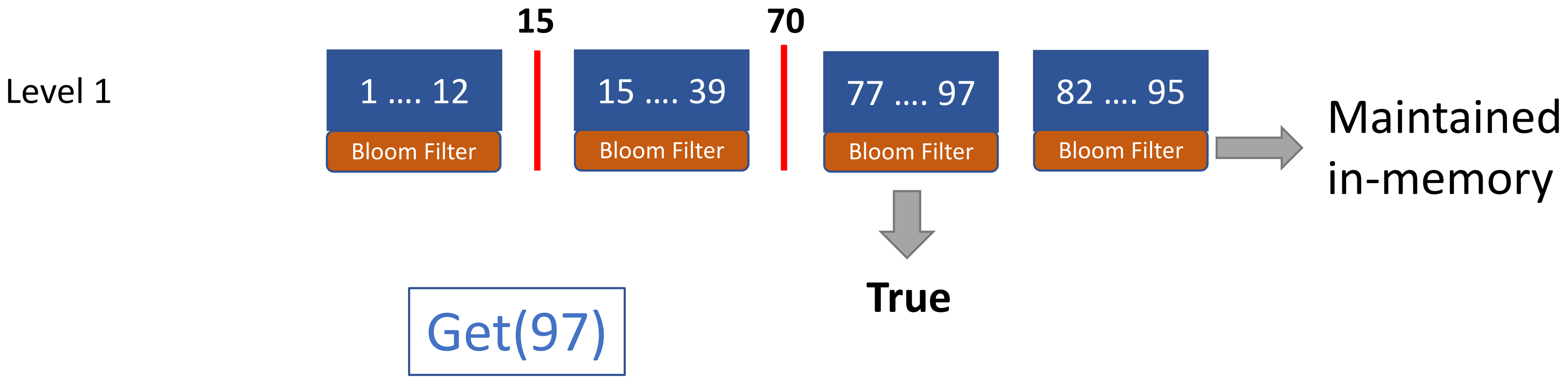
Optimizations in PebblesDB

- **Challenge with reads:** Multiple sstable reads per level
- Optimized using sstable level bloom filters
- Bloom filter: determine if an element is in a set



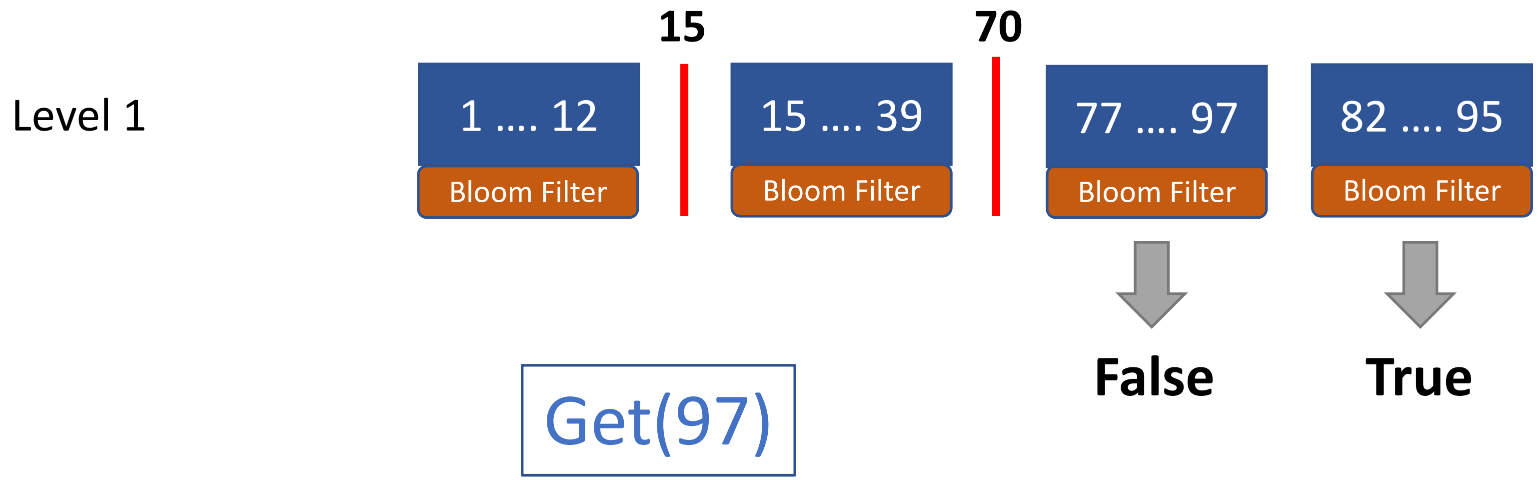
Optimizations in PebblesDB

- **Challenge with reads:** Multiple sstable reads per level
- Optimized using sstable level bloom filters
- Bloom filter: determine if an element is in a set



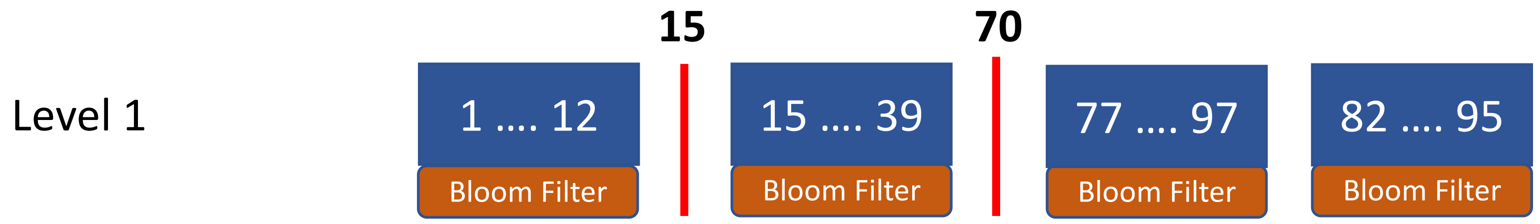
Optimizations in PebblesDB

- **Challenge with reads:** Multiple sstable reads per level
- Optimized using sstable level bloom filters
- Bloom filter: determine if an element is in a set



Optimizations in PebblesDB

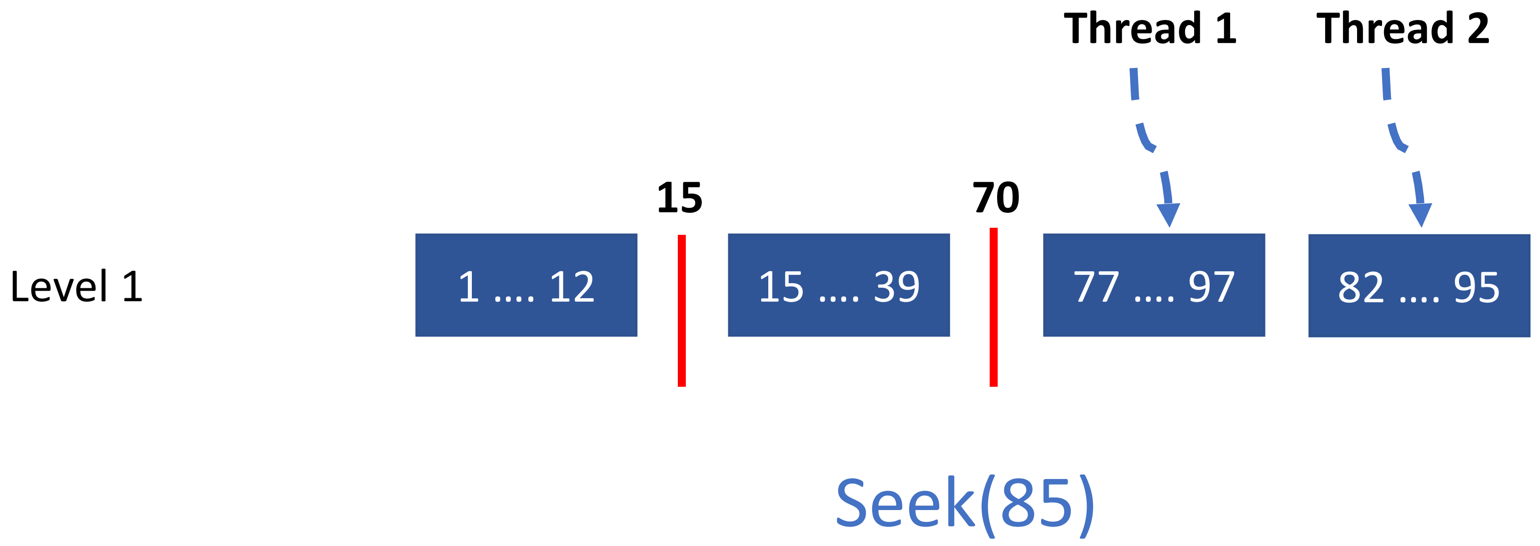
- **Challenge with reads:** Multiple sstable reads per level
- Optimized using sstable level bloom filters
- Bloom filter: determine if an element is in a set



PebblesDB reads at most one file per guard with high probability

Optimizations in PebblesDB

- **Challenge with seeks:** Multiple sstable reads per level
- **Parallel seeks:** Parallel threads to seek() on files in a guard



Optimizations in PebblesDB

- **Challenge with seeks:** Multiple sstable reads per level
- **Parallel seeks:** Parallel threads to seek() on files in a guard
- **Seek based compaction:** Triggers compaction for a level during a seek-heavy workload
  - Reduce the average number of sstables per guard
  - Reduce the number of active levels

Seek based compaction increases write I/O but as a trade-off to improve seek performance

Tuning PebblesDB

- PebblesDB characteristics like
  - Increase in write throughput,
  - decrease in write amplification and
  - overhead of read/seek operation

  all depend on one parameter, **maxFilesPerGuard** (default 2 in PebblesDB)
- Setting this to a very high value favors write throughput
- Setting this to a very low value favors read throughput

Horizontal compaction

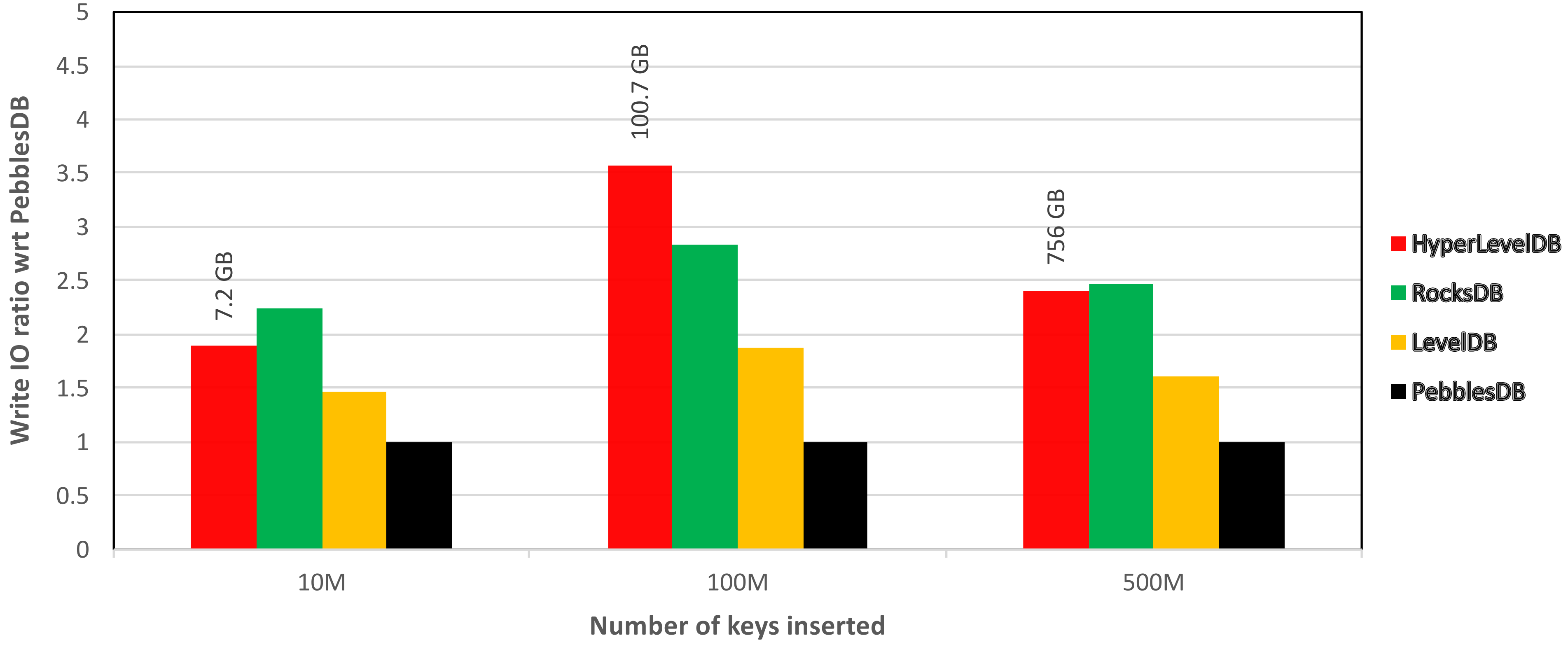
- Files compacted within the same level for the last two levels in PebblesDB
- Some optimizations to prevent huge increase in write IO

Experimental setup

- Intel Xeon 2.8 GHz processor
- 16 GB RAM
- Running Ubuntu 16.04 LTS with the Linux 4.4 kernel
- Software RAID0 over 2 Intel 750 SSDs (1.2 TB each)
- Datasets in experiments 3x bigger than DRAM size

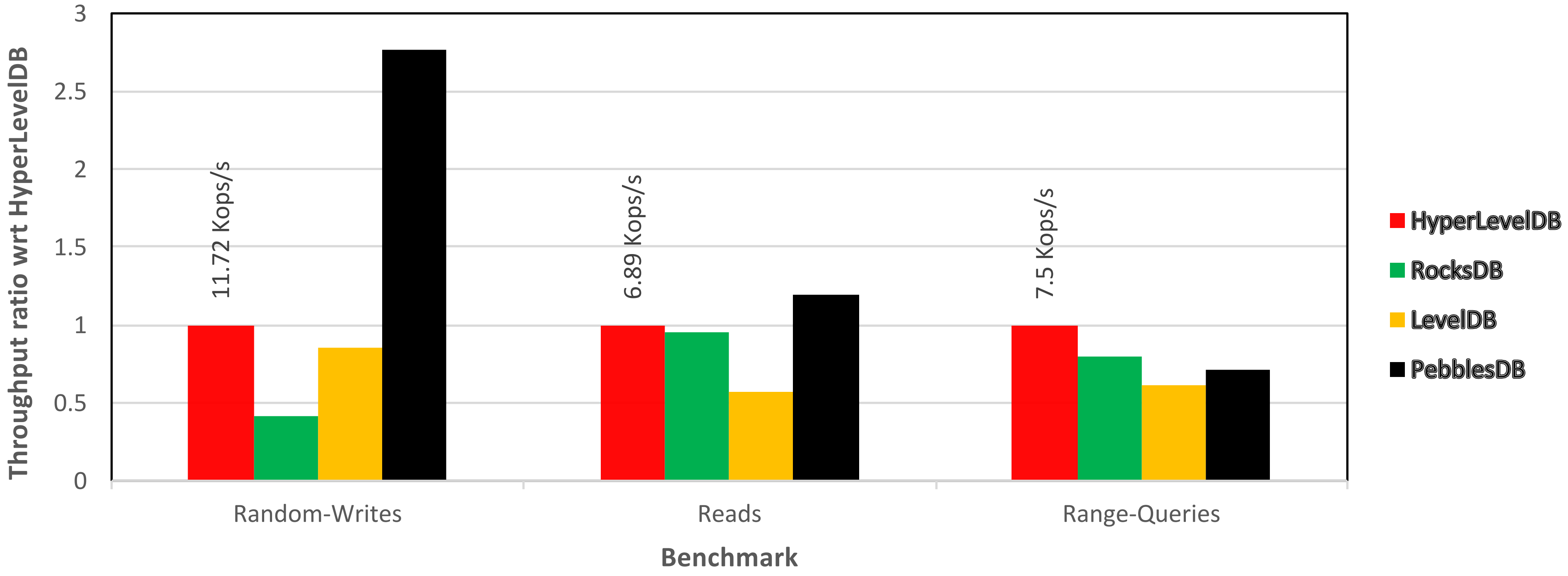
Write amplification

- Inserted different number of keys with key size 16 bytes and value size 128 bytes



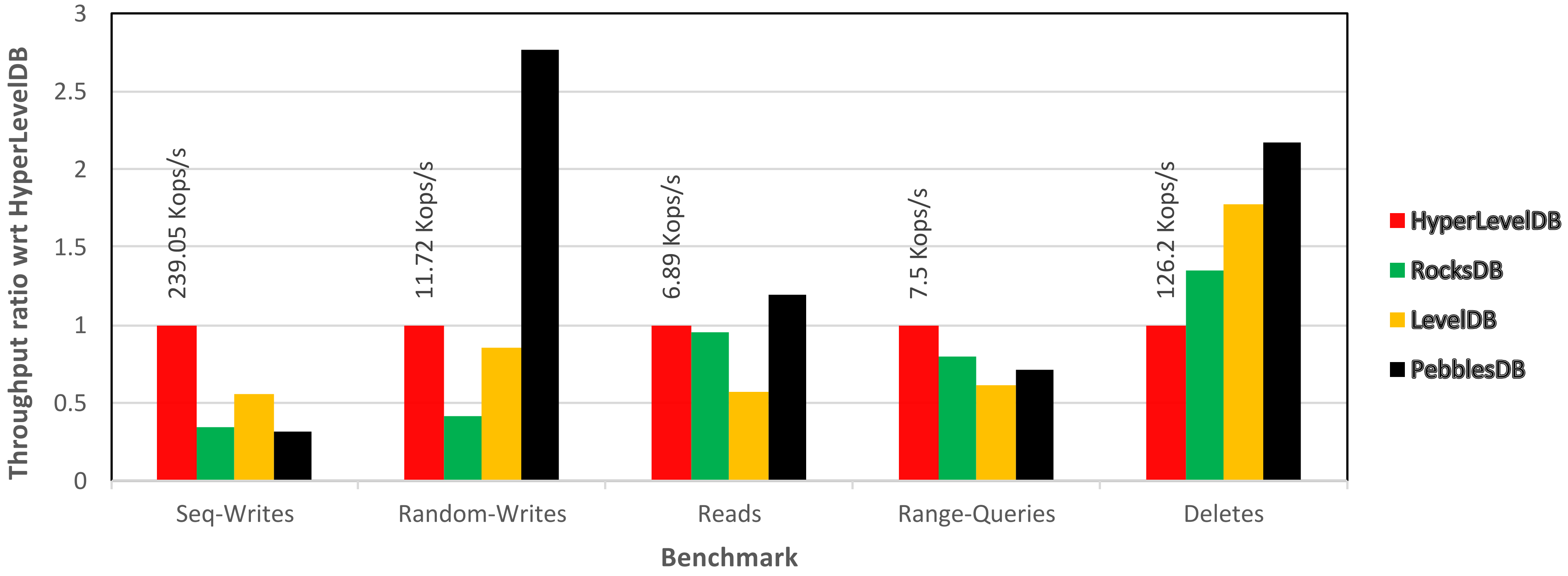
Micro-benchmarks

- Used **db_bench** tool that ships with LevelDB
- Inserted 50M key-value pairs with key size 16 bytes and value size 1 KB
- Number of read/seek operations: 10M



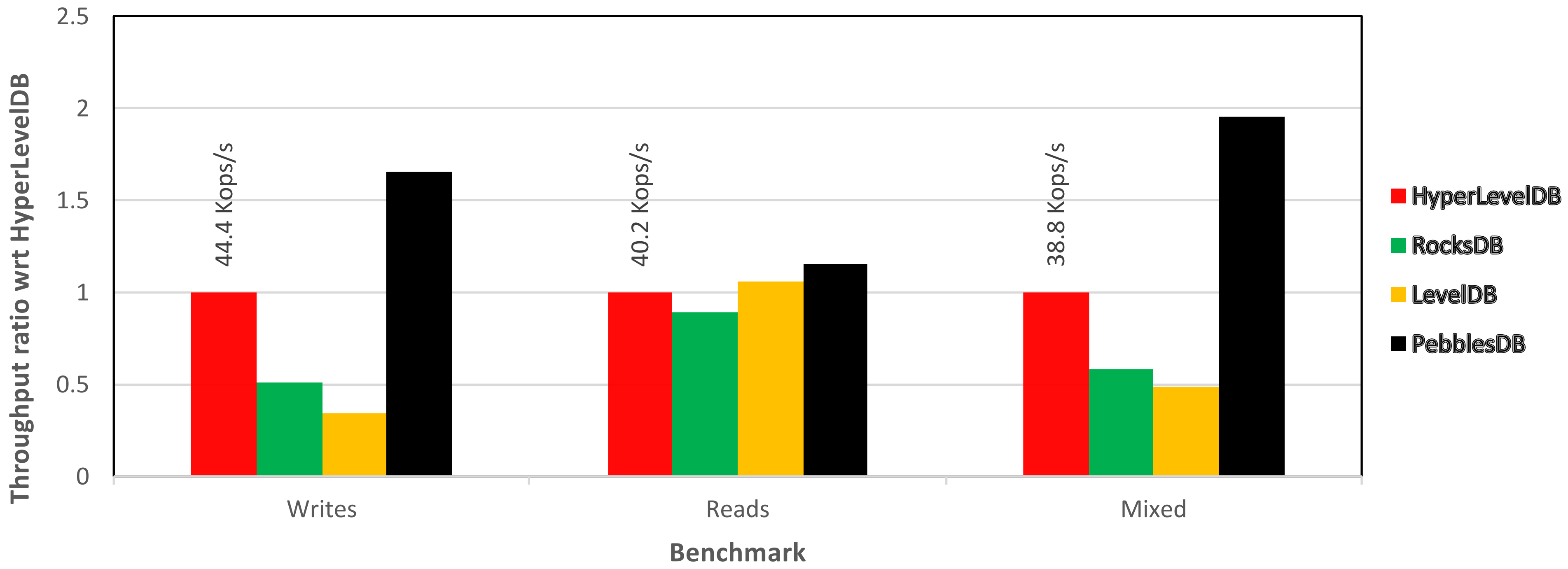
Micro-benchmarks

- Used **db_bench** tool that ships with LevelDB
- Inserted 50M key-value pairs with key size 16 bytes and value size 1 KB
- Number of read/seek operations: 10M



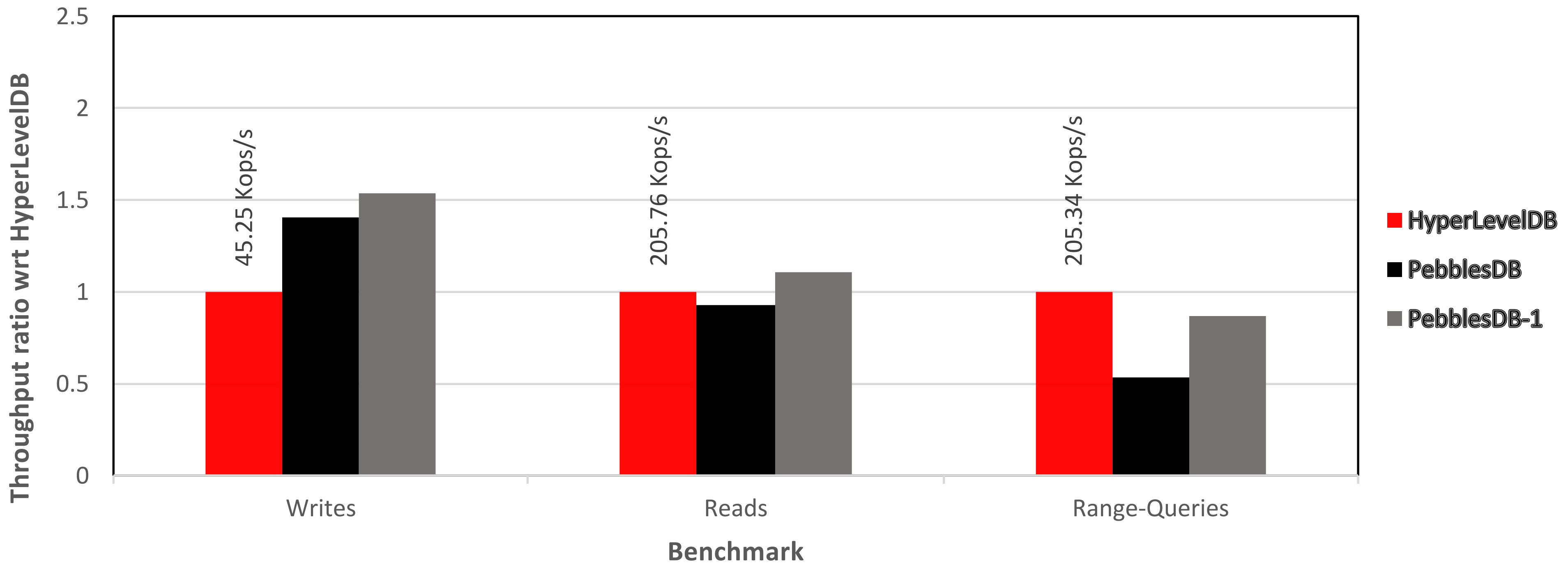
Multi threaded micro-benchmarks

- Writes – 4 threads each writing 10M
- Reads – 4 threads each reading 10M
- Mixed – 2 threads writing and 2 threads reading (each 10M)



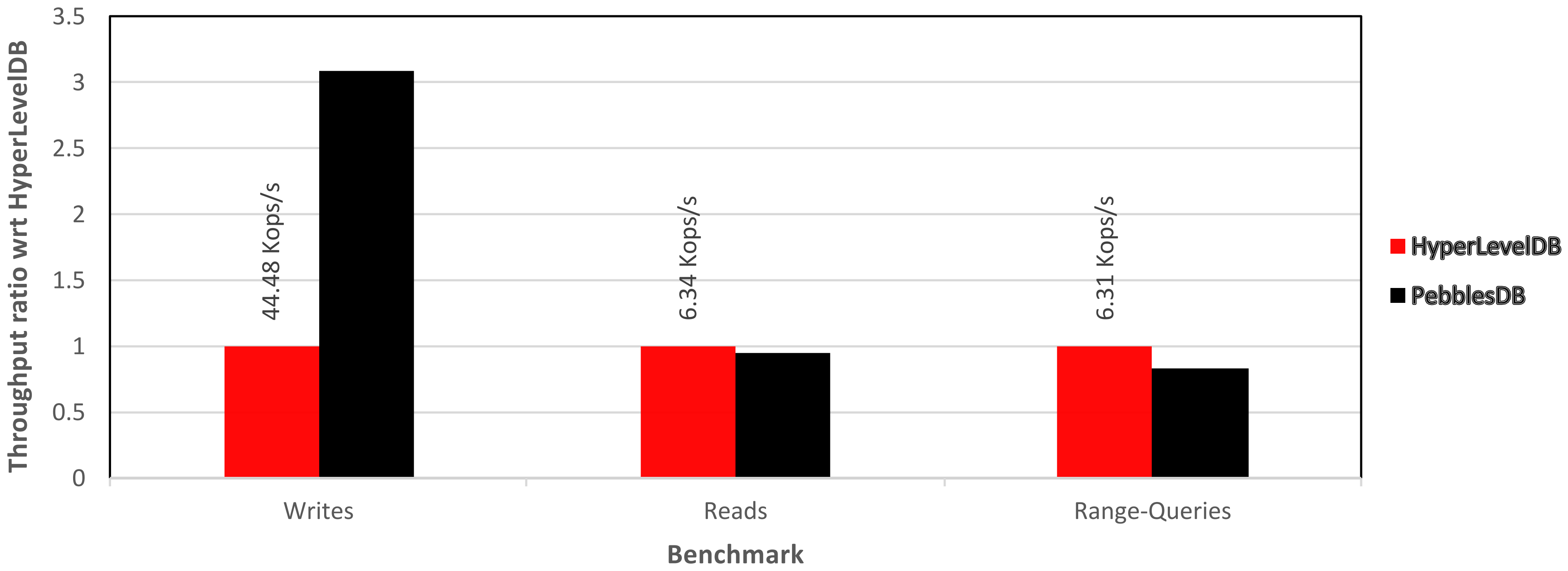
Small cached dataset

- Insert 1M key-value pairs with 16 bytes key and 1 KB value
- Total data set (~1 GB) fits within memory
- PebblesDB-1: with maximum one file per guard



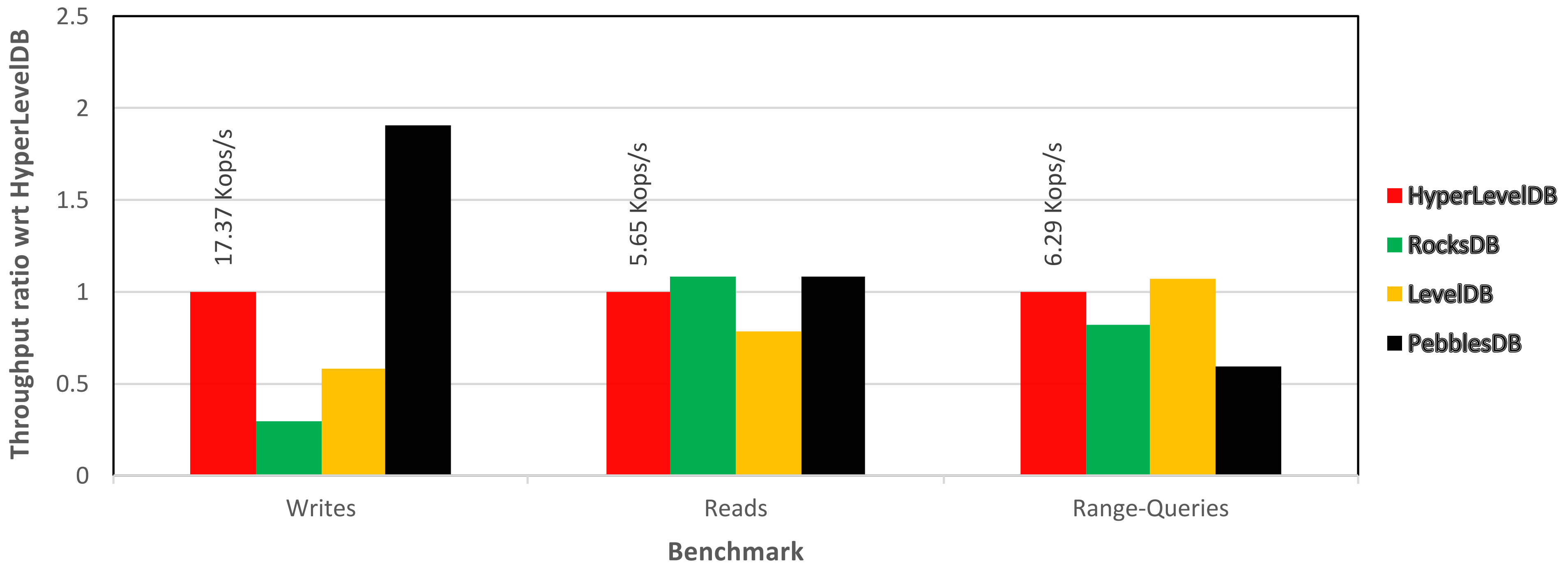
Small key-value pairs

- Inserted 300M key-value pairs
- Key 16 bytes and 128 bytes value



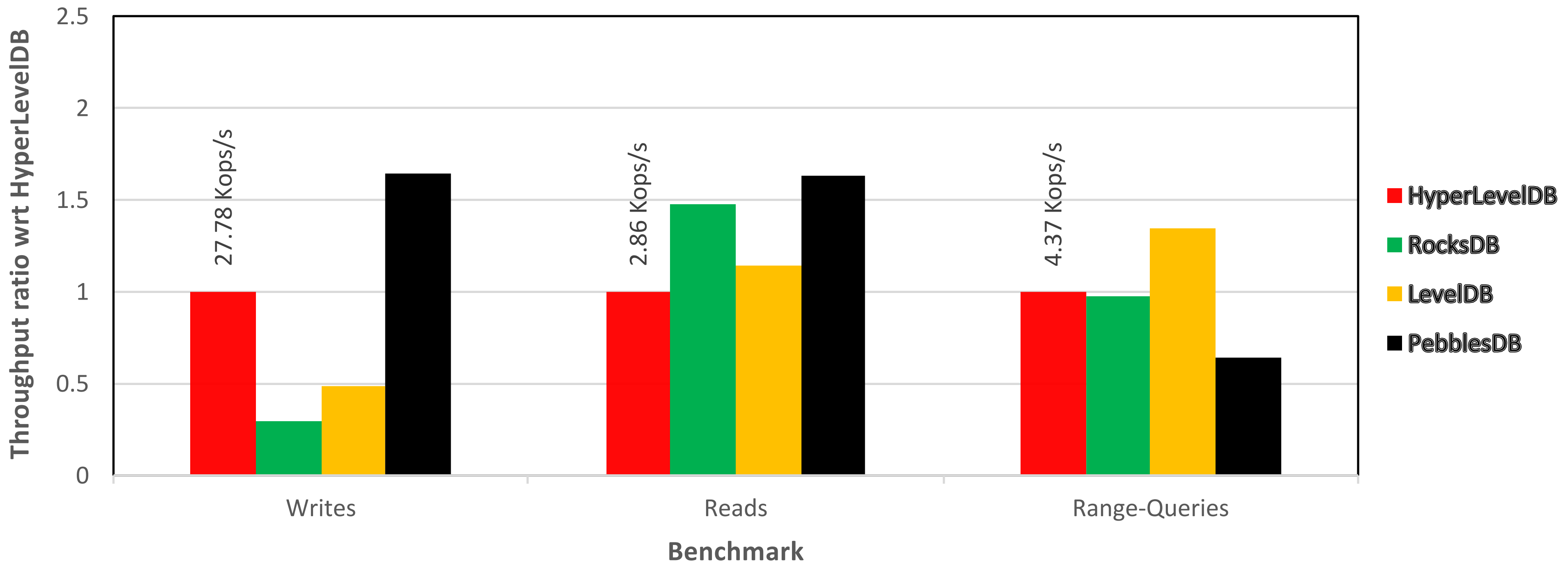
Aged FS and KV store

- File system aging: Fill up 89% of the file system
- KV store aging: Insert 50M, delete 20M and update 20M key-value pairs in random order



Low memory micro-benchmark

- 100M key-value pairs with 1KB (~65 GB data set)
- DRAM was limited to 4 GB

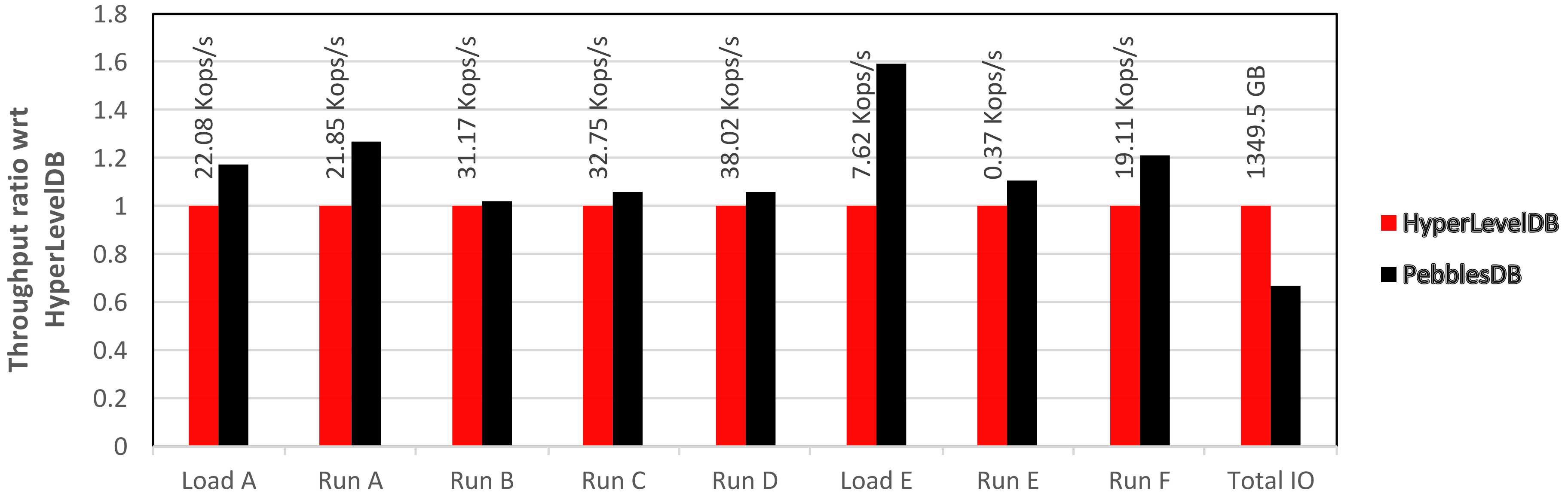


Impact of empty guards

- Inserted 20M key-value pairs (0 to 20M) in random order with value size 512 bytes
- Incrementally inserted new 20M keys after deleting the older keys
- Around 9000 empty guards at the start of the last iteration
- Read latency did not reduce with the increase in empty guards

NoSQL stores - HyperDex

- HyperDex – distributed key-value store from Cornell
- Inserted 20M key-value pairs with 1 KB value size and 10M operations



Load A - 100 % writes	Run D - 95% reads (latest), 5% writes
Run A - 50% reads, 50% writes	Load E - 100% writes
Run B - 95% reads, 5% writes	Run E - 95% range queries, 5% writes
Run C - 100% reads	Run F - 50% reads, 50% read-modify-writes

CPU usage

- Median CPU usage by inserting 30M keys and reading 10M keys
- PebblesDB: ~171%
- Other key-value stores: 98-110%
- Due to aggressive compaction, more CPU operations due to merging multiple files in a guard

Memory usage

- 100M records (16 bytes key, 1 KB value) – 106 GB data set
  - 300 MB memory space
  - 0.3% of data set size
- Worst case: 100M records (16 bytes key, 16 bytes value) ~3.2 GB
  - 9% of data set size

Bloom filter calculation cost

- 1.2 sec per GB of sstable
- 3200 files – 52 GB – 62 seconds

Impact of different optimizations

- Sstable level bloom filter improve read performance by 63%
- PebblesDB without optimizations for seek – 66%

Thank you!

Questions?