香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

Lecture 09:
Persistent Key-Value Stores

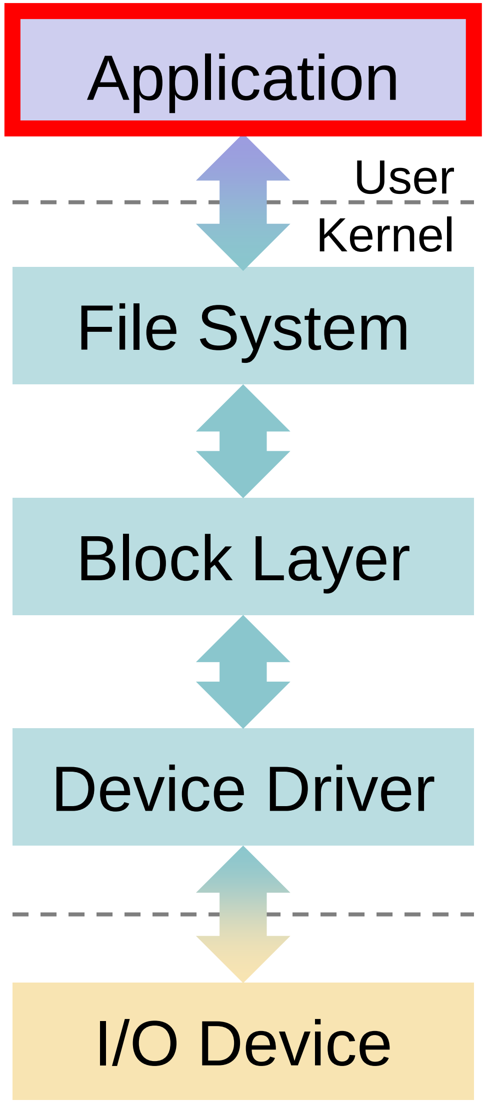
Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Outline

- Persistent Key-Value Store
  - Log-Structured Merge-Tree (LSM-tree)
- LevelDB (by Google)
  - Insertion and Compaction
  - Lookup
- WiscKey: Separating Keys from Values
  - Write and Read Amplification
  - Key-Value Separation
  - Benefits and Challenges
- Single-Level KV Store with PM
  - Single-Level Merge
  - Selective Compaction

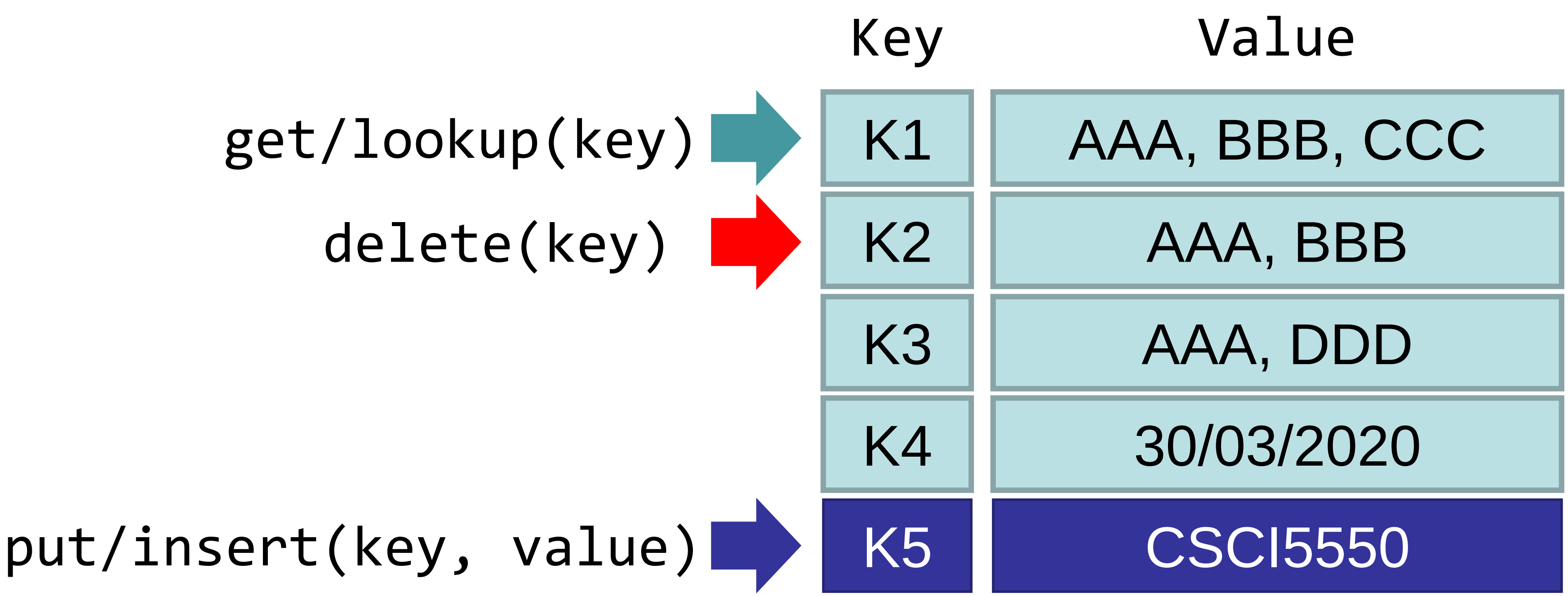
I/O Stack



The Log-Structured Merge-Tree (Acta Informatica'96)

Persistent Key-Value Store

- **Persistent key-value (KV) stores** play a critical role in a variety of modern data-intensive applications:
  - Such as e-commerce, cloud data, and social networking.
- In a KV store, data are stored as key-value pairs.
  - A unique `key` is associated with a `value` of "any form".



Log-Structured Merge-Tree (LSM-Tree)

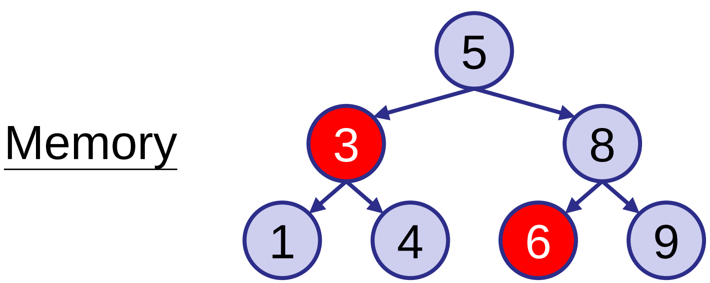
- For write-intensive workloads, KV stores based on **LSM-tree** have become the state of the art.
  - Various distributed or local stores built on LSM-trees are widely deployed in largescale environments, such as:
    - BigTable and LevelDB at **Google**;
    - Cassandra, Hbase, and RocksDB at **Facebook**; and
    - PNUTS at **Yahoo!**

- The main advantage of LSM-trees is that they maintain sequential access patterns for writes.
  - The success of LSM-tree is tied closely to its usage upon classic hard-disk drives (HDDs): In which, random I/Os are over 100× slower than sequential ones.

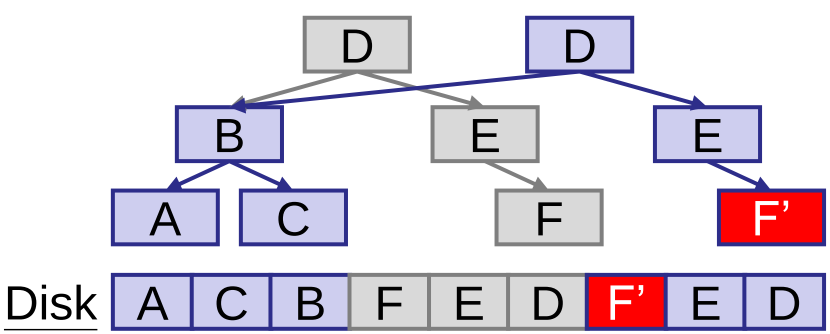
Overall Architecture of LSM-Tree

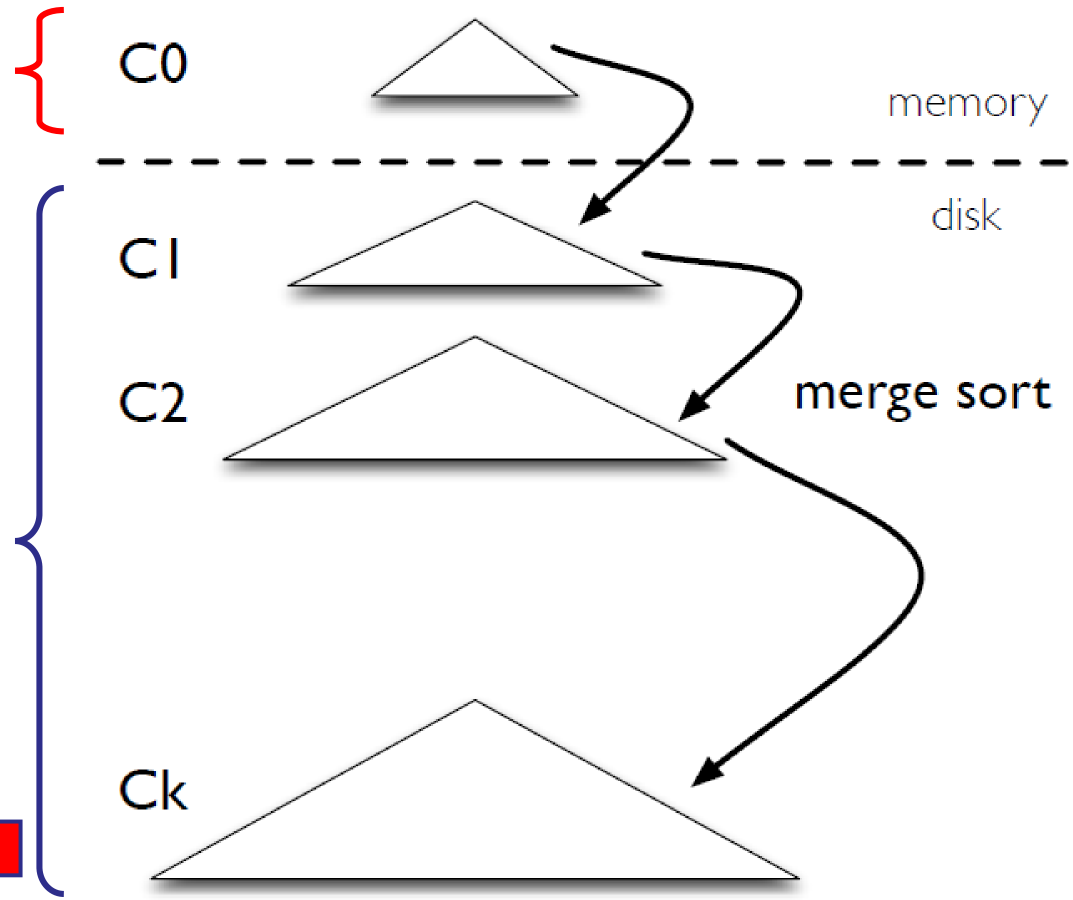
- An **LSM-tree** consists of a number of components of exponentially increasing sizes, **`C0`** to **`Ck`**:

`C0` is a memory-resident, update-in-place sorted tree.



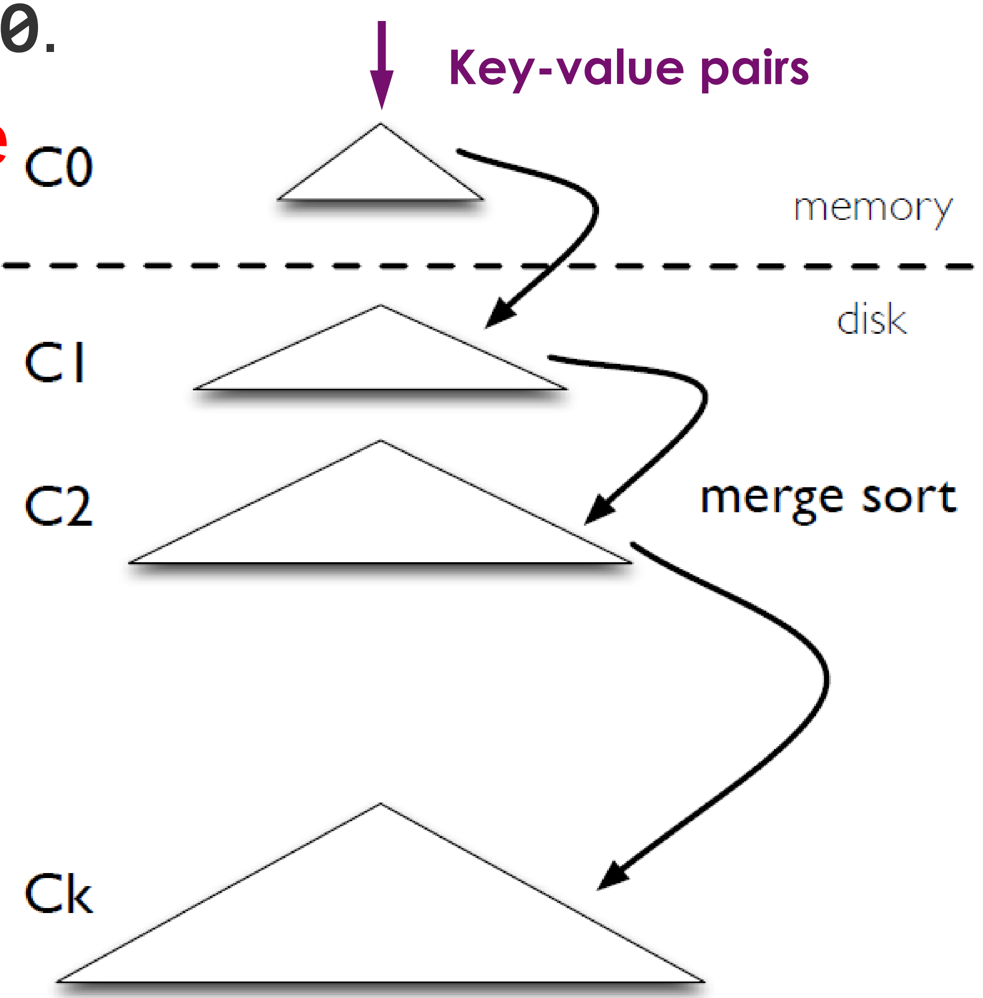
`C1~Ck` are disk-resident, append-only B-trees.





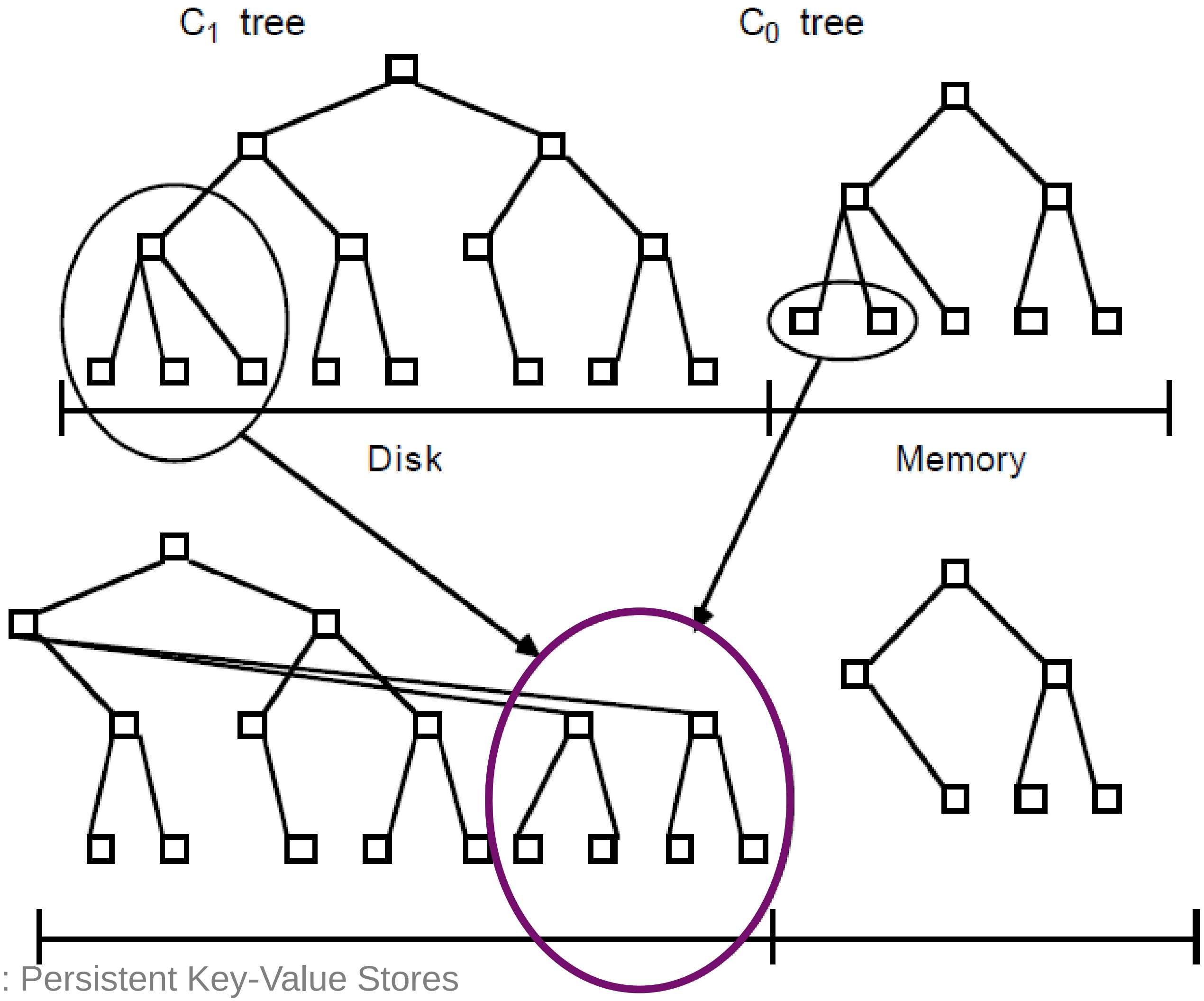
LSM-Tree: Insertion & Compaction (1/2)

- Key-value pairs are always **inserted** into the LSM-tree via the in-memory `C0`.
- Once `C0` reaches its size limit, `C0` will be merged with the on-disk `C1` by the compaction process.
  - The newly merged tree `C1'` will be appended into disk, replacing the old `C1`.
- Compaction also takes place for all on-disk components, when any `Ci` reaches its size limit.



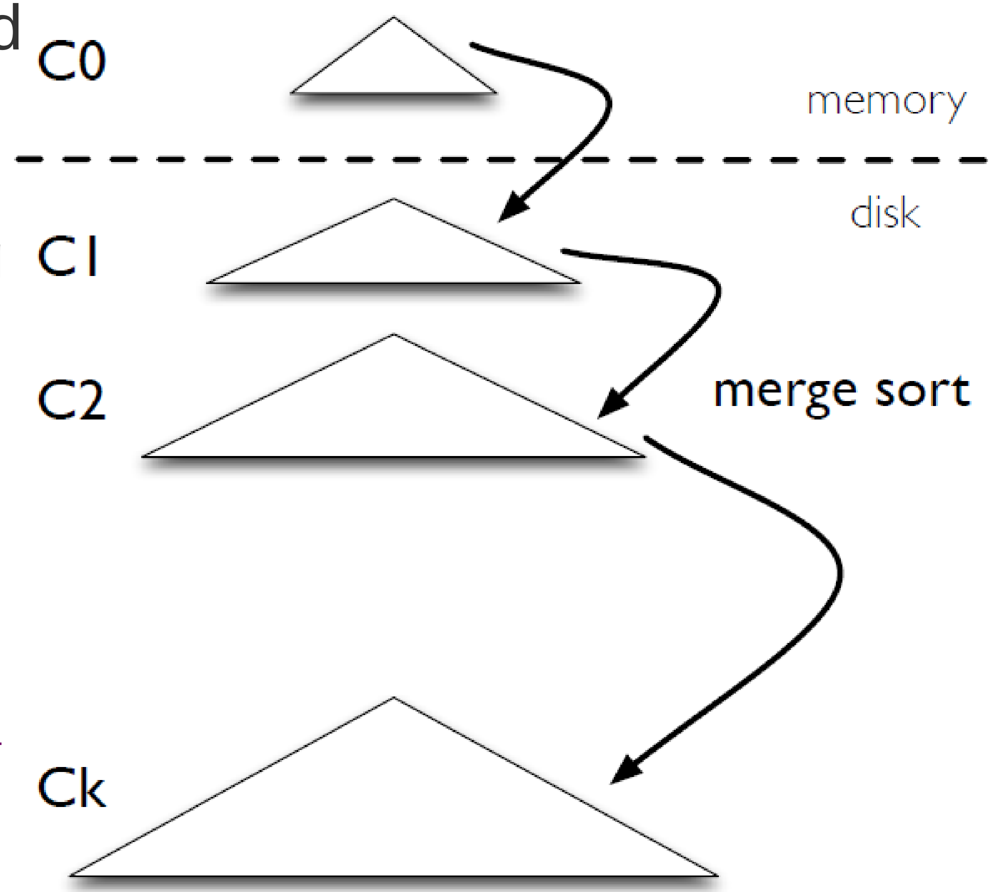
LSM-Tree: Insertion & Compaction (2/2)

- During the compaction, the newly merged blocks are written to **new** disk positions.



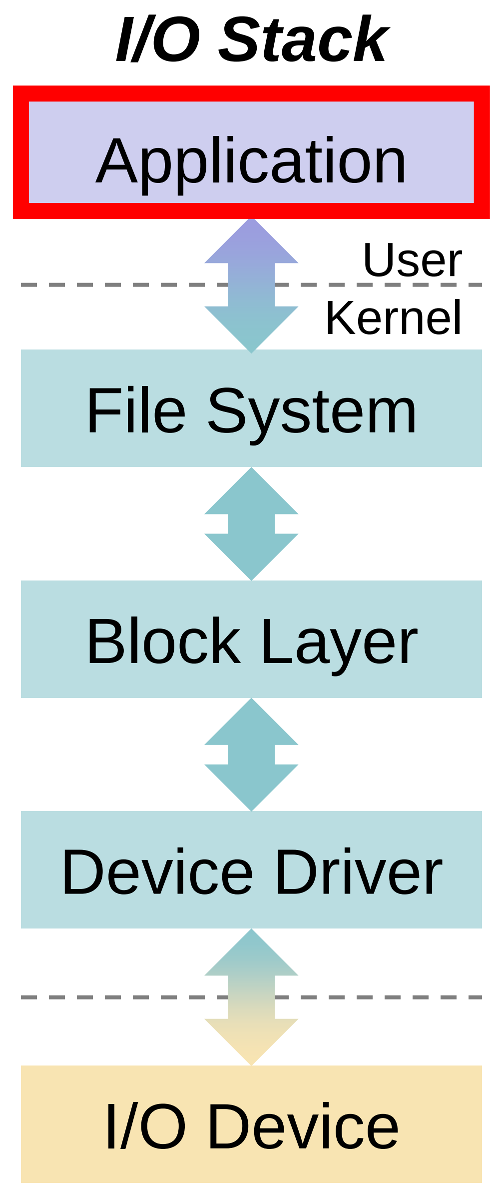
LSM-Tree: Lookup

- To serve a **lookup** operation, LSM-trees may need to search over multiple components.
  - Components are scanned in a cascading fashion, from **`C0`** to the smallest component **`Ci`** containing the requested data.
    - Why? **`C0`** contains the freshest data, followed by **`C1`**, and so on.
  - Hence LSM-trees are more useful when <u>inserts are more dominant than lookups</u>.



Outline

- Persistent Key-Value Store
  - Log-Structured Merge-Tree (LSM-tree)
- LevelDB (by Google)
  - Insertion and Compaction
  - Lookup
- WiscKey: Separating Keys from Values
  - Write and Read Amplification
  - Key-Value Separation
  - Benefits and Challenges
- Single-Level KV Store with PM
  - Single-Level Merge
  - Selective Compaction



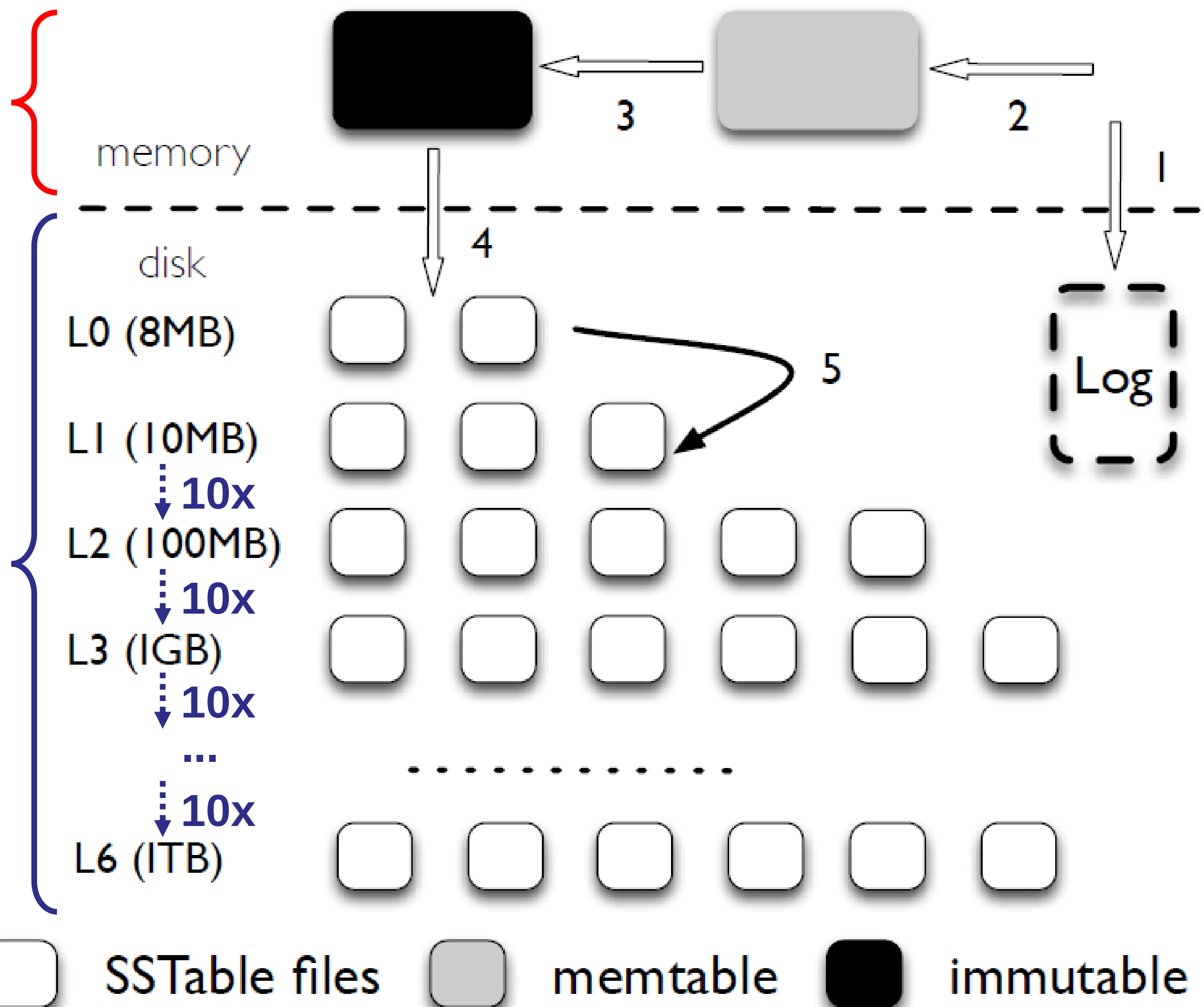
https://github.com/google/leveldb

LevelDB (by Google)

- **LevelDB** is a key-value store based on LSM-trees.

2 in-memory **sorted skiplists** (i.e., memtable and immutable memtable)

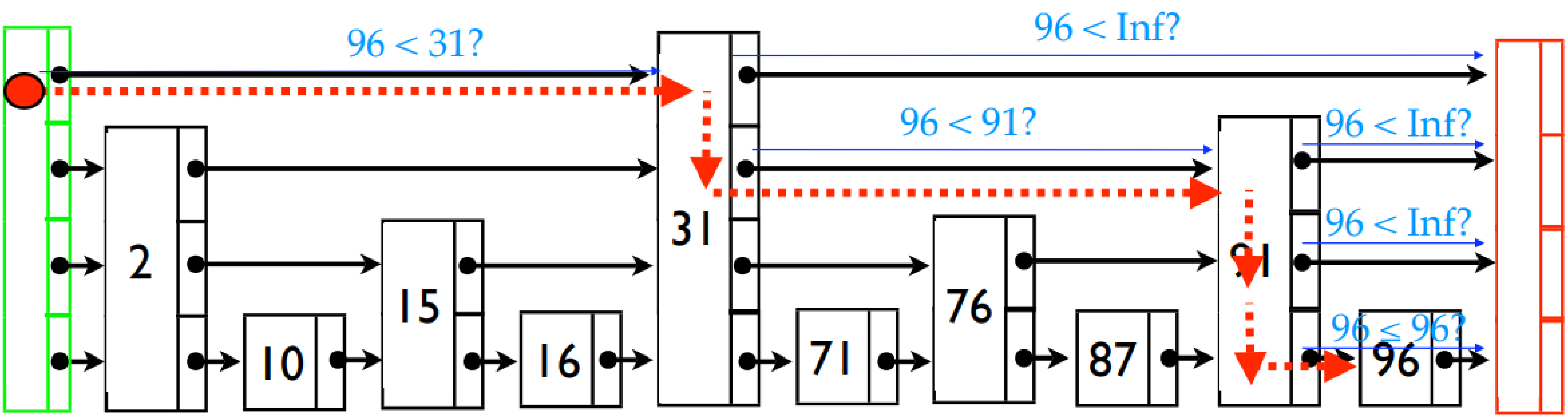
7 "levels" (**L0** to **L6**) of on-disk **sorted string tables** (SSTables)



Review: Sorted Skiplist

- A **skip list** is built in multiple layers:
  - The **bottom layer** is an ordinary ordered linked list.
  - The **higher layers** allow you to "skip over" many items when searching over an particular item.

Find 96?

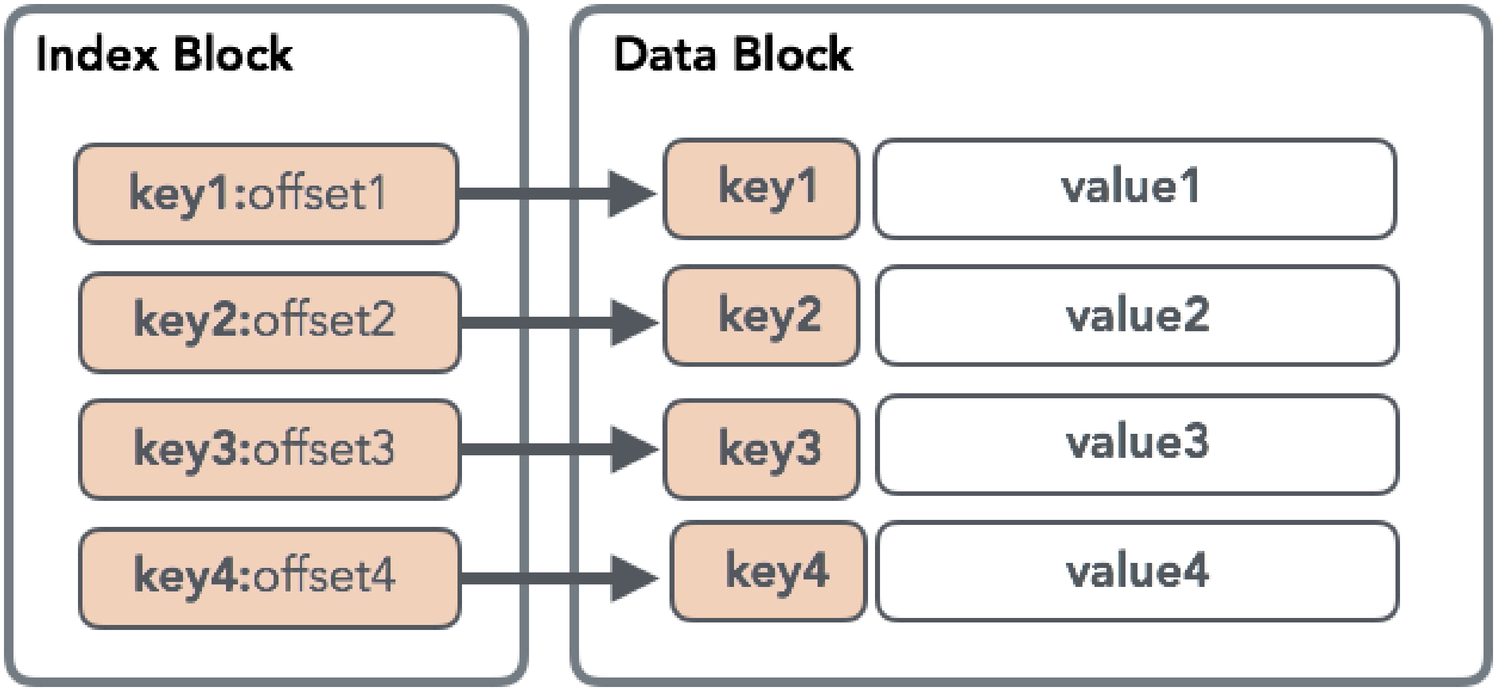


- It offers $O(\log n)$ search complexity and $O(\log n)$ insertion complexity within an ordered sequence of n elements.

https://en.wikipedia.org/wiki/Skip_list
https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf

Review: Sorted String Table

- A **sorted string table** (**SSTable**) is simply a file which contains a set of arbitrary, sorted key-value pairs.



  - **Strength**: High throughput for sequential I/O workloads
  - **Weakness**: Large I/O rewrite for random insert/deletion

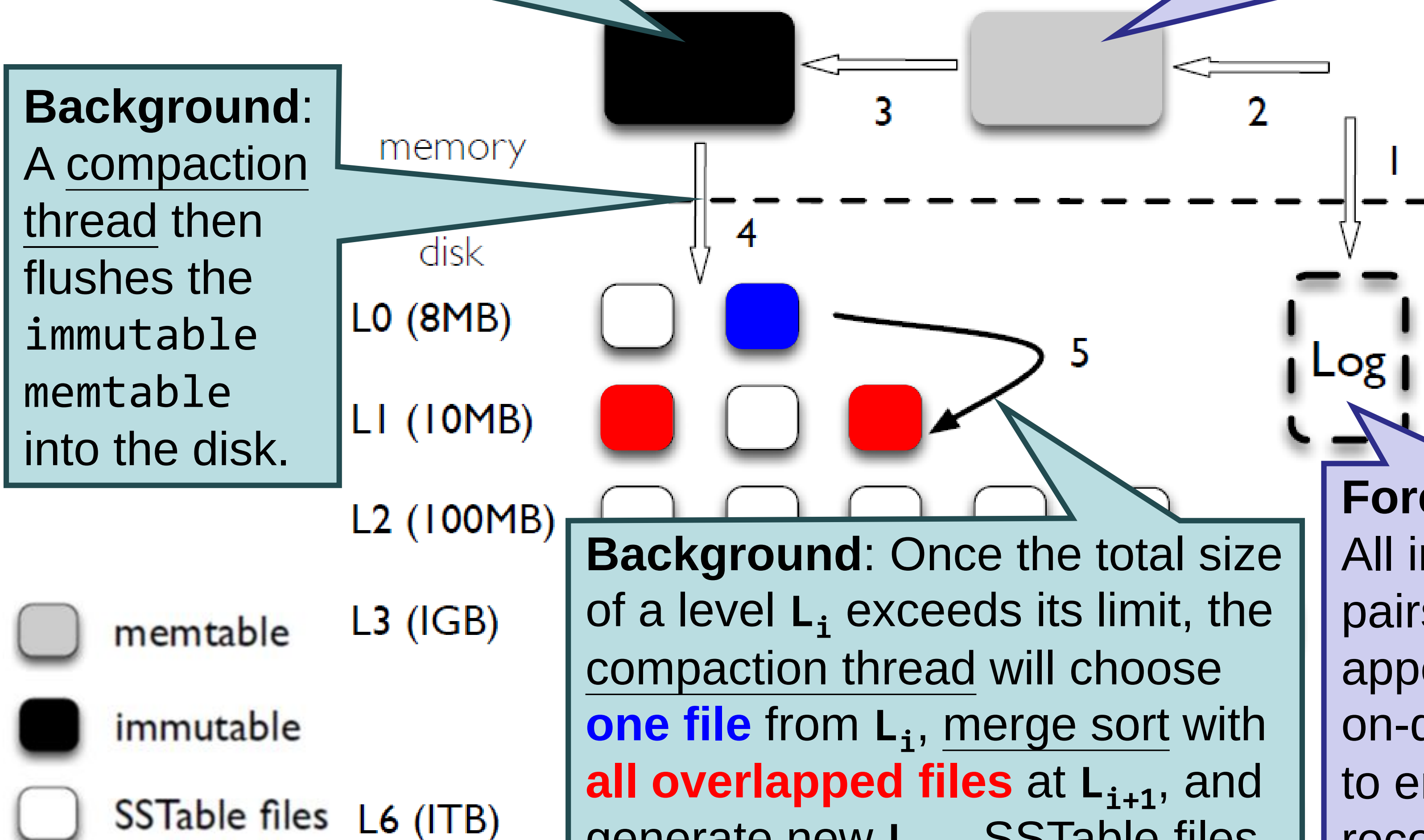
https://medium.com/databasss/on-disk-io-part-3-lsm-trees-8b2da218496f

LevelDB: Insertion & Compaction

Background: Once the `memtable` is full, it is converted into an `immutable memtable`.

Foreground: The KV pairs are then inserted into the in-memory `memtable`.

Background: A compaction thread then flushes the `immutable memtable` into the disk.



Background: Once the total size of a level $\mathbf{L_i}$ exceeds its limit, the compaction thread will choose **one file** from $\mathbf{L_i}$, merge sort with **all overlapped files** at $\mathbf{L_{i+1}}$, and generate new $\mathbf{L_{i+1}}$ SSTable files.

Foreground: All inserted KV pairs are first appended to an on-disk `log` file to enable recovery.

LevelDB: Lookup

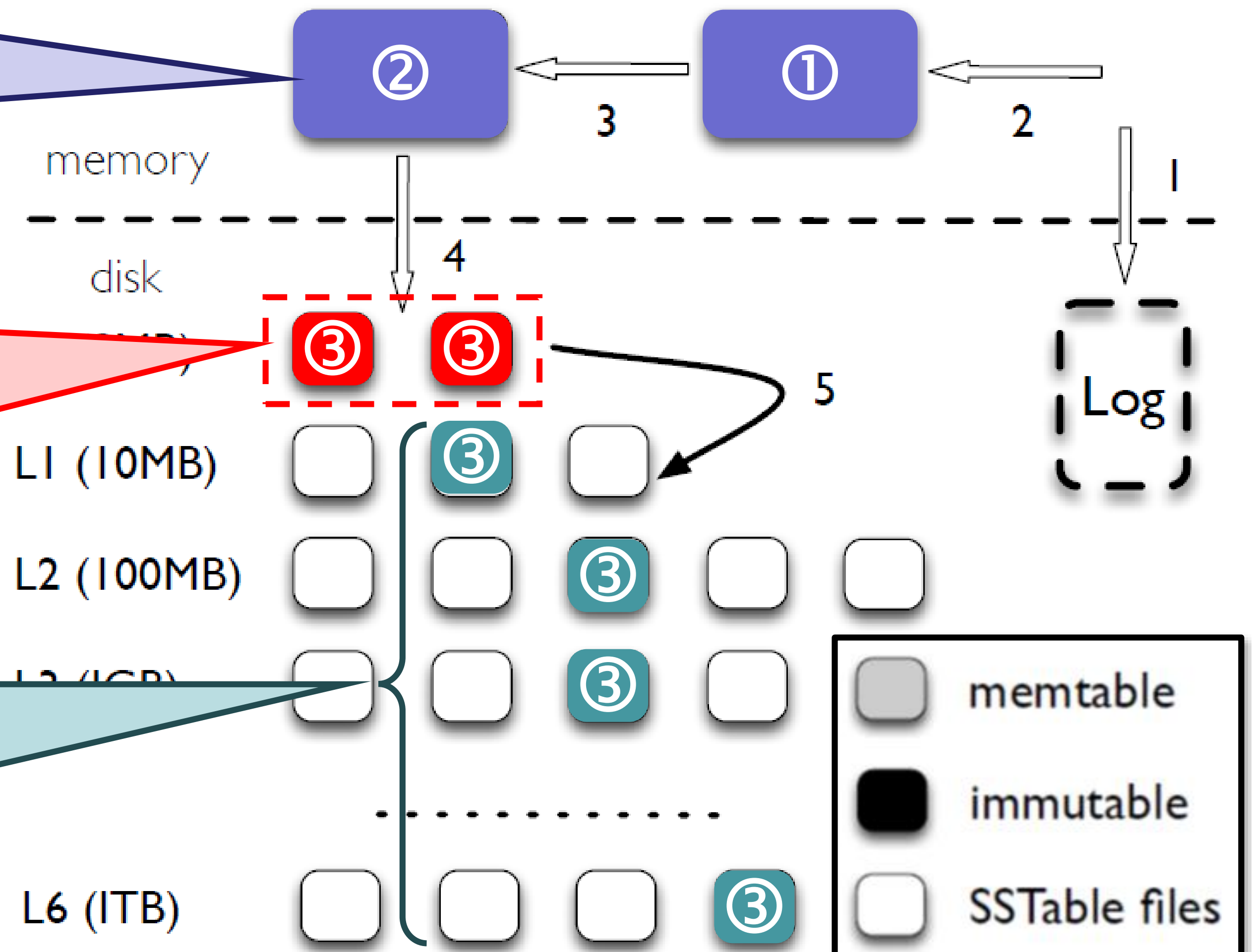
- LevelDB searches for a requested KV pair as follows:

① `memtable`, ② `immutable memtable`, ③ files of `L0` to `L6` in order

The **`memtable`** always contain the freshest data, followed by the **`immutable memtable`**.

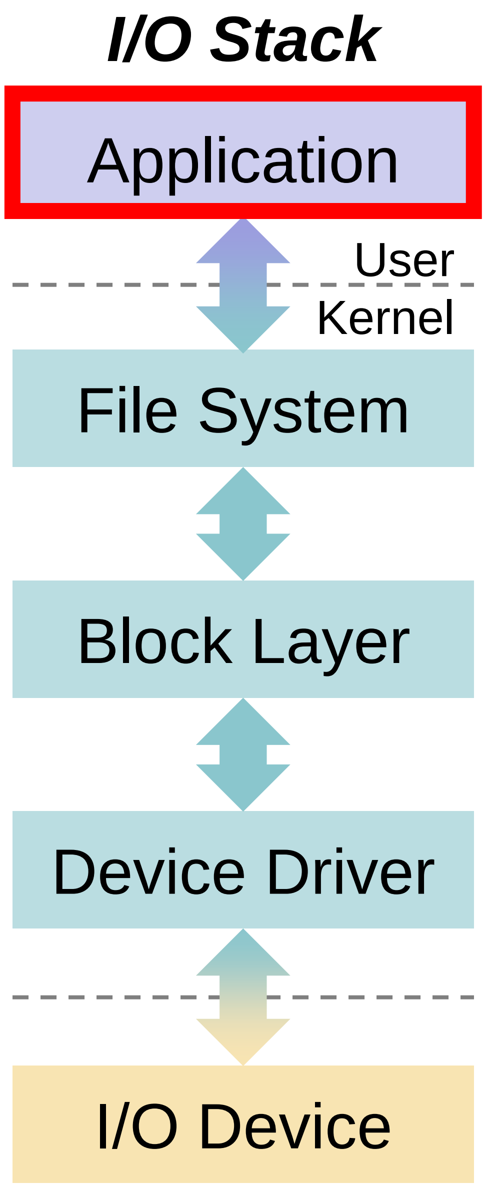
Since LevelDB allows **`SSTable`** files in **`L0`** to contain overlapping keys, multiple files at **`L0`** may be searched.

The total number of file searches can be bounded, since keys do not overlap among files in levels **`L1`** to **`L6`**.



Outline

- Persistent Key-Value Store
  - Log-Structured Merge-Tree (LSM-tree)
- LevelDB (by Google)
  - Insertion and Compaction
  - Lookup
- WiscKey: Separating Keys from Values
  - Write and Read Amplification
  - Key-Value Separation
  - Benefits and Challenges
- Single-Level KV Store with PM
  - Single-Level Merge
  - Selective Compaction



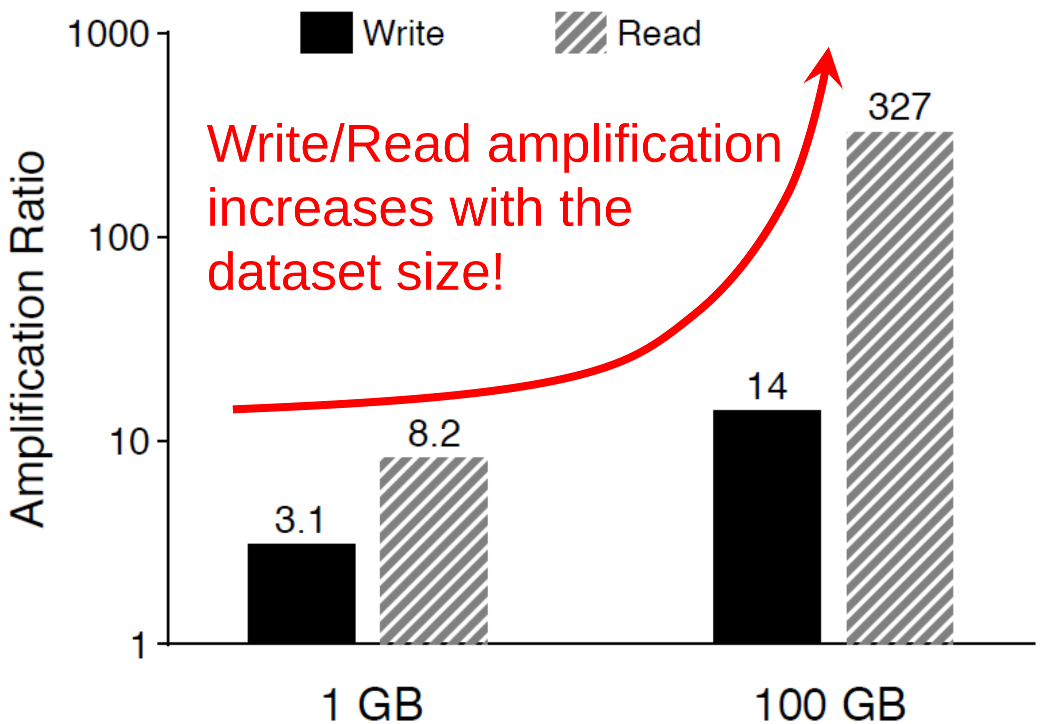
WiscKey - Separating Keys from Values in SSD-conscious Storage (FAST'16)

Write and Read Amplification (1/2)

- **Write** and **read amplification** are major problems in LSM-tree based key-value stores such as LevelDB.
  - **Write (Read) Amplification**: the ratio between the amount of data written to (read from) the storage and the amount of data requested by the user.
- The source of **write amplification** in LevelDB:
  - LevelDB writes more data than necessary to achieve mostly-sequential disk access.
- The sources of **read amplification** in LevelDB:
  - To lookup a key-value pair, LevelDB needs to check multiple SSTable files in multiple levels.
  - To find a key-value pair within a SSTable file, LevelDB needs to read multiple metadata blocks within the file.

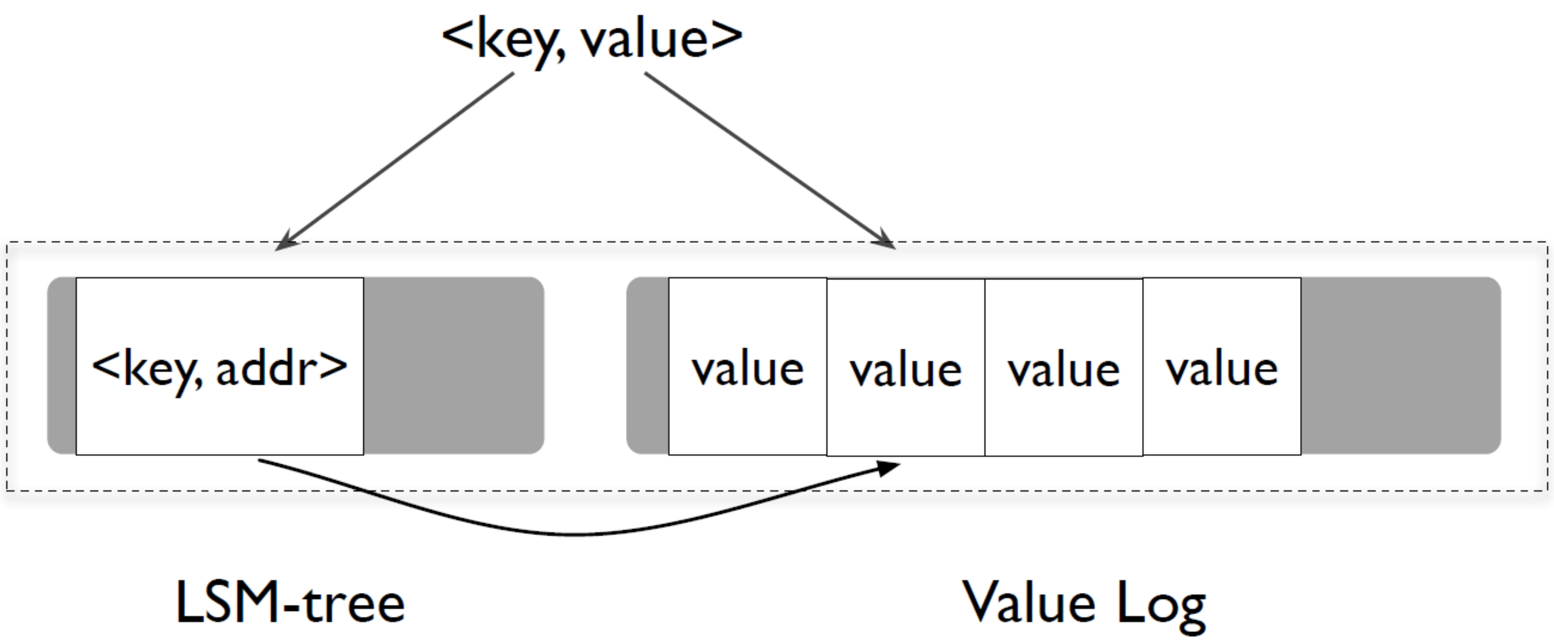
Write and Read Amplification (2/2)

- Experimental Setup:
  - Consider two different database sizes for the initial load
  - Load a database with 16B-key, 1KB-value pairs
  - Lookup 100,000 entries from the database
  - Choose keys randomly from a uniform distribution



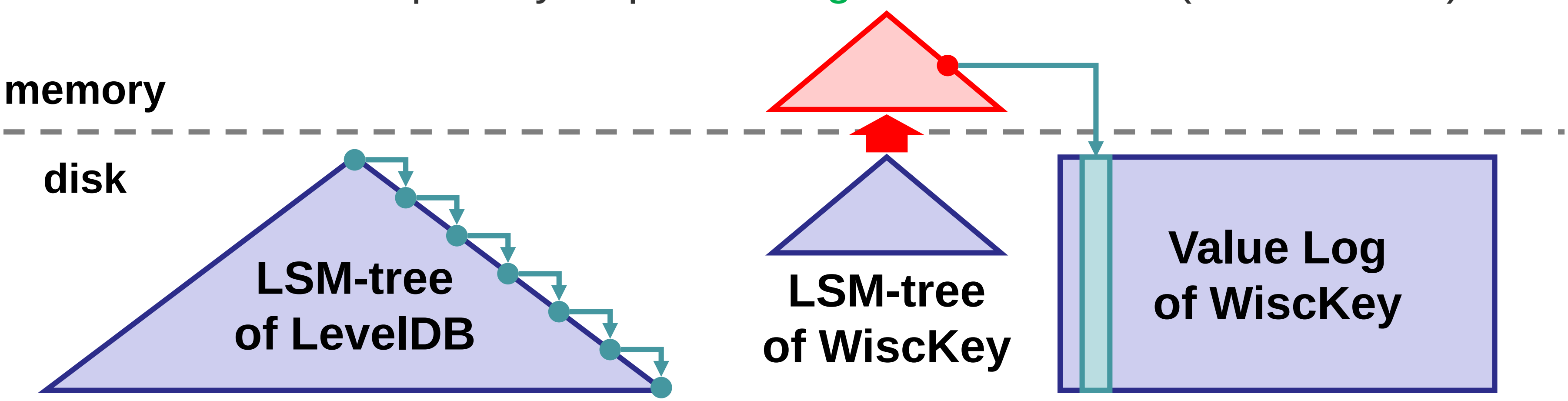
Key-Value Separation

- The major performance cost of LSM-trees is the compaction, which constantly sorts SSTable files.
- **Key-Value Separation**: Compaction only needs to sort keys, while values can be managed separately.
  - Only the “location” (**addr**) of value is stored in the LSM-tree, while real values are stored in a separate value log file.



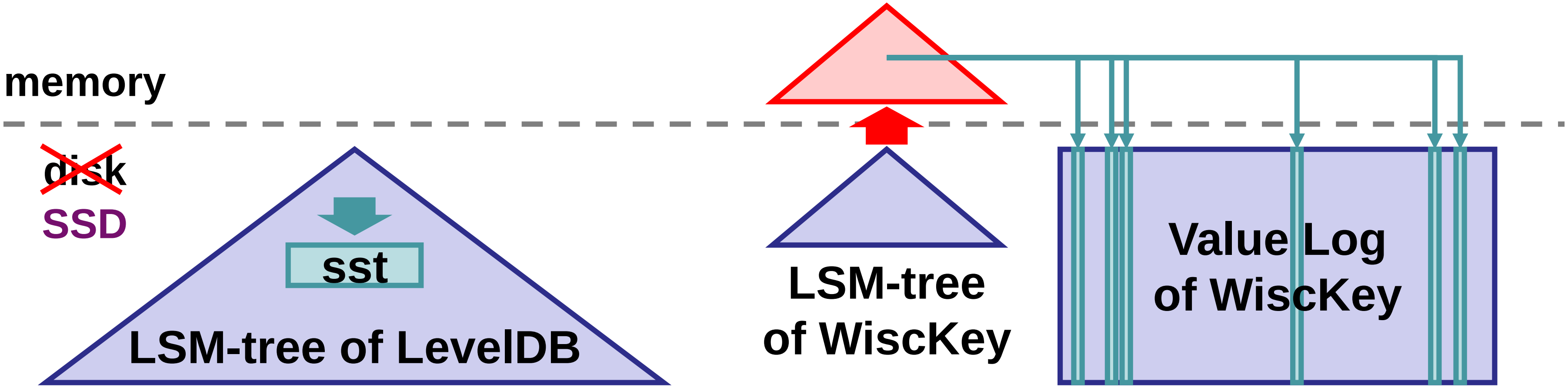
Benefits of Key-Value Separation

- The LSM-tree of WiscKey becomes much smaller than that of LevelDB.
  - Compacting only keys could significantly reduce the **write amplification**, especially for workloads that have a moderately large value size.
  - A significant portion of the LSM-tree can be possibly cached in memory (to reduce the **read amplification**).
    - A lookup may search fewer levels of table files in the LSM-tree.
    - Most lookups only require a single random read (for the value).



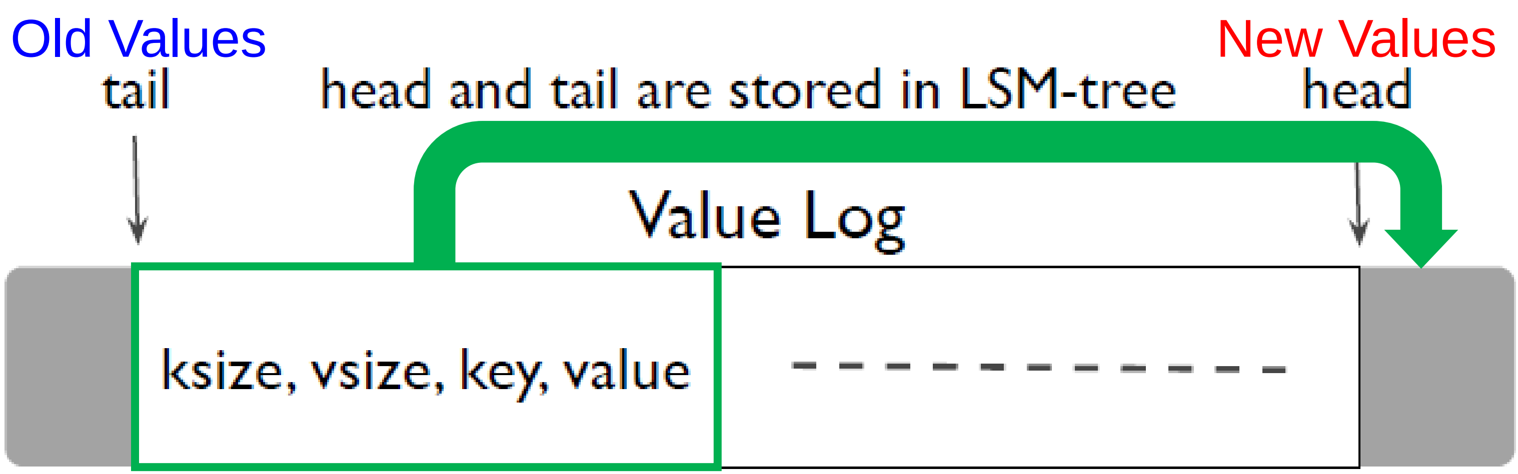
Challenges of Key-Value Separation (1/3)

- Key-value separation may leads to many **challenges**:
- **Challenge #1**: Since keys and values are separately stored in WiscKey, range queries require multiple random reads, which are not efficient to the disk.
- The design of WiscKey is highly SSD optimized.
  - **Parallel random reads** with a fairly large request size can fully utilize the internal parallelism of SSD, getting performance similar to sequential reads.



Challenges of Key-Value Separation (2/3)

- **Challenge #2**: Since WiscKey does not compact values, it needs a special garbage collector to reclaim space occupied by deleted/overwritten values in **vLog**.
- WiscKey targets a lightweight and online GC: It only keeps valid values in a contiguous range of **vLog**.
  - Valid values are appended back to the head of **vLog**.
  - Both keys and values should be kept in **vLog** to determine whether a value is valid or not (by querying the LSM-tree).

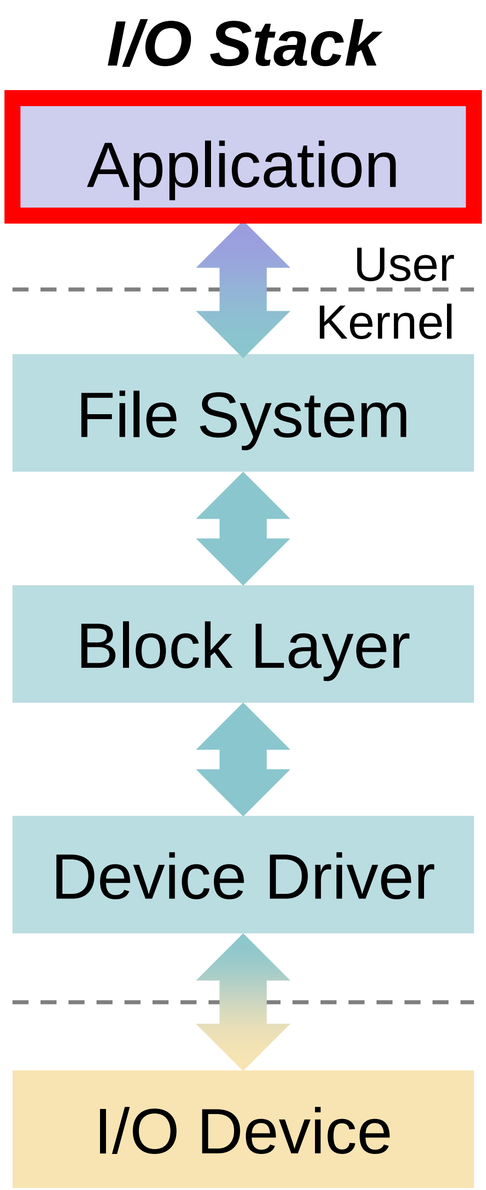


Challenges of Key-Value Separation (3/3)

- **Challenge #3**: Since WiscKey's architecture stores values separately from the LSM-tree, obtaining the same crash guarantees can appear complicated.
- WiscKey provides the following **crash guarantees**:
  - If the key cannot be found in the LSM-tree:
    - WiscKey informs the user that the key was not found.
  - If the key can be found in the LSM-tree:
    - WiscKey verifies ① whether the value address retrieved from the LSM-tree falls within the current valid range of vLog and ② whether the value found corresponds to the queried key.
    - If the verifications fail, WiscKey deletes the key from the LSM-tree, and informs the user that the key was not found.
  - WiscKey is not able to recovery the values, even if which had been written in **vLog** before the crash.

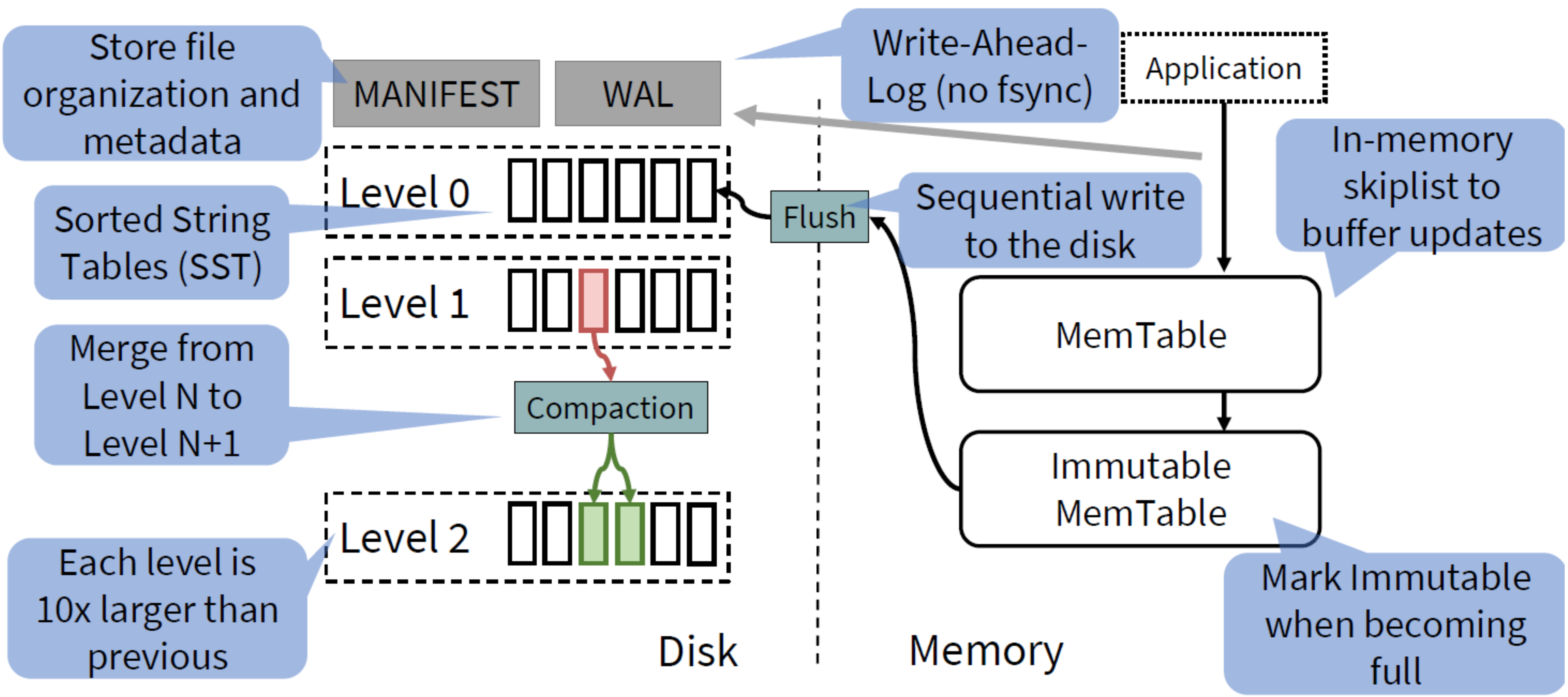
Outline

- Persistent Key-Value Store
  - Log-Structured Merge-Tree (LSM-tree)
- LevelDB (by Google)
  - Insertion and Compaction
  - Lookup
- WiscKey: Separating Keys from Values
  - Write and Read Amplification
  - Key-Value Separation
  - Benefits and Challenges
- Single-Level KV Store with PM
  - Single-Level Merge
  - Selective Compaction



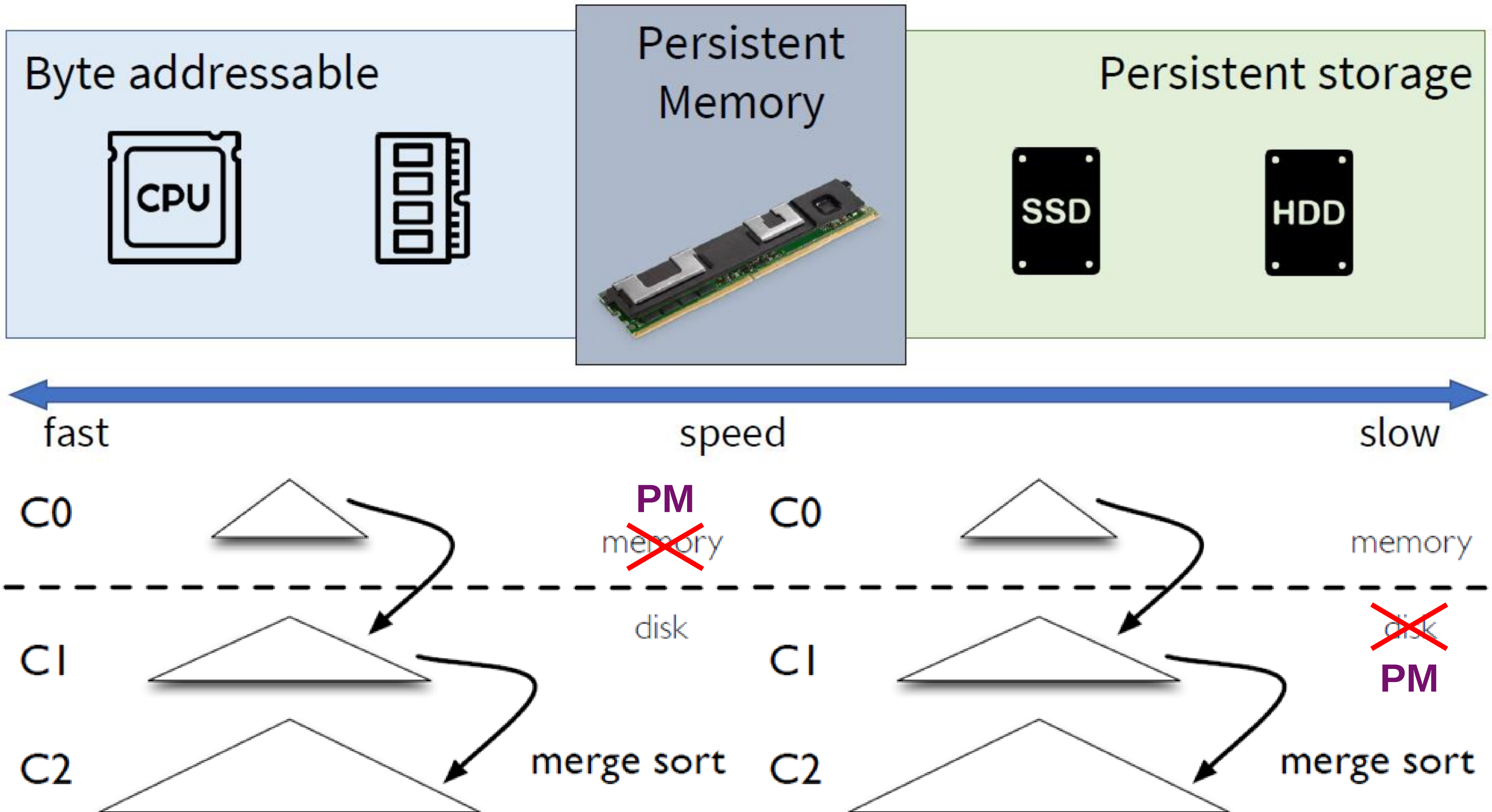
State-of-the-art LSM-tree: LevelDB

- Optimized for heavy write application.
- Designed for slow hard disk drives (HDDs).
- Suffered from serious write and read amplification.



Motivation: Byte-Addressable PM

- How can the **byte-addressable persistent memory** (PM) enhance the performance of key-value stores?

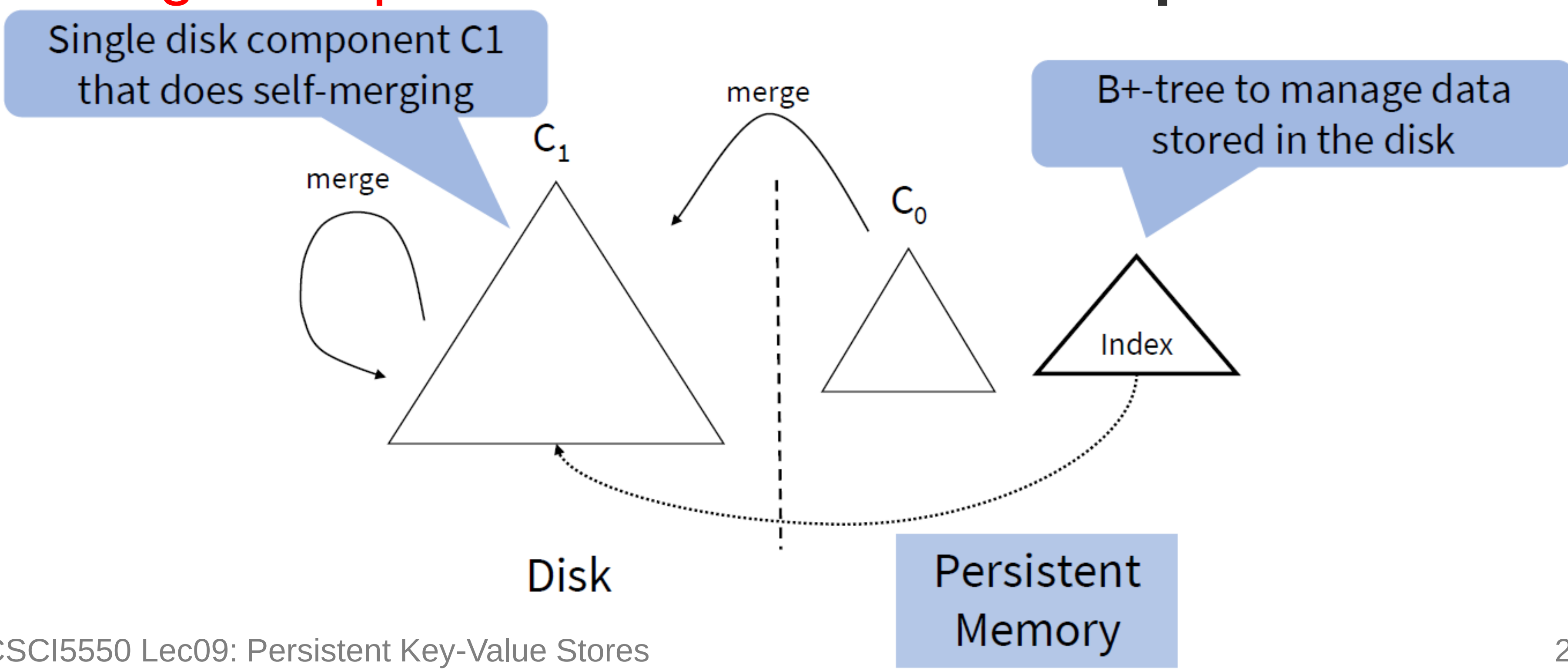


Idea: Single-Level Merge with PM

① Exploit **PM** to maintain a B+-tree index and stage KV pairs in a PM resident buffer (i.e., `C0`).

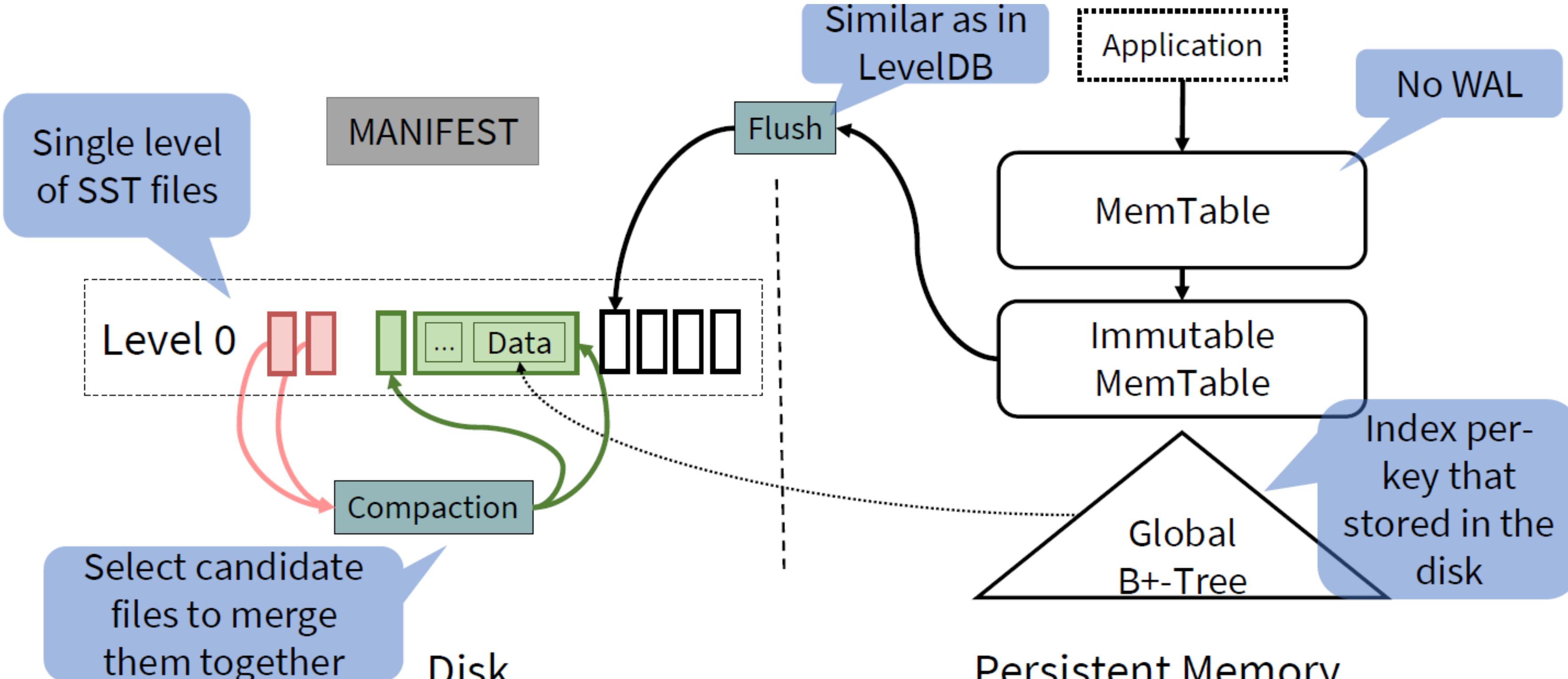
② Organize KV pairs in a single level on **disks** (i.e., `C1`).

→ Avoid write-ahead logging (WAL) and multi-leveled merge/compaction to reduce **write amplification**.

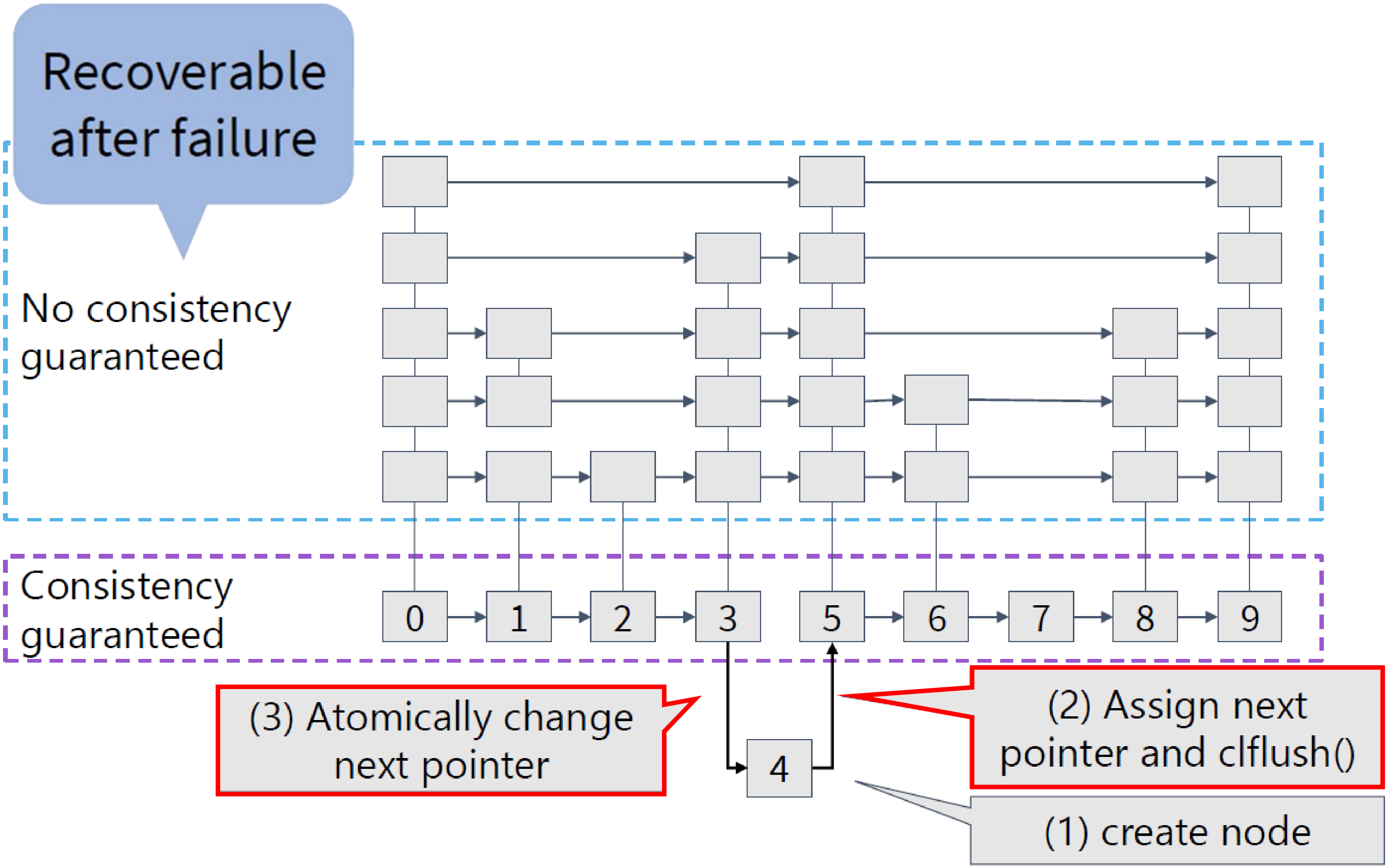


Single-Level Merge DB (SLM-DB)

- **Persistent memtable** avoids the write-ahead logging and provides stronger consistency than LevelDB.
- **Persistent B+-tree** avoids the on-disk multi-leveled merge structure and enables fast lookup.
  – No need to merge KV pairs of **one-level** SST files at all!

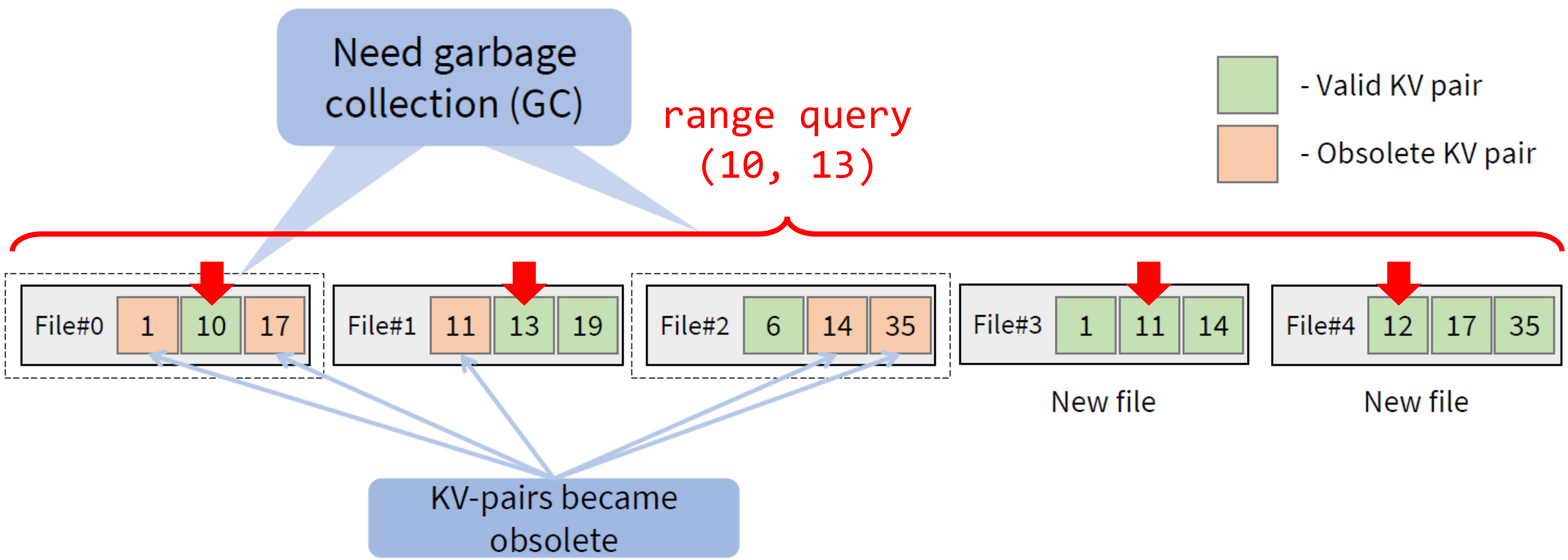


Persistent Memtable



Selective Compaction

- **Compaction** operation is still needed:
  1. ① To collect garbage of obsolete KV pairs, and
  2. ② To improve the sequentiality of KV pairs in SSTables.
- SLM-DM performs the compaction in a selective way.
  - A background thread compacts only candidate SSTables.



Summary

- Persistent Key-Value Store
  - Log-Structured Merge-Tree (LSM-tree)
- LevelDB (by Google)
  - Insertion and Compaction
  - Lookup
- WiscKey: Separating Keys from Values
  - Write and Read Amplification
  - Key-Value Separation
  - Benefits and Challenges
- Single-Level KV Store with PM
  - Single-Level Merge
  - Selective Compaction