2025 Winter RocksDB Study
2nd week

Hojin Shin, Guangxun Zhao

http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

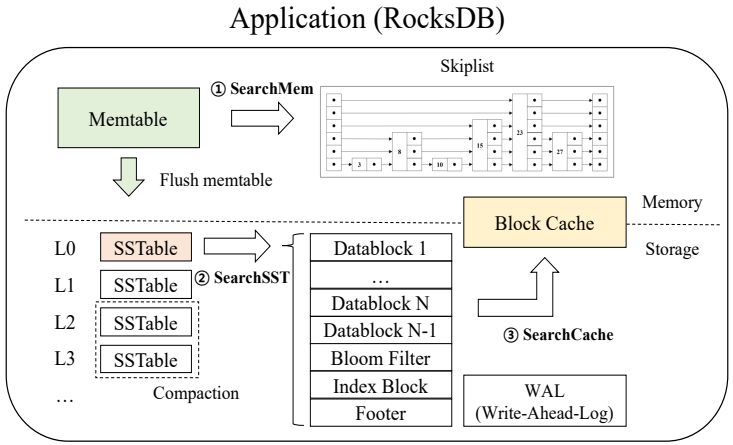
Presentation by Hojin Shin

hojin03s@dankook.ac.kr

Contents

RocksDB Summary

- RocksDB Summary
  - ✓Based on LSM (Log-Structured Merge)-tree
    - Layerd: C0, C1, …, Ck (exponentially increasing)
  - ✓Real Implementation: Memtable, SSTable
    - Memtable in memory, SSTable in storage (multiple levels: L0, L1, …, Lk)
  - ✓Interface: put, get, range scan, delete
  - ✓Put flow
    - WAL → Memtable → Immutable Memtable → Flush → Compaction (condition)



RocksDB Write (Put, Flush, Compaction)

- Put interface (WAL)
  - ✓Write-Ahead-Log: Do before write memtable
    - Crucial components of RocksDB that ensures data durability and recoverability
  - ✓A set of records where each record consists of CRC, size, type and payload
  - ✓Employs sequential disk writes and Crash recovery
  - ✓Options (on/off, configurable)

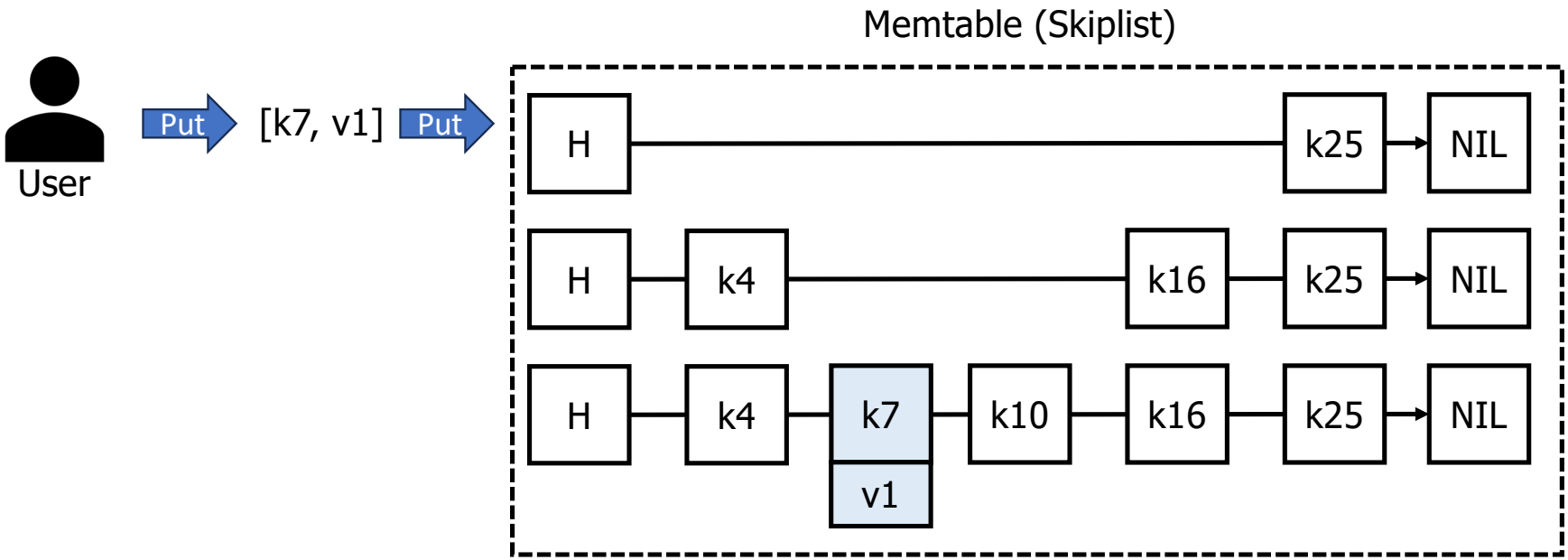
```
+---------+-----------+-----------+--- ... ---+
|CRC (4B) | Size (2B) | Type (1B) | Payload   |
+---------+-----------+-----------+--- ... ---+

CRC = 32bit hash computed over the payload using CRC
Size = Length of the payload data
Type = Type of record
       (kZeroType, kFullType, kFirstType, kLastType, kMiddleType )
       The type is used to group a bunch of records together to represent
       blocks that are larger than kBlockSize
Payload = Byte stream as long as specified by the payload size
```

WAL format

source: rocksdb github wiki

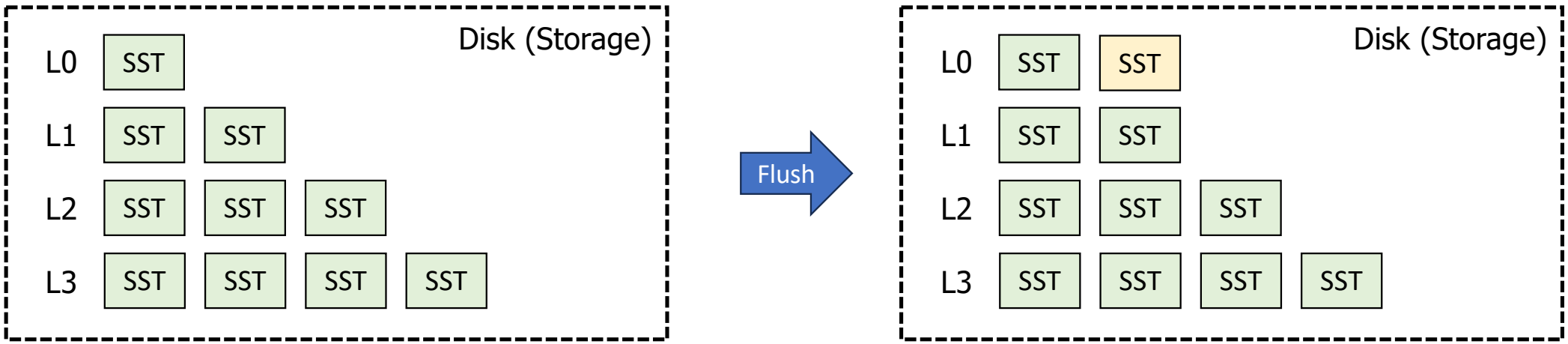
RocksDB Write (Put, Flush, Compaction)

- Put interface (memtable)
  - ✓Further separate into **mutable** and **immutable**
  - ✓Data structure: skiplist (or hashtable, hashskiplist …)
  - ✓Managing data in a **sorted** state
  - ✓Default size (64MB, configurable)



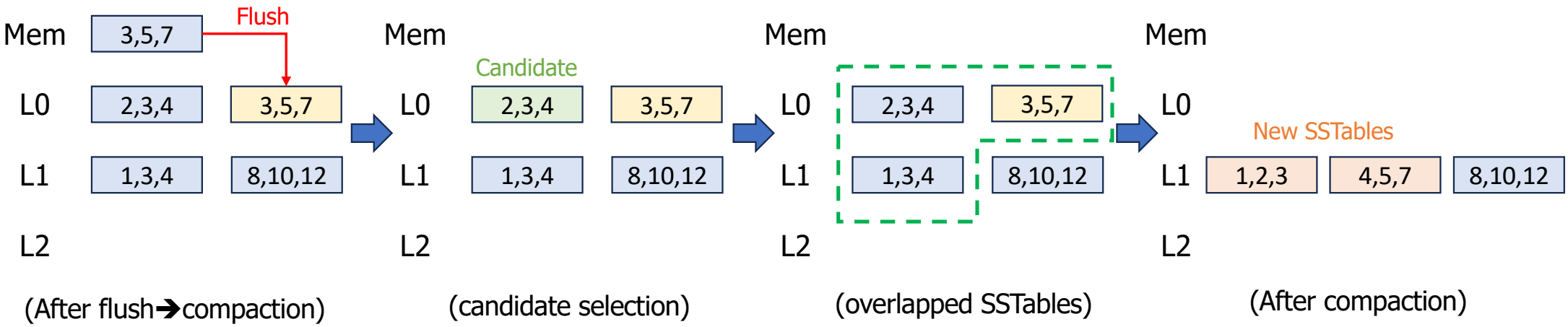
RocksDB Write (Put, Flush, Compaction)

- Put interface (Flush)
  - ✓Flush: Writing data from the in-memory Memtable to persistent SSTable files on disk
    - SSTable (Sorted String Table)
  - ✓Triggering the flush
    - Memory limit reached (64MB) → Memtable to Immutable memtable → Stored data in Level 0
  - ✓Occur in the background



RocksDB Write (Put, Flush, Compaction)

- Put interface (Compaction)
  - ✓Compaction procedure
    - 1) Select candidate (FIFO, Least overlapped, …)
    - 2) Read overlapped SSTables from current and next level
    - 3) Do **merge sort**
    - 4) Write new SSTables to the next level



RocksDB Write (Put, Flush, Compaction)

- Put interface (Compaction)
  - ✓Compaction effect
    - 1) Remove old data (reclaim)
    - 2) Sort keys at L1, L2, … (fast lookup)
  - ✓Compaction cost
    - Read/Write SSTables from/into storage ➔ **Heavy operation**
    - Cause **amplification**
  - ✓Amplification
    - An undesirable phenomenon where the actual amount of operations/space are more than intended
  - ✓Types
    - 1) WAF (Write Amplification Factor) = actual writes/intended writes
    - 2) SAF (Space Amplification Factor) = actual used space/required space
    - 3) RAF (Read Amplification Factor) = actual reads/intended reads

RocksDB Read (Get, Bloom filter)

- What is lookup (query)?
  - ✓Two types
    - 1) Point lookup (single query): get a value related to a given key
    - 2) Range lookup (scan): get values related to a range of keys
  - ✓How to: various data structures (e.g. array, list, sorted, hash, …)
    - 1) linear search (O(N)), 2) binary search (O(logN)), 3) hash (O(1))
    - Tradeoffs: search speed, update overhead, scan overhead, …

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

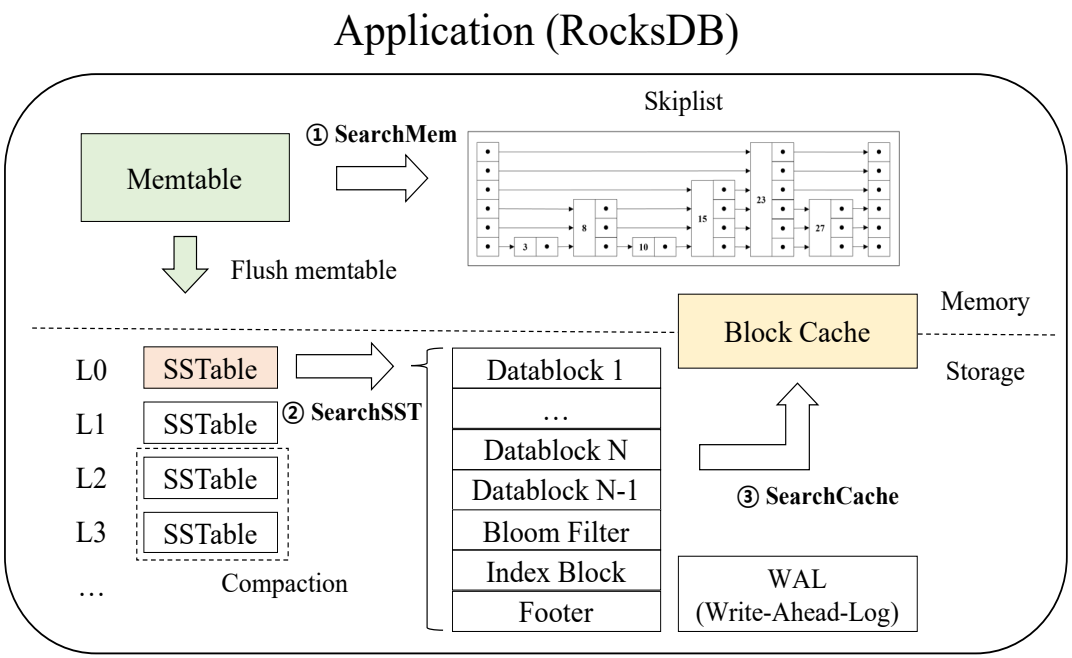
source: https://www.digitalvidya.com/blog/data-structures-and-algorithms-in-python/

RocksDB Read (Get, Bloom filter)

- How to materialize lookup in DB?
  - ✓In traditional RDB (e.g. InnoDB)
    - Make use of B-tree (or B+tree)
    - B-tree: generalized binary tree that has multiple children (n-ary tree)
    - B+tree: all KV are stored in leaves, all leaves are linked for scan
    - Good for lookup, Bad for update due to tree reconstruction
  - ✓RocksDB
    - Make use of LSM-tree, a write-optimized data structure
    - Employ B+tree for lookup may deteriorate its original merit
    - Utilize its own data structures for lookup purpose including **Skiplist**, Index block and Bloom filter

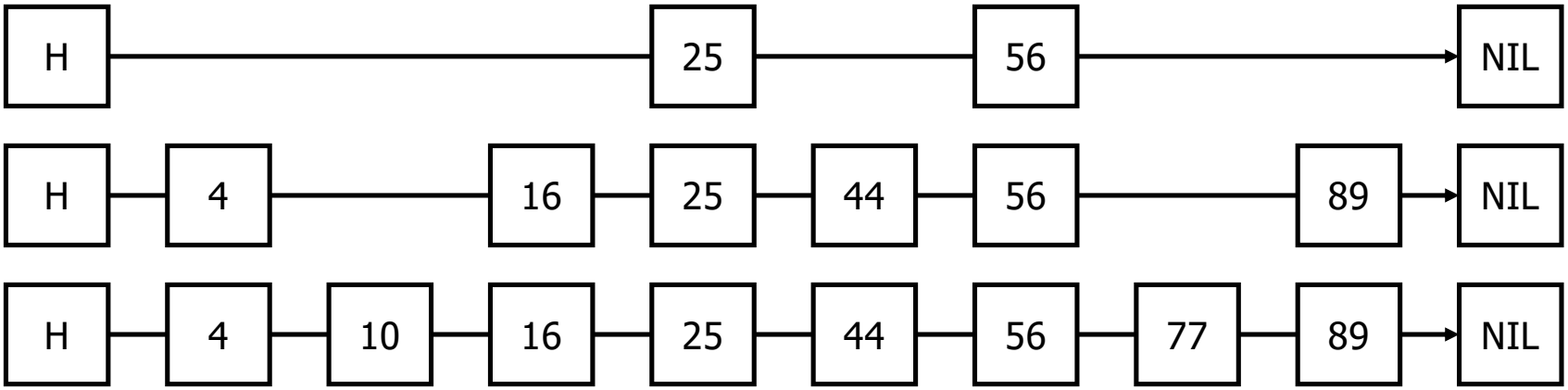
RocksDB Read (Get, Bloom filter)

- RocksDB lookup: Overview
  - ✓Lookup procedure
    - 1) Memtable and Immutable memtable
    - 2) SSTables (from L0 to Lmax)



RocksDB Read (Get, Bloom filter)

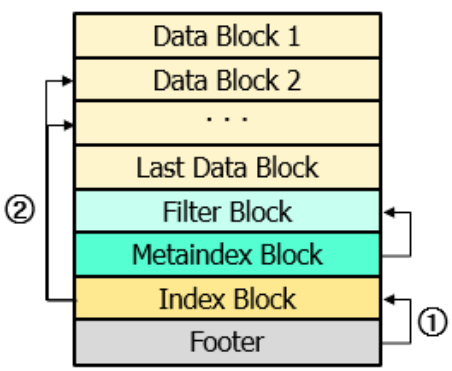
- Get interface (Memtable)
  - ✓KV pairs in memory, managed by Skiplist
    - Skiplist: a data structure with a set of sorted linked lists
    - All keys appears in the last list
    - Some keys also appear in the upper list (for fast search)
    - Good for both lookup and scan (O(logN))
    - Useful in multithreaded system architectures



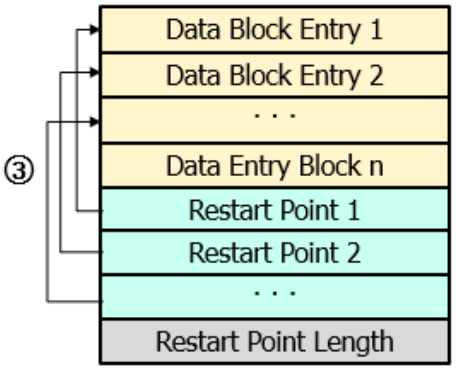
SkipList Structure

RocksDB Read (Get, Bloom filter)

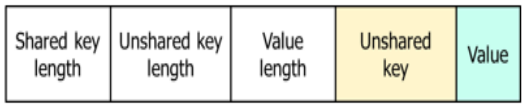
- Get interface (SSTable)
  - ✓KV pairs in storage, managed as a file
  - ✓Issue: SSTable is large (default 64MB)
    - Assume 1KB KV ➔ 64,000 pairs in a file
    - A file is divided into multiple disk blocks
  - ✓Solution: well-defined SSTable format
    - 1) SSTable is divided into data blocks
    - 2) Each KV is searched using index (Binary search)
    - 3) Filter block (**Bloom filter**)
    - 4) other meta blocks (e.g. compressions)
    - 5) metaindex: one entry for every meta blocks
    - 6) footer: pointers for metaindex & index blocks



(SSTable format)



(Data block format)

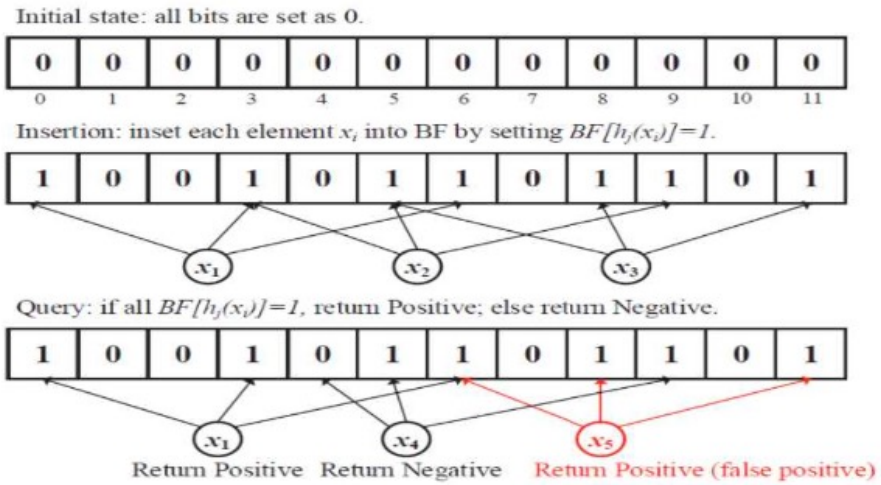


(Data block entry format)

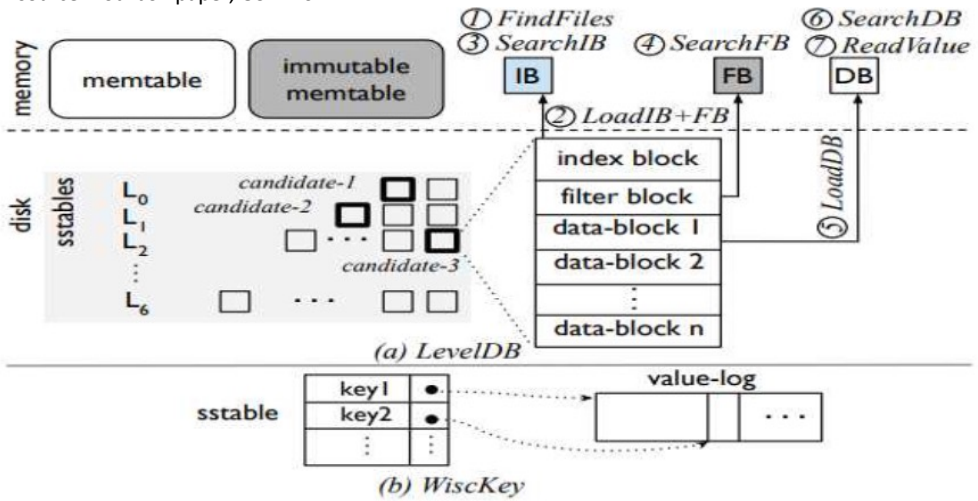
RocksDB Read (Get, Bloom filter)

- Get interface (Bloom filter)
  - ✓Used to reduce the read amplification (unnecessary read)
    - Only know the key range of each SSTable
  - ✓Bloom filter: a data structure for identifying membership
    - Based on bits and multiple hashes
    - Good property: No false negative
    - Issue: can yield false positive ➔ tradeoffs between bits and rate (1% false positive rate with 9.9 bits per key, from RocksDB wiki)

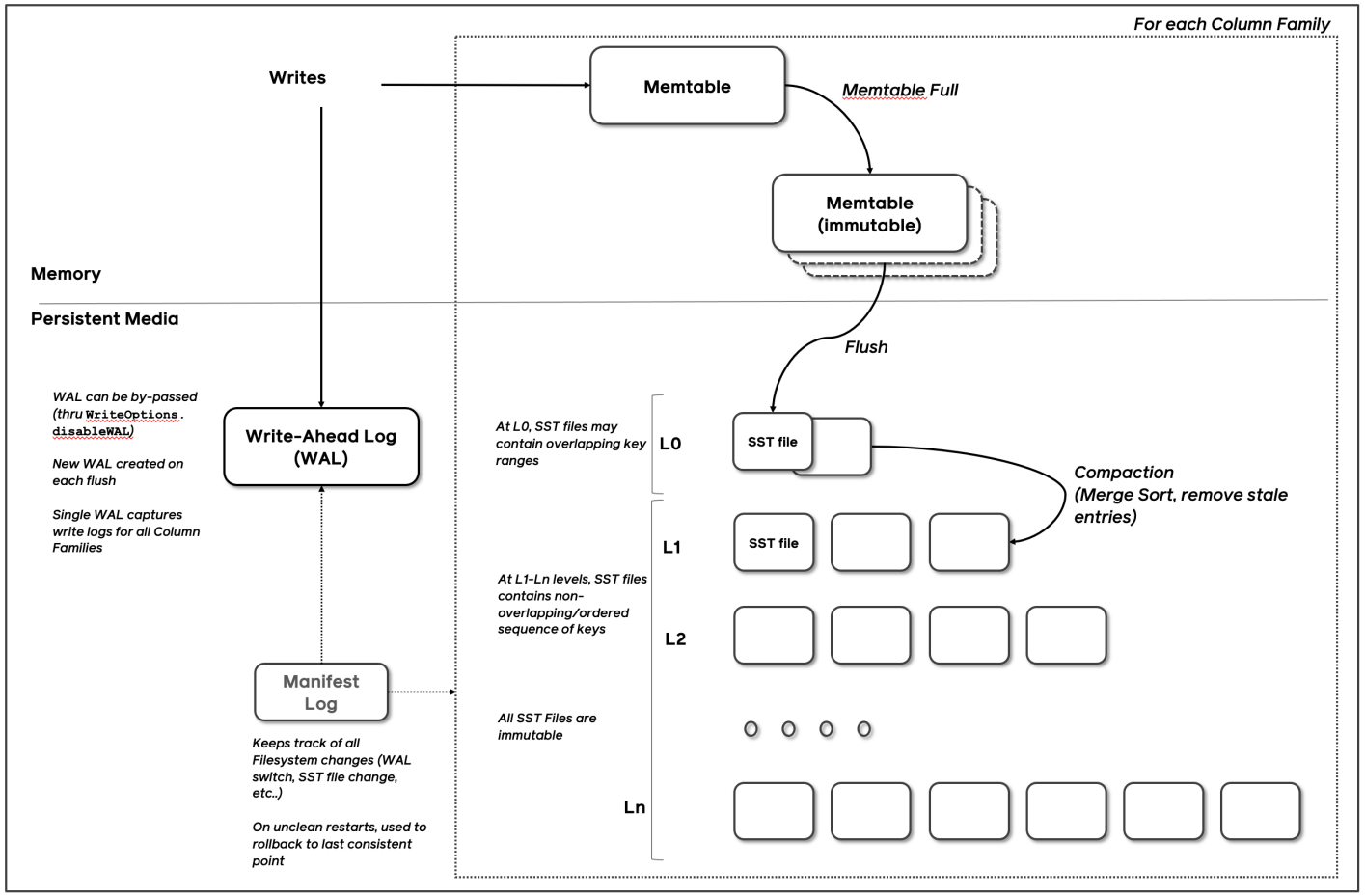
source: https://devopedia.org/bloom-filter



source: Bourbon paper, OSDI'20



RocksDB Wrap up



source: https://github.com/facebook/rocksdb/wiki/RocksDB-Overview

2025 Winter RocksDB Study
2nd week

Hojin Shin, Guangxun Zhao

http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

Thank You
Q & A ?

Presentation by Hojin Shin

hojin03s@dankook.ac.kr