Log-Structured Merge-Tree

Nope
May 29
Learn-Sys Group

Applications of LSM-Tree

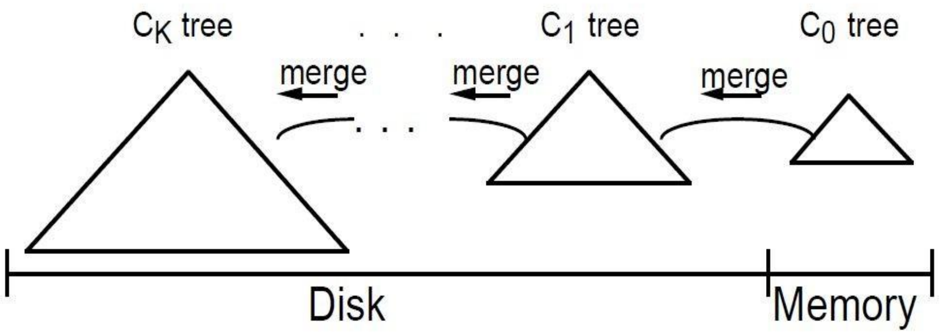
Key-Value Store (KVS) has become a necessary infrastructure.

- Indexing, Caching (Redis, Memcached) B-Tree based, HASH based
- Storage System (Persistent Key-value Store)
  - NoSQL
    - BigTable
    - HBase
    - Cassandra
  - Storage Engine
    - LevelDB
    - RocksDB
    - MyRocks

Beginning of LSM-Tree

- **1992:** The Design and Implementation of a **Log-Structured File System (LSF)**
  - Out-of-place Updates (compared with FFS In-place Updates)
  - Write Sequentially
  - Garbage Collection
  - …

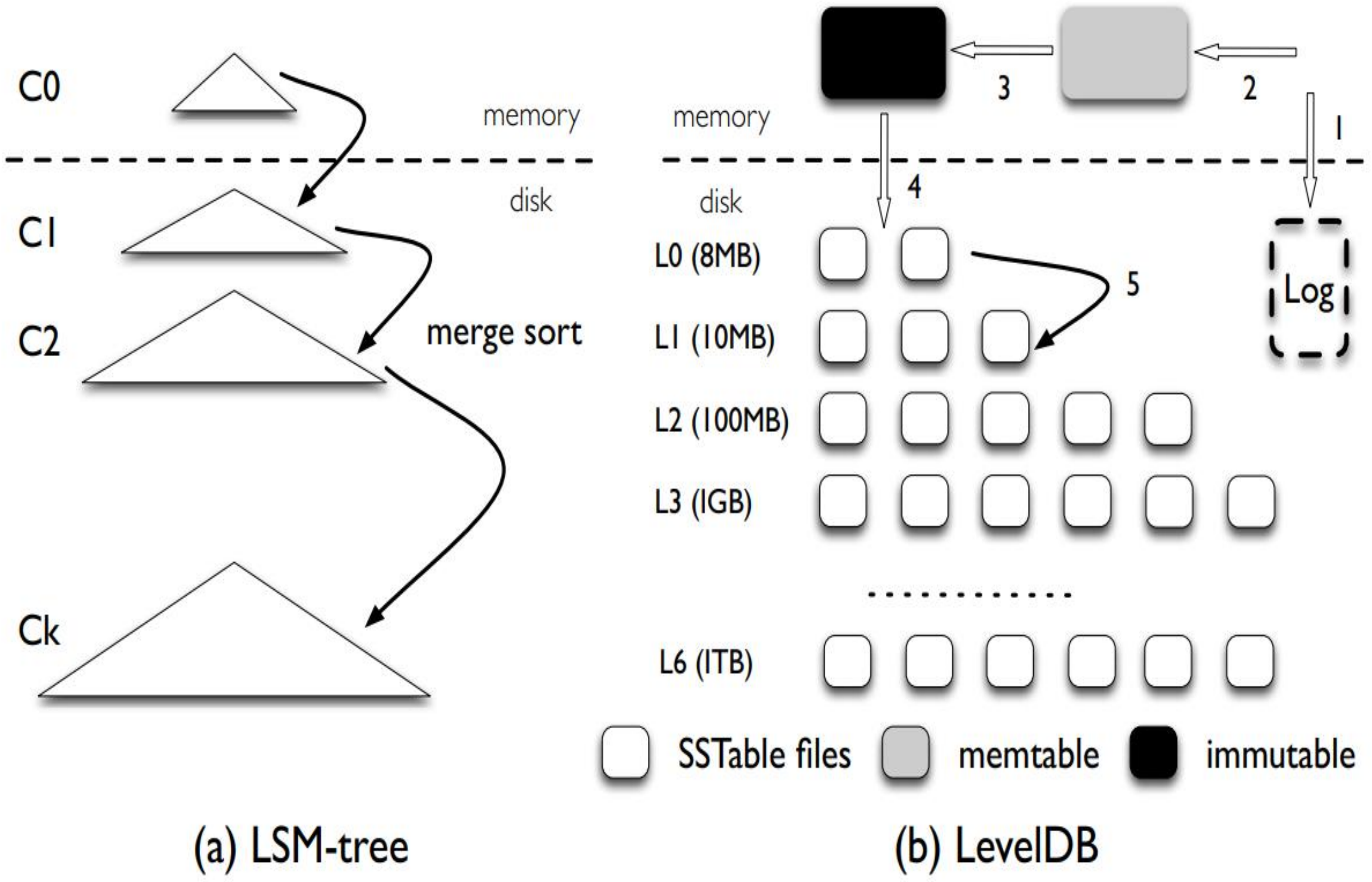
- **1996: The Log-Structured Merge-Tree (LSM-tree)**
  - Out-of-place update
  - Optimized for write / Sacrifice read
  - Require data reorganization (merge/compaction)
  - …



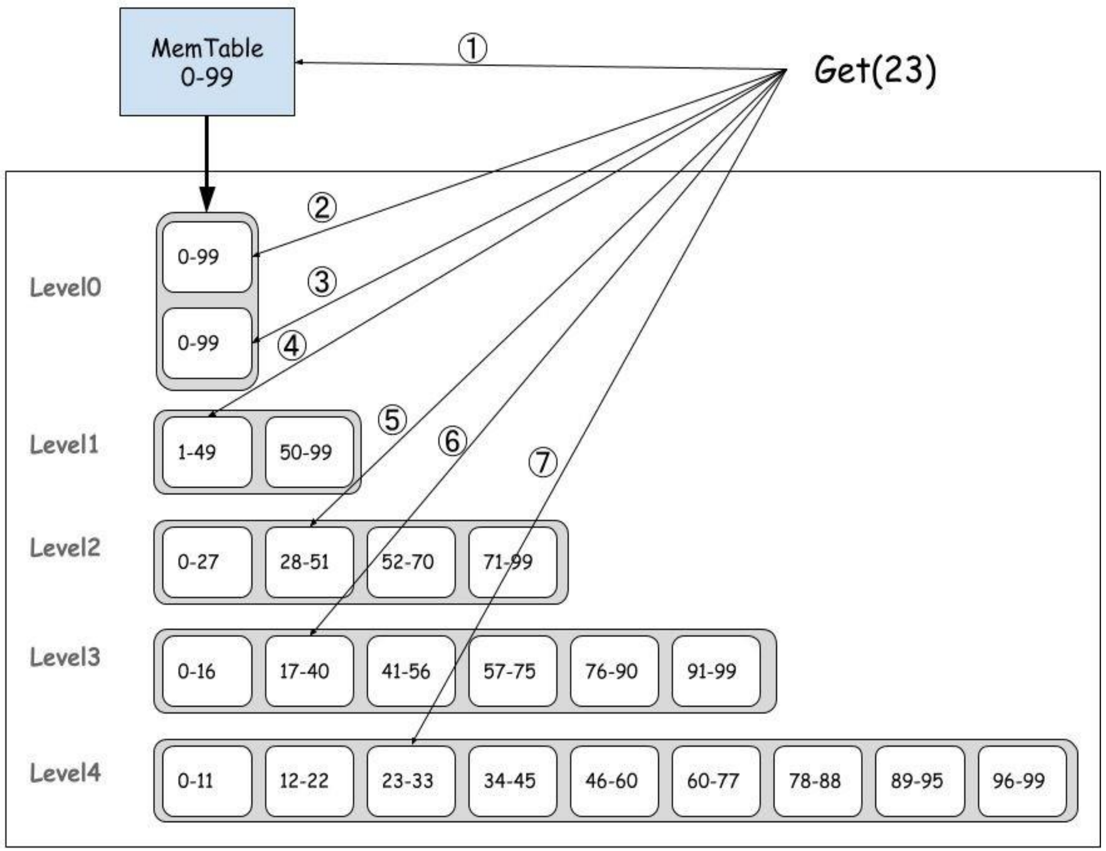
Modern Structure of LSM-Tree

- **Structure**
  - Memory Component (Memtable)
  - Disk Component (SSTable)
    - Level 0
    - Other Levels
  - Log (WAL/MANIFEST)

- **Operations**
  - Read (Point Query / Range Query)
  - Write
    - Memory Write
    - Flush
    - Compaction
  - Delete is also Write. (Write tombstone)



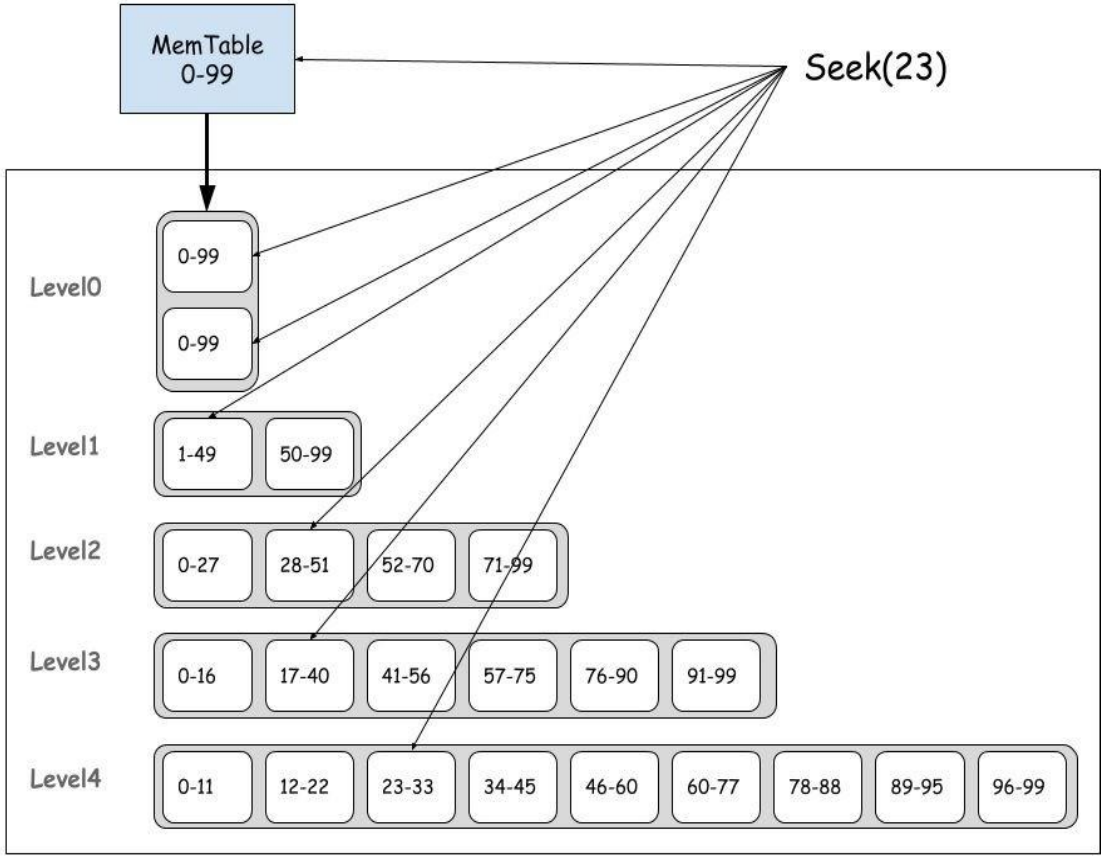
Operations – Point Query



- **Returns immediately when something found**

- **Problems:**
  - Read Amplification:
    - Files: (Worst Case) Search (N - 1 + files num of level-0) files.
    - Inside the file
- **Optimization**
  - Page cache/Block cache
  - Bloom filter
  - Other Index/Filter

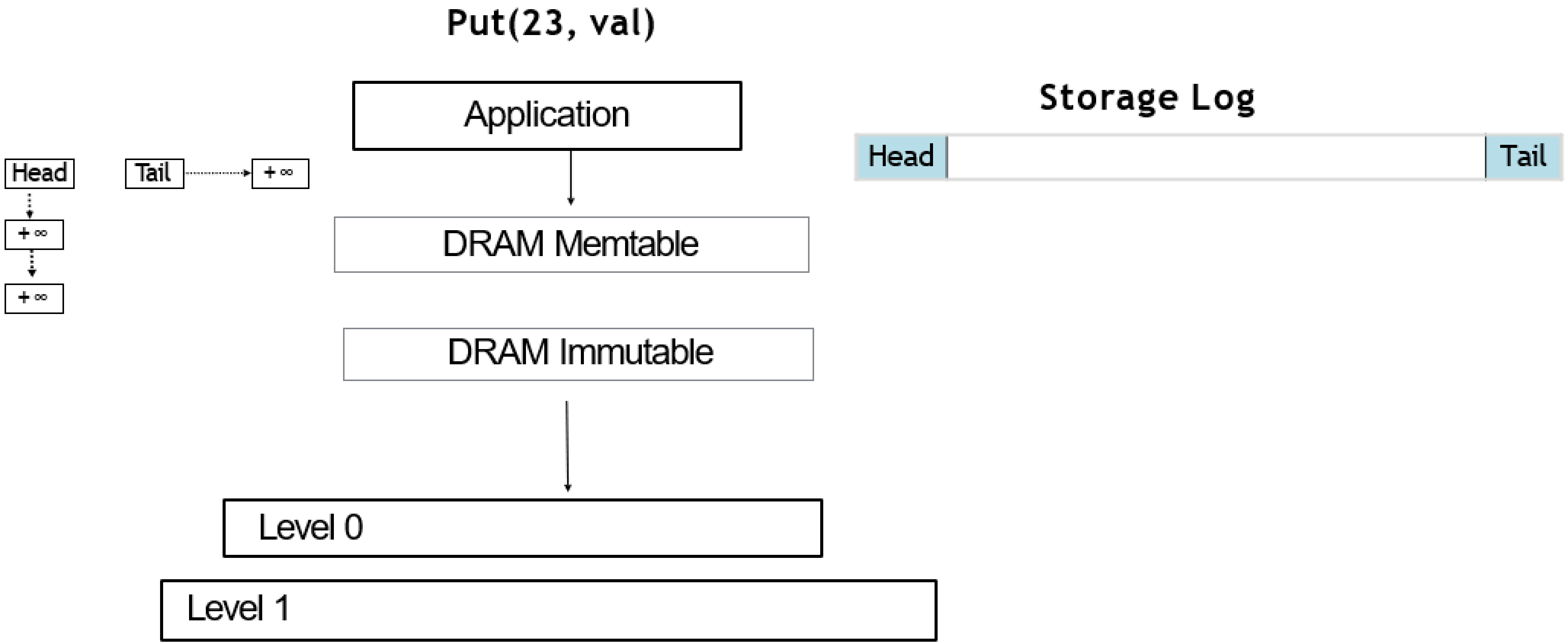
Operations – Range Query



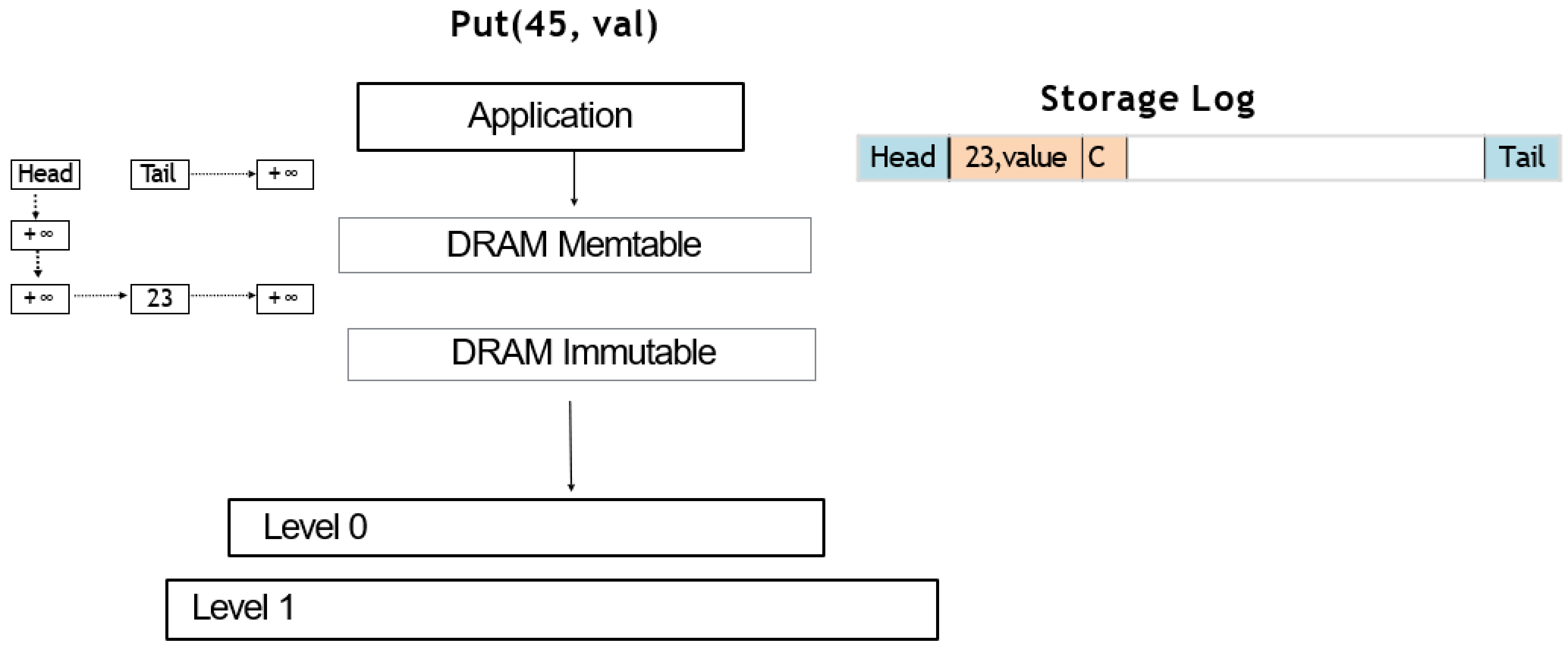

```
for (itr->Seek(23); itr->Valid(); itr->Next()) {  if (itr->key() < 40) {
   ...
  } else ...
}
```

- **Must seek every sorted run**
- Bloom filter not support range query
- **Optimization**
  - Parallel Seeks
  - Prefix bloom filter(RocksDB)
  - Other Index

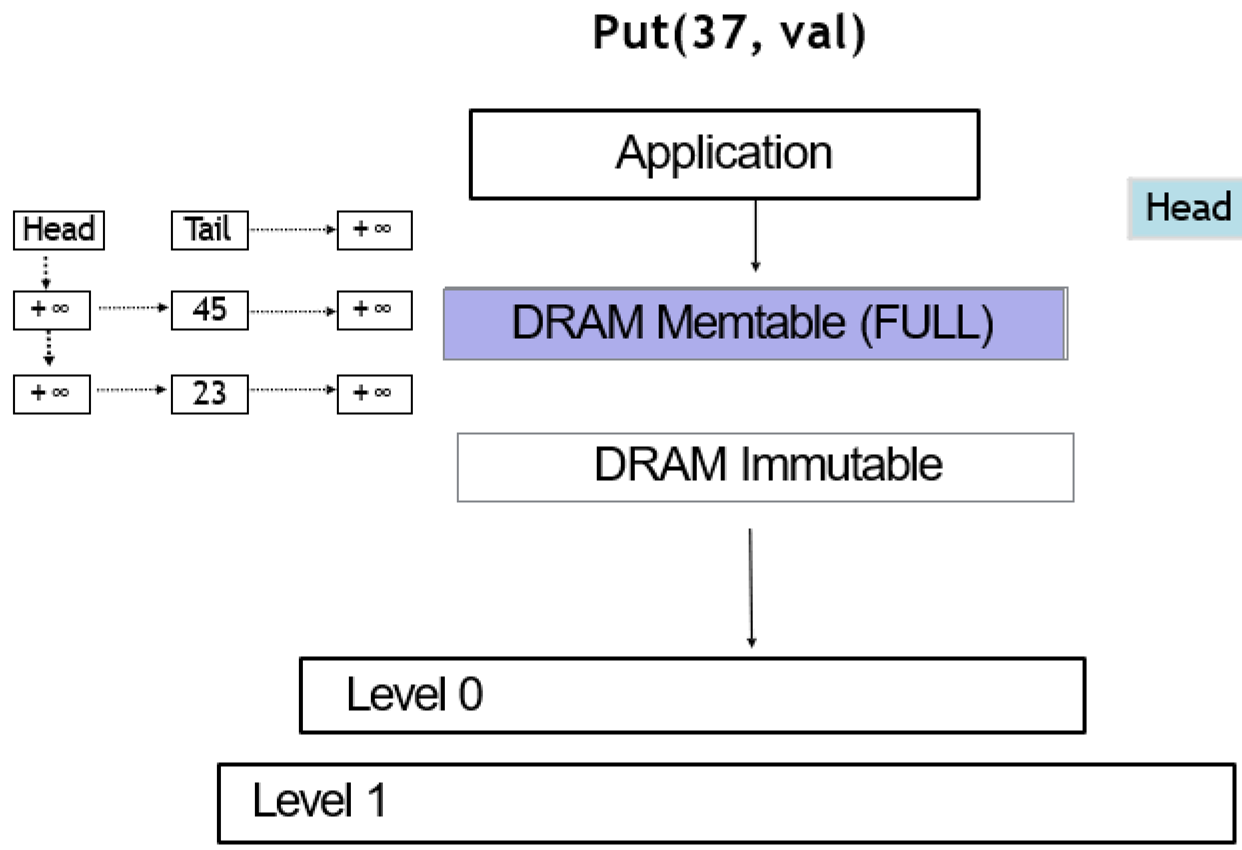
Operations – Simple Write



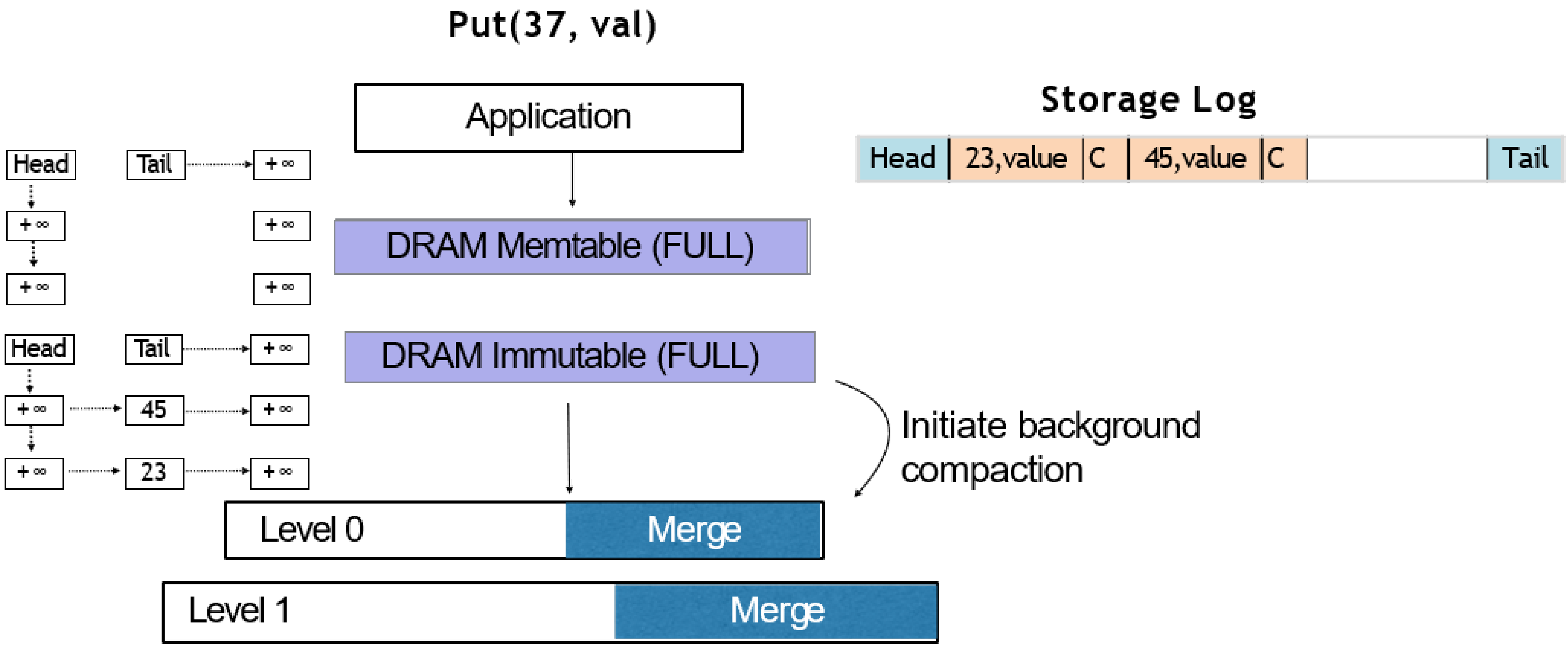
Operations – Simple Write



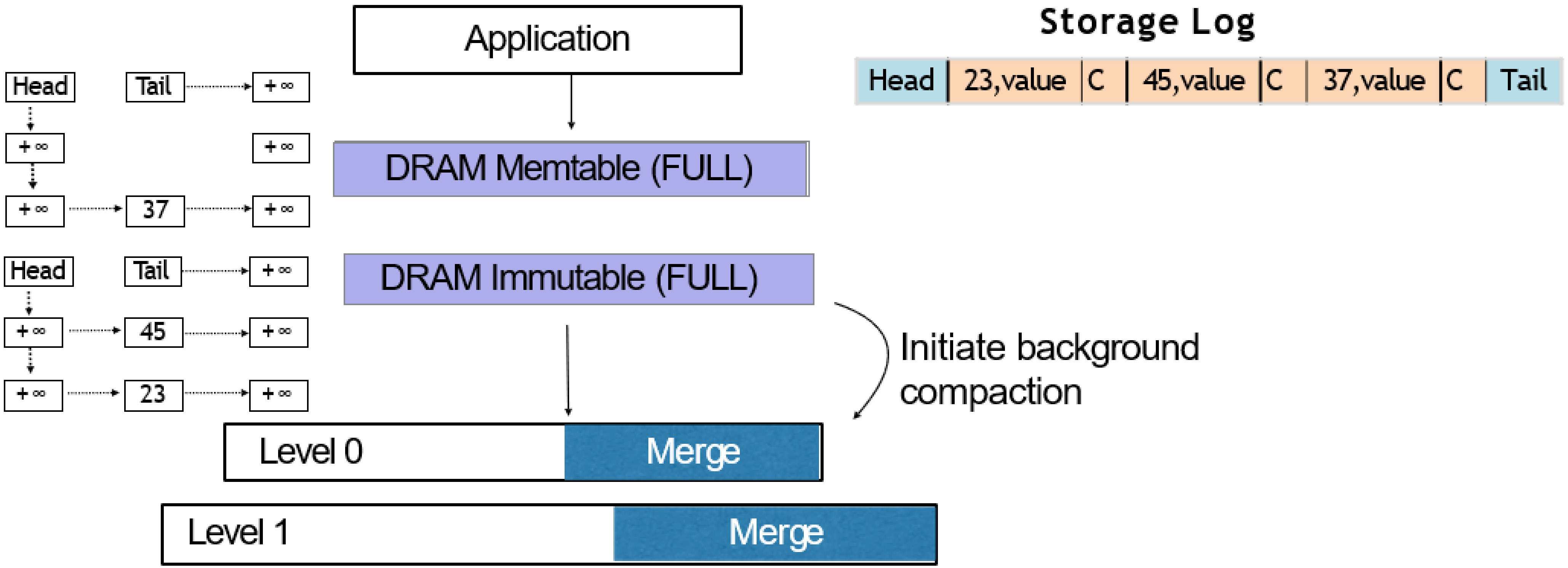
Operations – Simple Write



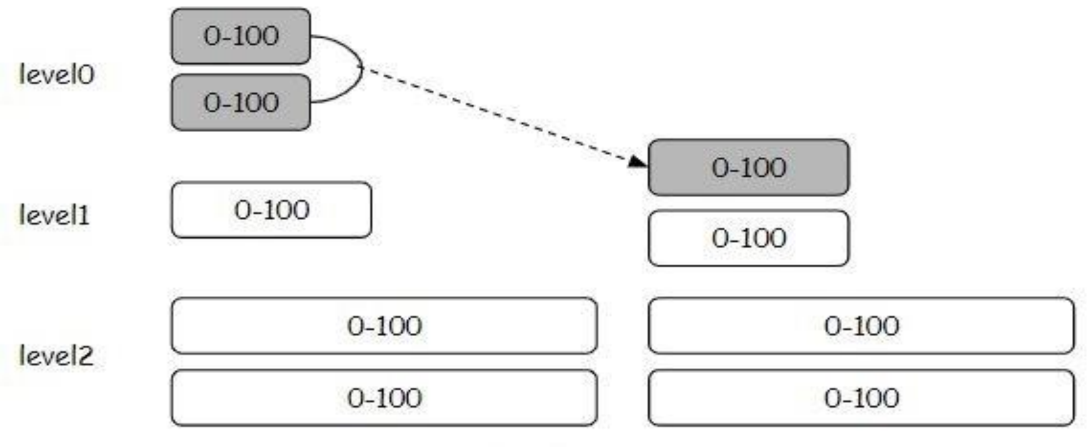
Operations – Simple Write



Operations – Simple Write

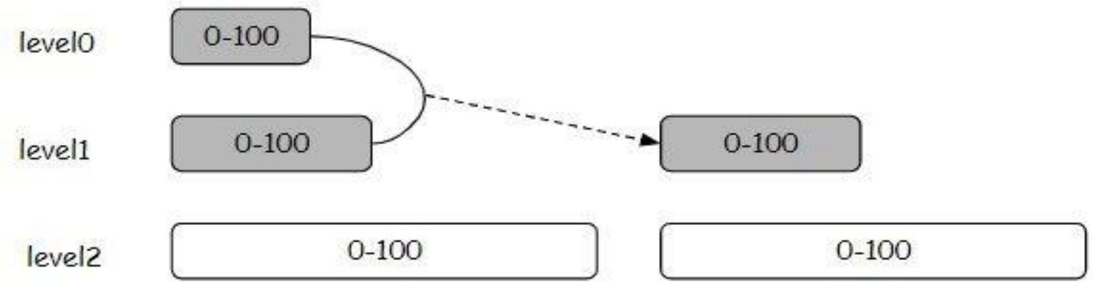


Operations – Compaction Tiered vs Leveled



Tired Compaction

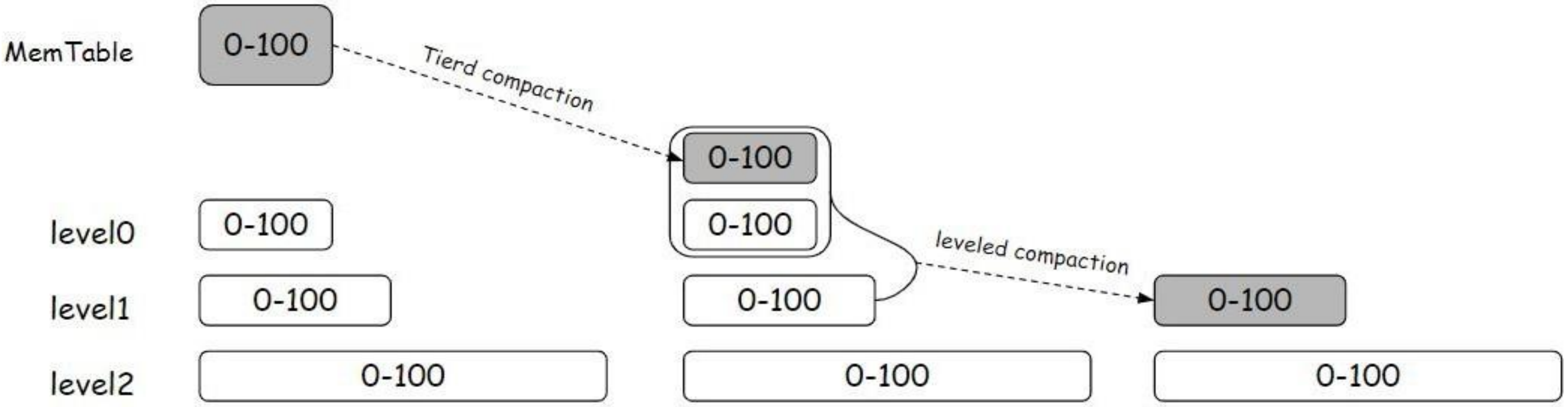
- Each level has N sorted runs (overlapped).
- Compaction merges all sorted runs in one level to create a new sorted run in the next level.
- **Minimizes write amplification at the cost of read and space amplification.**



Leveled Compaction

- Each level is one sorted run.
- Compaction into Ln merges data from Ln-1 into Ln.
- Compaction into Ln rewrites data that was previously merged into Ln.
- **Minimizes space amplification and read amplification at the cost of write amplification.**

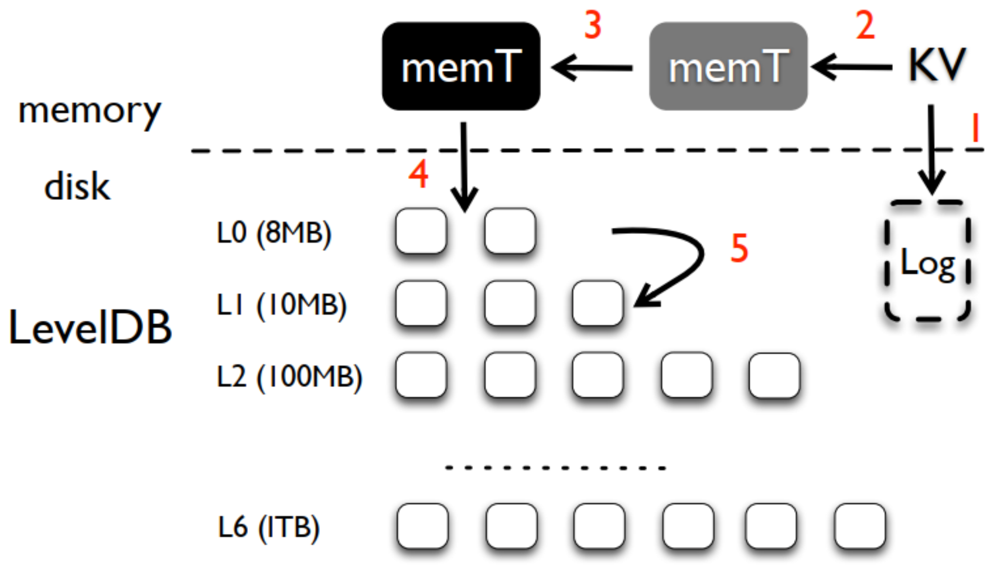
Operations – Compaction Tiered + Leveled



- Less write amplification than leveled and less space amplification than tiered.
- More read amplification than leveled and more write amplification than tired.
- It is flexible about the level at which the LSM tree switches from tiered to leveled.

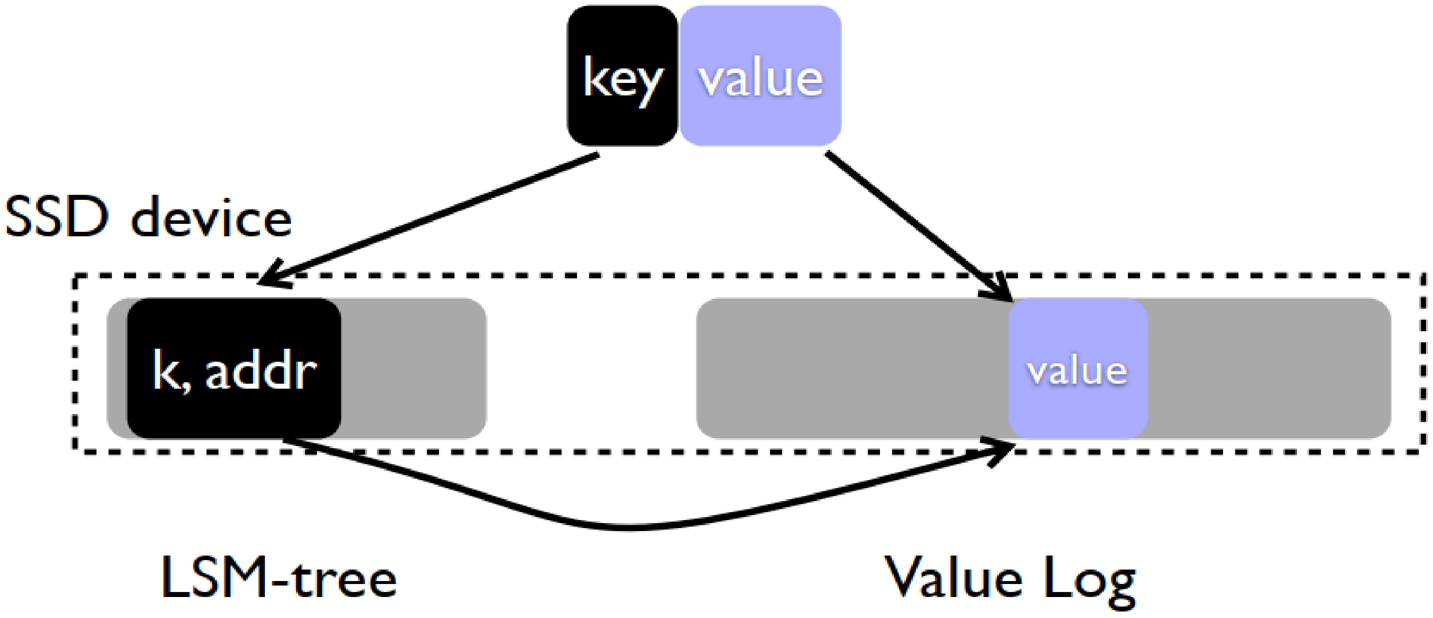
Write Operation Summary

- Write log firstly, and write in memory. (Critical Path)
- Flush from memory to disk. (Write Stall)
- Compactions. (Write Amplification)
- Optimization
  - Compaction Algorithm
  - Client Operation & Internal Operation Tradeoff
  - Cache…



FAST16’ WiscKey

- ➢ **Decouple sorting and garbage collection**
- ➢ **Harness SSD’s internal parallelism for range queries**
- ➢ **Online and light-weight garbage collection**
- ➢ **Minimize I/O amplification and crash consistent**



WiscKey Range Query

- **Parallel range query**
  - leverage **parallel random reads** of SSDs
  - prefetch key-value pairs in advance
    - detect a sequential pattern
    - prefetch concurrently in background

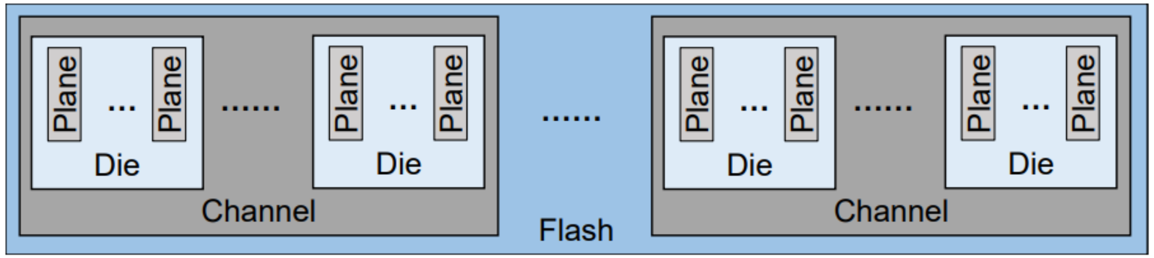


Figure 2: SSD Architecture: Internal parallelism in SSDs creates opportunities for hardware-level isolation.

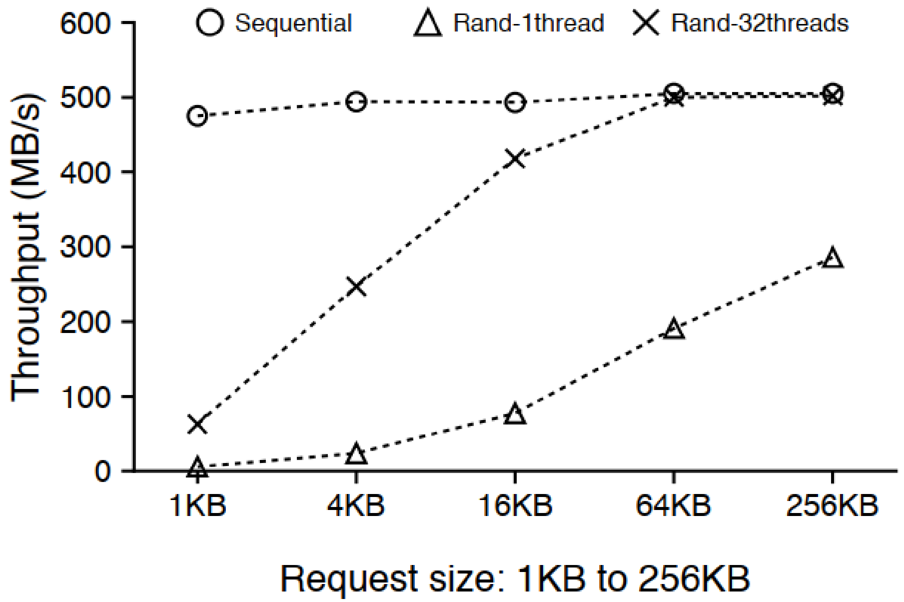
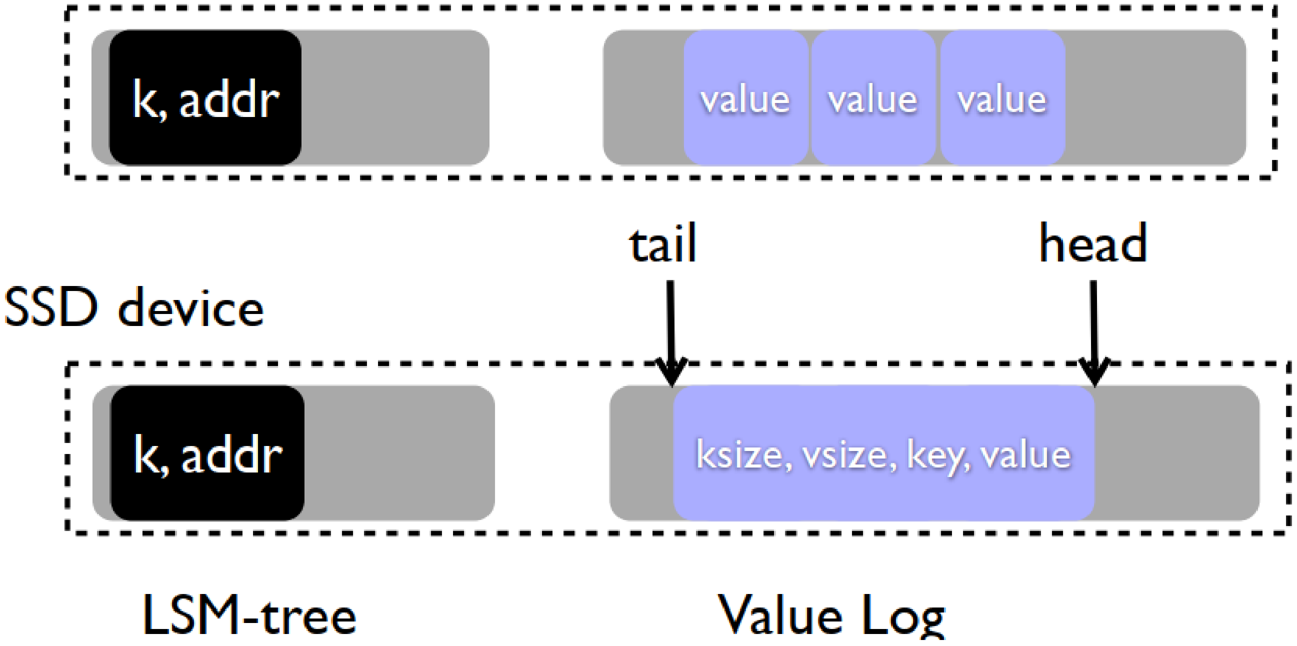


Figure 3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100-GB file on ext4.*

WiscKey Garbage Collection

- **Online and light-weight garbage collection**
  - append (ksize, vsize, key, value) in value log

- **Remove LSM-tree log in WiscKey**
  - store head in LSM-tree periodically
  - scan the value log from the head to recover



Summary

- Designing Access Methods: The RUM Conjecture
  - Read, Update, Memory(Space) – Optimize Two at the Expense of the Third
  - Read Amplification vs Write Amplification vs Space Amplification

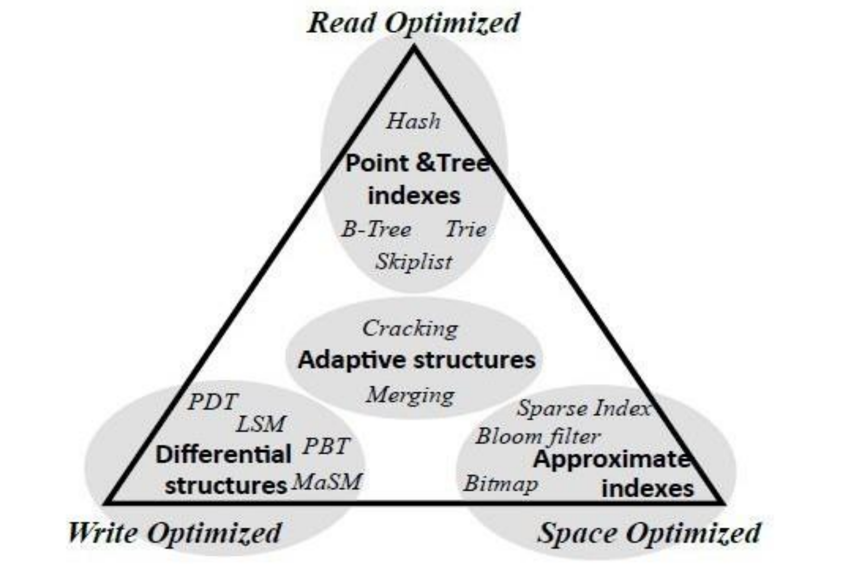
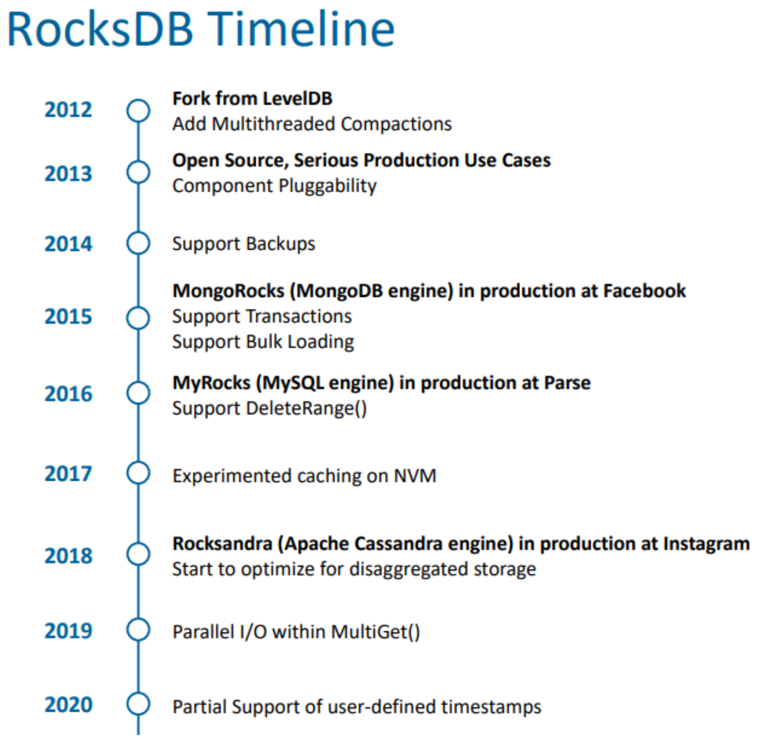


Figure 1: Popular data structures in the RUM space.



- Initial Optimization Targets are Write Amplification
- Majority of use cases are bounded by SSD space
- Reducing CPU overheads is becoming more important for efficiency
- Now working on disaggregated storage to achieve balanced CPU and SSD usage

Thanks