A Study of LSM-Tree

林金河

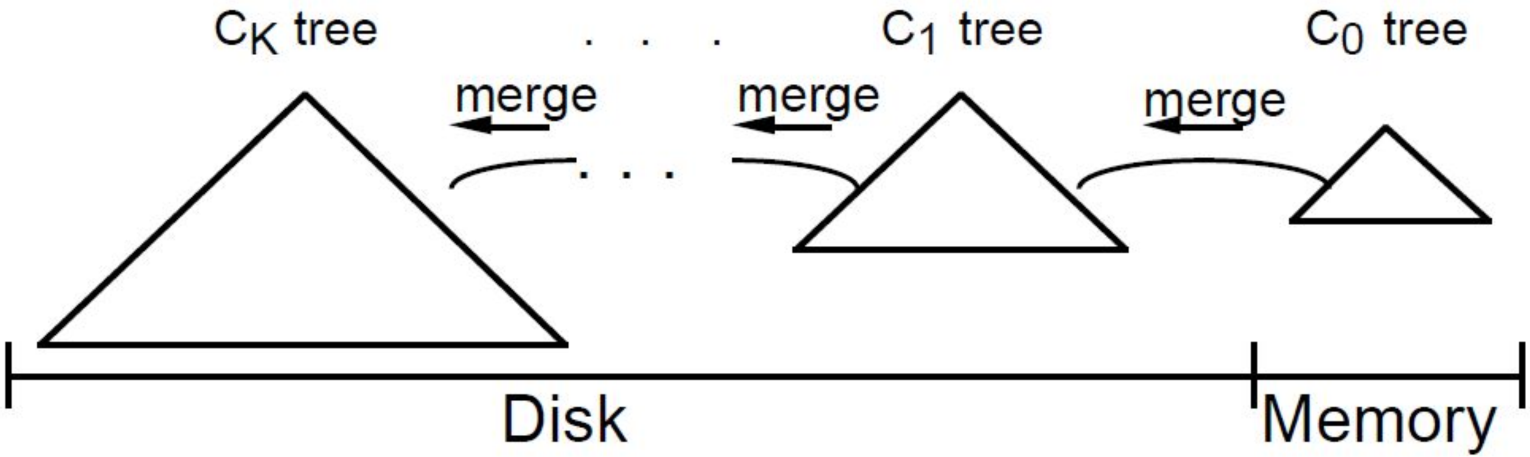
Applications of LSM-Tree

- NoSQL
  - Bigtable
  - HBase
  - Cassandra
  - Scylla
  - MongoDB
- Storage Engine
  - LevelDB
  - RocksDB
  - MyRocks
- NewSQL
  - TiDB(TiKV)
  - CockroachDB

Beginning of LSM-Tree

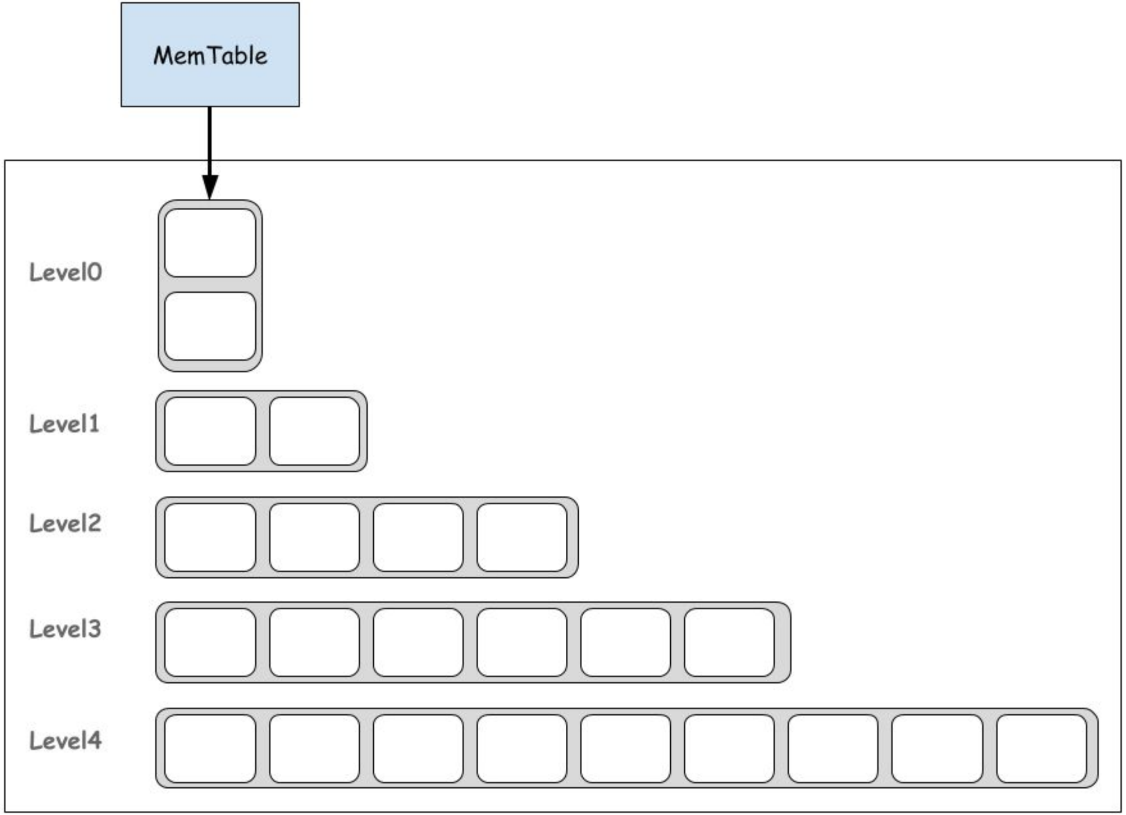
- 1996: The Log-Structured Merge-Tree (LSM-Tree)
  - Memory is small
  - Disk is slow for random access(r/w)
  - Originally designed for fast-growing History table
    - Data and indexes
    - Write heavy
    - Read sparse

Beginning of LSM-Tree



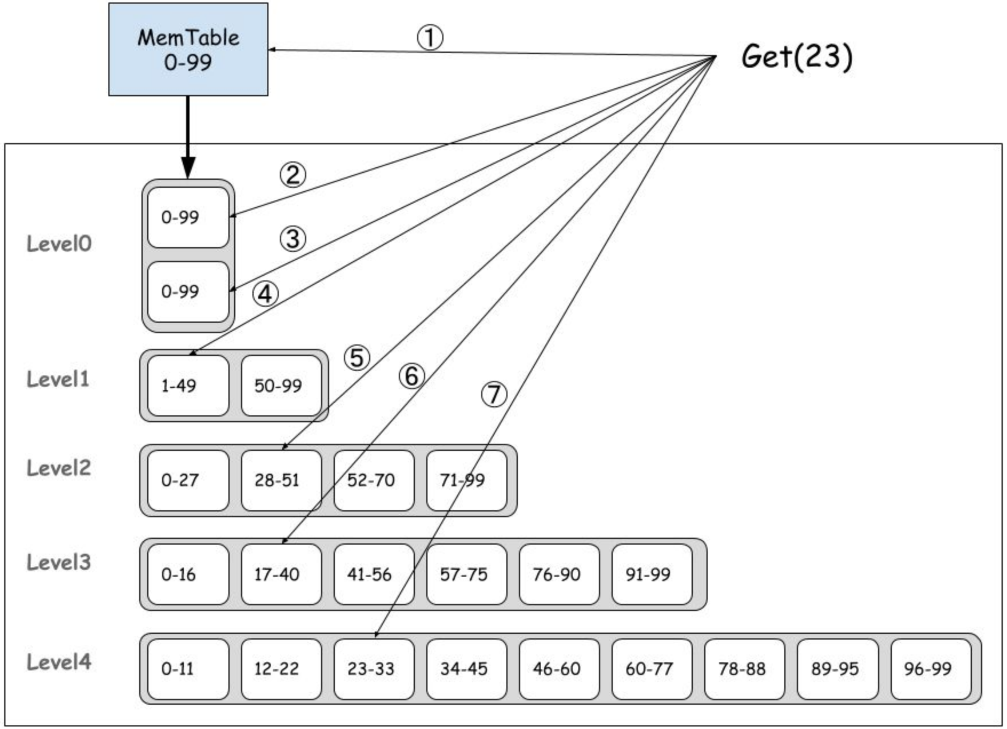
- Out-of-place update
- Optimized for write
- Sacrifice read
- Not optimized for space
- Require data reorganization (merge/compaction)

Modern Structure of LSM-Tree



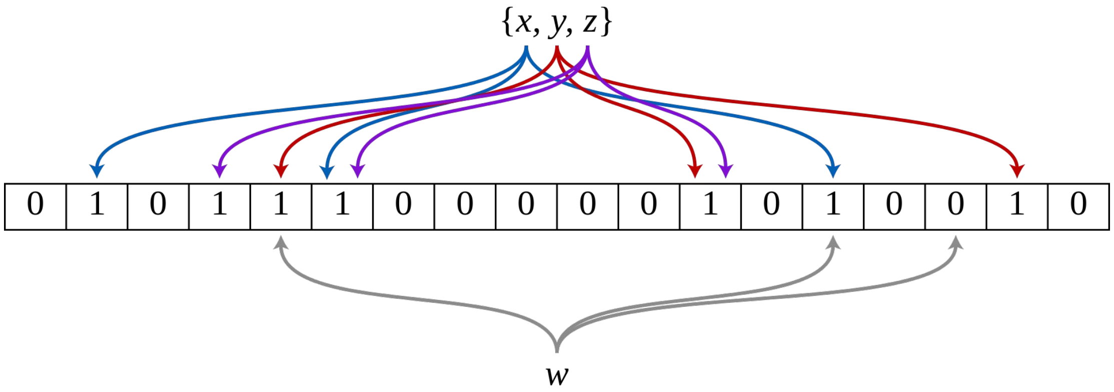
- Read
  - Point Query
  - Range Query
- Write(Insert/Delete/Update)
- Compaction
  - Leveled
  - Tired

Point Query



- **Returns immediately when something found**
- Read amplification, worst-case I/O: 2 * (N - 1 + files num of level-0)
- Optimization
  - Page cache/Block cache
  - Bloom filter

Bloom Filter

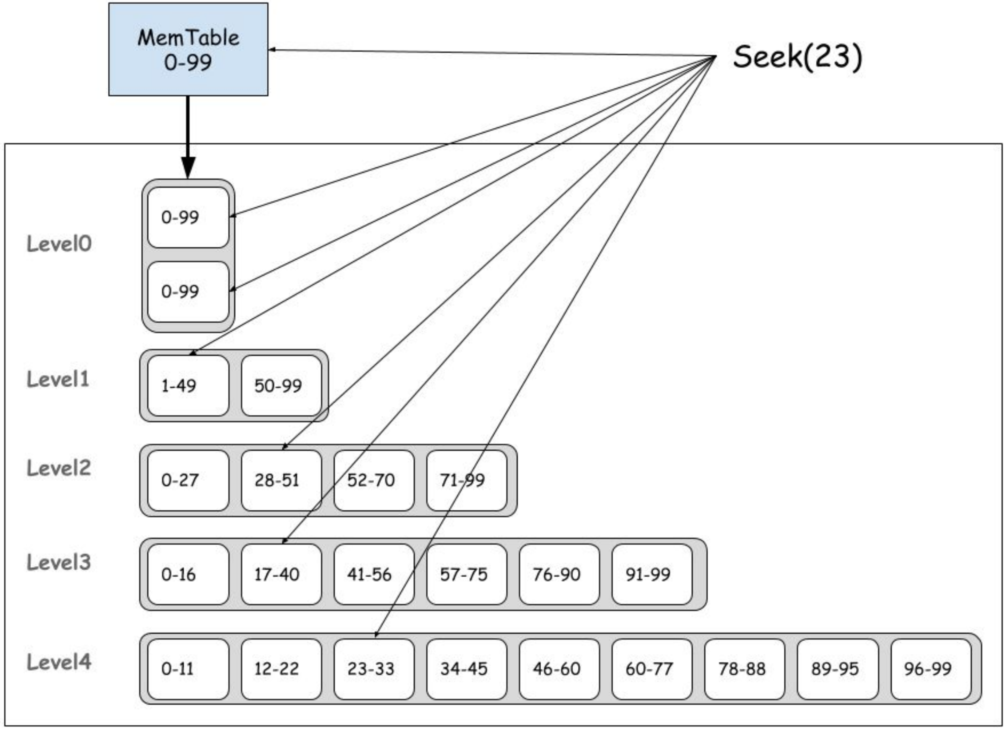


$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p$$

$$k = -\frac{\ln p}{\ln 2} = -\log_2 p$$

- Simple
- Space-efficient
- False positive is possible, buf false negative is impossible
- Range query is **not** supported

Range Query



SELECT * FROM t WHERE key >= 23 AND key < 40;

```
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)
  - ❏ SuRF (SIGMOD 2018)

SuRF: **Succinct Range Filter**

- SuRF: Practical Range Query Filtering with Fast Succinct Tries
- Fast Succinct Tries
  - **L**evel-**O**rdered **U**nary **D**egree **S**equence
  - LOUDS-**Dense**
    - The upper levels using a fast bitmap-based encoding scheme.
    - Choosing performance over space.
  - LOUDS-**Sparse**
    - The lower levels using the space-efficient encoding scheme.
    - Choosing space over performance.
  - LOUDS-DS

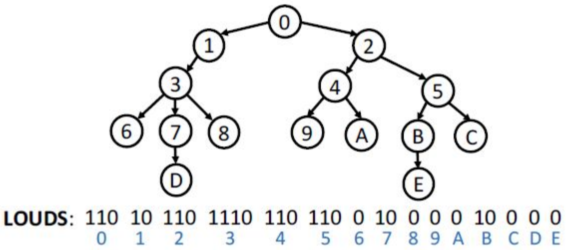


Figure 1: An example ordinal tree encoded using LOUDS

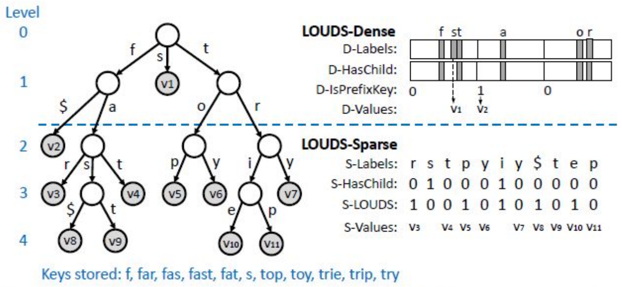
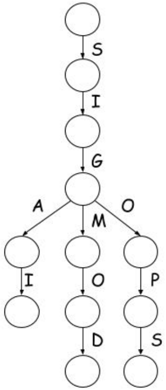


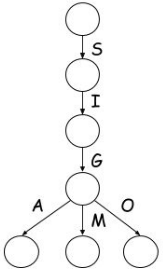
Figure 2: LOUDS-DS Encoded Trie – The $ symbol represents the character whose ASCII number is 0xFF. It is used to indicate the situation where a prefix string leading to a node is also a valid key.

SuRF: **S**u**ccinct R**ange **F**ilter

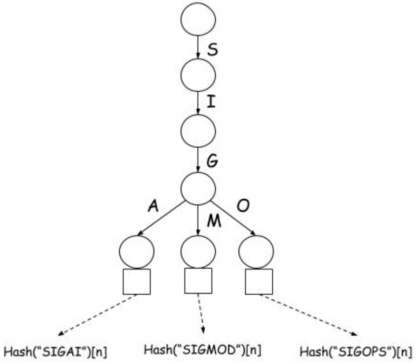
- Trie => SuRF-Base => SuRF-Hash/SuRF-Real
- Example: ("SIGAI", "SIGMOD", "SIGOPS")



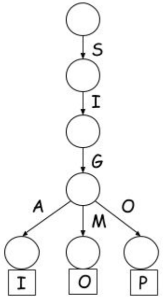
Trie



SuRF-Base stores the minimum-length key prefixes such that it can uniquely identify each key.
Specifically, SuRF-Base only stores an additional byte for each key beyond the shared prefixes.



SuRF-Hash adds a few hash bits per key to SuRF-Base to reduce its FPR.
The extra bits in SuRF-Hash do not help range queries because they do not provide ordering information on keys.



SuRF-Real stores the n key bits immediately following the stored prefix of a key.
Both point and range queries benefit from the real suffix bits to reduce false positives.

SuRF: Succinct Range Filter

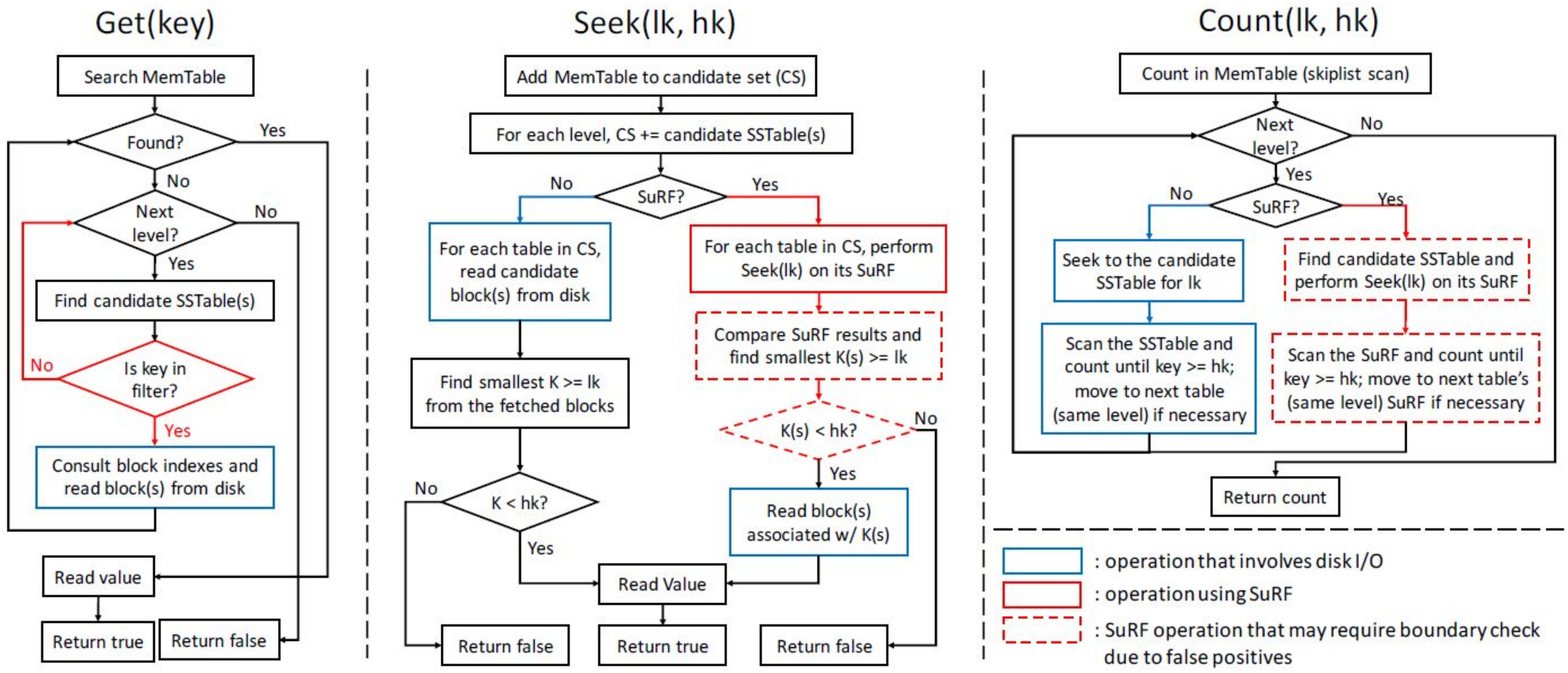
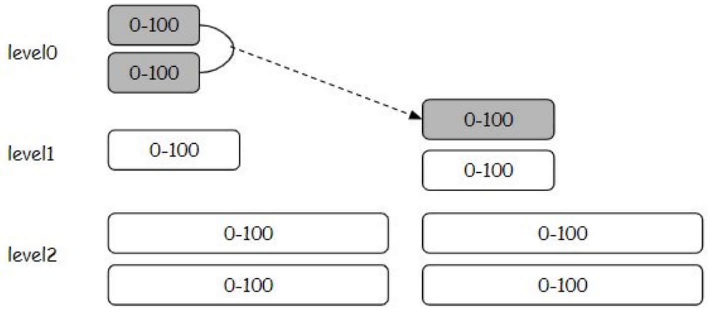


Figure 11: Execution paths for Get, Seek, and Count in RocksDB

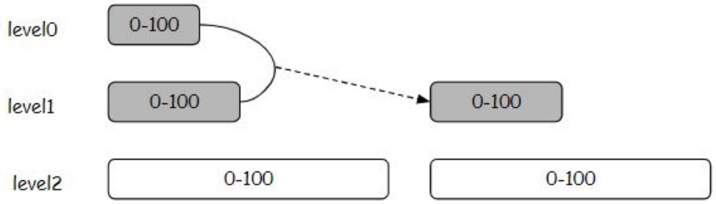
Read Operation Summary

- Read amplification ⇒ Use filters to reduce unnecessary I/O
- Space-efficient
- Lookup performance
- Low false positive, no false negative
- Bloom filter/Prefix bloom filter
- SuRF
- Others: Cuckoo Filter...

Compaction - Tiered vs Leveled



Tired Compaction

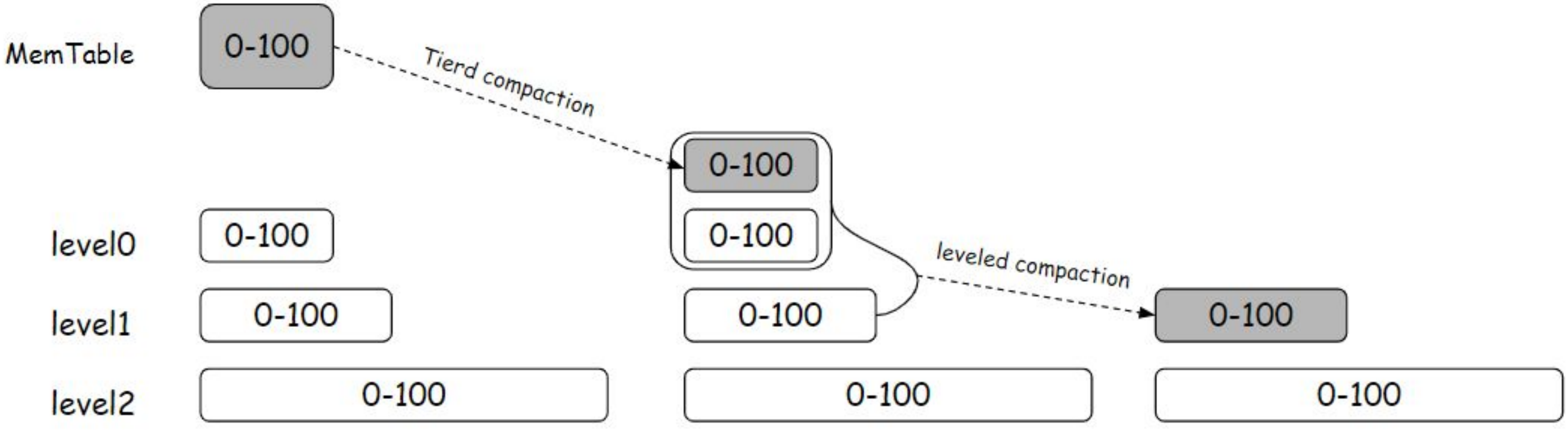


Leveled Compaction

- 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.

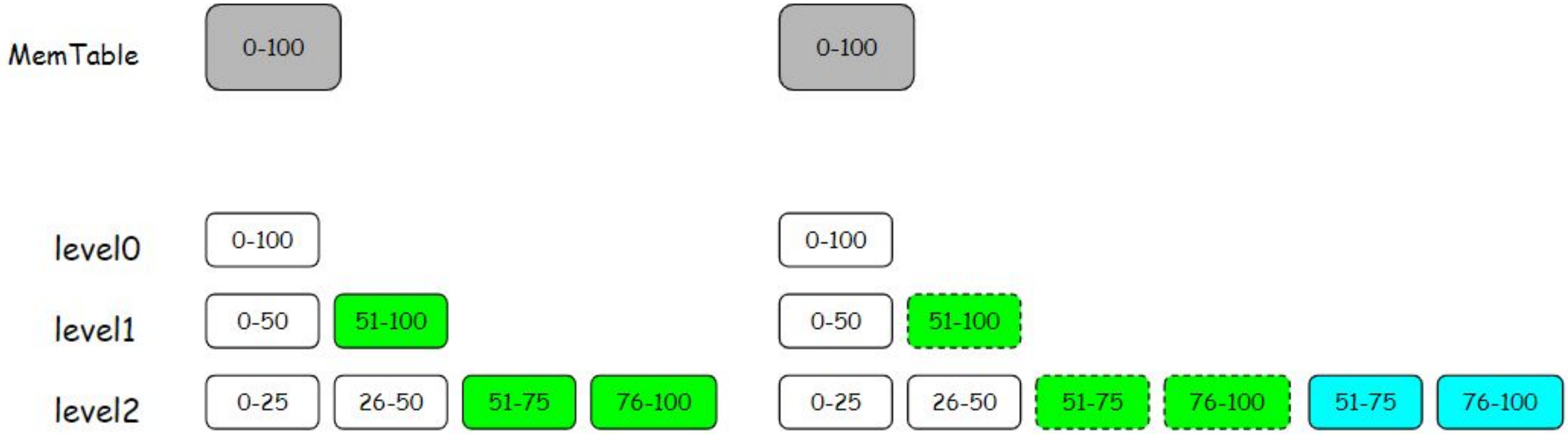
- Each level is **one sorted run**.
- Compaction into Ln merges data from Ln-1 into Ln.
- Compaction into Ln rewrites data that was previously merged into Ln.
- Minimizes space amplification and read amplification at the cost of write amplification.

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.

Tiered + Leveled + Partition

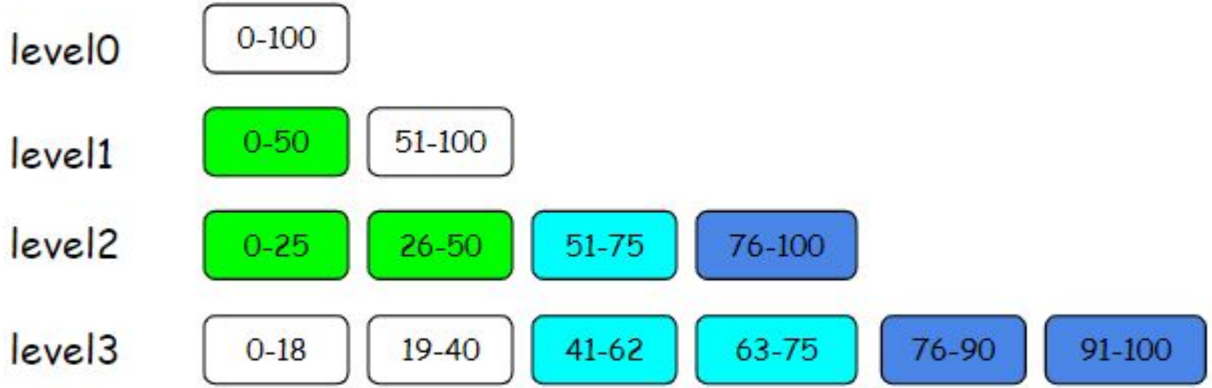


- Default compaction of LevelDB and RocksDB.

Problems of compaction

- Write amplification
  - System jitter(CPU、I/O)
  - SSD wear out
- Invalid block cache/page cache
- Compaction rate is a problem
  - Too fast - write amplification
  - Too slow - read amplification and space amplification.

Multi-Thread Compaction



Pipelined Compaction

- Pipelined Compaction for the LSM-tree
- The compaction procedure:
  - **Step 1: Read data blocks.**
  - Step2: Checksum.
  - Step3: Decompress.
  - Step4: Merge sort.
  - Step5: Compress.
  - Step6: Re-checksum.
  - **Step7: Write to the disk.**

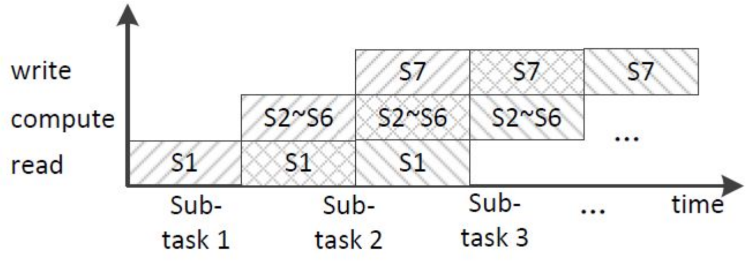
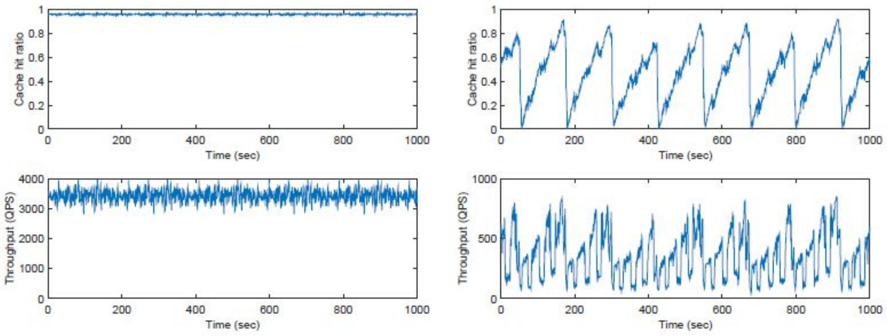


Fig. 4. Ideal Pipelined Compaction Procedure: *Step 1* and *Step 7* are scheduled on disk, and all the other steps are scheduled on CPU.

- Read -> Compute -> Write
  - It is difficult to devide the stages evenly.
  - If data blocks must flows through multiple processors, it will result in low CPU cache performance. Let S2~S6 as on stage will be more CPU cache friendly.

Compaction Buffer

- ❑ Re-enabling high-speed caching for LSM-trees
- ❑ LSbM-tree: Re-enabling Buffer Caching in Data Management for Mixed Reads and Writes



(a) Read only workload (b) Read and write workload

Fig. 2. The failure of caching in LSM-tree

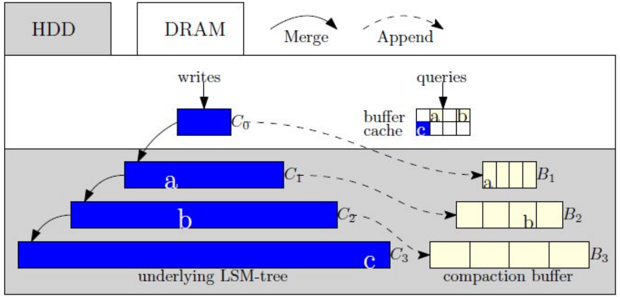


Fig. 5: The basic structure of a LSbM-tree

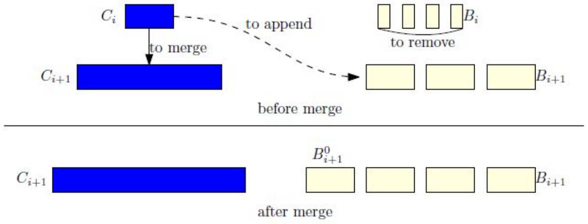
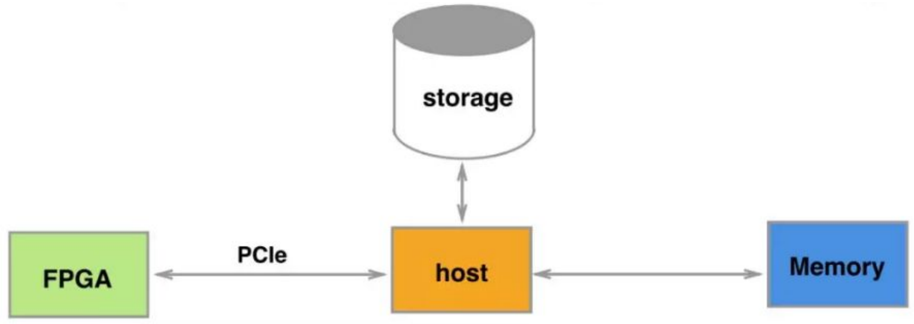


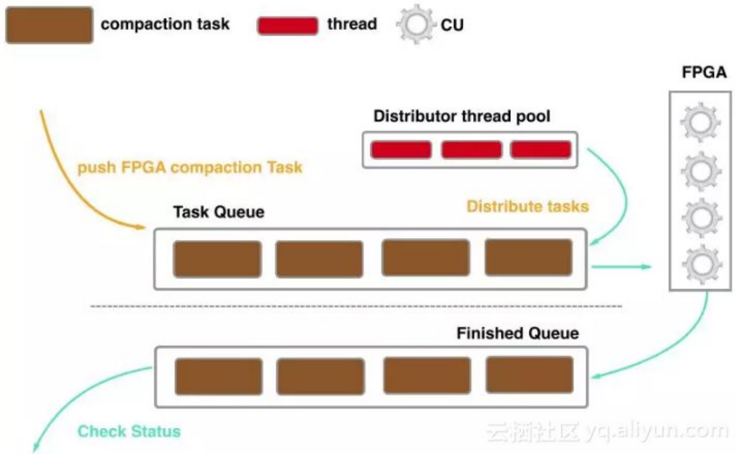
Fig. 6: An illustration of the buffered merge

Coprocessor

- ❑ Co-KV: A Collaborative Key-Value Store Using Near-Data Processing to Improve Compaction for the LSM-tree
- ❑ 当数据库遇见FPGA:X-DB异构计算如何实现百万级TPS？

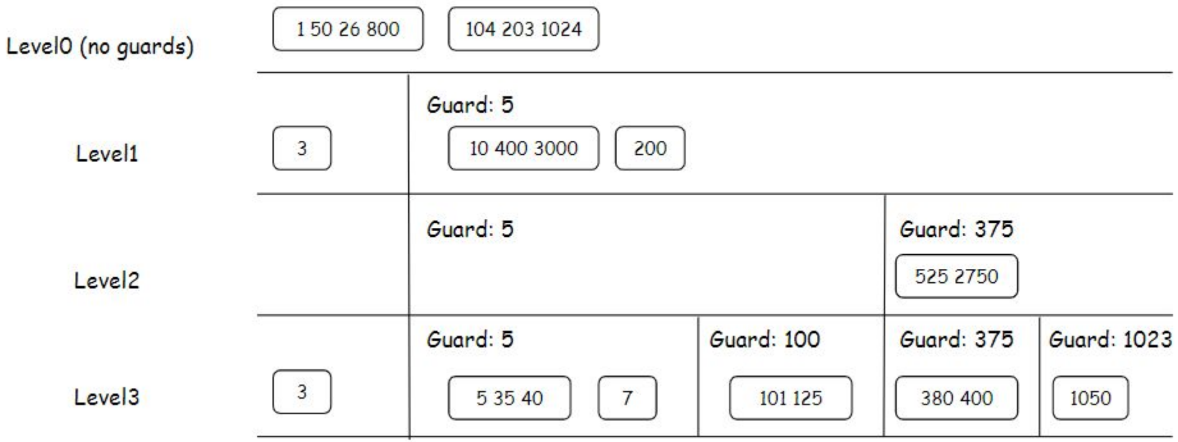


(b) Hybrid design, FPGA is integrated as a real co-processor.



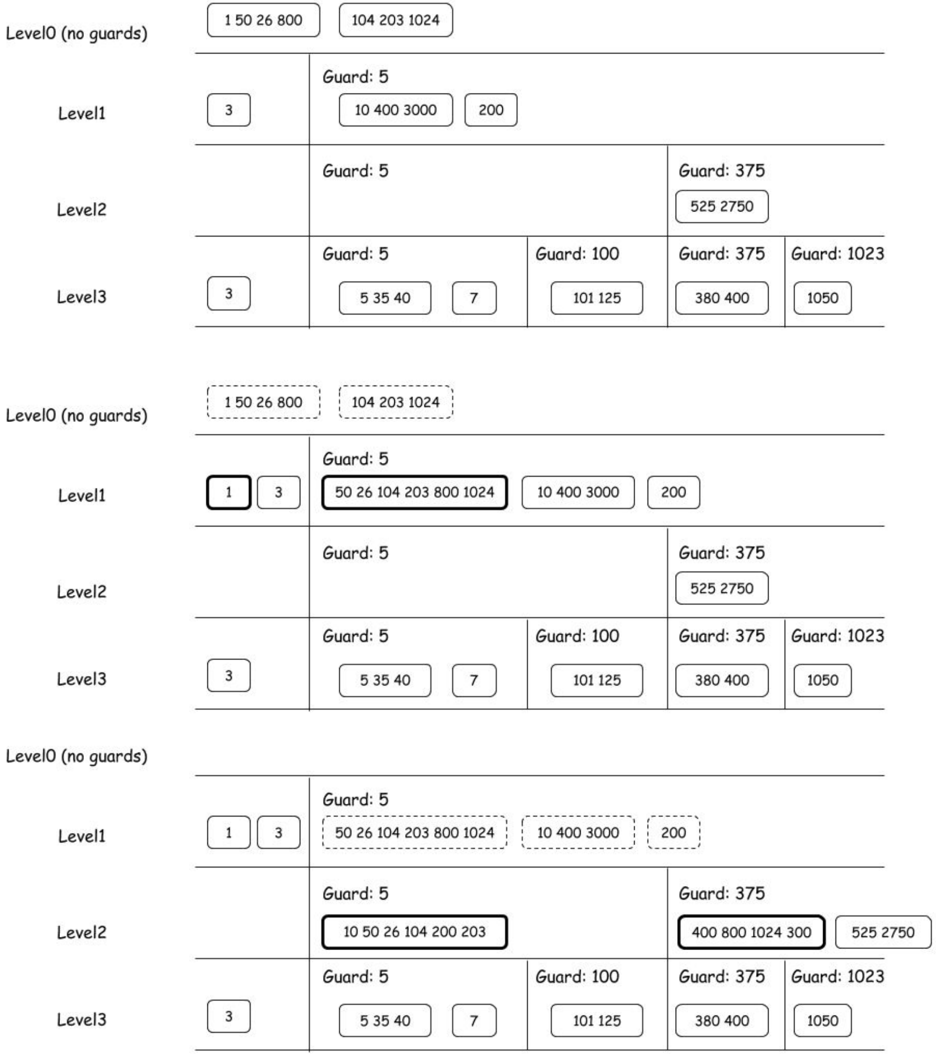
PebblesDB

- PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees
- **Guards** divide the key space (for that level) into disjoint units
- Less write amplification, more read amplification and space amplification.



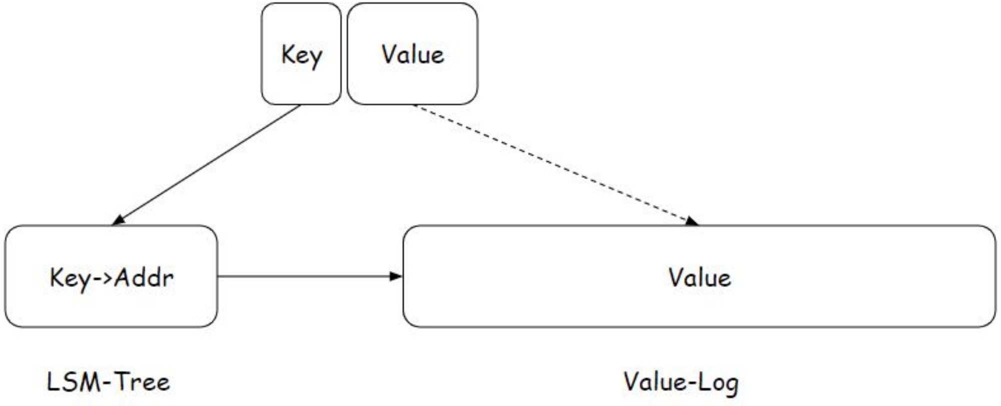
PebblesDB Compaction

- Similar to Tired + Partition



WiscKey

- **WiscKey: Separating Keys from Values in SSD-conscious Storage**
- Compaction = sorting + garbage collection
- Only keys are required to be sorted
- Keys are usually smaller than values
- Key-Value separation ⇒ Decouple sorting and garbage collection



WiscKey

- Key-Value separation’s challenges and optimizations
  - An additional I/O may be required for each query
    - WiscKey’s LSM-Tree is small and can be easily cached in memory
  - Range queries require random I/O
    - Prefetch: leverages the parallel I/O characteristic of SSD
  - Garbage Collection
    - Invalid keys are reclaimed by compaction
    - Value log needs a special garbage collector
  - Crash Consistency
    - Atomicity of inserted key-value pairs is complicated
  - Combine value log and WAL

Summary

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

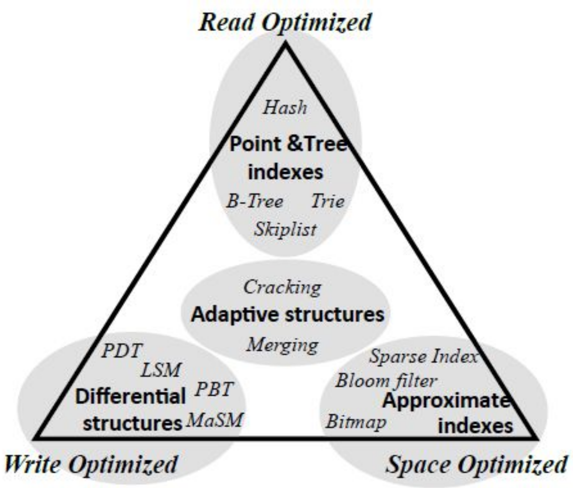


Figure 1: Popular data structures in the RUM space.

Q&A