Implementation of an LSM-Tree-Based Key-Value Store

Minghong Zou, Haonan Wu

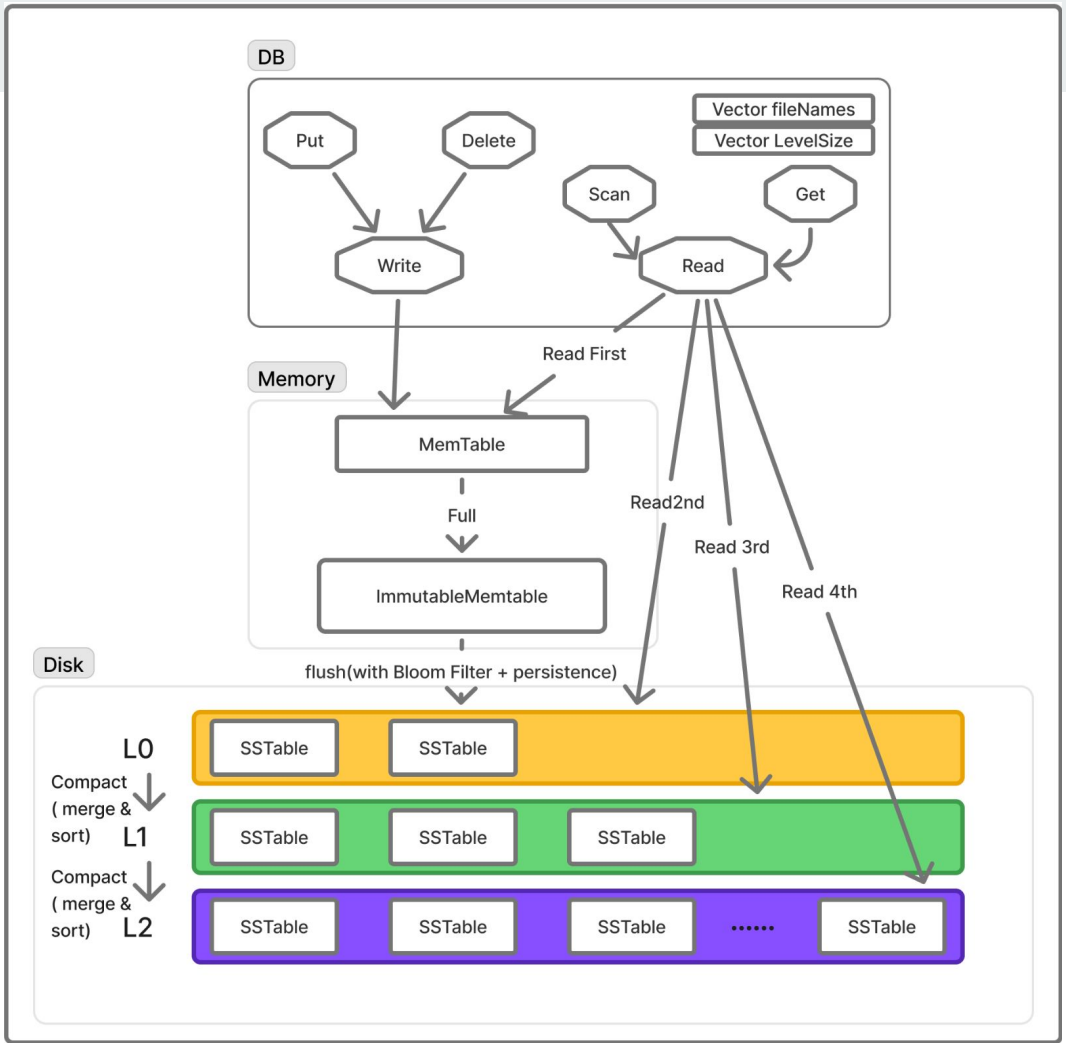
Motivation

LSM-tree is widely used in modern key-value stores such as RocksDB and LevelDB due to its write-optimized design. However, different compaction policies and features such as Bloom filters and range deletions have a significant impact on performance in real-world workloads.

Our goal is to implement a modular, high-performance LSM-tree from scratch to explore these tradeoffs, compare tiering vs leveling compaction, and investigate how features like Bloom filters affect query latency. This also serves as a practical way to deepen our understanding of the internal mechanics of modern storage engines.

Implementation

- **Implemented both Leveling and Tiering Compaction Strategies**
  a. Leveling: older SSTables are merged and deduplicated
  b. Tiering: newer SSTables are merged within the same level
- **Used SkipList in MemTable**
- **Bloom Filter Integration**
  - Each SSTable has a Bloom filter stored **persistently** to reduce false reads
- **Compact and Scan Logic**
  - Implemented key de-duplication, range tombstone handling, and merge-heap scan


DB
Put
Delete
Write
Scan
Get
Read
Vector fileNames
Vector LevelSize
Read First
Memory
MemTable
Full
ImmutableMemtable
Read2nd
Read 3rd
Read 4th
Disk
flush(with Bloom Filter + persistence)
L0
Compact
(merge &
sort)
L1
Compact
(merge &
sort)
L2
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
.....
SSTable

Saving Format

```
# header
10                  ← All size
3                   ← range tomb size
10                  ← min
70                  ← max
1                   ← start seq
10                  ← bit per elements
46                  ← entries offset
120                 ← range delete offset
144                 ← key index offset
xx                  ← numElements
xxx                 ← Bloom filter offset
# entries
1 0 10 1 3          ← Put(10, [1,3]) seq 1
2 0 20 2 3
3 0 30 3 3
4 0 40 4 4
6 0 50 5 3
10 1 70             ← point delete(70) seq 10
9 0 70 7 3
# range tombstones
7 15 35             ← DeleteRange(15,35) seq 7
8 25 45
5 42 60
# Bloom filter
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ← just for show, only include 0 or 1
# key index offset list
10 46               ← Key offset key 10 offset 46
20 57
30 68
40 79
50 90
70 101
```

Compaction-Leveling

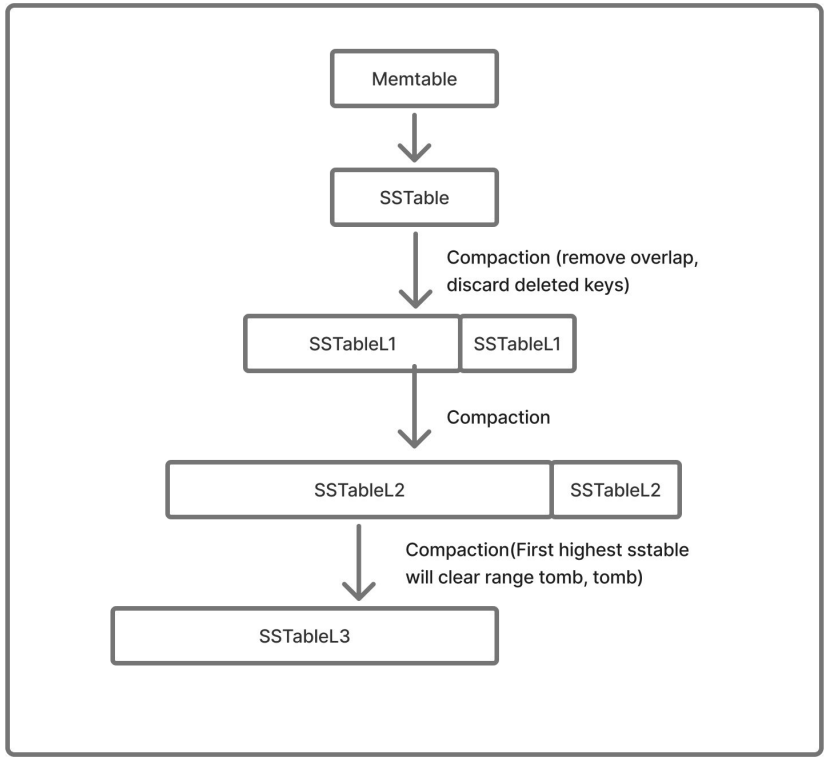
- Always select the **oldest SSTable** from the current level as the compaction source
- Find SSTables in the **next level** that overlap in key range
- Use a **min-heap** to merge entries; keep the one with the **highest sequence number**
- Skip entries **covered by range tombstones**
- Write the merged result to the next level and delete old files
- If the target level exceeds its size limit, **trigger recursive compaction**

If at the bottom level (`level >= max_level`):

- No further compaction is performed
- Clean up entries deleted or covered by range tombstones
- If result is empty → **delete the file**
- Otherwise, **create a new SSTable** and dynamically expand to a new level

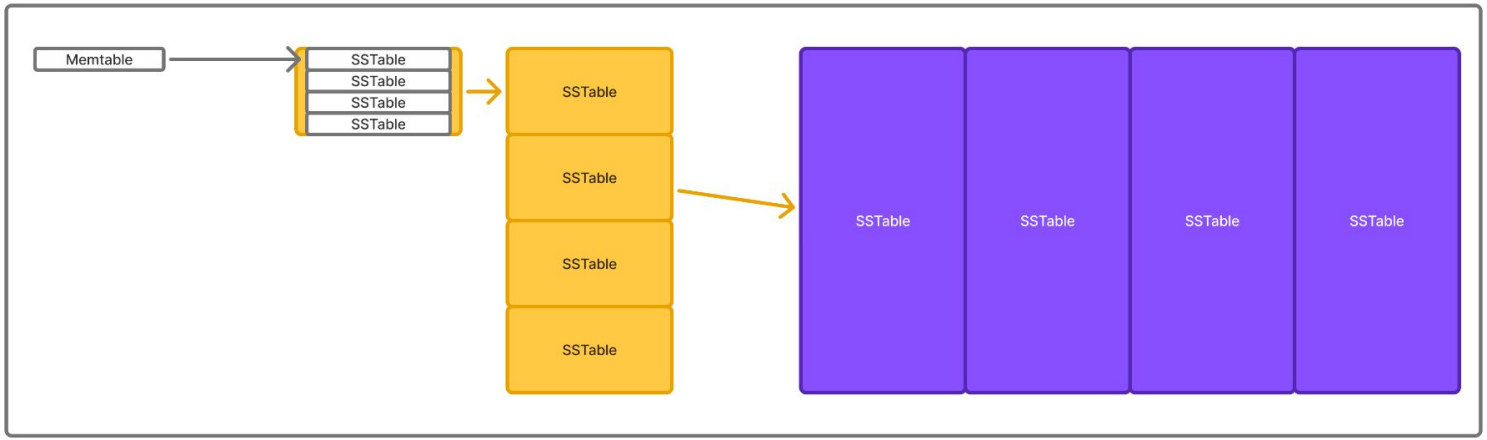
After compaction:

- Delete all input files from current levels
- Update vectors (`sstables_file`, `levels_size`)
- If the next level is too large, **recursively call compaction**


Memtable
SSTable
Compaction (remove overlap,
discard deleted keys)
SSTableL1
SSTableL1
Compaction
SSTableL2
SSTableL2
Compaction(First highest sstable
will clear range tomb, tomb)
SSTableL3

Compaction-Tiering

- Each level holds **multiple overlapping SSTables**
- Once the **total number of entries** in the level exceeds a threshold, trigger compaction
- All entries and range tombstones from the level are **merged directly** into one SSTable
- No deduplication or tombstone filtering is performed during merge
- The result SSTable is written to the **next level**

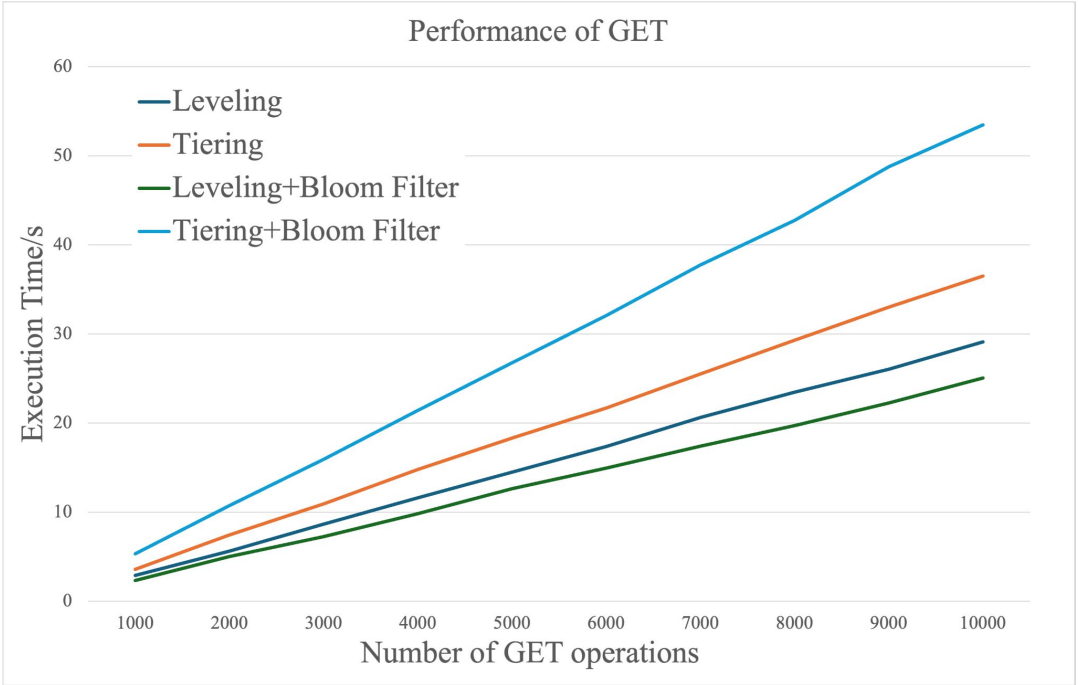

Memtable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable

Experiment

- Pure Workloads: GET, SCAN, PUT, DELETE
- Mixed Workloads: PUT+DELETE, PUT+GET
- Level Ratio T

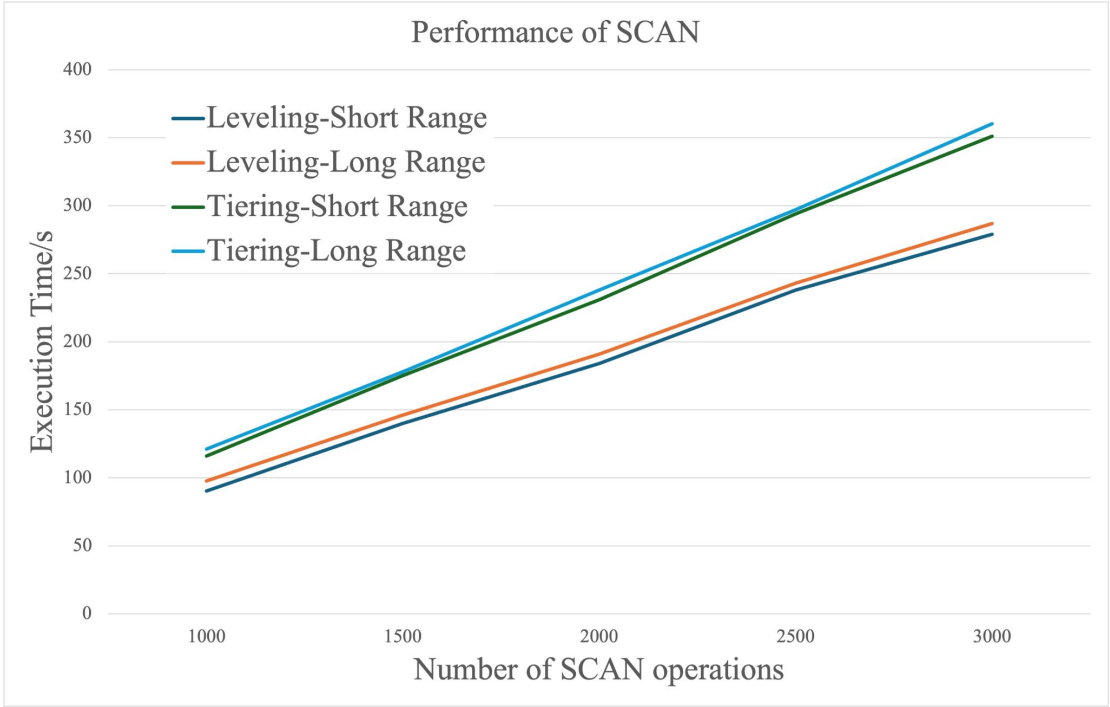
GET-Only Workload

- Bloom filters speed up GET in **Leveling**
- But may less effective in **Tiering**, and in some time may increase query time due to:
  - In Tiering, each level contains multiple overlapping SSTables
  - The same key may appear in several SSTables
  - Bloom filters are applied per SSTable and often return false positives



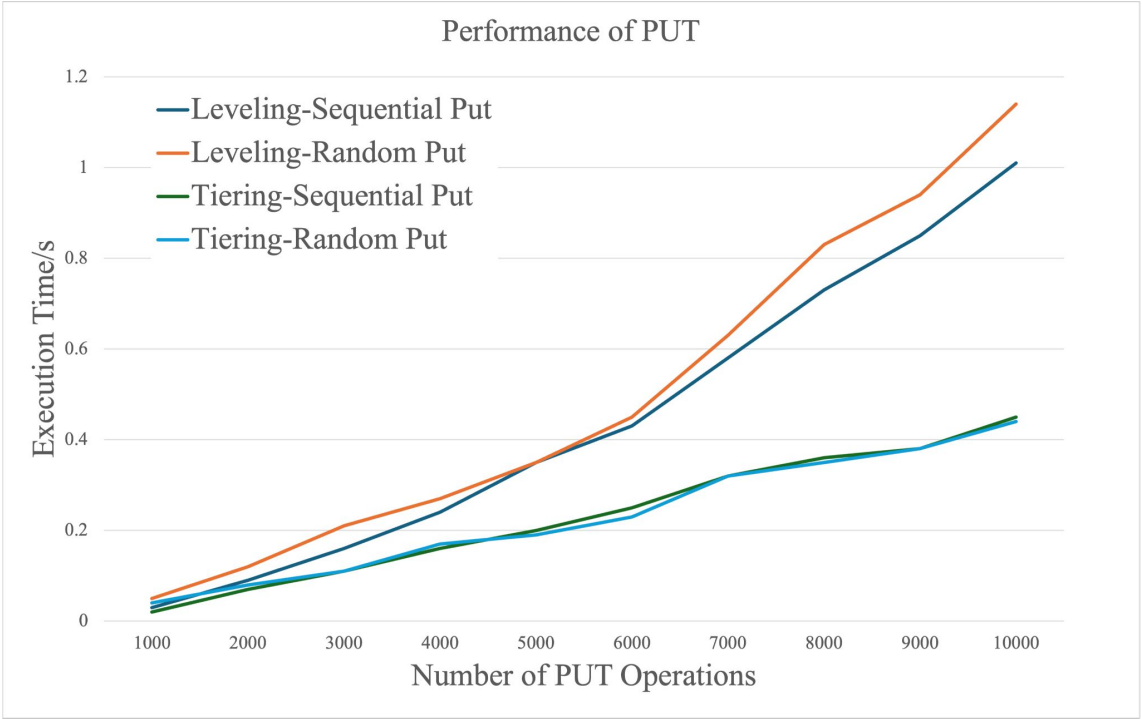
SCAN-Only Workload

- **Leveling** outperforms **Tiering**
  - Leveling keeps data with non-overlapping files
  - Tiering accumulates many overlapping files per level, increasing merge complexity and degrading scan efficiency
- The performance gap between short-range and long-range scans is minimal for both strategies.



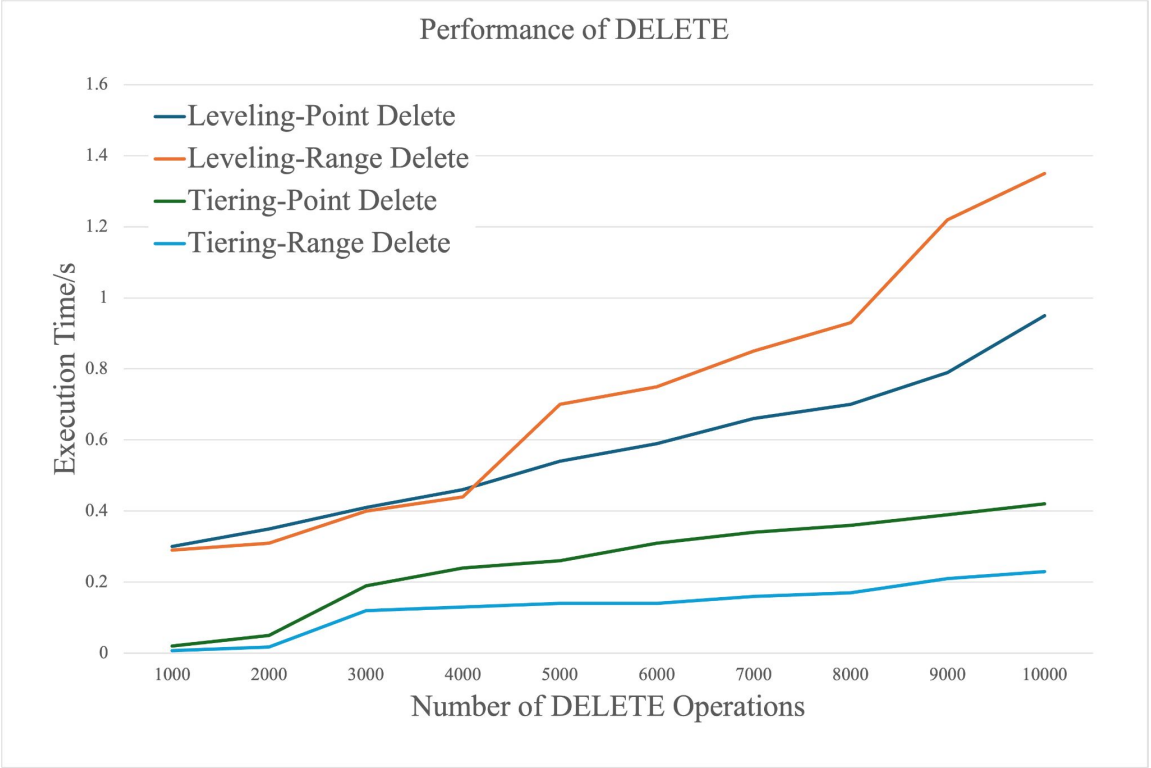
PUT-Only Workload

- **Tiering** is more efficiently
  - Less frequent compactions reduce write amplification
- Write pattern (**sequential vs random**) has minimal impact
  - All writes go through MemTable and are flushed to disk in **sorted order**



DELETE-Only Workload

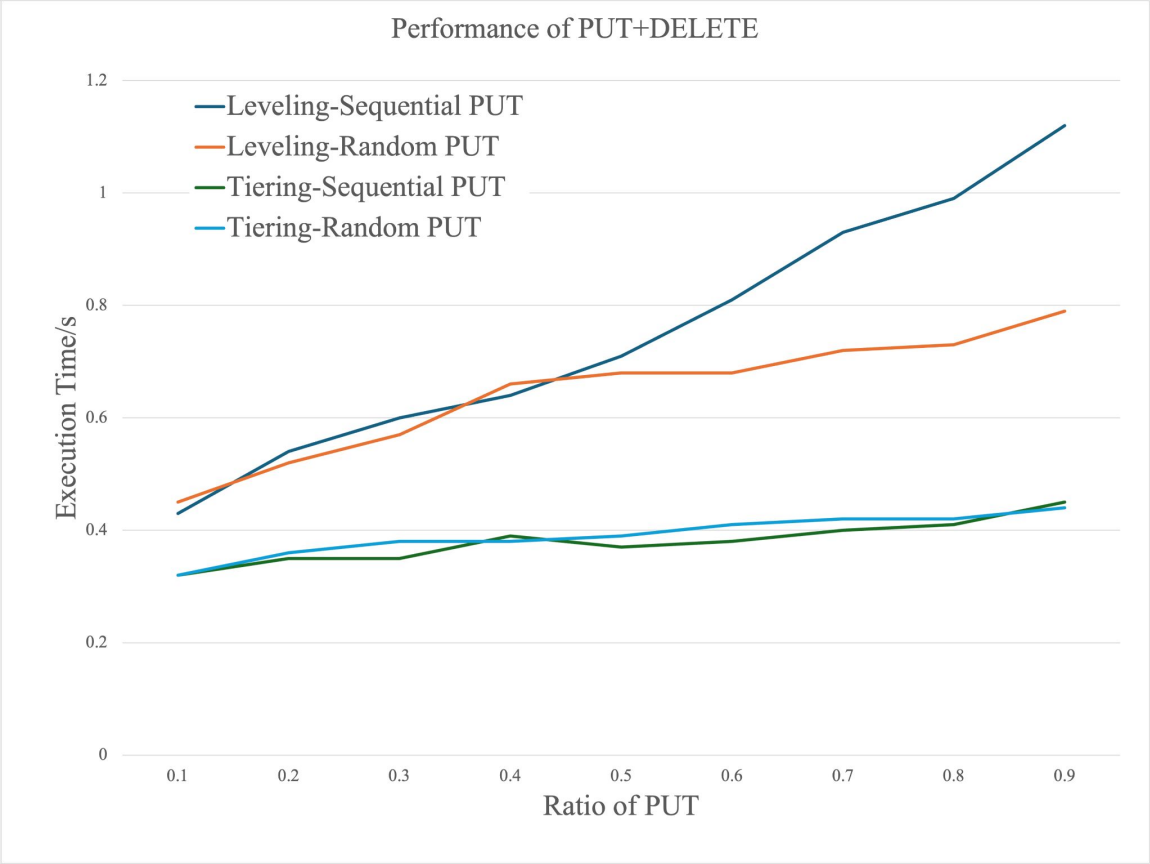
- DELETE is similar to PUT —> **Tiering** outperforms **Leveling**
  - Tiering keeps tombstones longer without merging
- In **Leveling**, range deletes trigger costly compactions and repeated merges
- In **Tiering**, range tombstones are lazily handled



PUT-Range DELETE Workload

- Write-only workloads —> **Tiering** outperforms **Leveling**

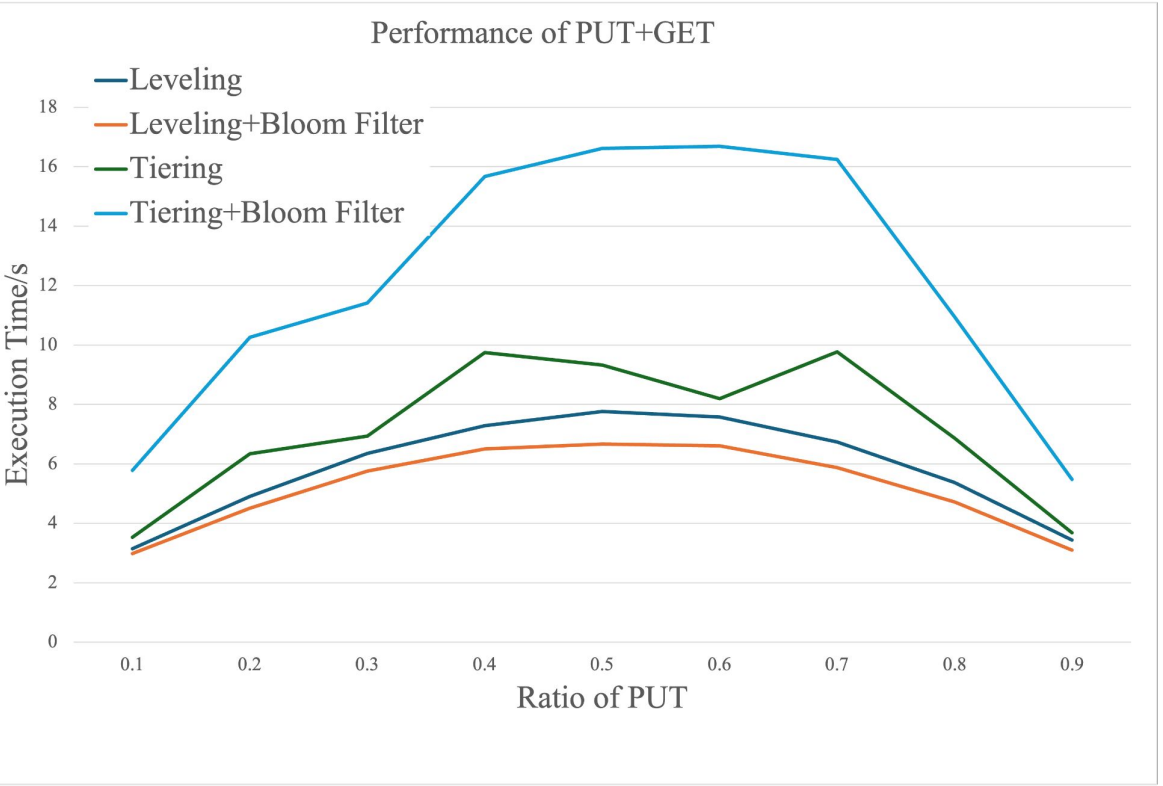
- **Leveling** performs worse **under sequential writes**
  - Sequential PUTs overlap heavily with range tombstones
  - This triggers large compactions, increases write amplification



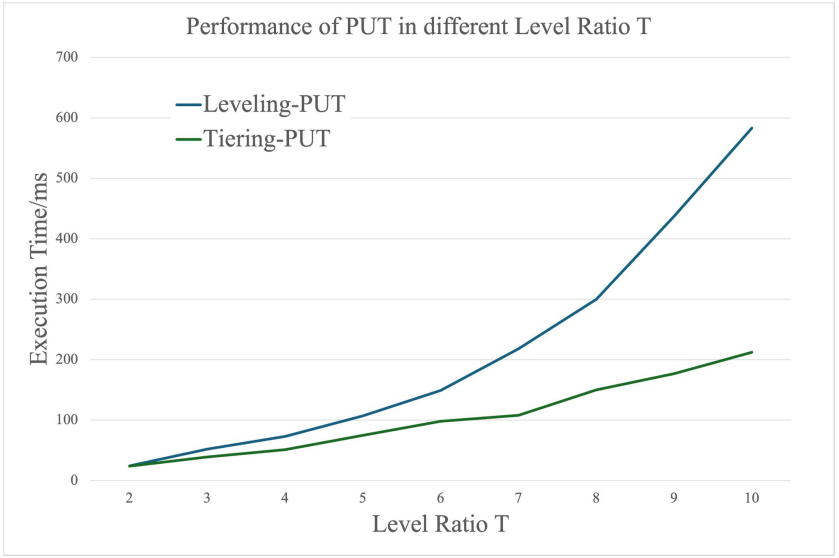
PUT-GET Workload

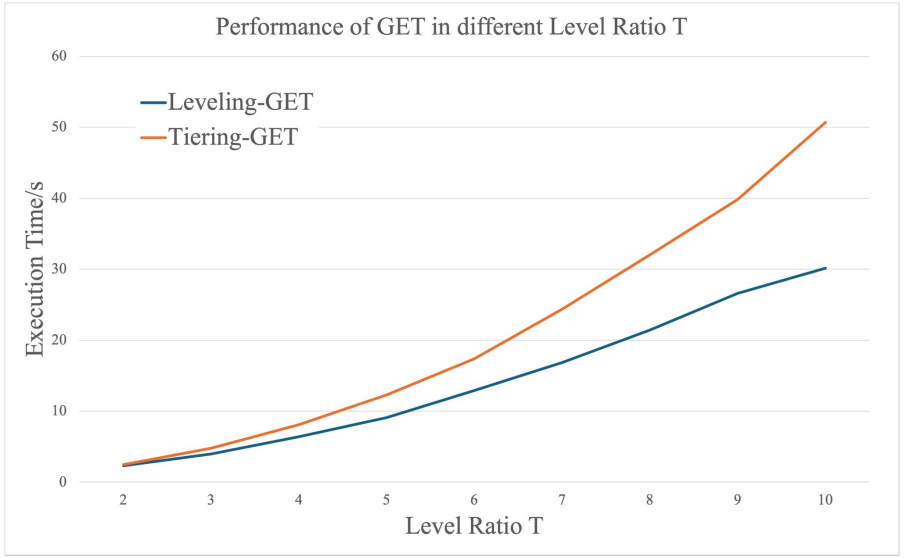
- PUT+GET workload behaves similarly to pure GET
  - Bloom filters are still effective in **Leveling**
  - File overlap in **Tiering** limits Bloom filter usefulness

- Peak overhead occurs at **PUT ratio ≈ 0.5**
  - **Read** and **write** operations interfere with each other



Impact of Level Ratio T





- As level ratio T increases, compaction cost grows faster in Leveling than in Tiering
- **Leveling** forces aggressive merge to maintain non-overlapping layout

- As level ratio T increases, GET cost grows faster in Tiering than in Leveling
- **Tiering** accumulates more overlapping files, increasing lookup overhead

Summary

- **Tiering performs better in write-heavy workloads**
- **Leveling is superior for point queries and scan-heavy workloads**
- **Bloom filters help Leveling, but are less effective in Tiering due to overlapping files**
- **Leveling's PUT performance and Tiering's GET is more sensitive to the level ratio T than in Leveling, due to increasing overlap among files**

Thank You!