Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree

Conference：2021 IEEE 37th ICDE

Authors: Kecheng Huang∗, Zhiping Jia∗, Zhaoyan Shen∗, Zili Shao†, and Feng Chen‡

University: ∗Shandong University, †The Chinese University of Hong Kong, and ‡Louisiana State University

Background

山东大学 SHANDONG UNIVERSITY

➢ LSM-tree based KV Store

- Components
  - In-memory
  - On-disk
- Write operations
  - Minor Compaction
  - Major Compaction
- Query
  - From L0 to Ln

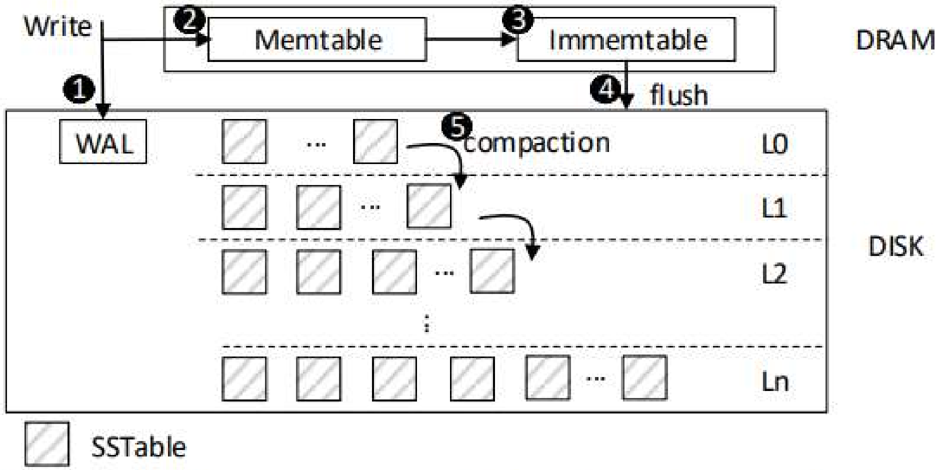


Figure 1: The structure of LSM-tree based KV store.

Problem

- **A small number of frequently updated key-value items could quickly pollute the entire tree structure, causing repeated changes in the structure and quickly amplifying the amount of disk IOs across the levels in the tree.**

Motivation

- **Intensive and time-consuming maintenance operations, incurring huge I/O overheads.**
  - A simple example
  - Preliminary test on LevelDB

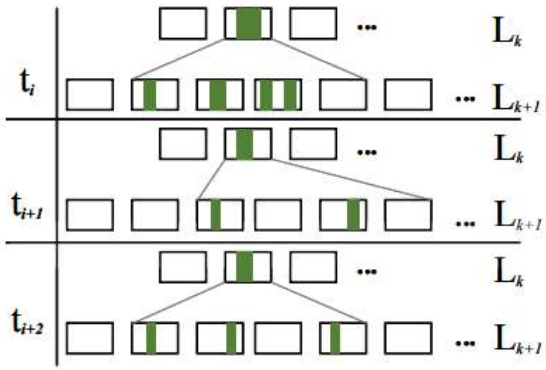


Fig. 1. An illustration of maintenance overhead in LSM-tree.

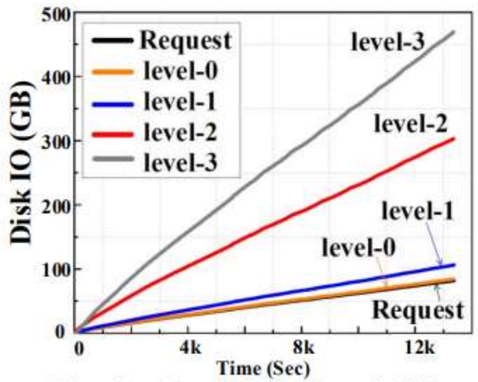


Fig. 2. Total disk IOs of different levels.

Solution

- We present a novel scheme, called Log-assisted LSM-tree (L2SM), which adopts a small-size, multi-level log structure to isolate selected key-value items that have a disruptive effect on the tree structure, accumulates and absorbs the repeated updates in a highly efficient manner, and removes obsolete and deleted key-value items at an early stage.

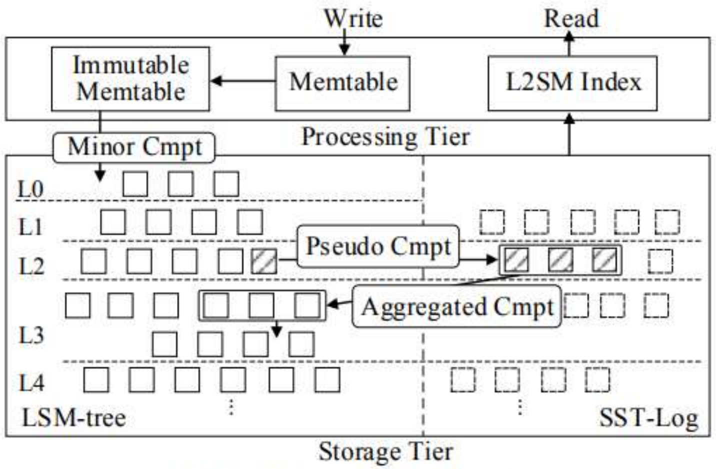


Fig. 3. Overview of L2SM architecture.

Challenge Issues & Techniques

➢ **How to retain the advantages of the LSM-tree based structure but avoid its negative effects?**

✓ **KV data that are rarely updated and have dense key coverage are maintained in the LSM-tree part.**

✓ **Extend the current LSM-tree with a separated log structure, called SST-Log.**

- A buffer to isolate KV items that receive frequent updates.
- Identify and move "sparse" SSTables out of the tree, making them condensed in the log.
- Delay changes to the LSM-tree, collapsing multiple changes into a few number of operations.

Challenge Issues & Techniques

➢ How to identify the frequently updated and the sparse SSTables?

✓ Hotness

- Bloom Filter
- Hotness Detecting Bitmap
  - *M* aligned bloom filters
  - *i*-th update to a KV item sets the corresponding bits in the *i*-th bloom filter
- Hotness value calculation
  - $\sum_{i=1}^{m-1}(x_i \times 2^i)$
  - *Xi:* the number of keys being updated for *i* times
  - Higher layer with higher weights
- Configuring HotMap
  - *M: 5*
  - *P: 4 million bits*
  - Auto-tuning

$$M = \lceil r/n \rceil$$

$$P = \frac{K \times N}{\ln 2}$$

- Overhead
  - HotMap ranges from 2.5 million to 40 million bytes
  - Hash functions incurring computational overhead

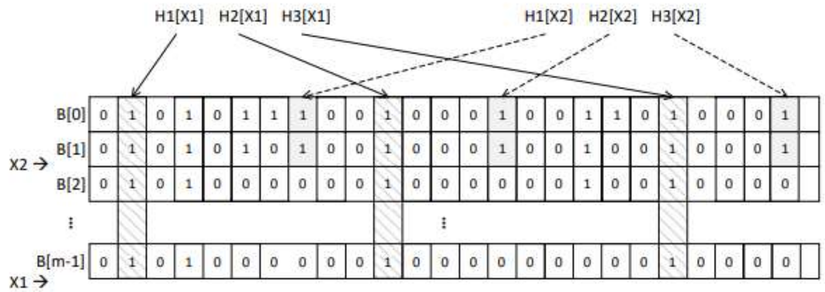


Fig. 4. An illustration of the HotMap scheme.

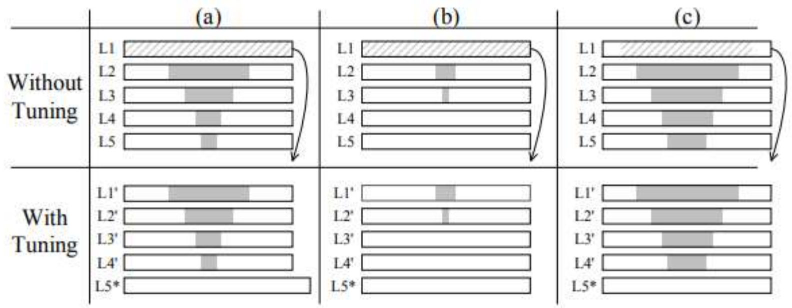


Fig. 5. An illustration of HotMap auto-tuning.

Challenge Issues & Techniques

➢ **How to identify the frequently updated and the sparse SSTables?**

✓ **Density**

- The ratio of the number of KV items to the key range of an SSTable to the key range of an SSTable
- Convert the keys into binary value (ASCII) and find the highest bit that differs in the two keys
  - the highest bit: i-th bit
  - roughly estimated key range: 2^i
  - k KV items in the SSTable
  - dstimated density: $k / 2^i \longrightarrow lg\ k - i$
  - $S = i - lg\ k$ (sparseness)

Challenge Issues & Techniques

➢ **How to do compaction operations?**

✓ **Pseudo Compaction (PC)**

- Extract selected SSTables from the LSM-tree into the SST-log
- Triggered when a tree level is full
- Victim SSTables
  - $W_i = \alpha \times \frac{H_i}{H_{max} - H_{min}} + (1 - \alpha) \times \frac{S_i}{S_{max} - S_{min}}$
  - Only metadata updates without physical data movement on disk
  - Organized in a linked list in SST-log
  - Repeat until the number of SSTables is below the limit
- Miscellaneous issues
  - Maintain in-memory bloom filters for SSTables in the log
  - Begin search from the newest SSTable

Challenge Issues & Techniques

➢ **How to do compaction operations?**

✓ **Aggregated Compaction (AC)**

- Retain the most structure-impactful SSTables in the log and return the cold and dense SSTables back to the lower level of the tree.
- Maintain the query correctness through a strict chronological order
- Consider both density and hotness
- Control the involves I/Os by estimating the number of involved SSTables
- Remove deleted and obsolete data
- Process
  - ■ Triggered when the log exceeds its size limit
  - ■ Calculate *W* of all the SSTables in the log
  - ■ Compaction Set and Involved Set (keeping the ratio of SSTables in the IS and CS lower than 10)
  - ■ Merge sort

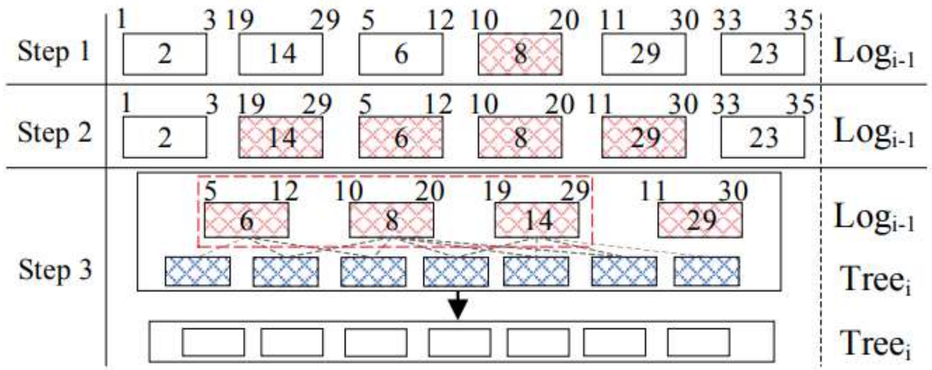


Fig. 6. An example for aggregated compaction.

Evaluation

➢ Overall Performance

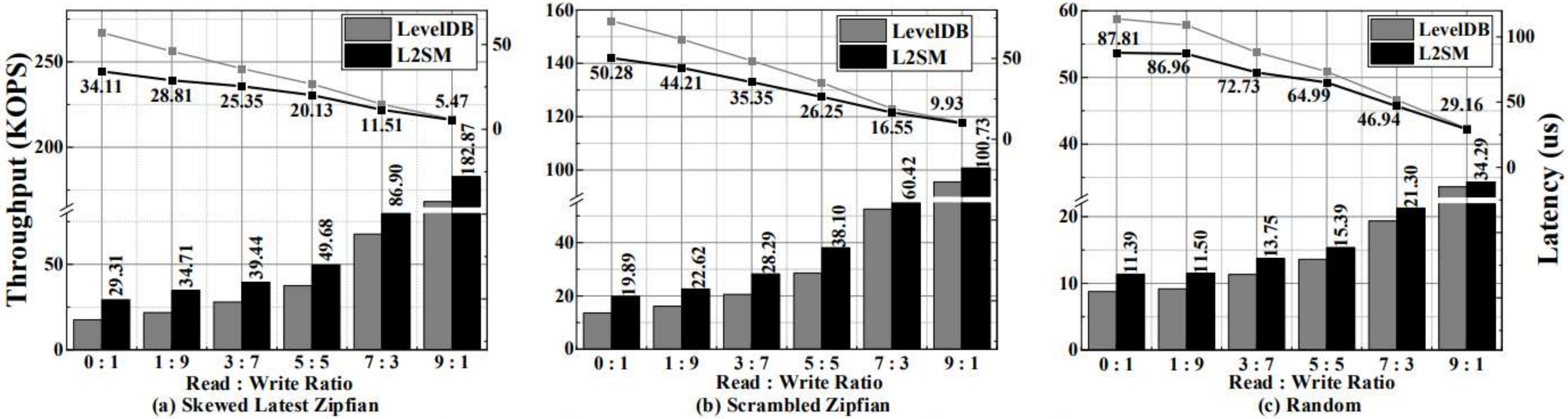


Fig. 7. Throughput and latency for workloads with different Read:Write ratios.

Evaluation

➢ Compaction Effect

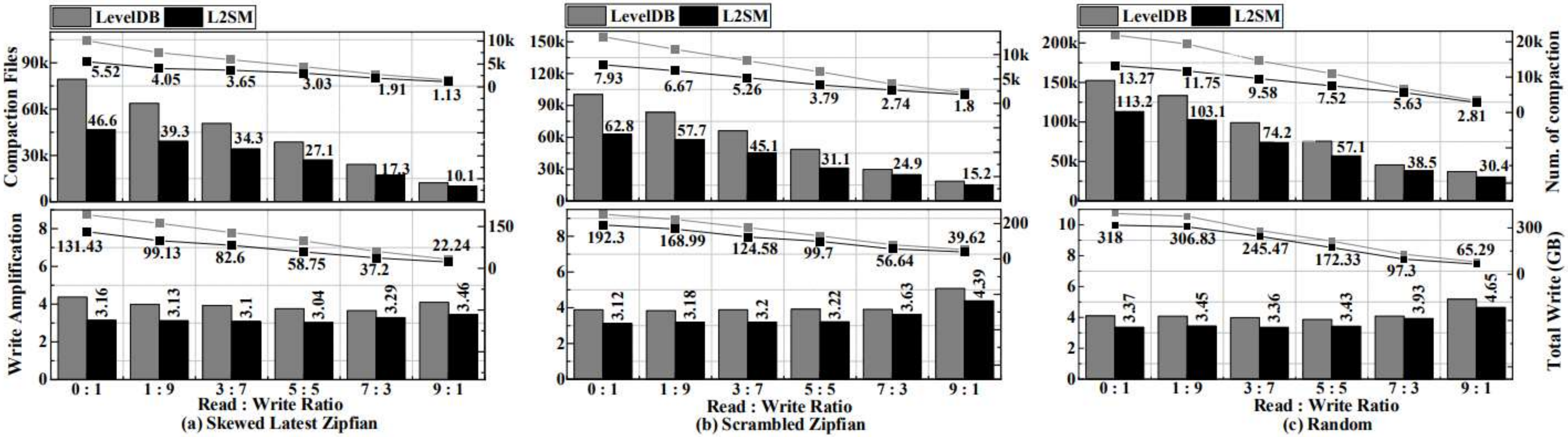


Fig. 8. Occurrences of compaction, involved files, write amplification and total writes for workloads with different Read:Write ratios.

Evaluation

➢ Read Limitation

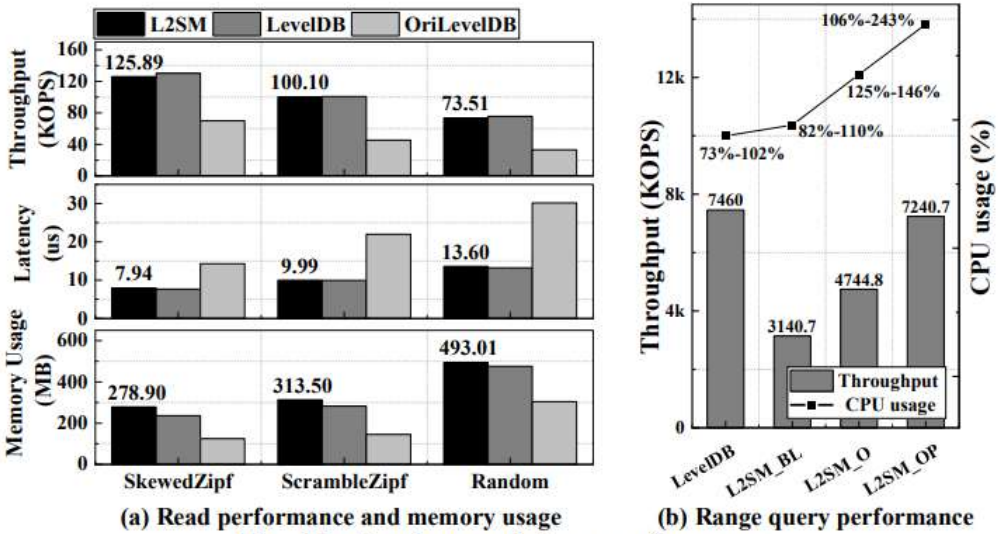


Fig. 11. Lookup and Scan performance.

Evaluation

➢ **Scalability**

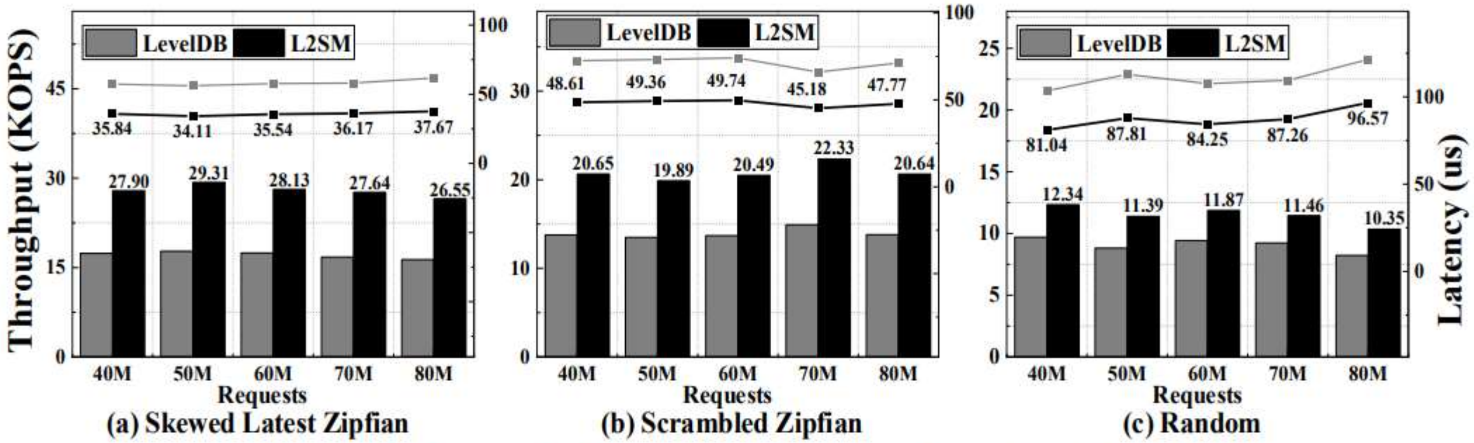


Fig. 9. Performance for workloads with different numbers of requests.

Evaluation

➢ Comparison with RocksDB and PebblesDB

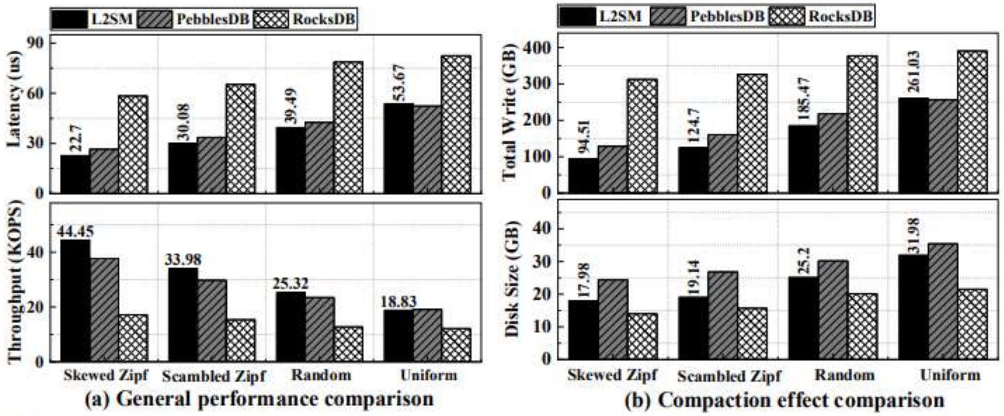


Fig. 12. Performance and I/O comparison with PebblesDB and RocksDB.

Evaluation

➢ Overhead Analysis

- Storage overhead (less than 10%)
- Memory overhead (ranges from 4.2% to 8.7%)

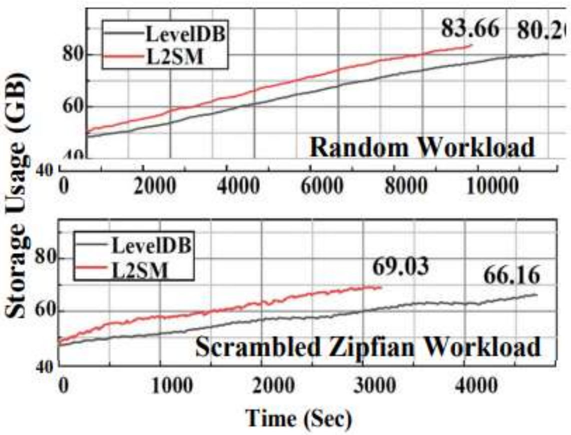
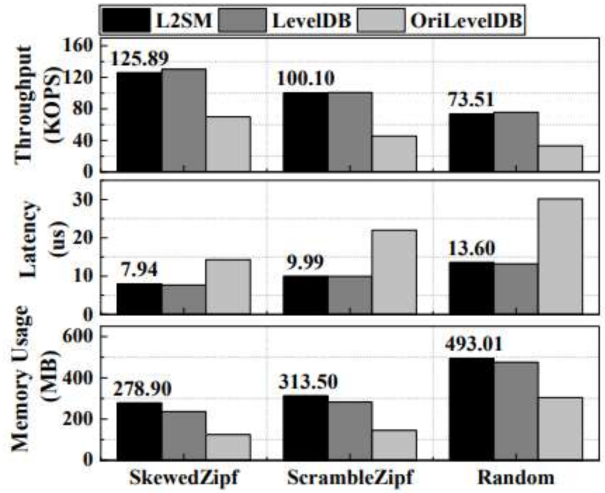


Fig. 10. Storage usage overhead.



(a) Read performance and memory usage

Thank You!