LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data

Xingbo Wu
Yuehai Xu
Song Jiang

Zili Shao

WAYNE STATE UNIVERSITY

The Hong Kong Polytechnic University
The Challenge on Today’s Key-Value Store

• Trends on workloads
  – Larger single-store capacity
    • Multi-TB SSD
    • Flash array of over 100 TB
  – Smaller key-value items
    • In a Facebook KV pool 99% of the items are ≤ 68B.

• Large metadata set on a single node
Consequences of a Large Metadata Set

- **Less** caching space for hot KV items.
  - Low hit ratio compromises system throughput.

- **Long** warm-up time.
  - It may take tens of minutes to read all metadata into memory.

- **High** read cost for out-of-core metadata.
  - It’s expensive to read multiple pages to serve a single GET.

- **LevelDB** has managed to reduce the metadata size.
LevelDB Reduces Metadata Size with SSTable

• To construct an **SSTable**:
  – Sort data into a list.
  – Build memory-efficient **block-index**.
  – Generate **Bloom filters** to avoid unnecessary reads.

• How to support insertions on SSTable?

![Diagram showing insertion process with Bloom filter](image)
Reorganizing Data Across Levels

• LSM-tree (Log-Structured Merge-tree)
  – New items are first accumulated in MemTable.
  – Each filled MemTable is converted to an SSTable at Level 0.
  – LevelDB conducts compaction to merge the SSTables.

• A store can exponentially grow to several TBs with a few levels.

MemTable

Level 0

Level 1

1-13  14-26  27-37  38-49  50-62  63-74  75-86  87-99

Very Expensive!

Compaction
A Closer Look at Compaction

Steps in compaction:
1. Read overlapping SSTables into memory.
2. Merge-Sort the data in memory to form a list of new SSTables.
3. Write the new SSTables onto the disk to replace the old ones.

In one compaction (1:10 size ratio):
- Read 11 tables
- Write 11 tables
- Add only 1 table to the lower level

45x WA for a 5-level store (WA: write amplification)
Compaction can be very Expensive

• The workload:
  – PUT 2 billion items of random keys (~250GB).
  – 16-byte key and 100-byte value.

• PUT throughput reduces to **18K QPS (2MB/s)**.
Metadata I/O is Inefficient

- Facts about LevelDB’s metadata:
  - Block Index: ~12 bytes per block.
  - Bloom filter: ~10 bits per key.
- How large is it in a 10-TB store of 100-byte KV items?
  - 155GB metadata: 30 GB block index + 125 GB Bloom filter.

4-GB memory holds 25% of the metadata for a 1-TB store.
Our solution: LSM-trie

• Build an ultra-large KV store for small data.
  – Using a trie structure to improve compaction efficiency.
  – Clustering Bloom filters for efficiently reading out-of-core metadata.
Organizing Tables in a Trie (Prefix Tree)

- KV items are located in the trie according to their hashed key.
- Each trie node contains a pile of immutable tables.
- The nodes at the same depth form a conceptual level.
- How does LSM-trie help with efficient compaction?
Efficient Compaction in LSM-trie

Compaction steps:
1. Read tables from the parent node into memory.
2. Assign the items to new tables according to hash-prefixes.
3. Write new tables into its child nodes.

For one compaction:
• Read 8 tables (1:8 fan-out)
• Write 8 tables
• Add 8 tables to the lower level

Only 5x WA for a 5-level LSM-trie

Tables linearly grow at each node
Introducing HTable*

• **HTable**: Immutable hash-table of key-value items
  – Each **bucket** has 4KB space by default.

• Some buckets have overflowed items.
  – Migrating the overflowed items.

*It’s not the HTable in HBase.*
Selecting Items for Migration

- Sorting items in a bucket according to their key’s hash.
- Migrating the items above the watermark (HashMark).
- Recording the HashMark and the corresponding IDs.
  - 2B Source ID, 2B Target ID, 4B HashMark

Move to another bucket  

HashMark: 0xa0

```
0xef
0xa0
0x9a
0x6d
0x56
0x33
0
```
Caching HashMarks for Efficient GETs

• Only cache HashMark for most overloaded buckets.
  
  – 1.01 amortized reads per GET.
  
  – A 1-TB store only needs ~400MB in-memory HashMark.

The only item that triggers a 2nd read.

Metadata in bucket 1
Bucket ID: 1
HashMark: 0x95
Target ID: 7

Item owned by Bucket 1
0x86
0x60
0x33
0x2a

Item owned by Bucket 7
0x95
0xac
0x10
0x35
Most Metadata is in the Last Level

- The first four levels contain 1.9 GB Bloom filters (BF).
- The last level may contain over one hundred GB BFs.
- We explicitly cache the BFs for Level 0 to Level 3.
- The BFs at the last level are managed differently.

Data size distribution across the levels:

- 256 MB
- 2 GB
- 16 GB
- 128 GB
- N TB

147 GB data
Clustering Bloom Filter for Efficient Read

• Each hash(key) indicates a **column** of 4-KB buckets.

• We collect all the BFs in a column to form a **BloomCluster**.

• Each GET requires **one SSD read** for all the out-of-core BFs.
Exploiting Full SSD Performance

• Using an additional small SSD to host **BloomClusters**.
  – e.g., a **10-TB** SSD for data + a **128-GB** SSD for metadata.

• Plenty of memory space is left for your **data cache**!

![Diagram with SSDs and Bloom-Clusters](image)
Performance Evaluation of PUT

- Expected high TP on SSD
- TP dropped due to static wear-leveling in SSD
- High TP lasts longer on two SSDs
- Consistent Throughput (TP) on HDD: 2x-3x higher than the others
Write Amplification Comparison

Consistent 5x WA ratio
Read Performance with 4GB Memory

Only one SSD is used.

*No data cache for LSM-trie*
Read Performance with 4GB Memory

Gains 96% raw SSD IOPS with an additional SSD.

*No data cache for LSM-trie
Summary

• LSM-trie is designed to manage a large set of small data.

• It reduces the write-amplification by an order of magnitude.

• It delivers high throughput even with out-of-core metadata.

The LSM-trie source code can be downloaded at:
https://github.com/wuxb45/lsm-trie-release
Thank you!

😊

Q & A