TreeLine: An Update-In-Place Key-Value Store for Modern Storage


Code: github.com/mitdbg/treeline
Paper: tinyurl.com/treeline-paper
What this talk is about

The Opportunity and Problem

• Disk-based key-value stores: usually log-structured merge trees (LSMs)
  • High write performance (sequential), competitive on reads
  • NVMe SSDs: Parallel random writes ≈ sequential write performance
  • Are LSMs still the right choice?

This Work: TreeLine – Making update-in-place great again

• Update-in-place design to provide stellar read performance
  • Our angle: Leverage workload patterns to be competitive at writes
  • Key results: Up to 10.95x and 7.52x over RocksDB, LeanStore on YCSB
The Motivation

Key-value stores? Skew?
Persistent, concurrent KVSs abound

Configurations

User preferences

Profile metadata
Not all keys are created equal

• Updates >> Inserts
• Varying hotness
• Hotness independent of key
• Frequently-updated and frequently-read keys not necessarily the same.
• How to handle such a workload efficiently?
The Orientation

From log-structured databases to storage interfaces
LSM-trees efficiently absorb writes

- Usual solution: Log-Structured Merge (LSM) tree.
- Basic principles:
  - Buffer writes.
  - Write to disk when full.
  - Periodically “compact” logarithmically.
  - Read from memtables, or from cache; fresher versions are in lower-numbered levels.
Some RUM-induced thoughts

- RUM conjecture [Athanassoulis et al. 2016] – access methods trade off:
  - Read performance
  - Update performance
  - Memory performance
- Efficient updates: dump into memtable and flush periodically.
- LSM trees: slow reads unless hot write keys match hot read keys.
- High memory use: same key can be in many places.
Modern storage provides new trade-offs

- HDDs: sequential access unlocks performance.
- Flushing memtable achieves high throughput.
- NVMe SSDs: random writes are also performant
- Sequential *reads* still better than random.
- Speculative pre-fetching.
- Closer for large blocks.
- Can we improve read and memory performance?
The Innovation

How to make an update-in-place design workable
Key Idea A: Record Caching

Workload skew doesn’t care about layout

• LSMs use block (page) cache.
• One hot record in each page?
• **Key Idea A:** use instead a record cache.
  • Lower memory amplification.
  • Higher I/O amplification.
  • Balance in our favor.
• In-memory index with one key per page.
• When page is full, allocate overflow.
• When overflow page is full, reorganize.
Key Idea B: Page Grouping

Small pages, large pages – why not both?

• Point requests? Make pages small!
• Scans? Make pages large!
• **Key Idea B**: Page grouping.
  • Co-locate pages, forming *segments*.
  • For scans, read the entire segment.
• Use linear models to shrink index.
• Synchronization contention point.
• Only index one key per *segment*.
• Use model to navigate within segment.
Key Idea C: Insert Forecasting

Half-full pages are usually half-empty

• One page for a record – if full, must reorganize.
• How much space to leave?
  • Too much: Bad I/O amplification.
  • Too little: Must reorganize often.
• **Key Idea C**: Insert Forecasting.
  • Predict inserts using recent sample.
  • On reorganization, leave empty space based on estimate.
The Evaluation

So, how well does this work?
Experimental setup

- **Hardware:**
  - 20-core 2.10 GHz Intel Xeon Gold 6230 CPU, 128 GiB of memory
  - 1 TB Intel DC P4510 NVMe SSD

- **Workload:** Yahoo! Cloud Serving Benchmark suite (YCSB)
  - Amazon reviews dataset (33 million keys), 33% fits in memory
  - Zipfian and uniform requests

- **Baselines:**
  - RocksDB (LSM)
  - LeanStore (Update-in-place)
We look at throughput under parallelism

![Graph showing throughput comparison between TreeLine, RocksDB, and LeanStore across different thread counts.](image-url)
TreeLine shines across the board

- Only keep the high-level trends from here.
- We will dive deeper in the following slides.
Physical I/O drives the win over RocksDB

- **Against RocksDB**: 1.62x for 64 B records, 2.99x for 1024 B records.
- **Case study**: Workload A (50% reads, 50% updates)
  - TreeLine writes 3.09 GiB physical data, RocksDB writes 4.27 GiB. Memtables cannot consolidate updates.
  - TreeLine reads 12 GiB physical data, RocksDB reads 27 GiB. RocksDB needs background compactions.
  - Same trend present for the rest of the point workloads.
Caching drives the win over LeanStore

- Same plots as last slide.
- **Against LeanStore:** 2.81x for 64 B records, 1.53x for 1024 B records.
- LeanStore caches pages, achieving worse cache utilization that again drives physical I/O up.
  - Notice how it outperforms RocksDB for 1024 B records, when the record size approaches the 4 KiB page size.
Update-in-place helps scans

- TreeLine only stores one version of each record on disk.
- No need to merge results from different levels.
- Scan throughput (requests/second) outshines competition.
- Less data to read than RocksDB.
- Better throughput than LeanStore due to page grouping.

**TreeLine: An Update-In-Place Key-Value Store for Modern Storage**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeLine 64 B</td>
<td>13.4 GiB</td>
<td>550 MiB/s</td>
<td>100 kreq/s</td>
</tr>
<tr>
<td>RocksDB 64 B</td>
<td>31.1 GiB</td>
<td>797 MiB/s</td>
<td>42 kreq/s</td>
</tr>
<tr>
<td>LeanStore 64 B</td>
<td>17.0 GiB</td>
<td>581 MiB/s</td>
<td>56 kreq/s</td>
</tr>
<tr>
<td>TreeLine 1024 B</td>
<td>75.9 GiB</td>
<td>1079 MiB/s</td>
<td>12 kreq/s</td>
</tr>
<tr>
<td>RocksDB 1024 B</td>
<td>147 GiB</td>
<td>958 MiB/s</td>
<td>5.4 kreq/s</td>
</tr>
<tr>
<td>LeanStore 1024 B</td>
<td>76.4 GiB</td>
<td>155 MiB/s</td>
<td>1.7 kreq/s</td>
</tr>
</tbody>
</table>
Caching & grouping are complementary

- For point workloads (A-D, F), record caching provides most of the benefit: keeping the hot working set in memory.
- For scan-heavy workload E, page grouping doubles the throughput.
- Grouping does not hurt point workload performance.
Forecasting inserts gives an extra boost

- Dataset: NYC taxi pickups (key is inlined location)
- 64B case: Closes more than half of the gap to perfect.
- 64B case: Reorganizations reduced by 63% on average (not plotted).
- 512B case: Not enough granularity on 4KiB page.
- Overall, improves base by 1.22x and reduces reorganizations by 41% on average.
Coarse-grained epoch tuning is enough

• Epoch length affects throughput.
• Small epochs: Can capture trends at small timescales, but lots of background work.
• Long epochs: Can get a more representative sample, but might “average out” some trends.
• Still, even an epoch length 1/10 or 2x of what we used in the paper would be an improvement over no forecasting.
Key takeaways

- NVMe SSDs: Parallel random writes ≈ sequential write performance
  - Opportunity to revisit KVS design

- TreeLine: Update-in-place with three key ideas
  - Record caching: Efficient memory use for skewed read/write workloads
  - Page grouping: Large physical reads for scans, single-page reads for point lookups
  - Insert forecasting: Proactively "leave space" for inserts

- Key results (YCSB throughput)
  - Point workloads: 2.20x and 2.07x over RocksDB, LeanStore on average
  - Uniform scan-heavy (16 threads): 2.50x and 2.80x over RocksDB, LeanStore
  - Up to 10.95x and 7.52x over RocksDB, LeanStore overall

Paper: tinyurl.com/treeline-paper