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

The Motivation

Key-value stores? Skew? Modern SSDs?
KVSs abound, but not all keys are created equal

- Varying hotness
- Hotness independent of key
- Frequently-updated and frequently-read keys not necessarily the same.
- Updates >> Inserts
- How to handle such a workload efficiently?
LSM-tree designs optimize for writes

- **Common**: Log-Structured Merge (LSM) tree.
- **Basic principles**:
  - Buffer writes.
  - Write to disk when full.
  - Periodically “compact” logarithmically.
  - Read from memtables or cache; fresher versions are in lower-numbered levels.
- ✓ **Efficient writes**: dump new values into memtable and flush periodically.
- ✗ **Slow reads and high memory use**: multiple possible locations for each key.
Update-in-place designs optimize for reads

- **Update-in-place**: Classic B+ trees

- ✓ **Efficient reads**: one physical location per key.

- ✗ **Writes need random I/O**: much worse than sequential writes in HDDs.

- LSMs more widely used due to this random I/O trade-off.
The storage landscape has evolved!

- **NVMe SSDs**: Random write throughput $\approx$ sequential write throughput at high parallelism.
- **Sequential reads** still better than random reads.
- Speculative pre-fetching.
- Larger random reads comparatively better.
Can we bridge the two design extremes?

**This work:** Can we make update-in-place designs competitive against LSMs **on writes**, while **still excelling at reads**?
The Innovation

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

For skewed point requests, cache records

• Point reads and updates hit cache first.
• LSMs and classic B+ Trees use block (page) caches.
• One hot record in each page?

• Key Idea A: use instead a record cache.
  • Lower memory amplification.
  • Higher I/O amplification (need to write out pages)
  • Balance in our favor.
Key Idea B: Page Grouping

For scans, group pages into segments

• Larger random reads are faster.

• **Key Idea B**: Page grouping.
  • Co-locate pages, forming *segments*.
  • For scans, read the entire segment.
  • Navigate within segment using linear models.
Key Idea C: Insert Forecasting

For inserts, leave space intelligently

- One page for a record – what if full?
- 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.
  - Make limited use of overflow pages to reduce reorganization frequency.

![Diagram of TreeLine: An Update-In-Place Key-Value Store for Modern Storage](https://github.com/mitdbg/treeline)
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 uniformly distributed requests
- **Baselines:**
  - RocksDB (LSM)
  - LeanStore (Update-in-place)
- **Metrics:**
  - Request throughput
  - Physical I/O
TreeLine shines across the board

- **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
## Physical I/O and caching drive our wins

<table>
<thead>
<tr>
<th></th>
<th>...on point workloads</th>
<th>...on scan workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Against RocksDB...</td>
<td>Read much less from disk: no need to access multiple levels or compact them</td>
<td>Read much less from disk: physical read throughput is lower (random I/O) but less data to read.</td>
</tr>
<tr>
<td>Against LeanStore...</td>
<td>Better cache utilization: cache hot records instead of entire pages</td>
<td>Larger reads, better physical throughput: page grouping allows for larger physical reads in TreeLine.</td>
</tr>
</tbody>
</table>
Our key ideas are complementary

- Record caching and page grouping work in tandem:
  - For point workloads (A-D, F), record caching provides most of the benefit.
  - For scan-heavy workload E, page grouping doubles the throughput.

- Insert forecasting boosts throughput by reducing reorganizations
  - 64B case: Closes more than half of the gap to perfect.
  - 512B case: Not enough granularity on 4KiB page.
More details in the paper

- Implementation details
- Concurrency control
- Durability & recovery

- Additional experiments
- Page grouping effectiveness
- Insert forecasting epoch length

- Discussion
- Possible extensions
- Workload 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