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

**Geoffrey X. Yu*, Markos Markakis***,
Andreas Kipf*, Per-Åke Larson,
Umar Farooq Minhas, Tim Kraska

MIT CSAIL
DSAIL
Data Systems and AI Lab

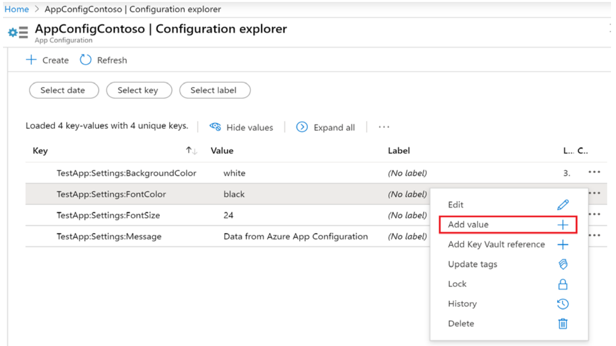
Photo by Richard Main on Unsplash

# The Motivation

Key-value stores? Skew? Modern SSDs?

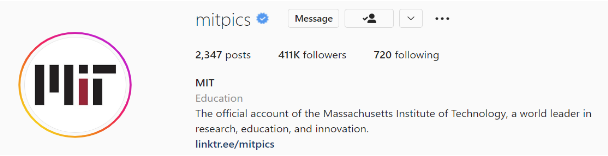
# KVSs abound, but not all keys are created equal

Configurations



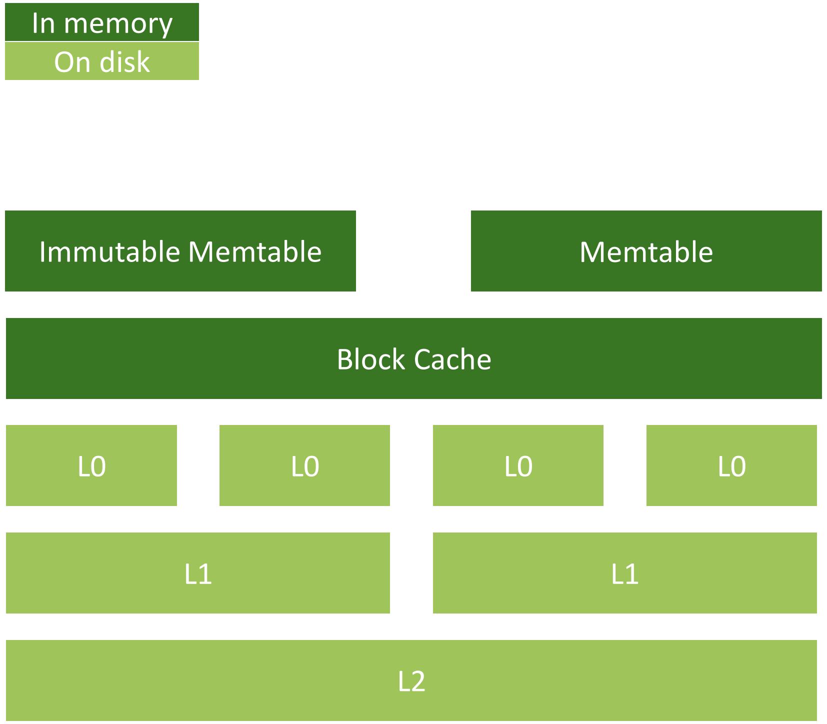

User preferences

Profile metadata




- 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?

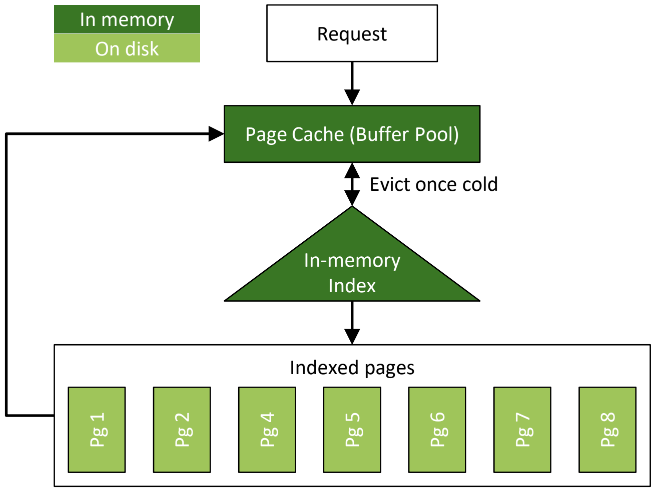
# LSM-tree designs optimize for writes



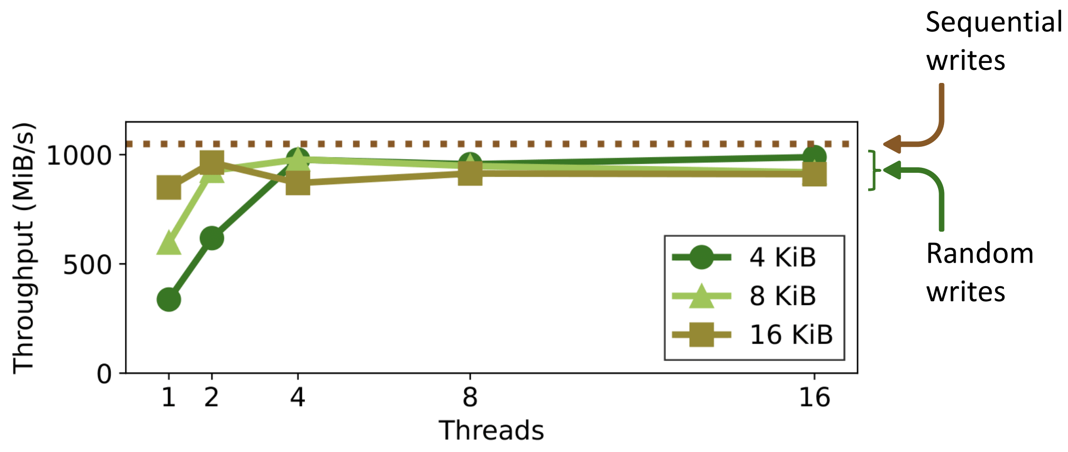

- **Common:** Log-Structured Merge (LSM) tree.
- **Basic principles:**
  - Buffer writes.
  - Write to disk when full.
  - Periodically "compact" logarithmically.
  - Read from memtables or cache; fresher versions are in lower-numbered levels.
- ✓ **Efficient writes**: dump new values into memtable and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

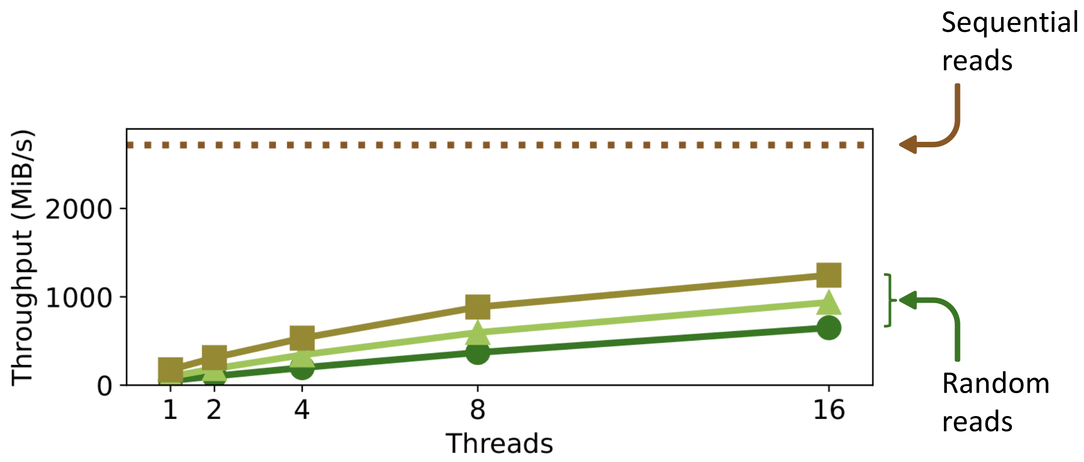
# Update-in-place designs optimize for reads

- **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.



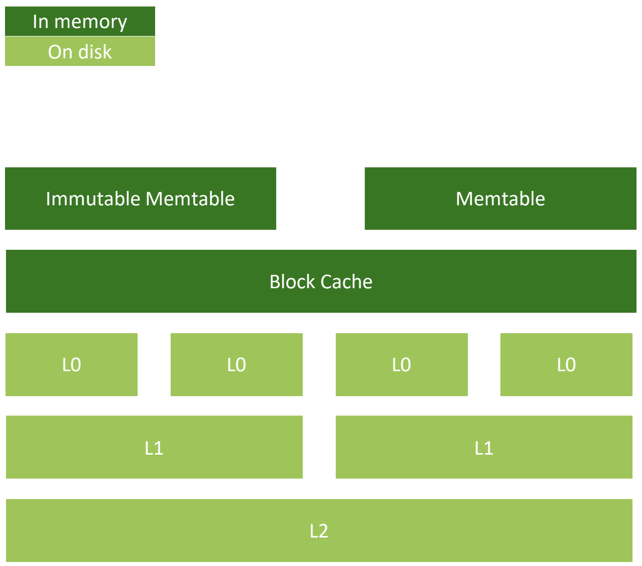
# The storage landscape has evolved!



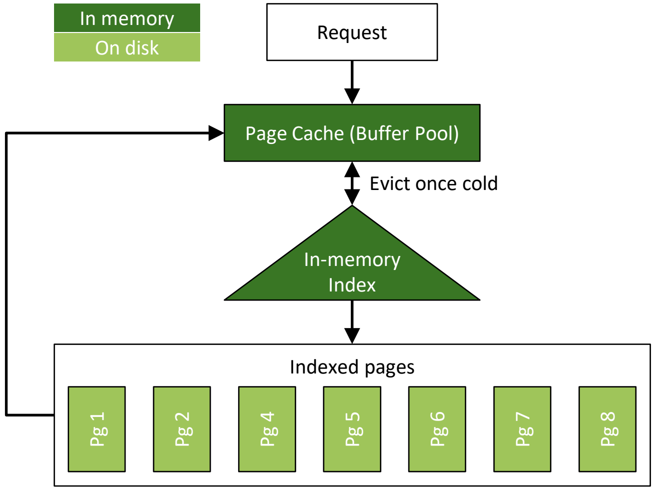




- **NVMe SSDs:** Random write throughput ≈ sequential write throughput *at high parallelism*
- Sequential *reads* still better than random reads.
  - Speculative pre-fetching.
  - Larger random reads comparatively better.

# 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

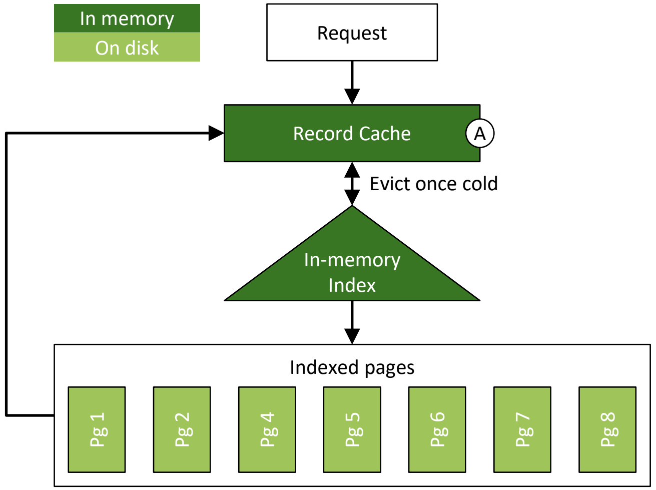
How to make an update-in-place design workable

## 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?

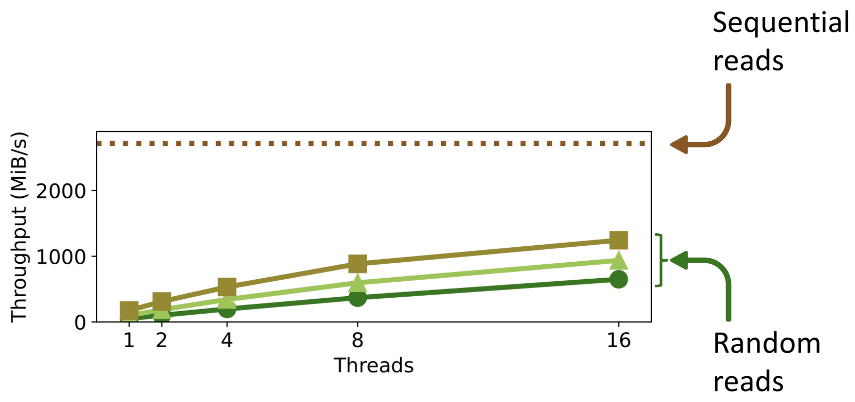
- **Key Idea A**: use instead a *record* cache.
  - Lower memory amplification.
  - Higher I/O amplification (need to write out pages)
  - Balance in our favor.

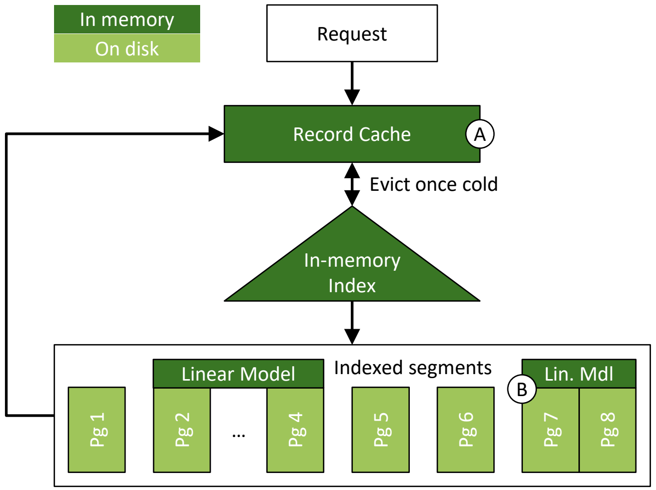


## Key Idea B: Page Grouping

# 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.

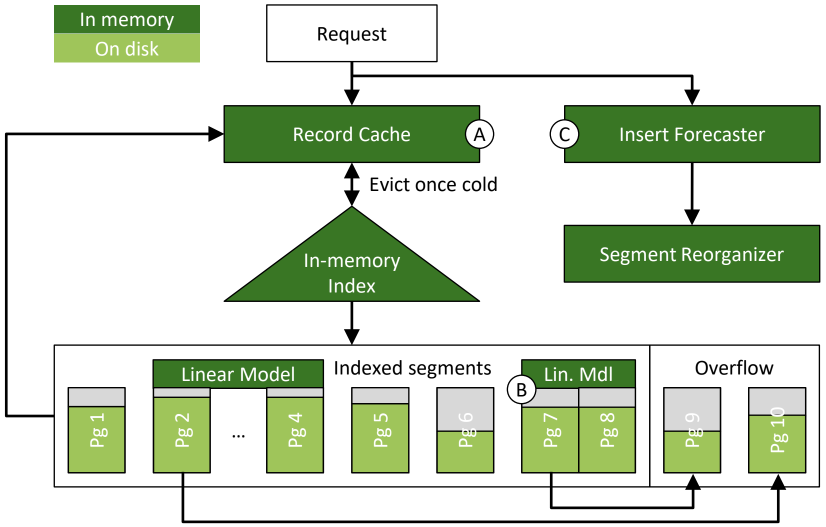


## Key Idea C: Insert Forecasting

# 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.



# The Evaluation

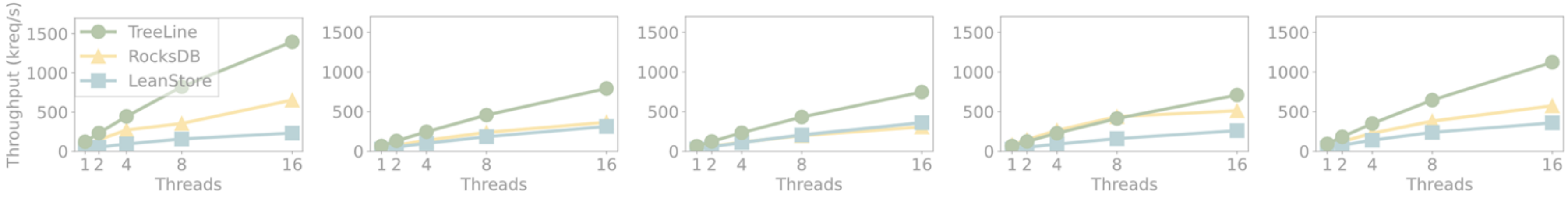
So, how well does this work?

# 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


RocksDB

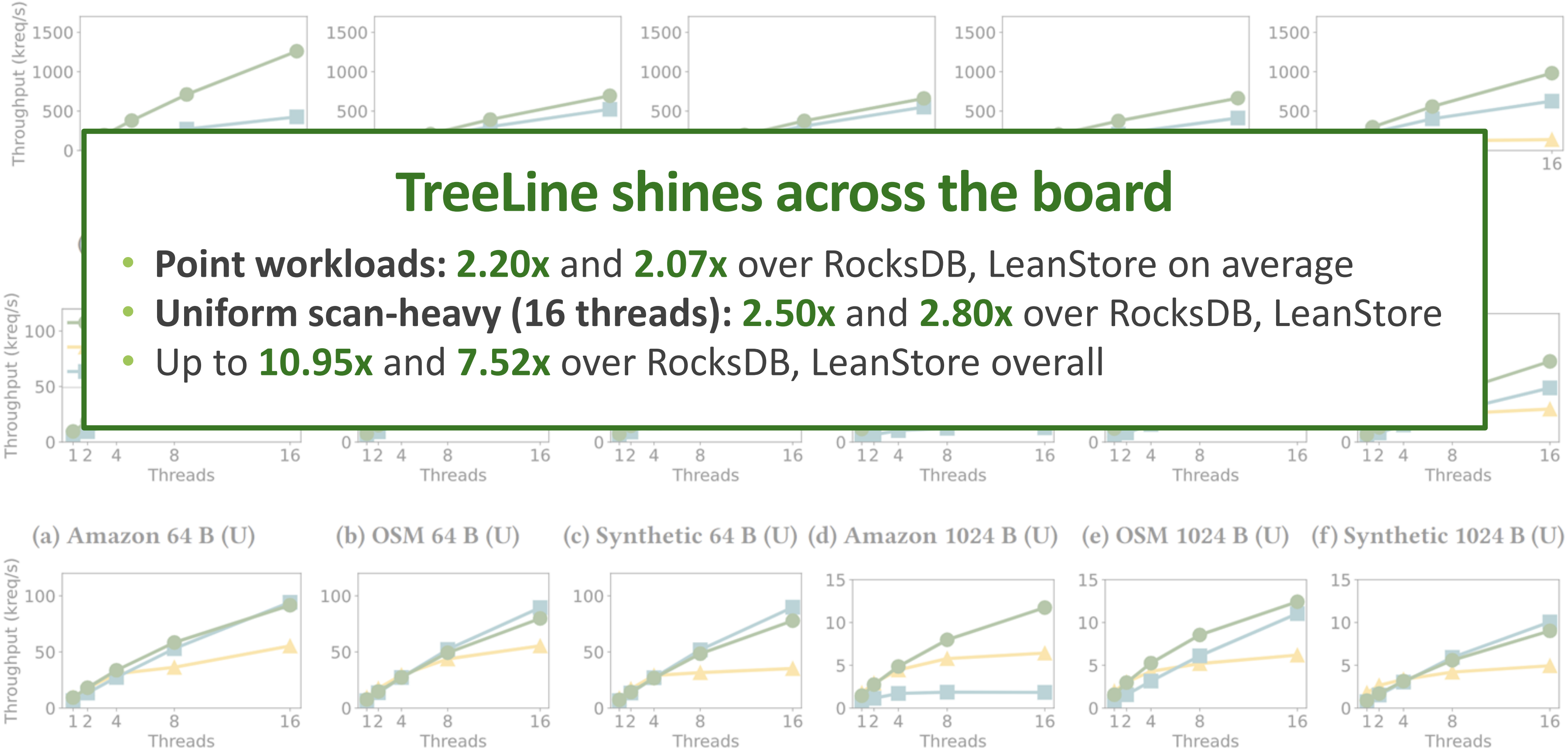
leanstore




(a) A (64 B) (b) B (64 B) (c) C (64 B) (d) D (64 B) (e) F (64 B)

## 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



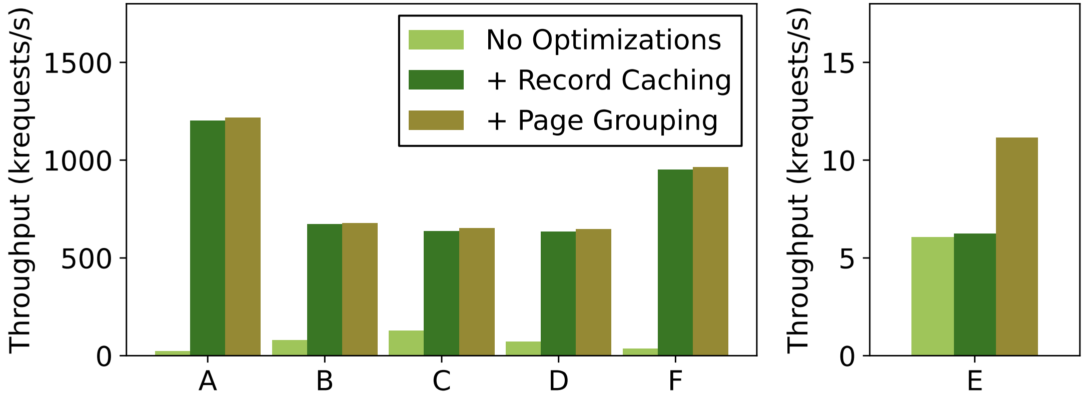

(a) Amazon 64 B (U) (b) OSM 64 B (U) (c) Synthetic 64 B (U) (d) Amazon 1024 B (U) (e) OSM 1024 B (U) (f) Synthetic 1024 B (U)

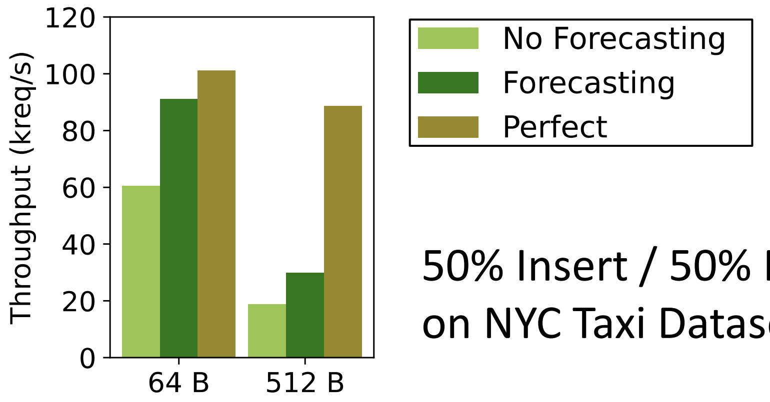
(g) Amazon 64 B (Z) (h) OSM 64 B (Z) (i) Synthetic 64 B (Z) (j) Amazon 1024 B (Z) (k) OSM 1024 B (Z) (l) Synthetic 1024 B (Z)

# Physical I/O and caching drive our wins

| | ...on point workloads | ...on scan workloads |
|---|---|---|
| **Against RocksDB...** | **Read much less from disk:** no need to access multiple levels or compact them | **Read much less from disk:** physical read throughput is lower (random I/O) but less data to read. |
| **Against LeanStore...** | **Better cache utilization:** cache hot records instead of entire pages | **Larger reads, better physical throughput:** page grouping allows for larger physical reads in TreeLine. |

# Our key ideas are complementary




50% Insert / 50% Read on NYC Taxi Dataset

- **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

Artifacts Available acm V1.1

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

Geoffrey X. Yu*
Massachusetts Institute of Technology
geoffxy@mit.edu

Markos Markakis*
Massachusetts Institute of Technology
markakis@mit.edu

Andreas Kipf *
Massachusetts Institute of Technology
kipf@mit.edu

Per-Åke Larson
University of Waterloo
gpalarson@outlook.com

Umar Farooq Minhas†
Apple
ufminhas@apple.com

Tim Kraska
Massachusetts Institute of Technology
kraska@mit.edu

**ABSTRACT**

Many modern key-value stores, such as RocksDB, rely on log-structured merge trees (LSMs). Originally designed for spinning disks, LSMs optimize for write performance by only making sequential writes. But this optimization comes at the cost of reads: LSMs must rely on expensive compaction jobs and Bloom filters—all to maintain reasonable read performance. For NVMe SSDs, we argue that trading off read performance for write performance is no longer always needed. With enough parallelism, NVMe SSDs have comparable random and sequential access performance. This change makes update-in-place designs, which traditionally provide excellent read performance, a viable alternative to LSMs.

In this paper, we close the gap between log-structured and update-in-place designs on modern SSDs with the help of new components that take advantage of data and workload patterns. Specifically, we explore three key ideas: (A) *record caching* for efficient point operations, (B) *page grouping* for high-performance range scans, and (C) *insert forecasting* to reduce the reorganization costs of accommodating new records. We evaluate these ideas by implementing them in a prototype update-in-place key-value store called *TreeLine*. On YCSB, we find that TreeLine outperforms RocksDB and LeanStore by 2.20× and 2.07× respectively on average across the point workloads, and by up to 10.95× and 7.52× overall.

**PVLDB Reference Format:**
Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. PVLDB, 16(1): 99 - 112, 2022.
doi:10.14778/3561261.3561270

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/mitdbg/treeline.

**1 INTRODUCTION**

Modern persistent key-value stores, such as RocksDB [48] and LevelDB [27], are typically built using log-structured merge trees (LSMs) [53]. The key idea behind LSMs is *buffered log structuring*. Writes are first buffered in memory, and then eventually flushed to immutable files on disk. These files are then periodically compacted (i.e., merged) in the background to remove overwritten and deleted records. LSMs are popular because they provide stellar write performance. They ensure that all disk writes are sequential, which exploits the high sequential write bandwidth of traditional disks.

Yet despite these benefits, LSMs are not a silver bullet in key-value store design. While their design makes writes efficient, it comes at the cost of reads, since records can be present in multiple locations on disk. This is why systems like RocksDB and LevelDB employ block caches, Bloom filters [7, 22], and various compaction strategies [9, 19, 50]—complex and hard-to-tune [47] techniques all aimed at reducing the I/O overhead of reads. For traditional disks (e.g., HDDs and SATA SSDs), this write versus read performance trade-off has been the preferred choice. Random I/O on traditional disks is prohibitively expensive, and so any design that minimizes the amount of random I/O outshines the competition. But is this trade-off still the right one for modern storage devices?

We make the observation that modern NVMe SSDs no longer suffer the same significant random write drawback as traditional disks [29]. With enough request parallelism, NVMe SSDs can achieve their peak sequential write throughput through random writes [29, 39, 56]. This naturally leads us to a research question: how should a persistent key-value store's design change for NVMe SSDs where random writes are comparable to sequential writes in performance?

Our hypothesis is that an *update-in-place* design is the answer for larger-than-memory workloads that are (i) read-heavy, or (ii) skewed write-heavy. Update-in-place designs, such as a classical disk-based B+ tree [16, 51, 52], can offer excellent read performance because each record is stored in a single location on disk—requiring only one I/O to read, if inner nodes are cached in memory. High read performance is desirable because read-heavy workloads such as caching [8, 45] or analytics [4, 11, 40] are common in practice [10].

While disk-based B+ trees do have these read benefits, they are also known to suffer from their own challenges. First, updating a single record on a page requires reading and writing the entire page, which leads to write amplification. Second, scans can lead to random reads because logically consecutive leaf pages are not necessarily stored sequentially on disk; on NVMe SSDs, we observe that random reads still underperform sequential reads. Third, inserts also cause write amplification because of the need to "make space" in the on-disk structure to hold the new records.

Thus, in order to validate our hypothesis, we need to develop a new design for NVMe SSDs that has the read benefits of a classical update-in-place design while also mitigating its traditional write

*The first three authors contributed equally to this paper.
†Work done while at Microsoft Research.



Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561270

# 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



**Code:** github.com/mitdbg/treeline
**Paper:** tinyurl.com/treeline-paper