leanstore: A High-Performance Storage Engine for NVMe SSDs and Multi-Core CPUs

Viktor Leis

Technische Universität München

Disk-Based Database Systems [SIGMOD'08]

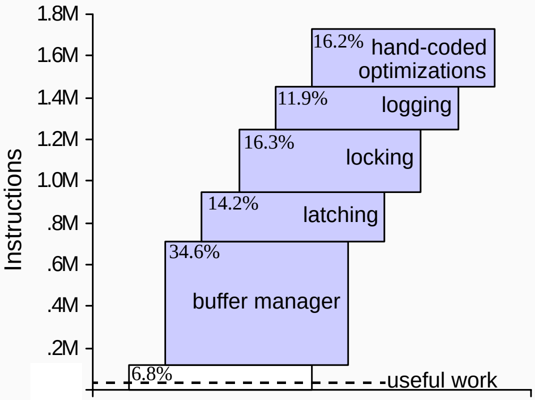
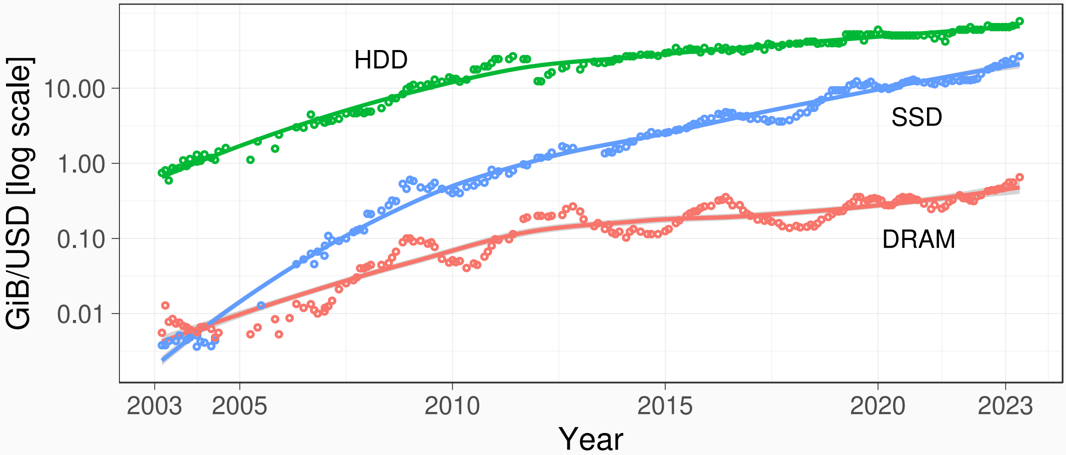


Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

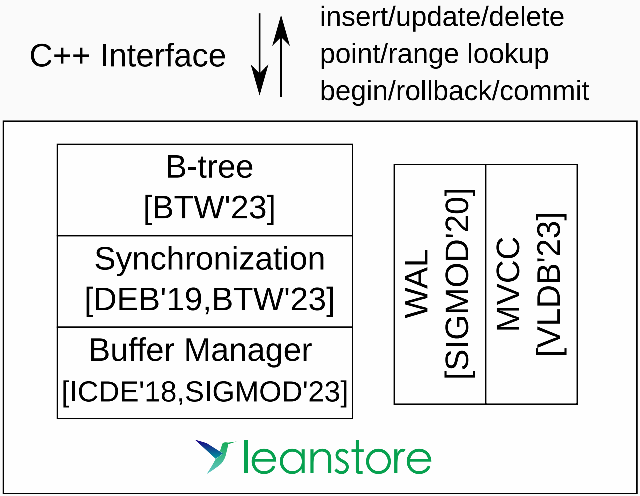
- "no single high pole in the tent"
- disk-based systems are hopeless
- only in-memory DBMS can be fast

The Problem With In-Memory Systems [Haas et al. CIDR'20]



DRAM stagnation + flash scalability = need for flash-optimized DBMS

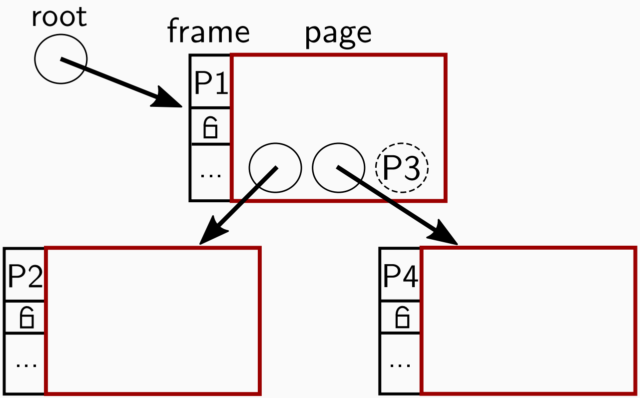
LeanStore Overview



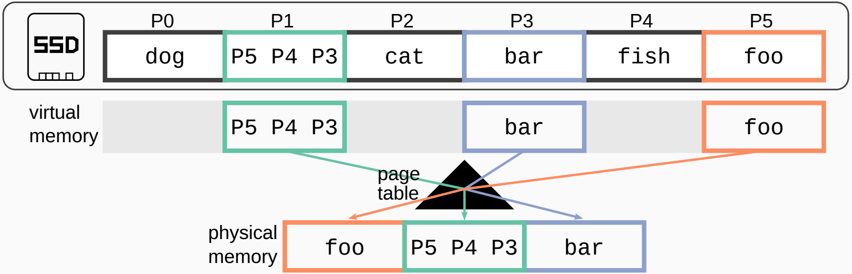
optimized for

- multi-cores CPUs
- in-memory performance
- out-of-memory NVMe performance

- page-based storage (4 KB) + pointer swizzling
- very low in-memory overhead for cached pages



Virtual Memory Assisted Buffer Management [Leis et al. SIGMOD'23]



DBMS controls everything:

- allocation: `mmap(NULL, ssdSize, ..., MAP_ANONYMOUS ...)`
- faulting: `pread(fd, virtMem + offset, pageSize, offset)`
- eviction: `madvise(virtMem + offset, pageSize, MADV_DONTNEED)`

(optional exmap Linux kernel module makes this fast and scalable)

Page Replacement Algorithm

- original algorithm: 10% of all cached pages are unswizzled and in a FIFO list
- similar to Second Chance

+ fast
+ scalable on multi-core CPUs
– replacement effectiveness
– not write aware

Write-Aware Timestamp Tracking

1. track per-page access timestamps:

i	1	2	3
t_i	15	8	0

$\Rightarrow$

i	1	2	3	4
t_i	42	15	8	0

2. compute sub-frequencies $SF_i(t_{now}) := \frac{i}{t_{now} - t_i}$

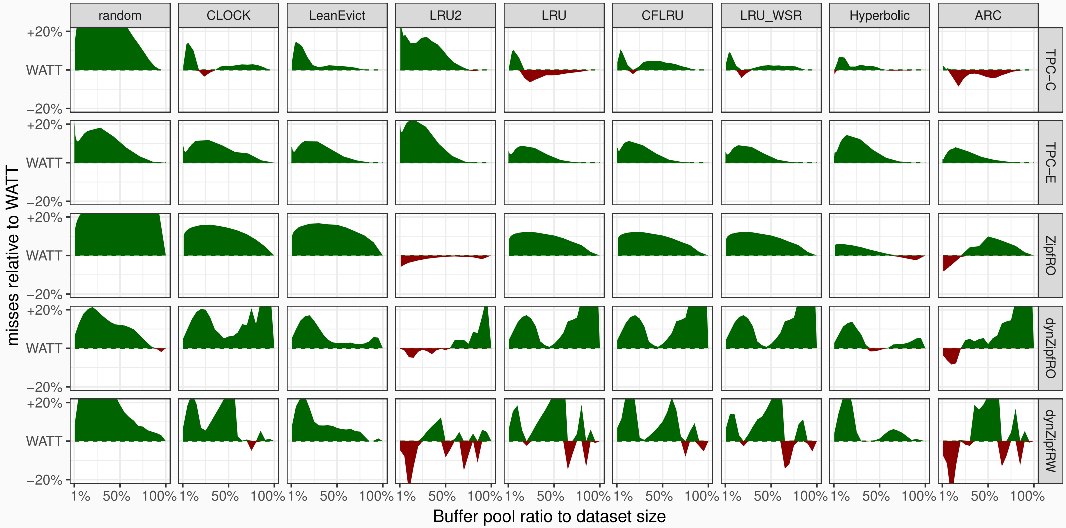
   e.g., at $t_{now} = 50$:

i	1	2	3	4
t_i	42	15	8	0
SF_i	$1/8 \approx 0.13$	$2/35 \approx 0.06$	$3/42 \approx 0.07$	$4/50 \approx 0.08$

3. compute page value by aggregating sub-frequencies:

   $PV^*_{access}(t_{now}) := \max_i SF_i(t_{now})$

Replacement Effectiveness (Simulation) [Vöhringer et al. VLDB'23]

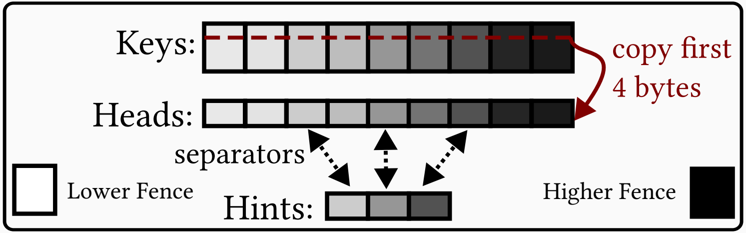


Making WATT Practical

[Vöhringer et al. VLDB'23]

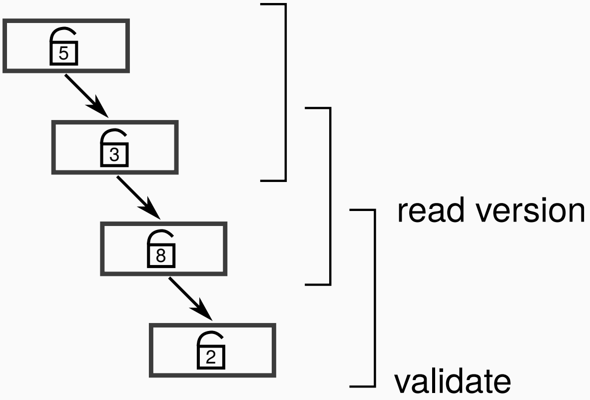
- write awareness: $PV(t_{now}) := PV^*_{access}(t_{now}) + write\_weight \cdot PV^*_{write}(t_{now})$
- scalability: increment global time only every k evictions
- space: limit tracking to 8+4 timestamps per in-memory page
- speed: sample random pages, computes *PV* for each and evict worst 10%
- speed: compute *PV* using SIMD (330 cycles vs. 130 cycles)
- speed: hide cache misses with software prefetching (130 cycles vs. 100 cycles)

- B^{+}-tree with variable-size keys/values using slotted page layout
- optimizations:
  - prefix: extract the common key prefix (Bayer and Unterauer, 1977)
  - heads: 4-byte key in slot ("poor man normalized keys", Graefe and Larson, 2001)
  - hints: store 16 heads redundantly



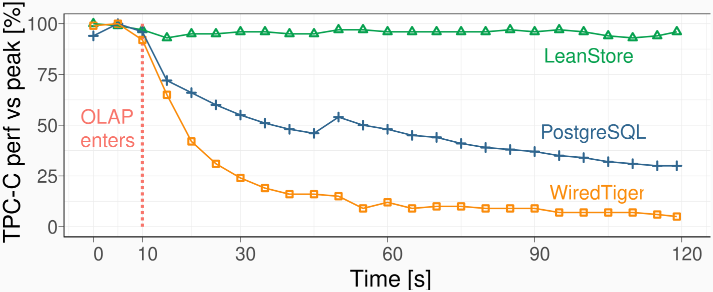
Synchronization Framework [DEB'19, BTW'23, SiMoD'23]

- each page has `atomic<uint64_t>` and `pthread_rwlock`
- three page access modes: optimistic, shared, exclusive
- enables Optimistic Lock Coupling
- additional memory reclamation mechanism (e.g., epochs, hazard pointers) not needed
- fast, scalable, easy-to-use



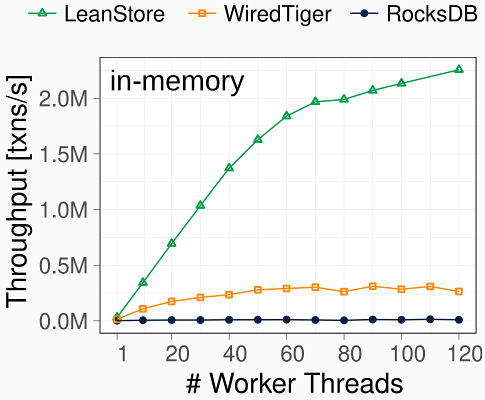
Scalable and Robust Snapshot Isolation [Alhomassi and Leis VLDB'23]

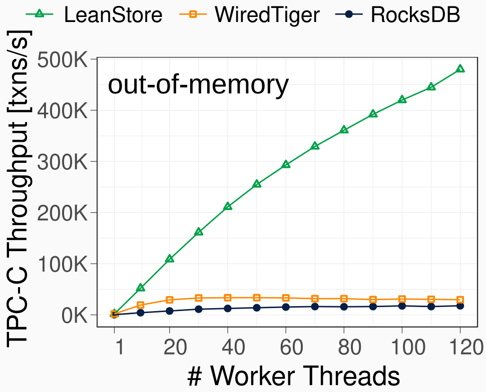
- snapshot isolation through multi-version concurrency control
- scalable, arbitrarily-large transactions, robust:
  - Ordered Snapshot and Instant Commit (OSIC) protocol
  - Graveyard Index: move logically-deleted tuples from index to separate structure
  - Adaptive Version Storage: Delta Index (default) and FatTuple (hot tuples only)
  - Garbage Collection: purge Delta and Graveyard Index using watermarks, prune long chains during processing, covert FatTuple to Delta Index on eviction



TPC-C Evaluation

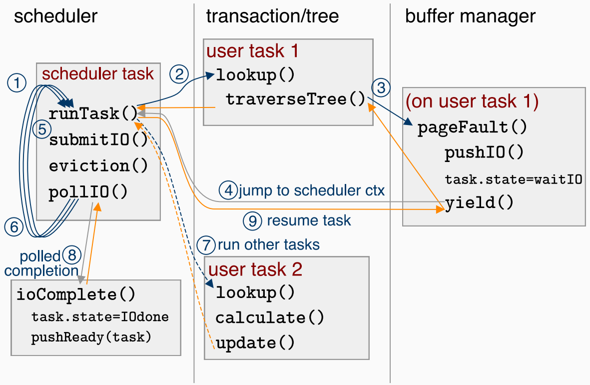
[Alhomassi and Leis VLDB'23]



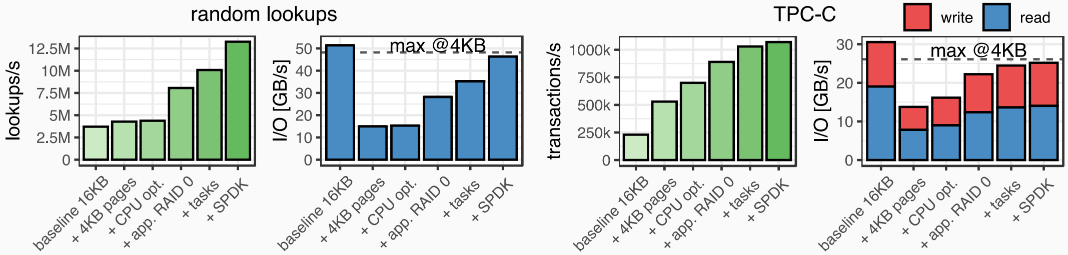


I/O Management [Haas and Leis VLDB'23]

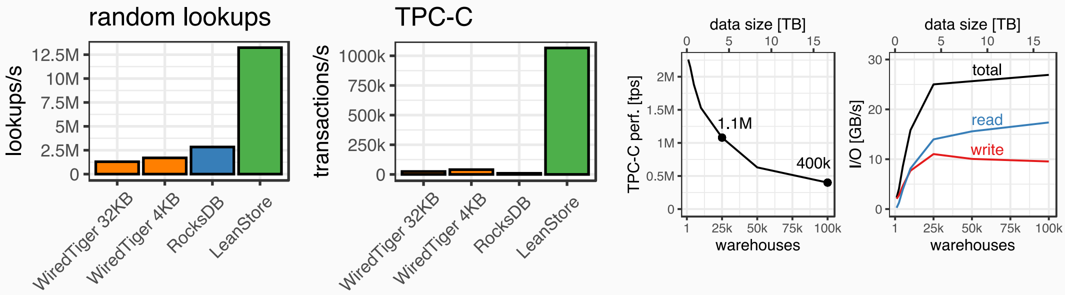
- worker threads = # hardware threads
- user-space scheduling, asynchronous I/O with libaio, uring, or SPDK
- Boost fcontext to switch between: user tasks, submission, polling, eviction
- highly-optimized I/O path: no global latches, no dynamic memory allocations



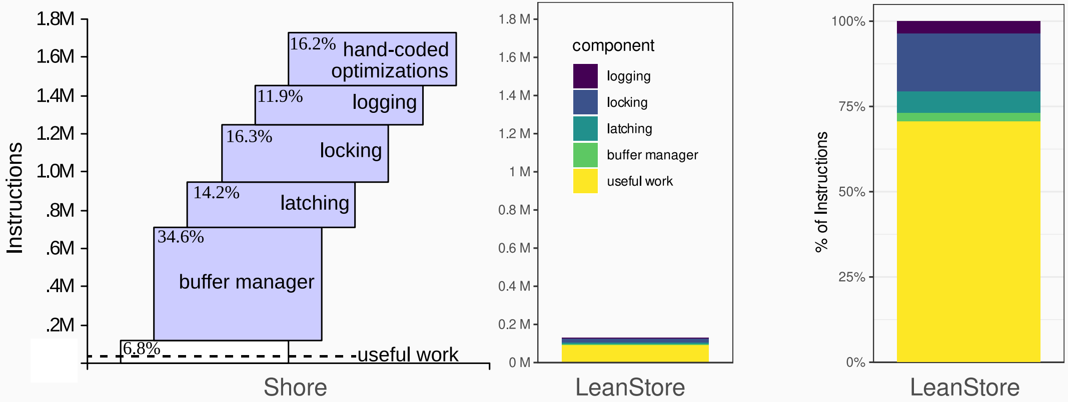
Out-Of-Memory Performance (400 GB buffer pool) [Haas and Leis VLDB'23]



Out-Of-Memory Performance (16 GB buffer pool, 160 GB data) [Haas and Leis VLDB'23]



Conclusions



outlook:

- Switch to vmcache, integration work, out-of-place writes
- Move to cloud, Unikernel co-design

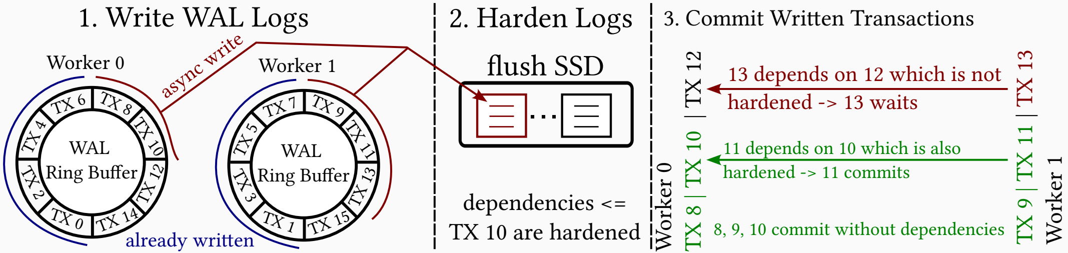
Commit Protocols

[VLDB'23]

	trad. DBMS*	in-mem. DBMS**	OSIC (LeanStore)
post-commit	$\Theta(1)$	$\Theta(\texttt{write set})$	$\Theta(1)$
snapshotting	$\Theta(T)$	$\Theta(1)$	$\Omega(1), O(T \log T)$
visibility check	$\Omega(1), O(T)$	$\Theta(1)$	$\Theta(1)$
memory usage	$\Theta(T^2)$	$\Theta(T)$	$\Theta(T^2)$

(*) PostgreSQL/InnoDB/WiredTiger (**) Hekaton/HANA/Hyper

T = #Threads or #Concurrent Transactions



1. Write WAL Logs
Worker 0
async write
Worker 1
TX 6
TX 8
TX 4
TX 10
TX 2
TX 12
TX 0
TX 14
WAL
Ring Buffer
TX 7
TX 9
TX 5
TX 11
TX 3
TX 13
TX 1
TX 15
WAL
Ring Buffer
already written
2. Harden Logs
flush SSD
dependencies <=
TX 10 are hardened
3. Commit Written Transactions
TX 12
TX 8 | TX 10
Worker 0
13 depends on 12 which is not
hardened -> 13 waits
11 depends on 10 which is also
hardened -> 11 commits
8, 9, 10 commit without dependencies
TX 13
TX 9 | TX 11
Worker 1