LeanStore: In-Memory Data Management Beyond Main Memory

Leis et. Al.

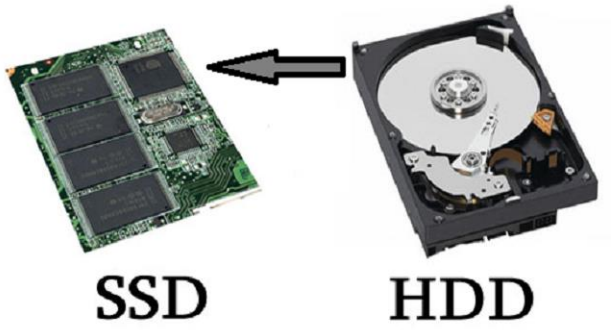
Presented by: Manoj Sharma

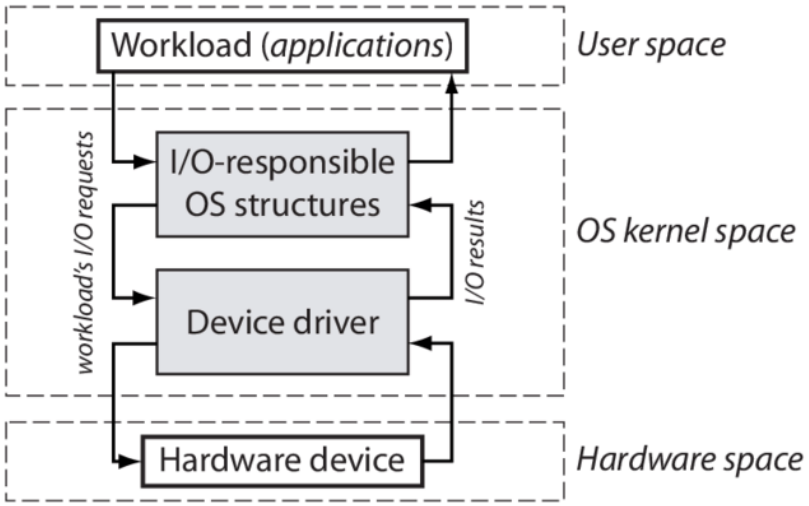
Basis

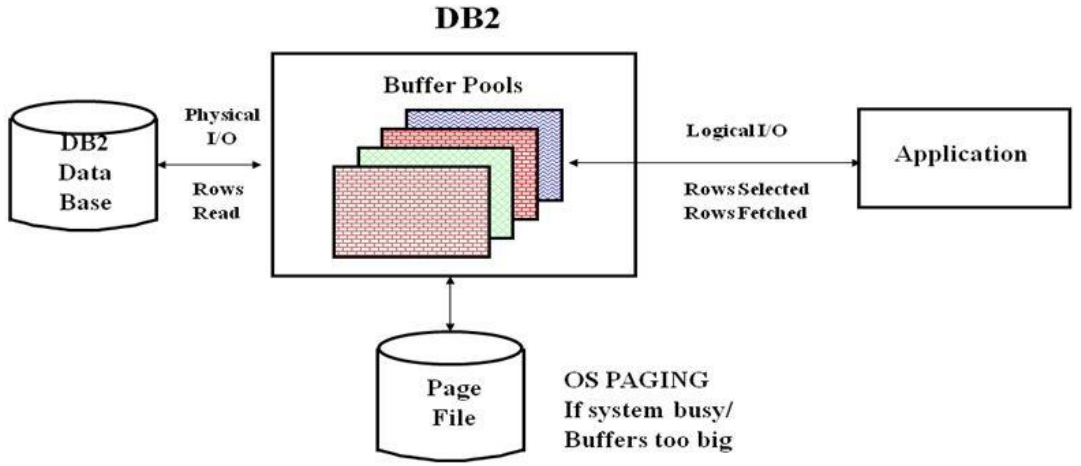
- Demand for low latency, higher processing speed for queries.
- Need for workloads to have higher throughput.
- Bottlenecks
    - I/O Path
    - Concurrent Execution
    - Logging

Tackling I/O Path

- I/O path is tightly coupled with OS
- Buffering
- Evolving storage technologies like using SDD.



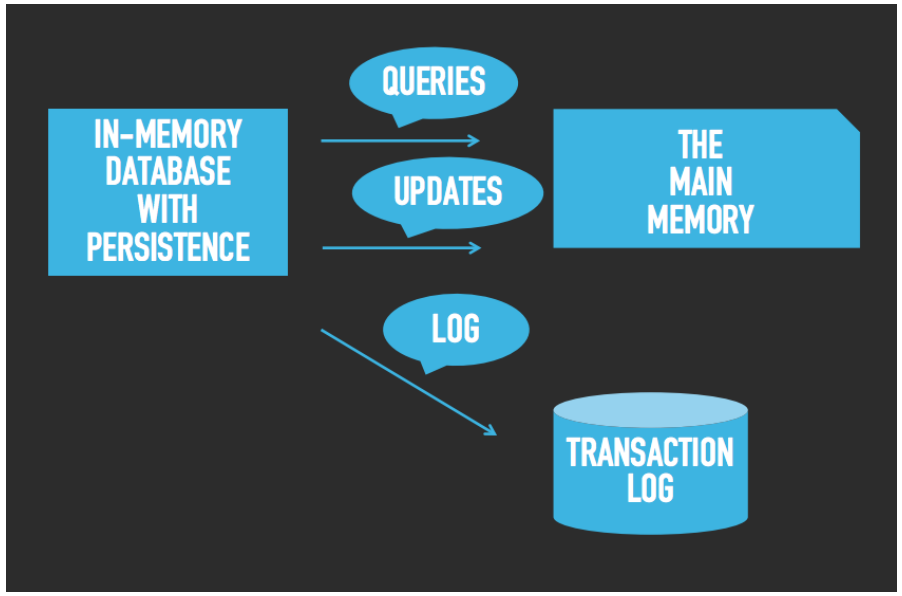




UNIVERSITY OF WATERLOO

Tackling I/O Path

- Cutting down I/O
- Is it InMemory Database ?



OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos
HP Labs
Palo Alto, CA
stavros@hp.com

Daniel J. Abadi
Yale University
New Haven, CT
dna@cs.yale.edu

Samuel Madden
Michael Stonebraker
Massachusetts Institute of Technology
Cambridge, MA
{madden, stonebraker}@csail.mit.edu

ABSTRACT

Online Transaction Processing (OLTP) databases include a suite of features — disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading — that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

Based on this observation, we look at some interesting variants of conventional database systems that one might build that exploit recent hardware trends, and speculate on their performance through a detailed instruction-level breakdown of the major components involved in a transaction processing database system (Shore) running a subset of TPC-C. Rather than simply profiling Shore, we progressively modified it so that after every feature removal or optimization, we had a (faster) working system that fully ran our workload. Overall, we identify overheads and optimizations that explain a total difference of about a factor of 20x in raw performance. We also show that there is no single "high pole in the tent" in modern (memory resident) database systems, but that substantial time is spent in logging, latching, locking, B-tree, and buffer management operations.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems — *transaction processing; concurrency.*

General Terms

Measurement, Performance, Experimentation.

Keywords

Online Transaction Processing, OLTP, main memory transaction processing, DBMS architecture.

1. INTRODUCTION

Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970's and 1980's, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to millions of dollars.

Today, the situation is quite different. First, modern processors are very fast, such that the computation time for many OLTP-style transactions is measured in microseconds. For a few thousand dollars, a system with gigabytes of main memory can be purchased. Furthermore, it is not uncommon for institutions to own networked clusters of many such workstations, with aggregate memory measured in hundreds of gigabytes — sufficient to keep many OLTP databases in RAM.

Second, the rise of the Internet, as well as the variety of data intensive applications in use in a number of domains, has led to a rising interest in database-like applications without the full suite of standard database features. Operating systems and networking conferences are now full of proposals for "database-like" storage systems with varying forms of consistency, reliability, concurrency, replication, and queryability [DG04, CDG+06, GBH+00, SMK+01].

This rising demand for database-like services, coupled with dramatic performance improvements and cost reduction in hardware, suggests a number of interesting alternative systems that one might build with a different set of features than those provided by standard OLTP engines.

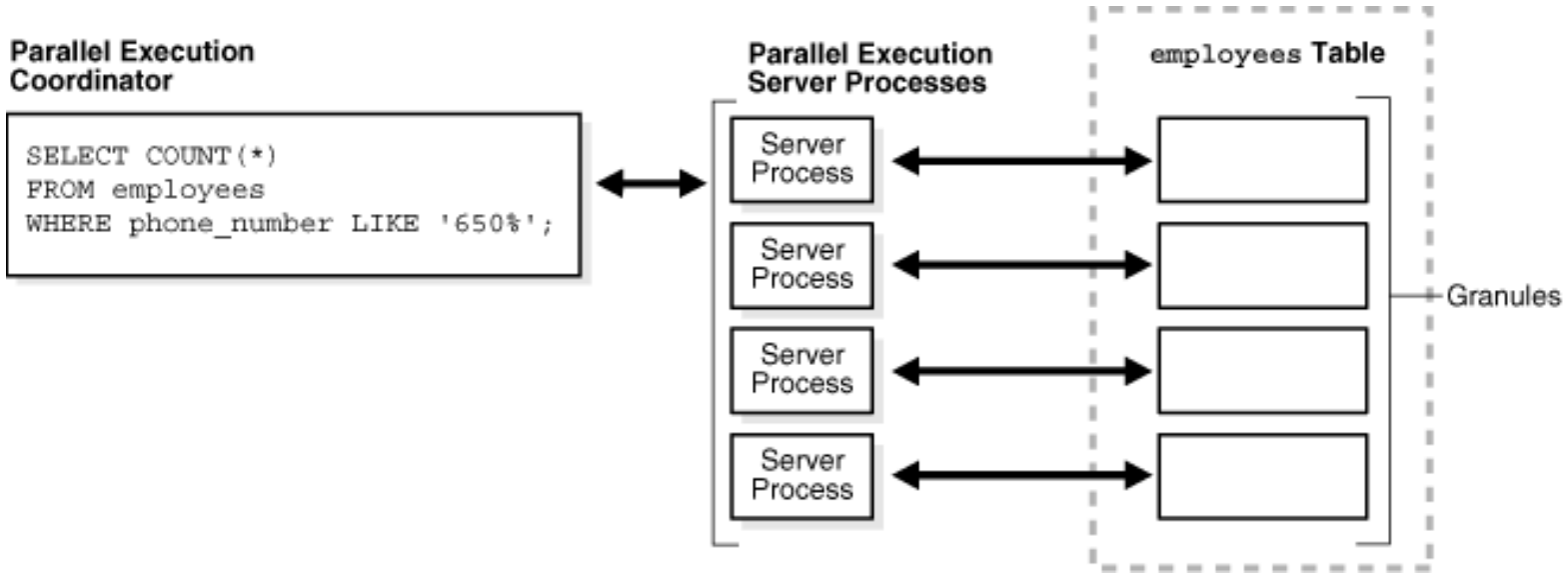
1.1 Alternative DBMS Architectures

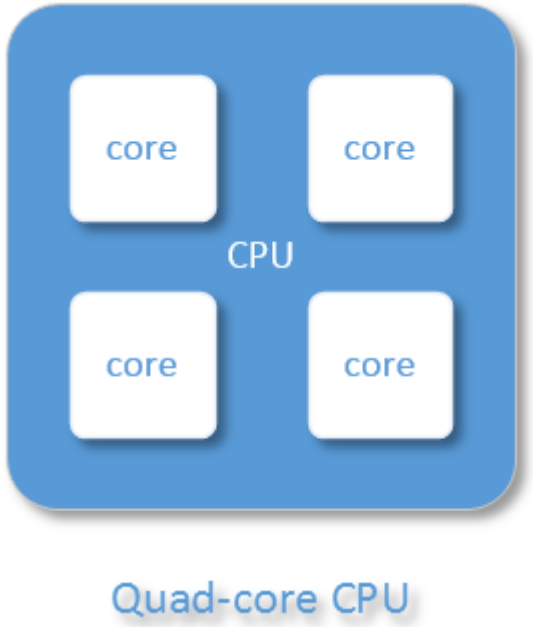
Obviously, optimizing OLTP systems for main memory is a good idea when a database fits in RAM. But a number of other database variants are possible; for example:

- **Logless databases**. A log-free database system might either not need recovery, or might perform recovery from other sites in a cluster (as was proposed in systems like Harp [LGG+91], Harbor [LM06], and C-Store [SAB+05]).
- **Single threaded databases**. Since multi-threading in OLTP databases was traditionally important for latency hiding in the

Tackling Concurrent Execution

- What is a query ? -> Similar operation done on multiple data
- Generalize queries to run in parallel
- Leverage hardware advancements.



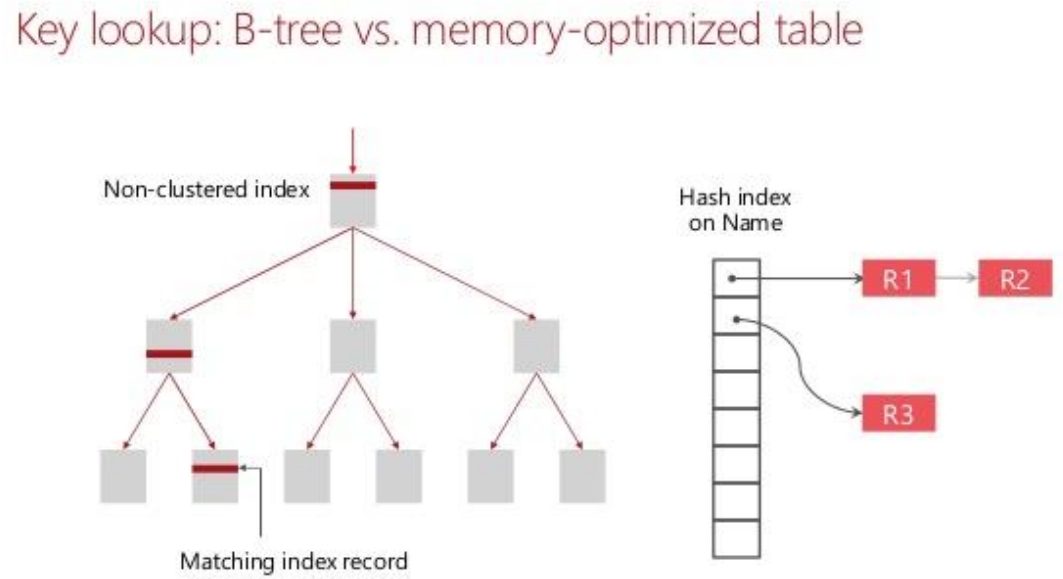


Quick Insights

- Everything in-memory -> higher throughput.
- Concurrent Execution -> low latency.
- Do we have the fastest database ?

In-memory databases Implementation – Just the beginning

- Underlying OS dependence - Memory management
- Use mmap ??
- Inmemory table layout.
- DataStructures for table mapping
- Hash Table vs B-tree
- Heavy index usage
- Concurrency -> increased latches

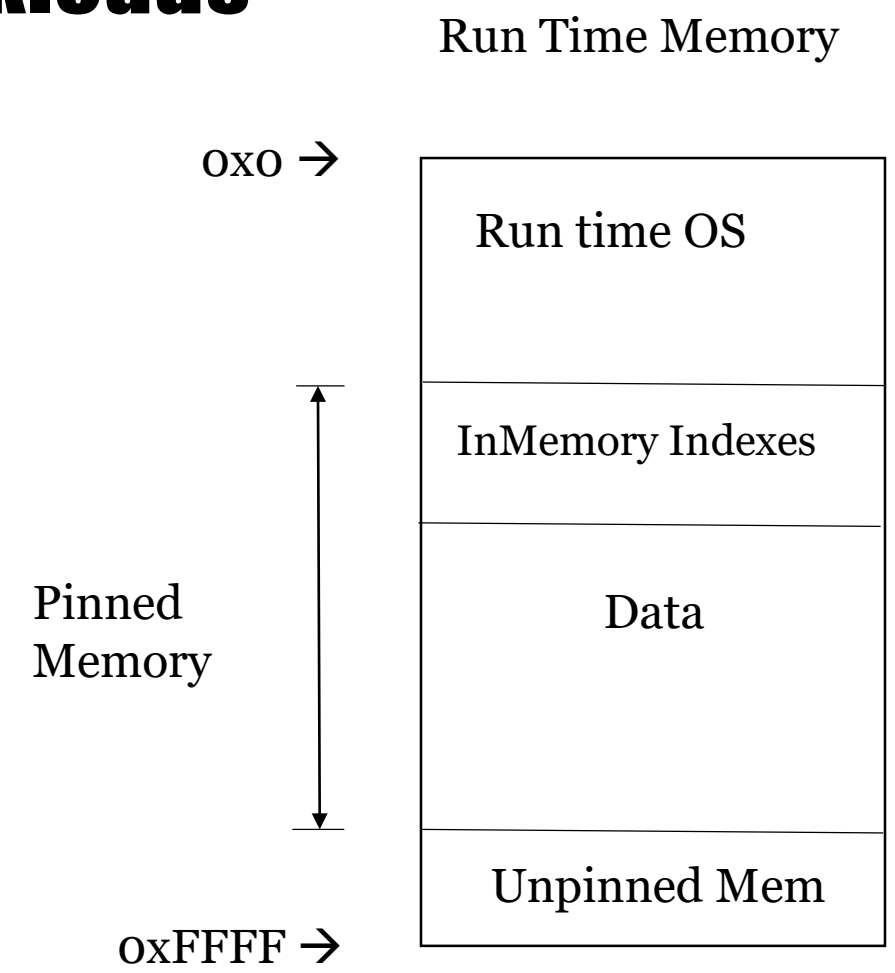


RoadMap

- In-Memory Database and Workloads
- Managing Data in-memory & cases
- Buffer Manager & Challenges
- LeanStore
- LeanStore Buffer Pool and Paging Overview
- Performance Evaluation
- Conclusion & Few Thoughts

InMemory Database and workloads

- Everything in-mem
- Scaleout ?
- Increase RAM size
- RAM is still costly.
- Limited by Address bus size.

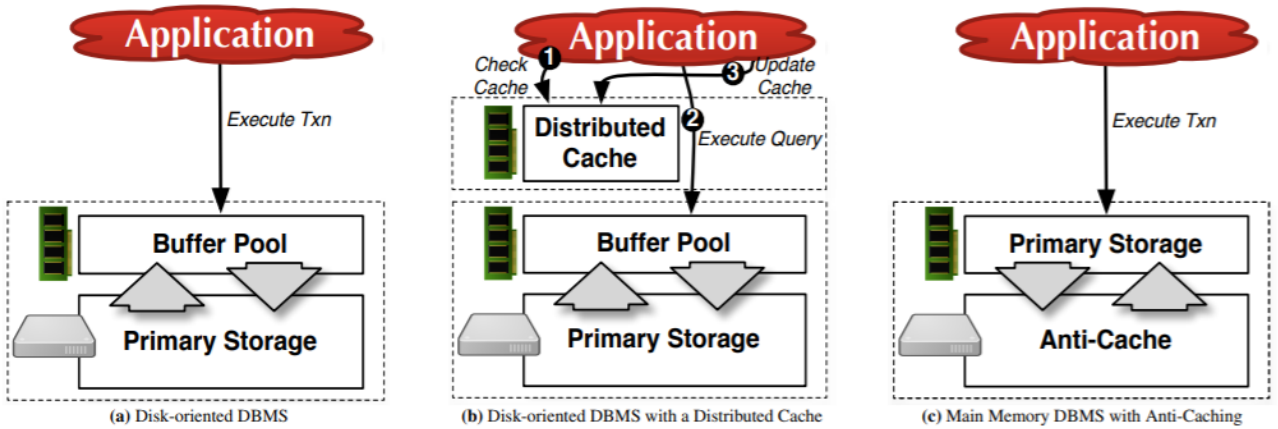


Managing data in-memory

- Gauge access patterns of data to have them in memory.
- Back to conventional memory management techniques – buffer management.
- No buffer management by H-Store, Hekaton, HANA etc.,

Managing data in-memory – cases

- AntiCaching
- Microsoft's Siberia



(a) Disk-oriented DBMS (b) Disk-oriented DBMS with a Distributed Cache (c) Main Memory DBMS with Anti-Caching

Figure 1: DBMS Architectures – In (a) and (b), the disk is the primary storage for the database and data is brought into main memory as it is needed. With the anti-caching model shown in (c), memory is the primary storage and cold data is evicted to disk.

Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database

Ahmed Eldawy*
University of Minnesota
eldawy@cs.umn.edu

Justin Levandoski
Microsoft Research
justin.levandoski@microsoft.com

Per-Åke Larson
Microsoft Research
palarson@microsoft.com

ABSTRACT

Main memories are becoming sufficiently large that most OLTP databases can be stored entirely in main memory, but this may not be the best solution. OLTP workloads typically exhibit skewed access patterns where some records are hot (frequently accessed) but many records are cold (infrequently or never accessed). It is still more economical to store the coldest records on secondary storage such as flash. This paper introduces Siberia, a framework for managing cold data in the Microsoft Hekaton main-memory database engine. We discuss how to migrate cold data to secondary storage while providing an interface to the user to manipulate both hot and cold data that hides the actual data location. We describe how queries of different isolation levels can read and modify data stored in both hot and cold stores without restriction while minimizing number of accesses to cold storage. We also show how records can be migrated between hot and cold stores while the DBMS is online and active. Experiments reveal that for cold data access rates appropriate for main-memory optimized databases, we incur an acceptable 7-14% throughput loss.

1. INTRODUCTION

Database systems have traditionally been designed under the assumption that data is disk resident and paged in and out of memory as needed. However, the drop in memory prices over the past 30 years is invalidating this assumption. Several database engines have emerged that store the entire database in main memory [3, 5, 7, 9, 11, 14, 19].

Microsoft has developed a memory-optimized database engine, code named Hekaton, targeted for OLTP workloads. The Hekaton engine is fully integrated into SQL Server and shipped in the 2014 release. It does not require a database be stored entirely in main memory; a user can declare only some tables to be in-memory tables managed by Hekaton. Hekaton tables can be queried and updated in the same way as regular tables. To speed up processing even further, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. Further details about the design of Hekaton can be found in [4], [11].

OLTP workloads often exhibit skewed access patterns where some records are "hot" and accessed frequently (the working set) while others are "cold" and accessed infrequently. Clearly, good performance depends on the hot records residing in memory. Cold records can be moved to cheaper external storage such as flash with little effect on overall system performance.

The initial version of Hekaton requires that a memory-optimized table fits *entirely* in main memory. However, even a frequently accessed table may exhibit access skew where only a small fraction of its rows are hot while many rows are cold. We are investigating techniques to automatically migrate cold rows to a "cold store" residing on external storage while the hot rows remain in the in-memory "hot store". The separation into two stores is only visible to the storage engine; the upper layers of the engine (and applications) are entirely unaware of where a row is stored.

The goal of our project, called Project Siberia, is to enable the Hekaton engine to automatically and transparently maintain cold data on cheaper secondary storage. We divide the problem of managing cold data into four subproblems.

- *Cold data classification*: efficiently and non-intrusively identify hot and cold data. We propose to do this by logging record accesses, possibly only a sample, and estimating accesses frequencies off line as described in more detail in [13]. One could also use a traditional caching approach such as LRU or LRU-2 but the overhead is high in both space and time. As reported in [13], experiments showed that maintaining a simple LRU chain added 25% overhead to the cost of lookups in an in-memory hash table and added 16 bytes to each record. This we deemed too high a price.
- *Cold data storage*: evaluation of cold storage device options and techniques for organizing data on cold storage.
- *Cold storage access reduction*: reducing unnecessary accesses to cold storage for both point and range lookups by maintaining compact and accurate in-memory access filters. We propose to achieve this by storing in memory compact summaries of the cold store content. We are investigating two techniques: a version of Bloom filters for point lookups and range filters, a new compact data structure that also supports range queries. More details can be found in [1, 17].
- *Cold data access and migration mechanisms*: mechanisms for efficiently migrating, reading, and updating data on cold storage that dovetail with Hekaton's optimistic multi-version concurrency control scheme [11].

In this paper, we focus on the fourth point, namely, how to migrate records to and from the cold store and how to access and update records in the cold store in a transactionally consistent manner. This paper is *not* concerned with exact indexing and storage mechanisms used; all we assume is that the cold store provides methods for inserting, deleting, and retrieving records. To allow for maximum flexibility in the choice of cold store implementations our only

* Work done while at Microsoft Research

Proceedings of the VLDB Endowment, Vol. 7, No. 11

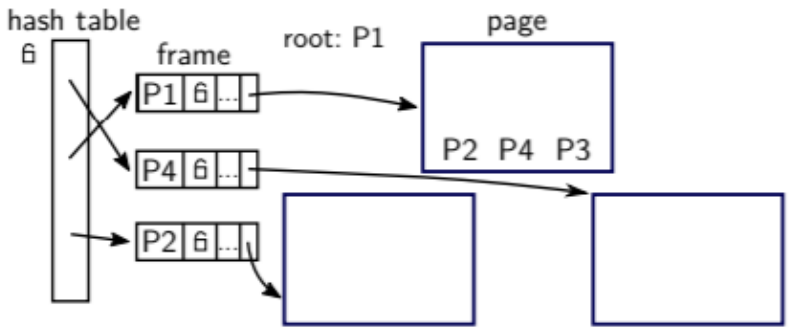
UNIVERSITY OF WATERLOO

Other Cases

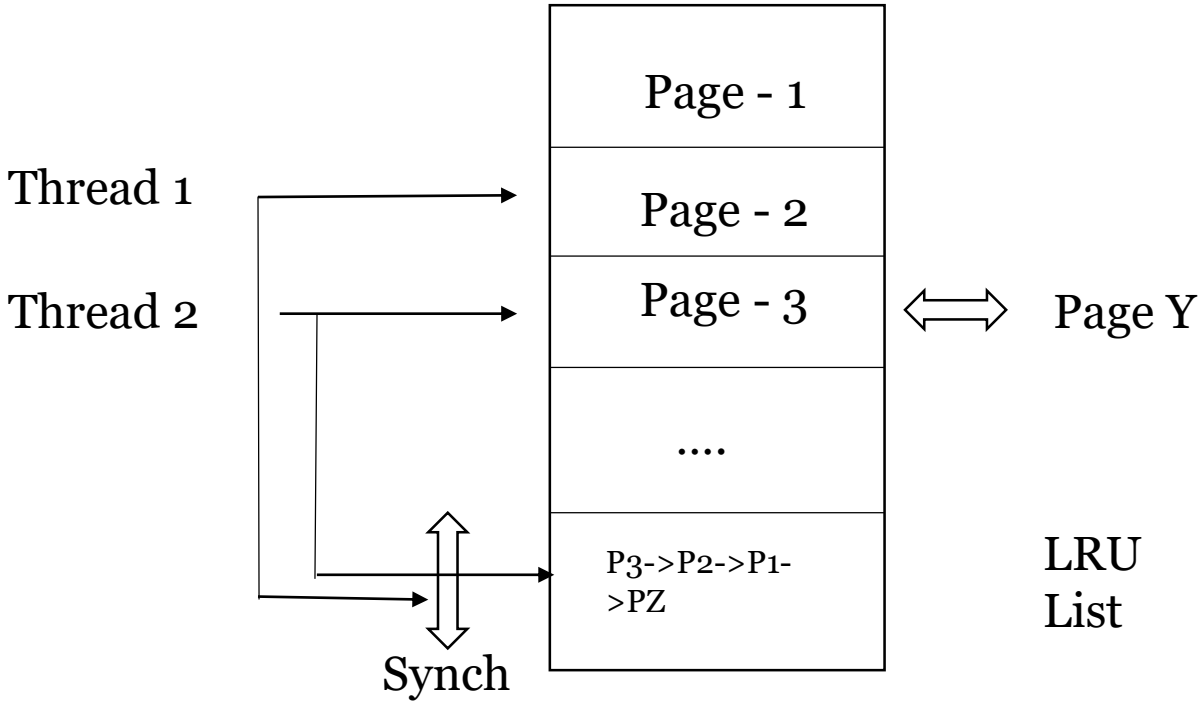
- Swapping at page level granularity – HStore
- Hardware assisted access tracking – Hyper
- Optimized storage engines like Bw-Tree/LLAMA
- Graefe et al. Swizzling for buffer managers

Buffer Manager - Challenges

- Granularity – chunk or page. (almost, similar terms)
- Handling References
- Page replacement strategy
- Page eviction
- Synchronization issues
- Address Translation



(a) traditional buffer manager

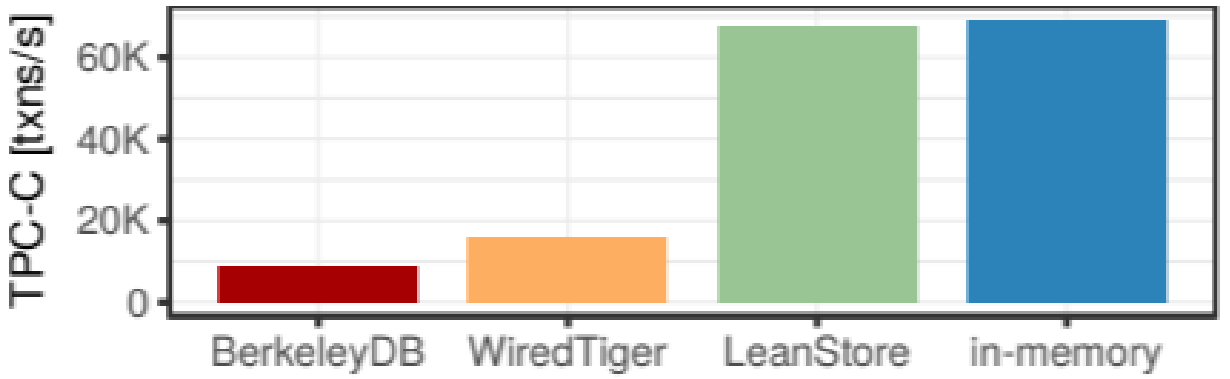


Run Time Space

UNIVERSITY OF WATERLOO

What is LeanStore ?

- LeanStore Overview
  - Hold everything in-mem. B-tree layout for pages and pointers as references to data
  - Granularity of operations – row level
  - Modified buffer manager
- Goal - Inmemory databases to use disks avoiding slow parts of a disk based database.



1. Single-threaded in-memory TPC-C performance (100 warehouses).

Pointer Swizzling

- Use way to indicate if a page referred is in memory pool or on disk.
- A reference containing an in-memory pointer is called swizzled, one that stores an ondisk page identifier is called unswizzled.
- Each page reference is 8 byte and is known as a **swip**.

```
if ( MSB set )  then
        object in memory
else
        object in disk
```

Page Replacement Policy

- Conventional policies –
  - LRU lists which have overhead of maintaining lists and references and memory
  - Using Counters – Extra operations and other concurrency issues
- Random page selection for eviction

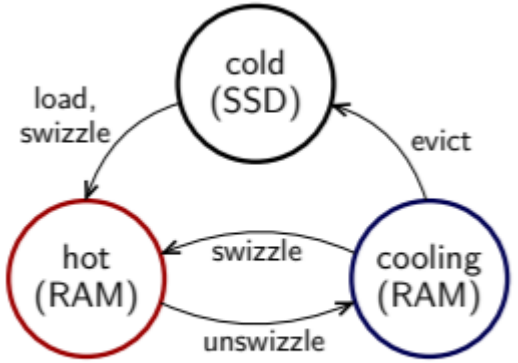
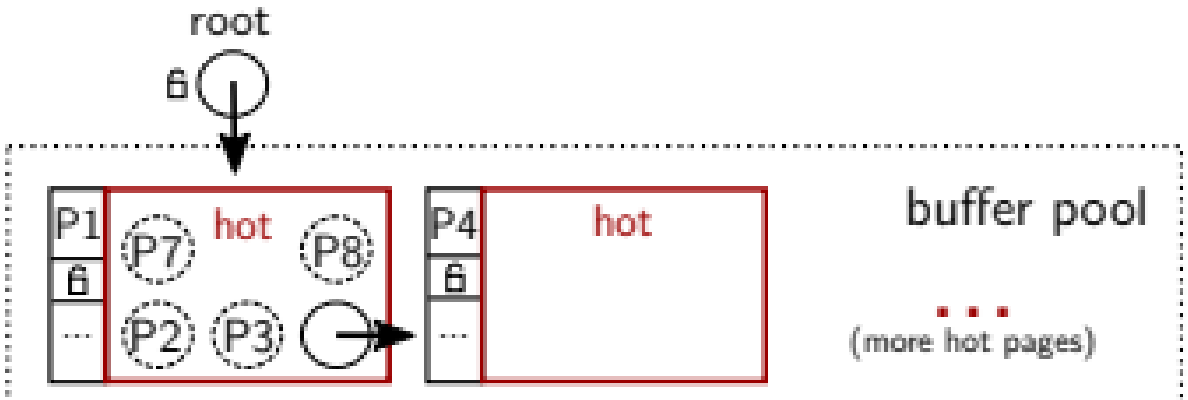


Fig. 3. The possible states of a page.

Synchronization

- Pages are organized in a tree hierarchy
- Each page has only one parent, single reference.
- Avoid latches – supported by swips



Handling Pages Effectively

- Identify pages to be evicted i.e., handle randomness
- Handle page eviction lists or order
- Handle concurrency during eviction
- Handling I/O

LeanStore Page cycle

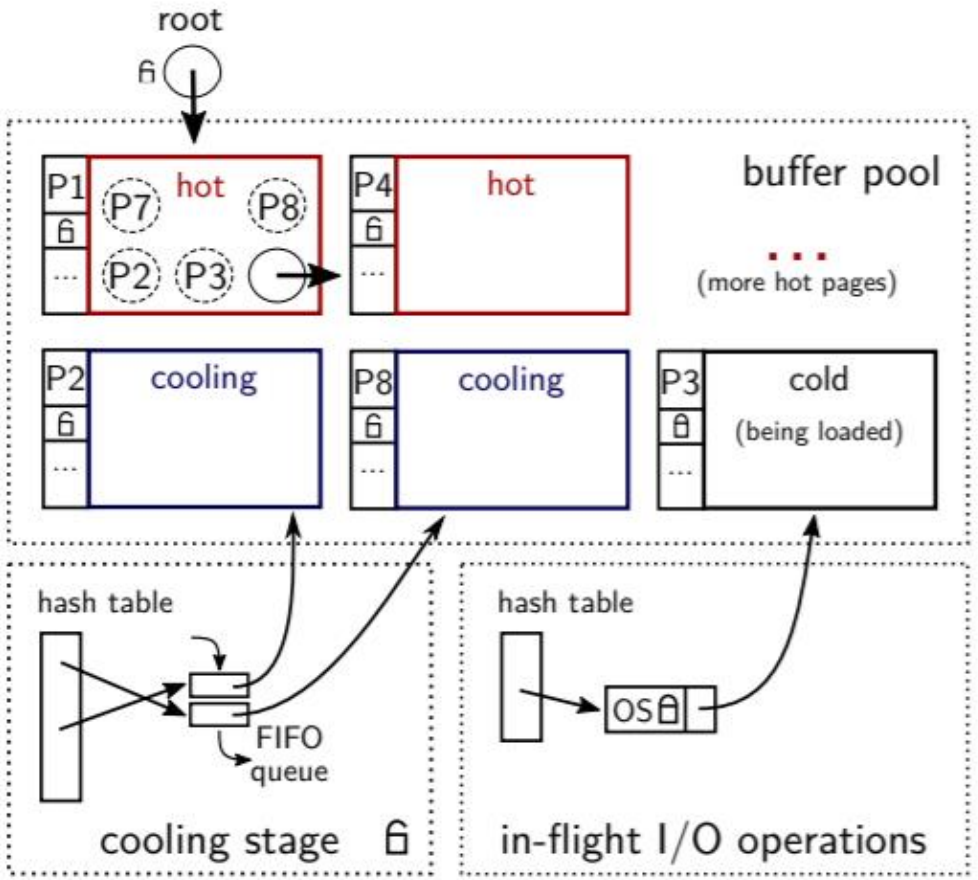


Fig. 4. Overview of LeanStore's data structures. Page P1 represents a root page (e.g., of a B-tree) with 5 child pages (P7, P8, P2, P3, P4). Pages P1 and P4 are hot (swizzled), while pages P2 and P8 are cooling (unswizzled). (In reality, the vast majority of in-memory pages will be classified as hot.) Pages P7 and P3 are on persistent storage with P3 currently being loaded.

UNIVERSITY OF WATERLOO

LeanStore Page cycle – Identifying pages for eviction

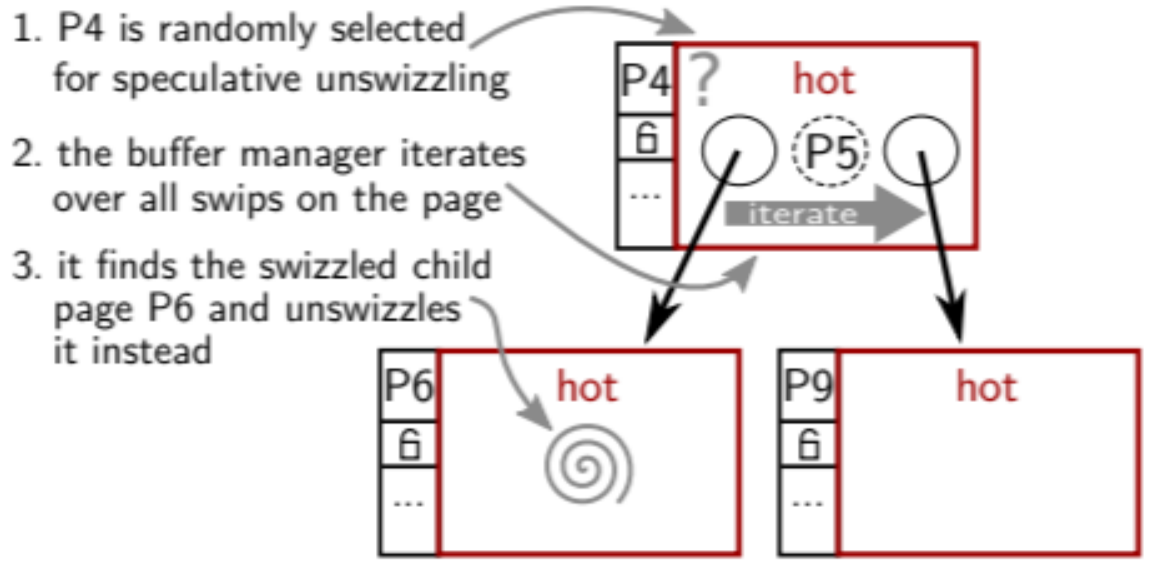


Fig. 5. Inner pages can only be unswizzled after all their child pages.

LeanStore Page cycle – Handling concurrent page access

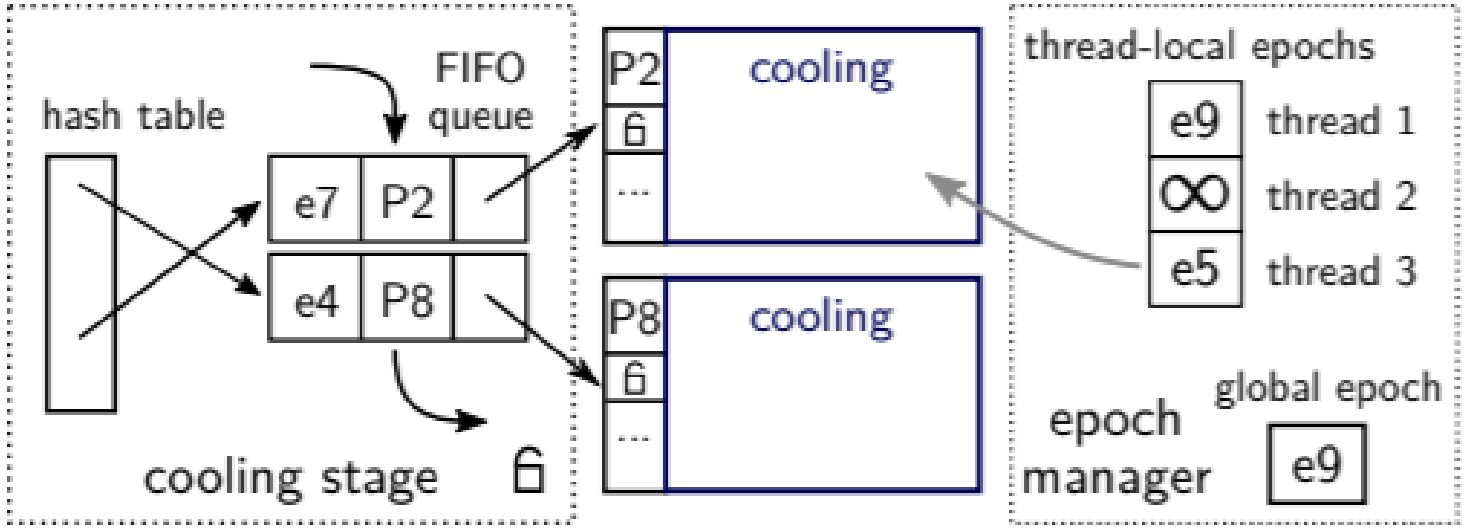


Fig. 6. Epoch-based reclamation.

LeanStore Implementation

- In place modifications
- For structural changes, policy is analogous to two phase locking
- Interleaving buffer frames with page content – cache coherence
- Reusing deleted pages via thread local cache
- Background task support for flushing modified pages
- I/O prefetching
- Hinting

Performance Analysis – In-memory

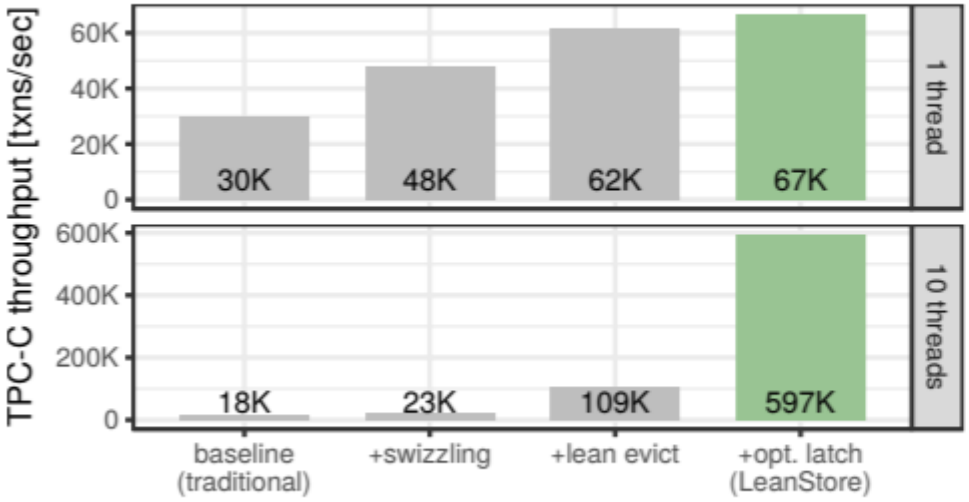


Fig. 7. Impact of the 3 main LeanStore features.

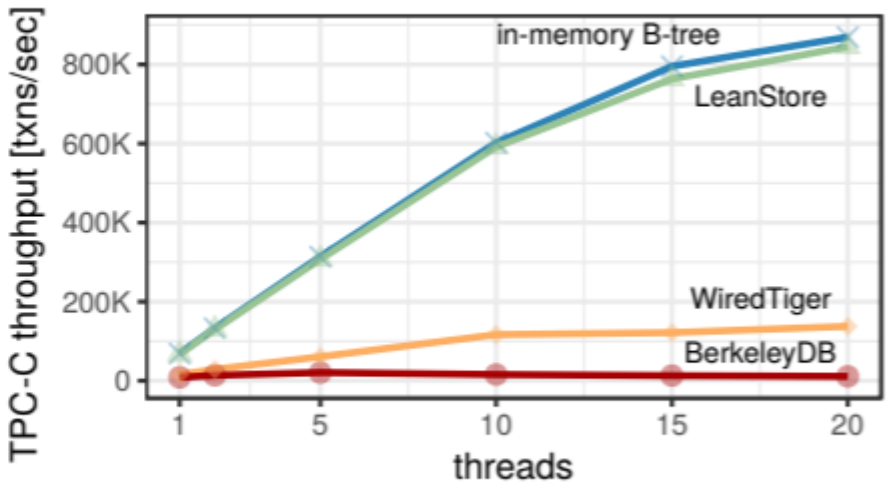


Fig. 8. Multi-threaded, in-memory TPC-C on 10-core system.

UNIVERSITY OF WATERLOO

Performance analysis – scale out : multiple cores, out of memory

TABLE I
LEANSTORE SCALABILITY RUNNING TPC-C ON 60-CORE NUMA SYSTEM

	txns/sec	speedup	remote accesses
1 thread	45K	1.0×	7%
60 threads: baseline	1,500K	33.3×	77%
+ warehouse affinity	2,270K	50.4×	77%
+ pre-fault memory	2,370K	52.7×	75%
+ NUMA awareness	2,560K	56.9×	14%

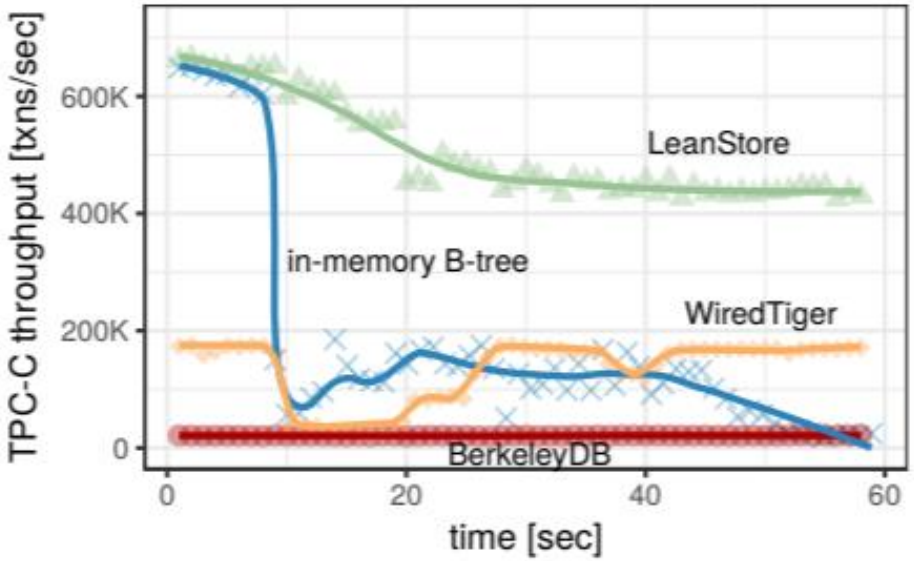


Fig. 9. TPC-C with 20 GB buffer pool (100 warehouses, 20 threads). The data grows from 10 GB to 50 GB—exceeding the buffer pool.

Performance analysis – Other Measurements

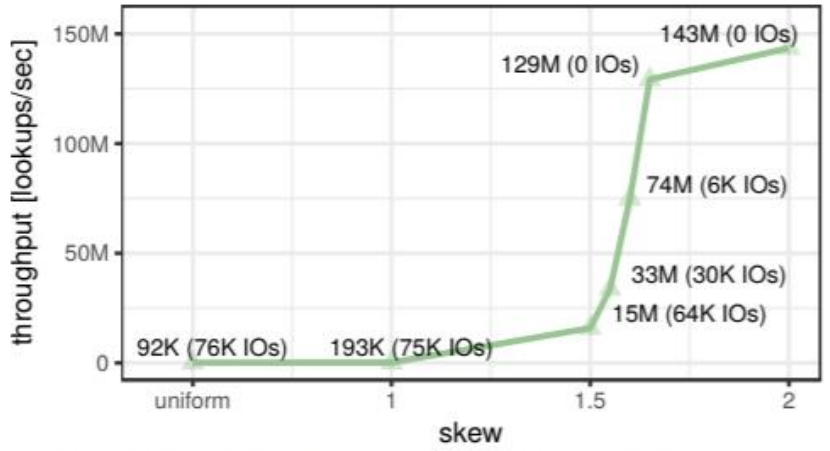


Fig. 10. Lookup performance and number of I/O operations per second (20 threads, 5 GB data set, 1 GB buffer pool).

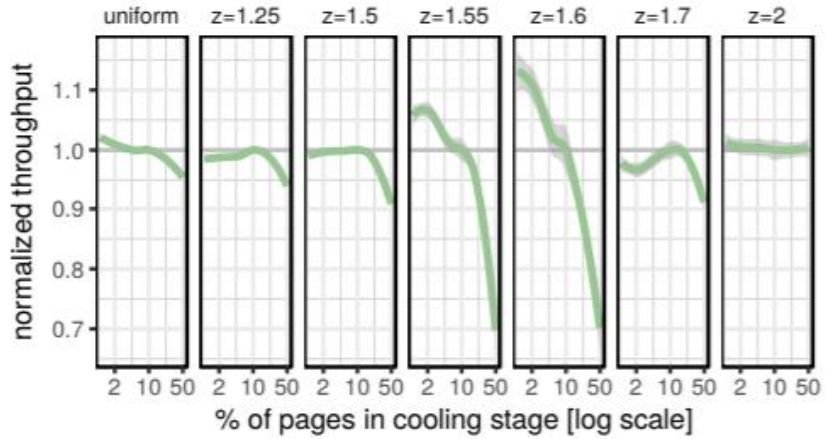


Fig. 11. Effect of cooling stage size on throughput. The throughput is normalized by the 10% cooling pages setting.

Random	FIFO	LeanEvict 5%	LeanEvict 10%	LeanEvict 20%	LeanEvict 50%	LRU	2Q	OPT
92.5%	92.5%	92.7%	92.8%	92.9%	93.0%	93.1%	93.8%	96.3%

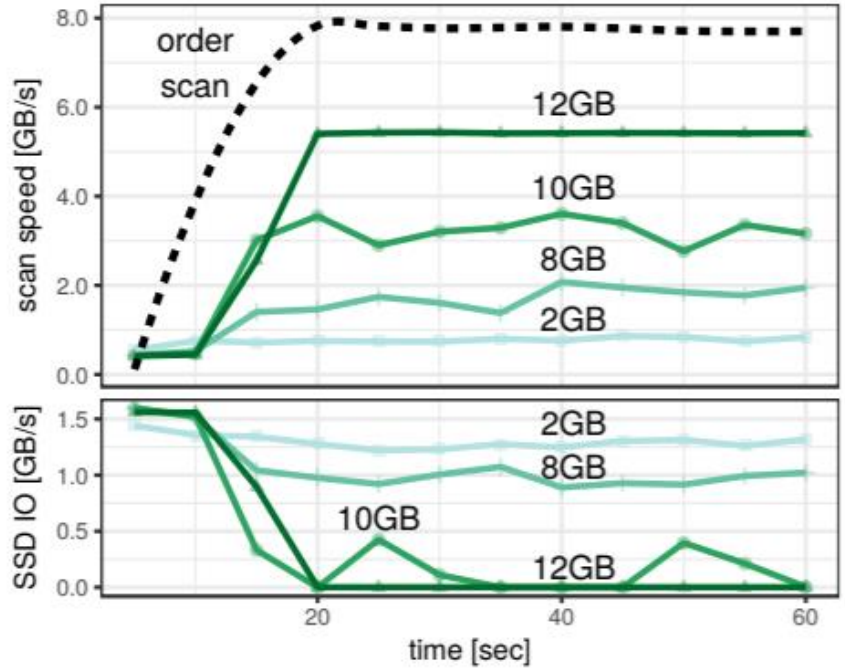


Fig. 12. Concurrent scan of the 0.7 GB order table and the 10 GB orderline table using buffer pool sizes between 2 GB and 12 GB.

Conclusion

As per me, in short - “A demand driven page in & out swap based buffer pool with controlled paths of references and using minimalistic runtime memory.”

Few thoughts

- Pointer swizzling is tweaking unused bits in referenced pointers. Is this a limitation or way of exploiting unused resources ?
- LeanStore provides page level granularity for holding data. With tuples how is it managed ? Chances are there that a given huge tuple can cross multiple pages and result in splitting of it.
- LeanStore's buffer pool : Does it guarantee fair eviction and data localisation ?
- Page Eviction : LRU vs randomly chosen page eviction. What is the guarantee that the required data swaps in without blocks ?
- Need of the hour : "Hybrid" systems (Disk + In-Memory) Are not these analogous to conventional databases with if's and buts ? If yes, are we trying to over engineer conventional dbs ? If no, what is making them different ?

References

- Viktor Leis, Michael Haubenschild, Alfons Kemper, Thomas Neumann. LeanStore: In-Memory Data Management beyond Main Memory. *Proc. 34th IEEE International Conference on Data Engineering*, pages 185-196, 2018
- Pictures are taken from google images.