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.
Tackling I/O Path

- Cutting down I/O
- Is it InMemory Database?
Tackling Concurrent Execution

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

```
SELECT COUNT(*)
FROM employees
WHERE phone_number LIKE '650%';
```
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

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 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 Hekatan 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 1-4% 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 drops 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, 11].

Microsoft has developed a memory-optimized database engine, code named Hekatan, targeted for OLTP workloads. The Hekatan 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 Hekatan. Table data can be queried and updated in the same way in regular tables. To speed up processing even further, a T-SQL stored procedure that references only Hekatan tables can be compiled into native machine code. Further details about the design of Hekatan 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 Hekatan requires that a memory-optimized table fit 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 (cold applications) are entirely unaware of where a row is stored.

The goal of our project, called Project Siberia, is to enable the Hekatan engine to automatically and transparently manage cold data on cheaper secondary storage. We avoid 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 access frequencies off line as described in more detail in [13]. One could also use a traditional caching approach such as LFU or LRU, but the benefit is high with speed and time. As reported in [17], experiments showed that summarizing product [L2] showed that cold data classification is an effective way of reducing the cost of looking up in an in-memory hash table and added 16 bytes to each record. This 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 methods**: 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 variation 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, 7].
- **Cold data access and migration mechanisms**: mechanisms for efficiently migrating, reading, and updating data on cold storage that dovetail with Hekatan’s optimistic multi-version concurrency control scheme [11].

In this paper, we focus on the first point, namely, how to migrate records to and from the cold store and how to store cold 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 or only...
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
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.

**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.

```plaintext
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.
LeanStore Page cycle – Identifying pages for eviction

1. P4 is randomly selected for speculative unswizzling
2. The buffer manager iterates over all swips on the page
3. It finds the swizzled child page P6 and unswizzles it instead

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

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>LEANSTORE SCALABILITY RUNNING TPC-C ON 60-CORE NUMA SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>txns/sec</td>
</tr>
<tr>
<td>1 thread</td>
<td>45K</td>
</tr>
<tr>
<td>60 threads: baseline</td>
<td>1,500K</td>
</tr>
<tr>
<td>+ warehouse affinity</td>
<td>2,270K</td>
</tr>
<tr>
<td>+ pre-fault memory</td>
<td>2,370K</td>
</tr>
<tr>
<td>+ NUMA awareness</td>
<td>2,560K</td>
</tr>
</tbody>
</table>

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.

Fig. 12. Concurrent scan of the 0.7 GB order table and the 10GB order line 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.