Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Larger-than-Memory Databases

@Andy_Pavlo // 15-721 // Spring 2020
ADMINISTRIVIA

April 22: Final Exam Released

April 29: Guest Speaker (Live)

May 4: Code Review #2 Submission

May 5: Final Presentations (Live)

May 13: Final Exam Due Date
OBSERVATION

DRAM is expensive, son.
→ Expensive to buy.
→ Expensive to maintain.

It would be nice if our in-memory DBMS could use cheaper storage without having to bring in the entire baggage of a disk-oriented architecture.
TODAY’S AGENDA

Background
Implementation Issues
Real-world Examples
LARGER-THAN-MEMORY DATABASES

Allow an in-memory DBMS to store/access data on disk without bringing back all the slow parts of a disk-oriented DBMS.
→ Minimize the changes that we make to the DBMS that are required to deal with disk-resident data.

Need to be aware of hardware access methods
→ In-memory Storage = Tuple-Oriented
→ Disk Storage = Block-Oriented
OLAP queries generally access the entire table. Thus, there is not anything about OLAP queries that an in-memory DBMS would handle differently than a disk-oriented DBMS.
OLTP

OLTP workloads almost always have **hot** and **cold** portions of the database.
→ We can assume txns will almost always access hot tuples.

The DBMS needs a mechanism to move cold data out to disk and then retrieve it if it is ever needed again.
LARGER-THAN-MEMORY DATABASES

**In-Memory Index**

- Tuple #00
- Tuple #01
- Tuple #02
- Tuple #03
- Tuple #04

**In-Memory Table Heap**

**Cold-Data Storage**
LARGER-THAN-MEMORY DATABASES

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

Tuple #00
Tuple #01
Tuple #02
Tuple #03
Tuple #04
LARGER-THAN-MEMORY DATABASES

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

Evicted Tuple Block

header
Tuple #01
Tuple #03
Tuple #04
LARGER-THAN-MEMORY DATABASES

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

header
Tuple #01
Tuple #03
Tuple #04

Evicted Tuple Block
LARGER-THAN-MEMORY DATABASES

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

SELECT * FROM table
WHERE id = <Tuple #01>
OLTP ISSUES

Run-time Operations
→ Cold Data Identification

Eviction Policies
→ Timing, Evicted Metadata

Data Retrieval Policies
→ Granularity, Retrieval Mechanism, Merging
COLD DATA IDENTIFICATION

Choice #1: On-line
→ The DBMS monitors txn access patterns and tracks how often tuples/pages are used.
→ Embed the tracking meta-data directly in tuples/pages.

Choice #2: Off-line
→ Maintain a tuple access log during txn execution.
→ Process in background to compute frequencies.
**EVICTION TIMING**

**Choice #1: Threshold**
- The DBMS monitors memory usage and begins evicting tuples when it reaches a threshold.
- The DBMS must manually move data.

**Choice #2: On Demand**
- The DBMS/OS runs a replacement policy to decide when to evict data to free space for new data that is needed.
EVICTED TUPLE METADATA

Choice #1: Tuple Tombstones
→ Leave a marker that points to the on-disk tuple.
→ Update indexes to point to the tombstone tuples.

Choice #2: Bloom Filters
→ Use approximate data structure for each index.
→ Check both index + filter for each query.

Choice #3: DBMS Managed Pages
→ DBMS tracks what data is in memory vs. on disk.

Choice #4: OS Virtual Memory
→ OS tracks what data is on in memory vs. on disk.
**EVICTED TUPLE METADATA**

**In-Memory Index**

- Tuple #01
- Tuple #03
- Tuple #04

**Access Frequency**

- Tuple #00
- Tuple #01
- Tuple #02
- Tuple #03
- Tuple #04
- Tuple #05

**In-Memory Table Heap**

- Tuple #00
- Tuple #01
- Tuple #02
- Tuple #03
- Tuple #04

**Cold-Data Storage**
EVICTED TUPLE METADATA

In-Memory Index

- In-Memory Table Heap
  - Tuple #00
  - Tuple #01
  - Tuple #02
  - Tuple #03
  - Tuple #04

Cold-Data Storage

Access Frequency

- Tuple #00
- Tuple #01
- Tuple #02
- Tuple #03
- Tuple #04
- Tuple #05
EVICTED TUPLE METADATA

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

Access Frequency

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Access Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple #00</td>
<td></td>
</tr>
<tr>
<td>Tuple #01</td>
<td></td>
</tr>
<tr>
<td>Tuple #02</td>
<td></td>
</tr>
<tr>
<td>Tuple #03</td>
<td></td>
</tr>
<tr>
<td>Tuple #04</td>
<td></td>
</tr>
<tr>
<td>Tuple #05</td>
<td></td>
</tr>
</tbody>
</table>
EVICTED TUPLE METADATA

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

Tuple #00

Tuple #02

<Tuple, Offset>

<Tuple, Offset>

<Tuple, Offset>

header

TUPLE #01

TUPLE #03

TUPLE #04
EVICTED TUPLE METADATA

Does 'x' exist?

In-Memory Index

Bloom Filter

In-Memory Table Heap

Tuple #00

Tuple #02

Cold-Data Storage

header

Tuple #01

Tuple #03

Tuple #04

Index
EVICTED TUPLE METADATA

Does 'x' exist?

In-Memory Index

In-Memory Table Heap

Cold-Data Storage

Bloom Filter

header

Tuple #01

Tuple #03

Tuple #04

Index
DATA RETRIEVAL GRANULARITY

Choice #1: All Tuples in Block
→ Merge all the tuples retrieved from a block regardless of whether they are needed.
→ More CPU overhead to update indexes.
→ Tuples are likely to be evicted again.

Choice #2: Only Tuples Needed
→ Only merge the tuples that were accessed by a query back into the in-memory table heap.
→ Requires additional bookkeeping to track holes.
MERGING THRESHOLD

Choice #1: Always Merge
→ Retrieved tuples are always put into table heap.

Choice #2: Merge Only on Update
→ Retrieved tuples are only merged into table heap if they are used in an **UPDATE** query.
→ All other tuples are put in a temporary buffer.

Choice #3: Selective Merge
→ Keep track of how often each block is retrieved.
→ If a block's access frequency is above some threshold, merge it back into the table heap.
RETRIEVAL MECHANISM

Choice #1: Abort-and-Restart
→ Abort the txn that accessed the evicted tuple.
→ Retrieve the data from disk and merge it into memory with a separate background thread.
→ Restart the txn when the data is ready.
→ Requires MVCC to guarantee consistency for large txns that access data that does not fit in memory.

Choice #2: Synchronous Retrieval
→ Stall the txn when it accesses an evicted tuple while the DBMS fetches the data and merges it back into memory.
IMPLEMENTATIONS

Tuples
- H-Store – Anti-Caching
- Hekaton – Project Siberia
- EPFL’s VoltDB Prototype
- Apache Geode – Overflow Tables

Pages
- LeanStore – Hierarchical Buffer Pool
- Umbra – Variable-length Buffer Pool
- MemSQL – Columnar Tables
H-STORE – ANTI-CACHING

On-line Identification
Administrator-defined Threshold
Tombstones
Abort-and-restart Retrieval
Block-level Granularity
Always Merge
HEKATON – PROJECT SIBERIA

Off-line Identification
Administrator-defined Threshold
Bloom Filters
Synchronous Retrieval
Tuple-level Granularity
Always Merge
Enabling Efficient OS Paging for Main-Memory OLTP Databases

Damon 2013
In-Memory Table Heap

Cold Data Storage

mlock

Hot Tuples

Cold Tuples

Tuple #00

Tuple #01

Tuple #02
In-Memory Table Heap

- Tuple #00
- Tuple #01
- Tuple #02

Hot Tuples

Cold Tuples

EPFL VOLTDB

mlock

Cold-Data Storage
EPFL VOLTDB

In-Memory Table Heap

mlock

Hot Tuples

Cold Tuples

Tuple #00

Tuple #03

Tuple #02

Tuple #01

Cold-Data Storage
In-Memory Table Heap

- Tuple #00
- Tuple #03
- Tuple #02

Cold Data Storage

- Tuple #01

mlock

Hot Tuples

Cold Tuples
In Memory Table Heap

Cold Tuples

Hot Tuples

mlock

EPFL VOLTDDB

Cold-Data Storage
In Memory Table Heap

Cold Data Storage

mlock

Hot Tuples

Cold Tuples

Tuple #00

Tuple #03

Tuple #02

Tuple #01
EPFL VOLTDDB

In-Memory Table Heap

Hot Tuples

Cold Tuples

Cold-Data Storage

mlock

Tuple #00

Tuple #03

Tuple #02

Tuple #01
APACHE GEODE – OVERFLOW TABLES

On-line Identification
Administrator-defined Threshold
Tombstones (?)
Synchronous Retrieval
Tuple-level Granularity
Merge Only on Update (?)

Source: Apache Geode
OBSERVATION

The approaches that we have discussed so far are based on tuples.
→ The DBMS must track meta-data about individual tuples.
→ Not reducing storage overhead of indexes.

Need a unified way to evict cold data from both tables and indexes with low overhead...
LeanStore

Prototype in-memory storage manager from TUM that supports larger-than-memory databases.
→ Handles both tuples + indexes
→ Not part of the HyPer project.

Hierarchical + Randomized Block Eviction
→ Use pointer swizzling to determine whether a block is evicted or not.
POINTER SWIZZLING

Switch the contents of pointers based on whether the target object resides in memory or on disk.
→ Use first bit in address to tell what kind of address it is.
→ Only works if there is only one pointer to the object.

(1)<PageId, Offset>
Switch the contents of pointers based on whether the target object resides in memory or on disk.

→ Use first bit in address to tell what kind of address it is.
→ Only works if there is only one pointer to the object.
REPLACEMENT STRATEGY

Randomly select blocks for eviction.
→ Don't have to maintain meta-data every time a txn accesses a hot block.
→ Only track accesses for cold data, which should be rare if it is cold.

Unswizzle their pointer but leave in memory.
→ Add to a FIFO queue of blocks staged for eviction.
→ If page is accessed again, remove from queue.
→ Otherwise, evict pages when reaching front of queue.
Blocks are organized in a tree hierarchy.
→ Each page has only one parent, which means that there is only a single pointer.

The DBMS can only evict a block if its children are also evicted.
→ This avoids the problem of evicting blocks that contain swizzled pointers.
→ If a block is selected but it has in-memory children, then it automatically switches to select one of its children.
BLOCK HIERARCHY

Unswizzled Pointer

Swizzled Pointer

Hash Table

Eviction Queue

Source: Viktor Leis
BLOCK HIERARCHY

Unswizzled Pointer

Swizzled Pointer

Hash Table

Eviction Queue

B0

B1

B2

B3

Hot Stage

Cooling Stage

Cold Stage

Source: Viktor Leis
BLOCK HIERARCHY

Hash Table

Eviction Queue

Unswizzled Pointer
Swizzled Pointer

B0

B1

B2

B3

Hot Stage
Cooling Stage
Cold Stage

Source: Viktor Leis
### BLOCK HIERARCHY

Unswizzled Pointer ➞ Hash Table ➞ Eviction Queue ➞ B0 ➞ Hot Stage

Swizzled Pointer ➞ Hash Table ➞ Eviction Queue ➞ B1 ➞ Cooling Stage ➞ Cold Stage

Source: Viktor Leis
UMBRA

New DBMS from German HyPer team at TUM.
→ Low overhead buffer pool with variable-sized pages.
→ Employs the same hierarchical organization and randomized block eviction algorithm from LeanStore.
→ Uses virtual memory to allocate storage but the DBMS manages block eviction on its own.

DBMS stores relations as index-organized tables, so there is no separate management needed to handle index blocks.
VARIABLE-SIZED BUFFER POOL

Source: Thomas Neumann
MEMSQL – COLUMNAR TABLES

Administrator manually declares a table as a disk-resident columnar table with zone maps.
→ Pre-2017: Used mmap but this was a bad idea.
→ Pre-2019: DBMS splits columns into 1m tuple segments.
→ Current: Unified single logical table format that combines delta store with column store.

No Evicted Metadata
Synchronous Retrieval
Always Merge

Source: MemSQL
PARTING THOUGHTS

Today was about working around the block-oriented access and slowness of secondary storage.

Fast and cheap byte-addressable NVM will make this lecture unnecessary.
NEXT CLASS

Server-side Application Logic