Lecture #11

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Larger-than-Memory Databases

@Andy_Pavlo // 15-721 // Spring 2019

ADMINISTRIVIA

Feb 27: Project #1 is due

Feb 27: Project #2 will be released

Mar 4: Extra Credit assignment will be released

Mar 6: Mid-term Exam

Mar 18: Project #2 Proposals

UPCOMING DATABASE EVENTS

Splice Machine Tech Talk

→ Thursday Feb 21st @ 12:00pm
→ CIC 4th Floor
→ CEO/Co-Found Monte Zweben (CMU'85)

BLOOM FILTERS

Probabilistic data structure (bitmap) that answers set membership queries.

→ False negatives will never occur.
→ False positives can sometimes occurs.

Insert(x):

→ Use *k* hash functions to set bits in the filter to 1.

Lookup(x):

→ Check whether the bits are 1 for each hash function.

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

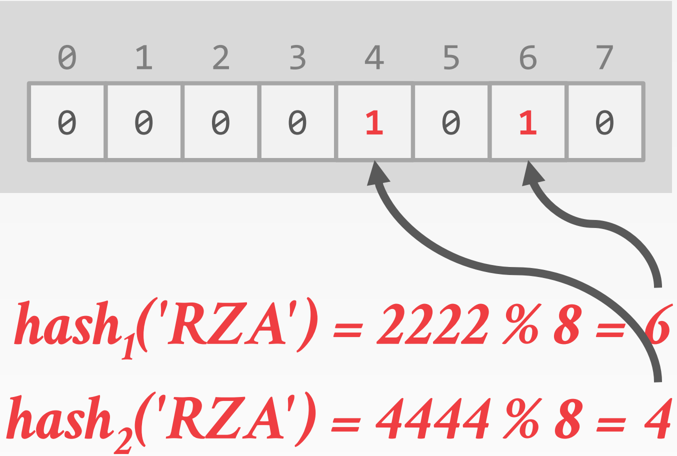
Insert('RZA')

$hash_1(\text{'RZA'}) = 2222 \% 8 = 6$

$hash_2(\text{'RZA'}) = 4444 \% 8 = 4$

BLOOM FILTERS

Bloom Filter



0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

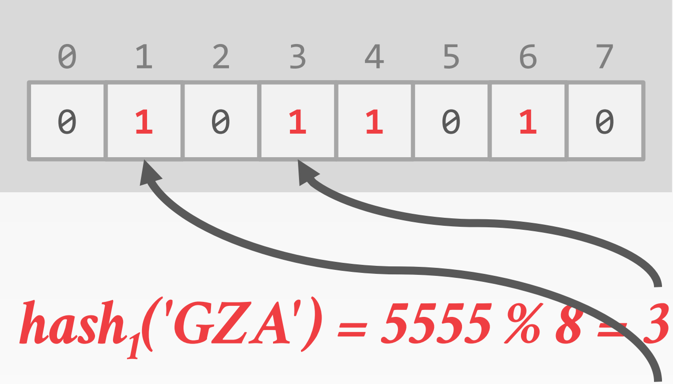
$hash_1('RZA') = 2222 \% 8 = 6$

$hash_2('RZA') = 4444 \% 8 = 4$

Insert('RZA')

BLOOM FILTERS

Bloom Filter



0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

Insert('GZA')

$hash_1$('GZA') = 5555 % 8 = 3

$hash_2$('GZA') = 7777 % 8 = 1

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

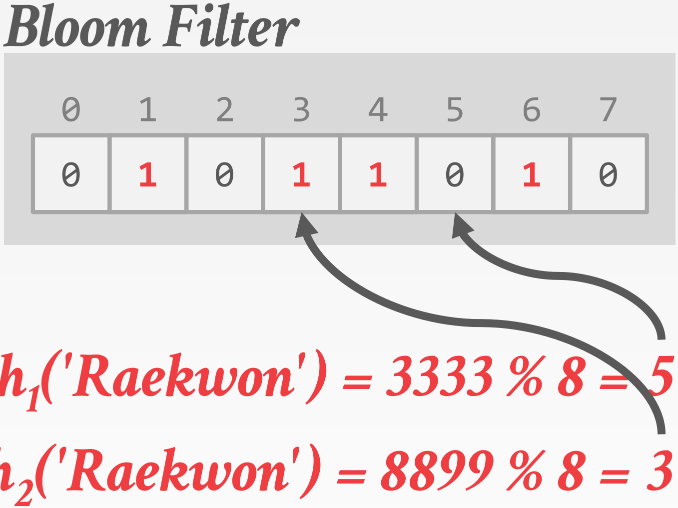
Insert('GZA')

Lookup('Raekwon')

$hash_1(\text{'Raekwon'}) = 3333 \% 8 = 5$

$hash_2(\text{'Raekwon'}) = 8899 \% 8 = 3$

BLOOM FILTERS



hash$_1$('Raekwon') = 3333 % 8 = 5

hash$_2$('Raekwon') = 8899 % 8 = 3

Insert('RZA')

Insert('GZA')

Lookup('Raekwon') → *FALSE*

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1(\text{'ODB'}) = 6699 \,\%\, 8 = 3$

$hash_2(\text{'ODB'}) = 9966 \,\%\, 8 = 6$

Insert('RZA')

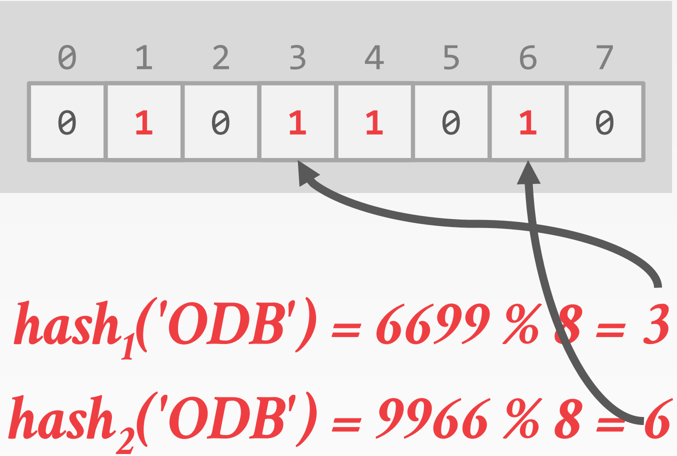
Insert('GZA')

Lookup('Raekwon') → ***FALSE***

Lookup('ODB')

BLOOM FILTERS

Bloom Filter



0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('ODB') = 6699 \% 8 = 3$

$hash_2('ODB') = 9966 \% 8 = 6$

Insert('RZA')

Insert('GZA')

Lookup('Raekwon') → ***FALSE***

Lookup('ODB') → ***TRUE***

OBSERVATION

DRAM is expensive, son.

- → Expensive to buy.
- → Expensive to maintain.

It would be nice if our in-memory DBMS could use cheaper storage.

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.

Need to be aware of hardware access methods

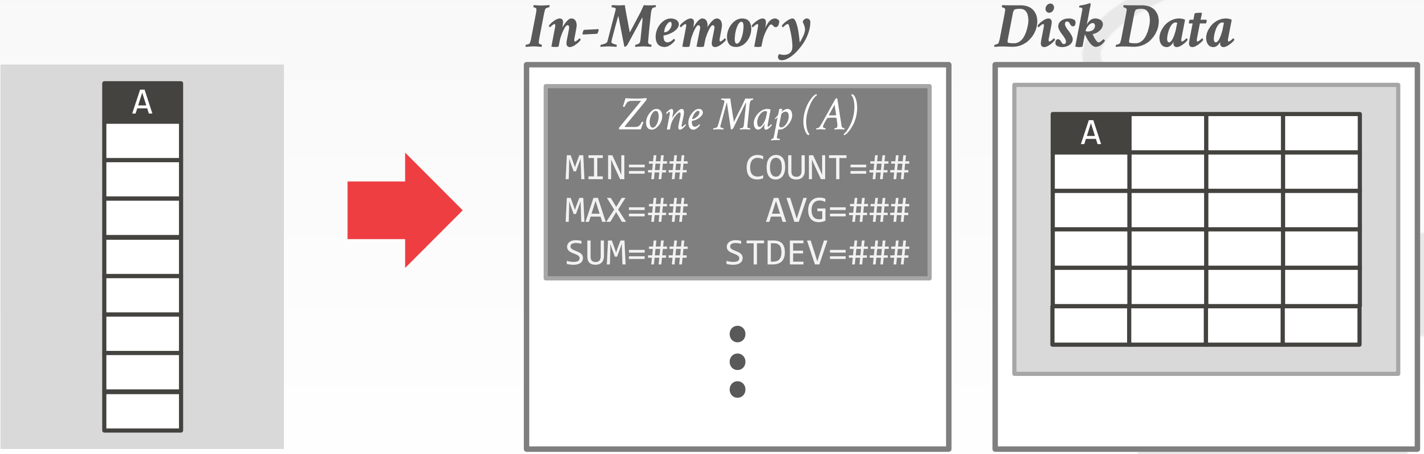
- → In-memory Storage = Tuple-Oriented
- → Disk Storage = Block-Oriented

OLAP

OLAP queries generally access the entire table. Thus, there isn't anything about the workload for the DBMS to exploit that a disk-oriented buffer pool can't handle.

OLAP

OLAP queries generally access the entire table. Thus, there isn't anything about the workload for the DBMS to exploit that a disk-oriented buffer pool can't handle.



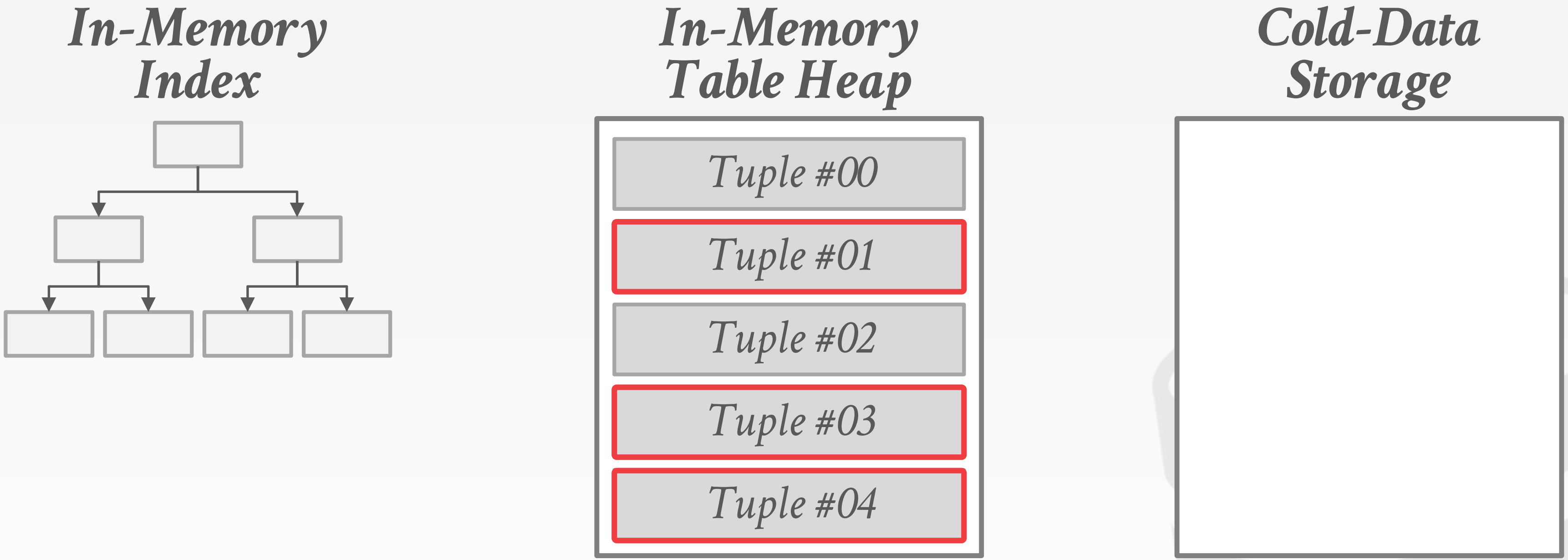
OLTP

OLTP workloads almost always have **hot** and **cold** portions of the database.

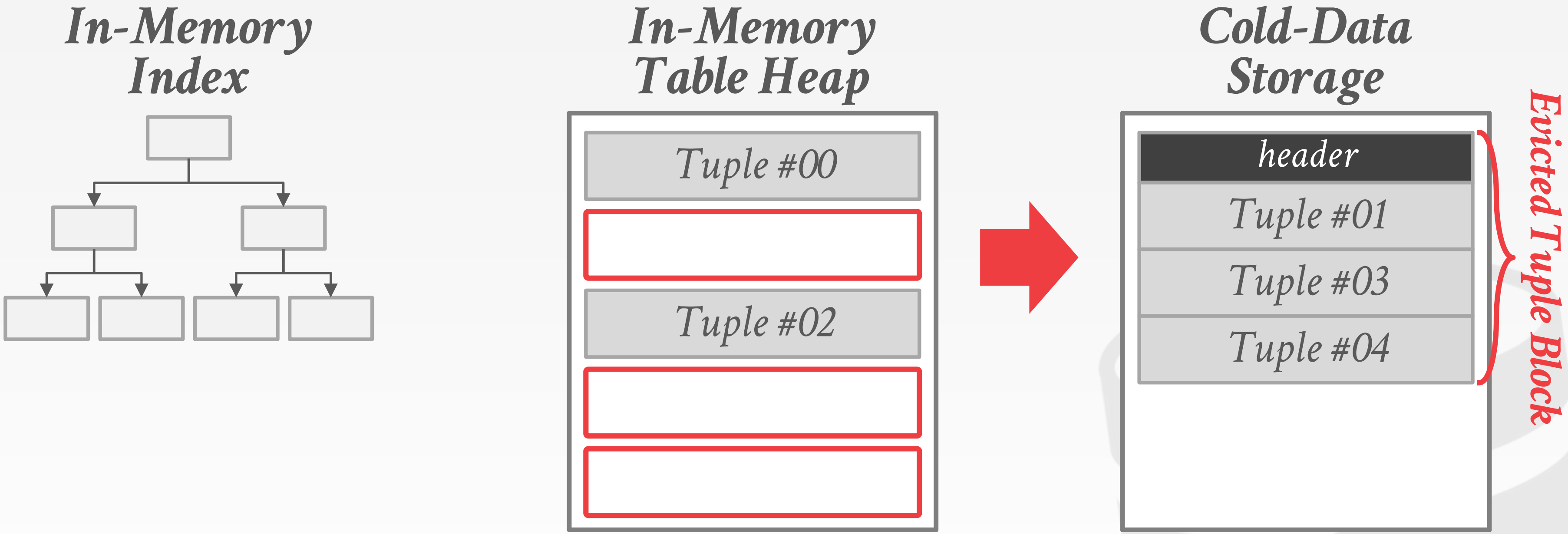
→ We can assume txns will almost always access hot tuples.

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



LARGER-THAN-MEMORY DATABASES



In-Memory Index

In-Memory Table Heap

Tuple #00

Tuple #02

Cold-Data Storage

header

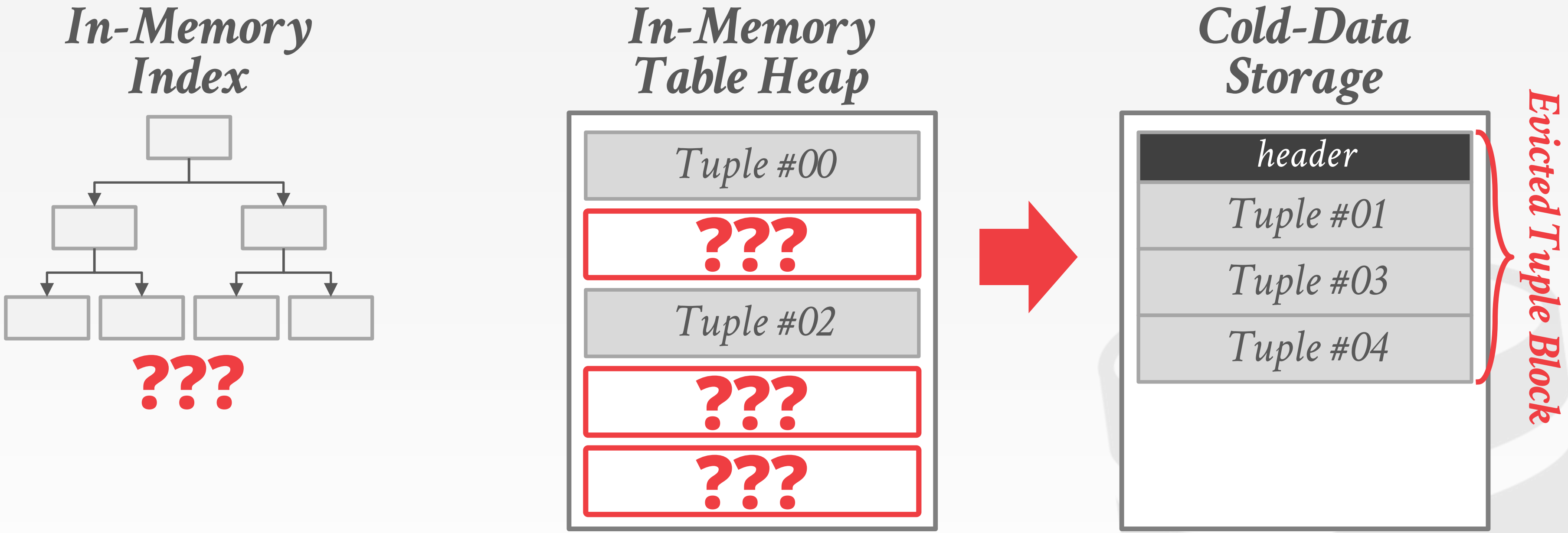
Tuple #01

Tuple #03

Tuple #04

Evicted Tuple Block

LARGER-THAN-MEMORY DATABASES

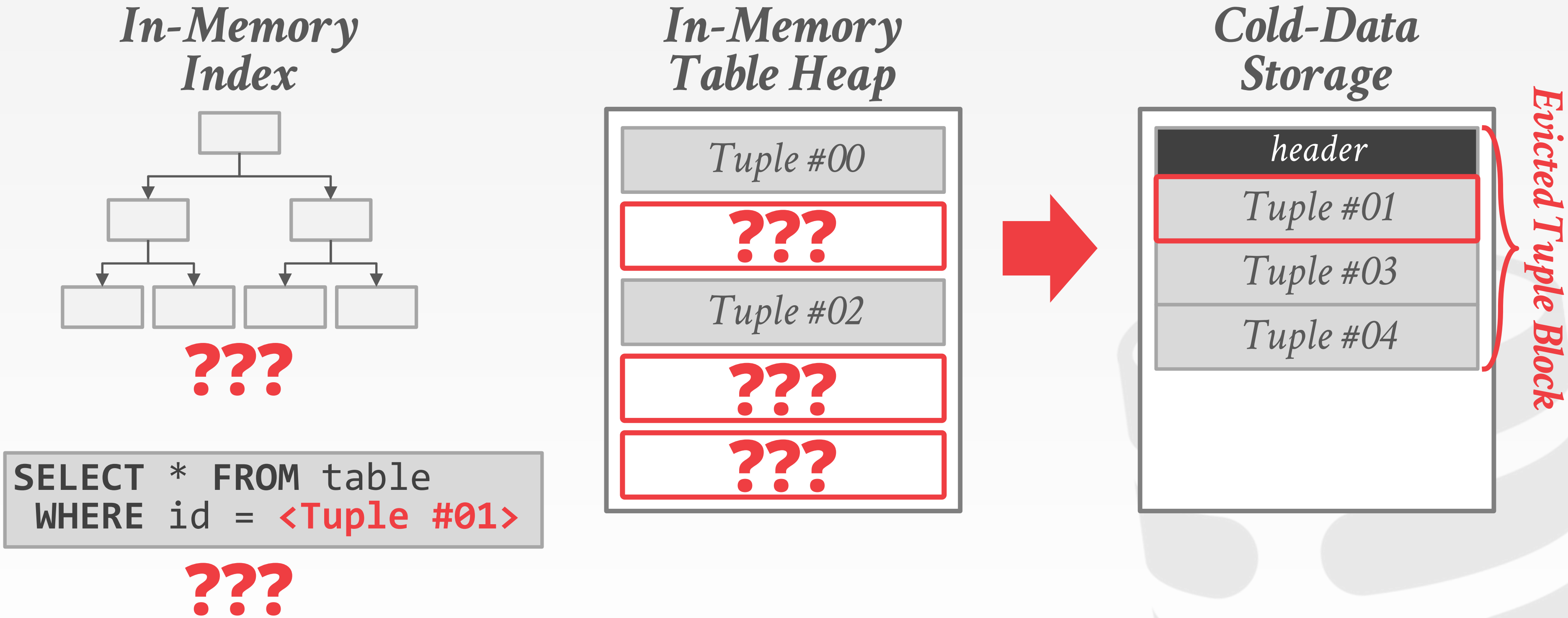


In-Memory Index

In-Memory Table Heap

Cold-Data Storage

LARGER-THAN-MEMORY DATABASES



In-Memory Index

???

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

???

In-Memory Table Heap

Tuple #00

???

Tuple #02

???

???

Cold-Data Storage

header

Tuple #01

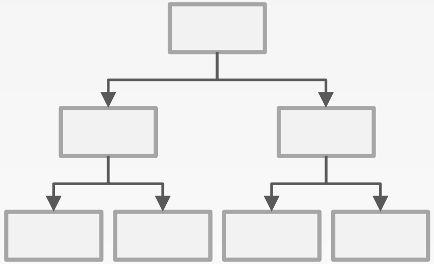
Tuple #03

Tuple #04

Evicted Tuple Block

LARGER-THAN-MEMORY DATABASES

In-Memory Index



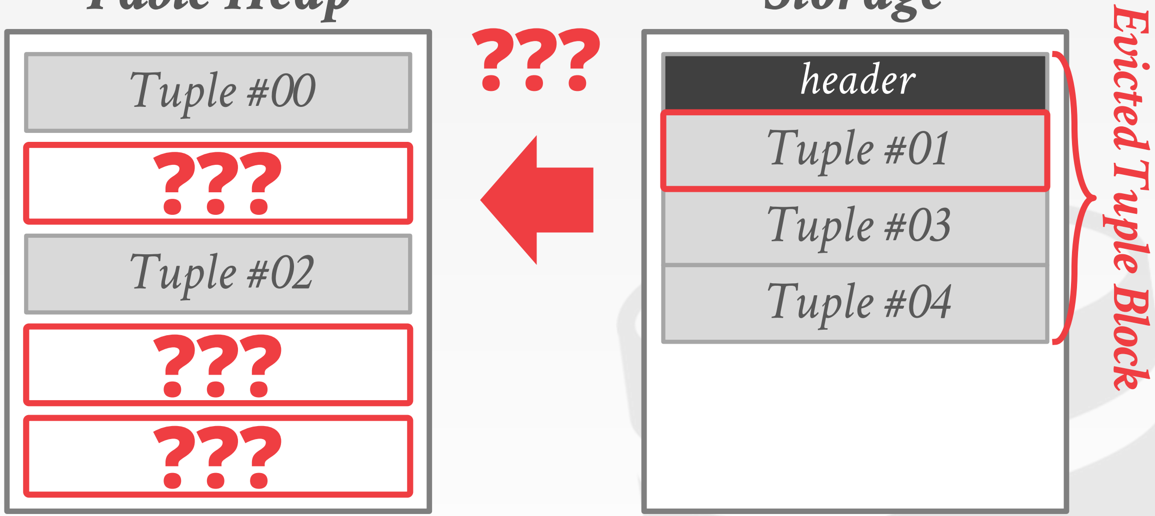
???

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

???

In-Memory Table Heap

Cold-Data Storage



AGAIN, WHY NOT MMAP?

Write-ahead logging requires that a modified page cannot be written to disk before the log records that made those changes is written.

There are no mechanisms for asynchronous read-ahead or writing multiple pages concurrently.

IN-MEMORY PERFORMANCE FOR BIG DATA
VLDB 2014

OLTP ISSUES

Run-time Operations

→ Cold Data Identification

Eviction Policies

→ Timing, Evicted Metadata

Data Retrieval Policies

→ Granularity, Retrieval Mechanism, Merging

LARGER-THAN-MEMORY DATA MANAGEMENT ON MODERN STORAGE HARDWARE FOR IN-MEMORY OLTP DATABASE SYSTEMS
DAMON 2016

COLD TUPLE IDENTIFICATION

Choice #1: On-line

→ The DBMS monitors txn access patterns and tracks how often tuples are used.

→ Embed the tracking meta-data directly in tuples.

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 has to manually move data.

Choice #2: OS Virtual Memory

- → The OS decides when it wants to move data out to disk. This is done in the background.

EVICTED TUPLE METADATA

Choice #1: Tombstones

→ Leave a marker that points to the on-disk tuple.
→ Update indexes to point to the tombstone tuples.

Choice #2: Bloom Filters

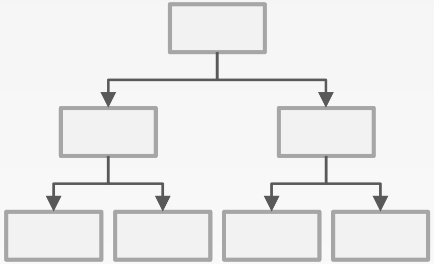
→ Use approximate data structure for each index.
→ Check both index + filter for each query.

Choice #3: OS Virtual Memory

→ The OS tracks what data is on disk. The DBMS does not need to maintain any additional metadata.

EVICTED TUPLE METADATA

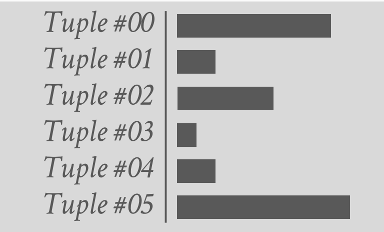
In-Memory Index



In-Memory Table Heap

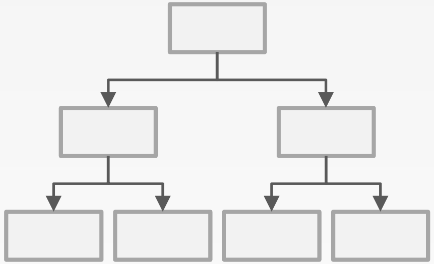
Cold-Data Storage

Access Frequency



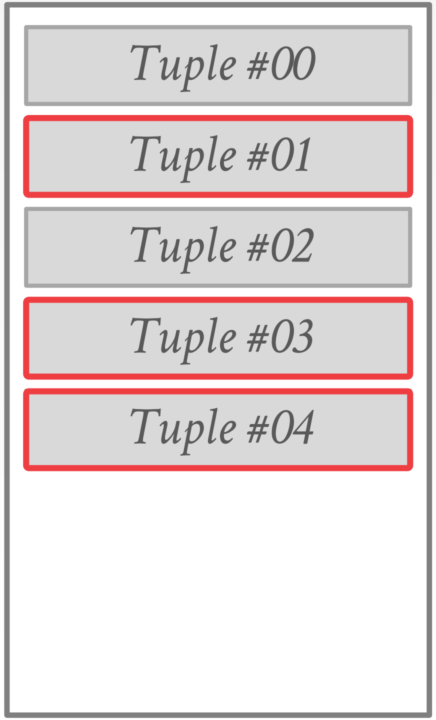
EVICTED TUPLE METADATA

In-Memory Index

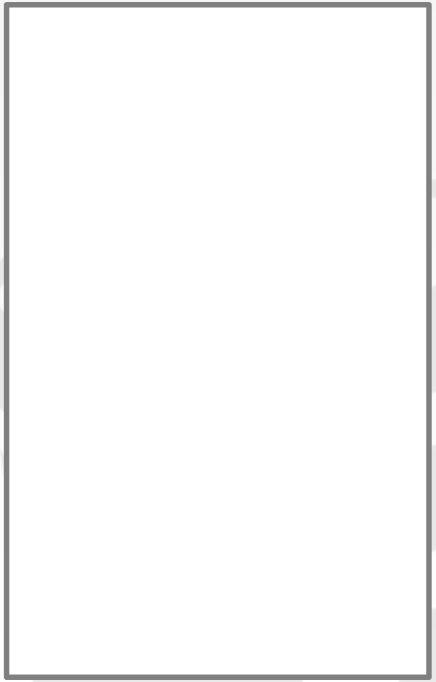


Access Frequency

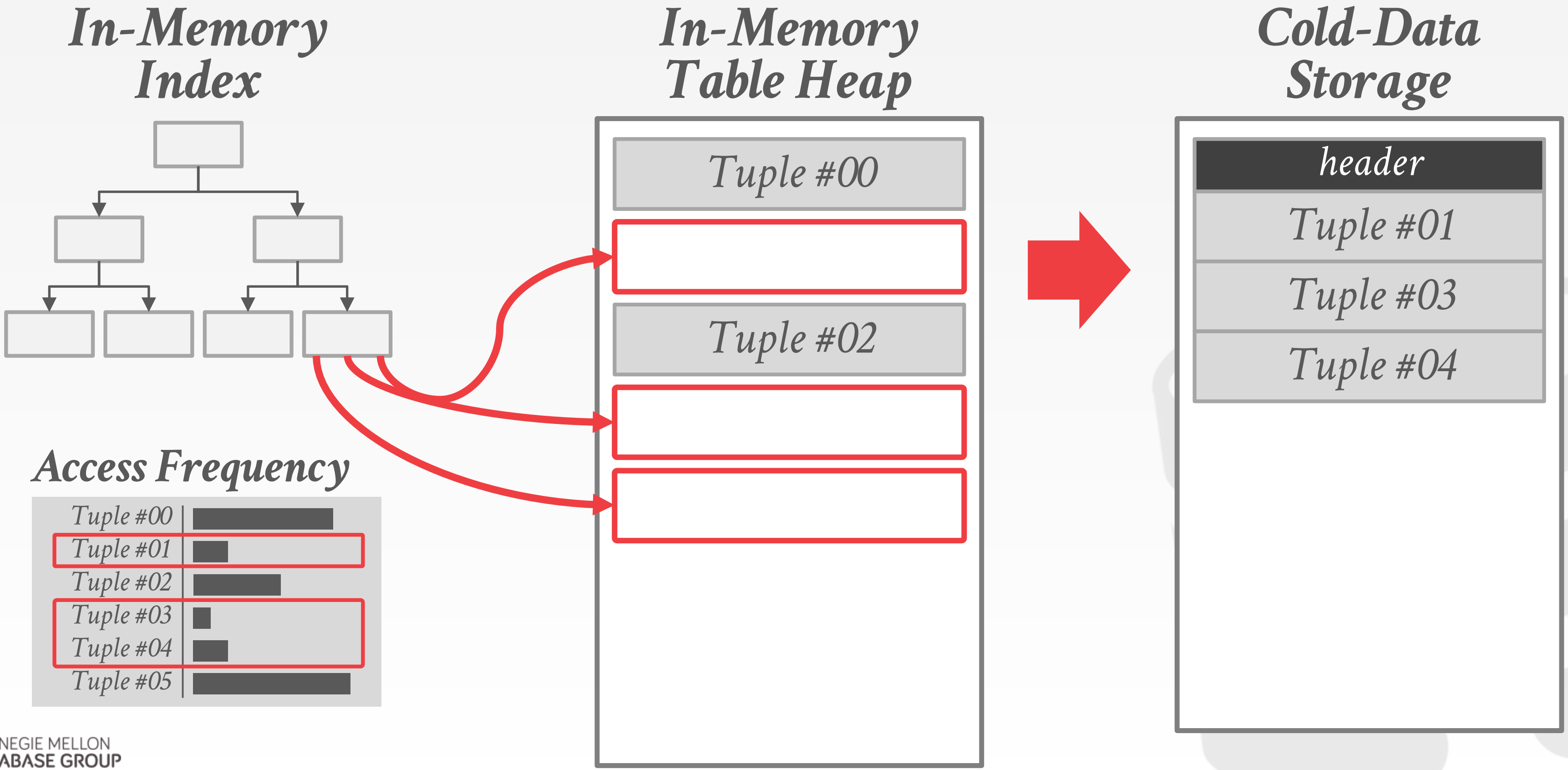
In-Memory Table Heap



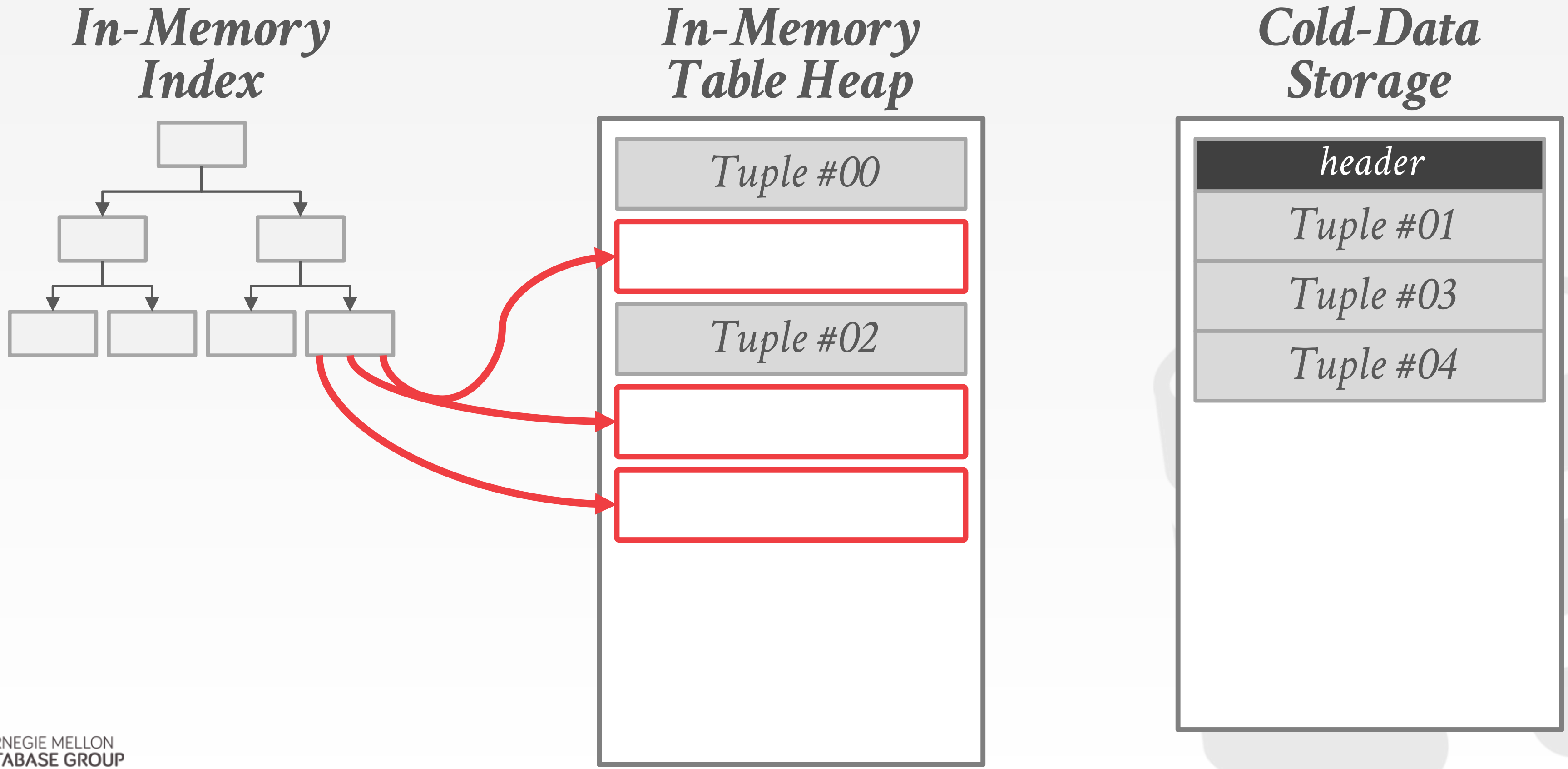
Cold-Data Storage



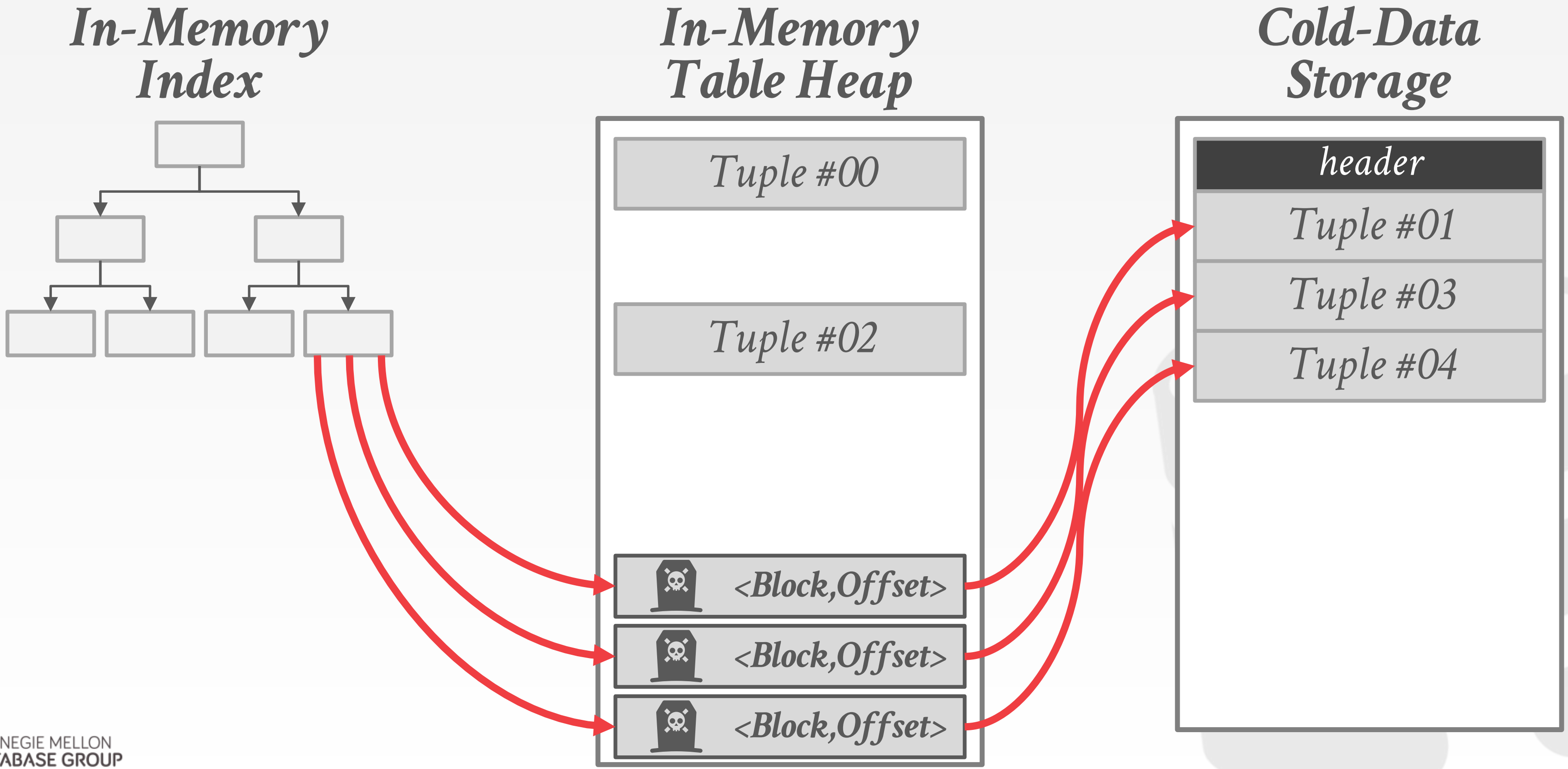
EVICTED TUPLE METADATA



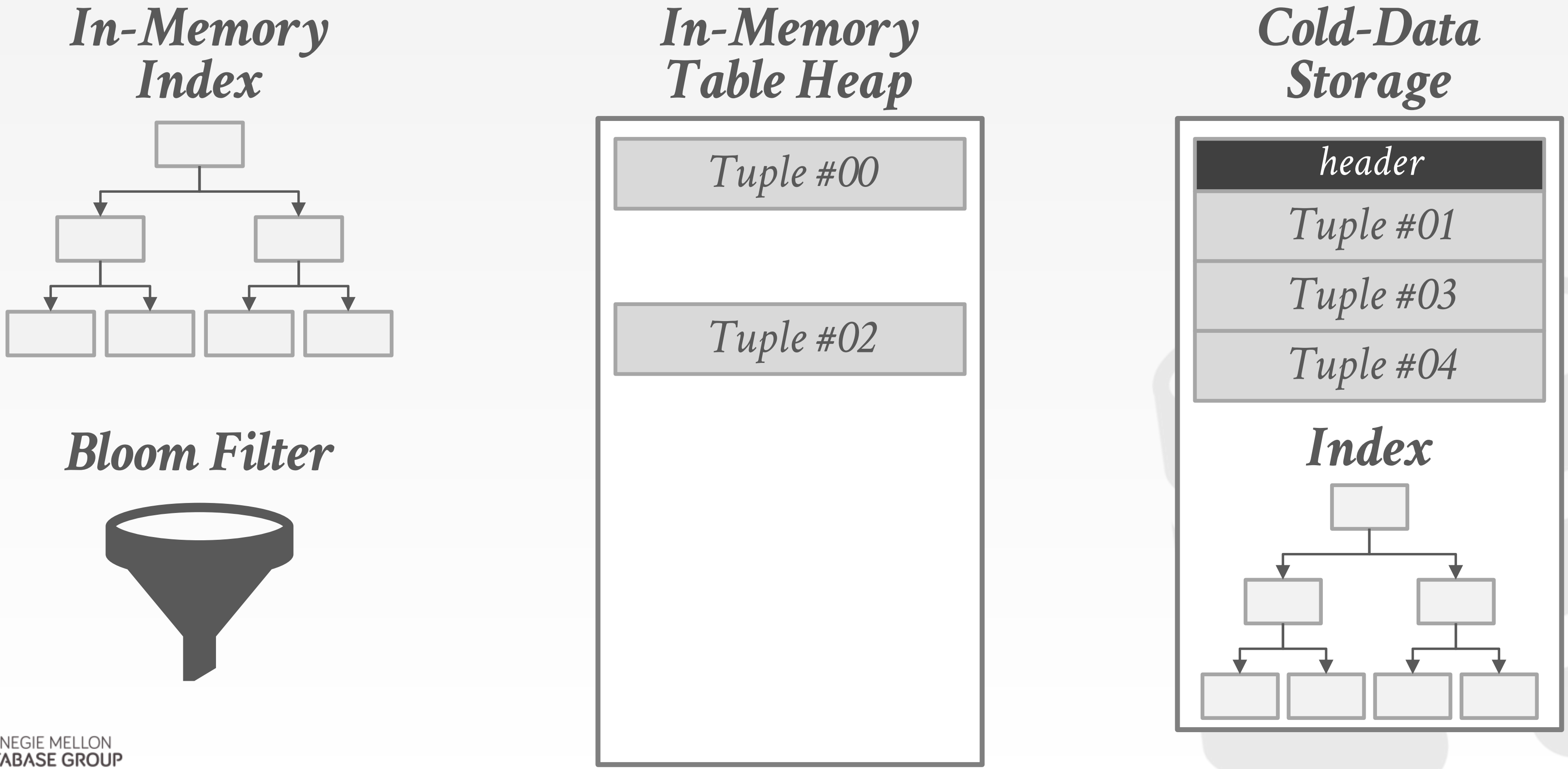
EVICTED TUPLE METADATA



EVICTED TUPLE METADATA



EVICTED TUPLE METADATA



In-Memory Index

Bloom Filter

In-Memory Table Heap

Tuple #00

Tuple #02

Cold-Data Storage

header

Tuple #01

Tuple #03

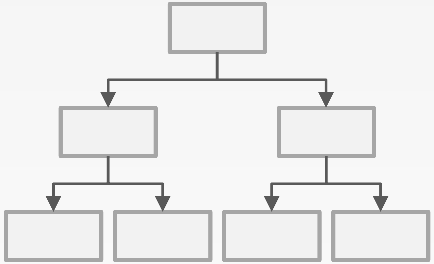
Tuple #04

Index

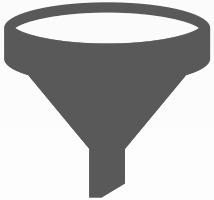
EVICTED TUPLE METADATA

Does 'x' exist?

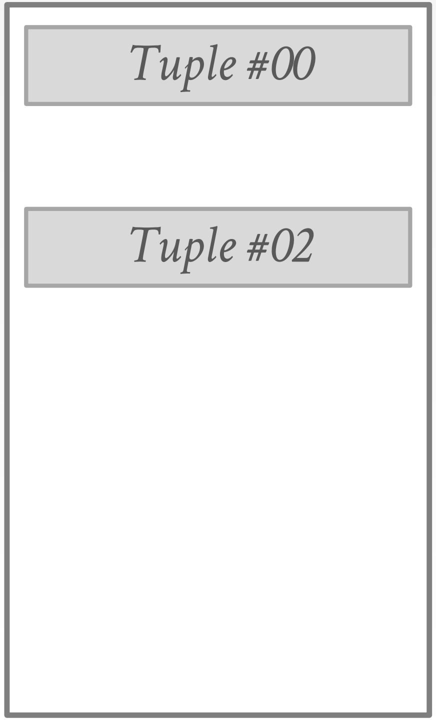
In-Memory Index



Bloom Filter



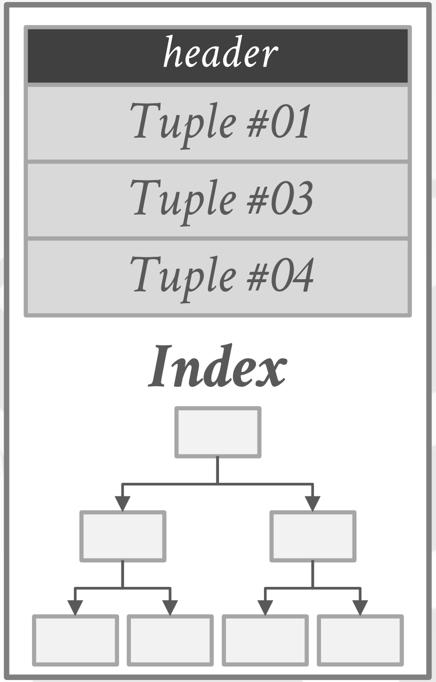
In-Memory Table Heap



Tuple #00

Tuple #02

Cold-Data Storage



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.
- → Cannot guarantee consistency for large queries.

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

Tuple-based

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

- LeanStore – Hierarchical 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

ANTI-CACHING: A NEW APPROACH TO DATABASE MANAGEMENT SYSTEM ARCHITECTURE
VLDB 2013

HEKATON – PROJECT SIBERIA

- Off-line Identification
- Administrator-defined Threshold
- Bloom Filters
- Synchronous Retrieval
- Tuple-level Granularity
- Always Merge

TREKKING THROUGH SIBERIA: MANAGING COLD DATA IN A MEMORY-OPTIMIZED DATABASE
VLDB 2014

EPFL VOLTDB

Off-line Identification

OS Virtual Memory

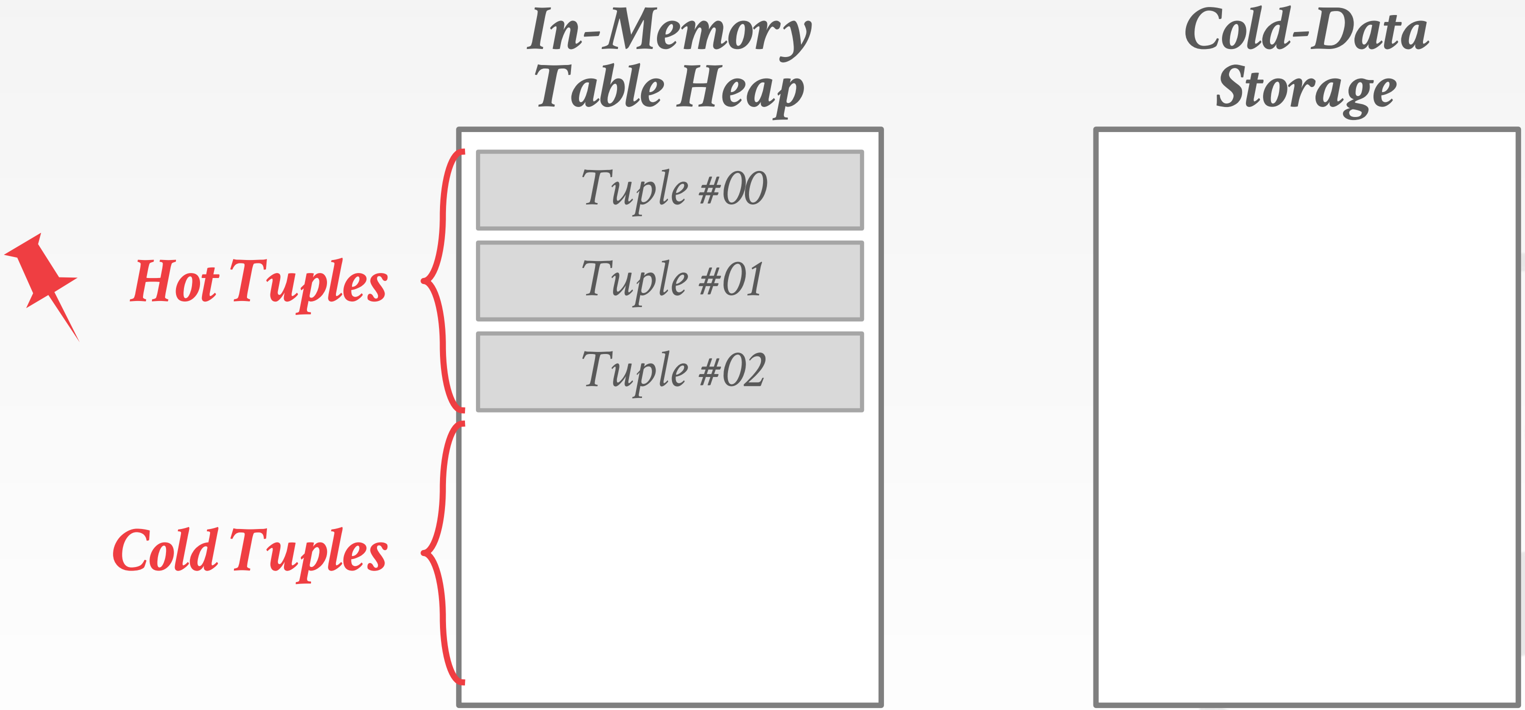
Synchronous Retrieval

Page-level Granularity

Always Merge

ENABLING EFFICIENT OS PAGING FOR MAIN-MEMORY OLTP DATABASES
DAMON 2013

EPFL VOLTDB



In-Memory Table Heap

Cold-Data Storage

Tuple #00

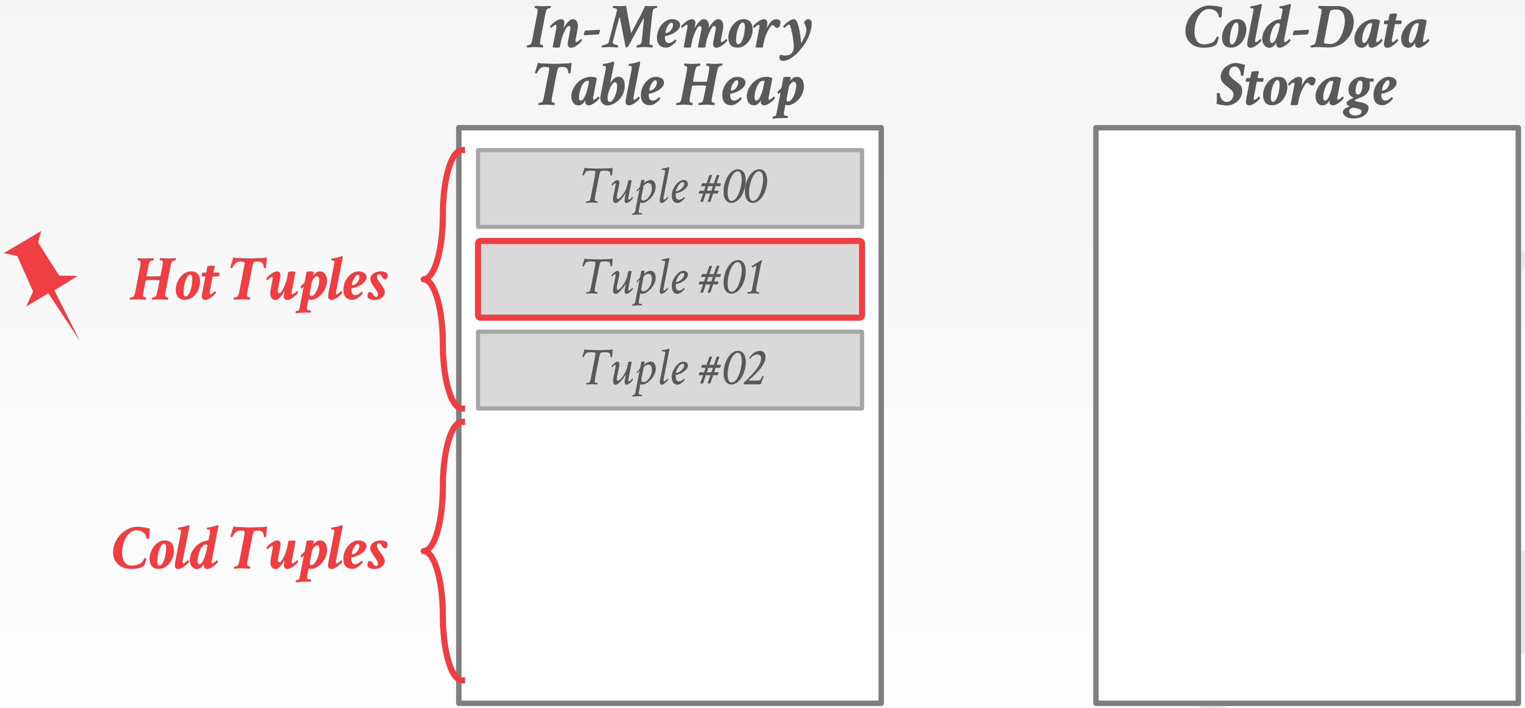
Tuple #01

Tuple #02

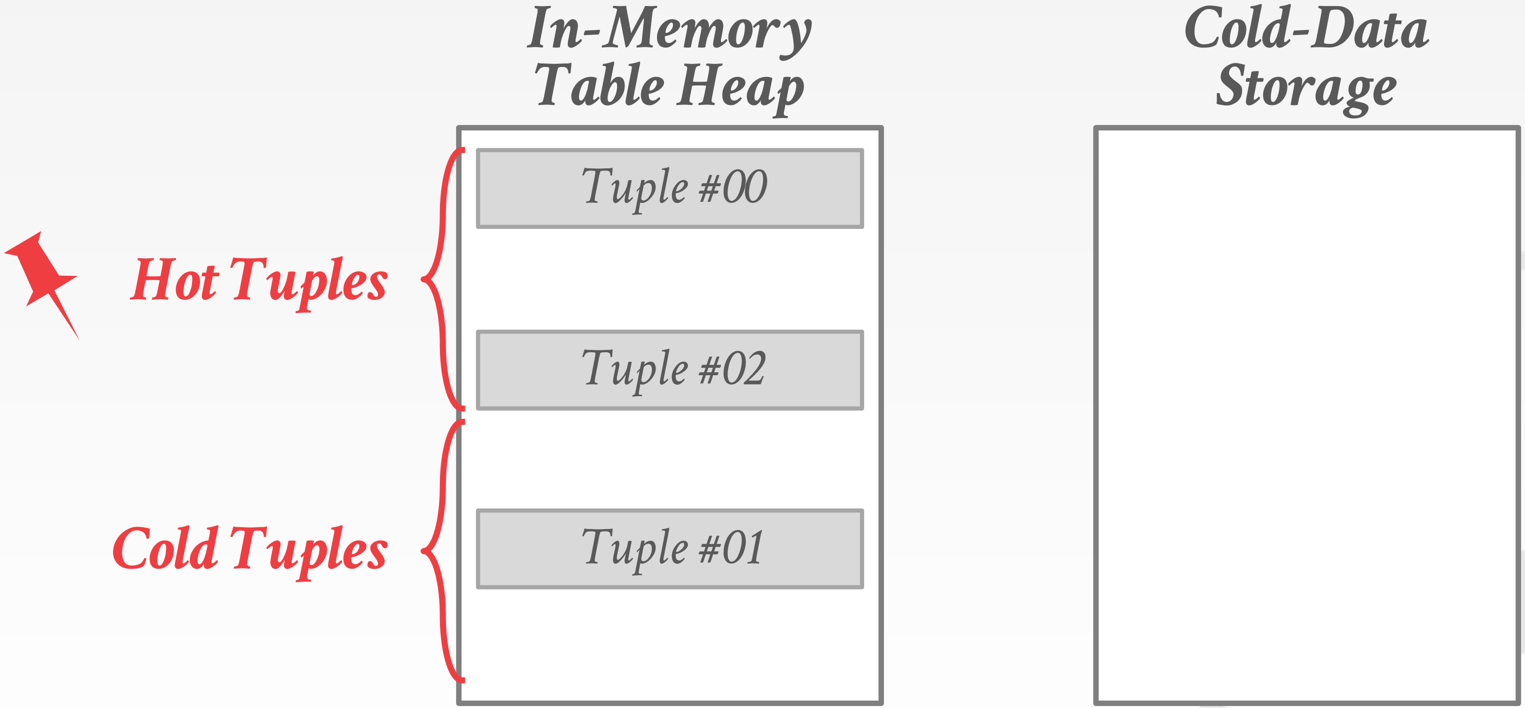
Hot Tuples

Cold Tuples

EPFL VOLTDB


In-Memory Table Heap
Cold-Data Storage
Hot Tuples
Cold Tuples
Tuple #00
Tuple #01
Tuple #02

EPFL VOLTDB



In-Memory Table Heap

Cold-Data Storage

Hot Tuples

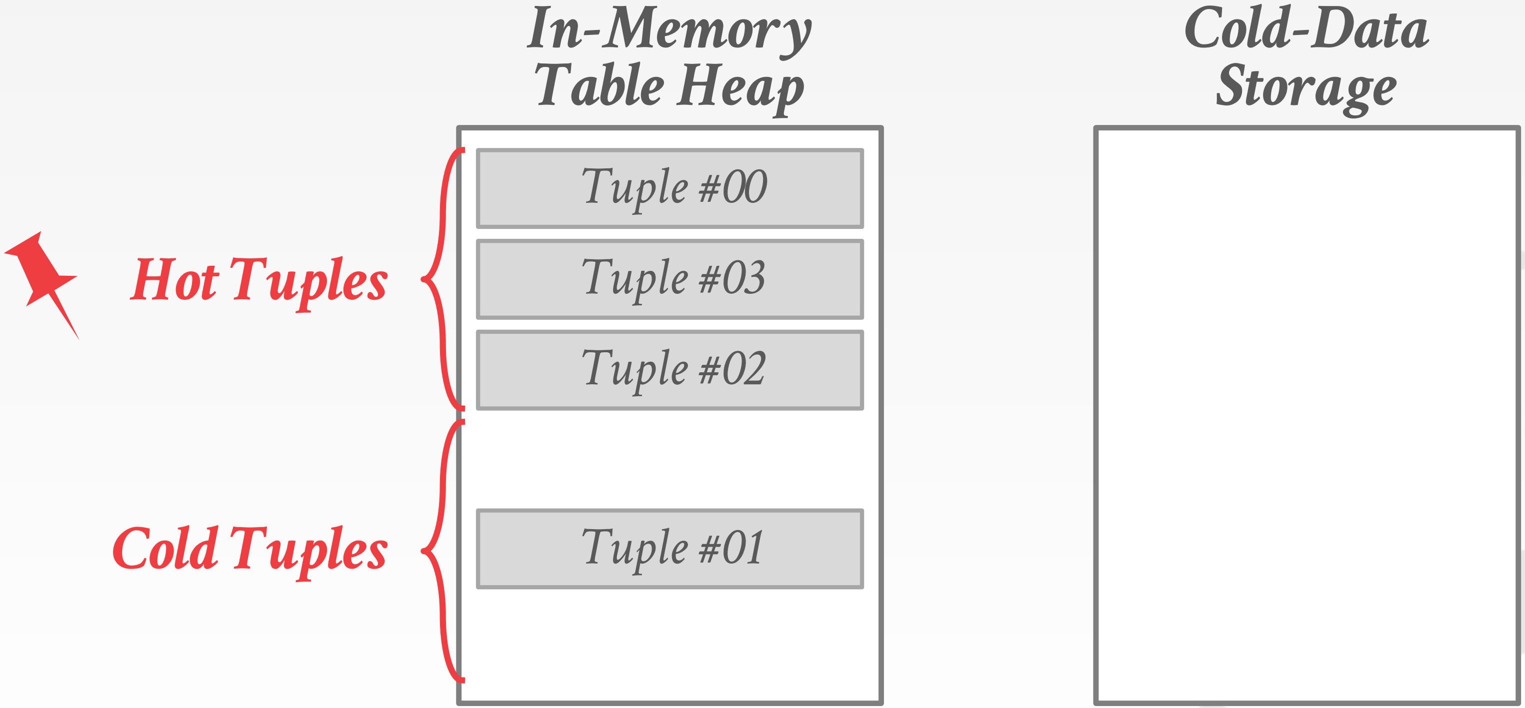
Tuple #00

Tuple #02

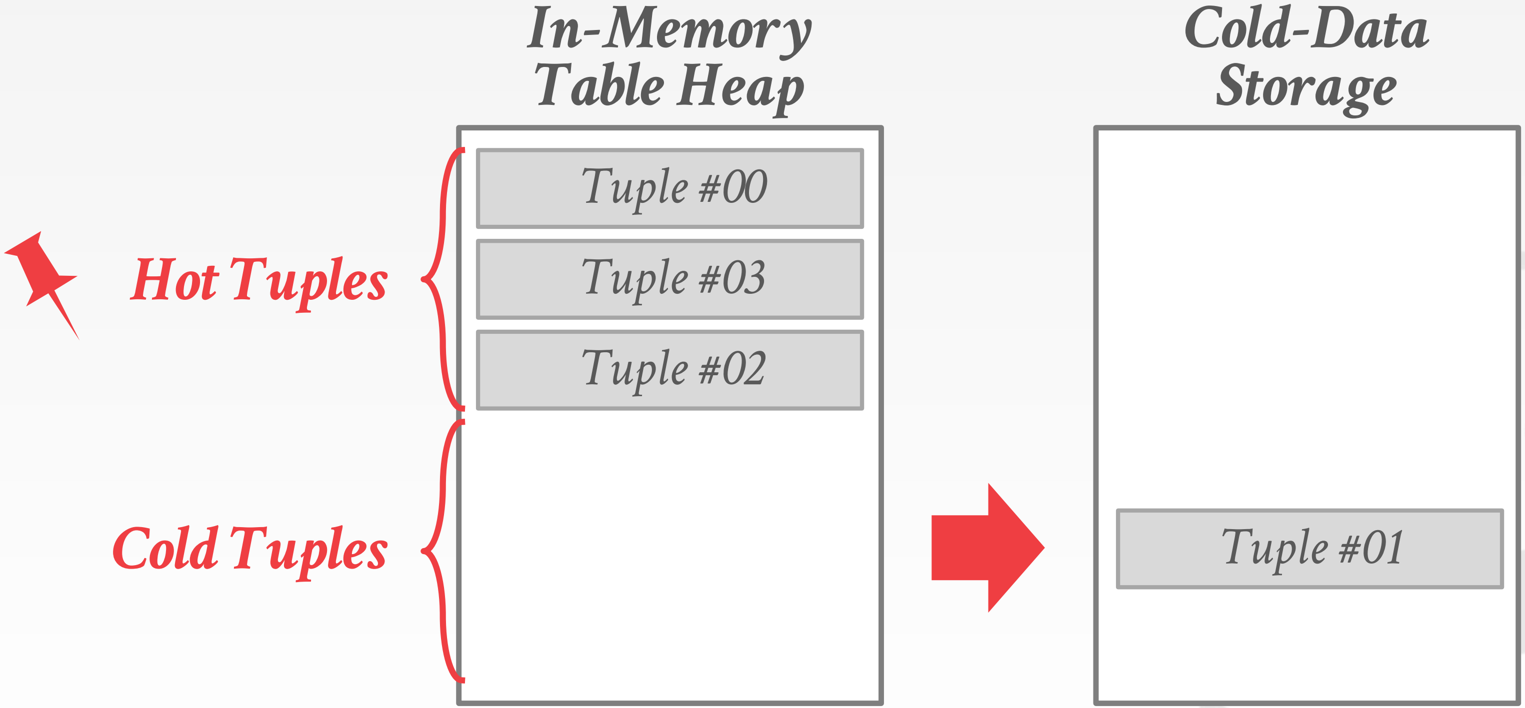
Cold Tuples

Tuple #01

EPFL VOLTDB


In-Memory Table Heap
Cold-Data Storage
Hot Tuples
Cold Tuples
Tuple #00
Tuple #03
Tuple #02
Tuple #01

EPFL VOLTDB



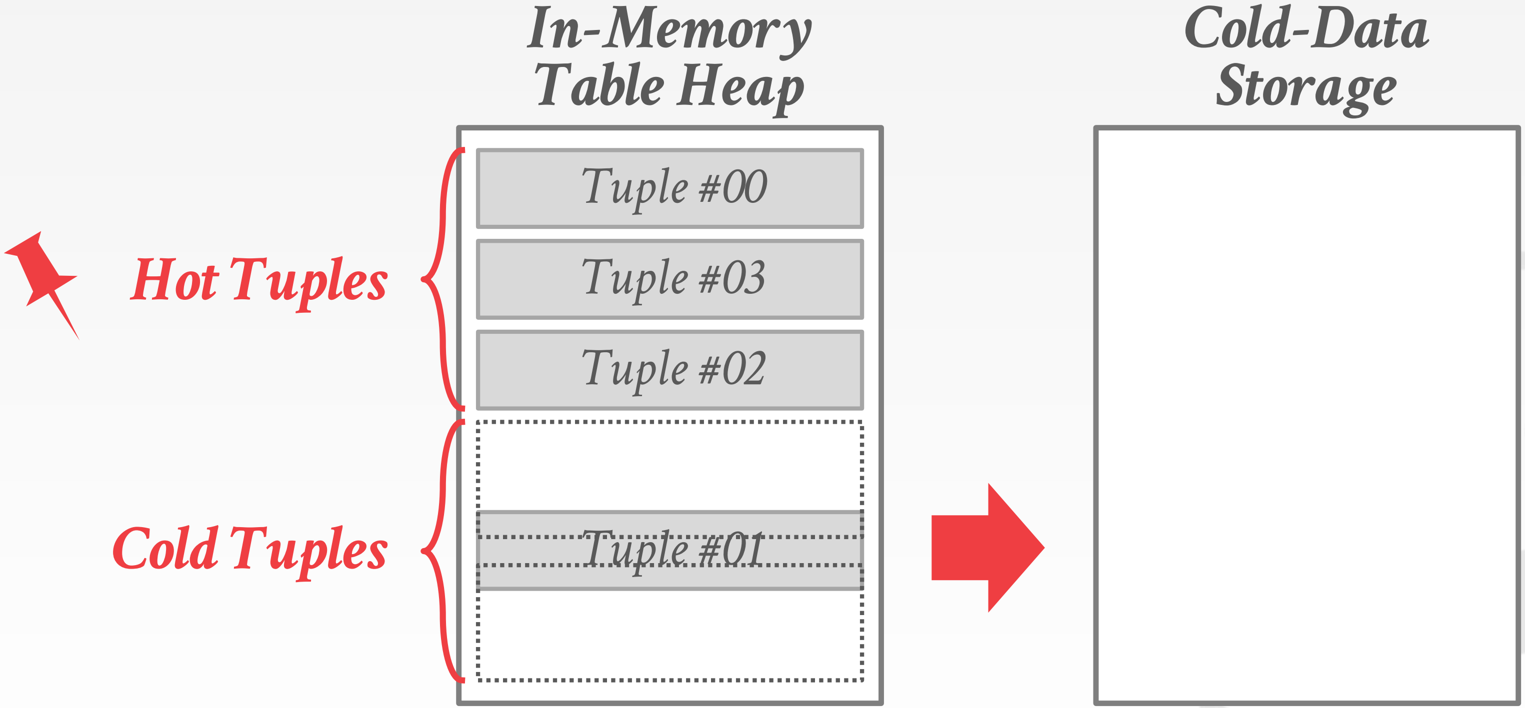
In-Memory Table Heap

Cold-Data Storage

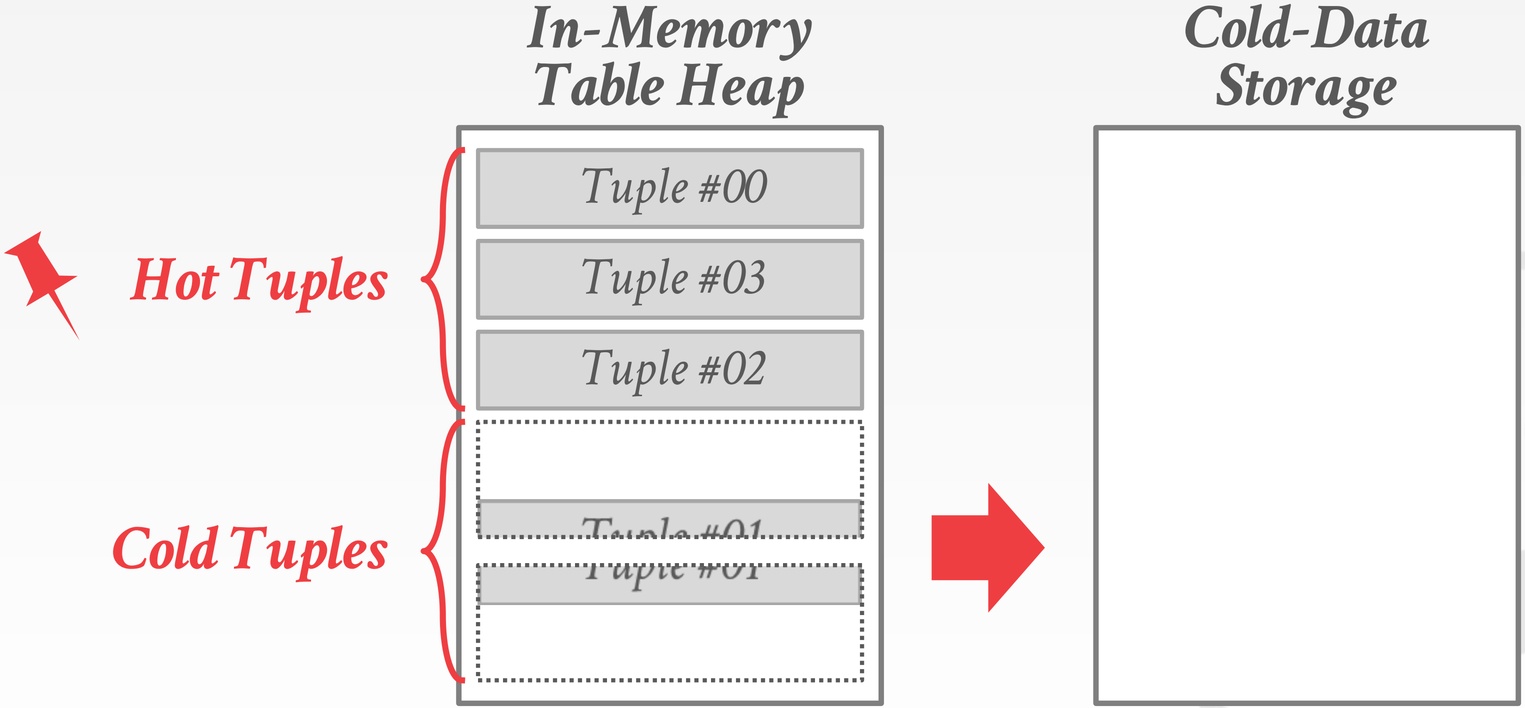
Hot Tuples

Cold Tuples

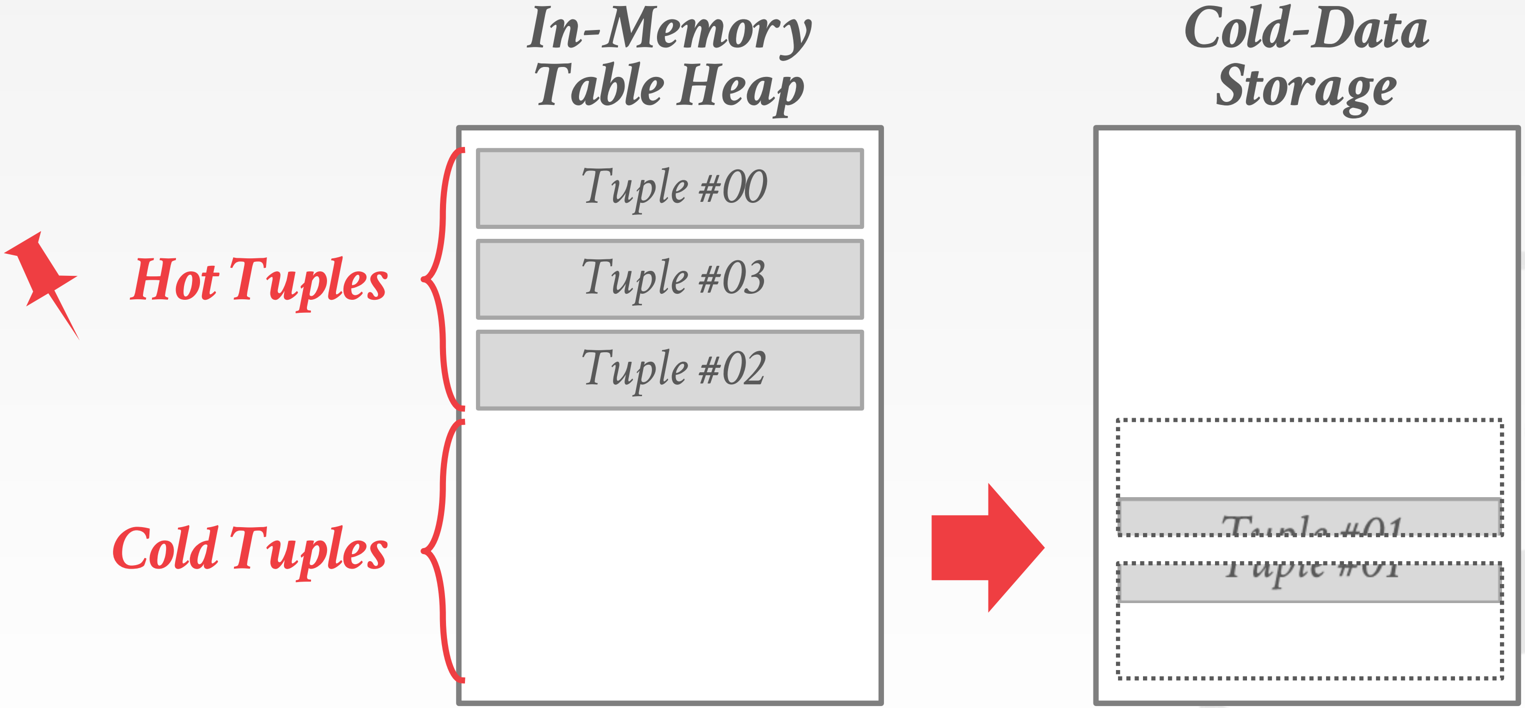
EPFL VOLTDB


In-Memory
Table Heap
Cold-Data
Storage
Hot Tuples
Cold Tuples
Tuple #00
Tuple #03
Tuple #02
Tuple #01

EPFL VOLTDB


In-Memory Table Heap
Cold-Data Storage
Hot Tuples
Cold Tuples
Tuple #00
Tuple #03
Tuple #02
Tuple #01
Tuple #01

EPFL VOLTDB


In-Memory Table Heap
Cold-Data Storage
Tuple #00
Tuple #03
Tuple #02
Hot Tuples
Cold Tuples
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

All of these approaches are based on tuples.

→ Have to 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.

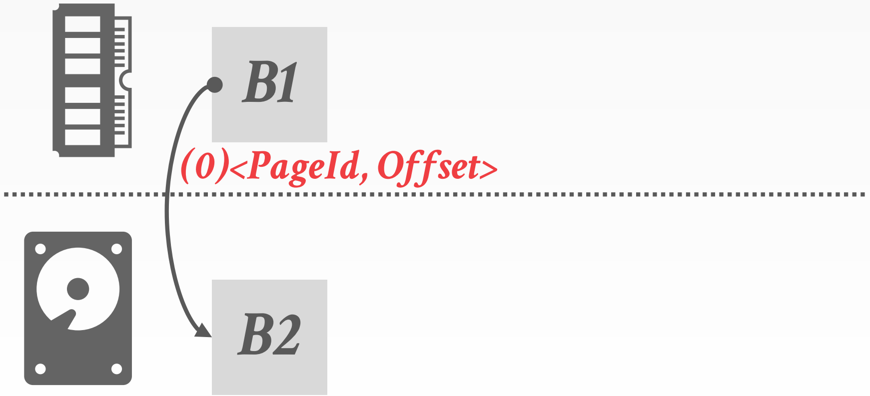
LEANSTORE: IN-MEMORY DATA MANAGEMENT BEYOND MAIN MEMORY
ICDE 2018

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.

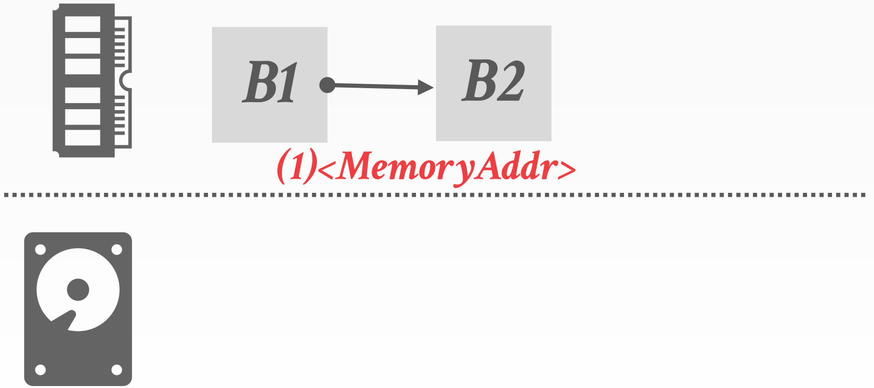


POINTER SWIZZLING

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.



REPLACEMENT STRATEGY

Randomly select blocks for eviction.

- → Don't have to update meta-data every time a txn accesses a hot tuple.

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.

BLOCK HIERARCHY

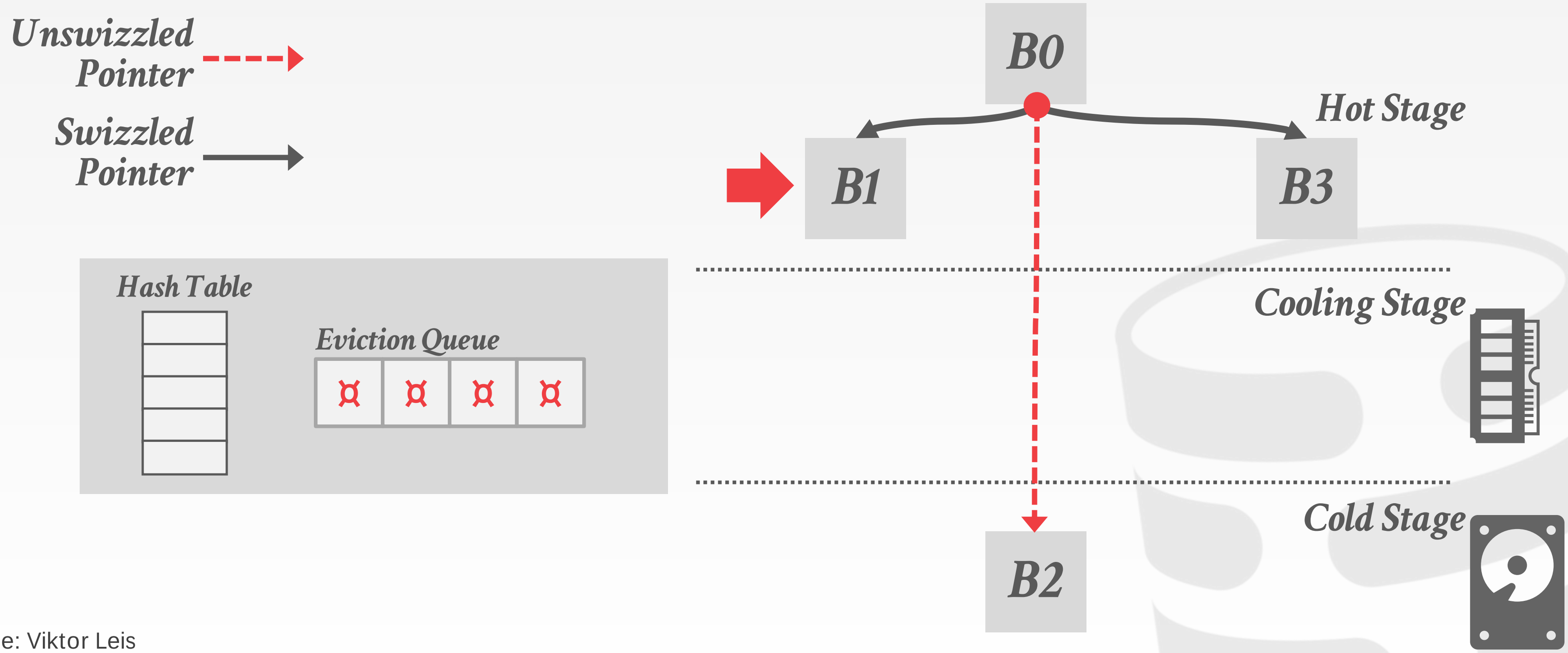
Blocks are organized in a tree hierarchy.

- → Each page has only one parent, which means that there is only a single pointer.

The DBMS can only evict a page if its children are also evicted.

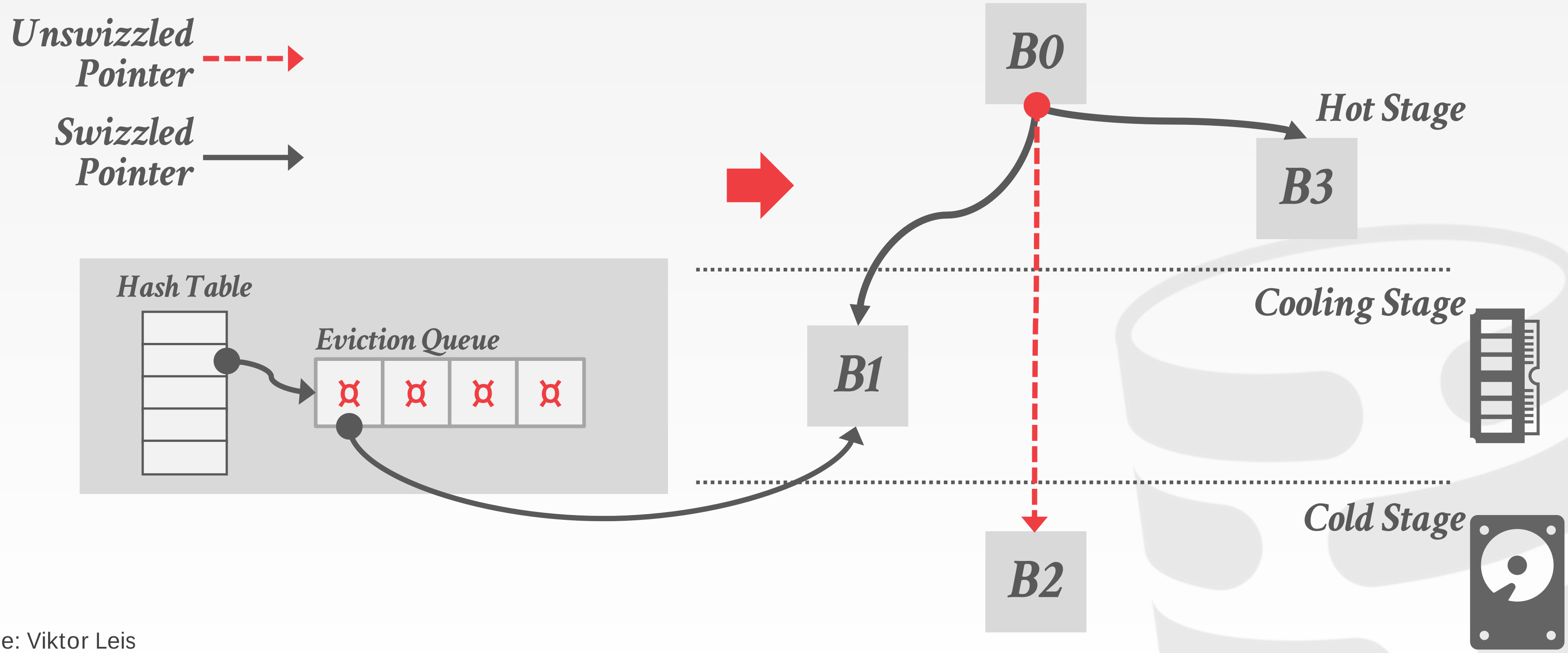
- → This avoids the problem of evicting pages that contain swizzled pointers.
- → If a page is selected but it has in-memory children, then it automatically switches to select one of its children.

BLOCK HIERARCHY



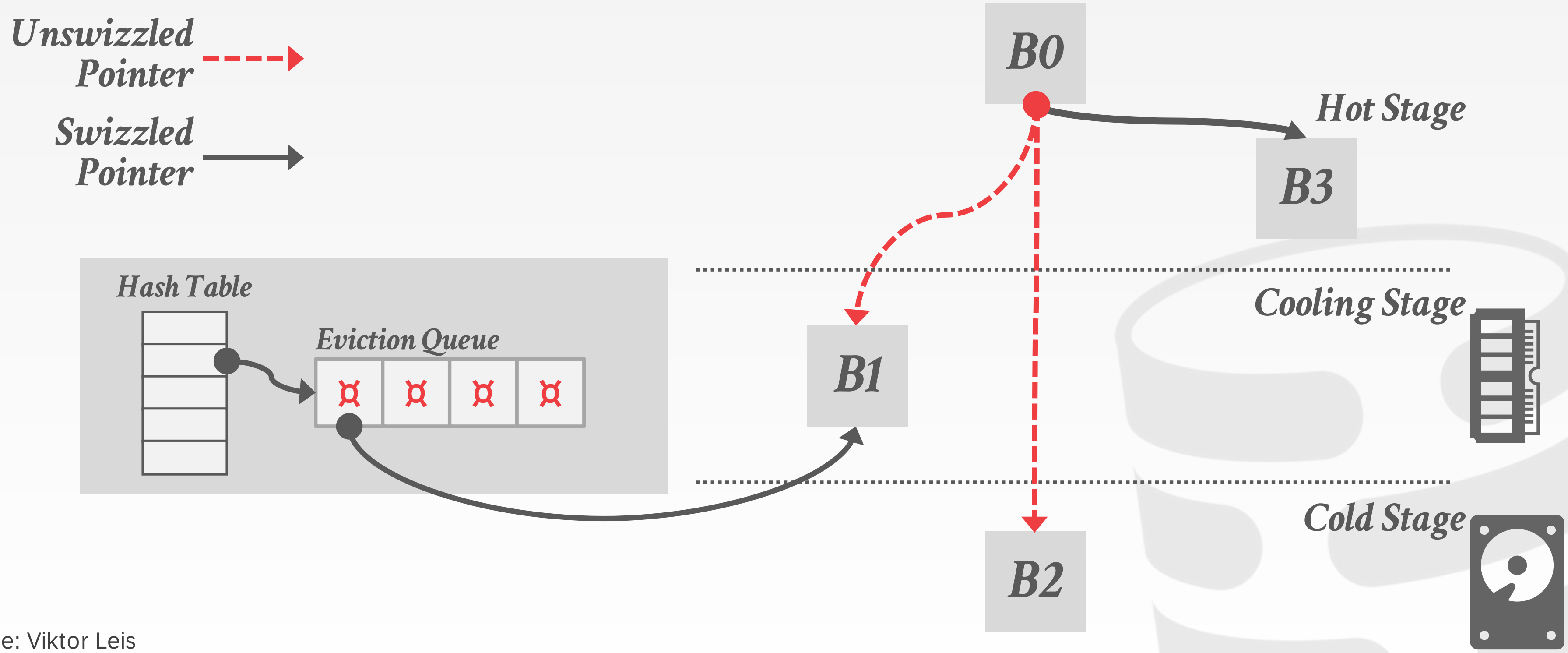
Source: Viktor Leis

BLOCK HIERARCHY



Source: Viktor Leis

BLOCK HIERARCHY



Source: Viktor Leis

MEMSQL – COLUMNAR TABLES

Administrator manually declares a table as a distinct disk-resident columnar table.

→ Appears as a separate logical table to the application.
→ Uses `mmap` to manage buffer pool.
→ Pre-computed aggregates per block always in memory.

Manual Identification

No Evicted Metadata is needed.

Synchronous Retrieval

Always Merge

Source: MemSQL

PARTING THOUGHTS

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

None of these techniques handle index memory.

Fast & cheap byte-addressable NVM will make this lecture unnecessary.

NEXT CLASS

Logging + Checkpoints!