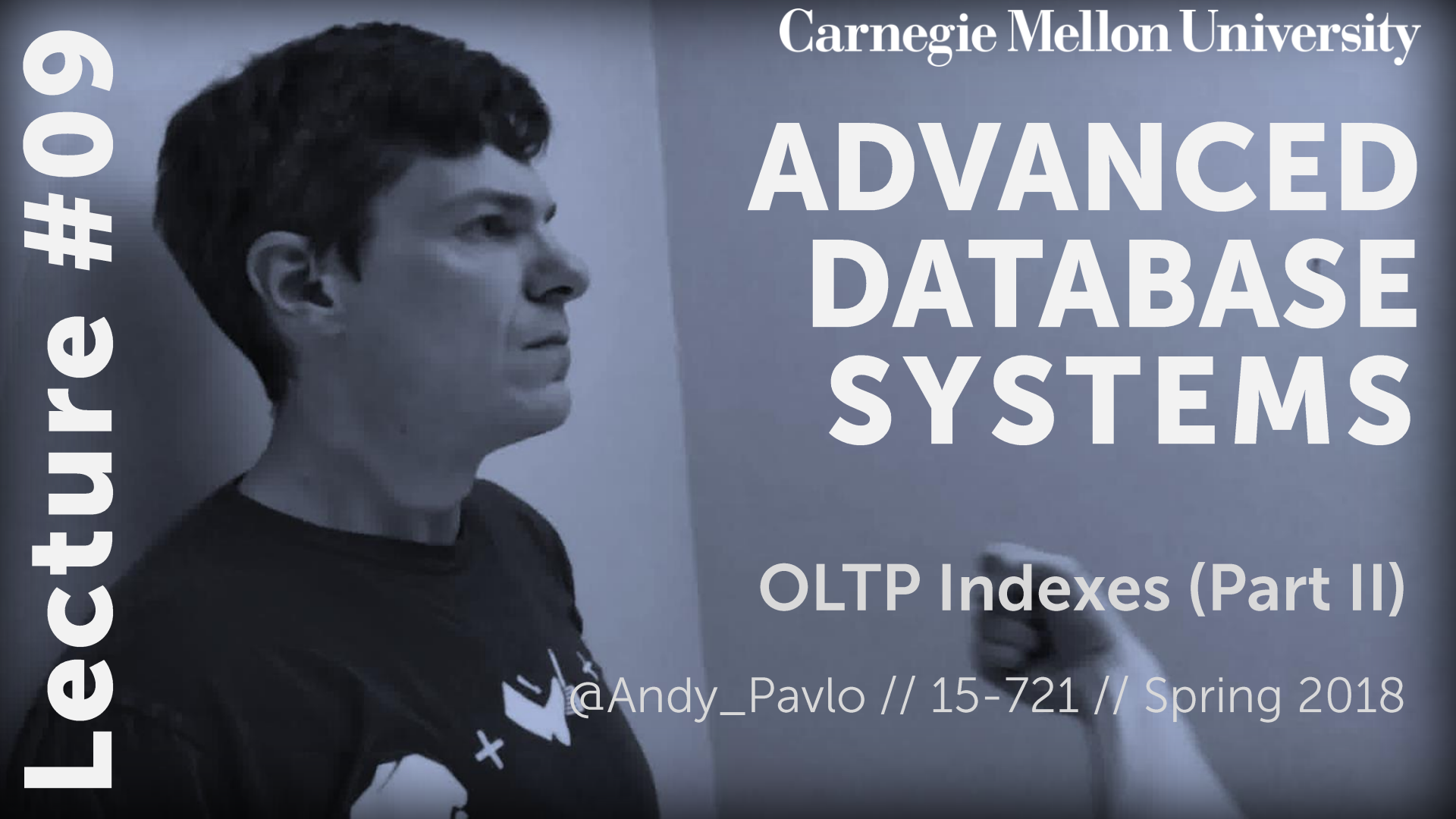

Lecture #09
Carnegie Mellon University
ADVANCED DATABASE SYSTEMS
OLTP Indexes (Part II)
@Andy_Pavlo // 15-721 // Spring 2018

TODAY'S AGENDA

Index Implementation Issues
ART Index
Profiling in Peloton

INDEX IMPLEMENTATION ISSUES

Memory Pools

Garbage Collection

Non-Unique Keys

Variable-length Keys

Prefix Compression

MEMORY POOLS

We don't want to be calling `malloc` and `free` anytime we need to add or delete a node.

If all the nodes are the same size, then the index can maintain a pool of available nodes.

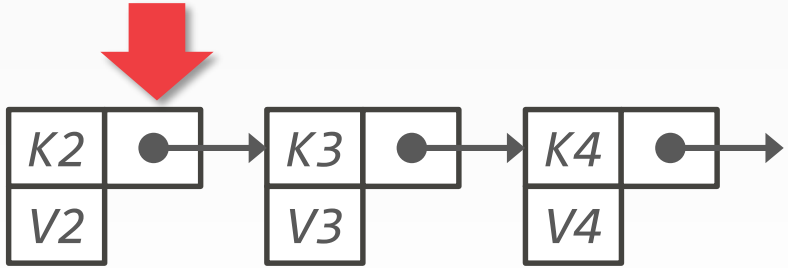
→ **Insert**: Grab a free node, otherwise create a new one.
→ **Delete:** Add the node back to the free pool.

Need some policy to decide when to retract the pool size.

GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

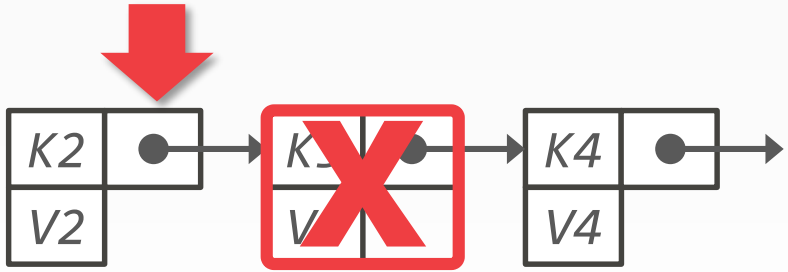
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

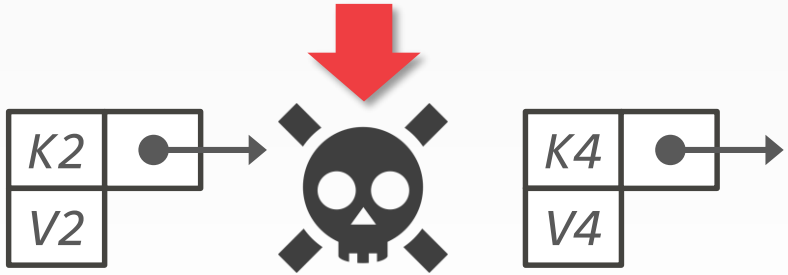
We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…

GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



REFERENCE COUNTING

Maintain a counter for each node to keep track of the number of threads that are accessing it.

→ Increment the counter before accessing.
→ Decrement it when finished.
→ A node is only safe to delete when the count is zero.

This has bad performance for multi-core CPUs

→ Incrementing/decrementing counters causes a lot of cache coherence traffic.

OBSERVATION

We don't actually care about the actual value of the reference counter. We only need to know when it reaches zero.

We don't have to perform garbage collection immediately when the counter reaches zero.

EPOCH GARBAGE COLLECTION

Maintain a global epoch counter that is periodically updated (e.g., every 10 ms).

→ Keep track of what threads enter the index during an epoch and when they leave.

Mark the current epoch of a node when it is marked for deletion.

→ The node can be reclaimed once all threads have left that epoch (and all preceding epochs).

Also known as ***Read-Copy-Update*** (RCU) in Linux.

NON-UNIQUE INDEXES

Approach #1: Duplicate Keys

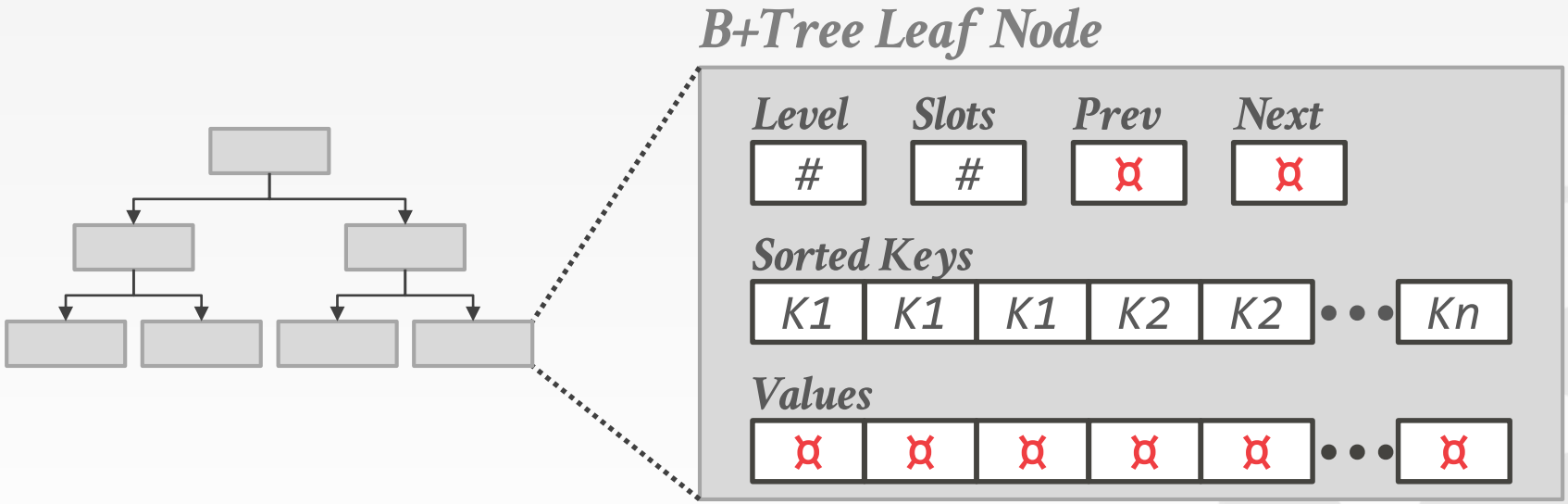
→ Use the same node layout but store duplicate keys multiple times.

Approach #2: Value Lists

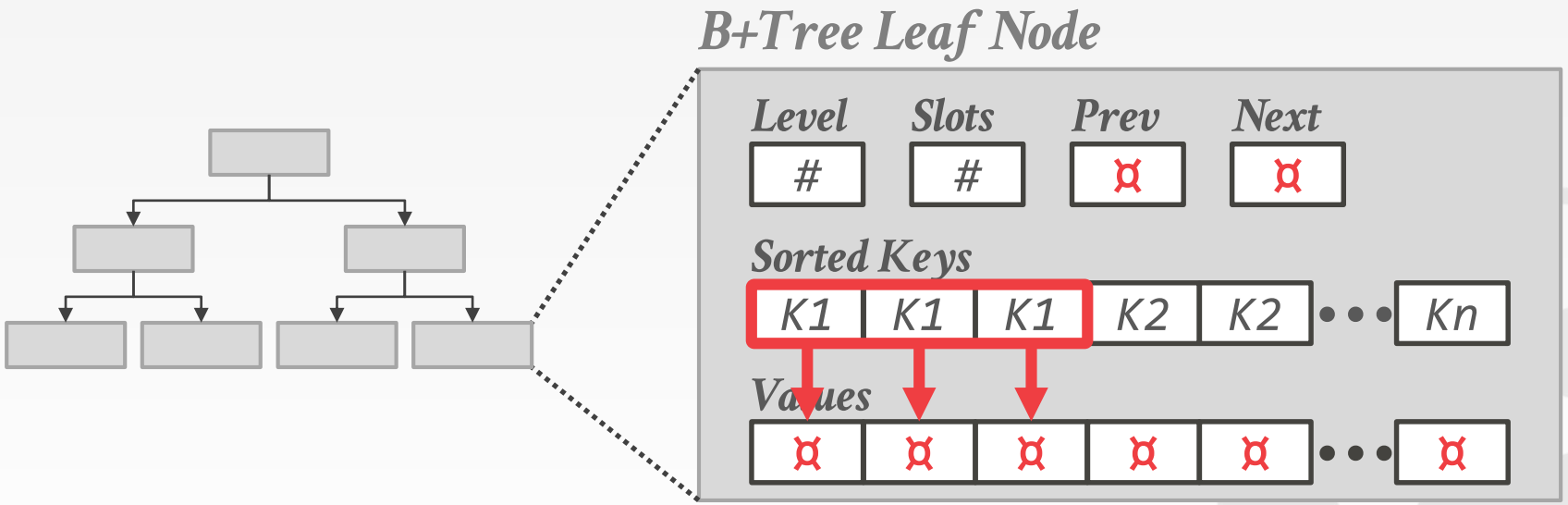
→ Store each key only once and maintain a linked list of unique values.

MODERN B-TREE TECHNIQUES
NOW PUBLISHERS 2010

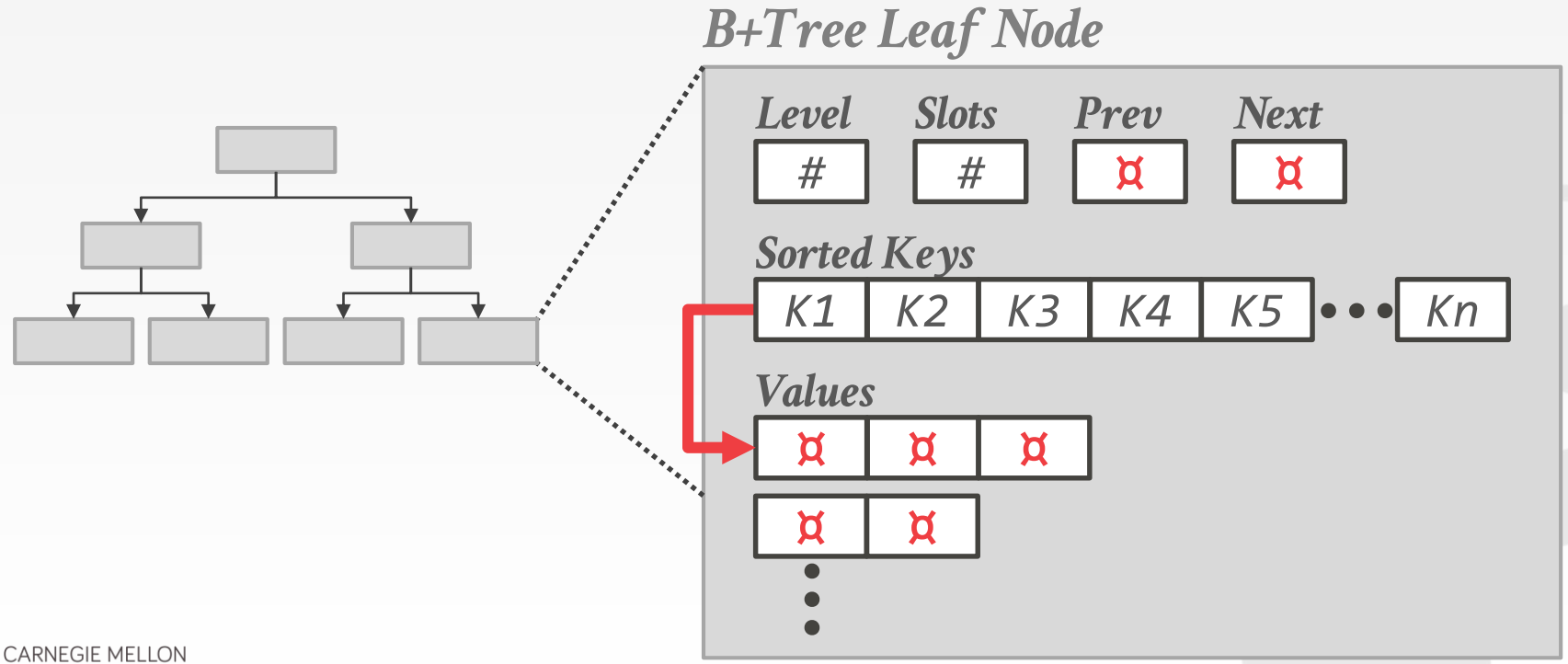
NON-UNIQUE: DUPLICATE KEYS



NON-UNIQUE: DUPLICATE KEYS



NON-UNIQUE: VALUE LISTS



VARIABLE LENGTH KEYS

Approach #1: Pointers

→ Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes

→ The size of each node in the index can vary.
→ Requires careful memory management.

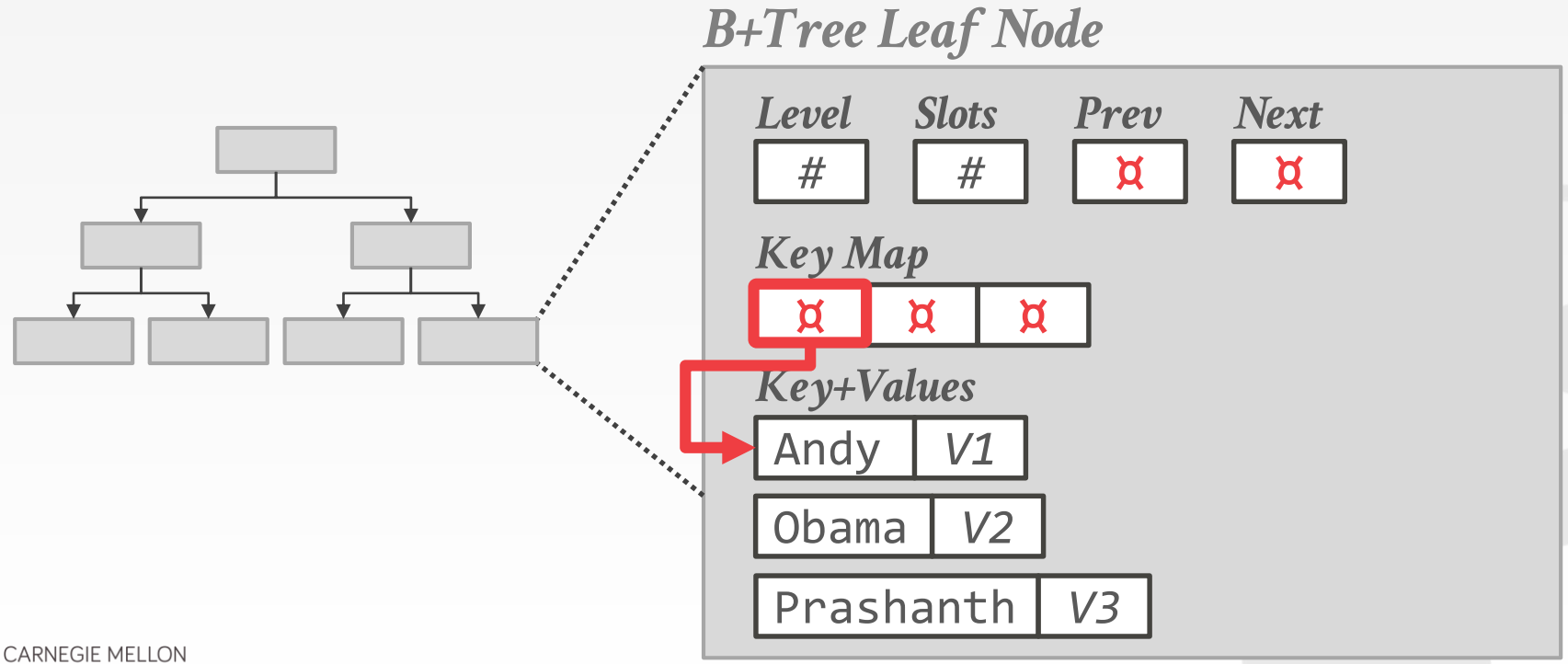
Approach #3: Padding

→ Always pad the key to be max length of the key type.

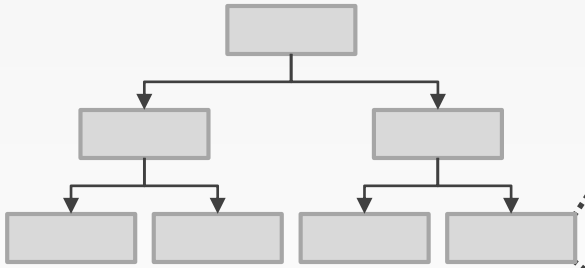
Approach #4: Key Map / Indirection

→ Embed an array of pointers that map to the key + value list within the node.

KEY MAP / INDIRECTION



KEY MAP / INDIRECTION



B+Tree Leaf Node

Level | *Slots* | *Prev* | *Next*

| # | ¤ | ¤

Key Map

¤ | ¤ | ¤

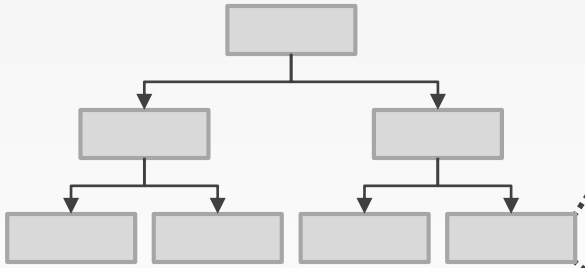
Key+Values

Andy | *V1*

Obama | *V2*

Prashanth | *V3*

KEY MAP / INDIRECTION



B+Tree Leaf Node

Level	*Slots*	*Prev*	*Next*
#	#	¤	¤

Key Map

A • ¤	O • ¤	P • ¤

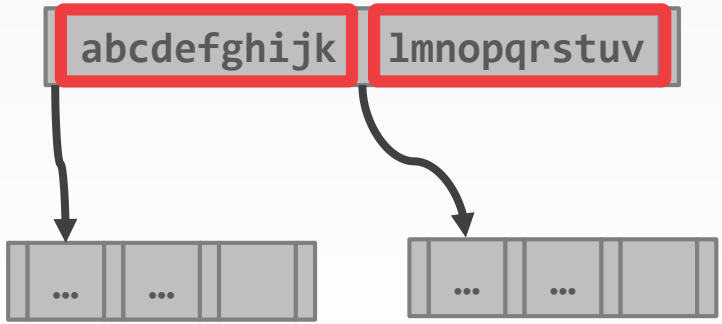
Key+Values

Andy	*V1*
Obama	*V2*
Prashanth	*V3*

PREFIX COMPRESSION

Store a minimum prefix that is needed to correctly route probes into the index.

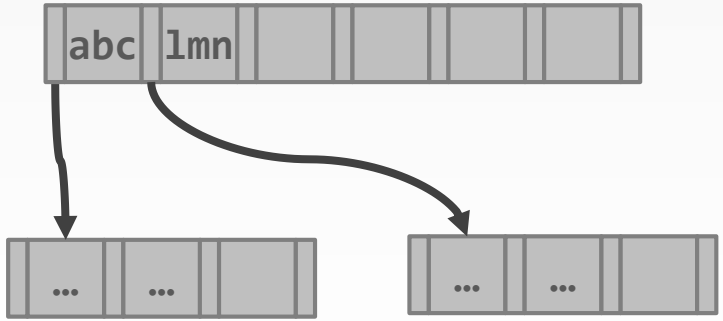
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

Store a minimum prefix that is needed to correctly route probes into the index.

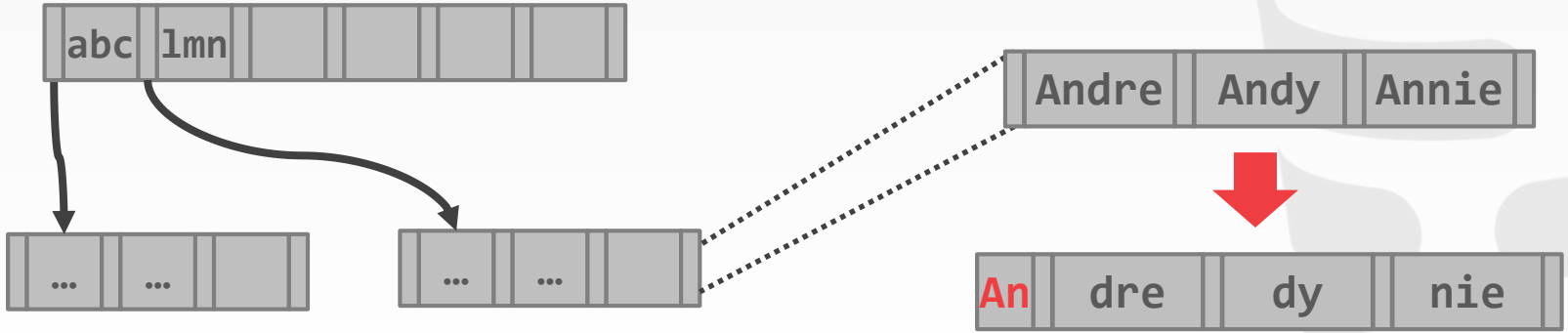
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

Store a minimum prefix that is needed to correctly route probes into the index.

Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



ADAPATIVE RADIX TREE (ART)

Uses digital representation of keys to examine prefixes 1-by-1 instead of comparing entire key.

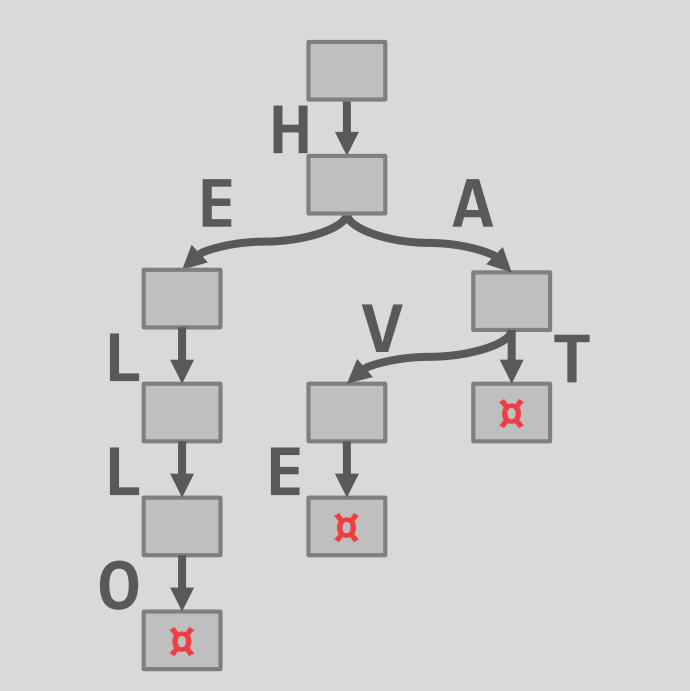
Radix trees properties:

→ The height of the tree depends on the length of keys.
→ Does not require rebalancing
→ The path to a leaf node represents the key of the leaf
→ Keys are stored implicitly and can be reconstructed from paths.

THE ADAPTIVE RADIX TREE: ARTFUL INDEXING FOR MAIN-MEMORY DATABASES
ICDE 2013

TRIE VS. RADIX TREE

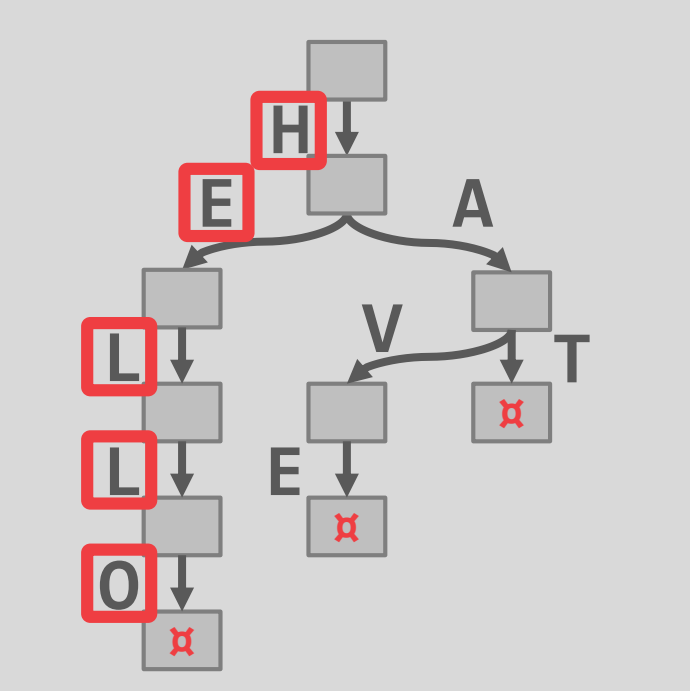
Trie



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

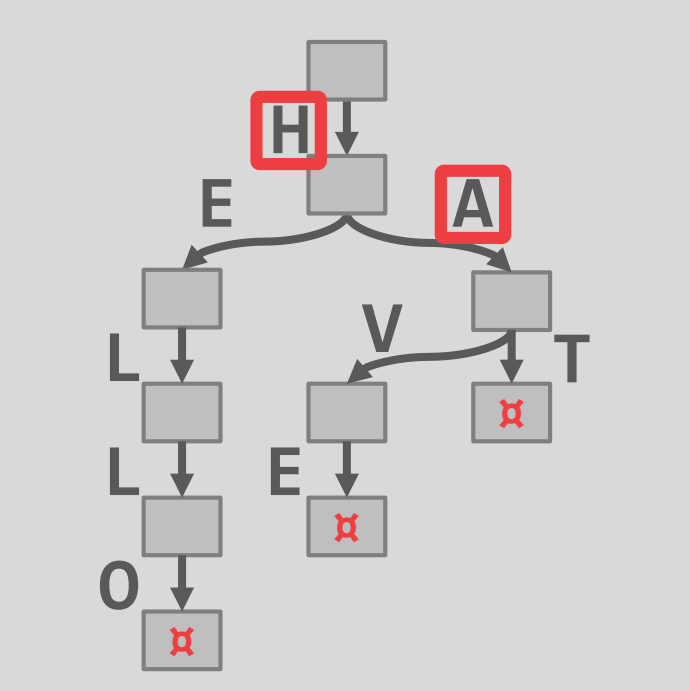
Trie



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

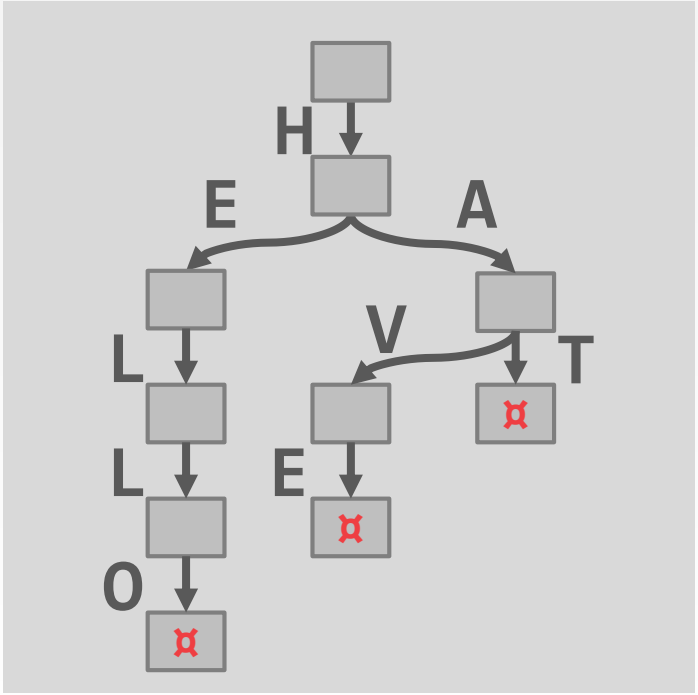
Trie



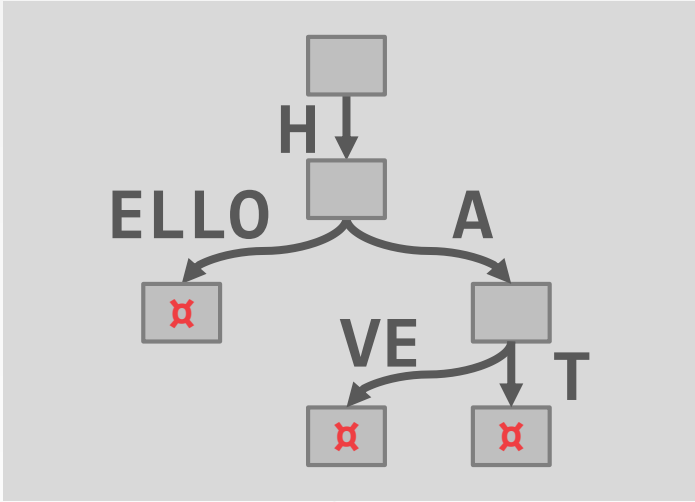
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie



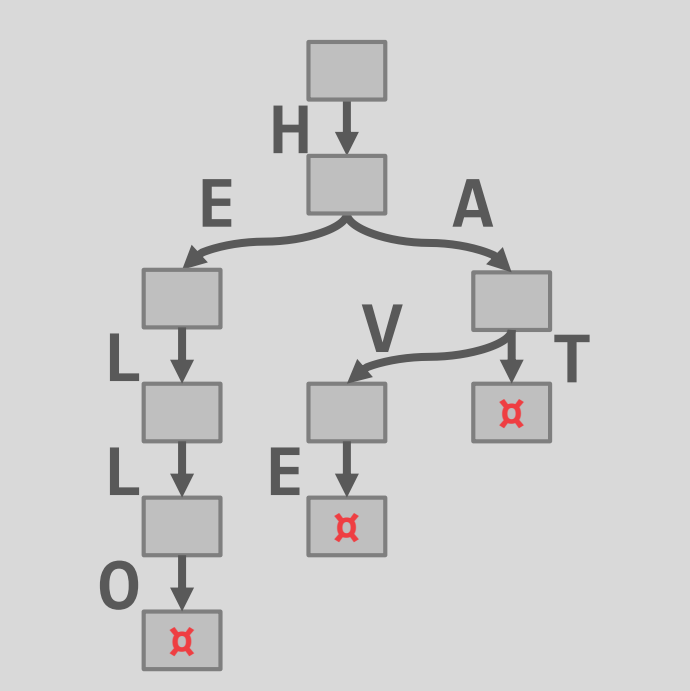
Radix Tree



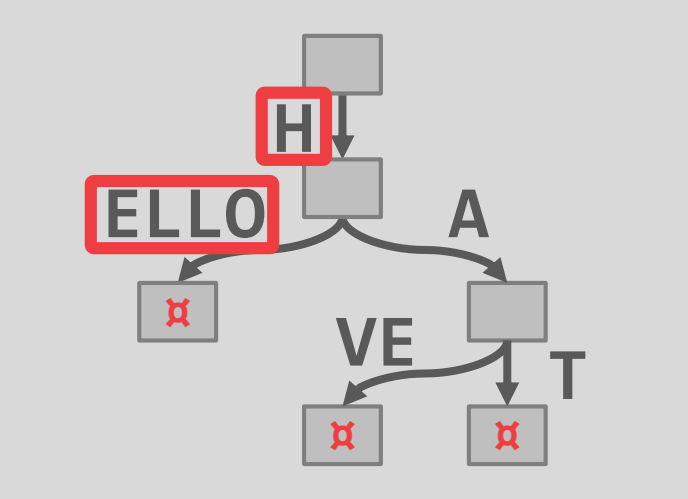
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie



Radix Tree



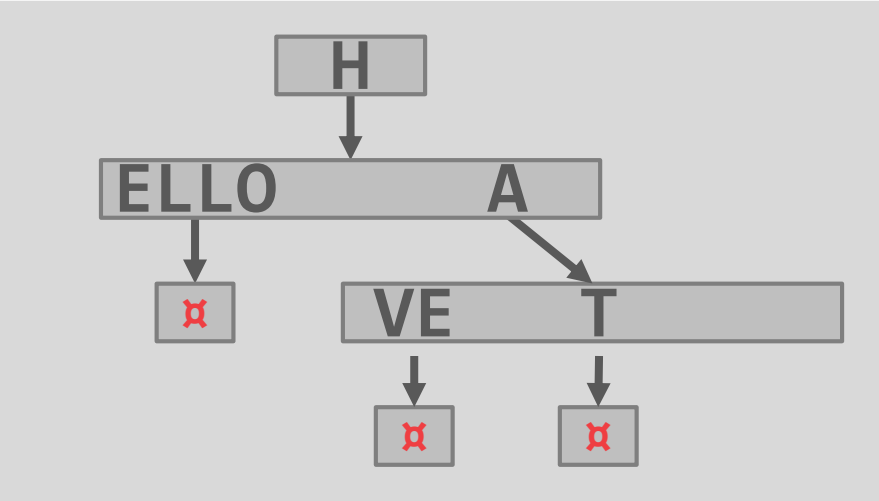
Keys: HELLO, HAT, HAVE

ART: ADAPTIVELY SIZED NODES

The index supports four different internal node types with different capacities.

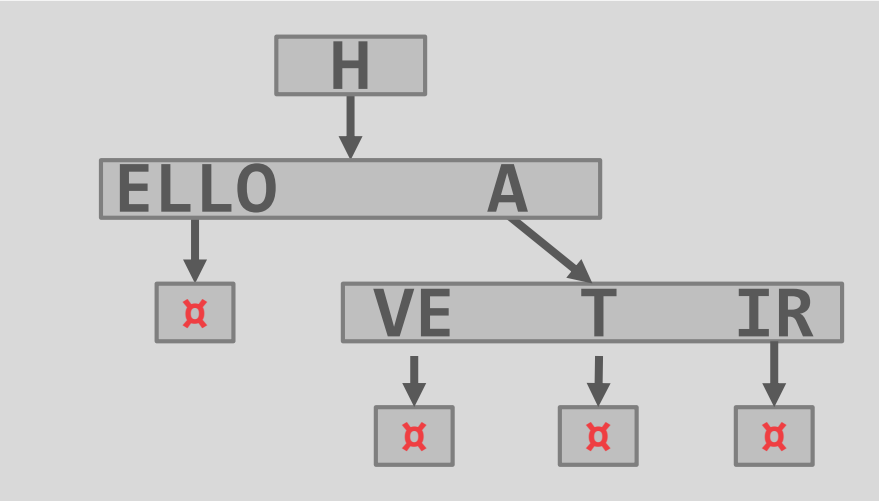
Pack in multiple digits into a single node to improve cache locality.

ART: MODIFICATIONS



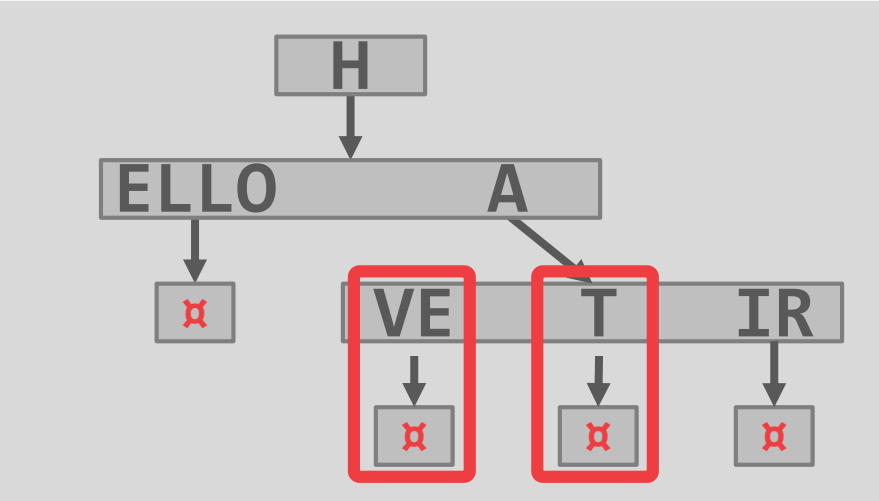
Operation: Insert **HAIR**

ART: MODIFICATIONS



Operation: Insert **HAIR**

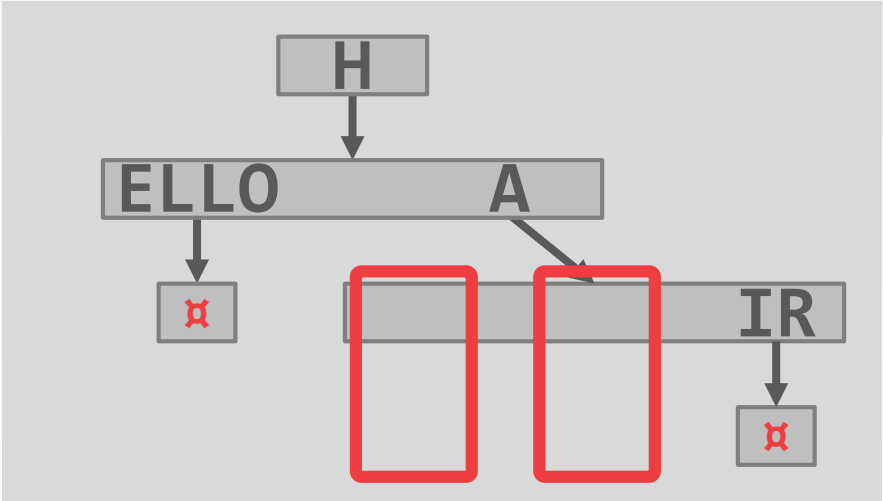
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

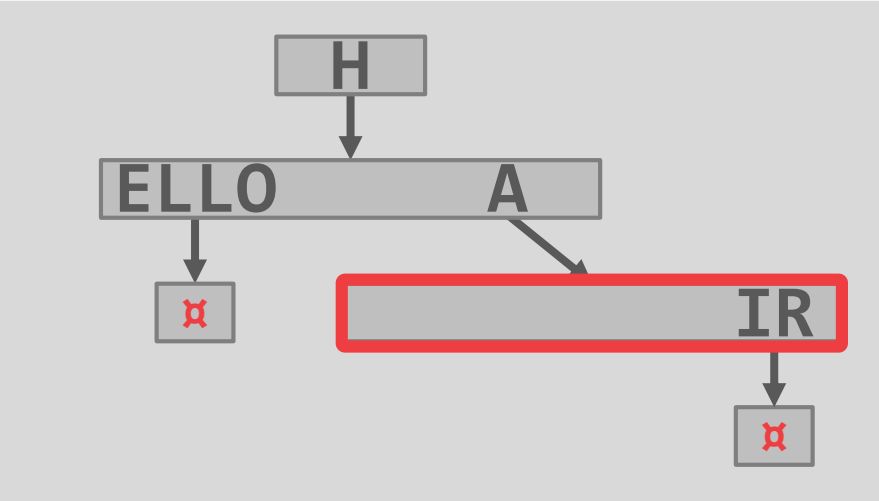
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

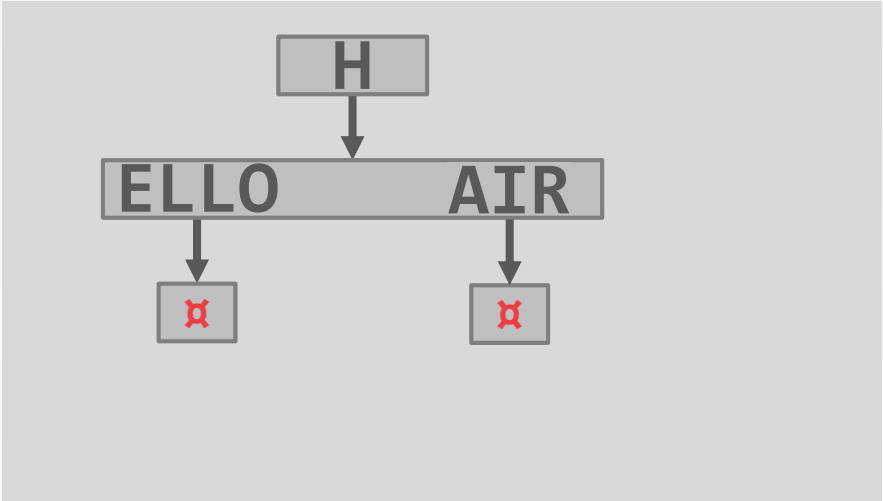
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

ART: BINARY COMPARABLE KEYS

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

→ **Unsigned Integers:** Byte order must be flipped for little endian machines.

→ **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.

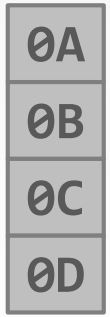
→ **Floats**: Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.

→ **Compound**: Transform each attribute separately.

ART: BINARY COMPARABLE KEYS

Int Key: 168496141

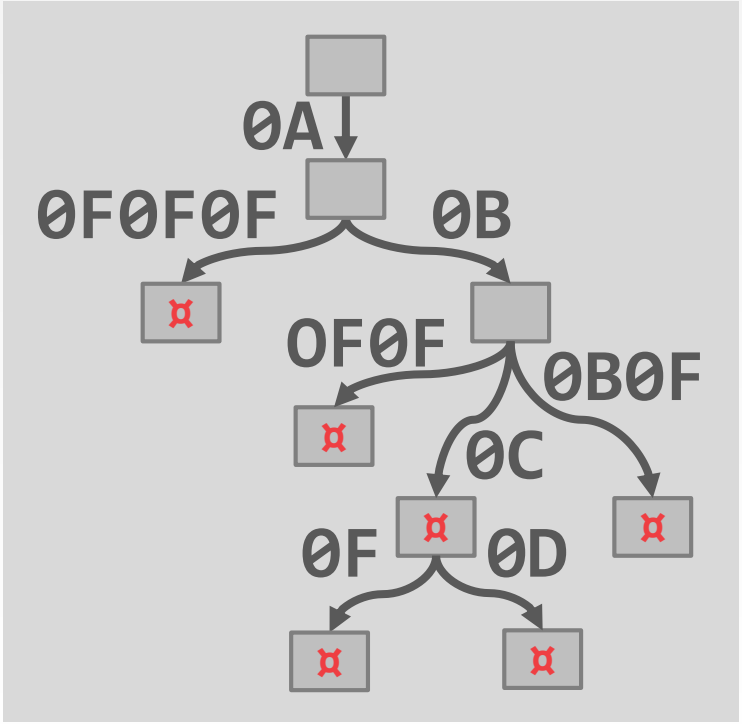
Hex Key: 0A 0B 0C 0D



Big Endian

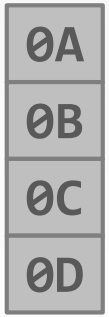
Little Endian

ART: BINARY COMPARABLE KEYS



Int Key: 168496141

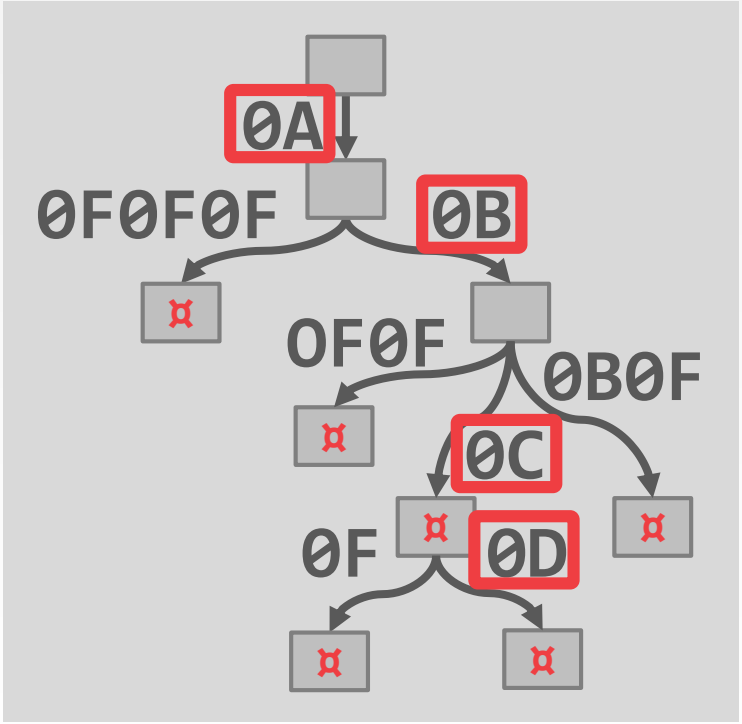
Hex Key: 0A 0B 0C 0D



Big Endian

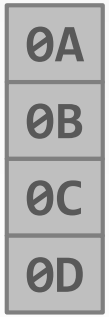
Little Endian

ART: BINARY COMPARABLE KEYS



Int Key: 168496141

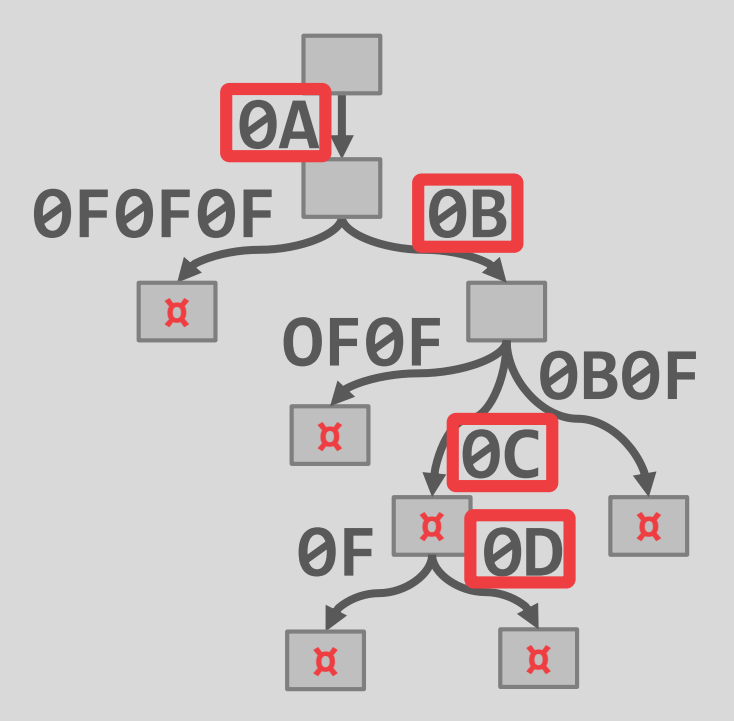
Hex Key: 0A 0B 0C 0D



Big Endian

Little Endian

ART: BINARY COMPARABLE KEYS



Int Key: 168496141

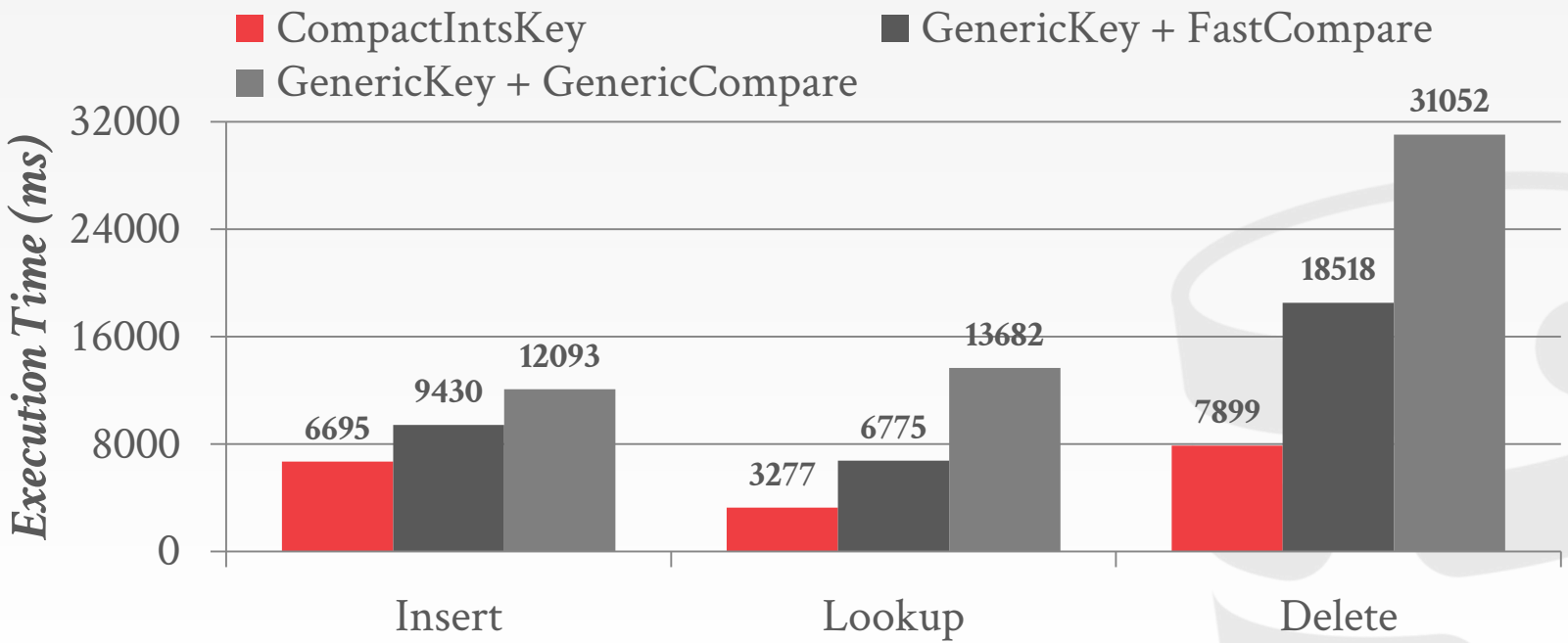
Hex Key: 0A 0B 0C 0D

Lookup: 658205

Hex: 0A 0B 1D

BINARY COMPRABLE KEYS

Peloton w/ Bw-Tree Index
Data Set: 10m keys (three 64-bit ints)



CONCURRENT ART INDEX

HyPer's ART is **not** latch-free.

→ The authors argue that it would be a significant amount of work to make it latch-free.

Approach #1: Optimistic Lock Coupling

Approach #2: Read-Optimized Write Exclusion

THE ART OF PRACTICAL SYNCHRONIZATION
DaMoN 2016

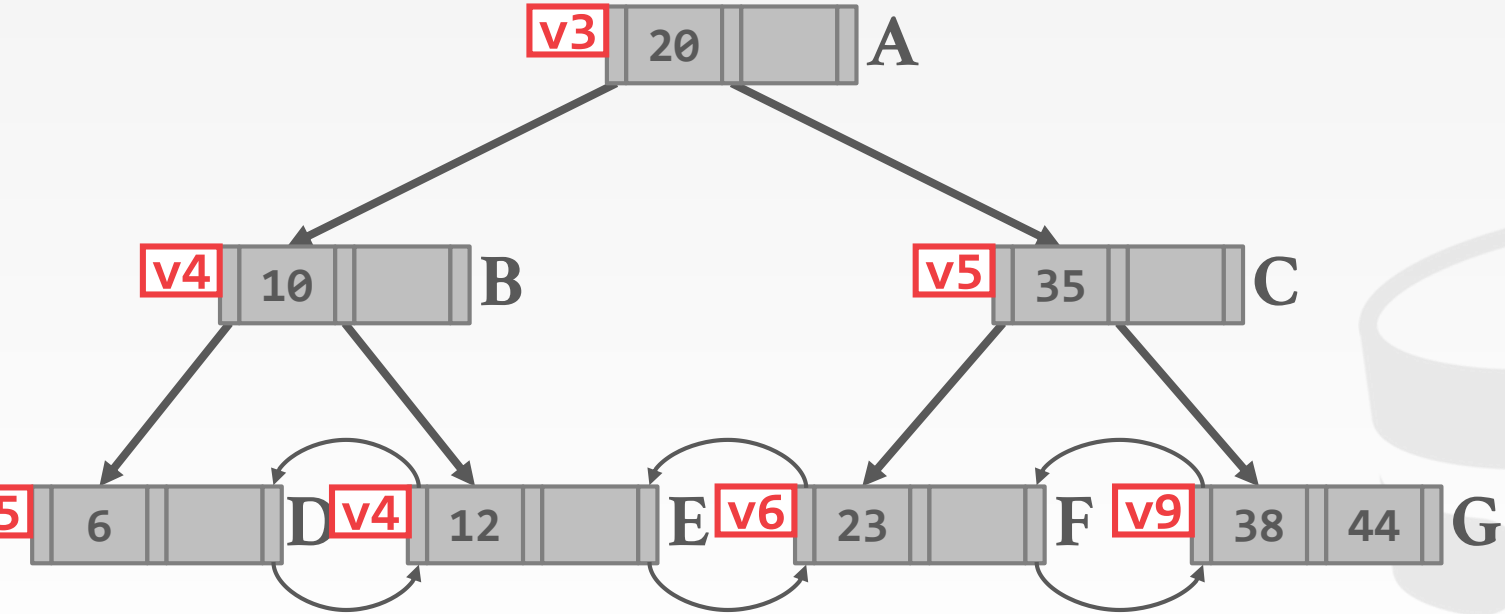
OPTIMISTIC LOCK COUPLING

Optimistic crabbing scheme where writers are not blocked on readers.

→ Writers increment counter when they acquire latch.
→ Readers can proceed if a node's latch is available.
→ It then checks whether the latch's counter has changed from when it checked the latch.

OPTIMISTIC LOCK COUPLING

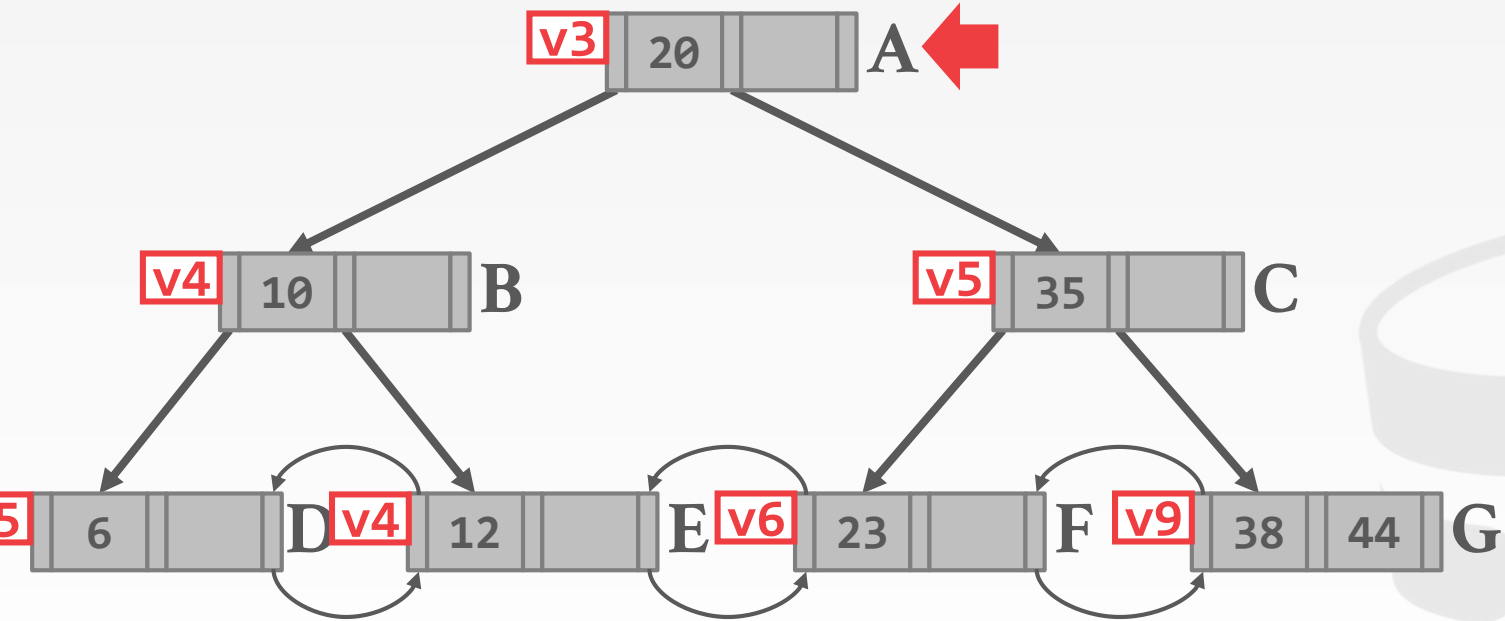
SEARCH 44



OPTIMISTIC LOCK COUPLING

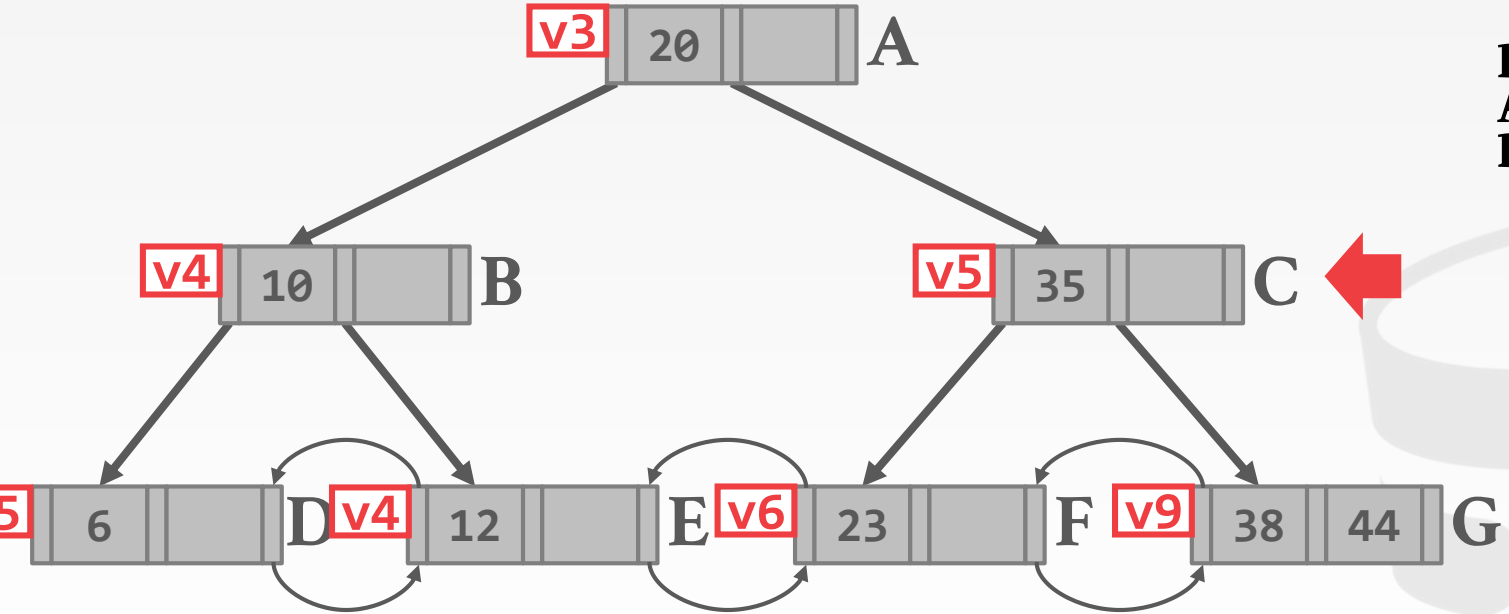
SEARCH 44

A: Read v3
A: Search Node



OPTIMISTIC LOCK COUPLING

SEARCH 44

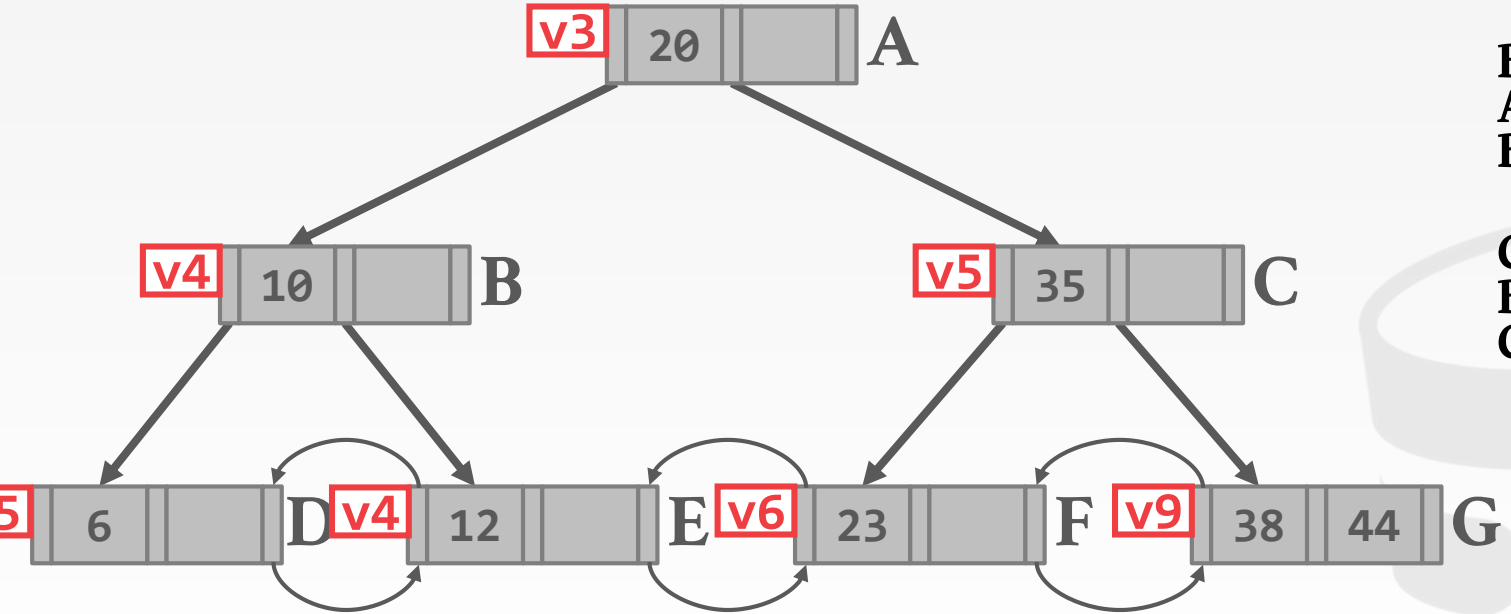


A: Read v3
A: Search Node

B: Read v5
A: Recheck v3
B: Search Node

OPTIMISTIC LOCK COUPLING

SEARCH 44



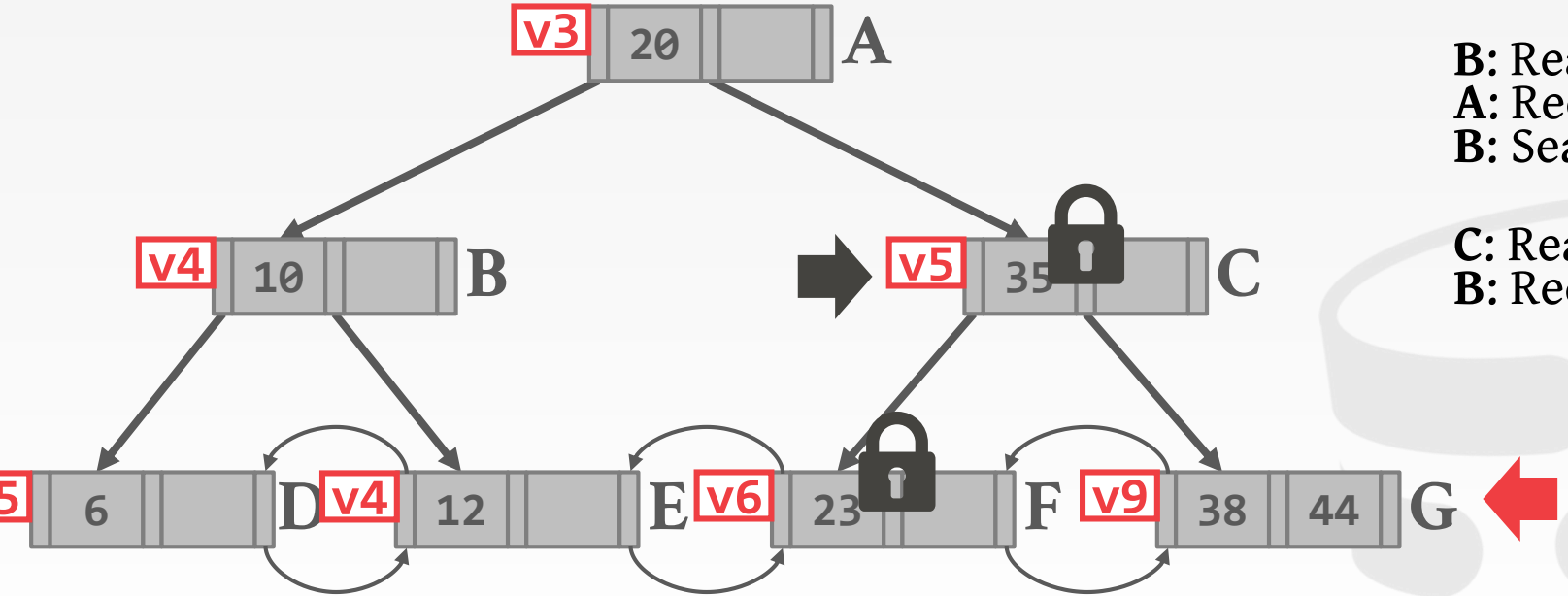
A: Read v3
A: Search Node

B: Read v5
A: Recheck v3
B: Search Node

C: Read v9
B: Recheck v5
C: Search Node

OPTIMISTIC LOCK COUPLING

SEARCH 44



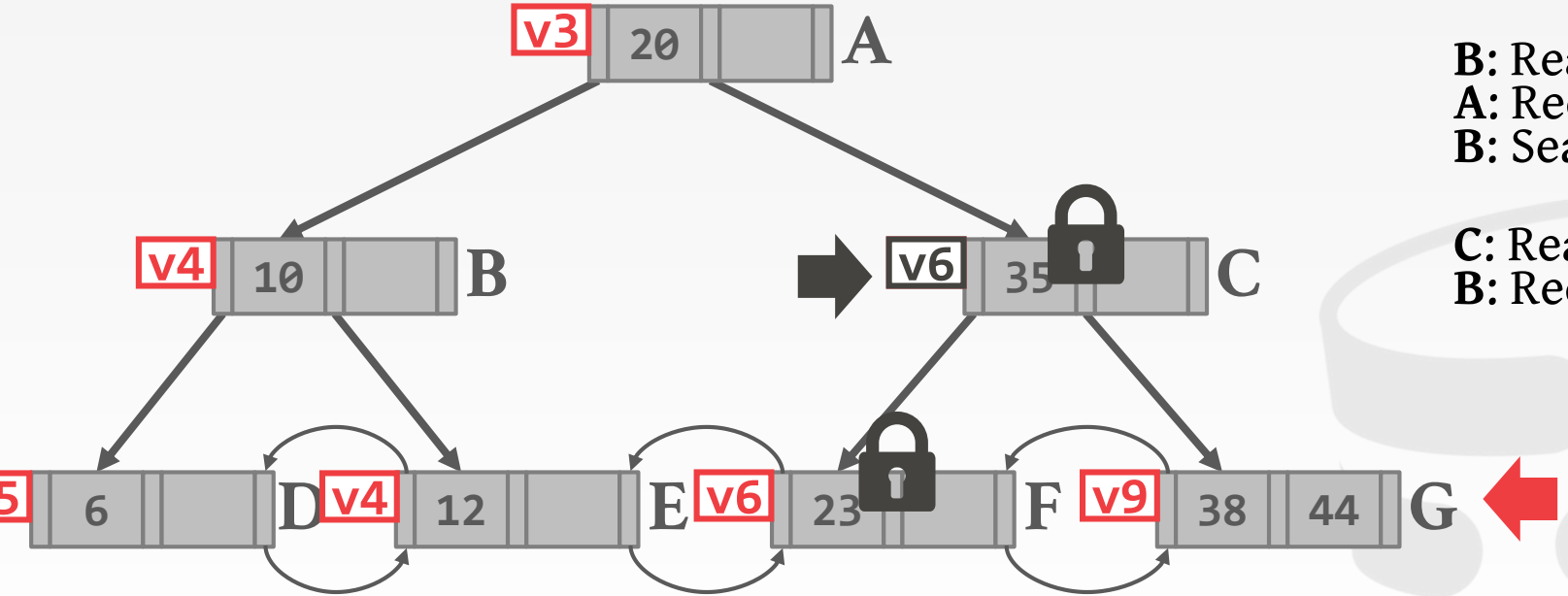
A: Read v3
A: Search Node

B: Read v5
A: Recheck v3
B: Search Node

C: Read v9
B: Recheck v5

OPTIMISTIC LOCK COUPLING

SEARCH 44



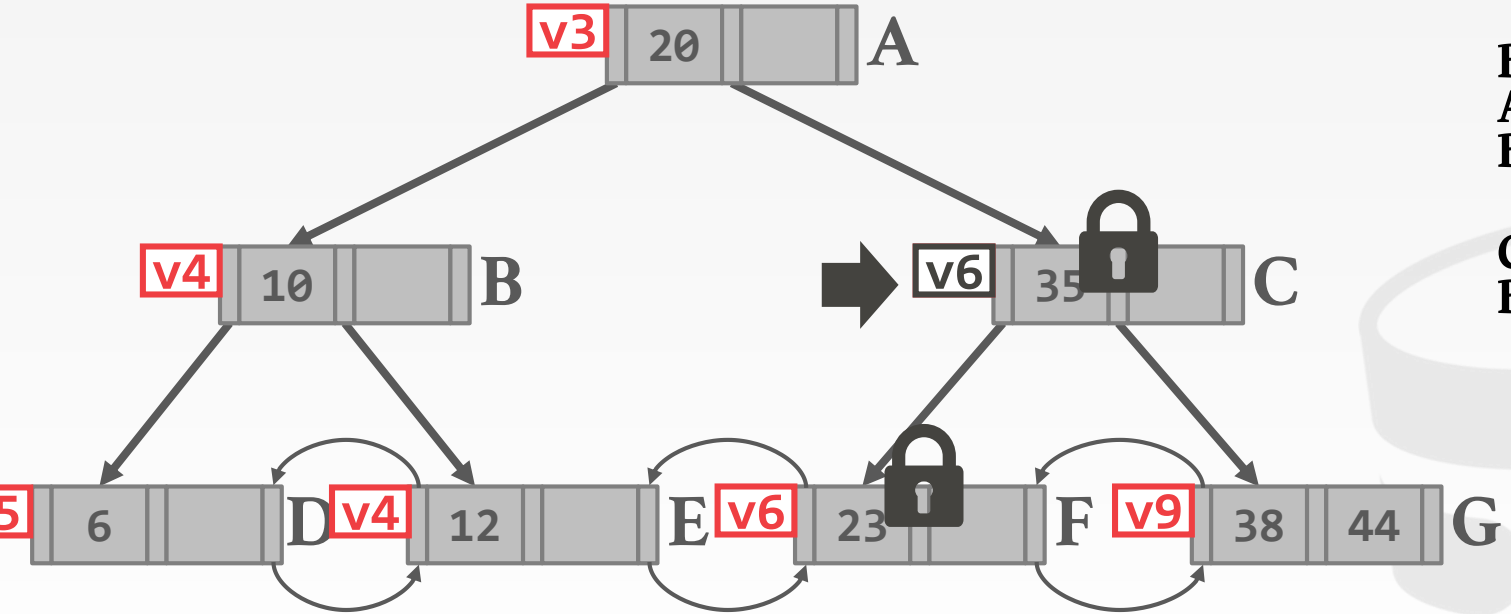
A: Read v3
A: Search Node

B: Read v5
A: Recheck v3
B: Search Node

C: Read v9
B: Recheck v5

OPTIMISTIC LOCK COUPLING

SEARCH 44



A: Read v3
A: Search Node

B: Read v5
A: Recheck v3
B: Search Node

C: Read v9
B: Recheck v5

READ-OPTIMIZED WRITE EXCLUSION

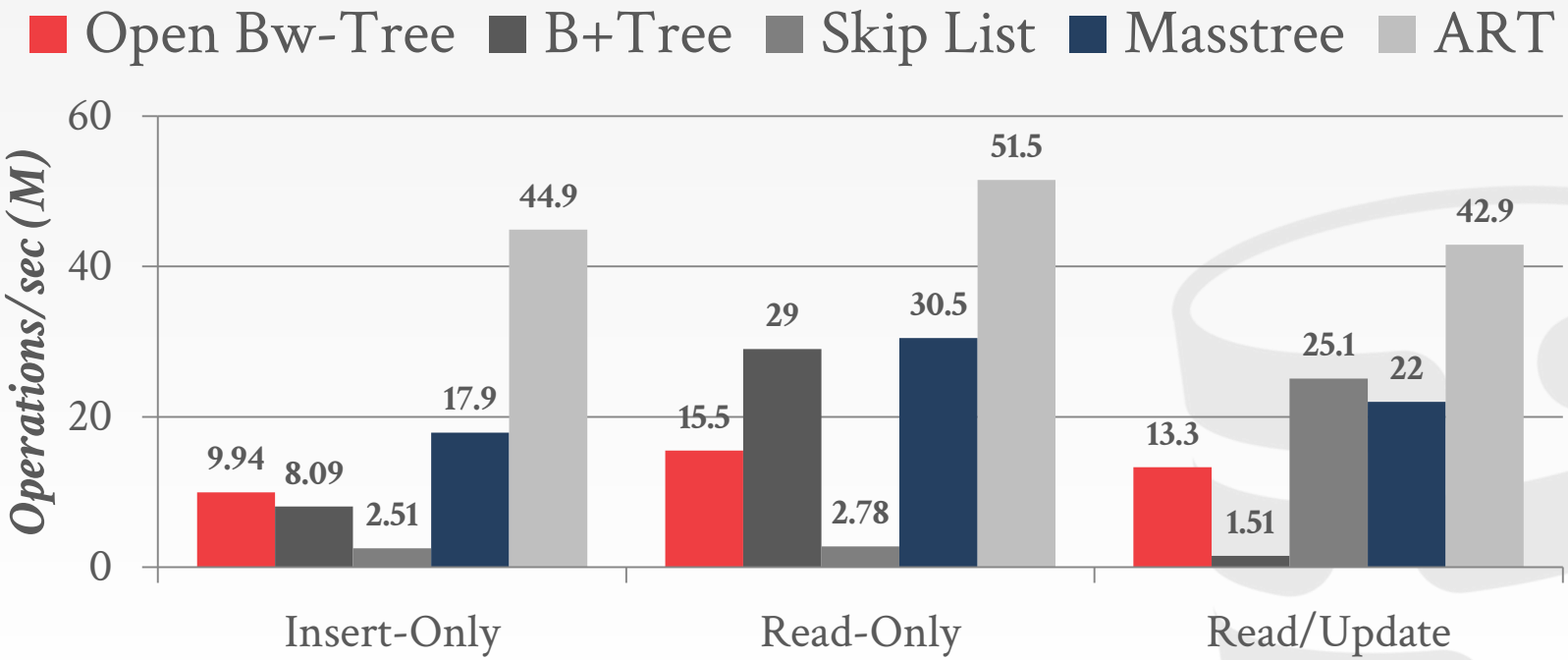
Each node includes an exclusive lock that blocks only other writers and not readers.

→ Readers proceed without checking versions or locks.
→ Every writer must ensure that reads are always consistent.

Requires fundamental changes to how threads make modifications to the data structure.

IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



Source: Ziqi Wang

PARTING THOUGHTS

Andy was wrong about the Bw-Tree and latch-free indexes.

ANDY'S

TIPS FOR PROFILING

MOTIVATION

Consider a program with functions `foo` and `bar`.

How can we speed it up with only a debugger ?

→ Randomly pause it during execution

→ Collect the function call stack

RANDOM PAUSE METHOD

Consider this scenario

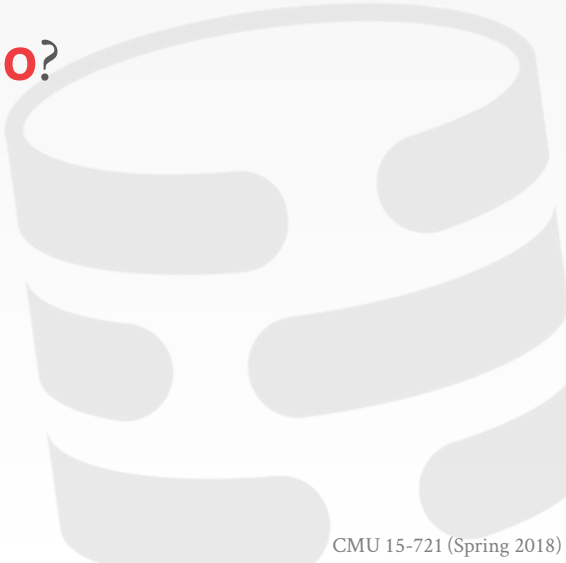
→ Collected 10 call stack samples

→ Say 6 out of the 10 samples were in **foo**

What percentage of time was spent in **foo**?

→ Roughly 60% of the time was spent in **foo**

→ Accuracy increases with # of samples



AMDAHL'S LAW

Say we optimized `foo` to run 2 times faster

What's the expected overall speedup ?

→ 60% of time spent in `foo` drops in half

→ 40% of time spent in `bar` unaffected

By Amdahl's law, overall speedup = $\frac{1}{\frac{p}{s}+(1-p)}$

→ p = percentage of time spent in optimized task

→ s = speed up for the optimized task

→ Overall speedup = $\frac{1}{\frac{0.6}{2}+0.4}$ = 1.4 times faster

PROFILING TOOLS FOR REAL

Choice #1: Valgrind

→ Heavyweight instrumentation framework with a lot of tools
→ Sophisticated visualization tools

Choice #2: Perf

→ Lightweight tool that can record different kinds of events
→ Console-oriented visualization tools

CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools

→ **`memcheck`**: a memory error detector
→ **`callgrind`**: a call-graph generating profiler
→ **`massif`**: memory usage tracking.

KCACHEGRIND

Using callgrind to profile the index test and Peloton in general:

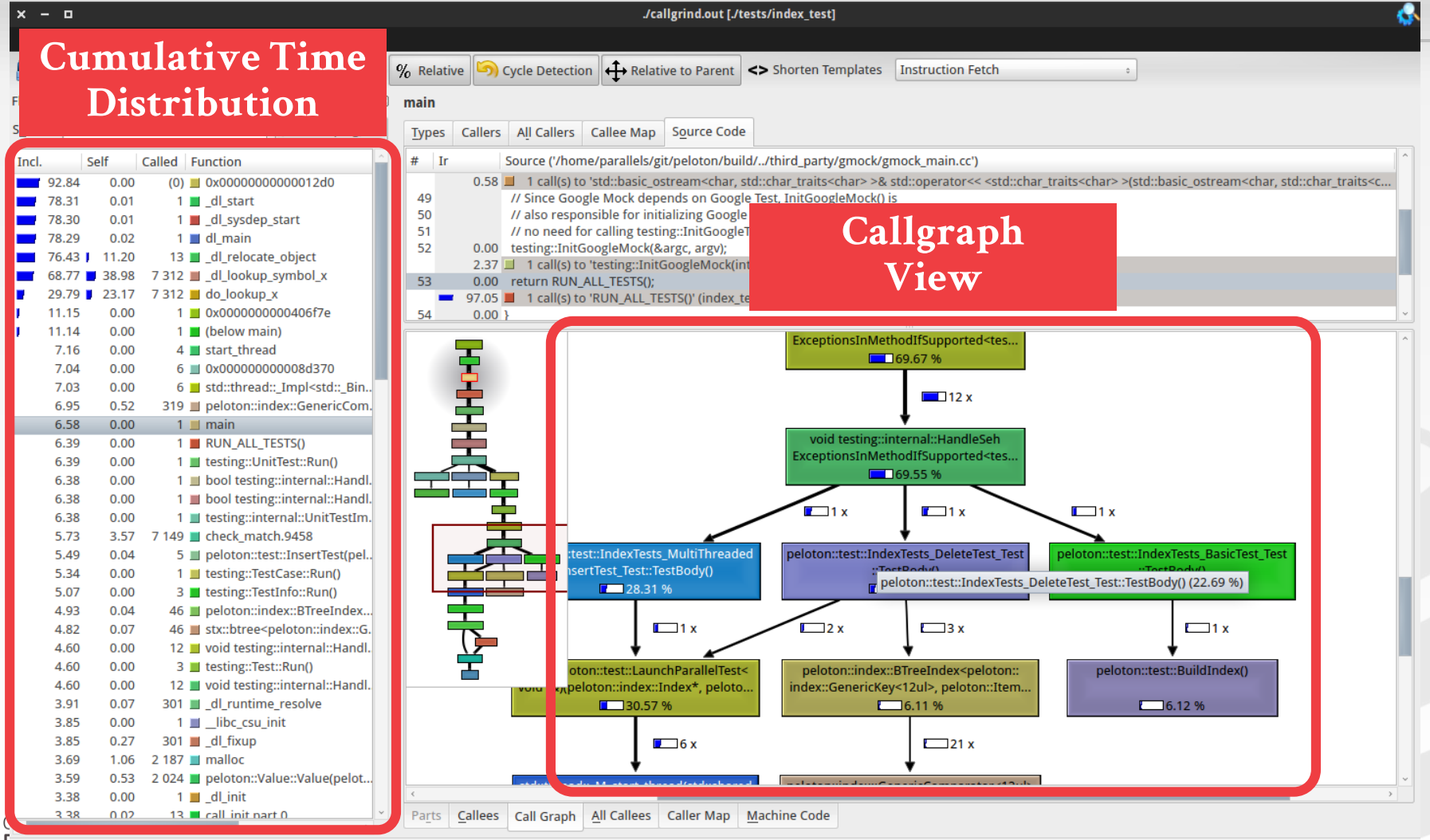
```
$ valgrind --tool=callgrind --trace-children=yes
./tests/skiplist_index_test

$ valgrind --tool=callgrind --trace-children=yes
./bin/peloton &> /dev/null&
```

Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```

Cumulative Time Distribution



CHOICE #2: PERF

Tool for using the performance counters subsystem in Linux.

→ `-e` = sample the event `cycles` at the user level only

→ `-c` = collect a sample every 2000 occurrences of event

```
$ perf record -e cycles:u -c 2000
./tests/skiplist_index_test
```

Uses counters for tracking events

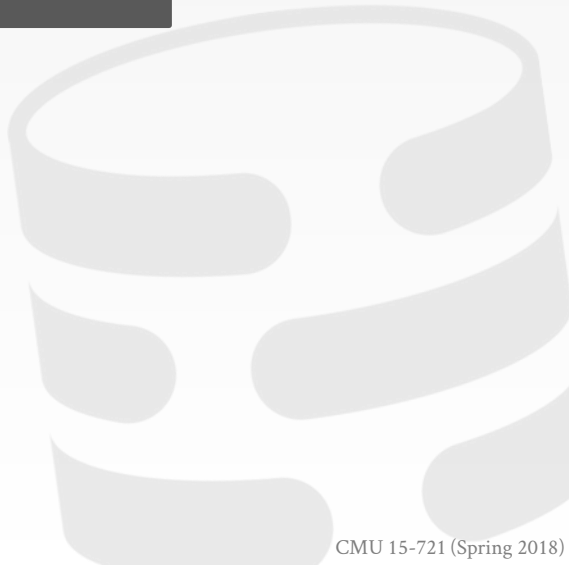
→ On counter overflow, the kernel records a sample

→ Sample contains info about program execution

PERF VISUALIZATION

We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```



PERF VISUALIZATION

perf report

File Edit View Search Terminal Help

```
Samples: 56  of event 'cpu-clock:u', Event count (approx.): 56
 25.00%  index_test  ld-2.19.so           [.] do_lookup_x
 25.00%  index_test  ld-2.19.so           [.] _dl_lookup_symbol_x
 21.43%  index_test  ld-2.19.so           [.] _dl_relocate_object
  7.14%  index_test  ld-2.19.so           [.] check_match.9458
  3.57%  index_test  libstdc++.so.6.0.21  [.] operator delete(void*)
  1.79%  index_test  libstdc++.so.6.0.21  [.] __dynamic_cast
  1.79%  index_test  libstdc++.so.6.0.21  [.] operator new(unsigned long)
  1.79%  index_test  libpelotonpg.so.0.0.0 [.] Json::Value::~Value()
  1.79%  index_test  libpeloton.so.0.0.0  [.] peloton::Value::CompareWithoutNull(peloton::Value) const
  1.79%  index_test  libc-2.19.so         [.] _int_free
  1.79%  index_test  libc-2.19.so         [.] __memcpy_sse2_unaligned
  1.79%  index_test  libc-2.19.so         [.] _dl_addr
  1.79%  index_test  libc-2.19.so         [.] __libc_dl_error_tsd
  1.79%  index_test  ld-2.19.so           [.] strcmp
  1.79%  index_test  index_test           [.] testing::TestEventListeners::TestEventListeners()
```

Cumulative Time Distribution

PERF EVENTS

Supports several other events like:
→ L1-dcache-load-misses
→ branch-misses

To see a list of events:

```
$ perf list
```

Another usage example:

```
$ perf record -e cycles,LLC-load-misses -c 2000
./tests/skiplist_index_test
```

REFERENCES

Valgrind

→ The Valgrind Quick Start Guide
→ Callgrind
→ Kcachegrind
→ Tips for the Profiling/Optimization process

Perf

→ Perf Tutorial
→ Perf Examples
→ Perf Analysis Tools

NEXT CLASS

Data Layout
Storage Models
System Catalogs