Arctic: a practical lock-free adaptive radix tree

Newton Ni[1], Nicolas Garza[1], Jenny Stinehour[1],
Michael Goppert[1], Michal Friedman[2], Emmett Witchel[1]



1 TEXAS The University of Texas at Austin

2 ETH zürich

Indexes associate keys with values…

```
{
    key-0: value-0,
    key-1: value-1,
    key-2: value-2,
}
```

For example,

Python **dict**

Rust **std::collections::HashMap**

and **concurrent** indexes can be safely accessed by multiple threads at once.

Concurrent indexes are widely used + performance critical

Databases

(MVCC indexes)



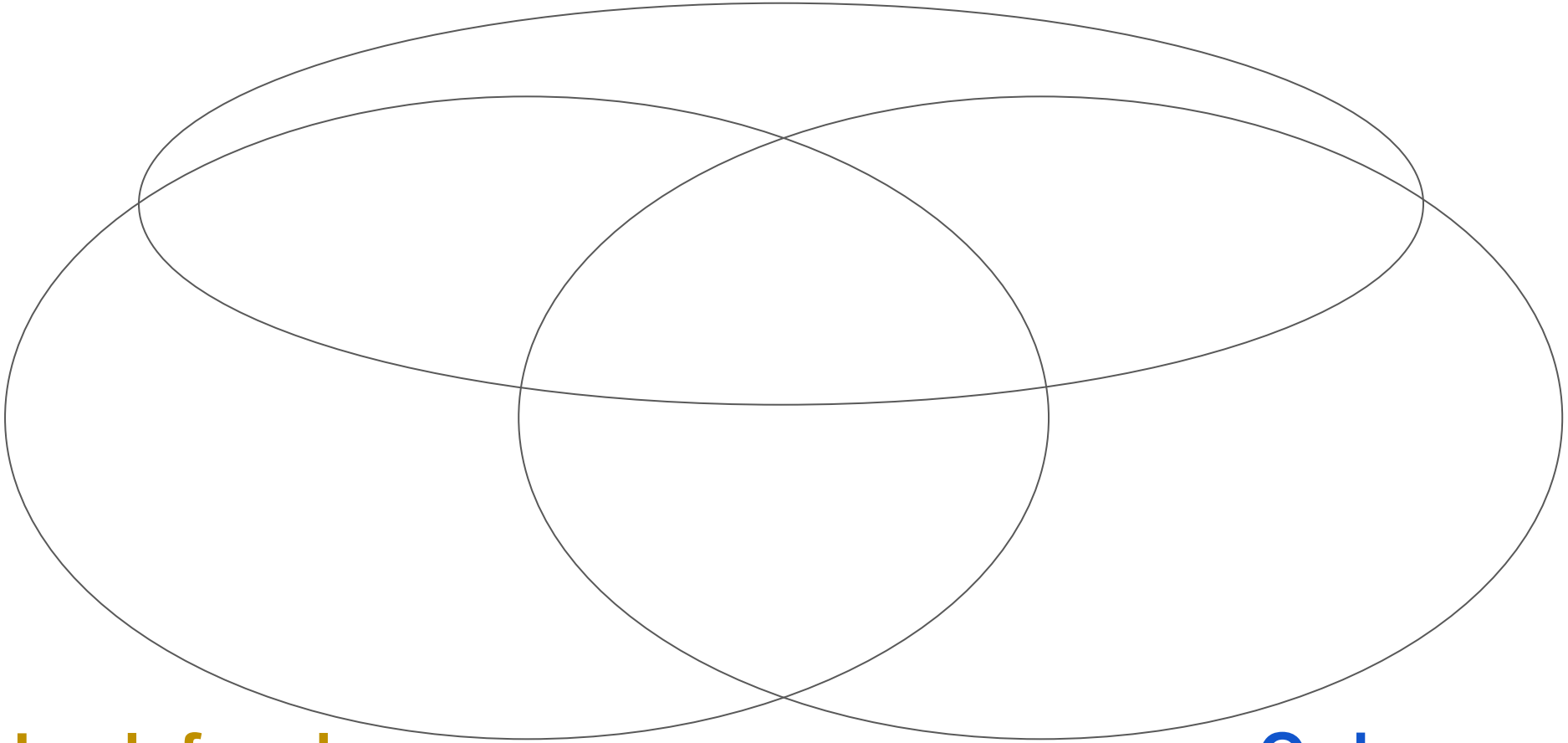
8% E2E throughput increase from skiplist to Arctic

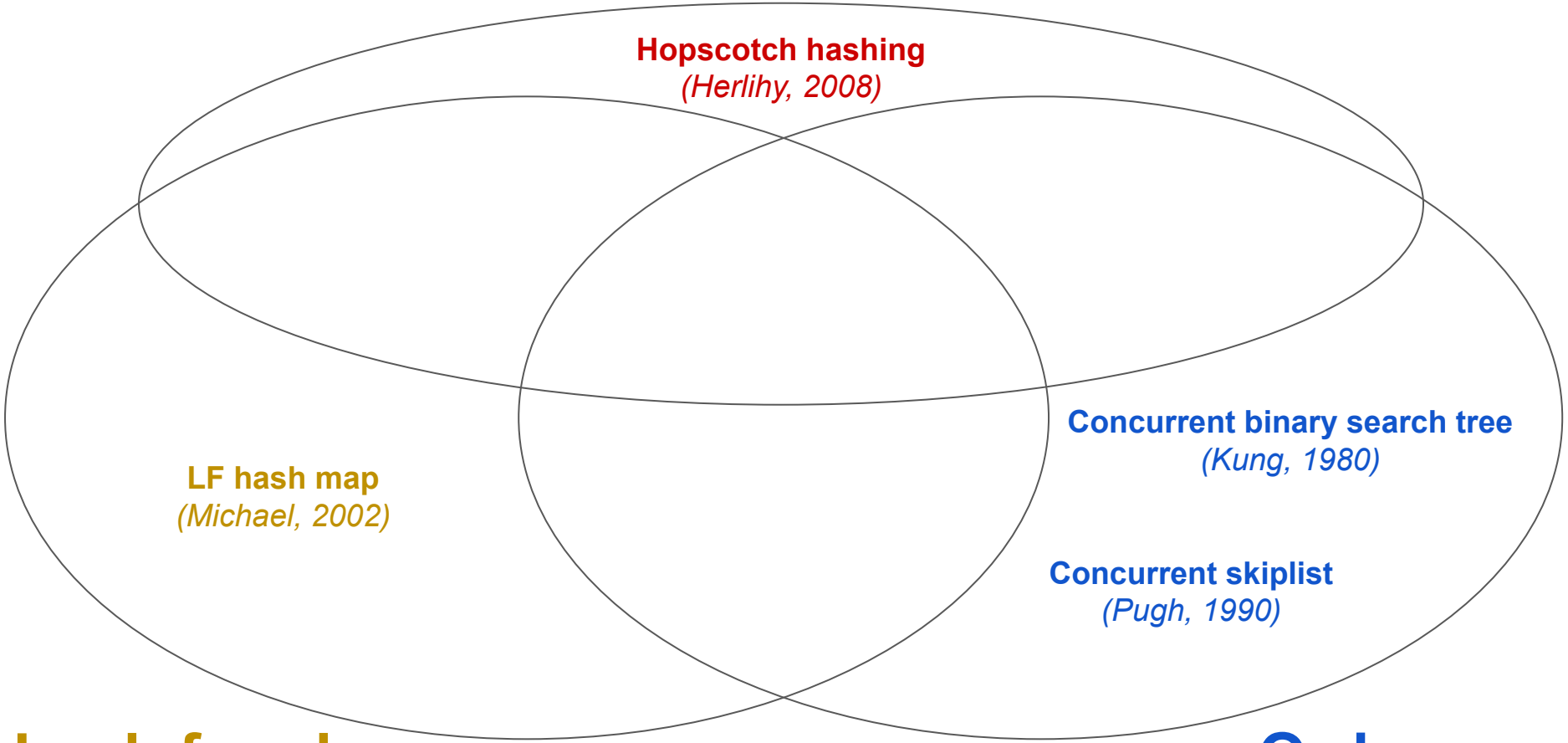
Key-value stores

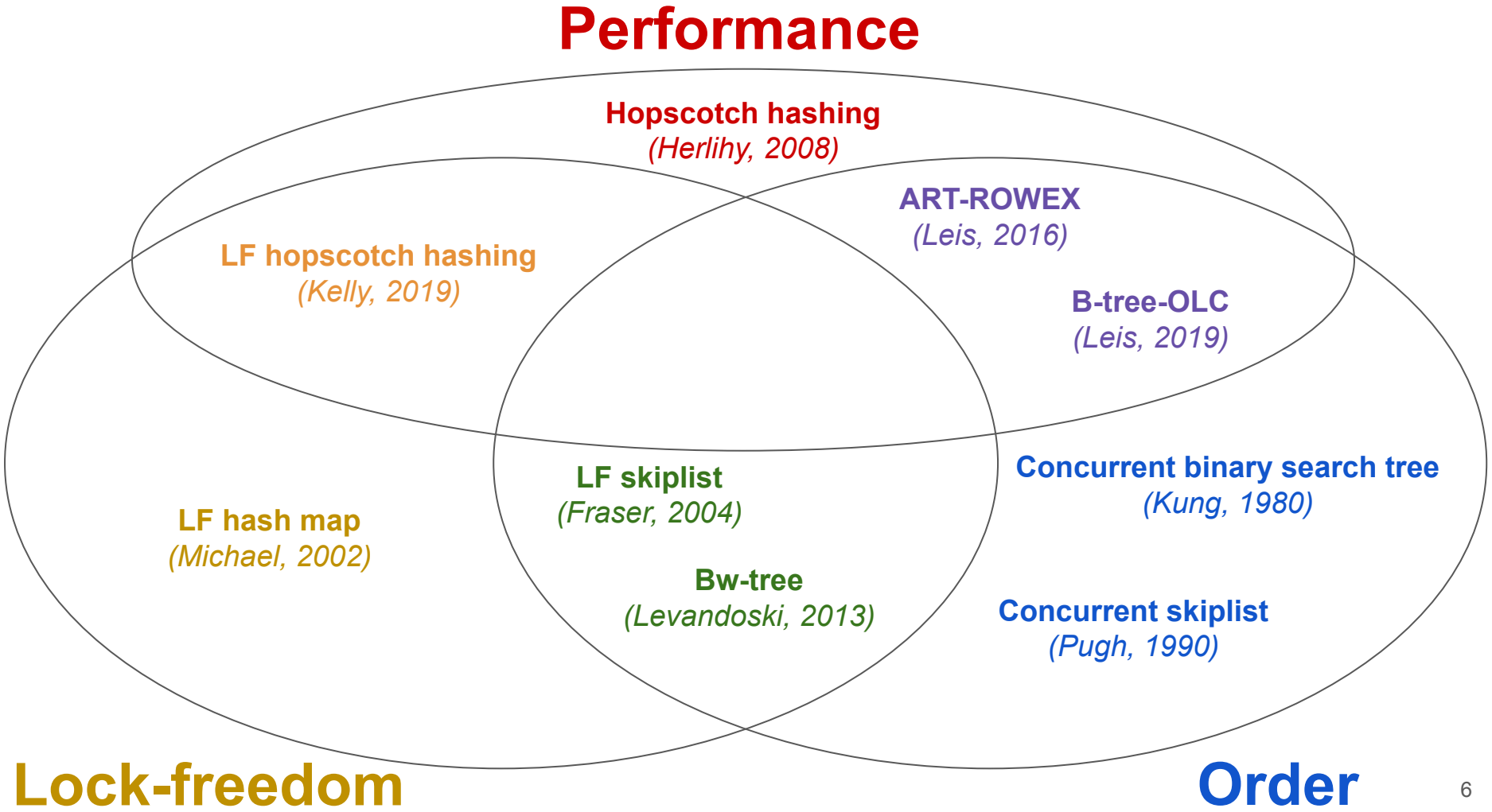
(memtables in LSM trees)



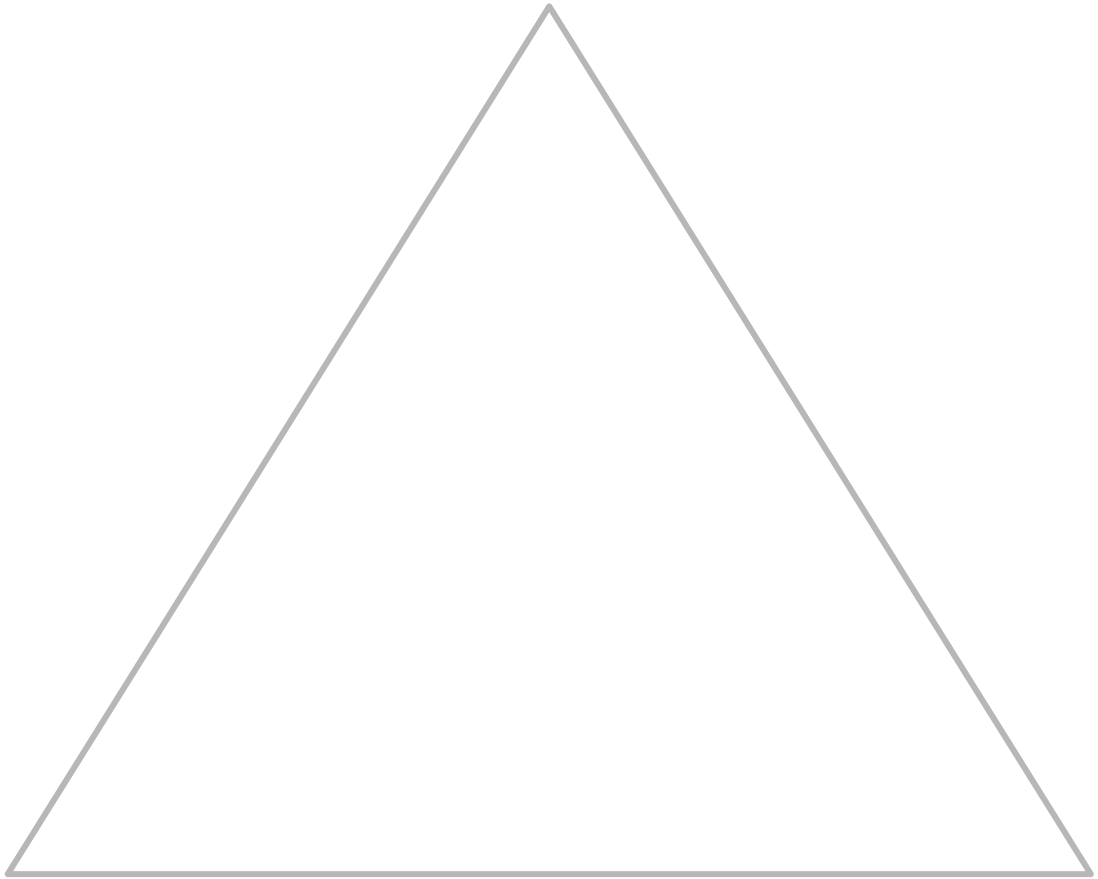
13% E2E throughput increase from skiplist to Arctic


Performance
Lock-freedom
Order


Performance
Hopscotch hashing
(Herlihy, 2008)
LF hash map
(Michael, 2002)
Concurrent binary search tree
(Kung, 1980)
Concurrent skiplist
(Pugh, 1990)
Lock-freedom
Order


Performance
Hopscotch hashing
(Herlihy, 2008)
ART-ROWEX
(Leis, 2016)
LF hopscotch hashing
(Kelly, 2019)
B-tree-OLC
(Leis, 2019)
LF skiplist
(Fraser, 2004)
Concurrent binary search tree
(Kung, 1980)
LF hash map
(Michael, 2002)
Bw-tree
(Levandoski, 2013)
Concurrent skiplist
(Pugh, 1990)
Lock-freedom
Order

Performance

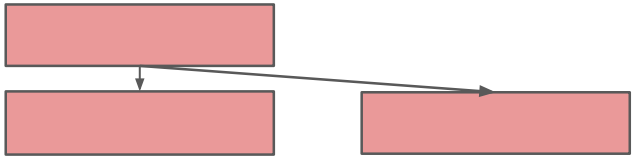


Lock-freedom

Order

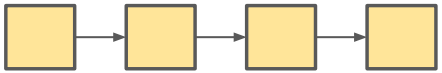
large allocations

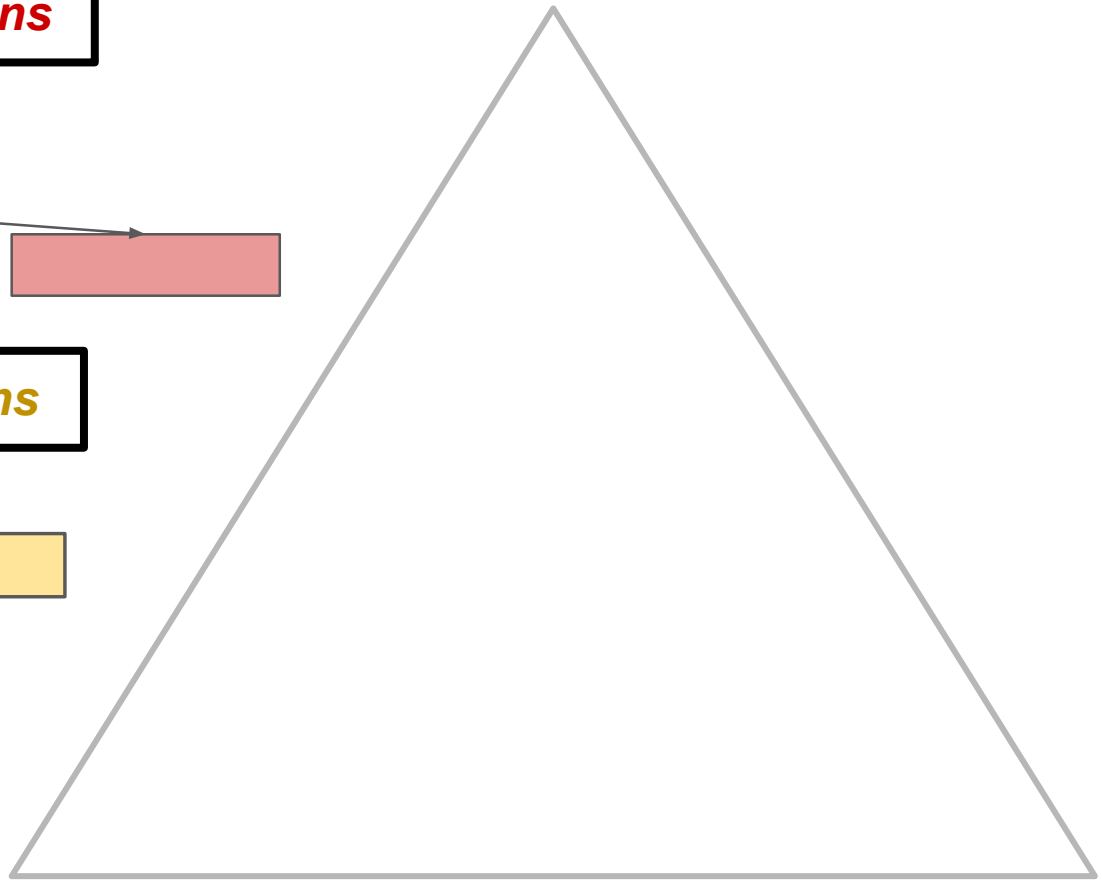
B-tree



small allocations

Linked list

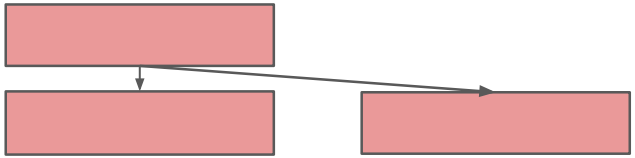




Performance

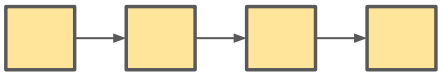
large allocations

B-tree



small allocations

Linked list



O(1), less contention

Hash map



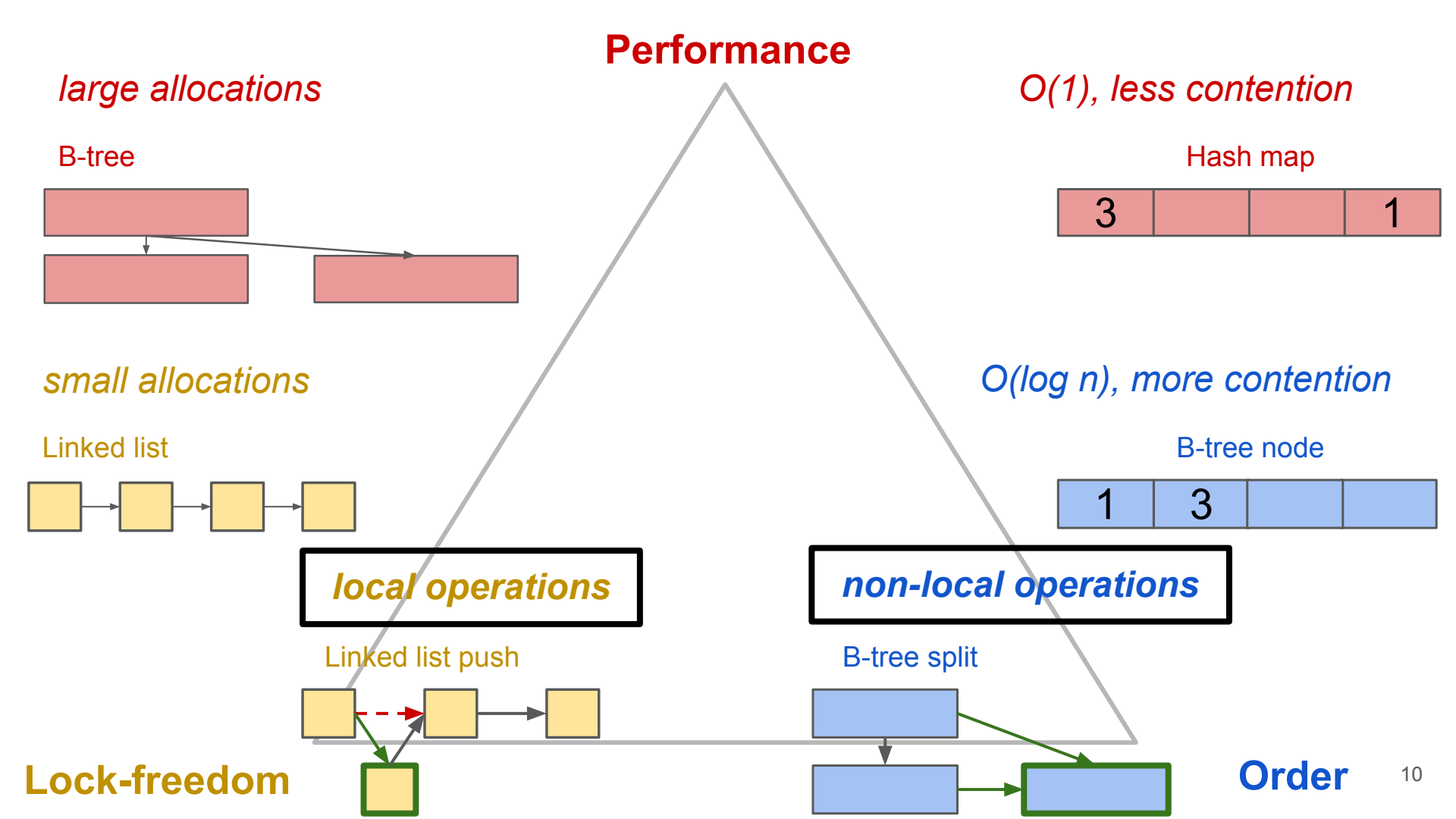
O(log n), more contention

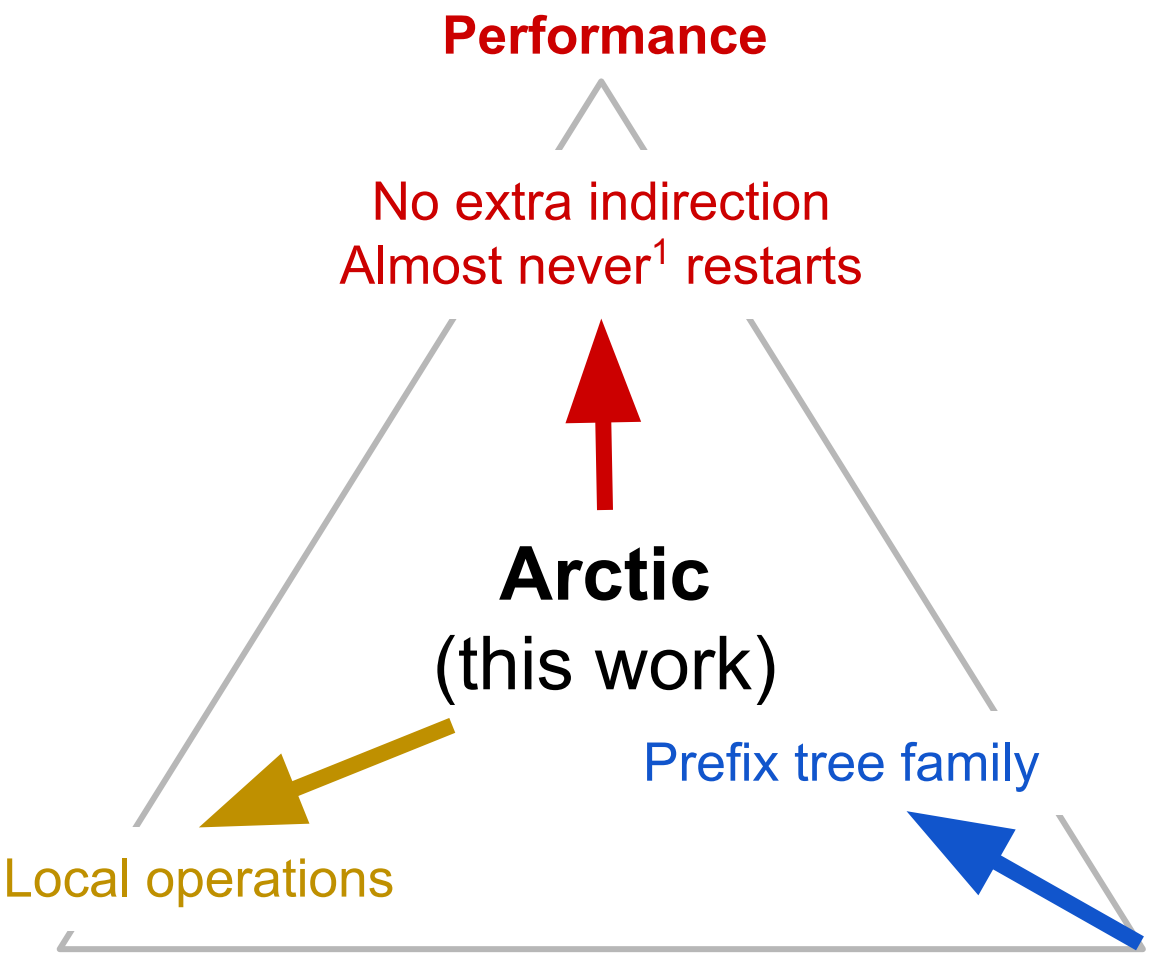
B-tree node



Lock-freedom

Order


Performance
large allocations
O(1), less contention
B-tree
Hash map
3
1
small allocations
O(log n), more contention
Linked list
B-tree node
1
3
local operations
non-local operations
Linked list push
B-tree split
Lock-freedom
Order



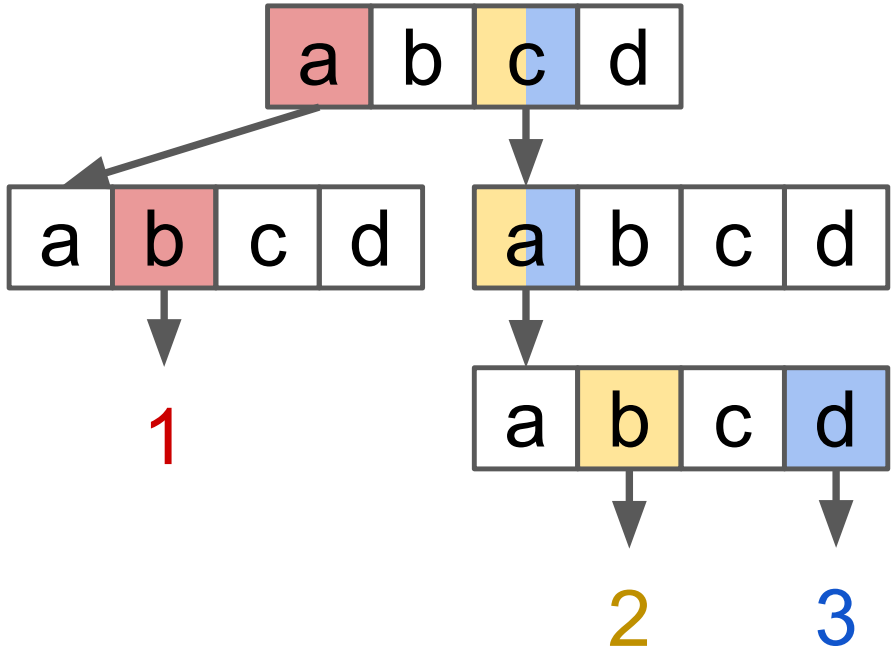
Lock-freedom

[1]< 0.0004% (see paper)

Order

Background: prefix trees (tries)

```
{
    ab: 1,
    cab: 2,
    cad: 3,
}
```



Background: prefix trees (tries)

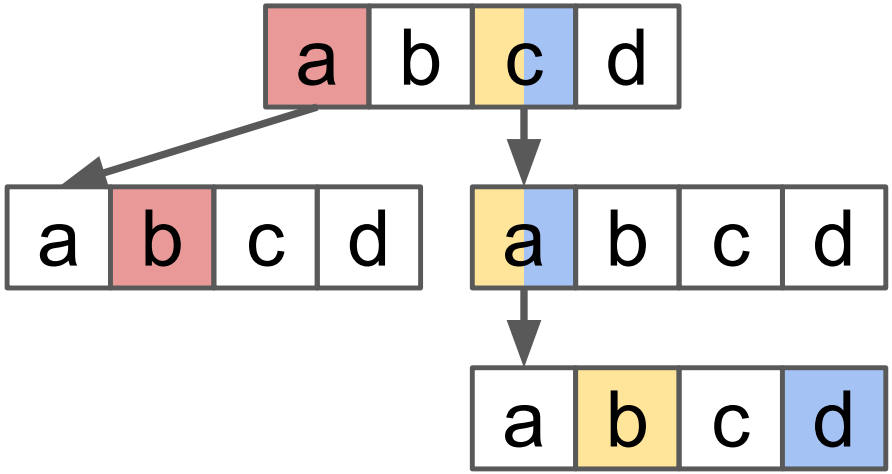
{

ab,

cab,

cad,

}

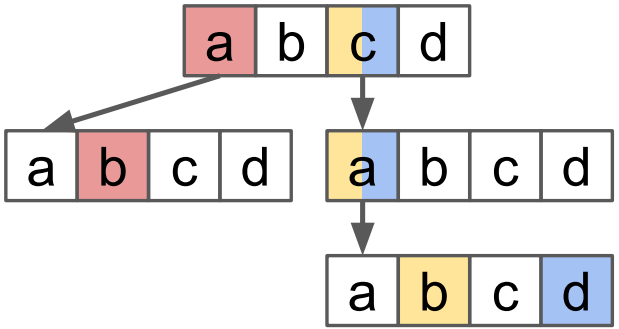


Evolution of prefix trees

Prefix tree (trie)

- **Tree depth = key length**
- **Node size = constant**

{ **ab**, **cab**, **cad** }

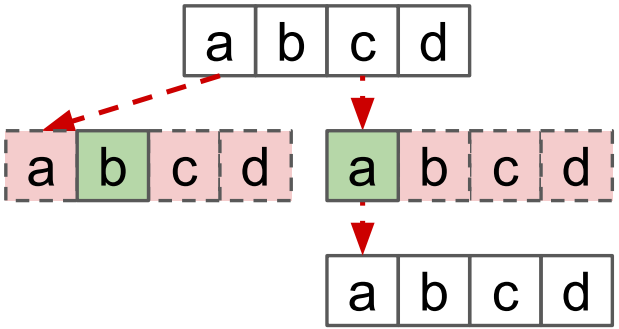


Evolution of prefix trees

Prefix tree (trie)

- Tree depth = key length
- Node size = constant

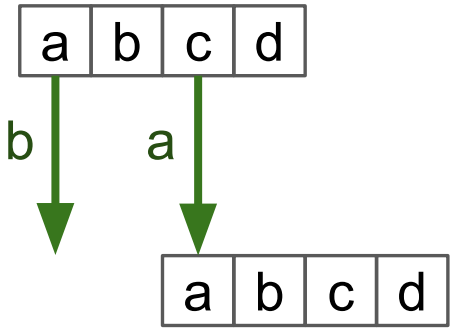
{ ab, cab, cad }



Radix tree

+ **Compress edges**
- **Node size = constant**

{ ab, cab, cad }

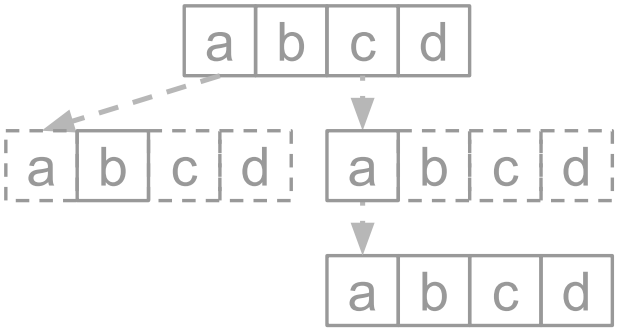


Evolution of prefix trees

Prefix tree (trie)

- Tree depth = key length
- Node size = constant

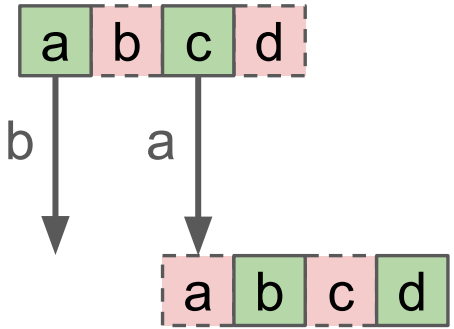
{ ab, cab, cad }



Radix tree

+ Compress edges
- Node size = constant

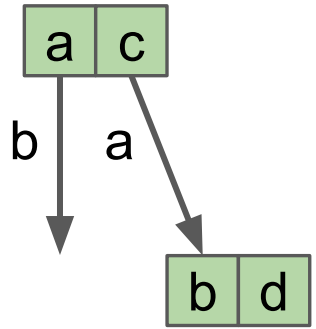
{ ab, cab, cad }



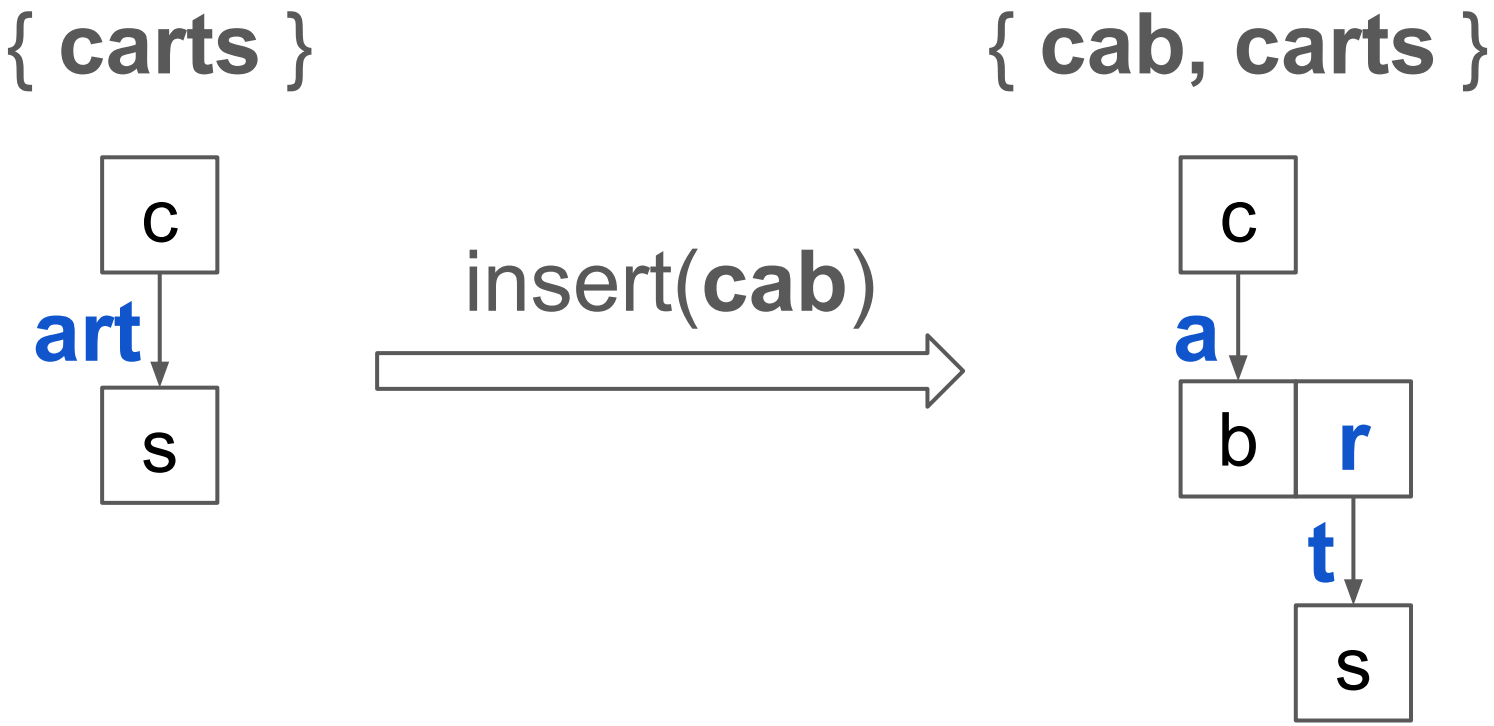
Adaptive radix tree

+ **Compress edges**
+ **Compress nodes**

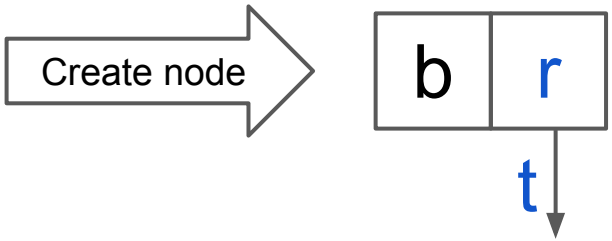
{ ab, cab, cad }



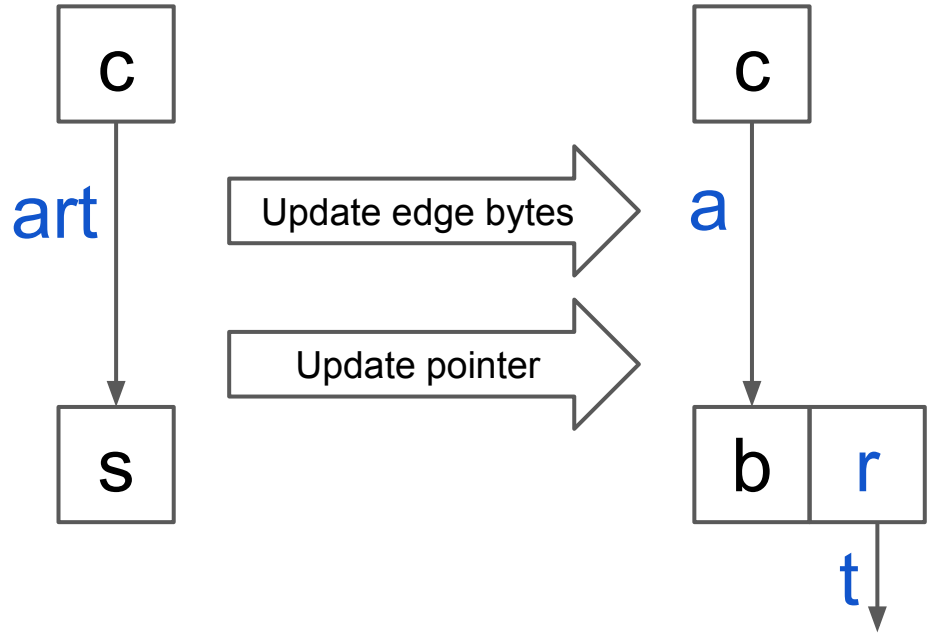
Design highlight 1/3: edge expansion



Non-atomic

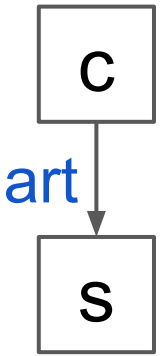


Atomic

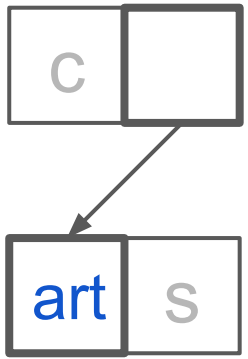


Edge expansion: memory layout

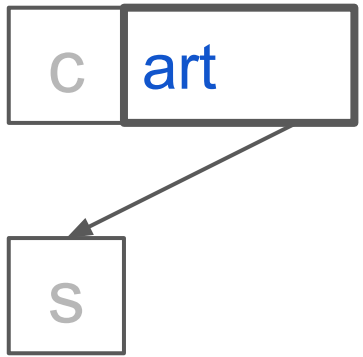
Logical



Edge bytes in **child**
(ART-ROWEX)

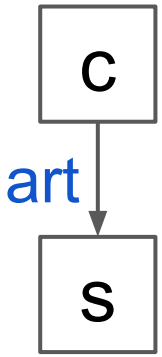


Edge bytes in **parent**
(Arctic)

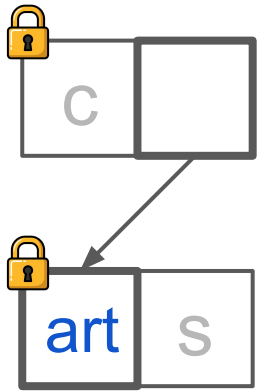


Edge expansion: synchronization

Logical

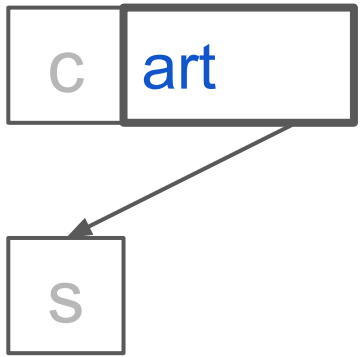


Edge bytes in **child**
(ART-ROWEX)



Lock parent and child

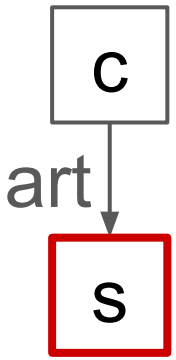
Edge bytes in **parent**
(Arctic)



Atomic 16B CAS

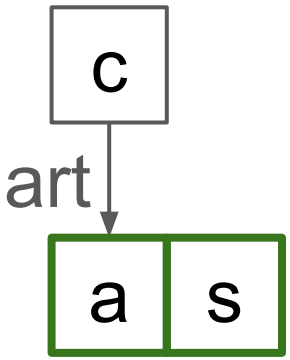
Design highlight 2/3: node replacement

{ **carts** }

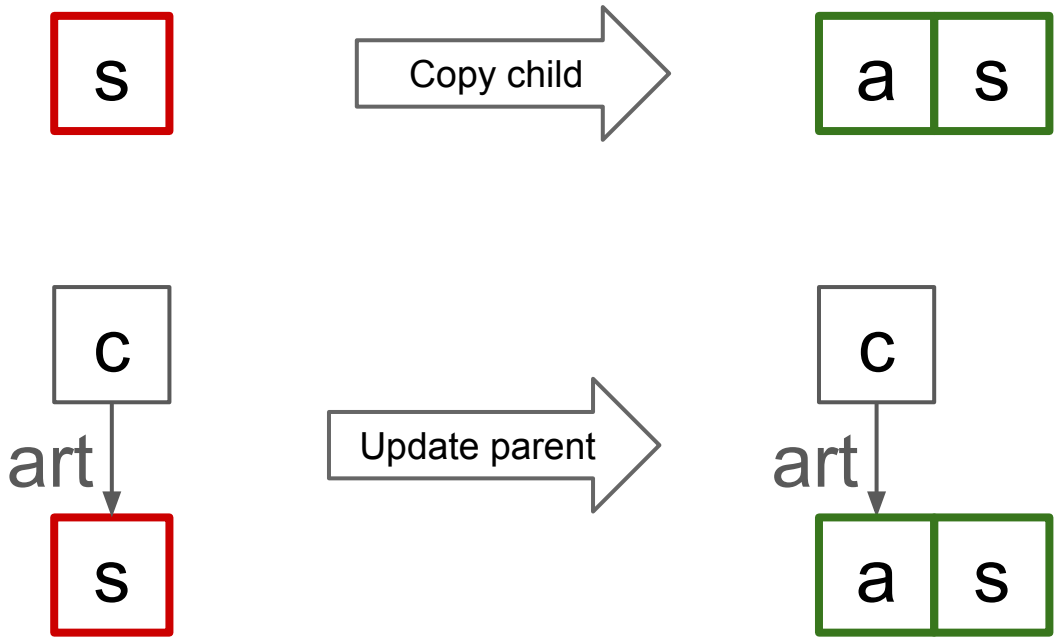


insert(**carta**)

{ **carta**, **carts** }

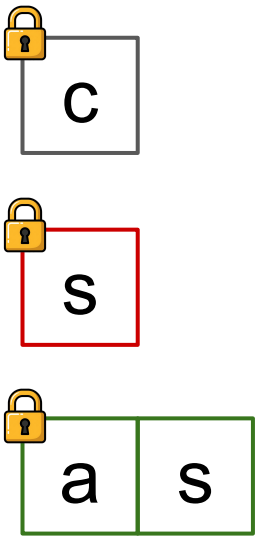


Atomic

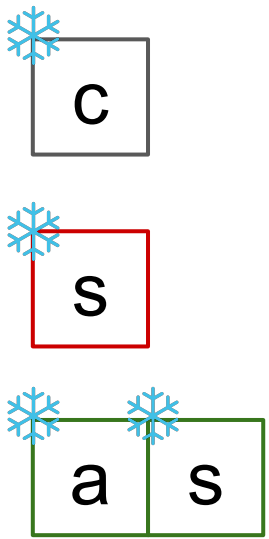


Node replacement: memory layout

Lock per node
(ART-ROWEX)

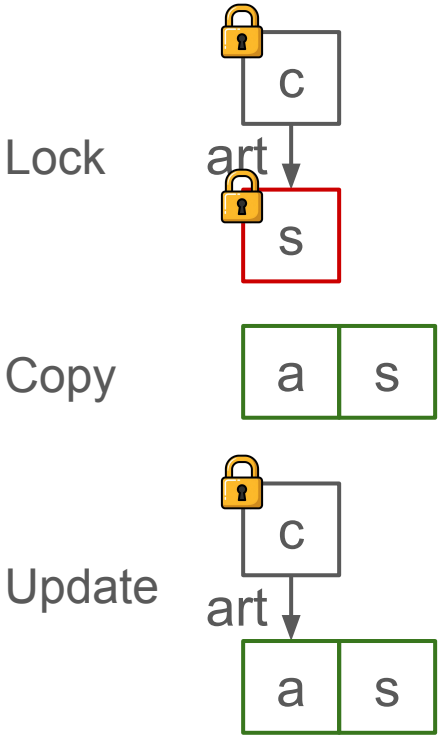


Frozen bit per CAS site (Braginsky '11)
(Arctic)

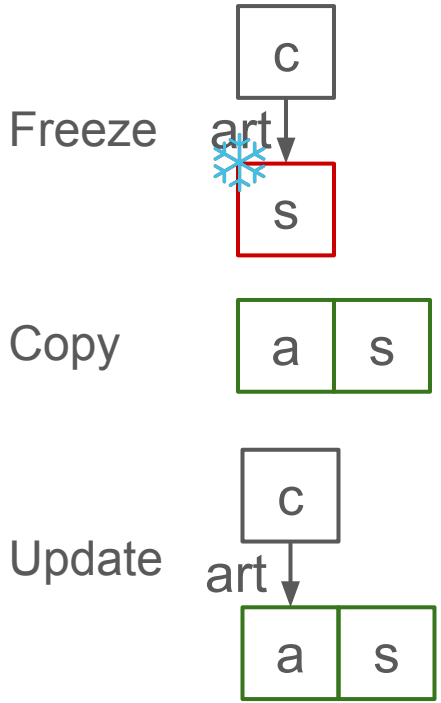


Node replacement: synchronization

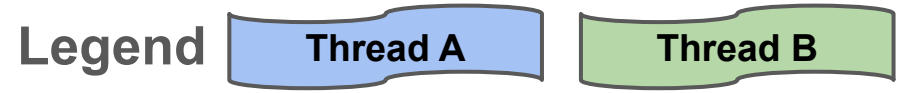
Lock per node
(ART-ROWEX)



Frozen bit per CAS site (Braginsky '11)
(Arctic)

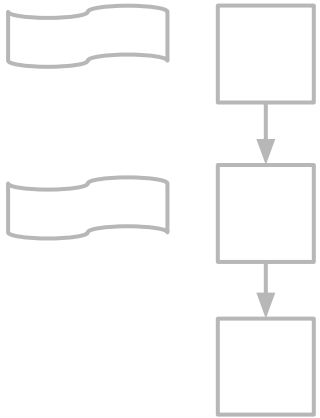


What's the point?

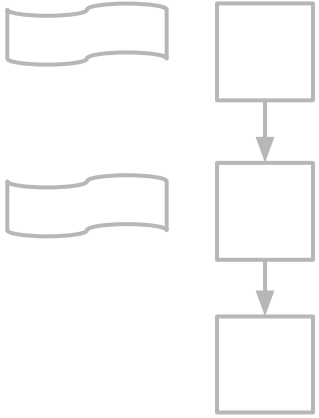


Vertical concurrency (between nodes)

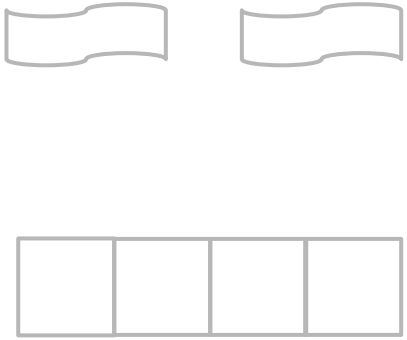
Horizontal concurrency (within node)



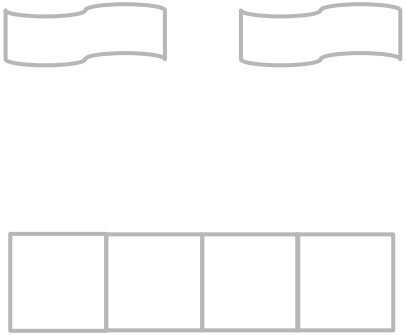
Edge byte in parent
(ART-ROWEX)



Edge byte in child
(Arctic)

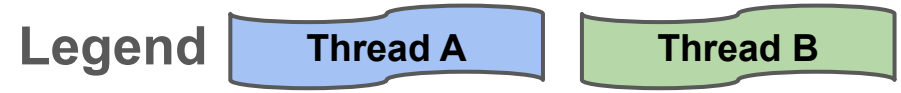


Lock per node
(ART-ROWEX)



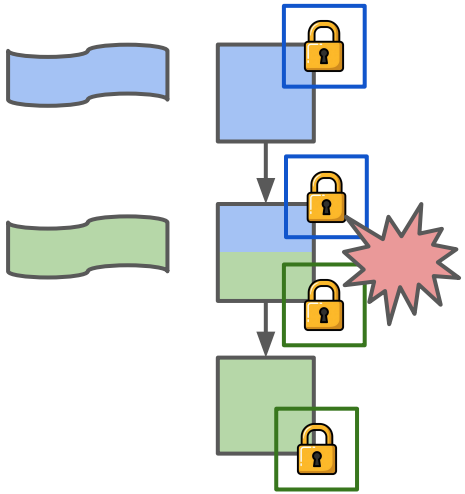
Frozen bit per CAS
(Arctic)

What's the point?

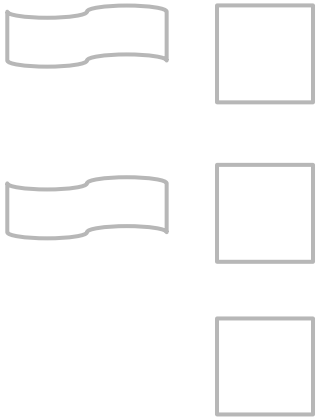


Vertical concurrency (between nodes)

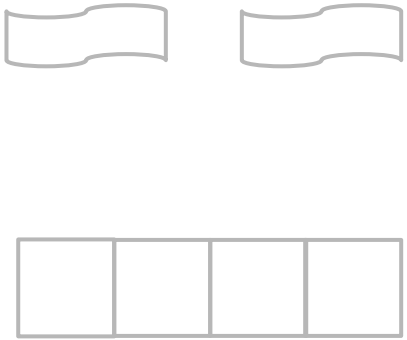
Horizontal concurrency (within node)



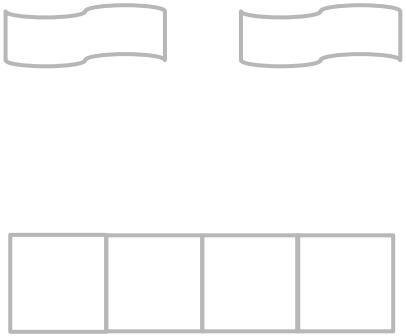
Edge byte in parent
(ART-ROWEX)



Edge byte in child
(Arctic)



Lock per node
(ART-ROWEX)

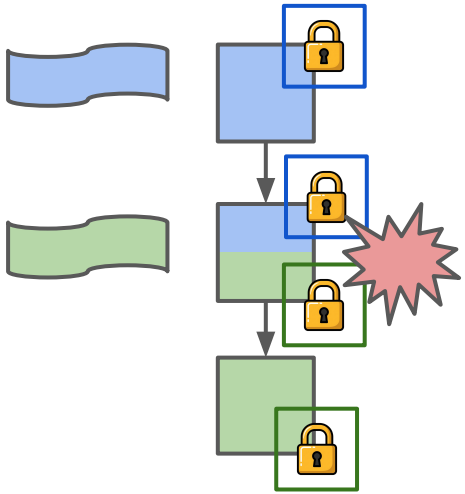


Frozen bit per CAS
(Arctic)

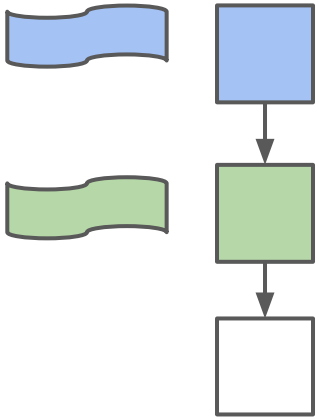
What's the point?

Legend: Thread A, Thread B

Vertical concurrency (between nodes)

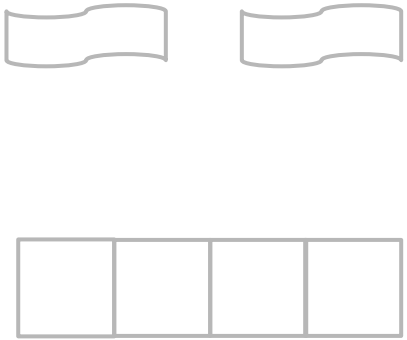


Edge byte in parent
(ART-ROWEX)

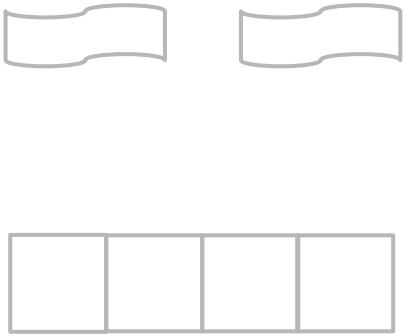


Edge byte in child
(Arctic)

Horizontal concurrency (within node)



Lock per node
(ART-ROWEX)

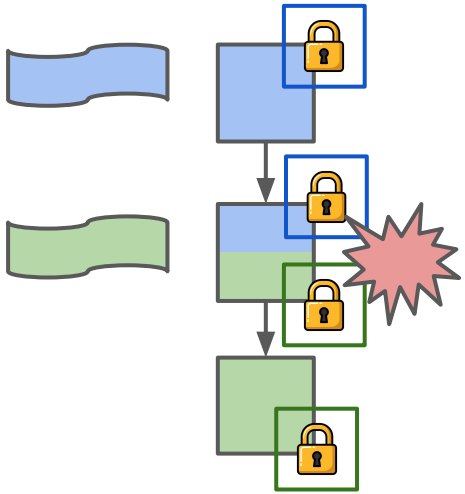


Frozen bit per CAS
(Arctic)

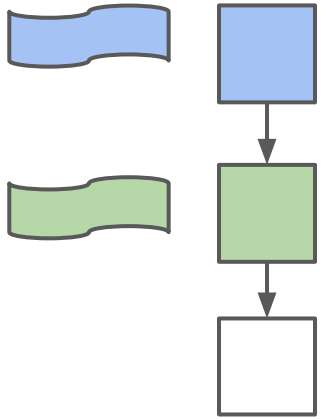
What's the point?



Vertical concurrency (between nodes)

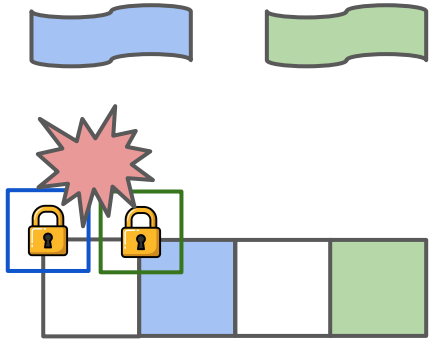


Edge byte in parent
(ART-ROWEX)

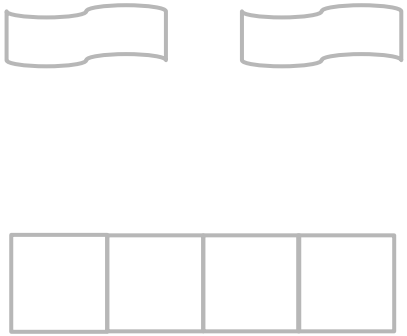


Edge byte in child
(Arctic)

Horizontal concurrency (within node)



Lock per node
(ART-ROWEX)

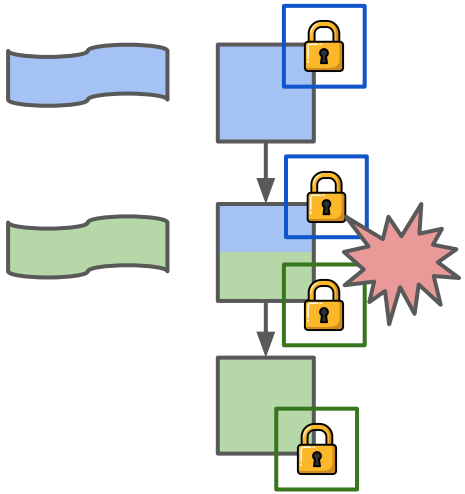


Frozen bit per CAS
(Arctic)

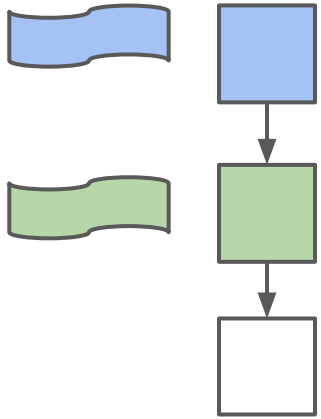
What's the point?



Vertical concurrency (between nodes)

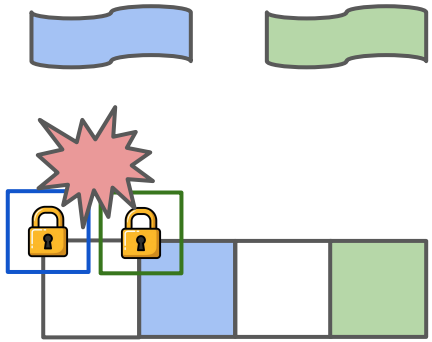


Edge byte in parent
(ART-ROWEX)

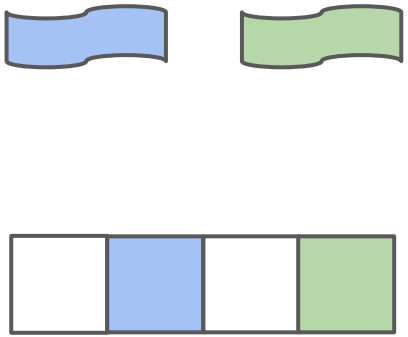


Edge byte in child
(Arctic)

Horizontal concurrency (within node)

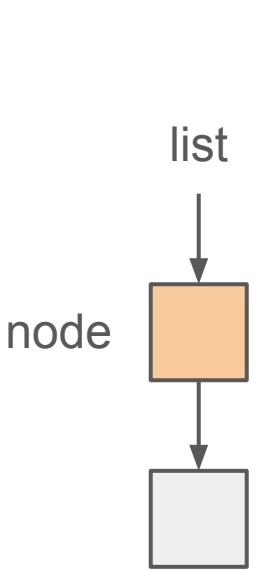


Lock per node
(ART-ROWEX)



Frozen bit per CAS
(Arctic)

Design highlight 3/3: safe memory reclamation (SMR)



Accessor

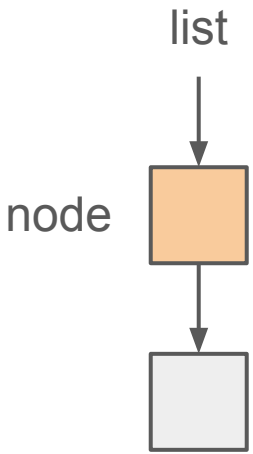
```
node = load(list)
// Lunch break
sleep(3600)
node.value = 5
```



Reclaimer

```
node = pop(list)
free(node)
```

Temporal schemes: track time periods



Accessor

stop = ∞; start = now()

node = load(list)

// Lunch break

sleep(3600)

node.value = 5

stop = now()

Reclaimer

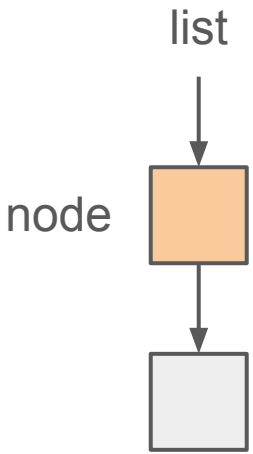
node = pop(list)

time = now()

while start ≤ time ≤ stop { }

free(node)

Spatial schemes: track pointers



Accessor

```
node = load(list)

hazard = node



// Lunch break

sleep(3600)

node.value = 5

hazard = null
```

Reclaimer

```
node = pop(list)

while node == hazard { }

free(node)
```

Arctic proposes semantic scheme: track operations

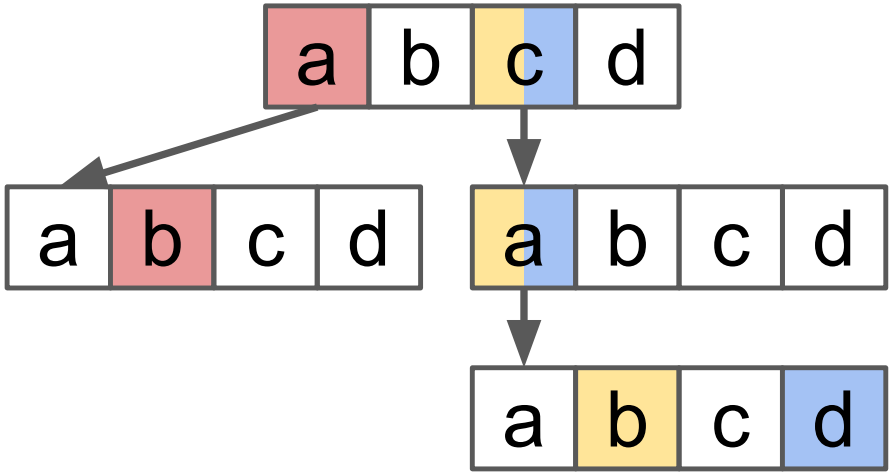
{

ab,

cab,

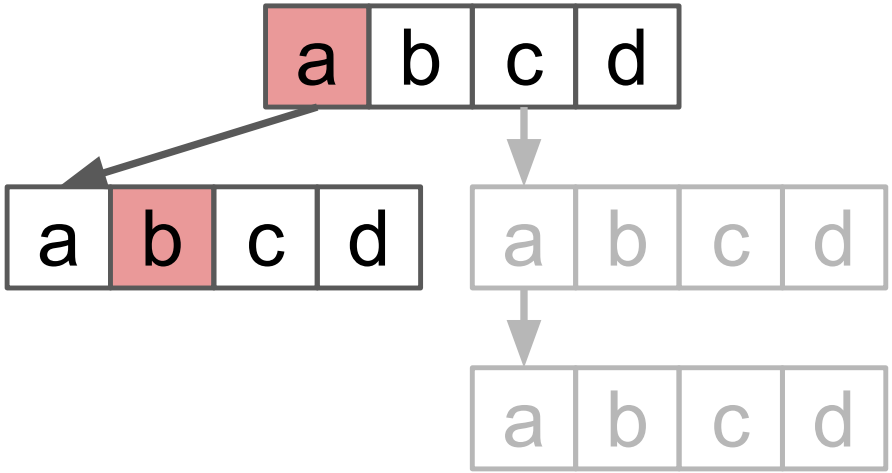
cad,

}



Arctic proposes semantic scheme: track operations

get(**ab**)



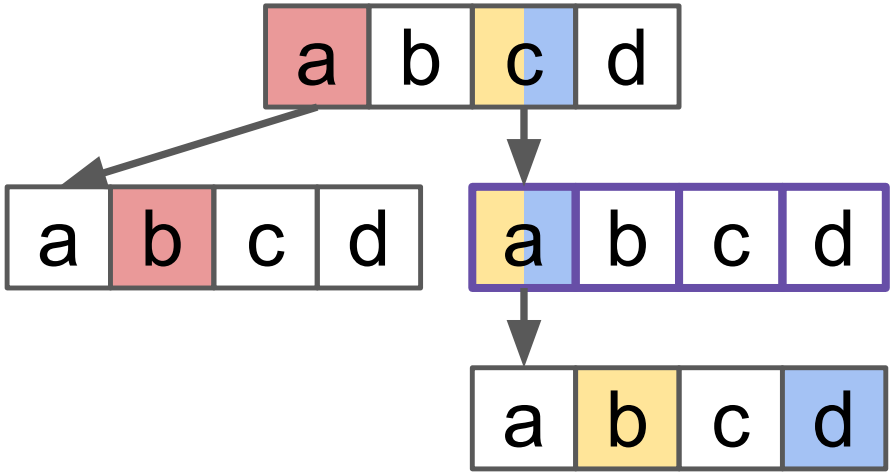
Arctic proposes semantic scheme: track operations

free(**c**)?

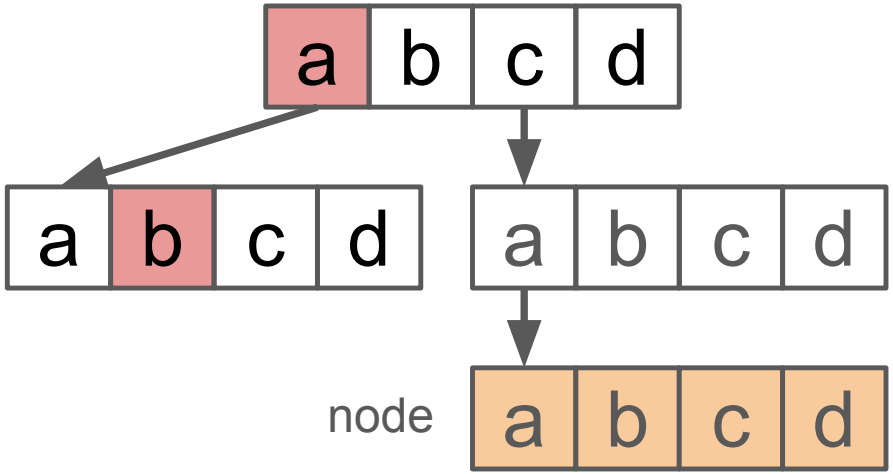
get(**ab**): ✓

get(**cab**): x

get(**cad**): x



Hazard keys



Accessor

hazard = key
get(key)
hazard = null

Reclaimer

node = remove(prefix)
while startswith(hazard, prefix) { }
free(node)

Safe memory reclamation tradeoffs

Temporal (epochs)

+ **Low read overhead (one epoch)**

Spatial (hazard pointers)

- **High read overhead (each pointer)**

Semantic (hazard keys)

+ **Low read overhead (one key)**

Safe memory reclamation tradeoffs

Temporal (epochs)	Spatial (hazard pointers)	Semantic (hazard keys)
+ Low read overhead (one epoch)	- High read overhead (each pointer)	+ Low read overhead (one key)
+ Low reclaim overhead (batch epoch)	**- High reclaim overhead (ptr * hazard ptr)**	**● Med reclaim overhead (ptr * hazard key)**

Safe memory reclamation tradeoffs

Temporal (epochs)	Spatial (hazard pointers)	Semantic (hazard keys)
+ Low read overhead (one epoch)	- High read overhead (each pointer)	+ Low read overhead (one key)
+ Low reclaim overhead (batch epoch)	- High reclaim overhead (ptr * hazard ptr)	● Med reclaim overhead (ptr * hazard key)
- Unbounded memory overhead (thread stall)	**+ Bounded memory overhead**	**- Unbounded memory overhead (high skew)**

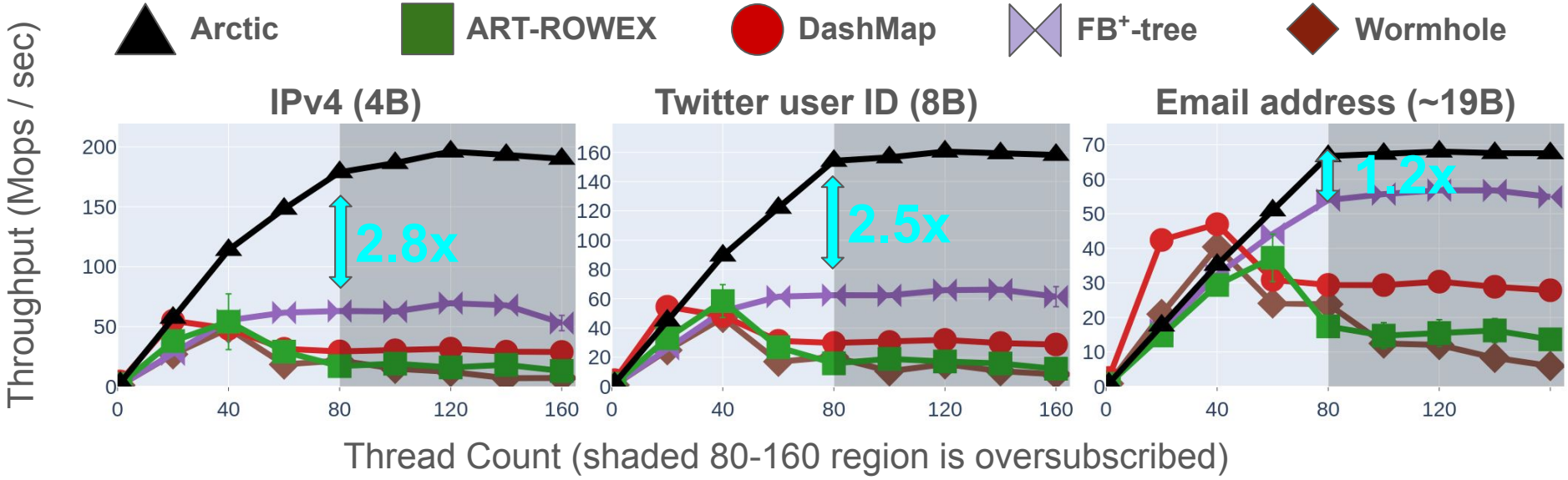
Evaluation

- Chameleon compute_icelake_r650
- **2x** Intel Xeon Platinum 8380
  - 2.30 GHz, **40 cores**, 120 MiB LLC, 128 GiB DDR4 3200 DRAM
- System configuration
  - **Disable HT**, turbo boost, NMI watchdog
  - Pin threads to cores
  - Interleave memory
  - Set CPU scaling governor
- YCSB workload parameters
  - 100M keys
  - 100M operations
  - 10 trials
  - Zipf 0.99 skew
  - 8B integer values

Baselines

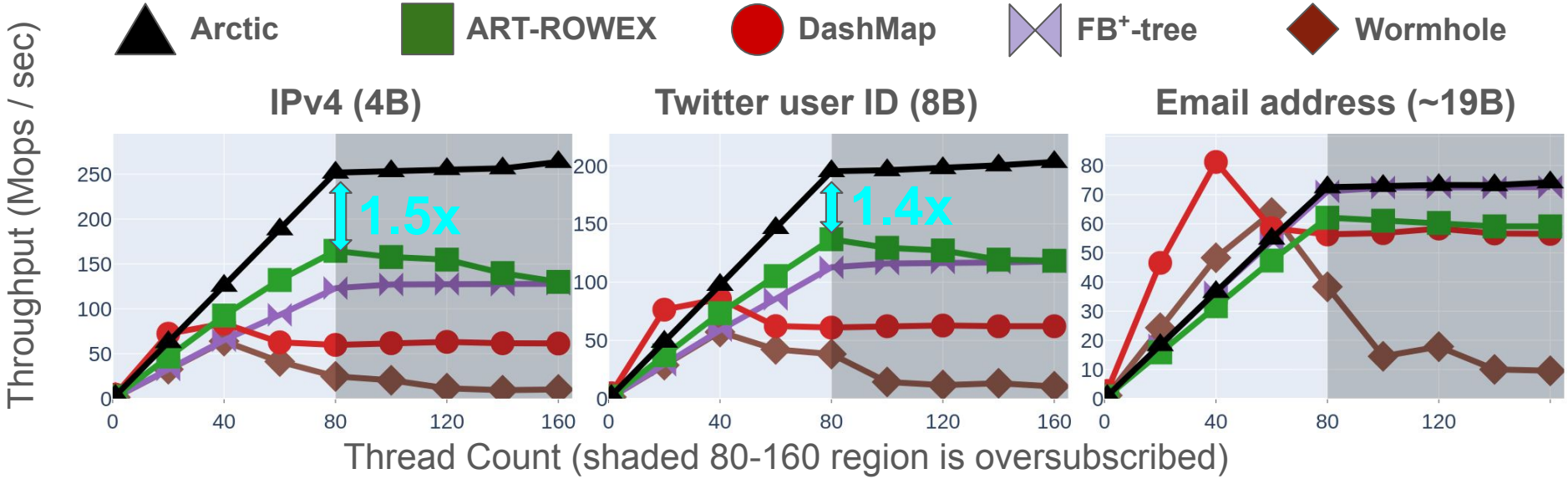
- **ART-ROWEX** (*DaMoN '16*)
  - Adaptive radix tree
  - Optimistic locks with lock-free read
- **DashMap**
  - Hash map
  - Sharded rwlocks
- **Fb$^+$-Tree** (*VLDB '25*)
  - Hybrid B-tree/prefix tree
  - OLC with lock-free update
- **Wormhole** (*EuroSys '19*)
  - Hybrid B-tree/prefix tree/hash map
  - Rwlocks, exclusive spinlocks

YCSB-A: 50% read, 50% update with different keys



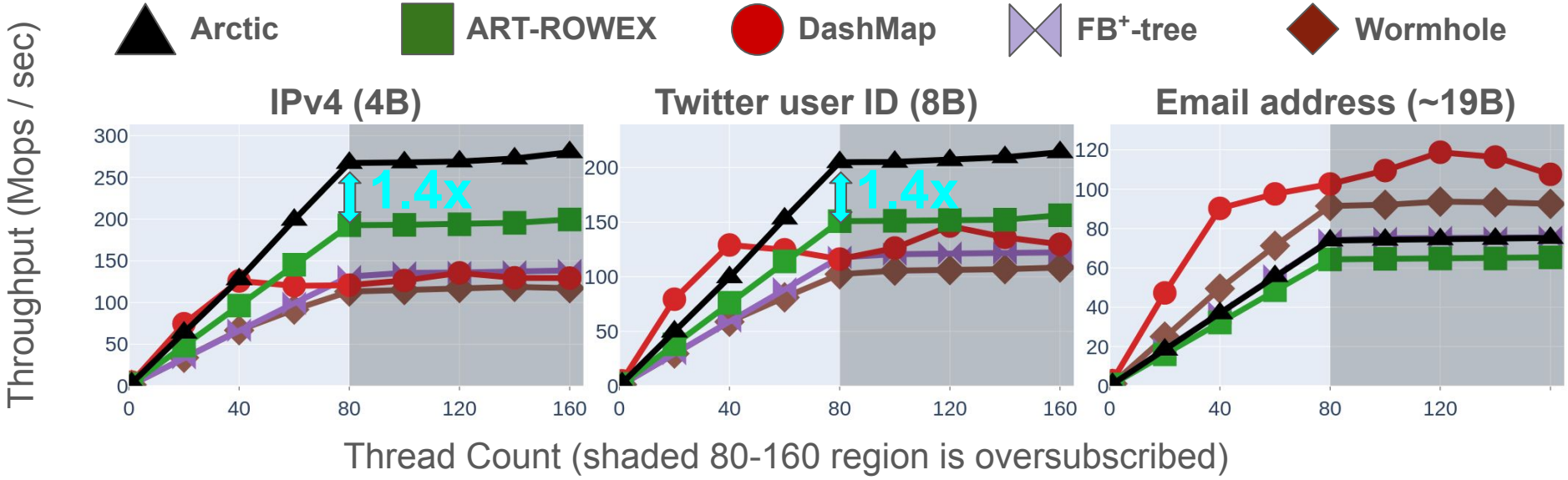
- Arctic dominates for integer keys
- Arctic scales linearly with thread count
- Arctic maintains stable throughput under thread oversubscription (80-160)

YCSB-B: 95% read, 5% update with different keys



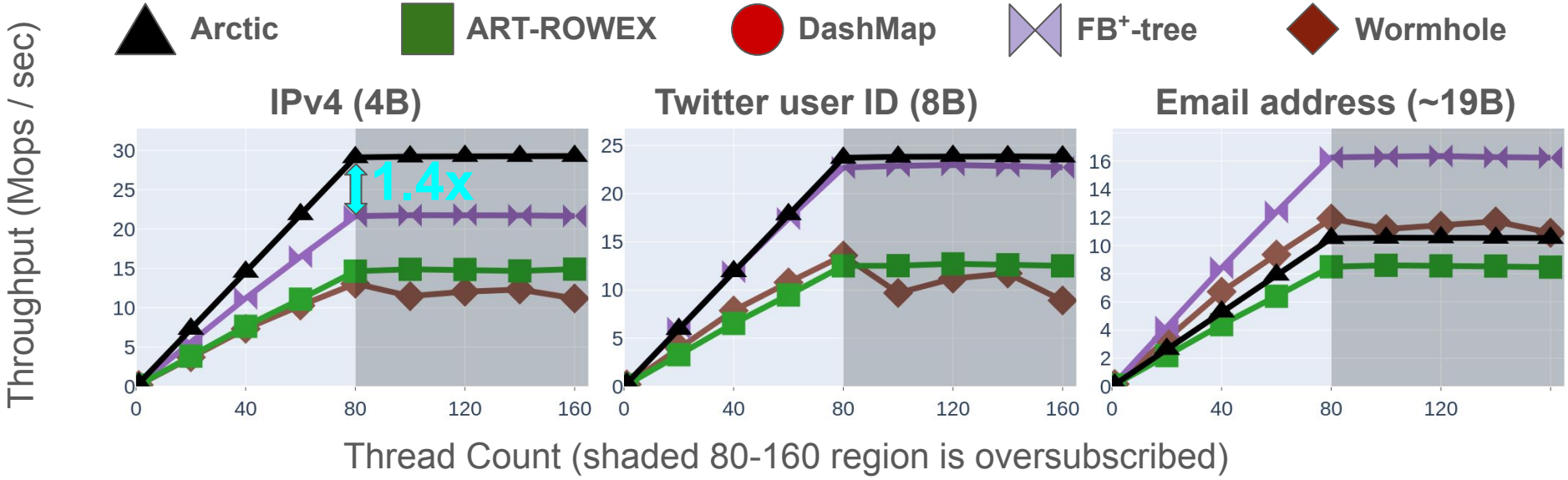
- Arctic's coordination protocol does **not** cost read performance

YCSB-C: 100% read with different keys



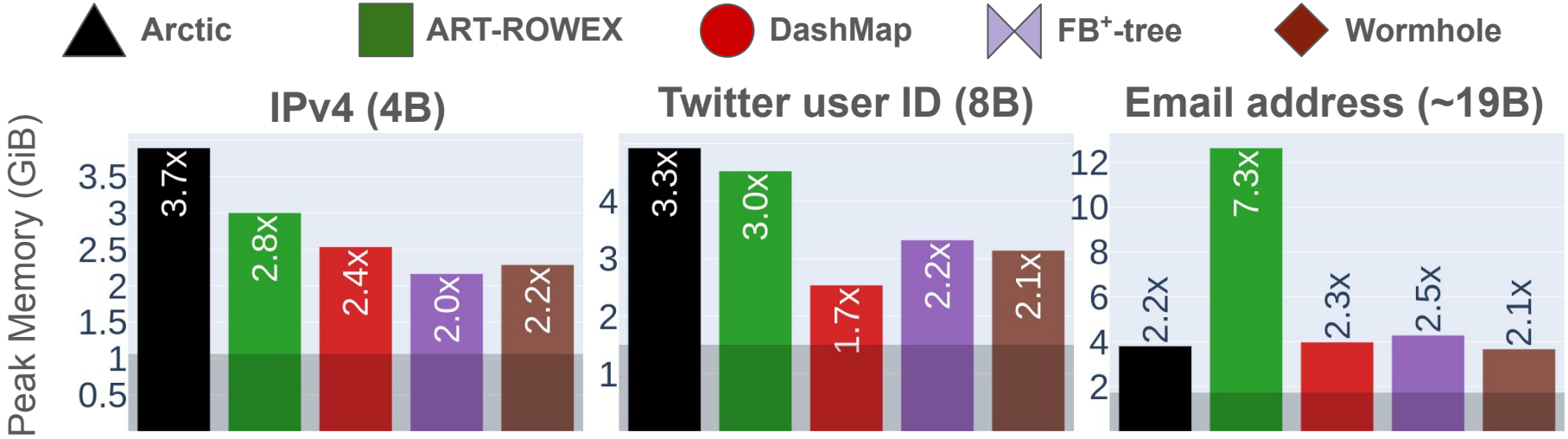
- Arctic's coordination protocol does **not** cost read performance
- DashMap and Wormhole overtake for string keys

YCSB-E: 95% scan, 5% insert with different keys



- For integer keys, Arctic can provide good scan performance
- For string keys, B+-tree sibling pointers win on scans

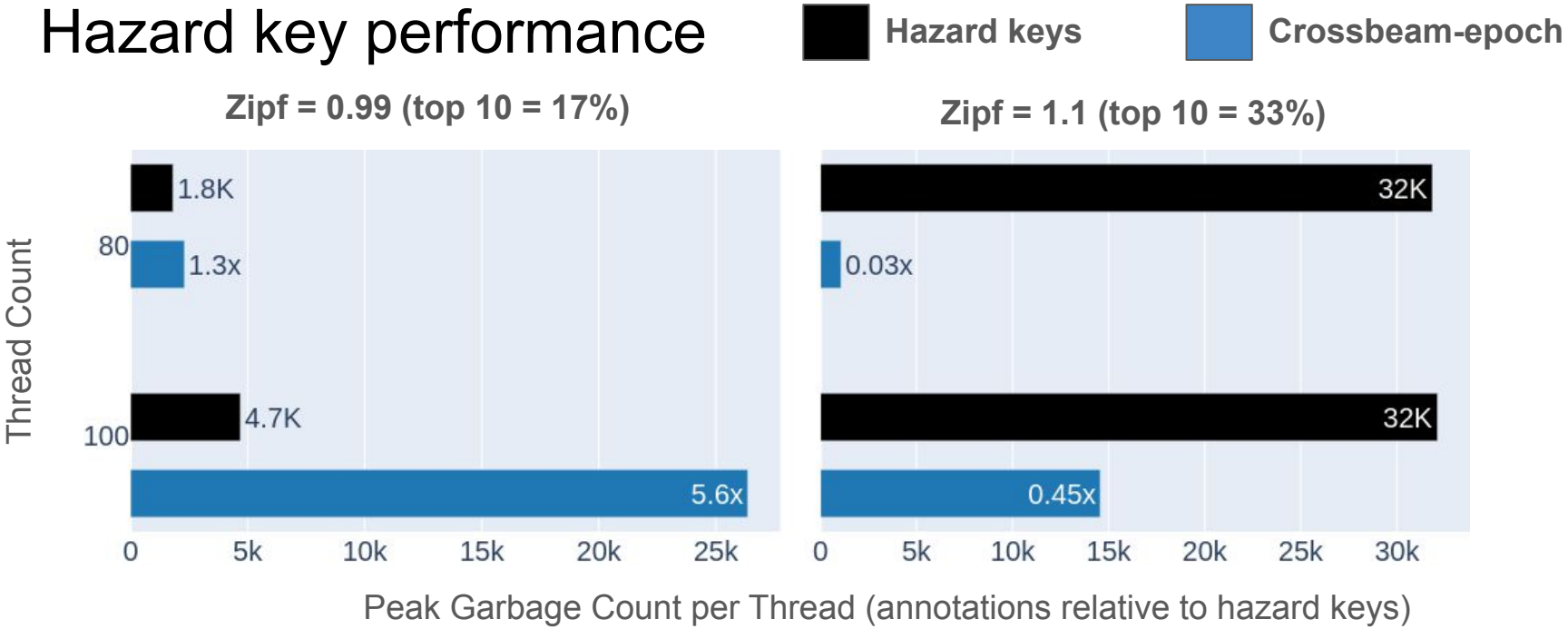
Peak memory usage after loading



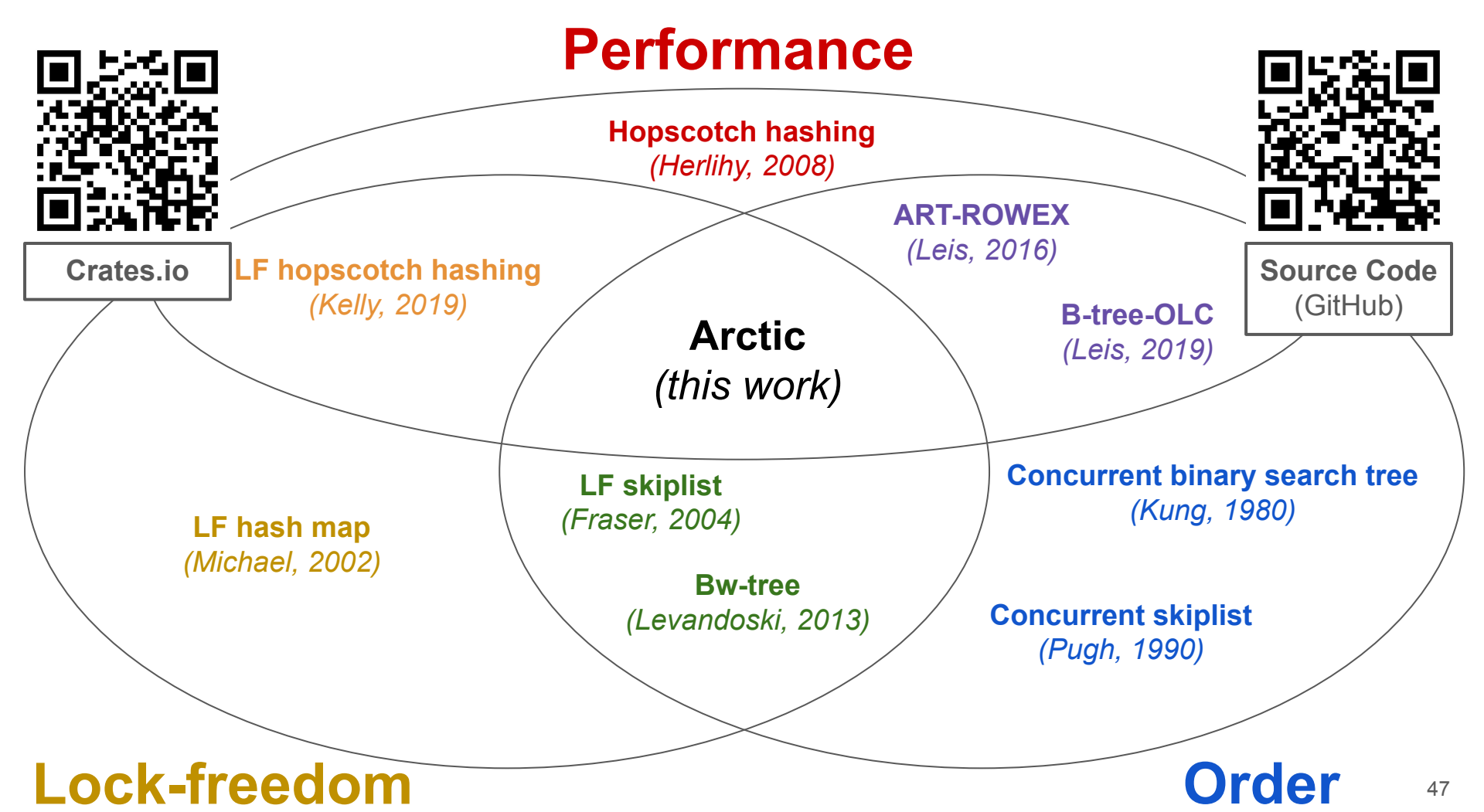
Annotations are relative to baseline size of keys and values (shaded)

- Arctic memory overhead increases with key sparsity
- Memory overhead generally comparable to ART-ROWEX

Hazard key performance



- SMR-intensive: YCSB-A, u64 keys, allocated u64 values, vary **thread count** and **skew**
- Throughput under hazard keys within ~10% under epochs
- Hazard keys are efficient at low skew (Zipf = 0.99)
- Epochs are efficient when not oversubscribed (thread count = 80)


Performance
Hopscotch hashing
(Herlihy, 2008)
ART-ROWEX
(Leis, 2016)
Crates.io
LF hopscotch hashing
(Kelly, 2019)
Source Code
(GitHub)
B-tree-OLC
(Leis, 2019)
Arctic
(this work)
LF skiplist
(Fraser, 2004)
Concurrent binary search tree
(Kung, 1980)
LF hash map
(Michael, 2002)
Bw-tree
(Levandoski, 2013)
Concurrent skiplist
(Pugh, 1990)
Lock-freedom
Order