Cache Craftiness for Fast Multicore Key-Value Storage

Yandong Mao, et al. EuroSys'12

2024. 02. 14

Presentation by Yeojin Oh, ZhuYongjie, Boseung Kim

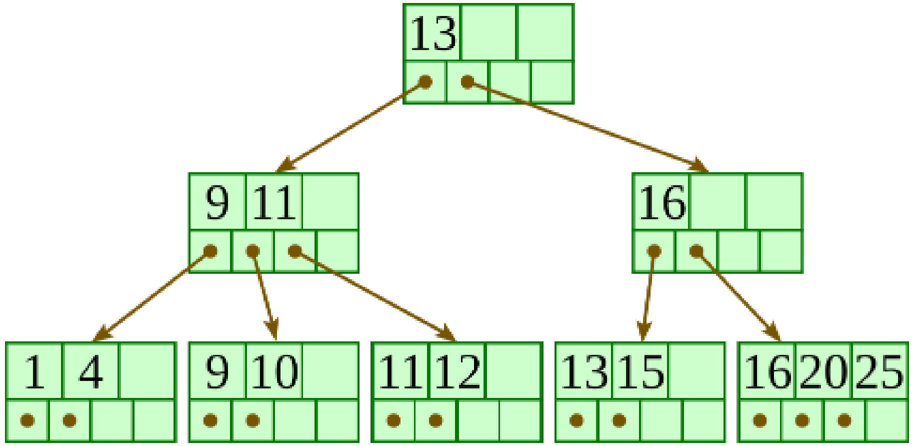
yeojinoh@dankook.ac.kr, aeashio1111@dankook.ac.kr, bskim1102@dankook.ac.kr

Contents

Introduction

- A flexible storage model

  - B+ Tree

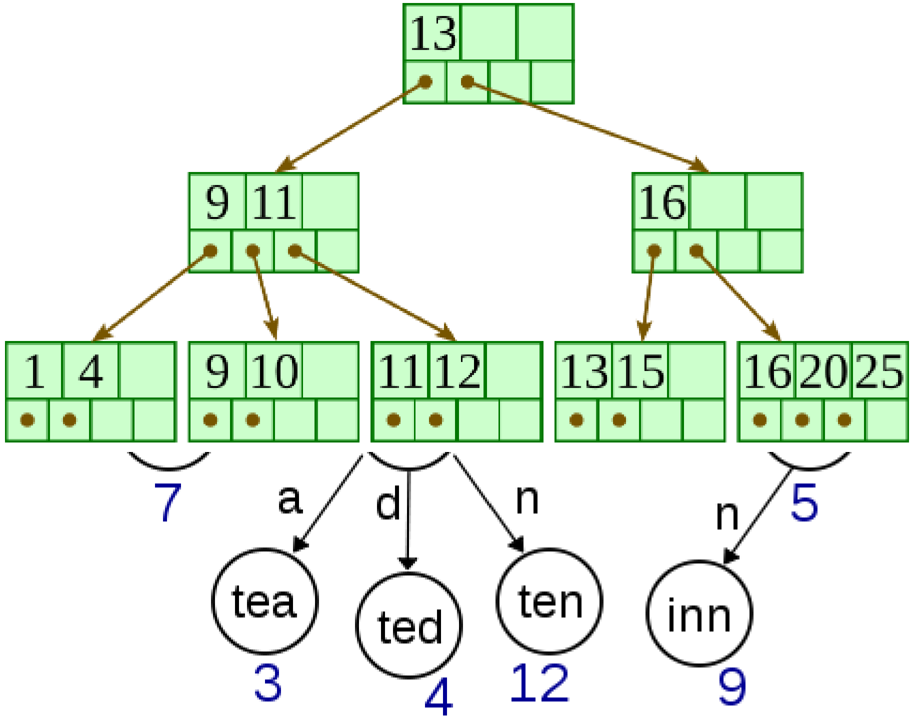


B+tree have excellent performance in managing disk IO and high cache utilization capabilities.

But they require more effort for multi-threaded concurrency, which can also lead to increased management difficulty.

Masstree

- A flexible storage model
  - Masstree --- Attempt to inherit the advantages of B+tree while optimizing its drawbacks



Masstree is composed of multiple B+trees to form a trie.

Masstree

- A flexible storage model
  - Masstree --- Attempt to inherit the advantages of B+tree while optimizing its drawbacks

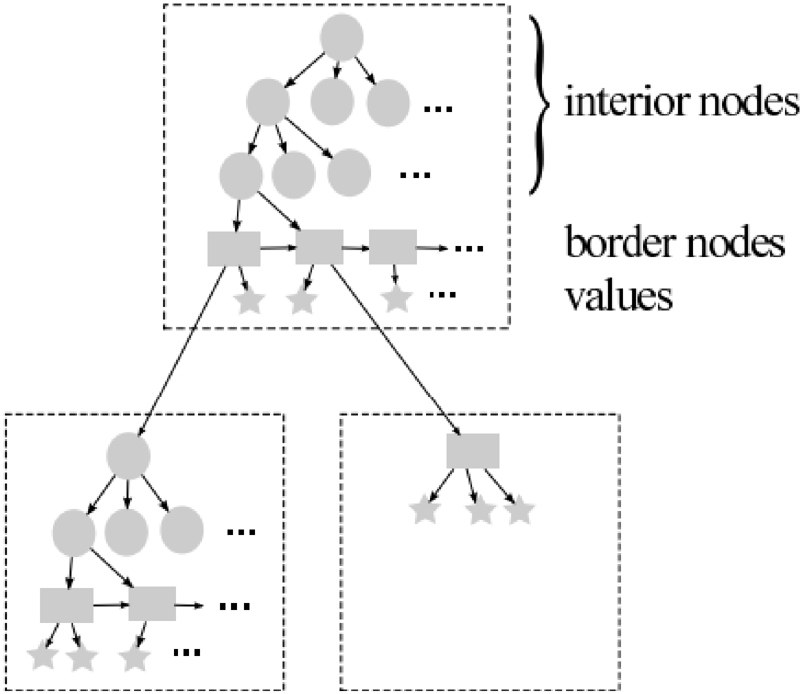
Layout considering cache efficiency

Non-cache-polluting lookup

Concurrent non-blocking lookup

Non-blocking insert

Persistent



Masstree

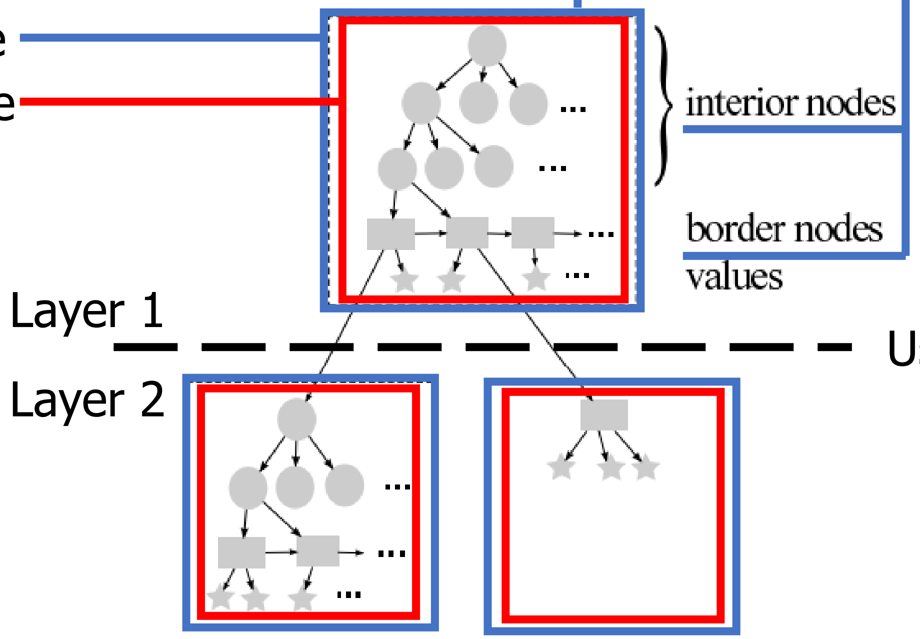
- Overview

Fanout = 2^{64} --- to build a flat B+Tree

Like B+Tree.
But border nodes also pointers leading to the next layer of trie nodes

Trie node

B+ Tree



0~7 byte

Using the length of the key for layering

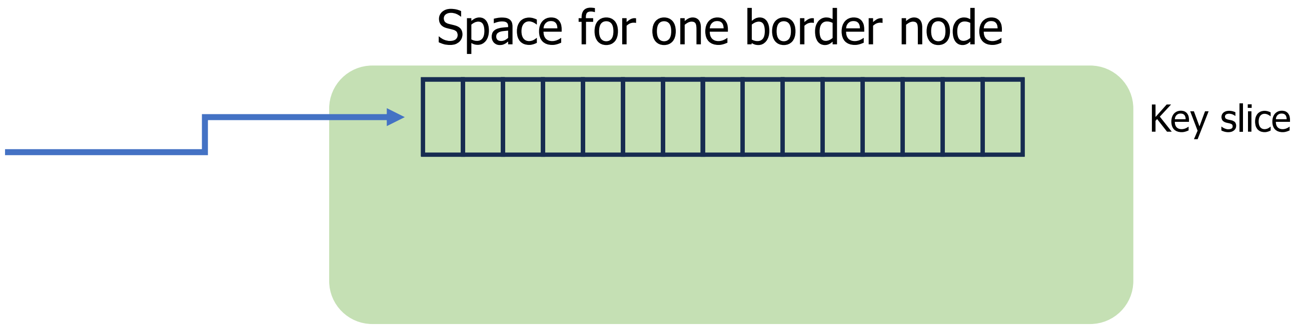
8~15 byte

This construction can achieve that all keys contained in the leaf node of trie have the same prefix

Masstree

- Layout

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

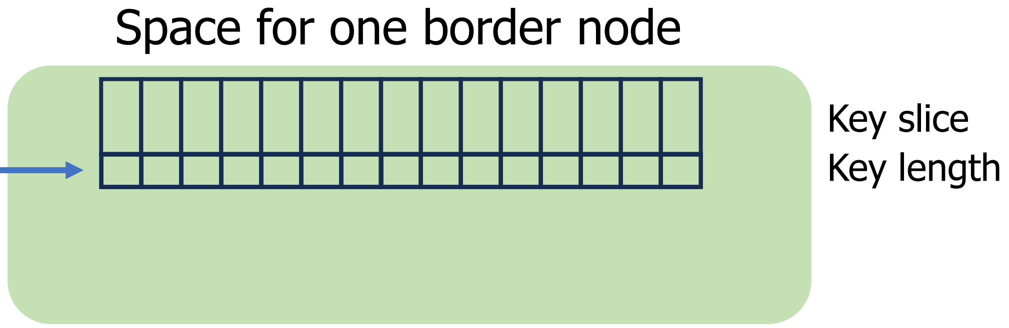


15 key slices (prefixes) with a length of 8 bytes form a border node

Masstree

- Layout

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

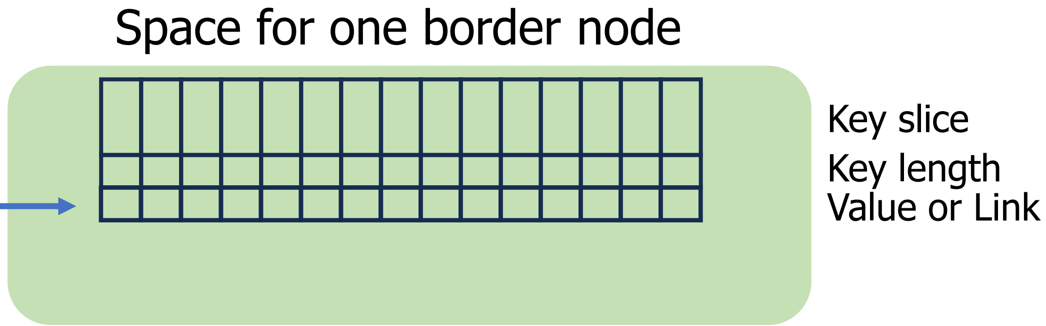


The total length of 15 1-byte array record keys

Masstree

- Layout

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

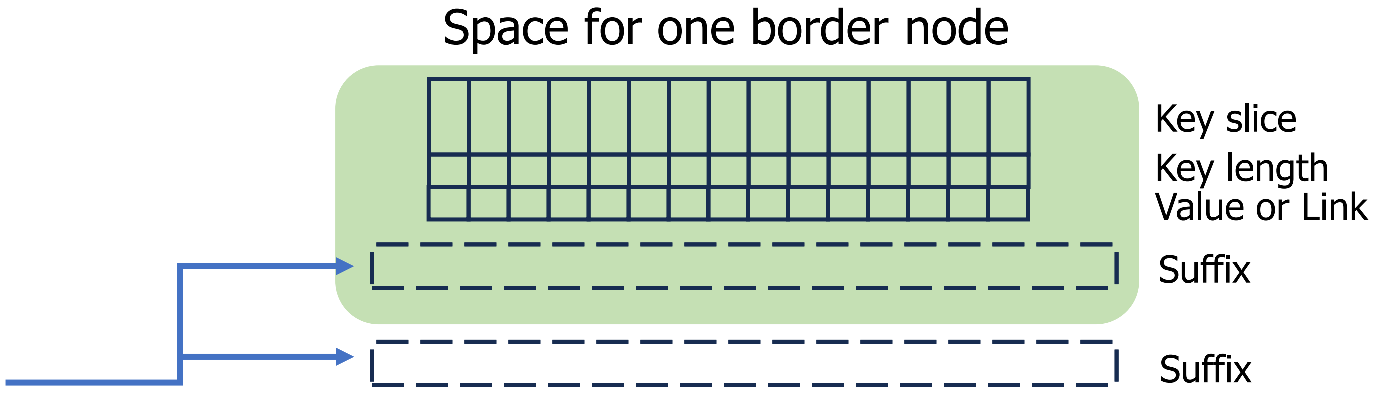


A list consisting of 15 elements is used to record whether the corresponding ke
y slice is an accurate key or a point to the next layer.

Masstree

- Layout

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

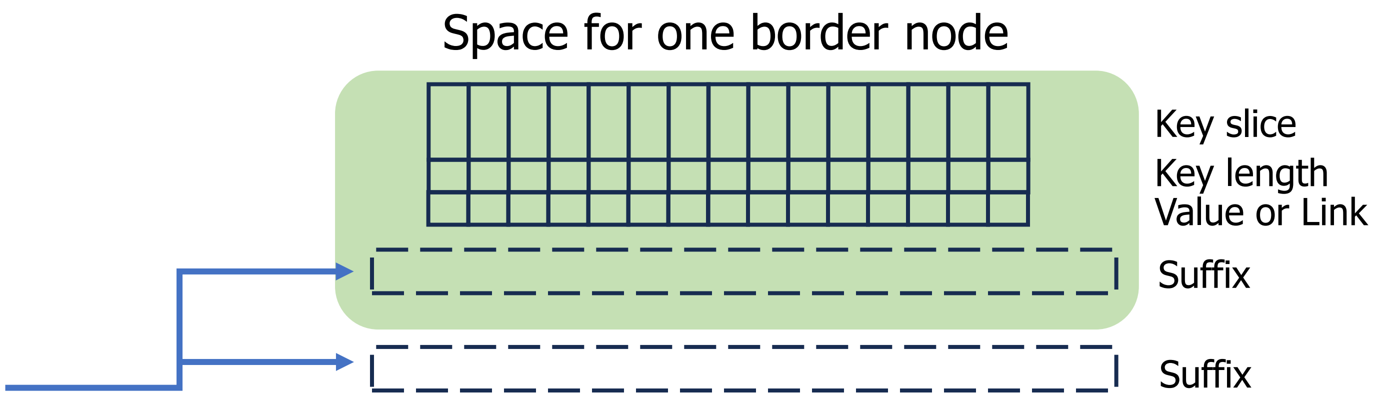


When the length of a key exceeds the processing length of this layer,
but there are not enough keys with the same prefix to construct the next layer's node,
it is responsible for storing the excess processing length.

Masstree

- Layout

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

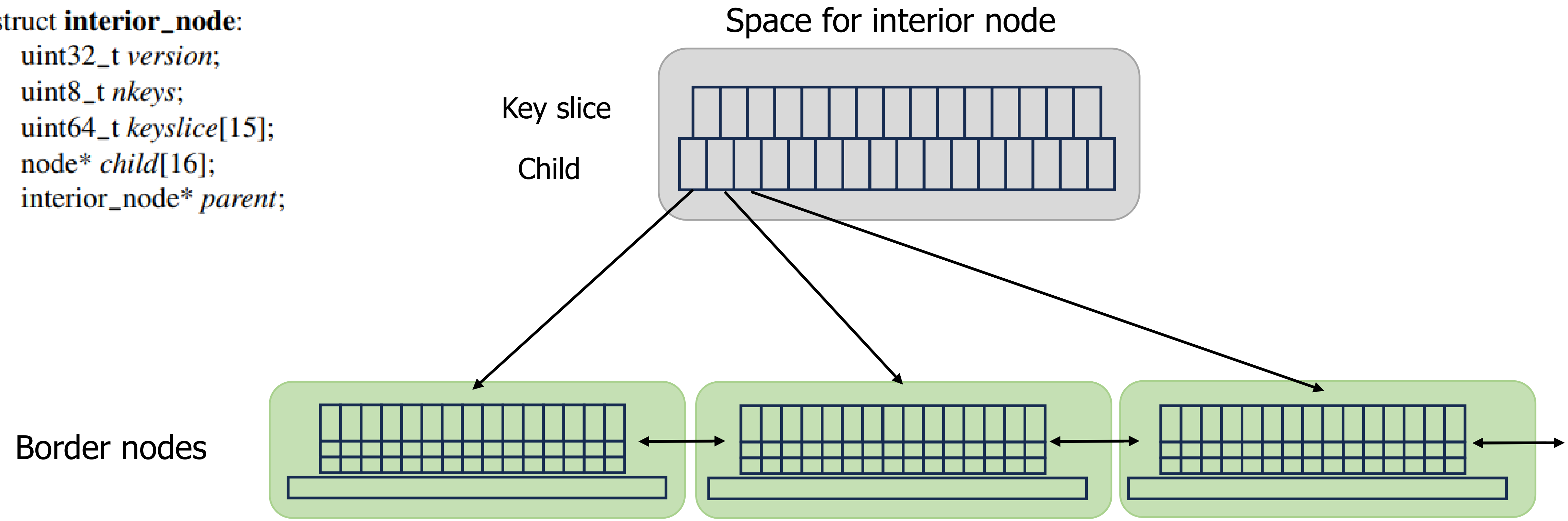


The size and storage location of keysuffixes will be selected to be stored in the border node structure or in an independent memory space according to the situation.

Masstree

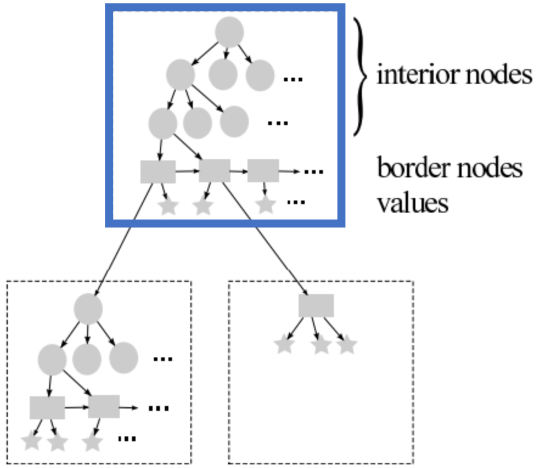
- Layout

```
struct interior_node:
    uint32_t version;
    uint8_t nkeys;
    uint64_t keyslice[15];
    node* child[16];
    interior_node* parent;
```



Masstree

- Layout

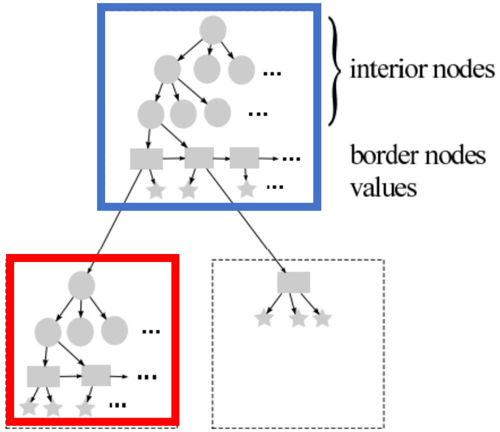


Key slice	01234567			...	
Key length	10				
Value or Link	0(value)				
Suffix	AB				

Border node in trie's root node

Masstree

- Layout
  - Put a key as “01234567XY”



				…	
Key slice	01234567			…	
Key length	10				
Value or Link	1(Link)				
Suffix					

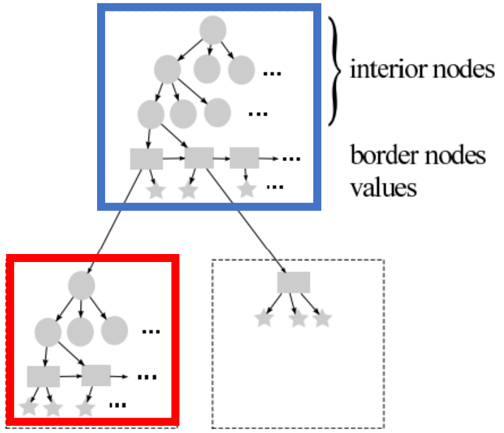
Layer 0

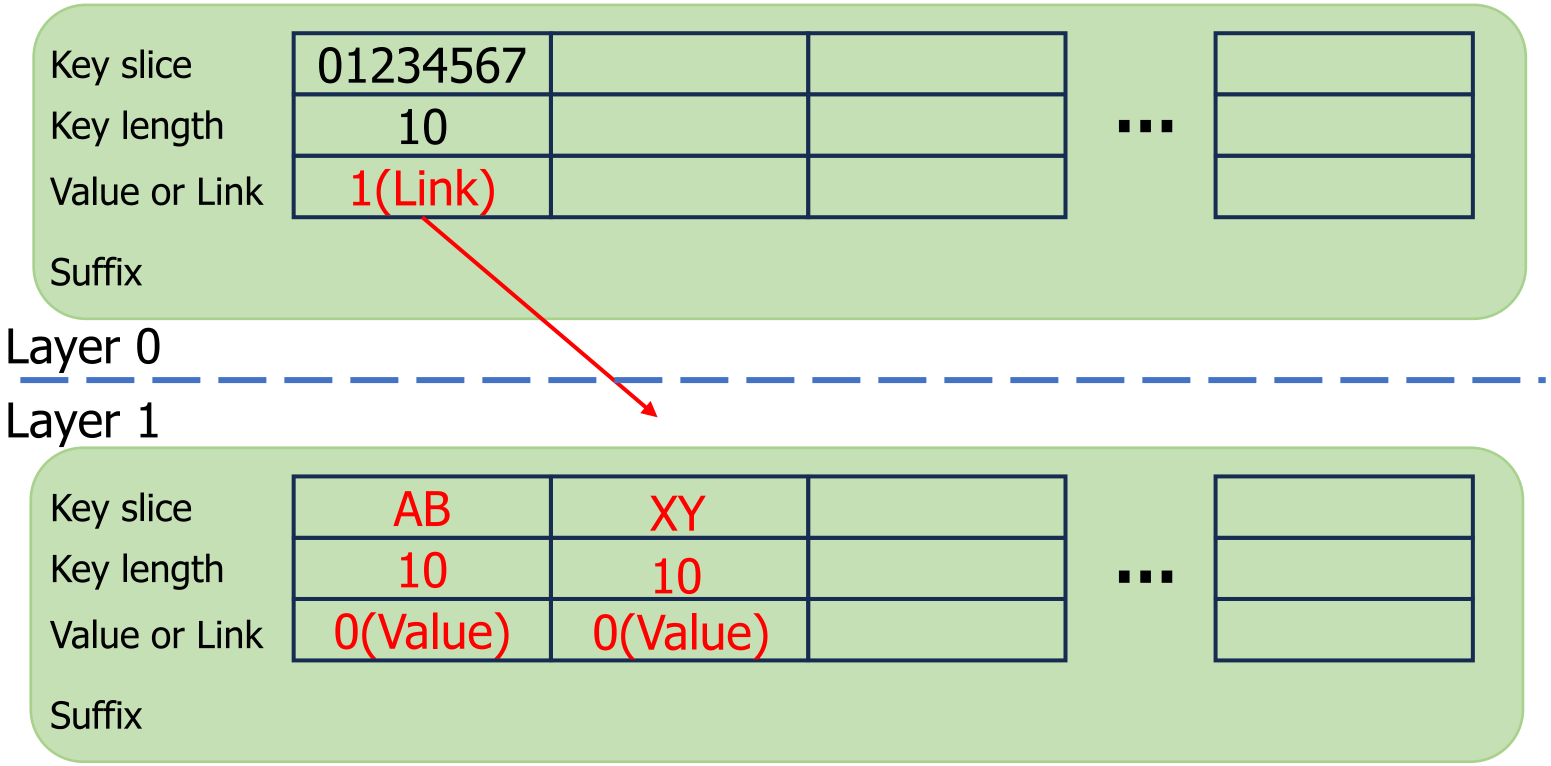
Layer 1

				…	
Key slice	AB	XY		…	
Key length	10	10			
Value or Link	0(Value)	0(Value)			
Suffix					

Masstree

- Layout
  - Remove a key as “01234567XY”





Masstree

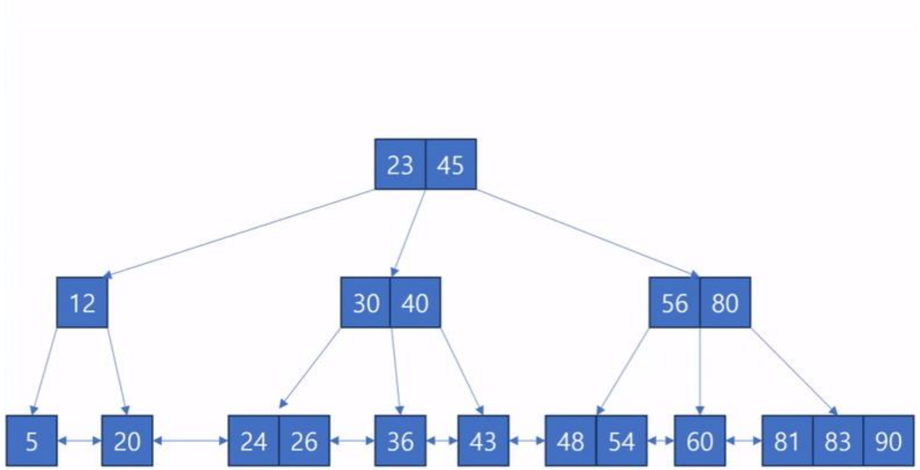
- Layout
  - Other things

    - Masstree must be able to distinguish between 8-byte key “0123456□n" and 7-byte key "0123456" with empty characters.
    - When there are too many keys with the same prefix, Masstree allows for the construction of up to 10 keys with the same prefix within the node layer above the stored key
    - The key suffix data structure will be stored inside or outside the trie node depending on the situation, and its size is variable.
    - About uint64_t keyslice[15] :

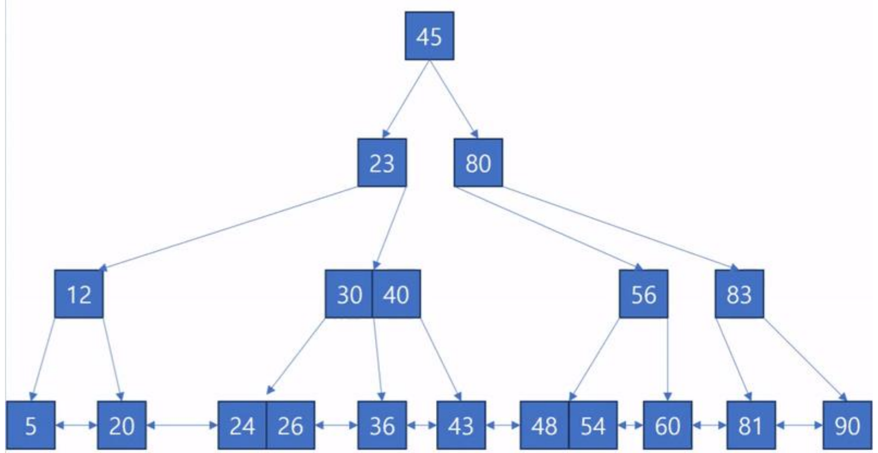
      Conduce to L3 cache to read an entire node at once.

Masstree

- Nonconcurrent modification



Insert

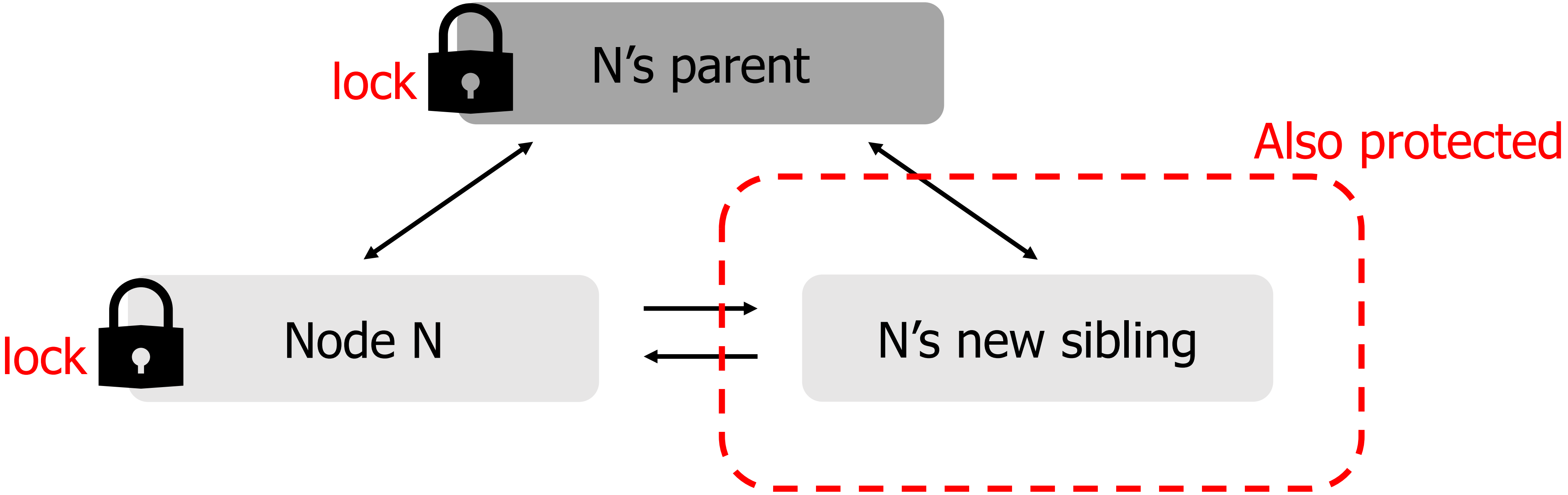


Remove

The B+Tree constructed by Masstree also has issues such as node splitting during insert and remove operations, and provides solutions accordingly

Concurrency

- Writer-Writer coordination
  - Splits and node deletions require a writer to hold several locks simultaneously



Concurrency

- Writer-Reader coordination

  Use ***version***

Node **k**

uint32_t ***version***

<before change>

Node **k**

uint32_t ***version***

dirty

<at change>

Node **k**

uint32_t ***version***

version counter++

<after change>

Concurrency

- Writer-Reader coordination
  - update

- Update operation must prevent concurrent readers from observing intermediate result
- Read-copy update technology ➔ Collect and process at once

Concurrency

- Writer-Reader coordination
  - Border insert

0	3	7								63
nkey	Keyindex [0]	Keyindex [1]	Keyindex [2]	Keyindex [3]	Keyindex [4]		…	…	…	Keyindex [15]

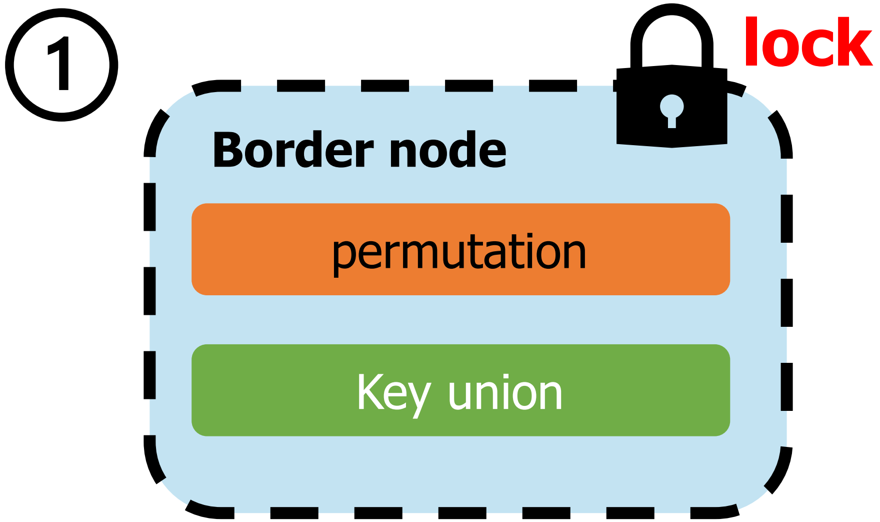
uint64_t ***permutation***

- nkey: Current number of key
- keyindex: Correct key order → store the indexes of the border node's live keys

Unlive key
➔ List currently-unused slots

Concurrency

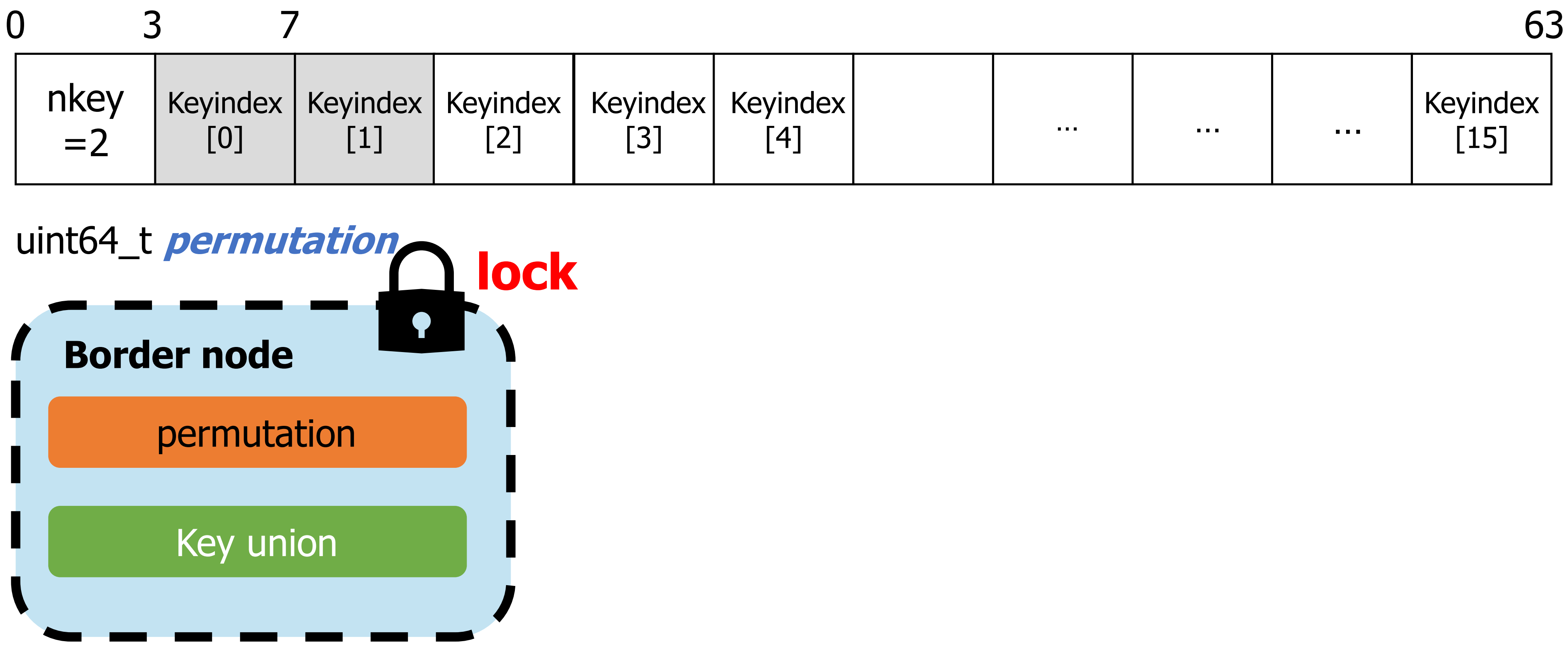
- Writer-Reader coordination
  - Border insert



Concurrency

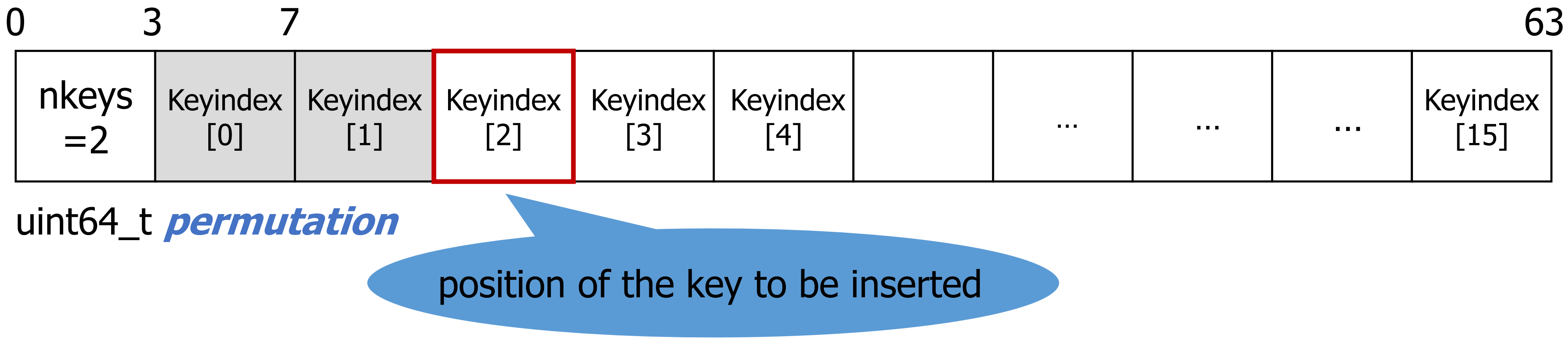
- Writer-Reader coordination
  - Border insert

② Load the permutation



Concurrency

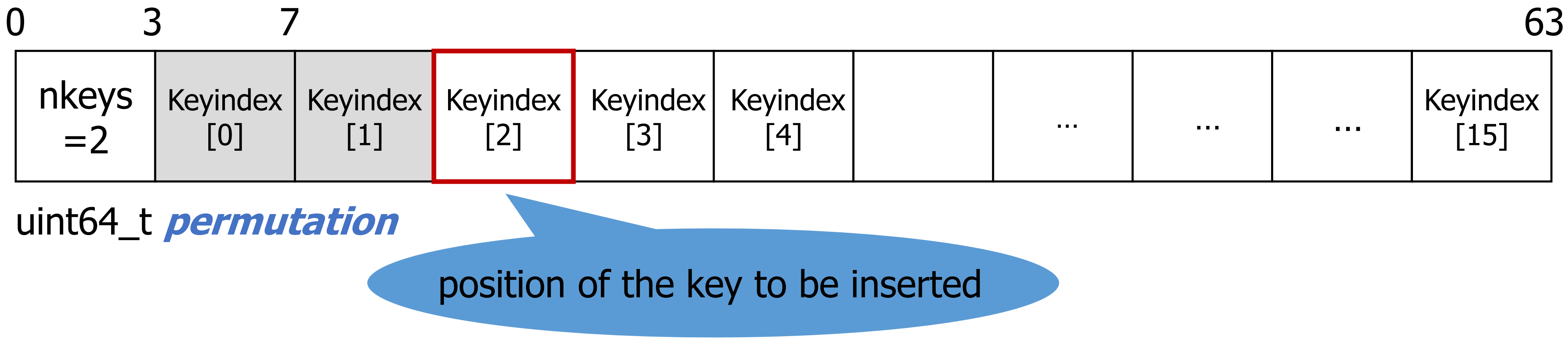
- Writer-Reader coordination
  - Border insert



3
- **Rearranges** the permutation to shift an unused slot to the correct insertion position
- increment nkeys

Concurrency

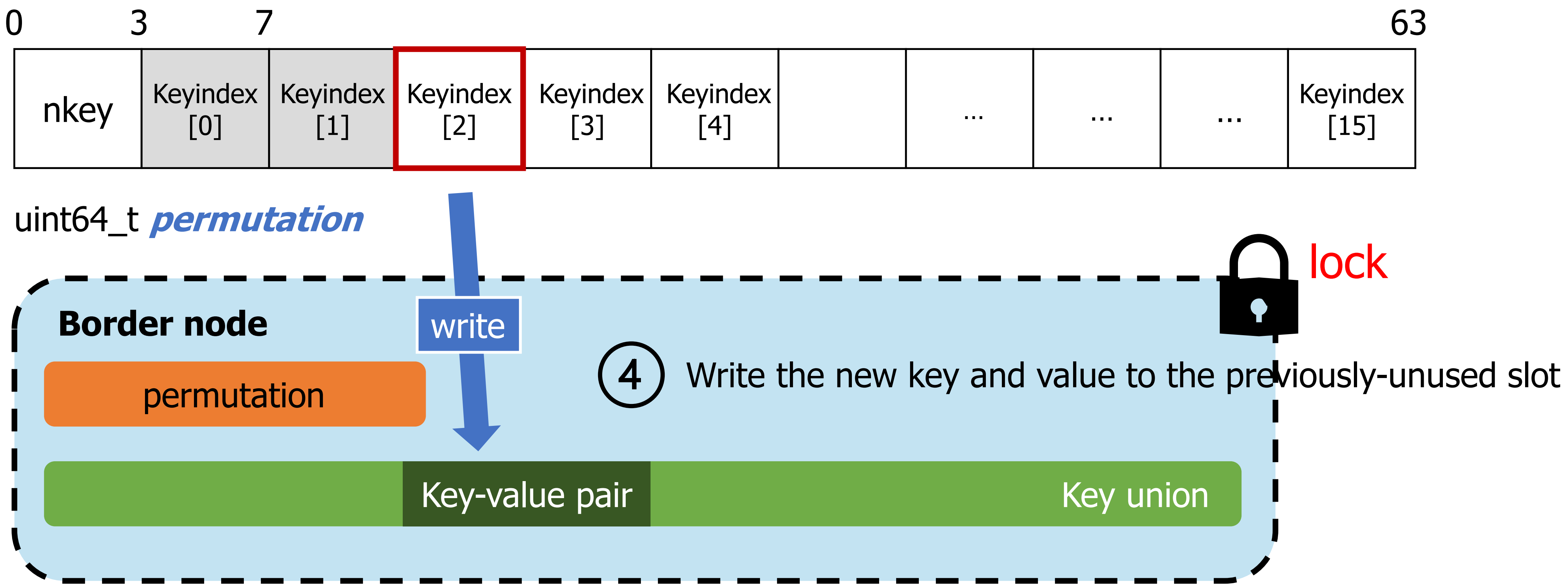
- Writer-Reader coordination
  - Border insert



3
- Rearranges the permutation to shift an unused slot to the correct insertion position
- **increment** nkeys

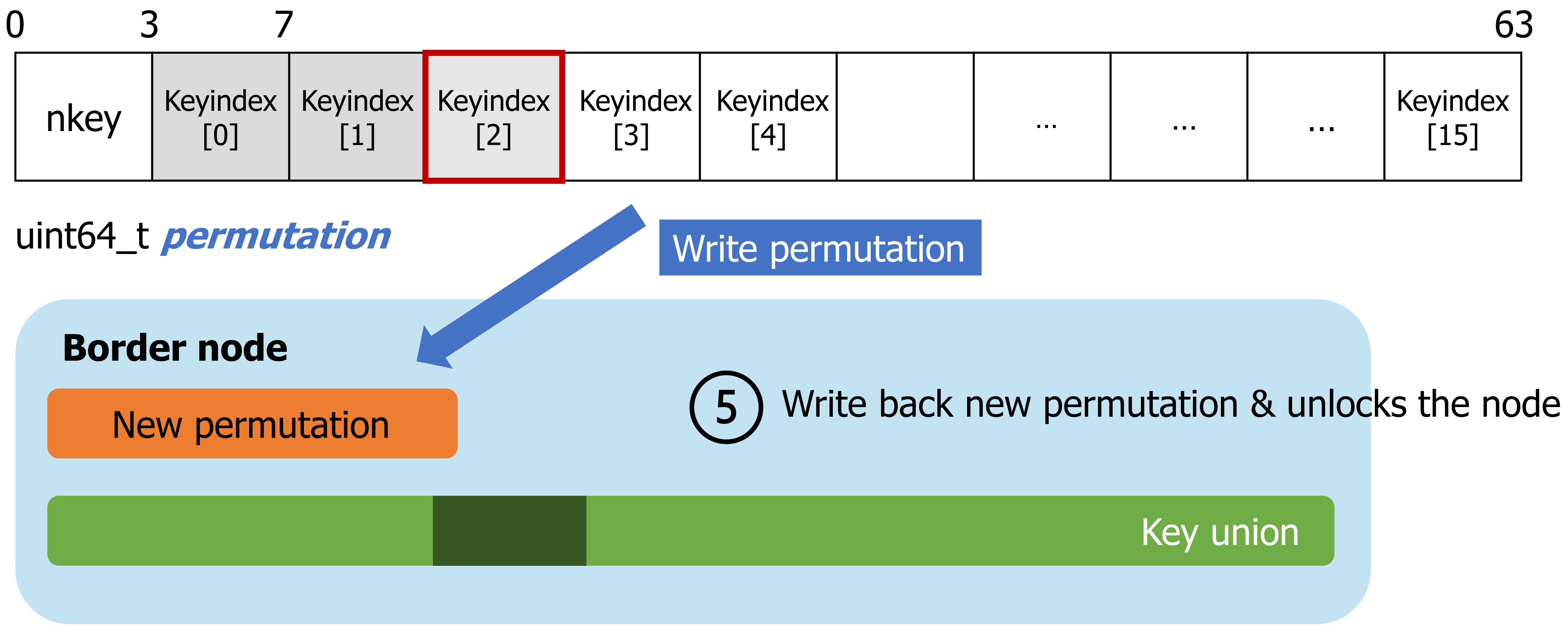
Concurrency

- Writer-Reader coordination
  - Border insert



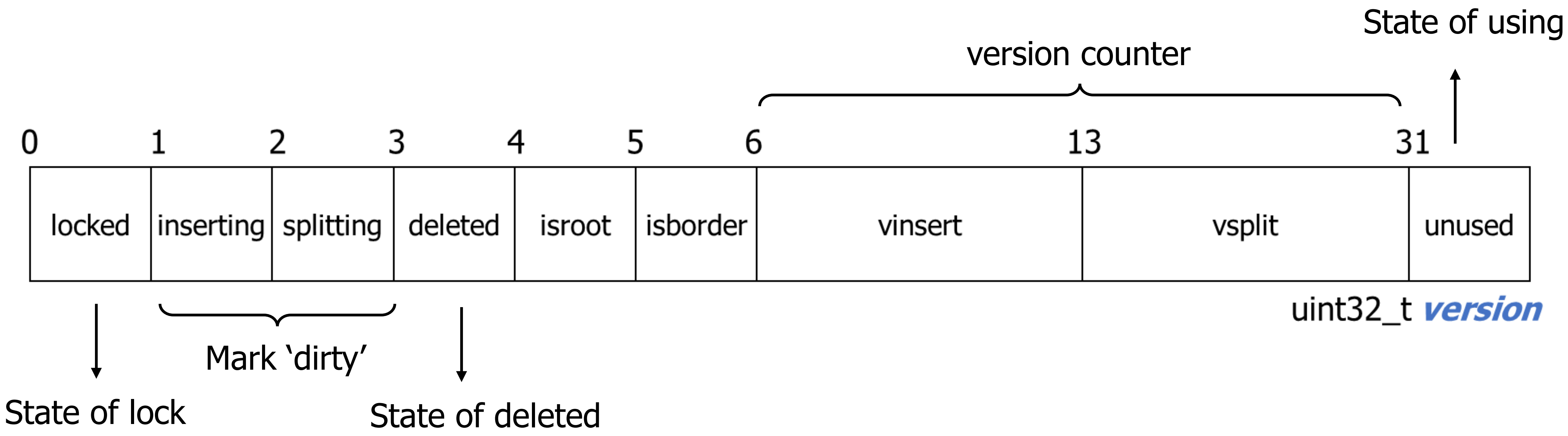
Concurrency

- Writer-Reader coordination
  - Border insert



Concurrency

- Writer-Reader coordination
  - Use hand-over-hand locking

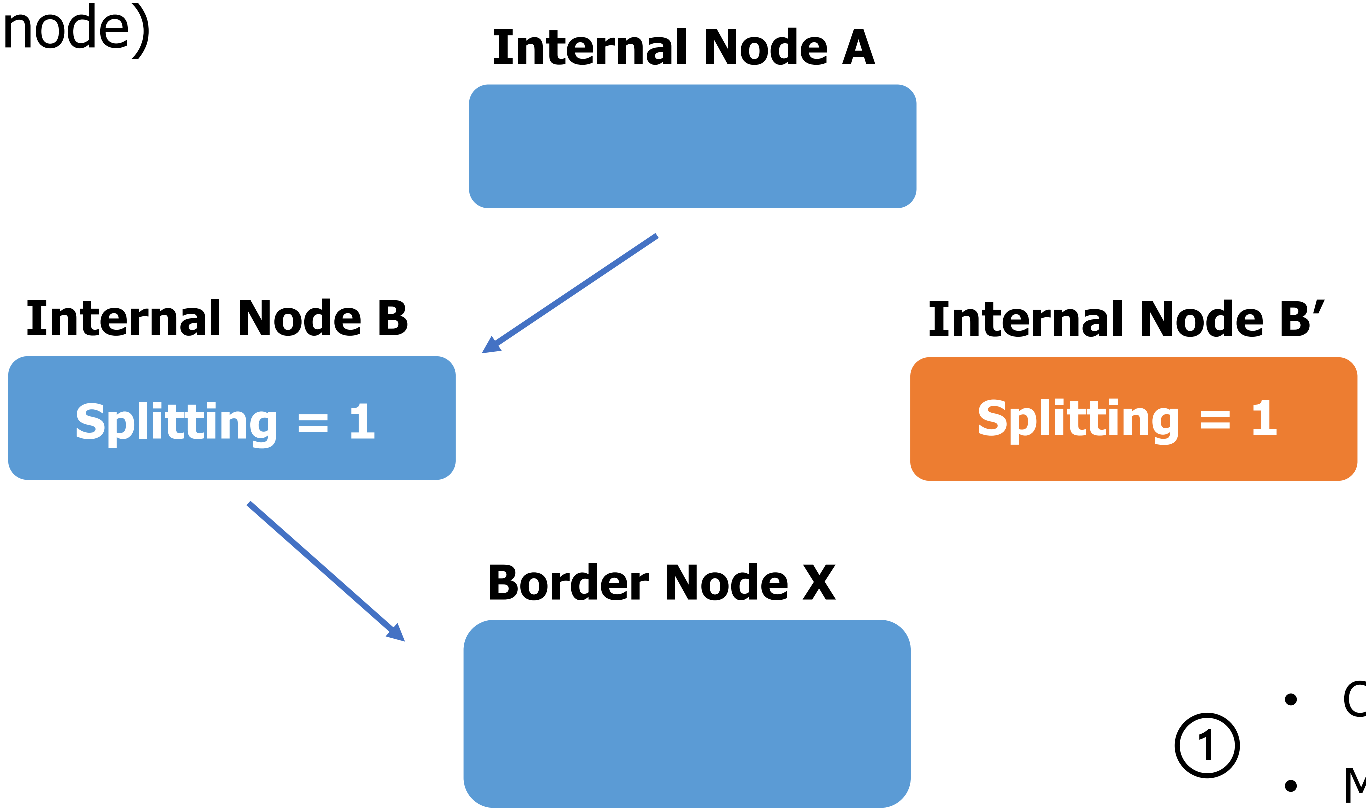


Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (internal node)



1

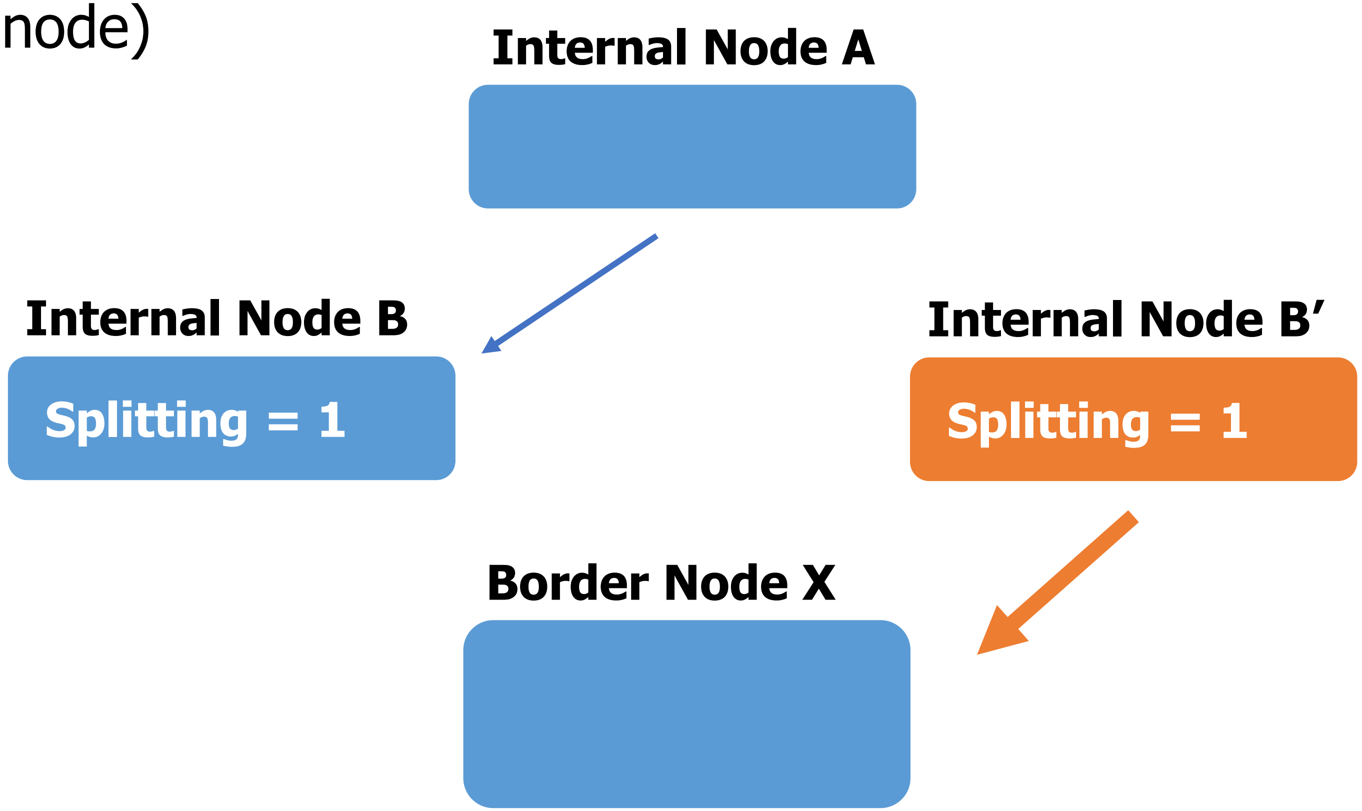
- Create new node B’
- Marking splitting to B & B’

Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (internal node)



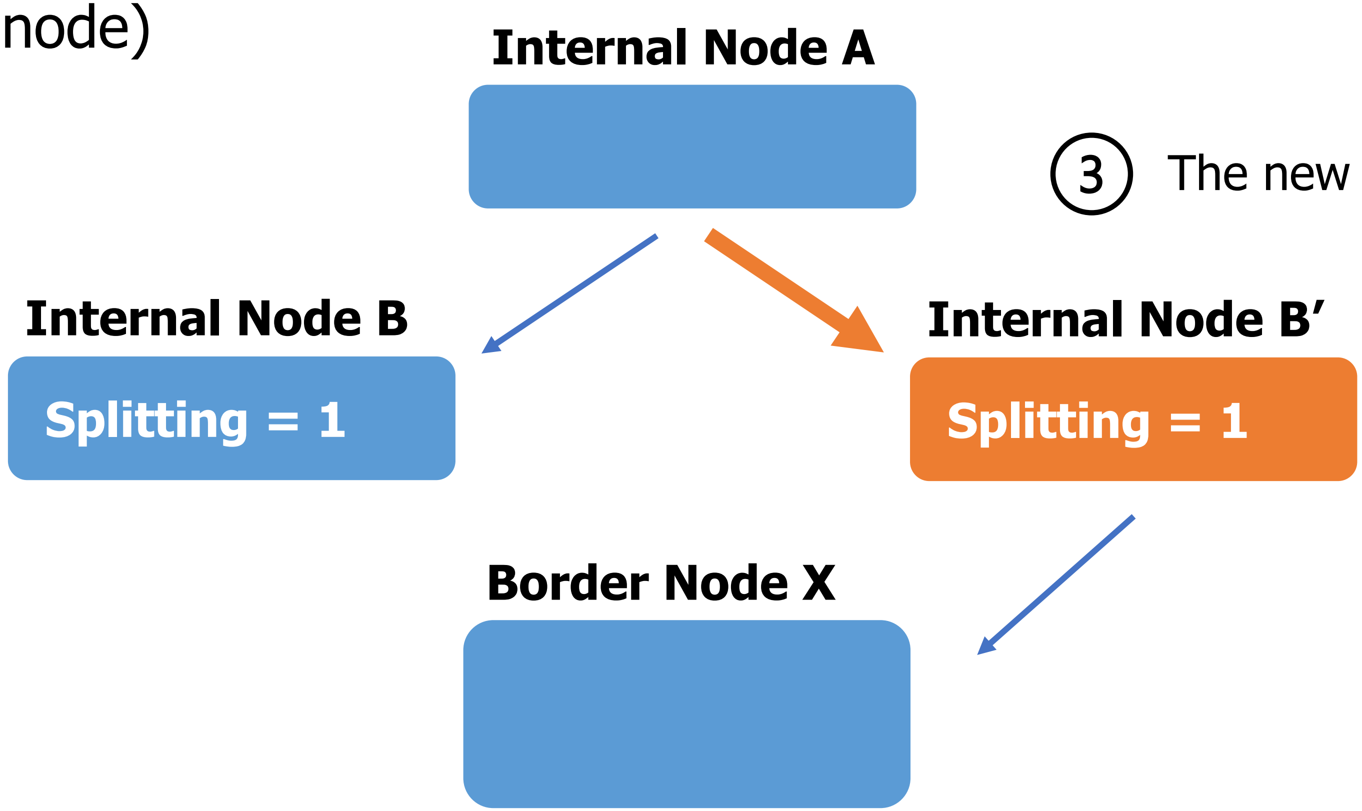
② Children (including X) are shifted from B to B’

Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (internal node)



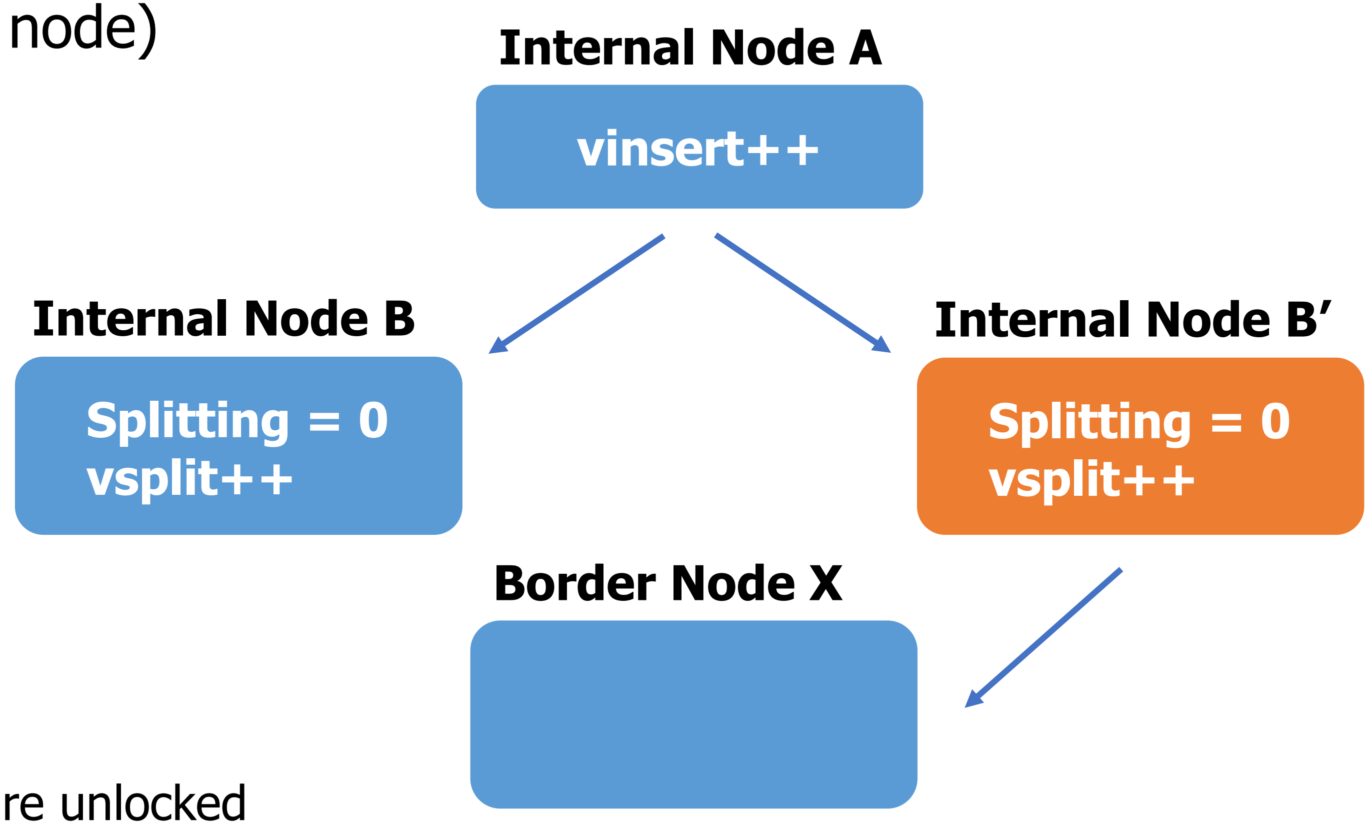
3 The new node B’ is inserted into A

Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (internal node)



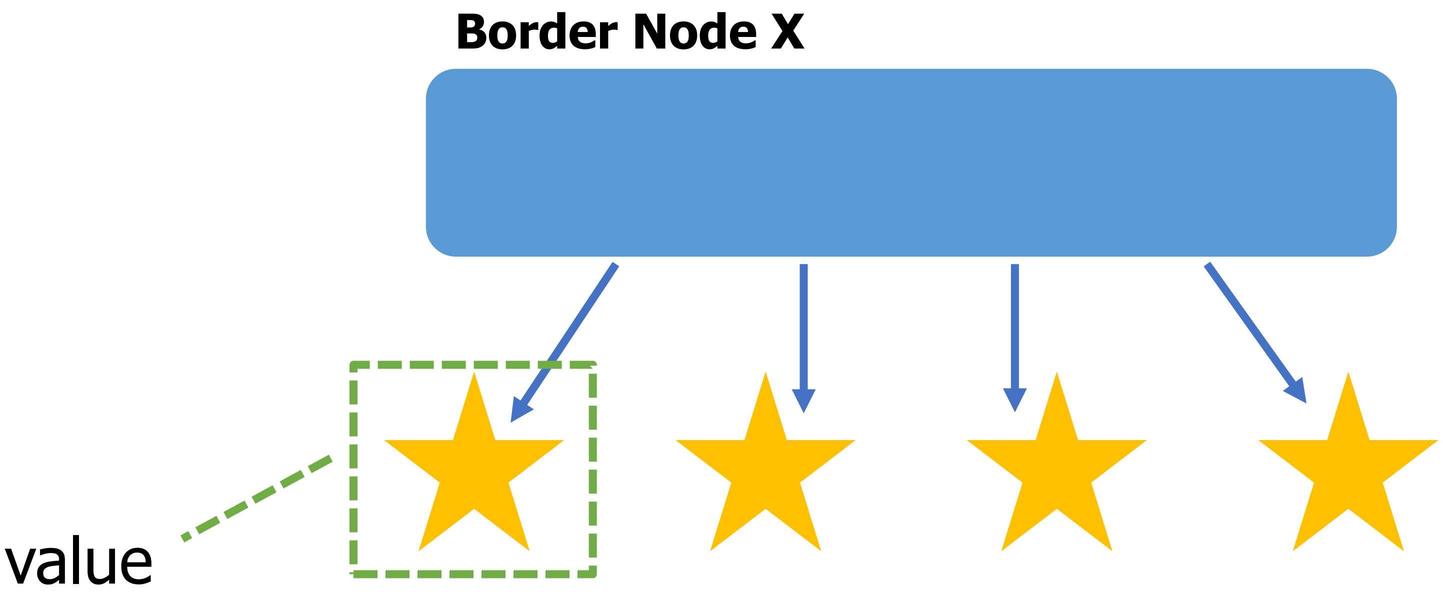
4
- A,B, and B' are unlocked
- Increments the A vinsert counter and the B and B' vsplit counters

Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (border node)



Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

- Writer-Reader coordination
  - Split (border node)

Border Node X

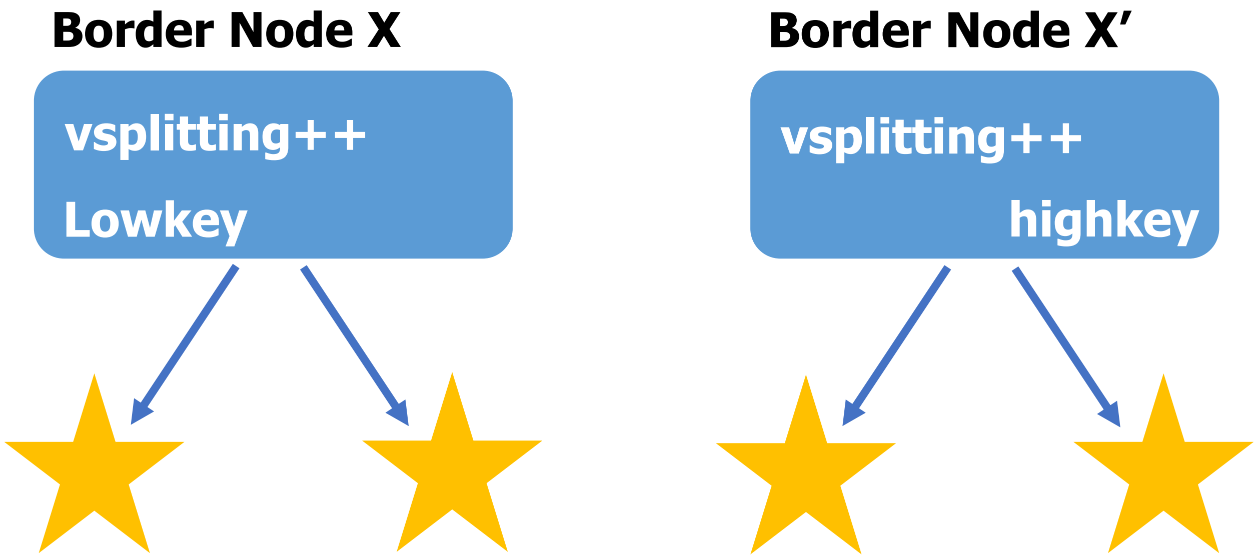
Range of key : [lowkey(n),highkey(n))

Concurrency

0	1	2	3	4	5	6	13	31
locked	inserting	splitting	deleted	isroot	isborder	vinsert	vsplit	unused

uint32_t *version*

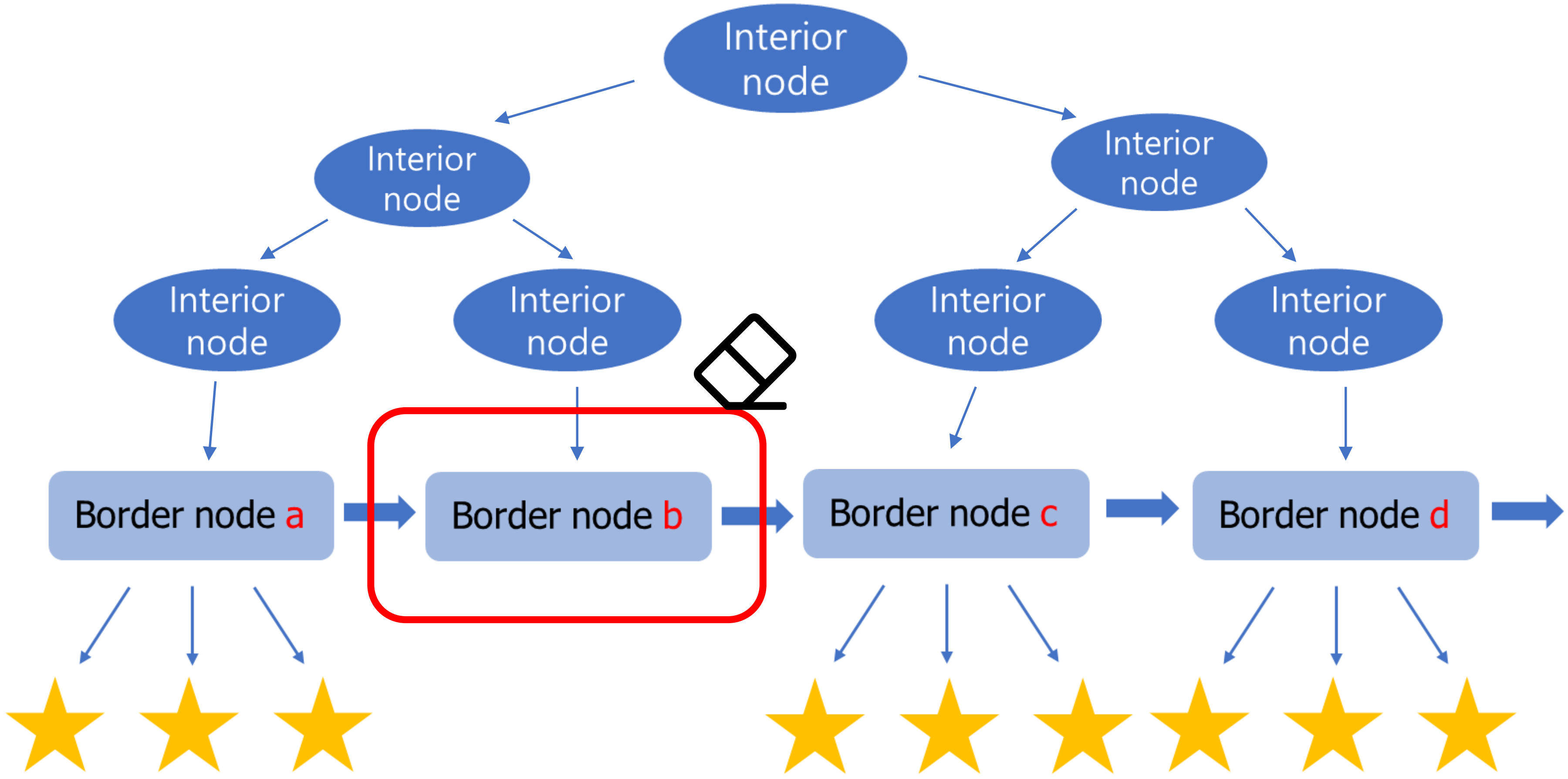
- Writer-Reader coordination
  - Split (border node)



Lowkey(n) remains constant over n's lifetime

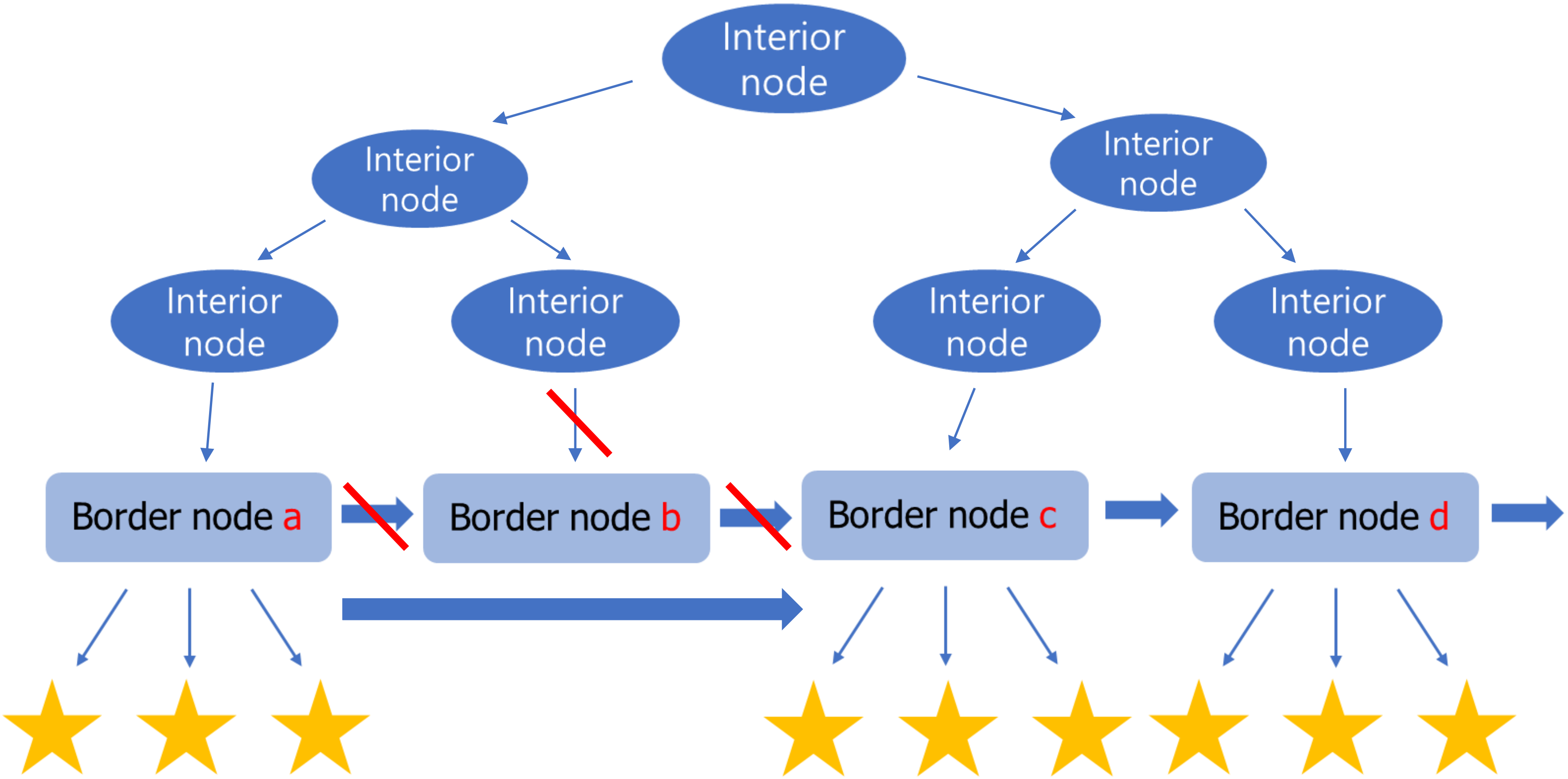
Concurrency

- Remove node



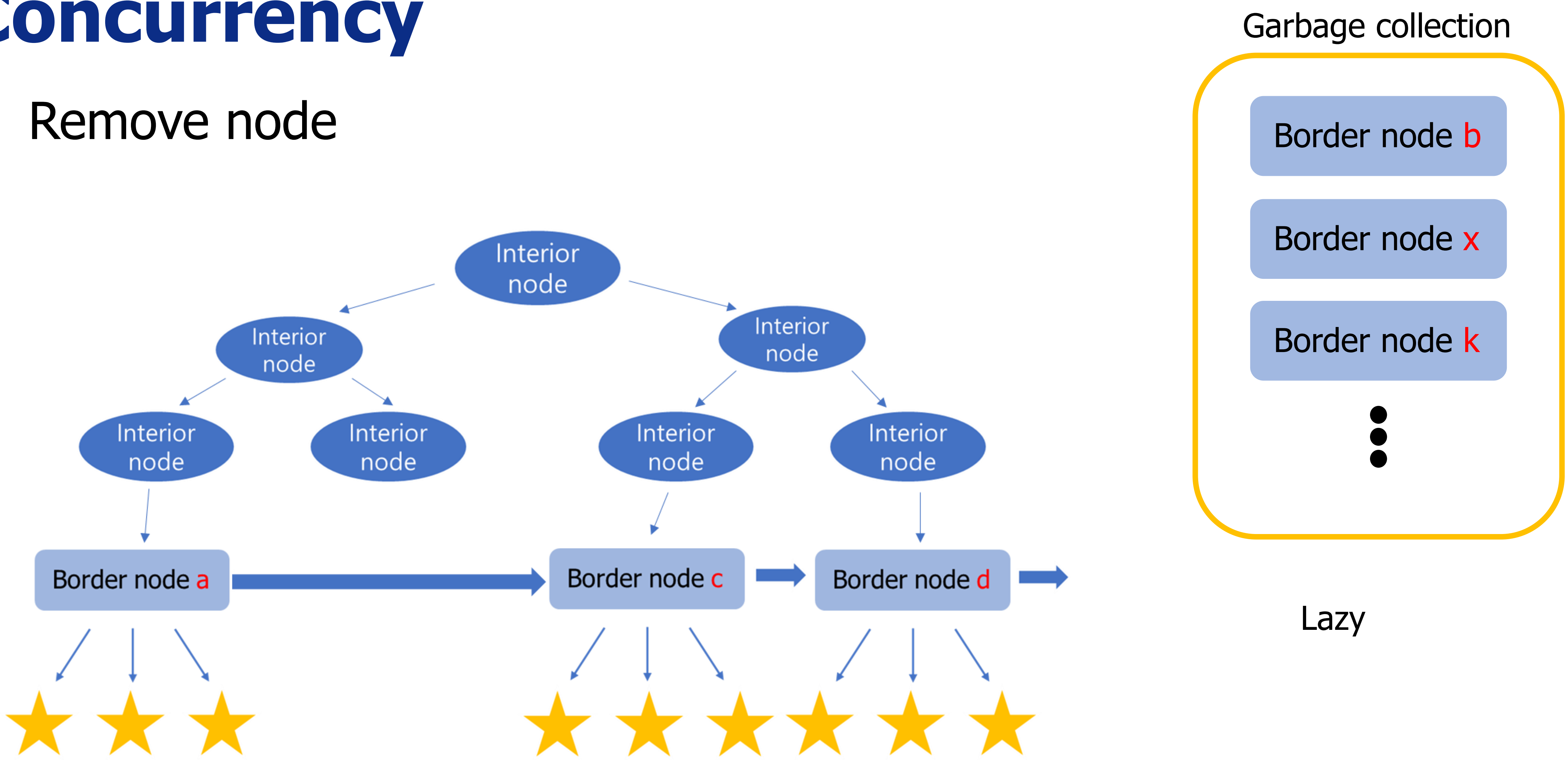
Concurrency

- Remove node



Concurrency

- Remove node



Concurrency

- Discussion
  - More than 30% of the cost of a Masstree lookup is in computation
    - Computation = key search within tree node
    - Linear search has higher complexity than binary search
    - Linear search exhibits better locality

Evaluation

- Two-part Evaluation
  - Masstree as a data structure
    - Compare with other data structures
    - Impact of various design choices and optimizations
  - Masstree as a system
    - Compare with other storage systems

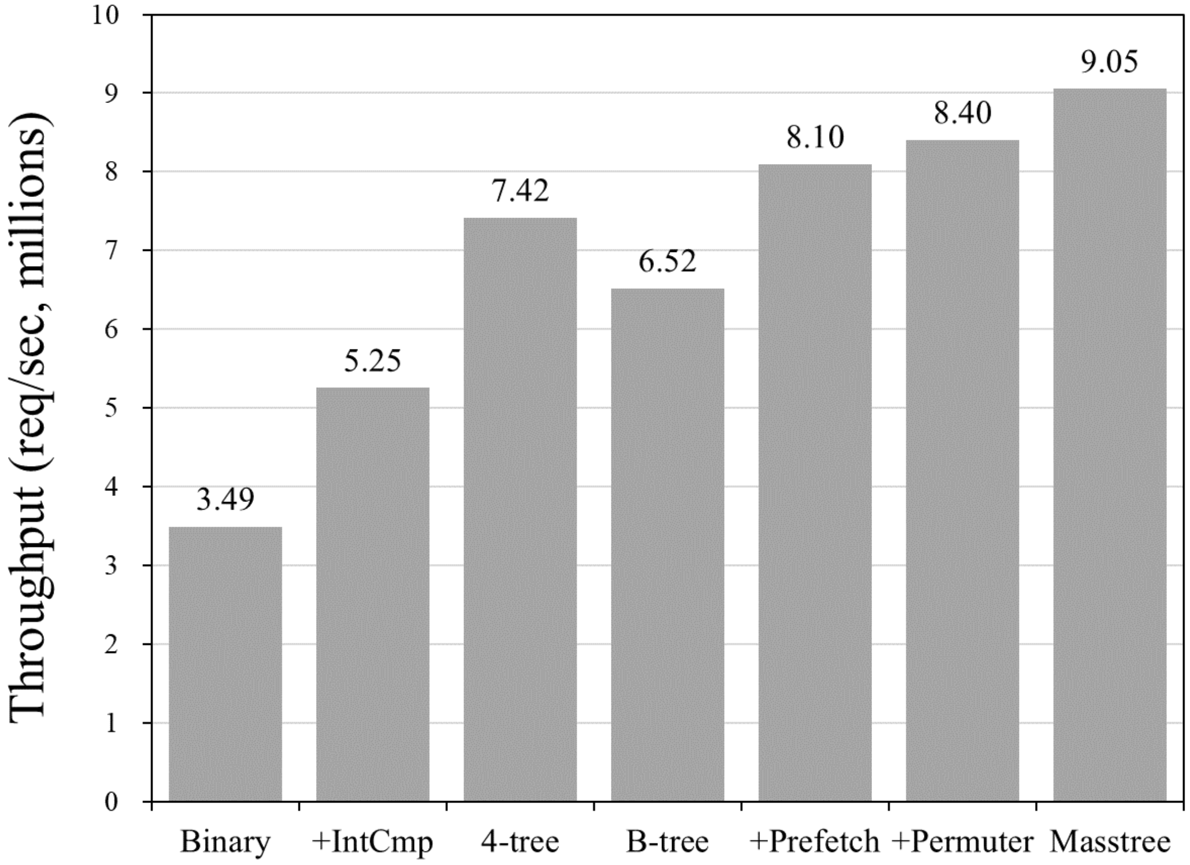
Evaluation

- Setup
  - 6-cores 2.4GHz AMD Opteron 8431 chips X 8 (On test only use 16-cores)
  - 8GB DRAM per chip (On test: 24GB DRAM)
  - SSD X 4 (write speed: 90 ~ 160 MB/s)
  - 10 Gb Ethernet card
  - 25 client machines send queries over TCP

Performance

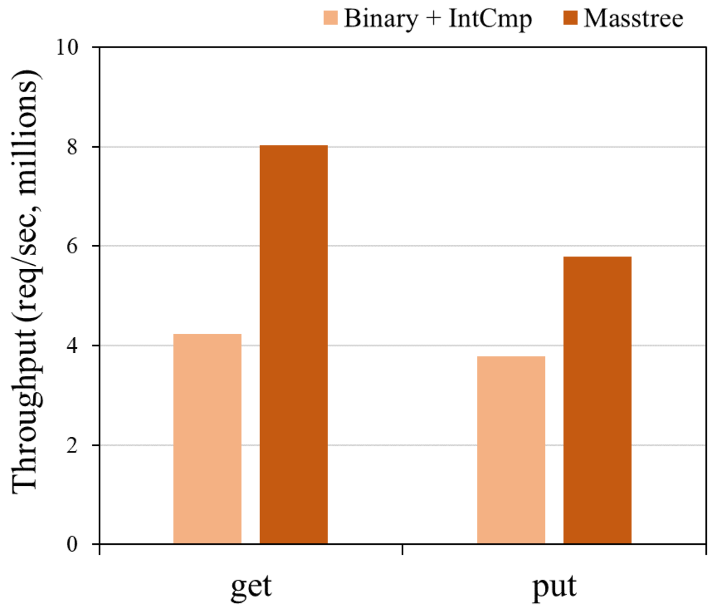
- Put (w/o log, network clients)
  - 16 cores put workload with 140M-key, 1-to-10-byte decimal

- Comparison Group
  - Binary tree
    - +IntCmp(+Flow, +Superpage)
  - 4-tree
  - B-tree
    - +Prefetch
    - +Permuter
  - Masstree



System relevance

- With logging, network clients
  - 16-cores get and put workload with 140M-key, 1-to-10-byte-decimal

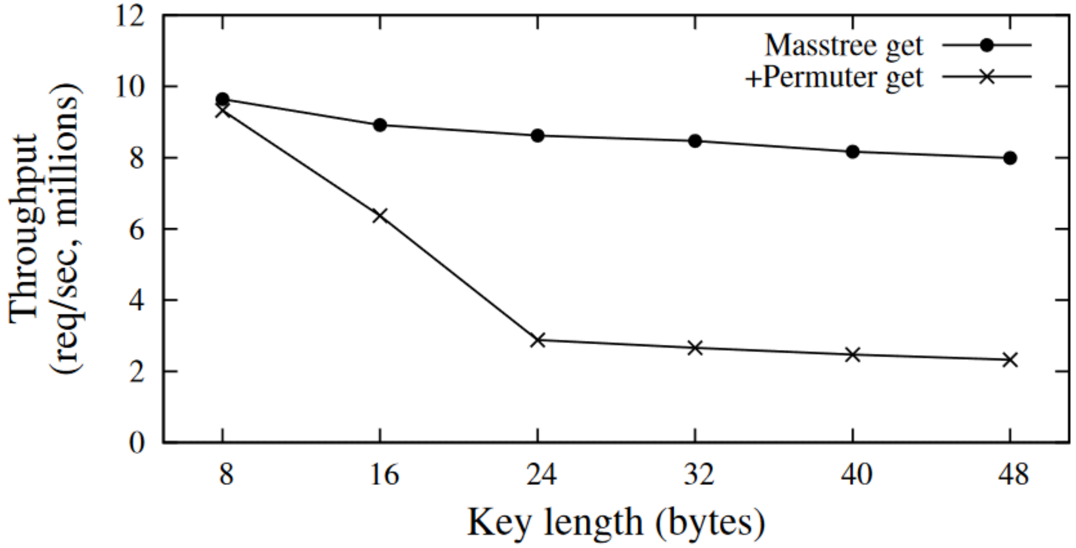


  - Masstree outperformed by **1.9x** and **1.53x** for get and put

➔ Masstree design can **improve system performance**

Flexibility

- Keys with common prefixes
  - Keys differ only in the last 8 bytes



  - "B-tree+Permuter" incurs a cache miss for the suffix of every key it compares
  - Masstree has **3.4**× the throughput for relatively long keys

Flexibility

- Variable-length keys
  - Compare with B-tree supporting 8-byte fixed size keys
    - 16-core get workload with 80M-keys, 8-byte decimal
  - Masstree's throughput was only **0.8%** behind

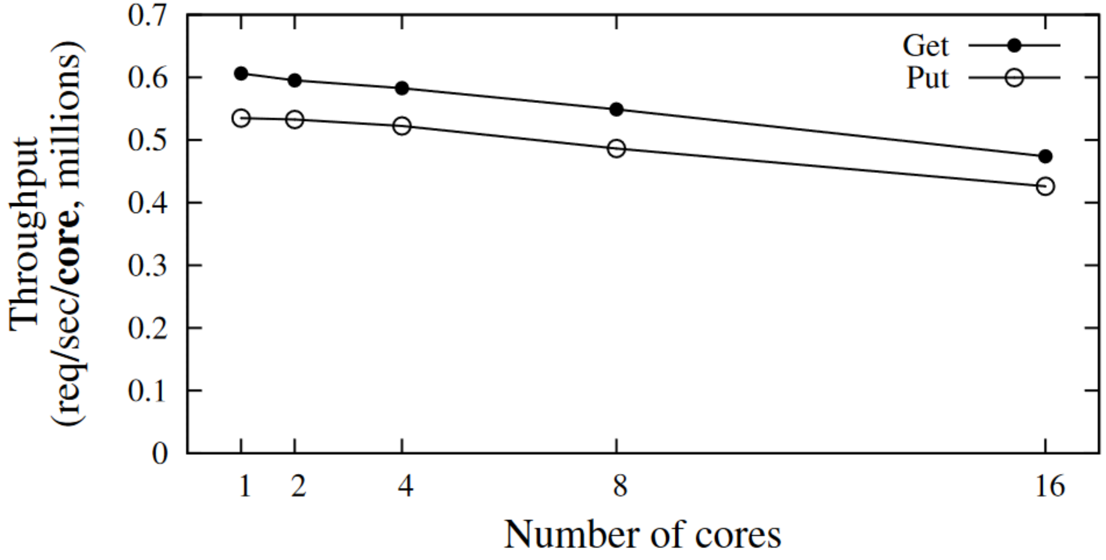
  ➔ Masstree design **effectively has fixed-size keys** in most tree nodes

- Concurrency
  - Compare with single-core version of Masstree
    - 1-core put workload with 140M-keys, 1-to-10-byte-decimal
  - Concurrent Masstree is beaten by single-core Masstree by just **13%**

  ➔ Additional overhead for Concurrent Masstree is **not significant**

Scalability

- Masstree's performance scales with the number of cores
  - 16-cores get and put workload with 140M-key, 1-to-10-byte-decimal
  - Y axis shows per-core throughput



  - 16-cores Masstree scales to 12.7× and 12.5× its 1-core performance for gets and puts

➔ **More cores means more competition** between cores for some limited resources

System Evaluation

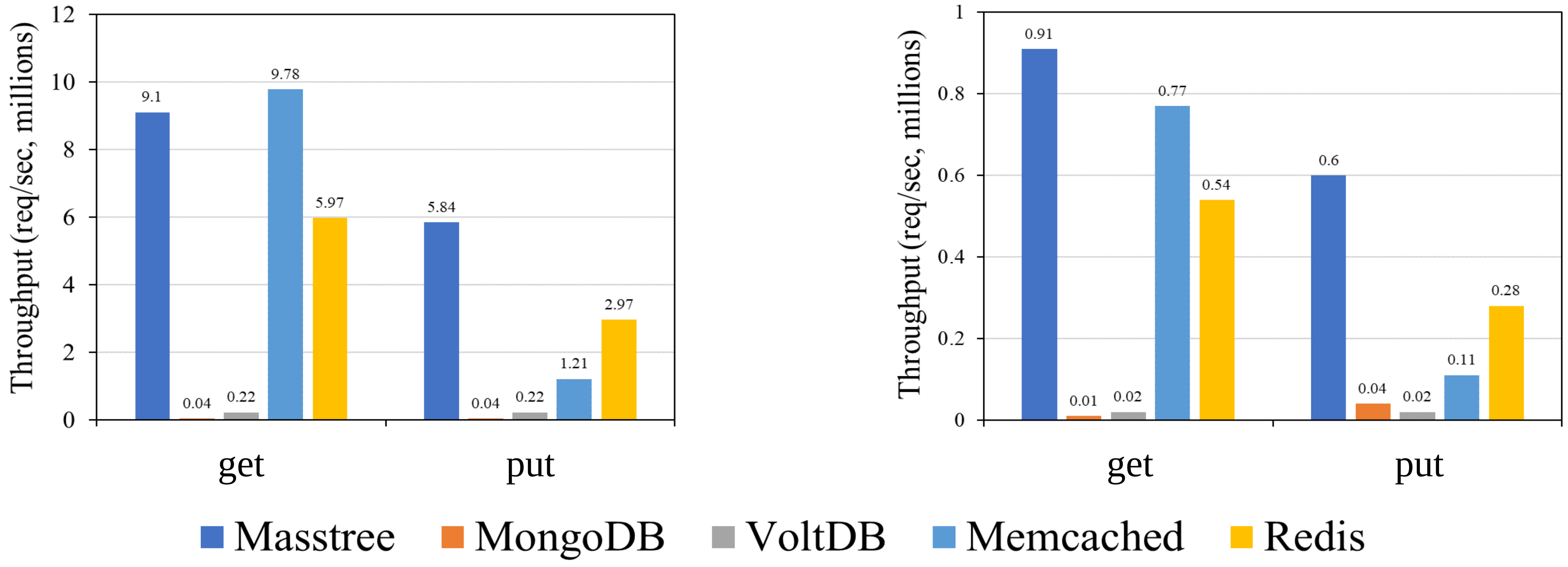
- Compare Masstree with other systems
  - MongoDB: key-value store using B-tree
  - VoltDB: in-memory RDBMS
  - memcached: in-memory key-value store using hash table
  - Redis: in-memory key-value store using hash table

Server	C/C++ client library	Batched query	Range query
MongoDB-2.0	2.0	No	Yes
VoltDB-2.0	1.3.6.1	Yes	Yes
memcached-1.4.8	1.0.3	Yes for get	No
Redis-2.4.5	latest hiredis	Yes	No

Figure 12. Versions of tested servers and client libraries.

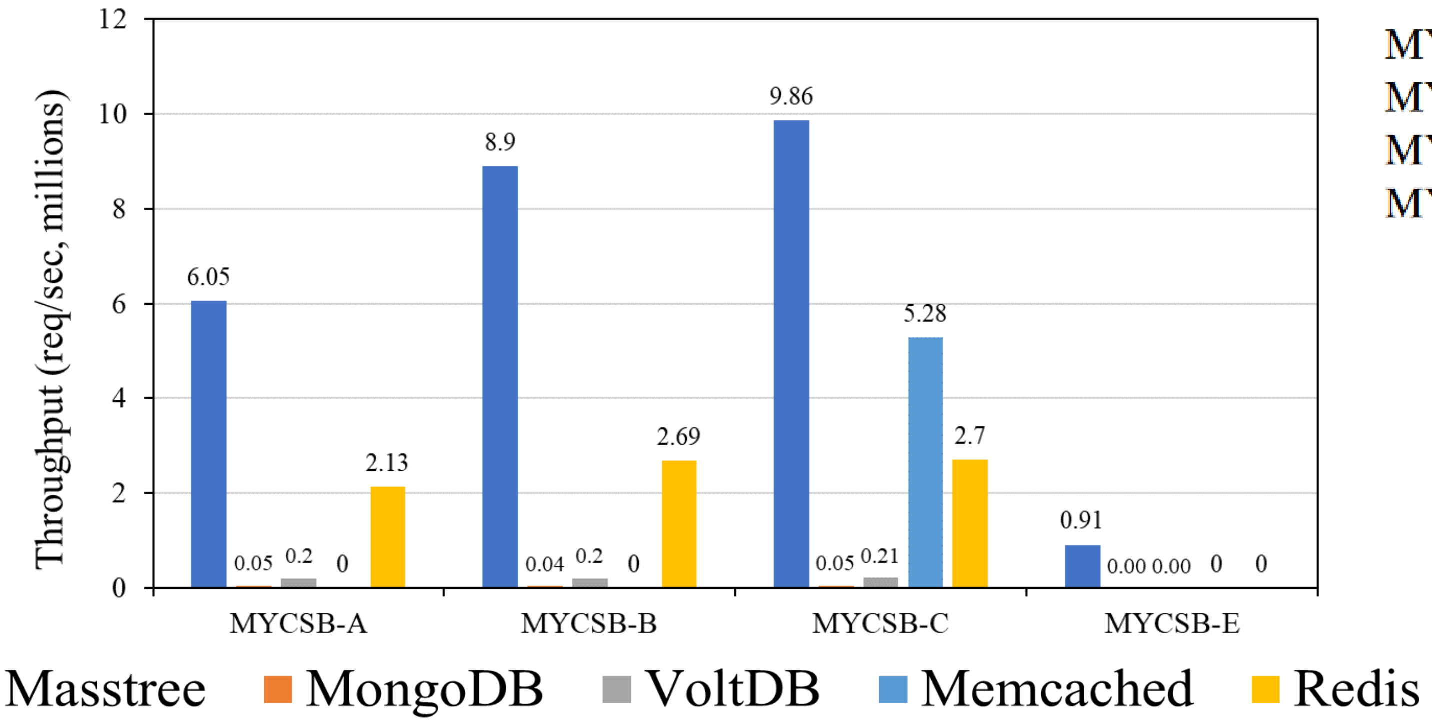
System Evaluation

- Compare Masstree with other systems
  - Uniform key popularity, 20M keys, 1-to-10-byte decimal, one 8-byte column
  - 16 cores
  - 1 core



System Evaluation

- Compare Masstree with other systems
  - 16 cores, Zipfian key popularity, 5-to-24-byte keys, ten 4-byte columns for get, one 4-byte column for update & get range



MYCSB-A (50% get, 50% put)
MYCSB-B (95% get, 5% put)
MYCSB-C (all get)
MYCSB-E (95% getrange, 5% put)

Conclusion

- Masstree is persistent in-memory key-value database
  - Consist of a trie-like concatenation of B+-trees
  - Keep all data in a tree, and the tree is shared among all cores
  - Provide consistency and durability by logging and checkpointing

- Masstree executes more than 6M simple queries per second
  - Masstree's performance is comparable to memcached and higher than VoltDB, MongoDB, and Redis

Thank you

Appendix

Masstree

- Layout
  - About time complexity

Essentially, a masstree is a multi-layered B+tree structure.

If the key has l byte, it will be stored in the l /8 layer.
In other words, the cost of searching for this key is l /8 times of B+Tree search.

$\boldsymbol{O(\log n)}$
The time complexity of a B+Tree.

$\boldsymbol{O\left(\frac{l}{8} \log n\right)}$
B+Tree with multiple levels.

$\boldsymbol{O(l \log n)}$
Omitting constants

Considering that the B+tree constructed by masstree always maintains a **flatter structure**, it can be foreseen that if the length of the key is **longer**, masstree will have an **advantage** over B+trees