Lock-free Concurrent Level Hashing for Persistent Memory

Zhangyu Chen, Yu Hua, Bo Ding, Pengfei Zuo

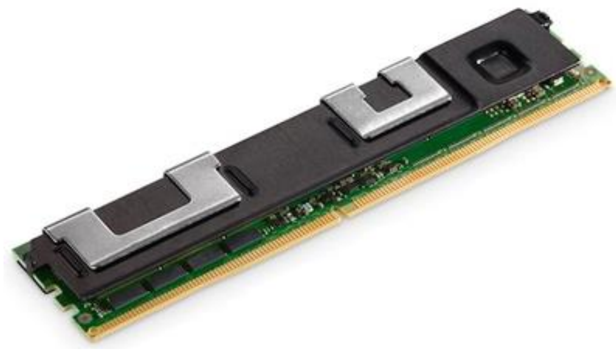
Huazhong University of Science and Technology

USENIX ATC 2020

Persistent Memory (PM)

- PM features
  - Non-volatility
  - Large capacity
  - Byte-addressability
  - DRAM-scale latency

- PM speedups storage systems
  - TB-scale memory for applications
  - Instant recovery from system failures



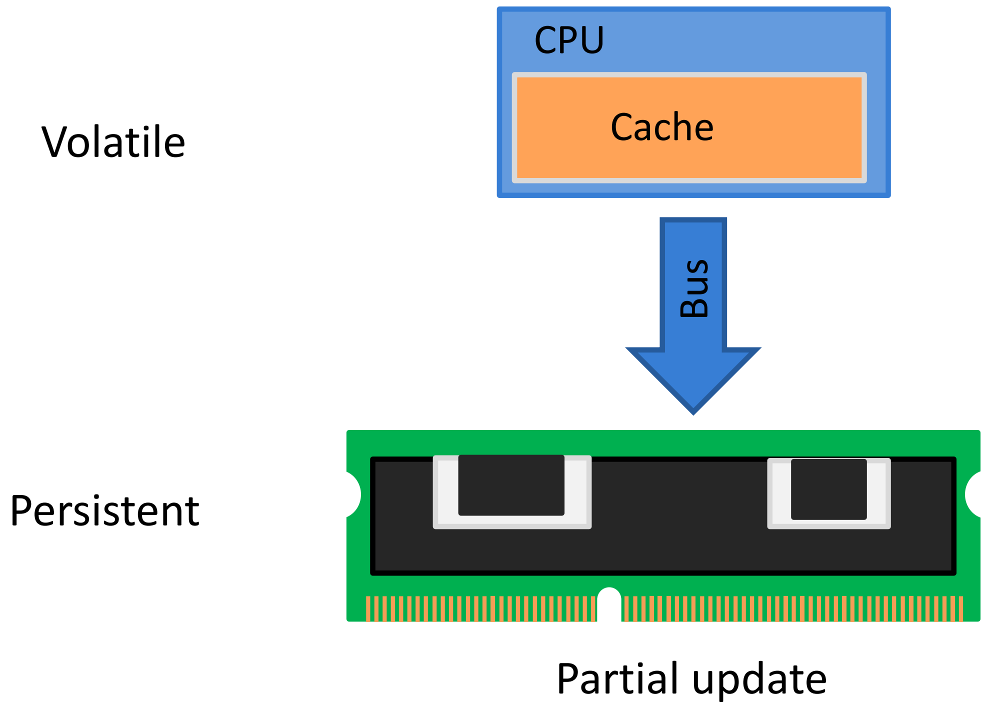
Intel Optane DC Persistent Memory
512 GB per module at most
DIMM compatible

PM Optimization

1. High overhead for writes
   - Limited endurance
   - Low write bandwidth of PM (Optane PM study in FAST ’20)
     - 1/6 DRAM
     - 1/3 read bandwidth of PM

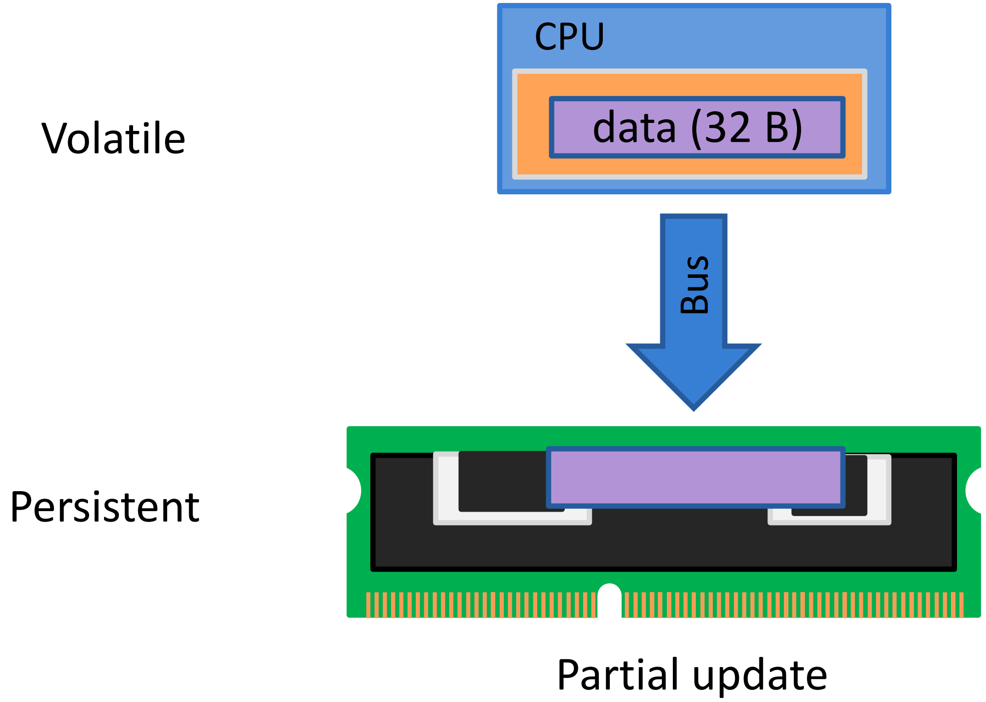
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging



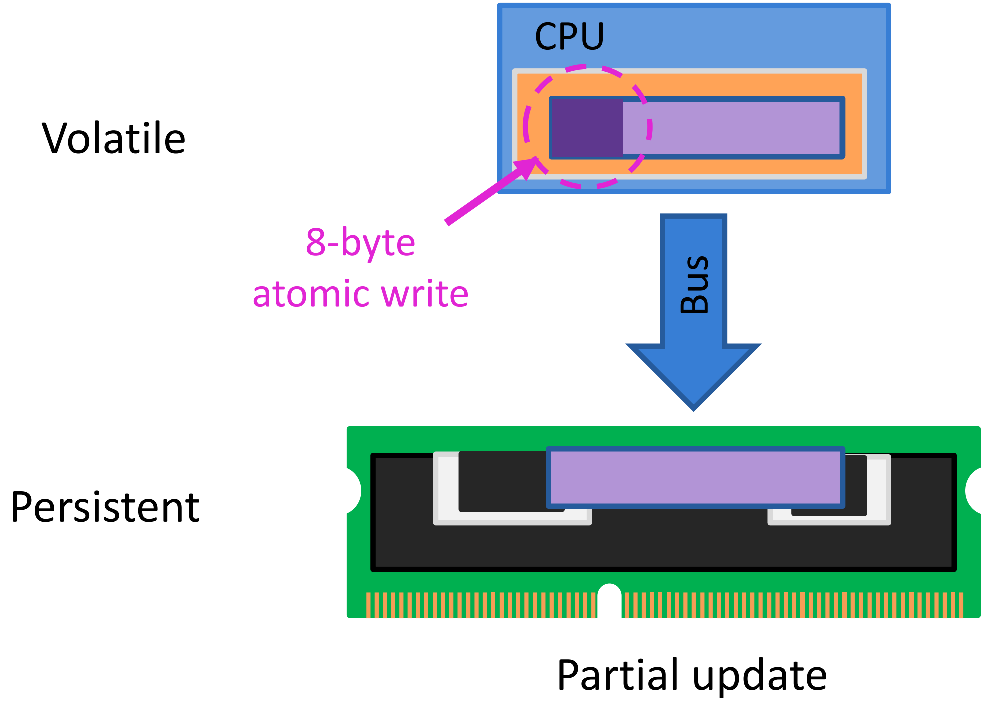
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging



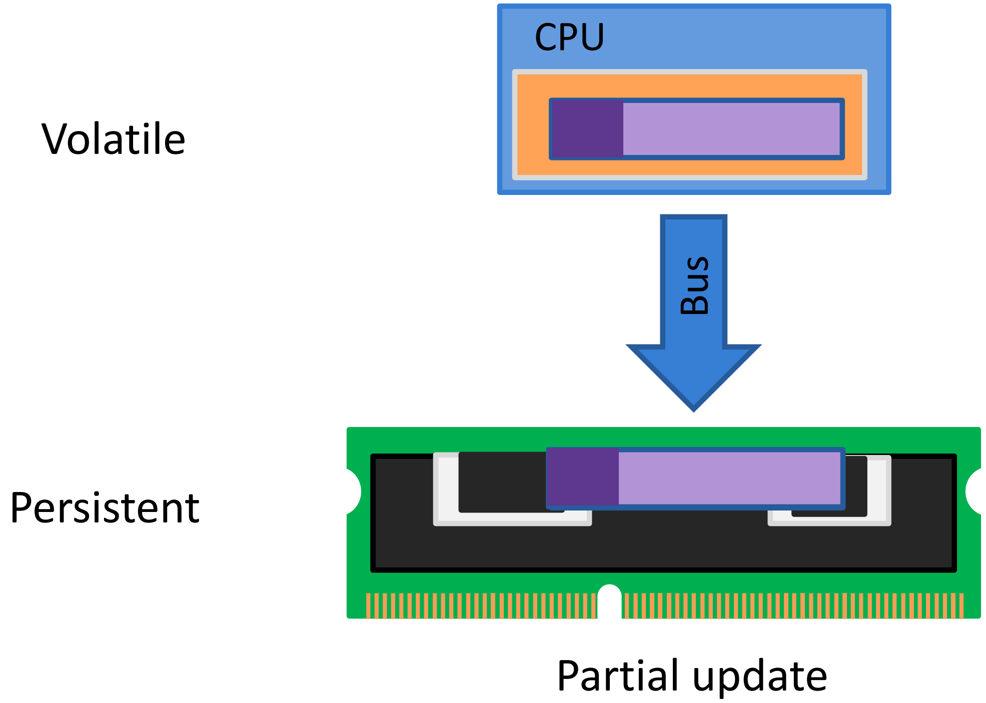
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   – Partial update: Copy-on-Write (CoW) or logging



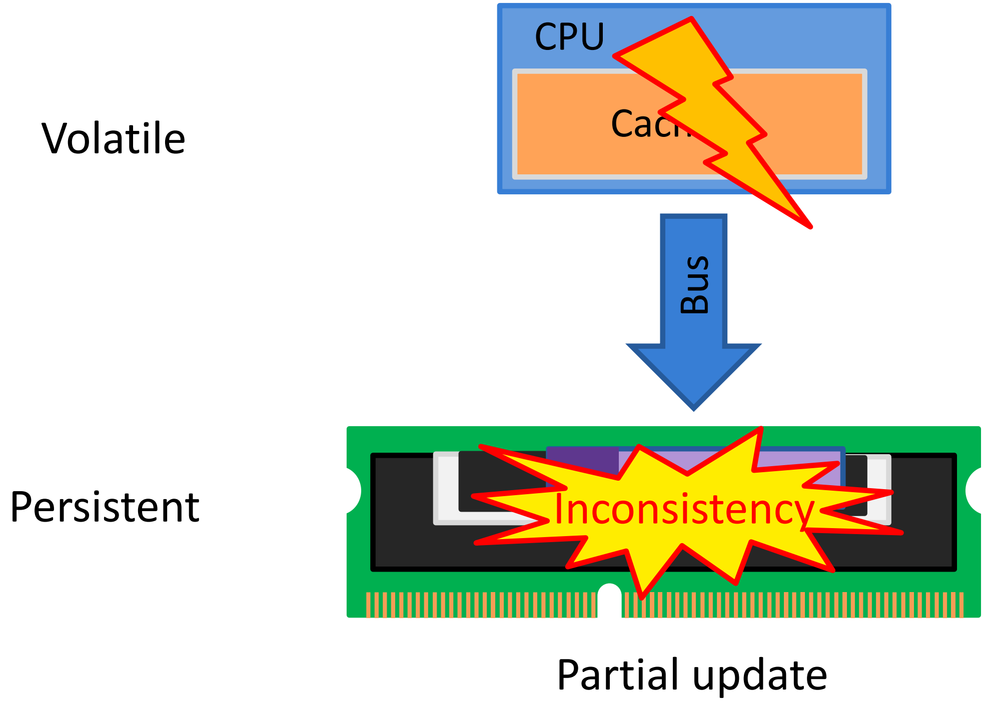
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging



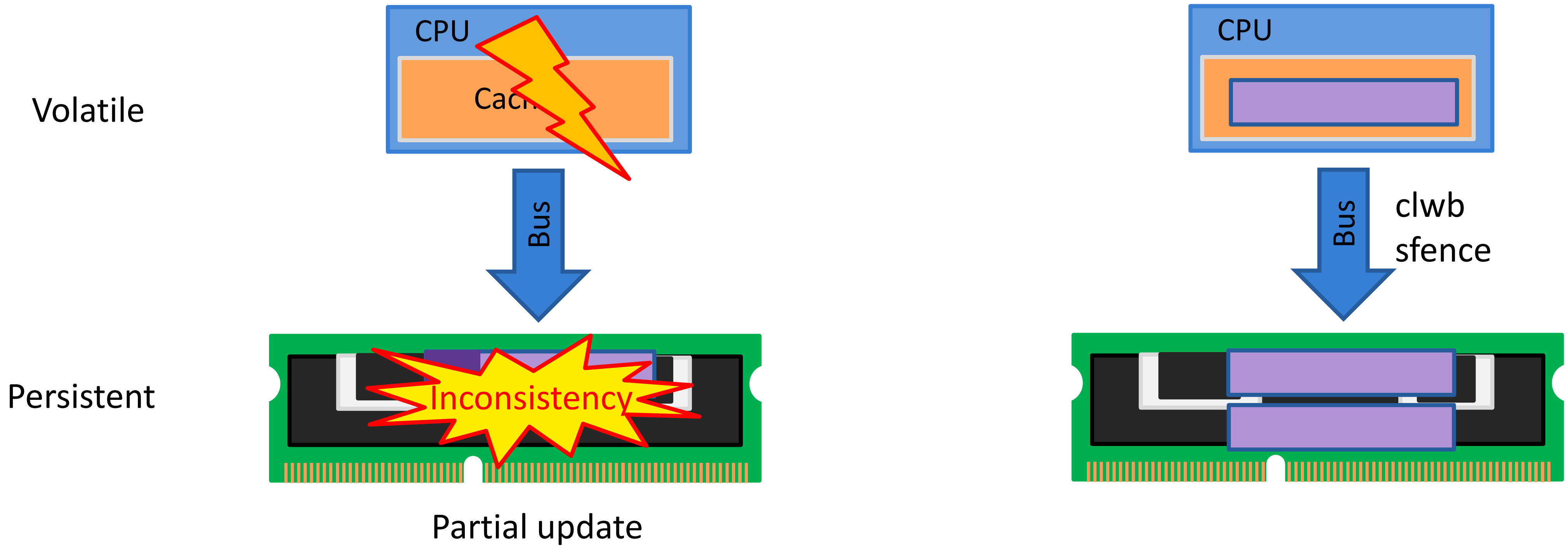
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging



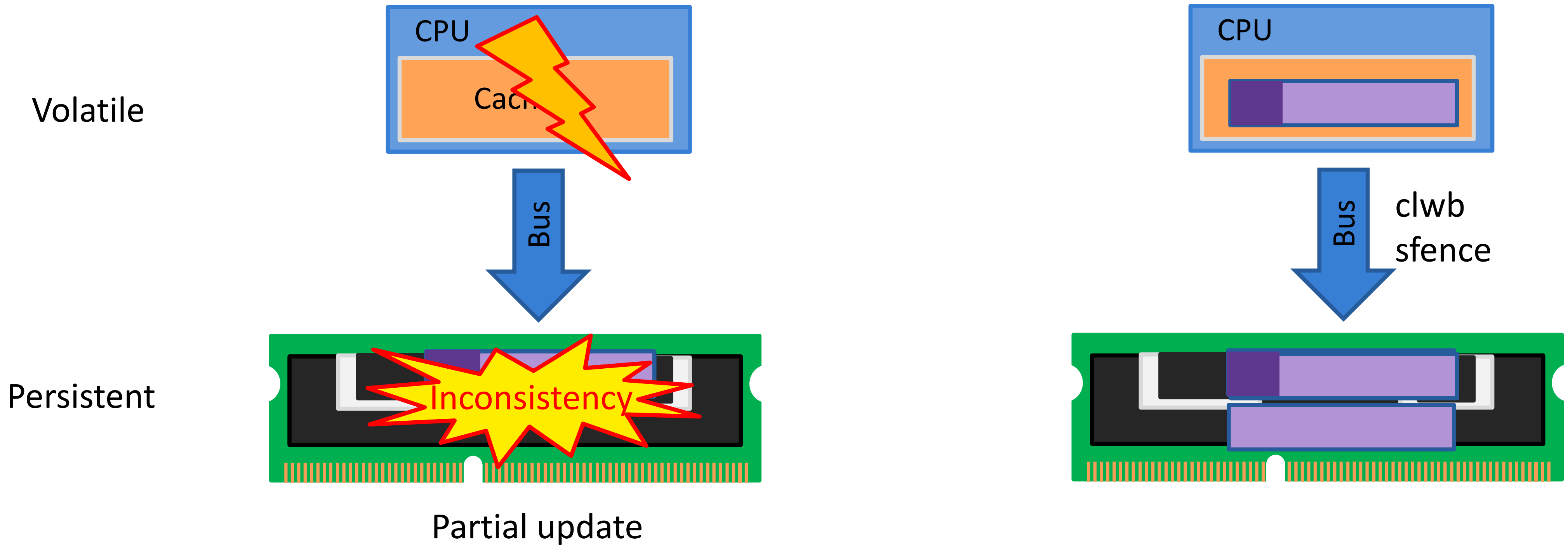
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   – Partial update: Copy-on-Write (CoW) or logging



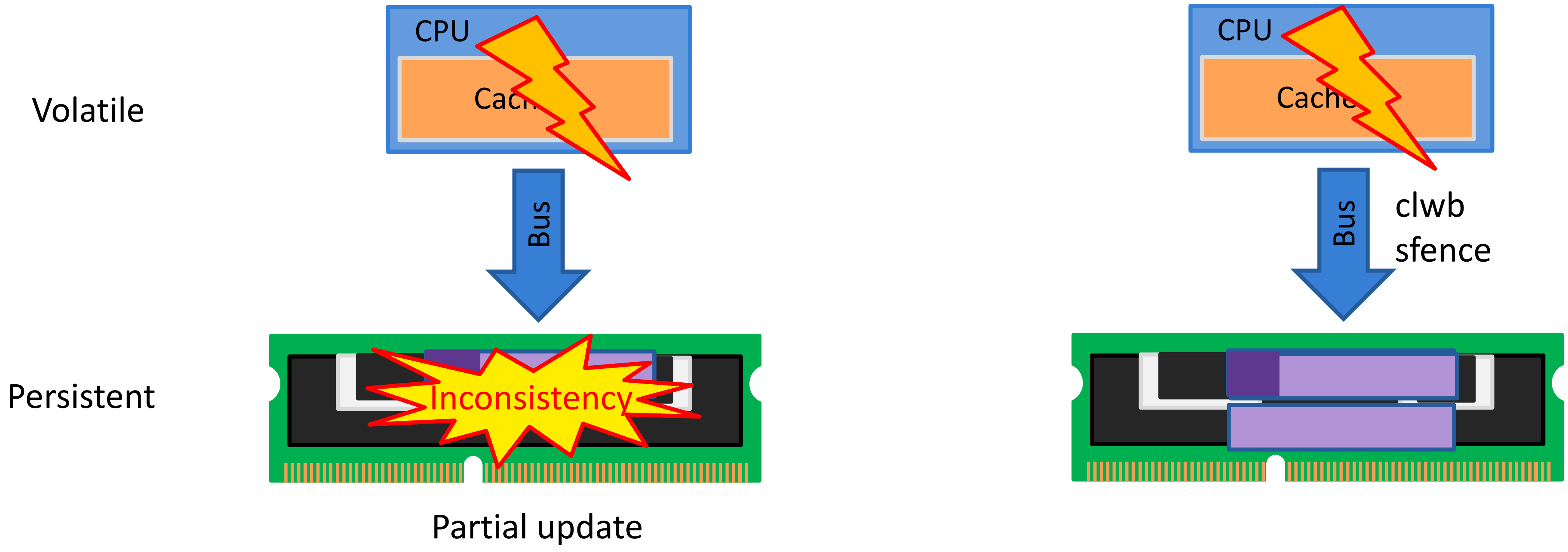
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   – Partial update: Copy-on-Write (CoW) or logging



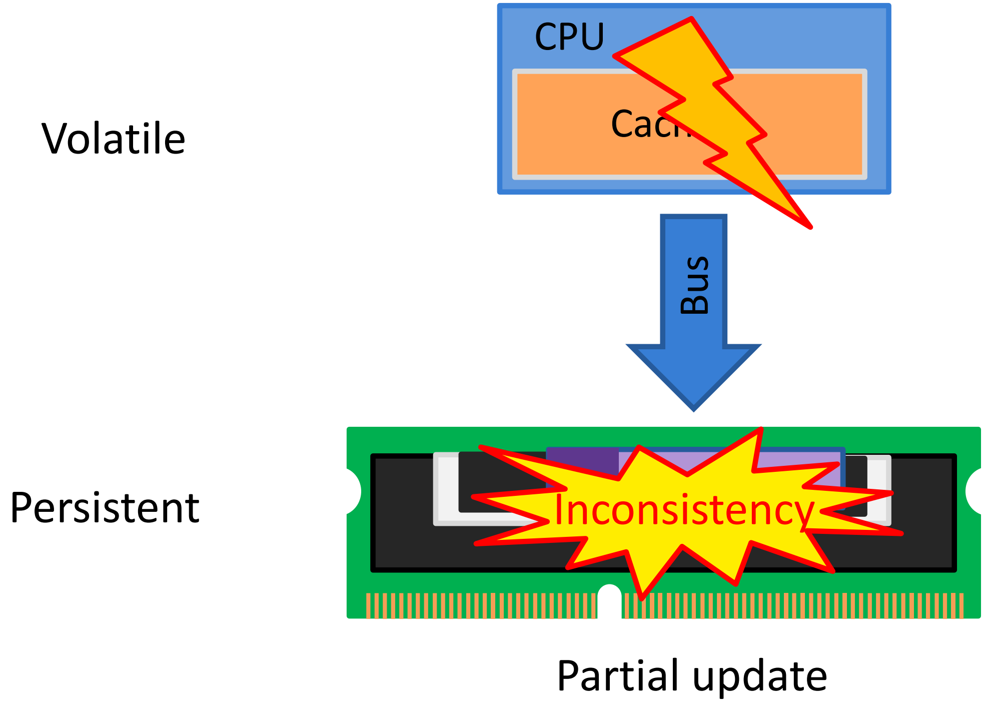
PM Optimization

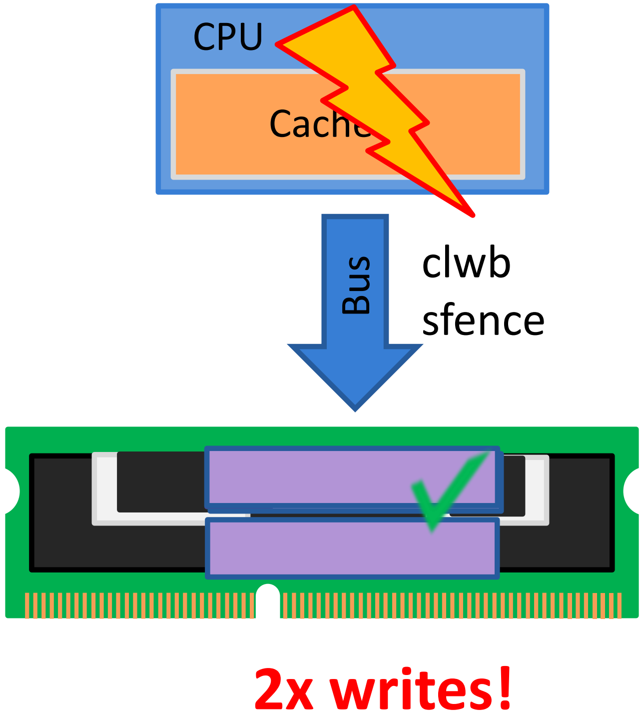
1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging



PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging

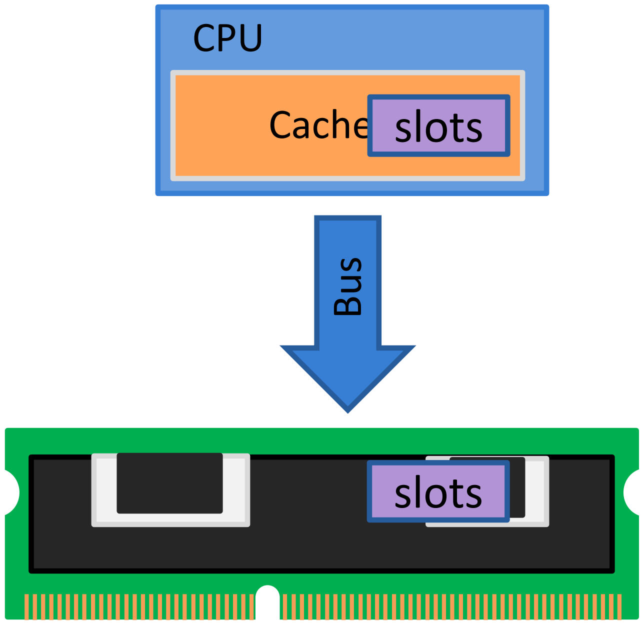




PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   – Partial update: Copy-on-Write (CoW) or logging
   – Reordering: memory fences

Program order

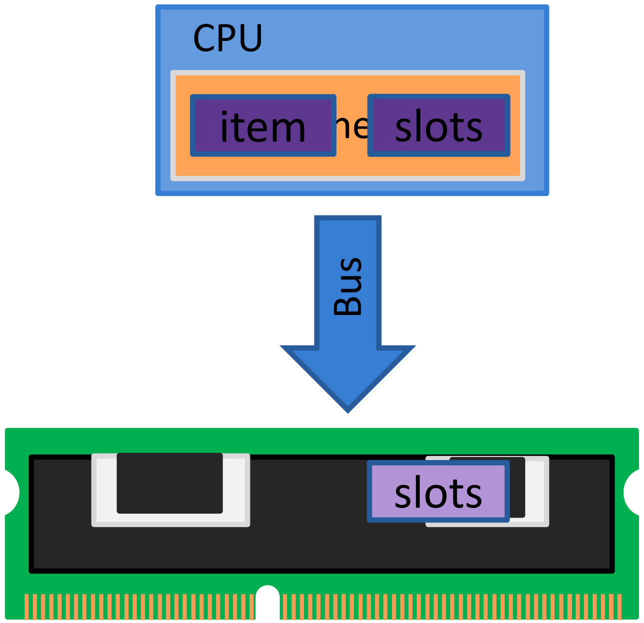


PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging
   - Reordering: memory fences

Program order

```
kv_t item = new kv_t(k, v);
slots[0] = &item;
```



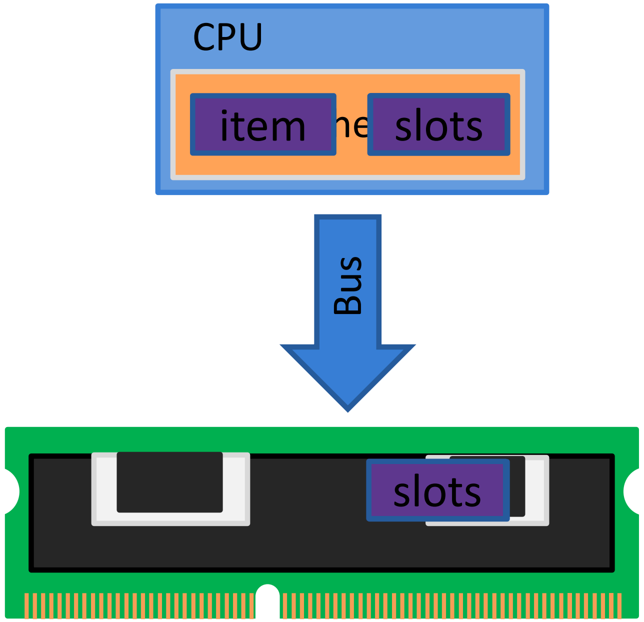
PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging
   - Reordering: memory fences

Program order

```
kv_t item = new kv_t(k, v);
slots[0] = &item;
```

Cache Reordering



PM Optimization

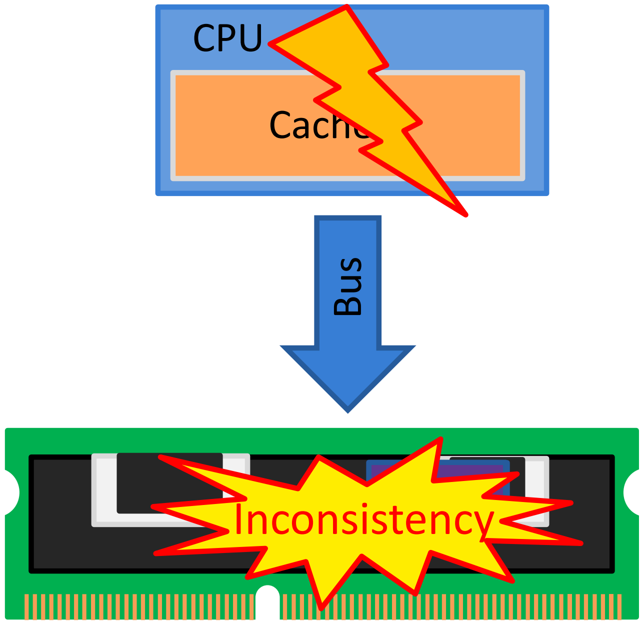
1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging
   - Reordering: memory fences

Program order

kv_t item = new kv_t(k, v);

slots[0] = &item;

Cache Reordering

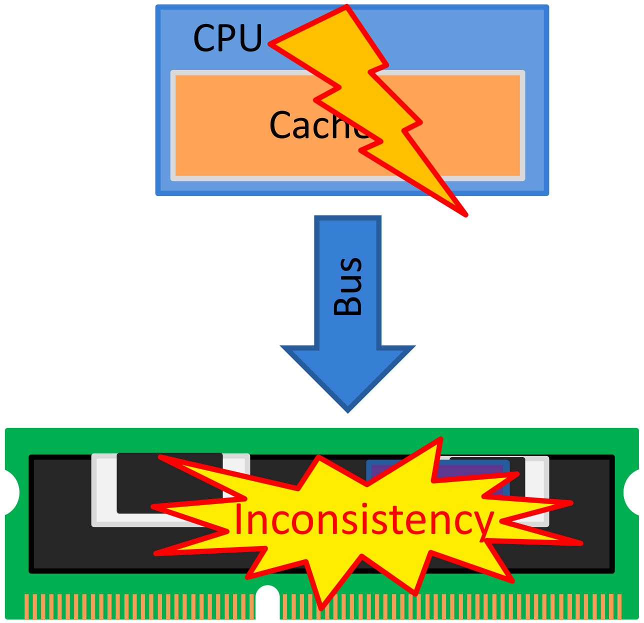


PM Optimization

1. High overhead for writes
2. Inconsistency due to non-volatility
   - Partial update: Copy-on-Write (CoW) or logging
   - Reordering: memory fences

Program order

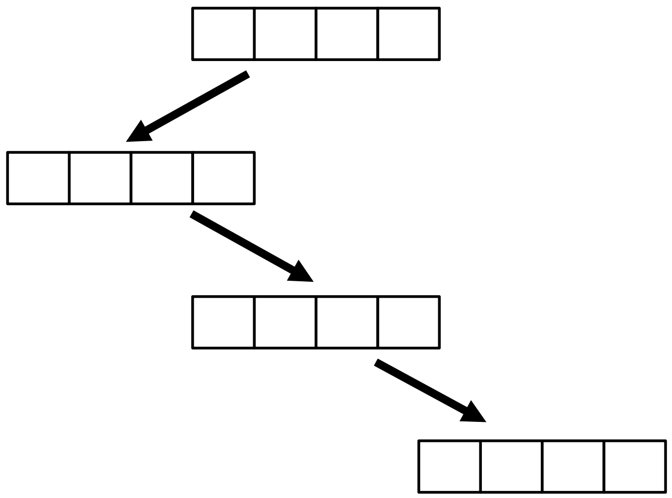
```
kv_t item = new kv_t(k, v);
clwb(item);
sfence;
slots[0] = &item;
```

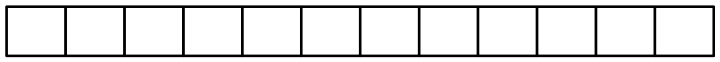


PM Index Structures

➢ PM index structures are important for large-scale storage systems to provide fast queries

- Tree-based structures
- Hashing-based structures

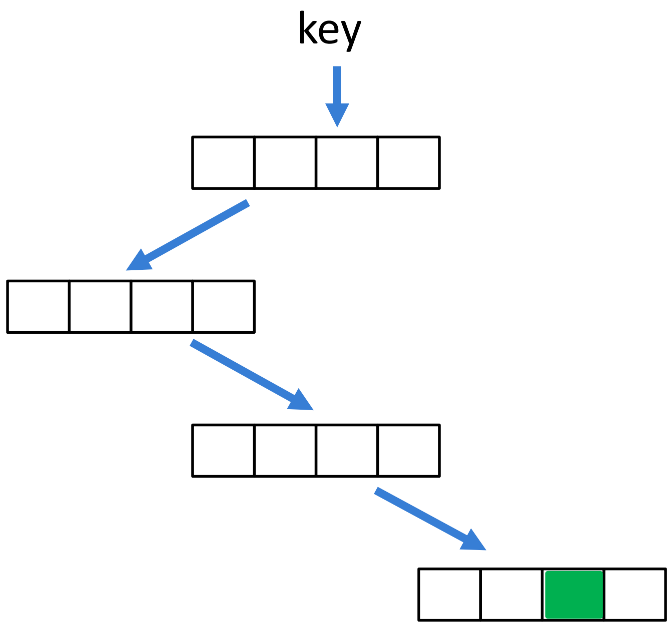




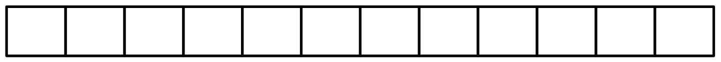
PM Index Structures

➢ PM index structures are important for large-scale storage systems to provide fast queries

- Tree-based structures



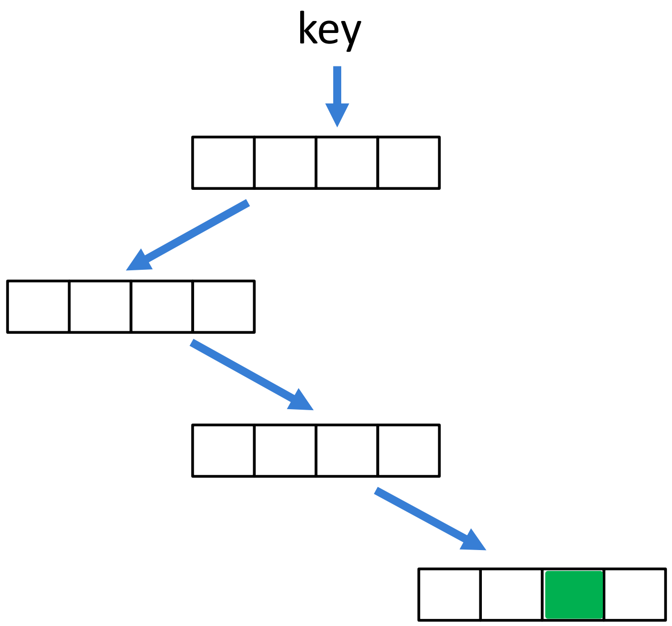
- Hashing-based structures



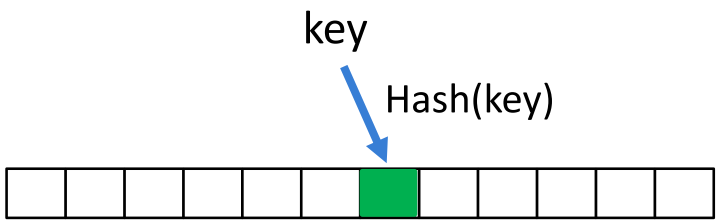
PM Index Structures

➢ PM index structures are important for large-scale storage systems to provide fast queries

- Tree-based structures



- Hashing-based structures

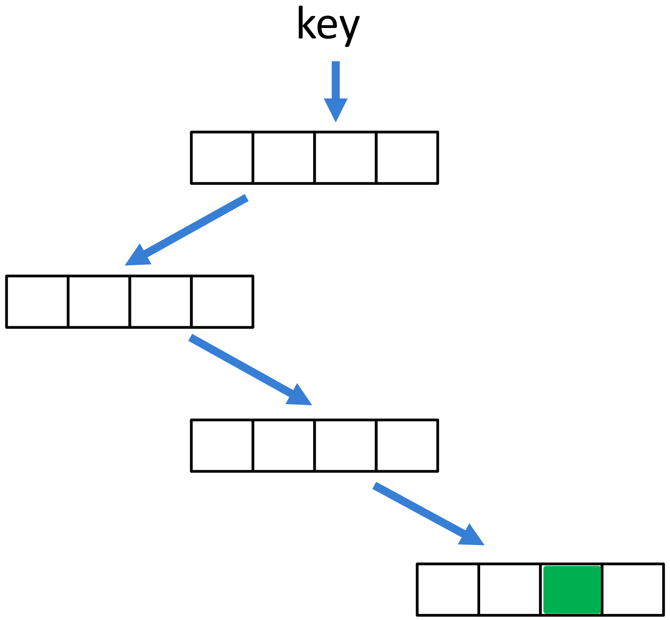


✓ *O(1) time complexity for point query*

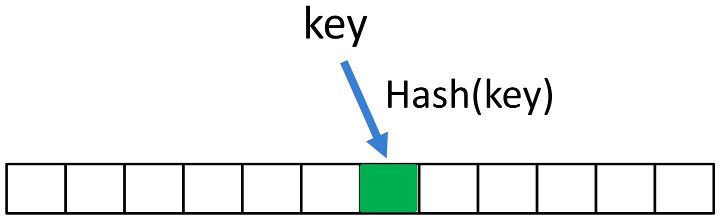
PM Index Structures

➢ PM index structures are important for large-scale storage systems to provide fast queries

- Tree-based structures



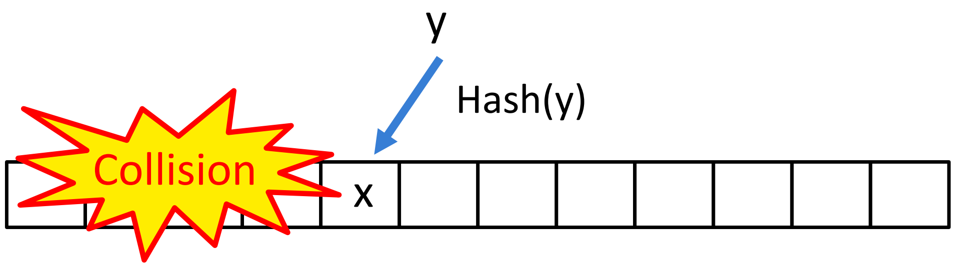
- Hashing-based structures



✓ ***O(1) time complexity for point query***

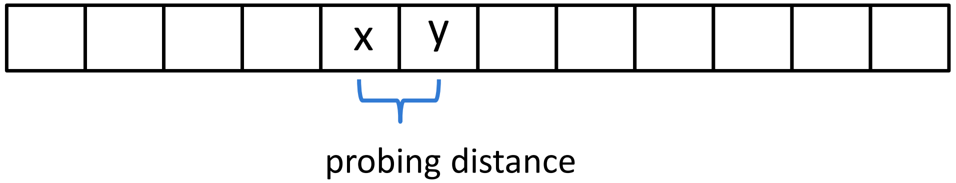
Hashing Collisions and Resizing

- Hash collisions



Hashing Collisions and Resizing

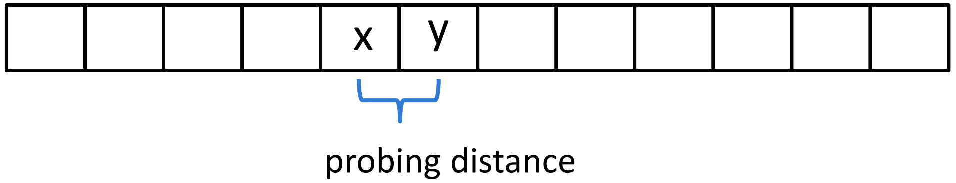
➢Hash collisions



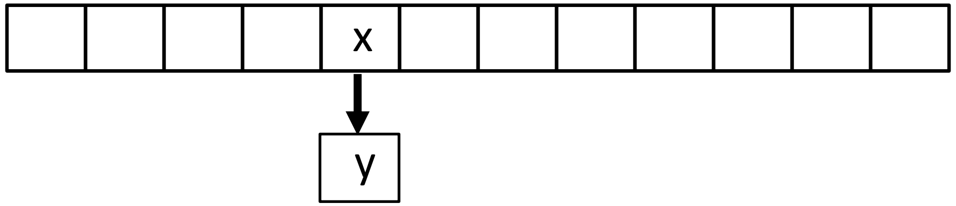
Linear probing

Hashing Collisions and Resizing

- Hash collisions



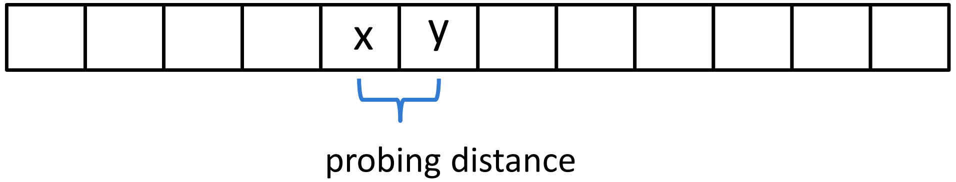
Linear probing



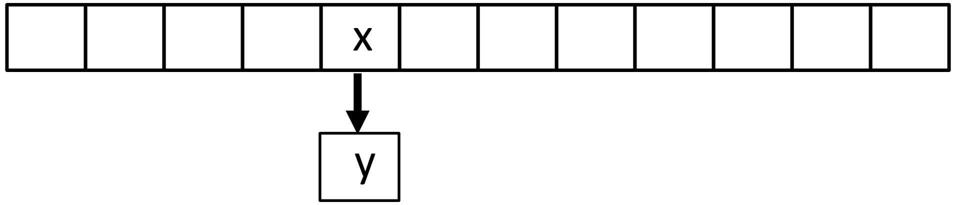
Linked list

Hashing Collisions and Resizing

- Hash collisions

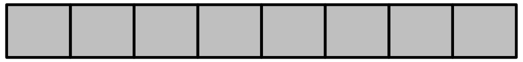


Linear probing



Linked list

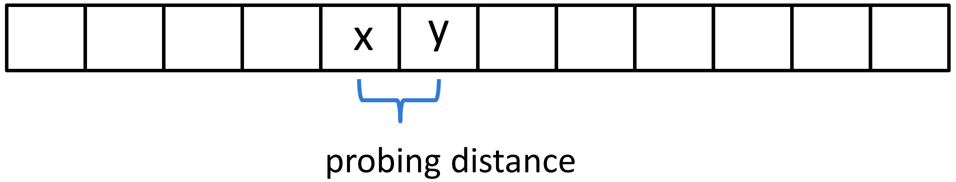
- Resizing



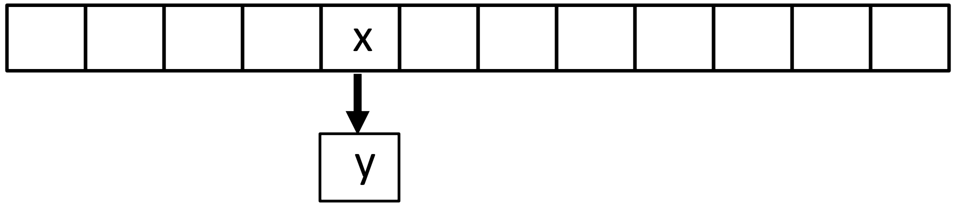
Old hash table

Hashing Collisions and Resizing

- Hash collisions

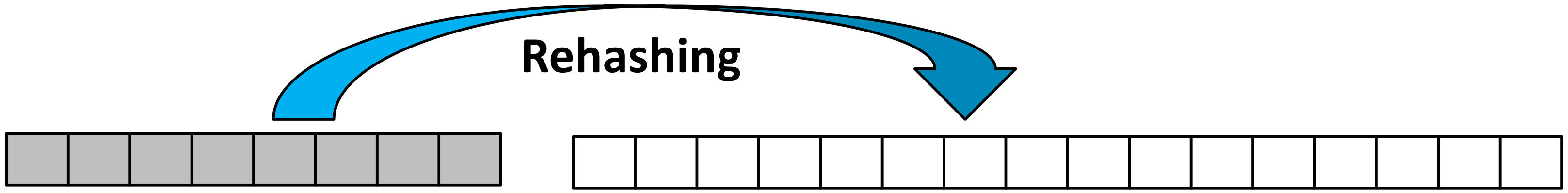


Linear probing



Linked list

- Resizing



Old hash table

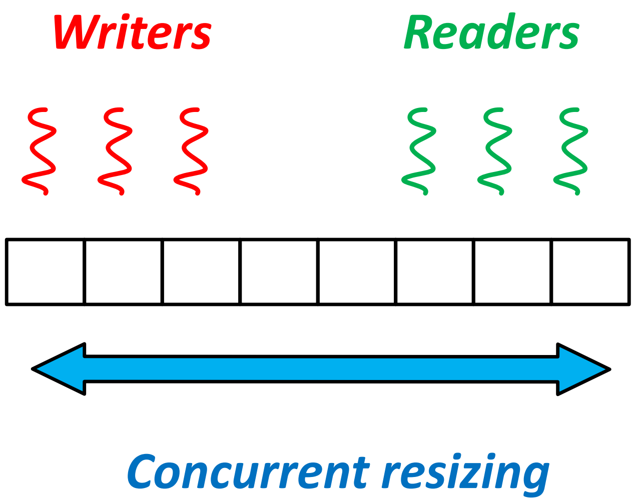
New hash table

High latency!

Concurrent PM Hashing

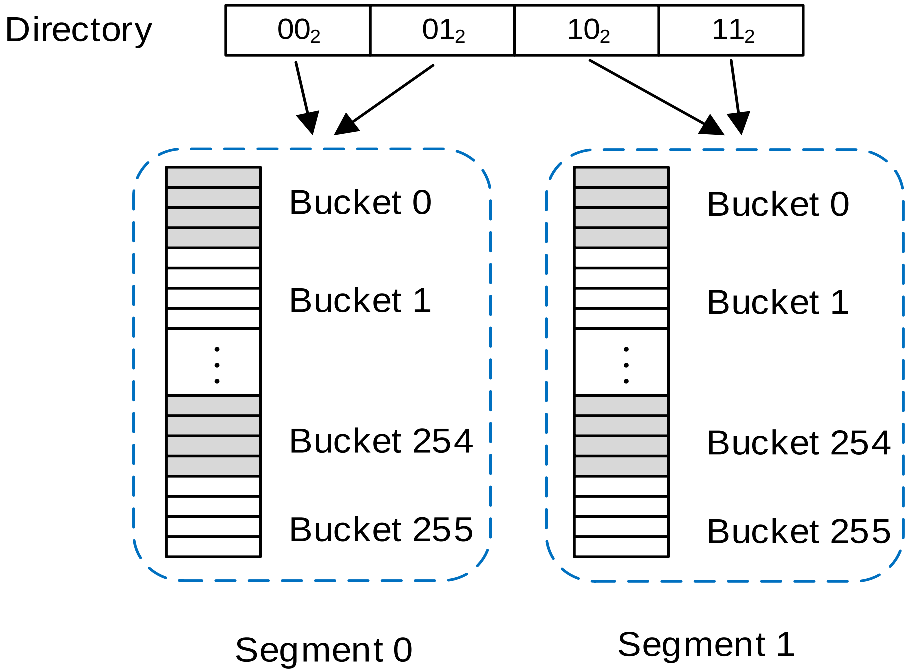
- The importance of concurrency
  - Fast indexing for TB-scale PM data
  - Multi-core environment for servers equipped with Optane PM

- Concurrency for PM hashing
  - Concurrent queries with correctness
    - Multi-reader concurrency
    - Multi-writer concurrency
  - Concurrent resizing



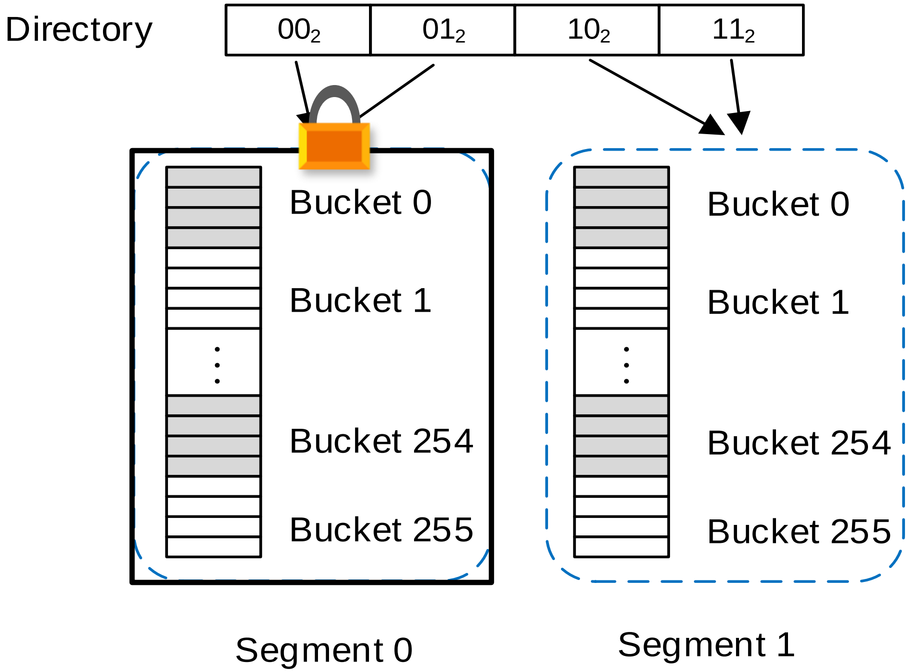
PM Variants of Concurrent Hashing

- CCEH [FAST '19]



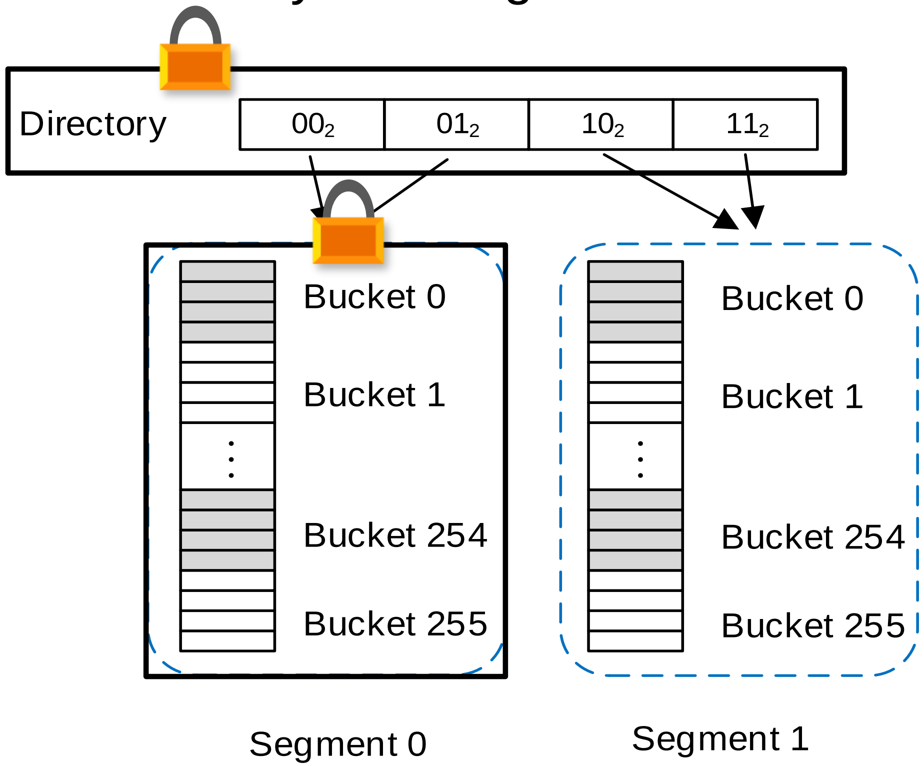
PM Variants of Concurrent Hashing

- CCEH [FAST '19]
  - Segment reader/writer locks for queries



PM Variants of Concurrent Hashing

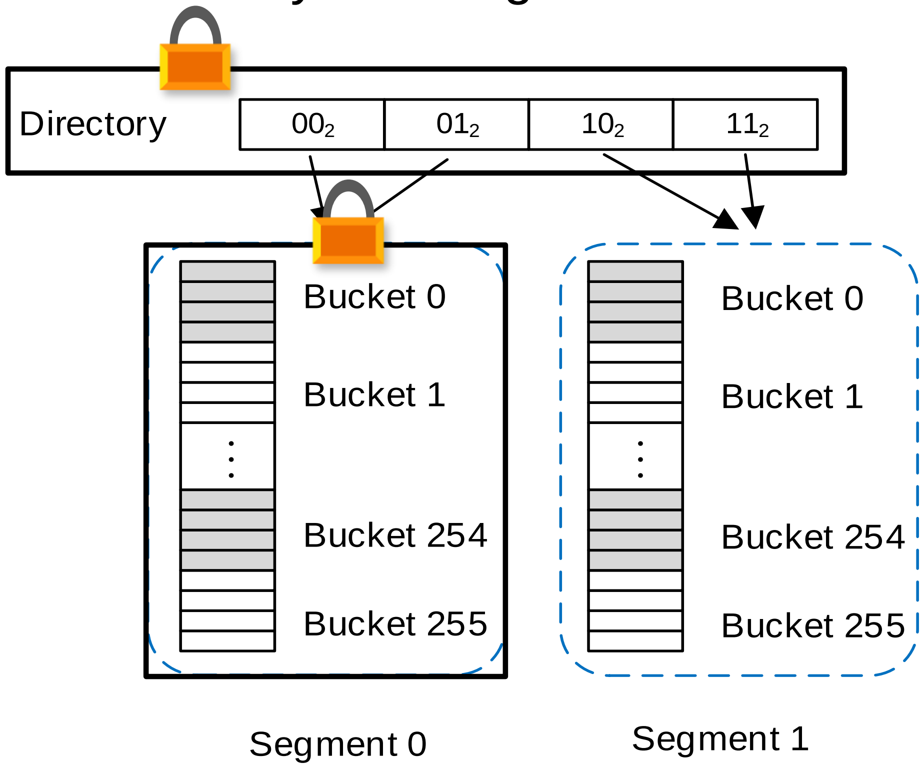
- CCEH [FAST '19]
  - Segment reader/writer locks for queries
  - Dynamic resizing with segment splitting and directory doubling



Coarse-grained locks!

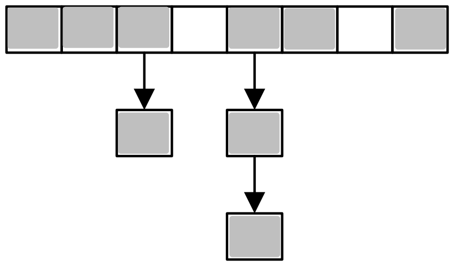
PM Variants of Concurrent Hashing

- CCEH [FAST '19]
  - Segment reader/writer locks for queries
  - Dynamic resizing with segment splitting and directory doubling



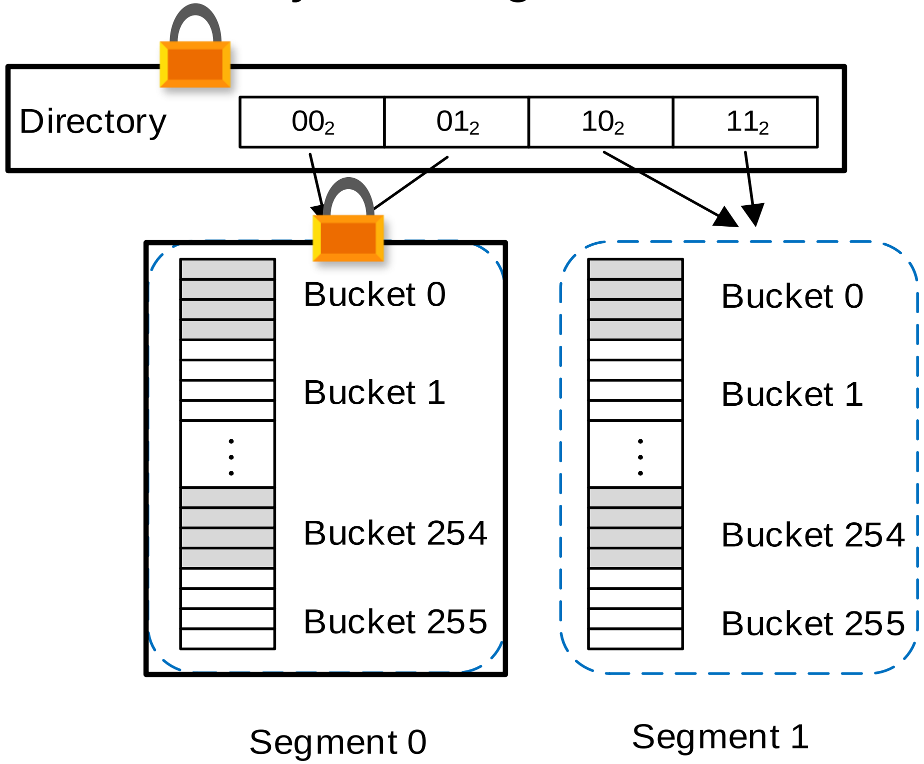
Coarse-grained locks!

- P-CLHT [SOSP '19]



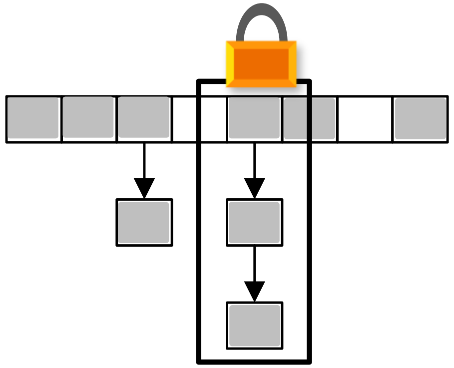
PM Variants of Concurrent Hashing

- CCEH [FAST ’19]
  - Segment reader/writer locks for queries
  - Dynamic resizing with segment splitting and directory doubling



Coarse-grained locks!

- P-CLHT [SOSP ’19]
  - Lock-free search and bucket lock for writes

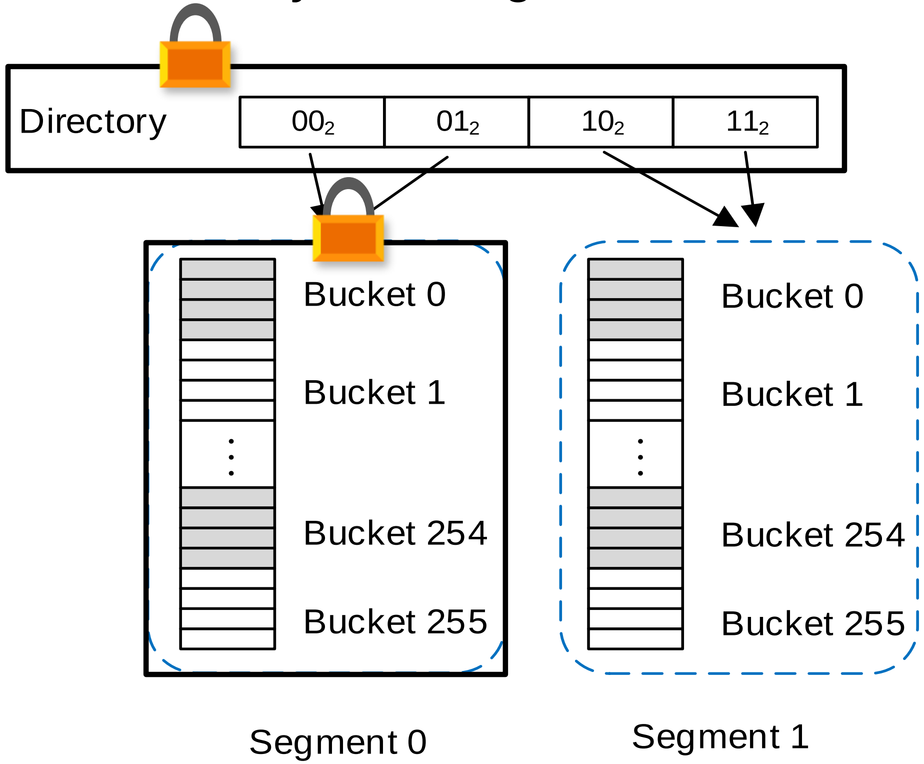


PM Variants of Concurrent Hashing

➢ CCEH [FAST '19]

– Segment reader/writer locks for queries

– Dynamic resizing with segment splitting and directory doubling

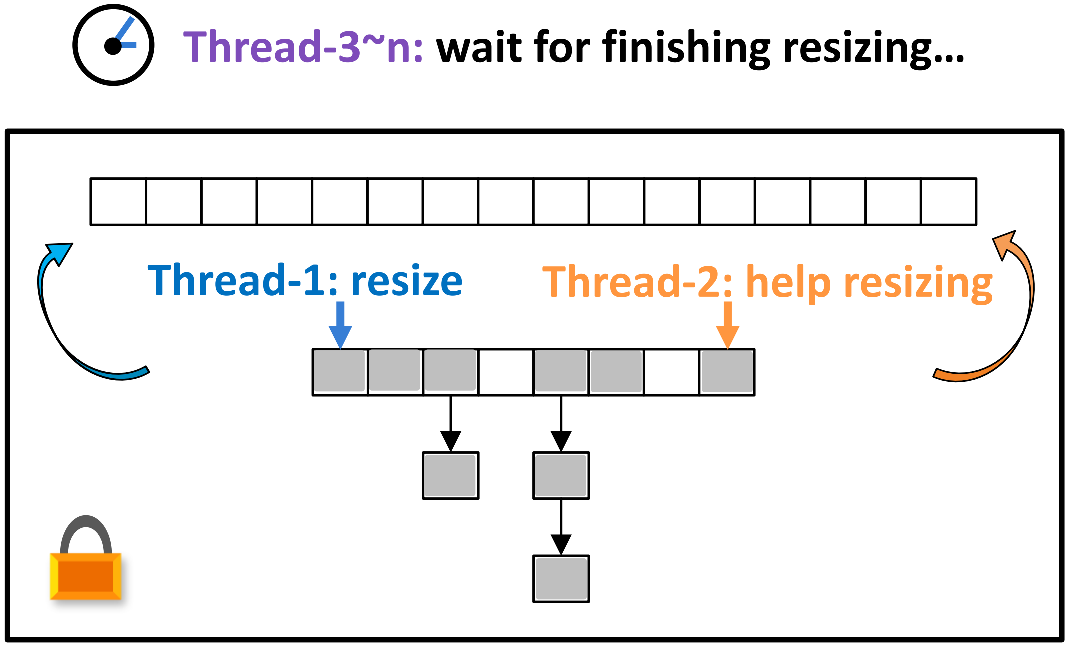


Coarse-grained locks!

➢ P-CLHT [SOSP '19]

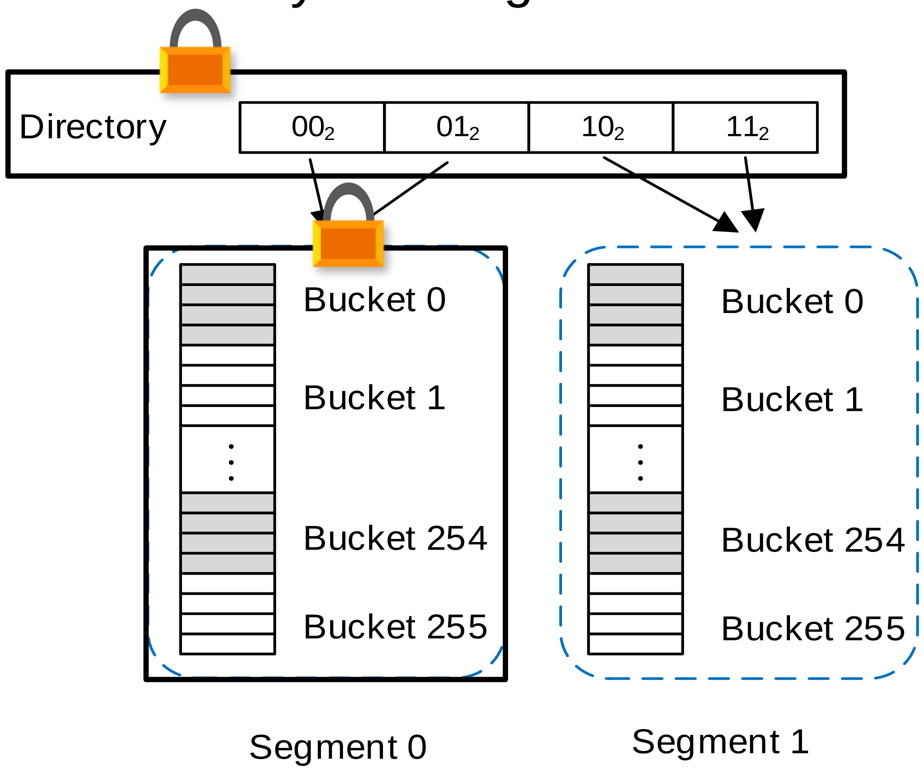
– Lock-free search and bucket lock for writes

– Full-table resizing with one helper thread



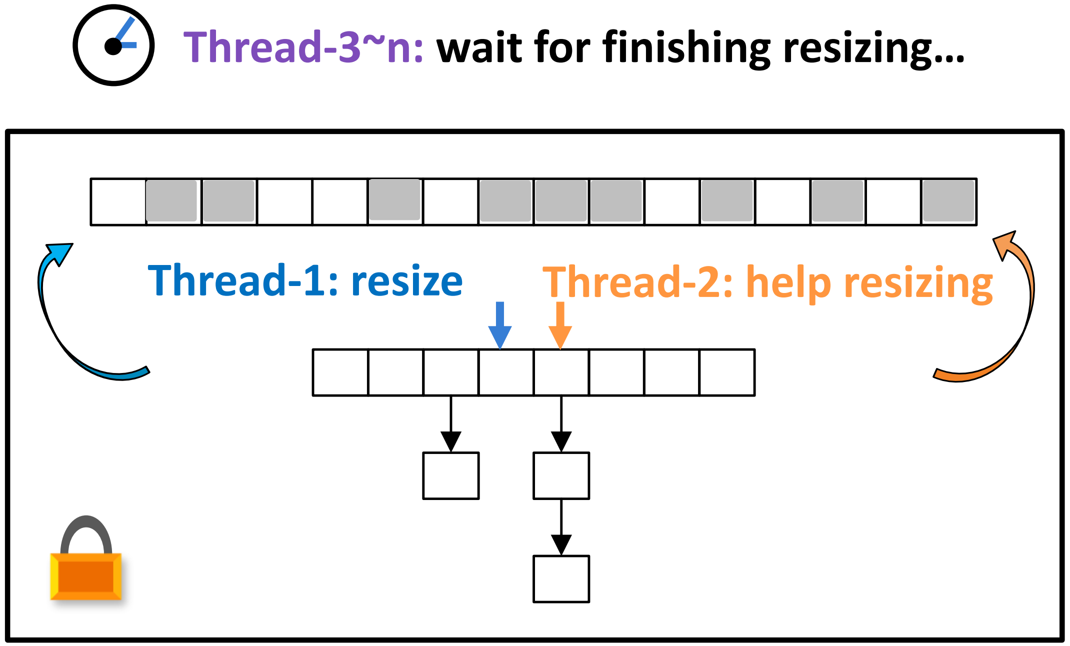
PM Variants of Concurrent Hashing

- CCEH [FAST ’19]
  - Segment reader/writer locks for queries
  - Dynamic resizing with segment splitting and directory doubling



Coarse-grained locks!

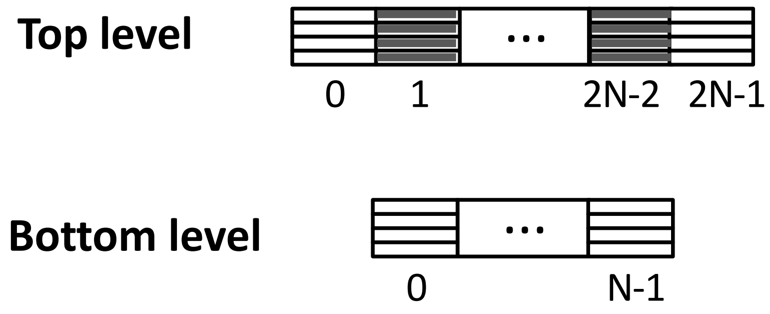
- P-CLHT [SOSP ’19]
  - Lock-free search and bucket lock for writes
  - Full-table resizing with one helper thread



Resizing blocks queries!

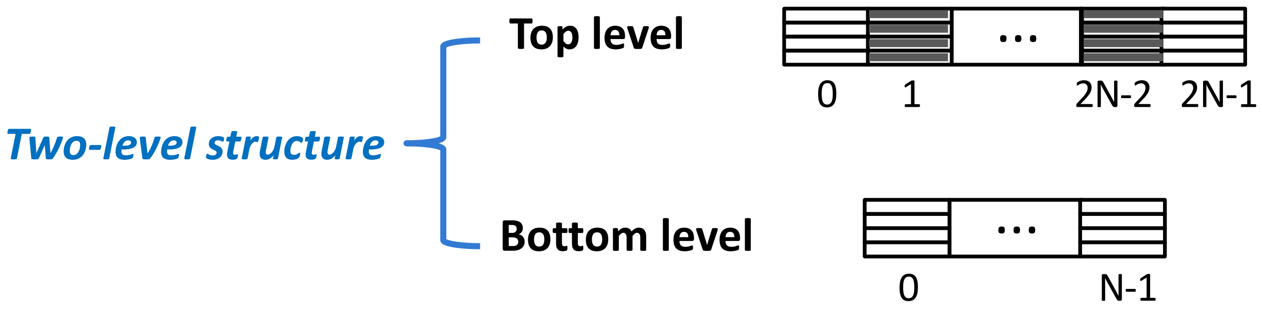
Level Hashing [OSDI '18]

- PM-friendly hashing index



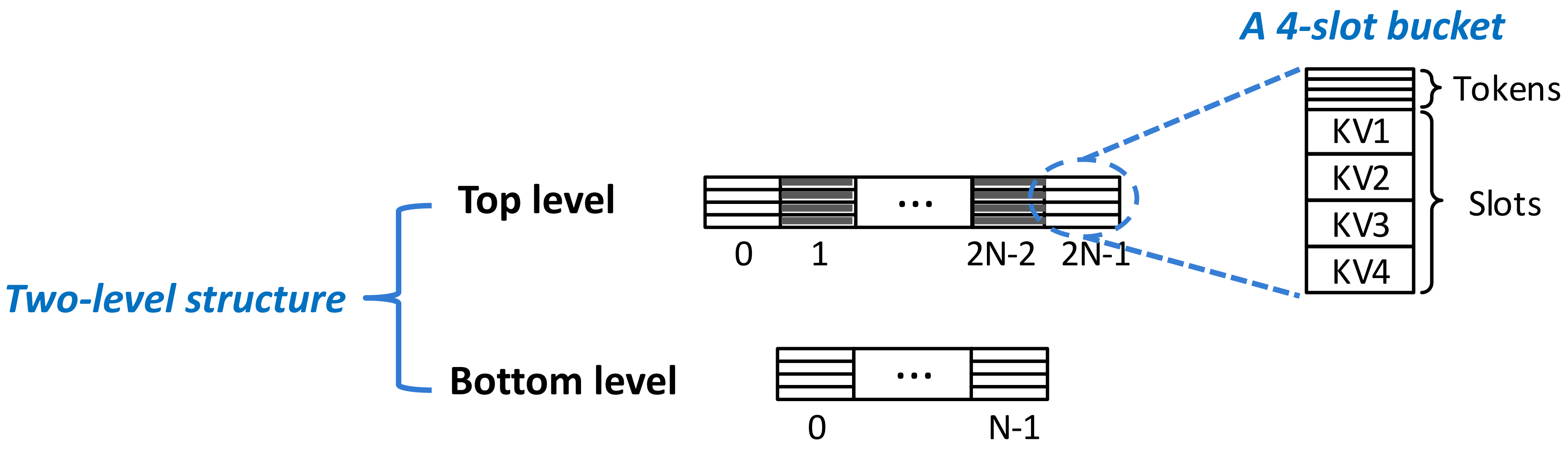
Level Hashing [OSDI '18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement



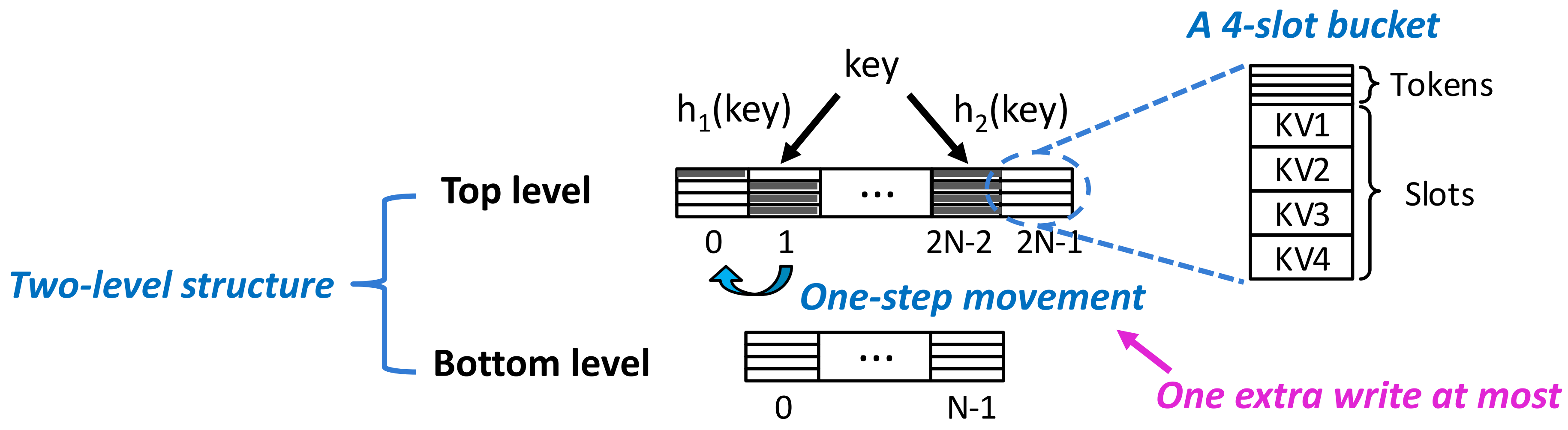
Level Hashing [OSDI '18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement



Level Hashing [OSDI '18]

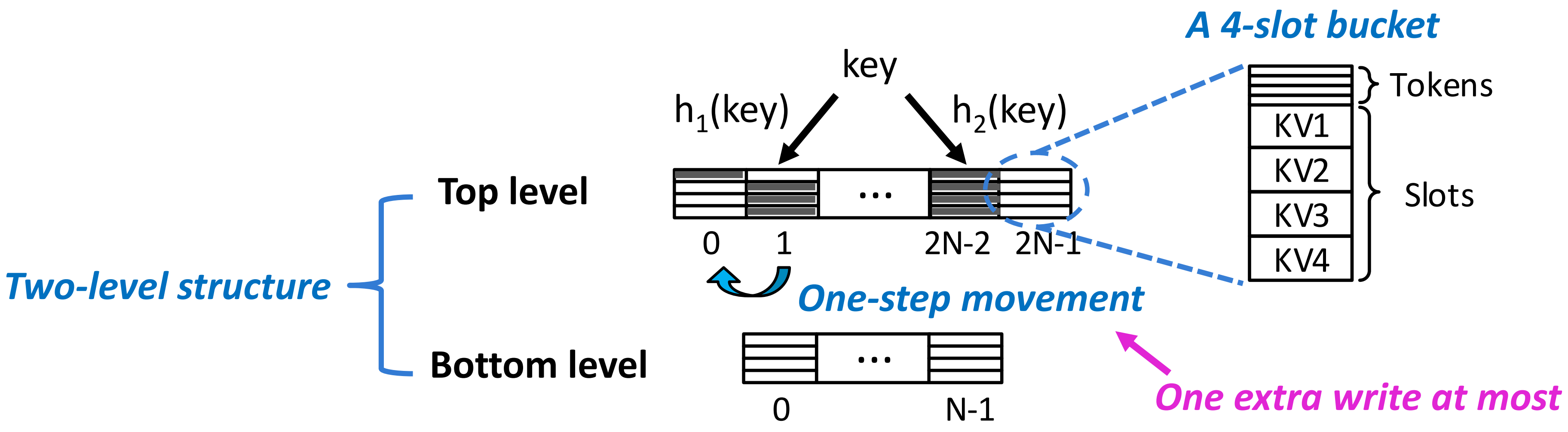
- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement



Level Hashing [OSDI '18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement

✓ *Write efficiency*

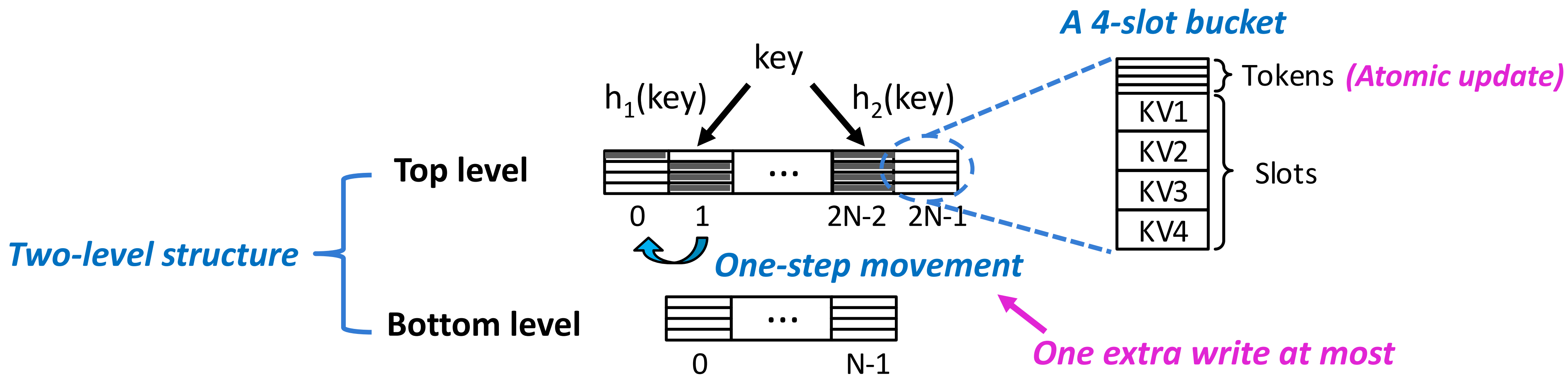


Level Hashing [OSDI '18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement
  - Low-overhead consistency guarantee via atomic token update

✓ ***Write efficiency***

✓ ***Crash consistency***

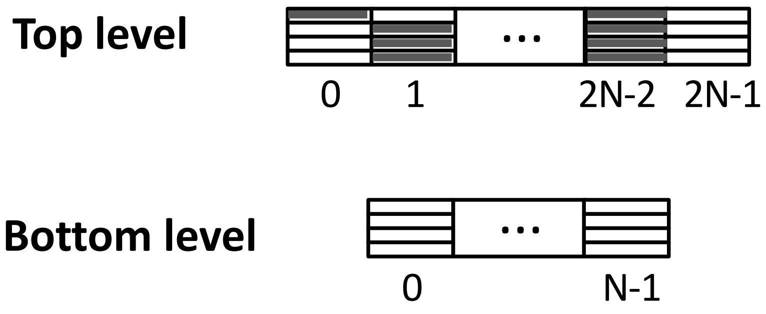


Level Hashing [OSDI '18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement
  - Low-overhead consistency guarantee via atomic token update
  - Rehashing 1/3 buckets for one resizing

✓ ***Write efficiency***

✓ ***Crash consistency***

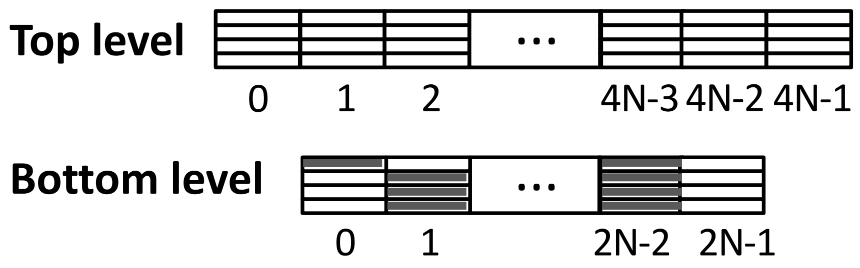


Level Hashing [OSDI ’18]

- PM-friendly hashing index
  - Two-level bucketized hash table with one-step movement
  - Low-overhead consistency guarantee via atomic token update
  - Rehashing 1/3 buckets for one resizing

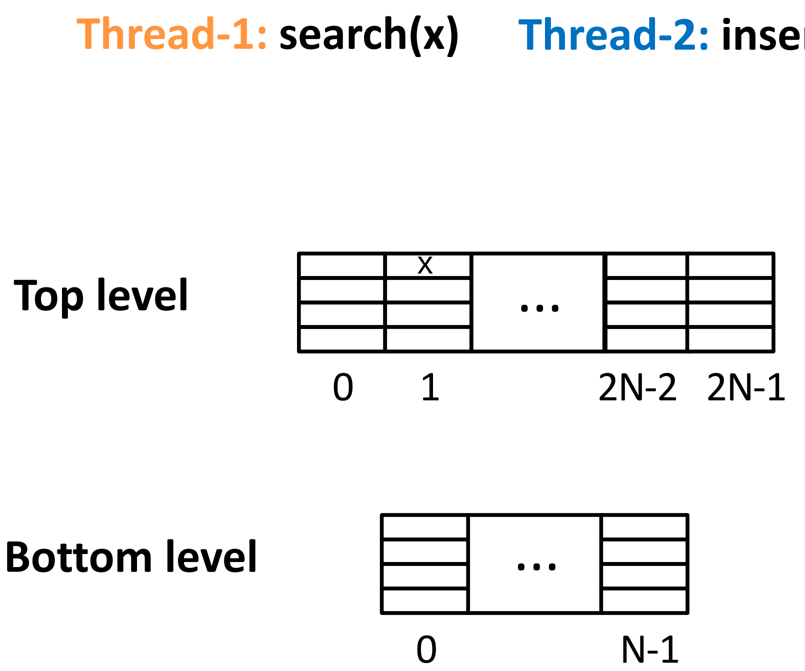
✓ ***Write efficiency***

✓ ***Crash consistency***

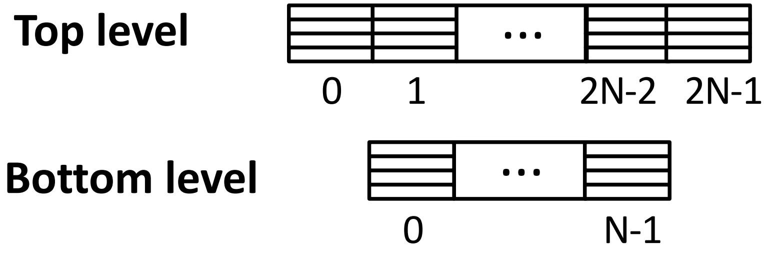


Concurrency in Level Hashing

- Slot-grained lock for queries



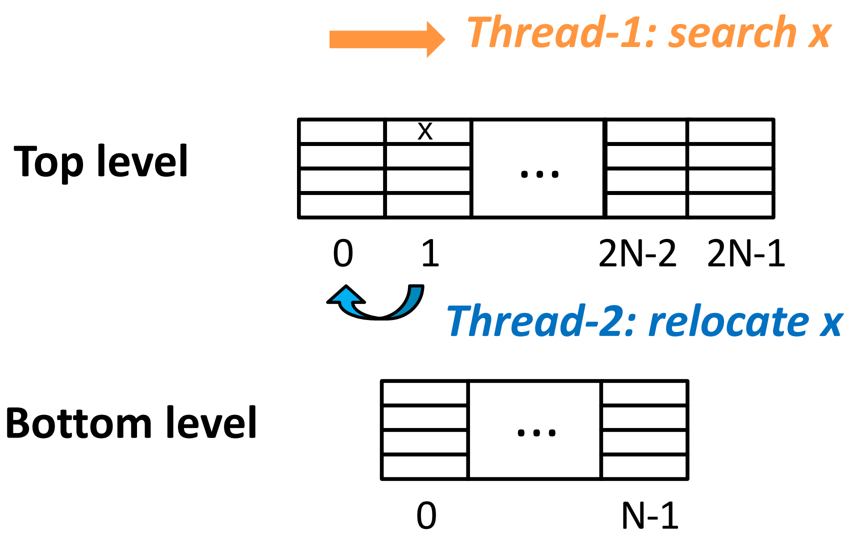
- Single-thread blocking resizing



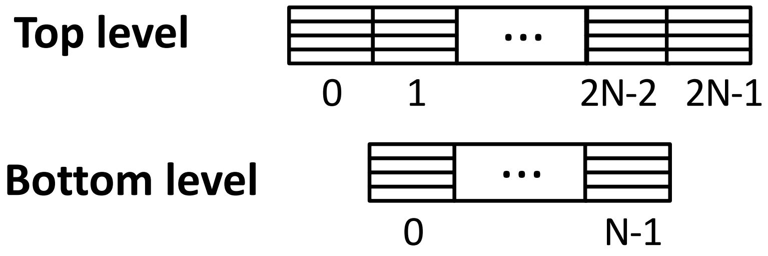
Concurrency in Level Hashing

- Slot-grained lock for queries

Thread-1: search(x)   Thread-2: insert(key)



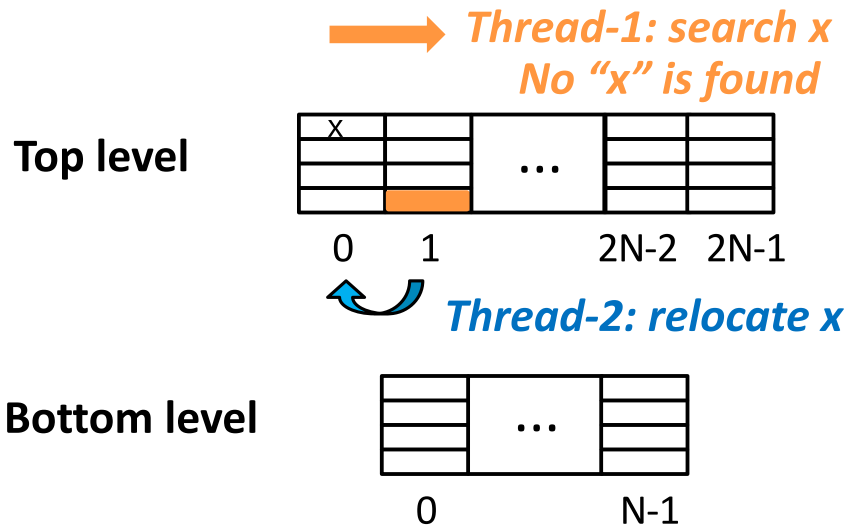
- Single-thread blocking resizing



Concurrency in Level Hashing

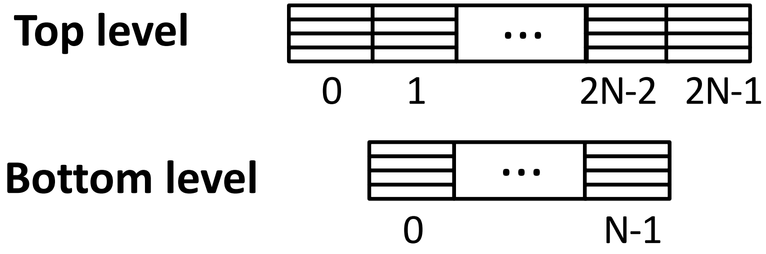
- Slot-grained lock for queries

Thread-1: **search(x)** **Thread-2:** **insert(key)**



Missing inserted items!

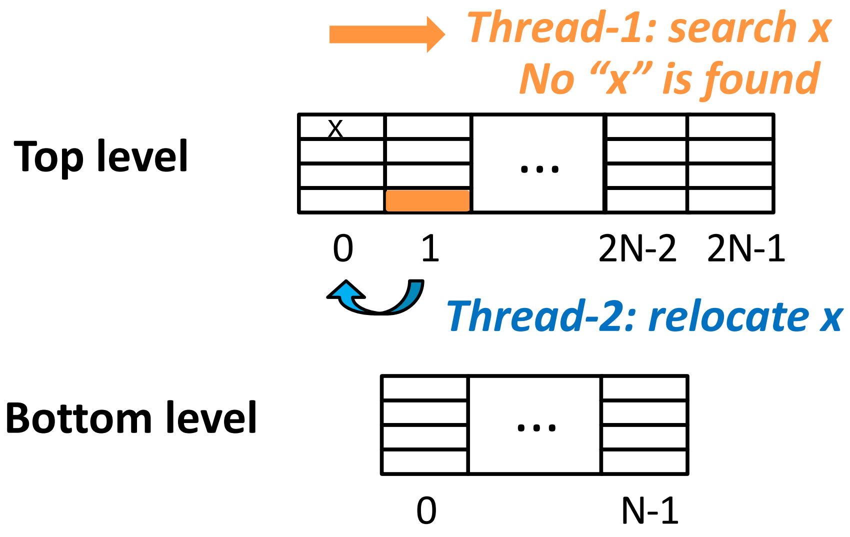
- Single-thread blocking resizing



Concurrency in Level Hashing

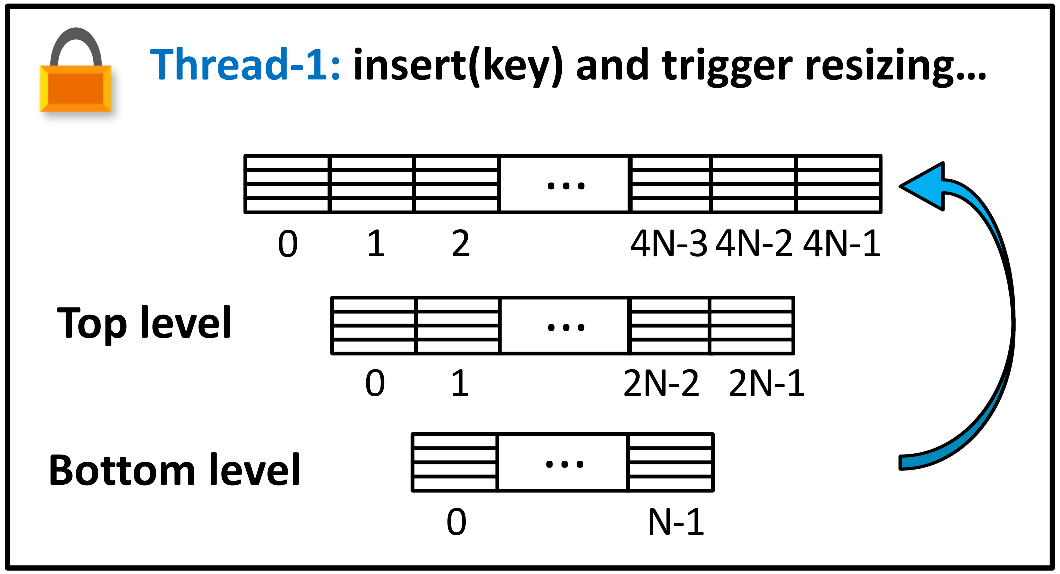
- Slot-grained lock for queries

Thread-1: **search(x)** Thread-2: **insert(key)**



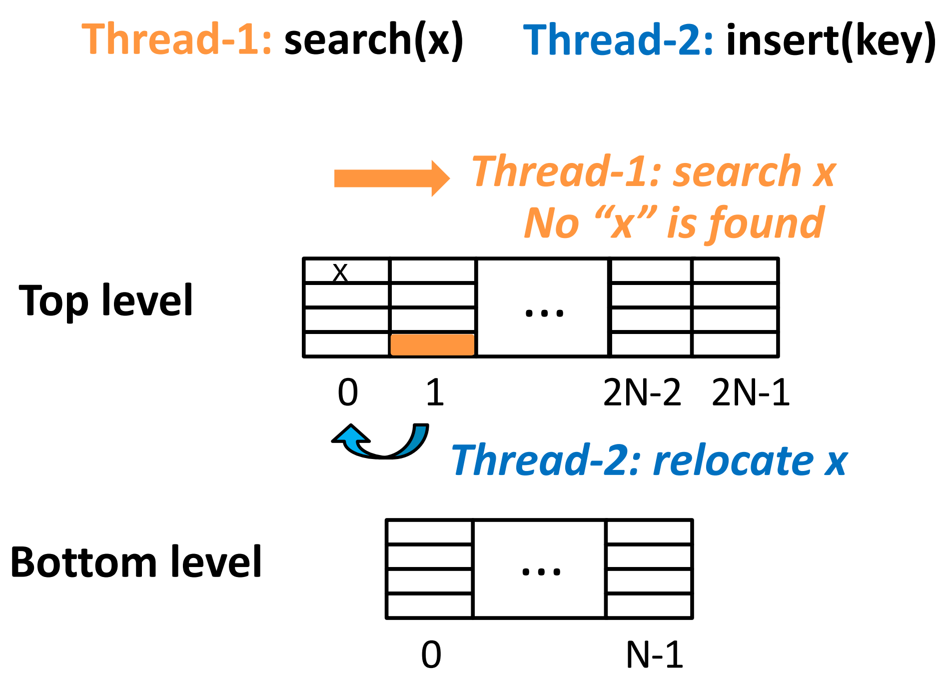
Missing inserted items!

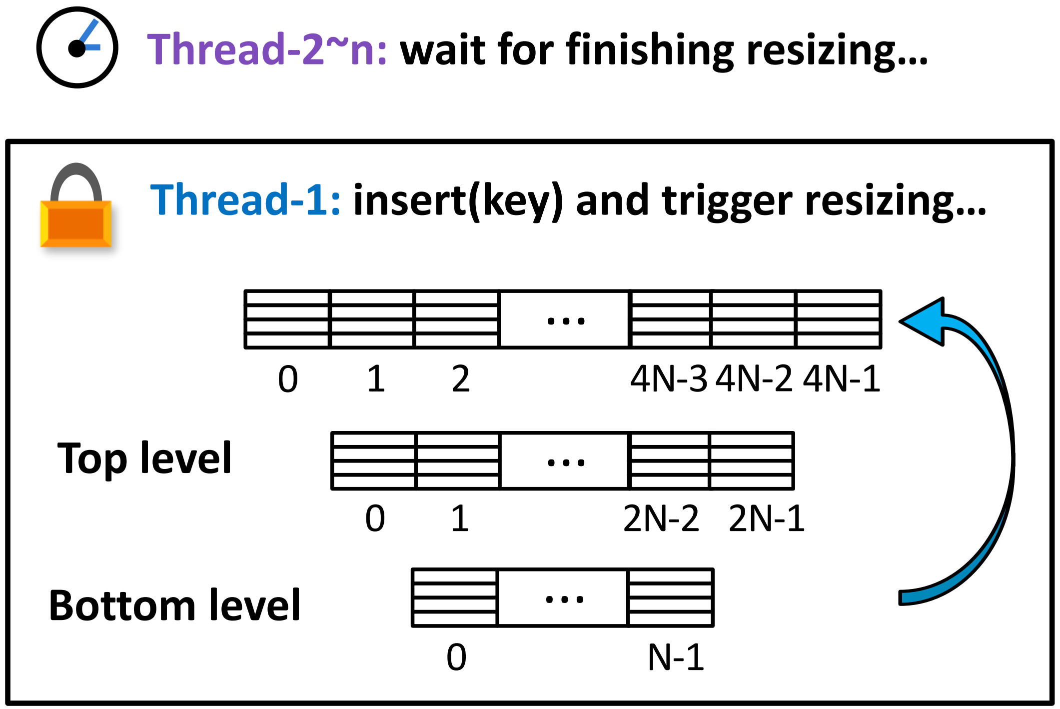
- Single-thread blocking resizing



Concurrency in Level Hashing

- Slot-grained lock for queries
- Single-thread blocking resizing



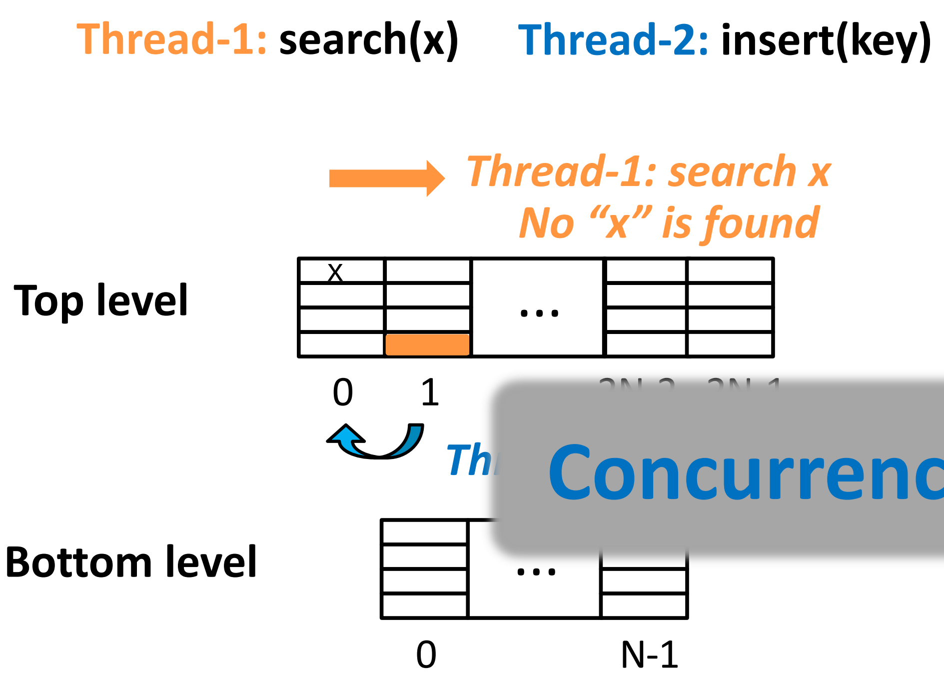


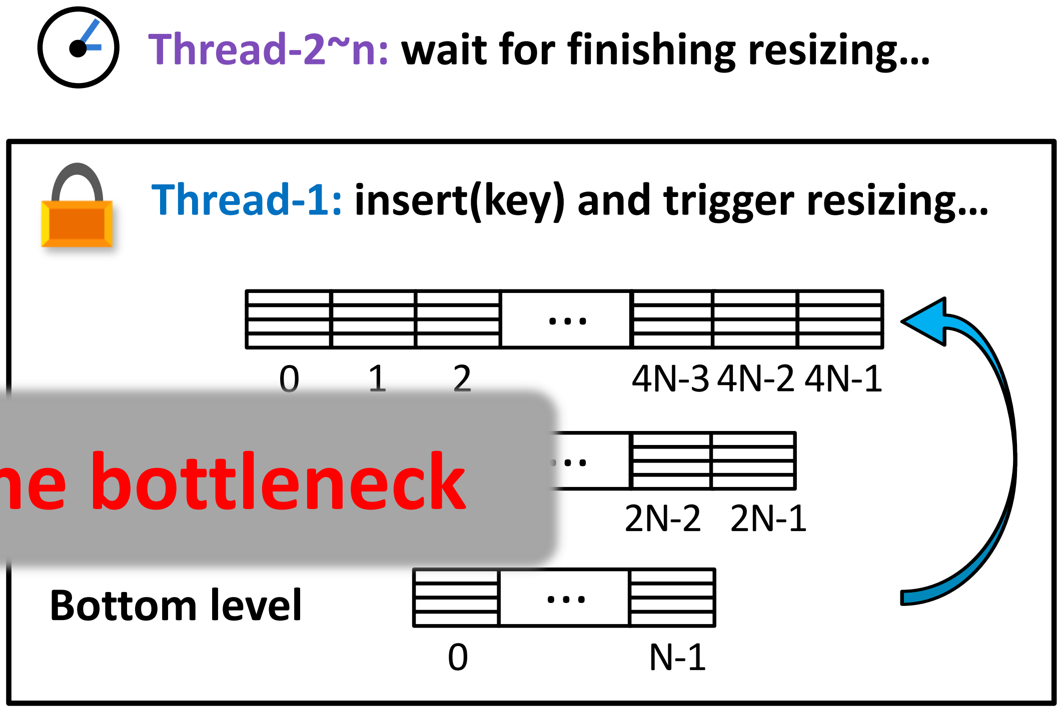
Missing inserted items!

Resizing blocks queries!

Concurrency in Level Hashing

- ➢ Slot-grained lock for queries
- ➢ Single-thread blocking resizing





Concurrency is the bottleneck

Missing inserted items!

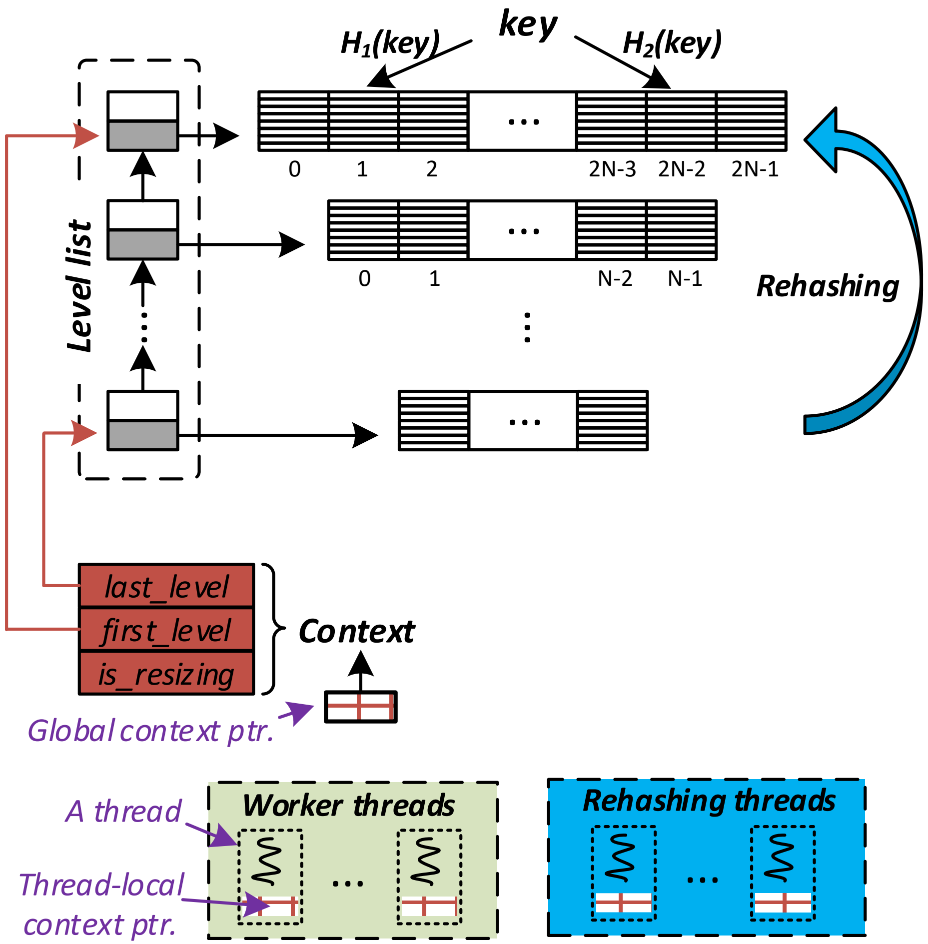
Resizing blocks queries!

Challenges for PM Hashing

- Challenges
  - Performance degradation for blocking resizing
    - High latency for coarse-grained locks
  - Limited scalability for lock-based concurrency control
    - Lock constraint for concurrent accesses
    - Persisting overheads in the critical path

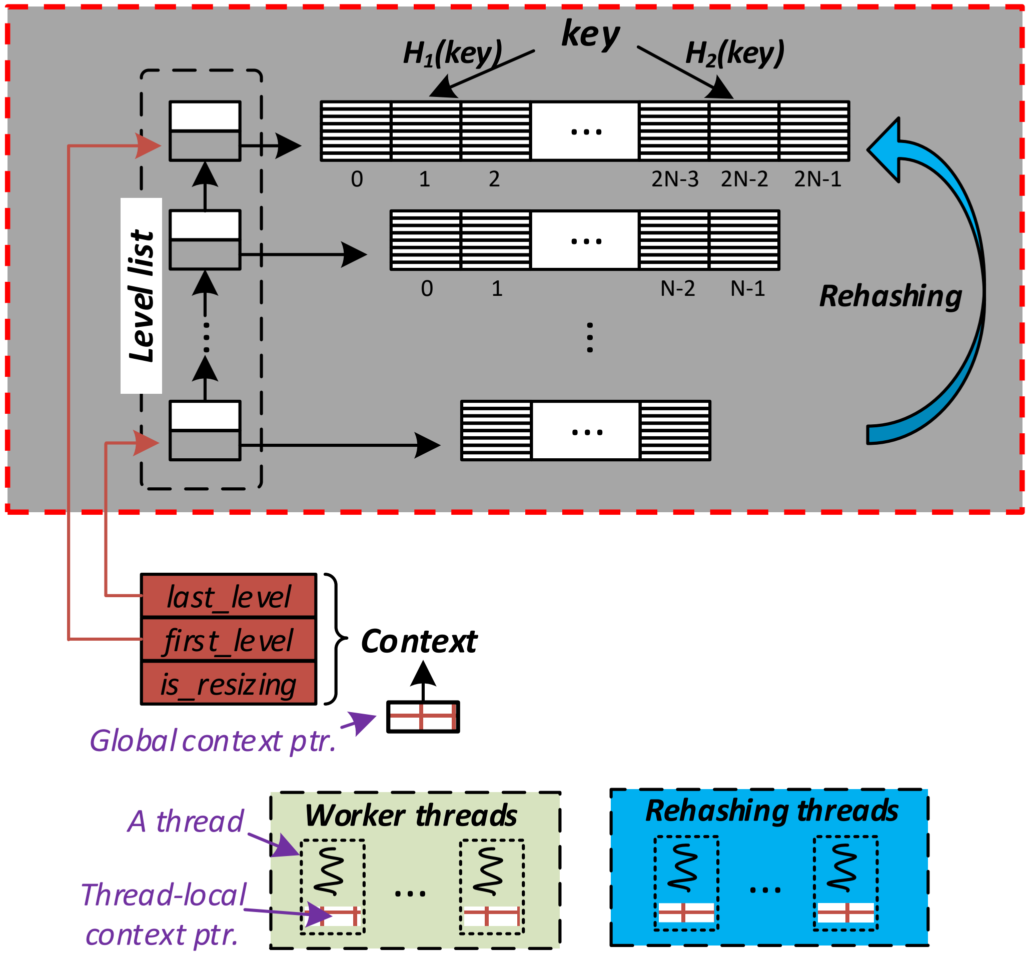
- Design goals
  - A PM-friendly and high-concurrency hashing scheme

Our Approach: Clevel Hashing



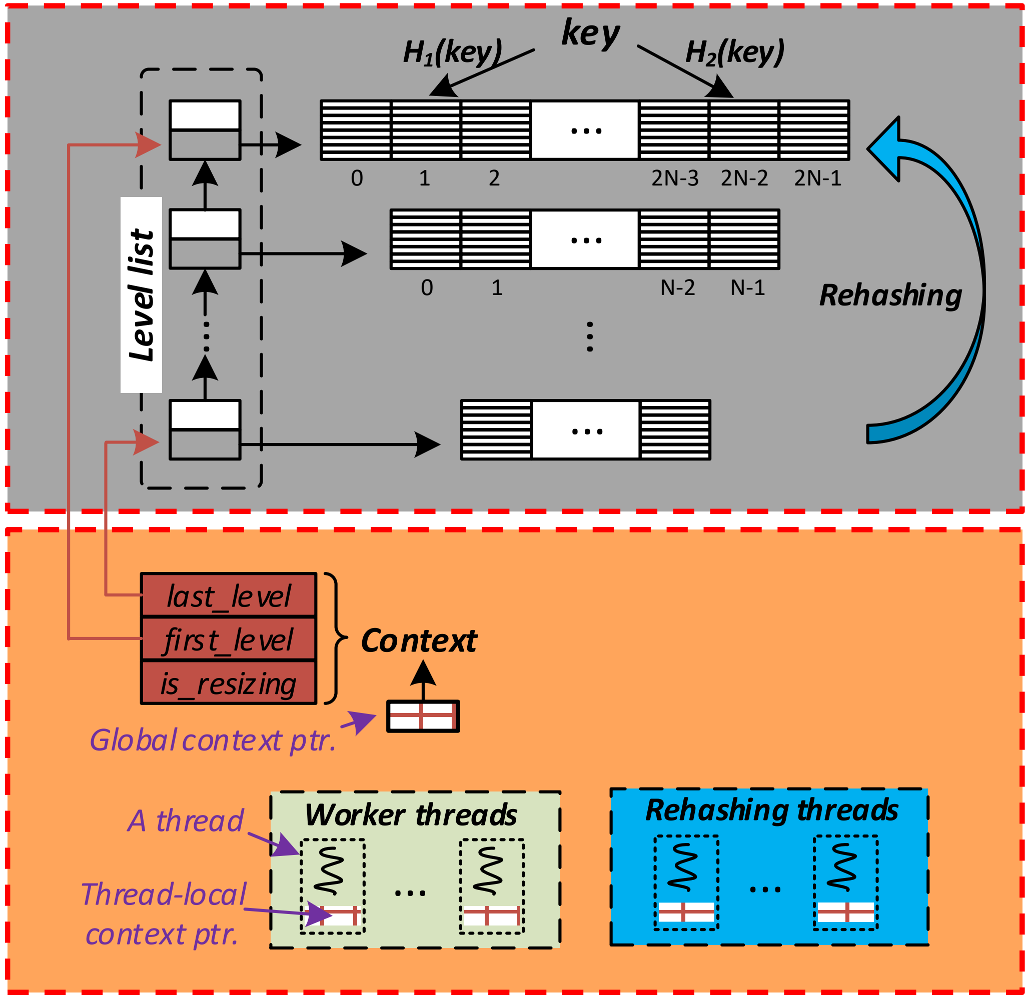
Our Approach: Clevel Hashing

- Dynamic multi-level structure w/o extra writes for insertion
  - ✓ Write-optimal insertion



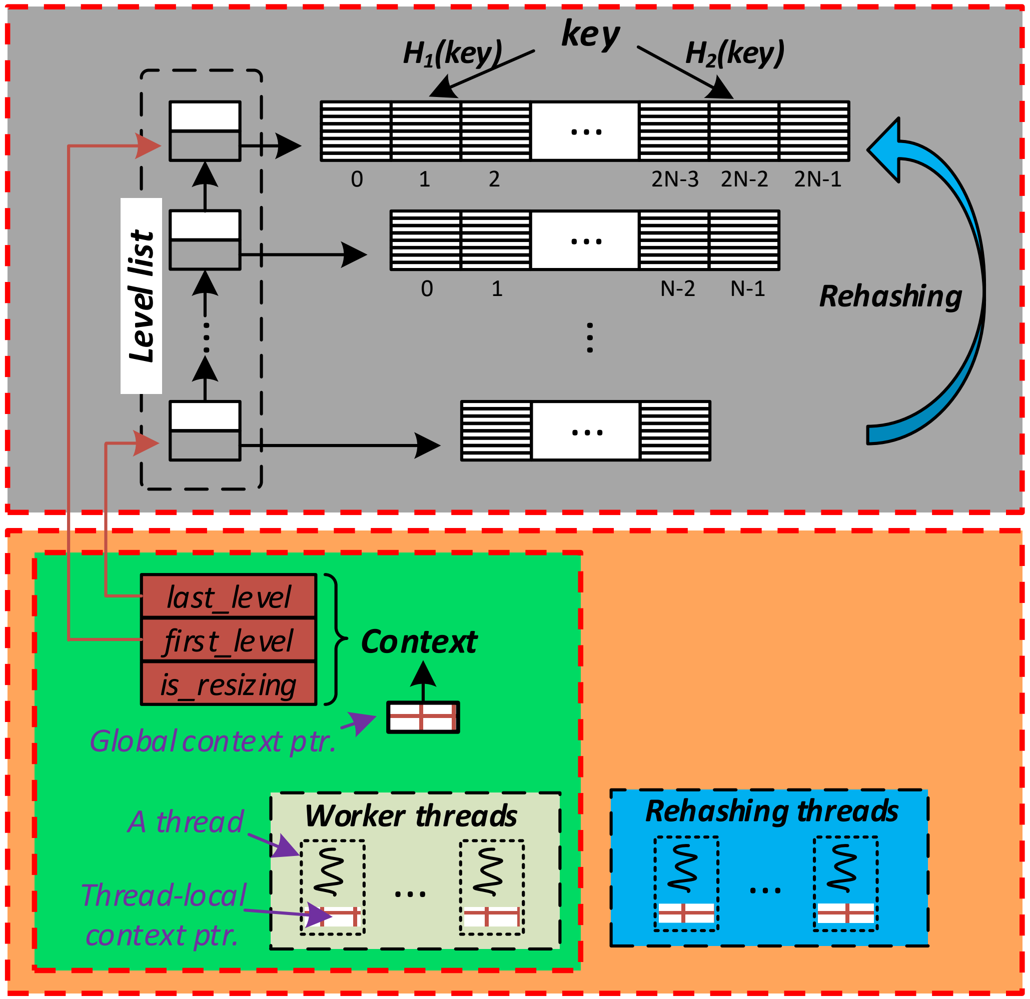
Our Approach: Clevel Hashing

- ➢ Dynamic multi-level structure w/o extra writes for insertion
  - ✓ Write-optimal insertion
- ➢ Asynchronous rehashing w/o blocking concurrent queries
  - ✓ Non-blocking resizing



Our Approach: Clevel Hashing

- Dynamic multi-level structure w/o extra writes for insertion
  - ✓ Write-optimal insertion
- Asynchronous rehashing w/o blocking concurrent queries
  - ✓ Non-blocking resizing
- Lock-free concurrency control
  - ✓ Lock-free queries



Components

- Dynamic Multi-level Structure

- Non-blocking Resizing

- Lock-free Concurrency Control

Components

- Dynamic Multi-level Structure
- Non-blocking Resizing
- Lock-free Concurrency Control

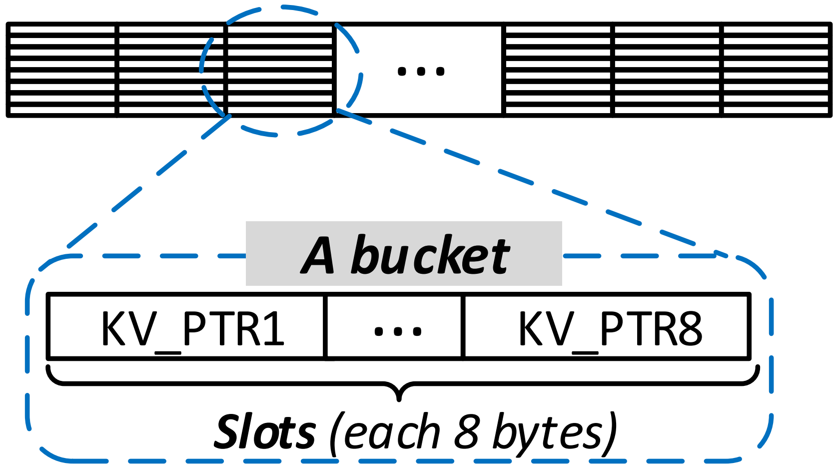
Dynamic Multi-level Structure

- Support for variable-length items
  - Store pointers in slots and actual items outside of the table

Dynamic Multi-level Structure

- Support for variable-length items

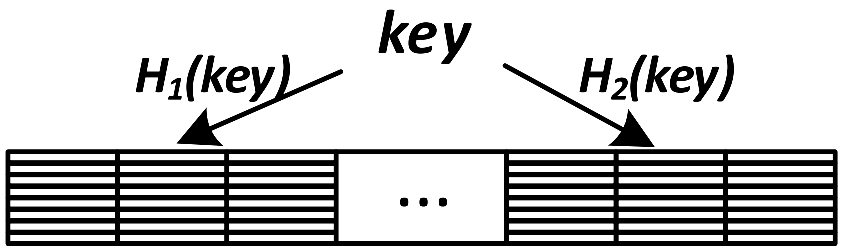
- Write-optimized hash table
  - 8 slots per bucket



Dynamic Multi-level Structure

- Support for variable-length items

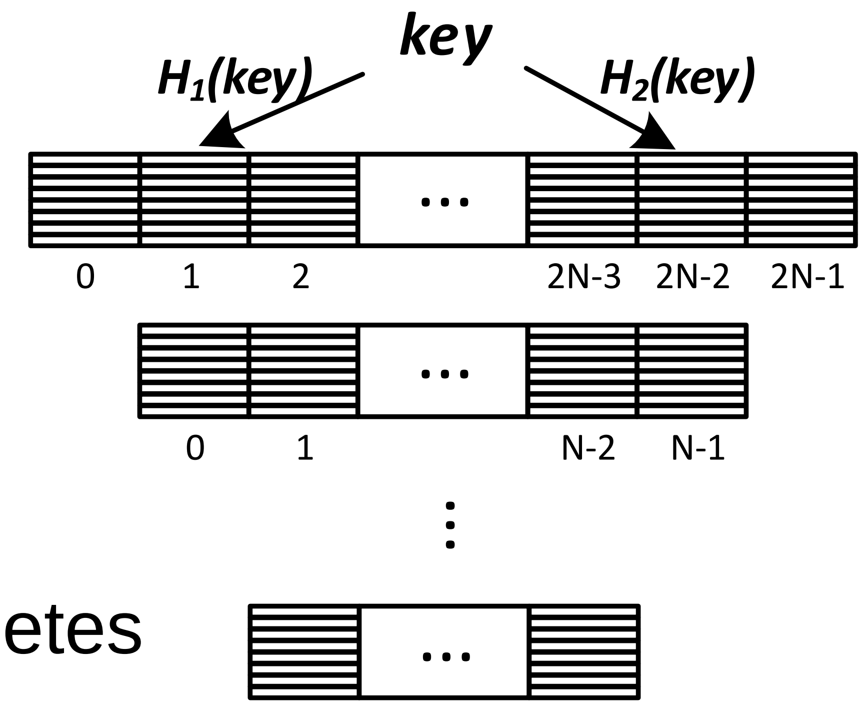
- Write-optimized hash table
  - 8 slots per bucket
  - 2 candidate buckets in one level



Dynamic Multi-level Structure

- Support for variable-length items

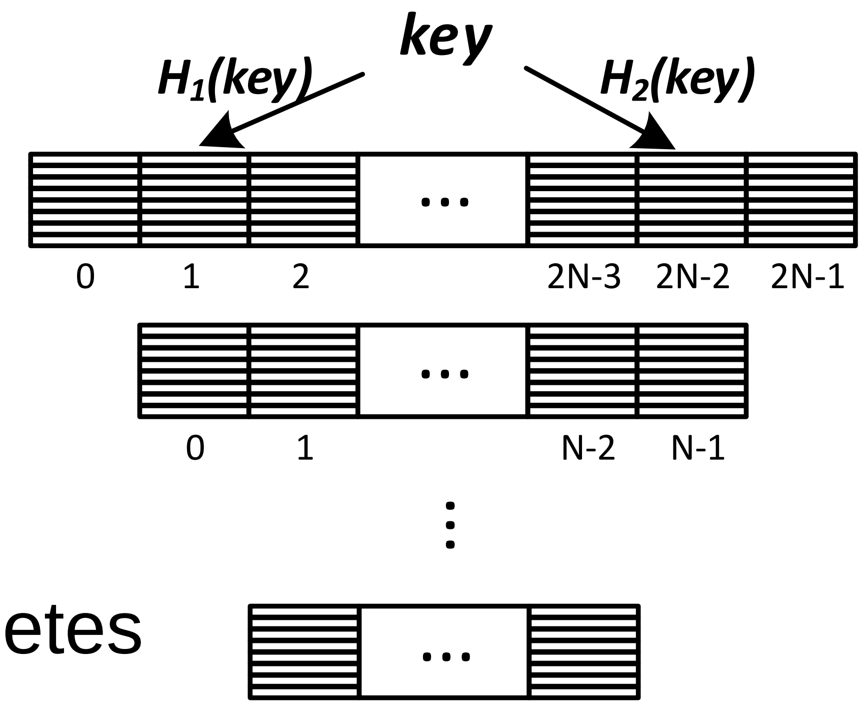
- Write-optimized hash table
  - 8 slots per bucket
  - 2 candidate buckets in one level
  - Sharing-based multiple levels
    - Add a level for resizing
    - Remove one when rehashing completes



Dynamic Multi-level Structure

- Support for variable-length items

- Write-optimized hash table
  - 8 slots per bucket
  - 2 candidate buckets in one level
  - Sharing-based multiple levels
    - Add a level for resizing
    - Remove one when rehashing completes

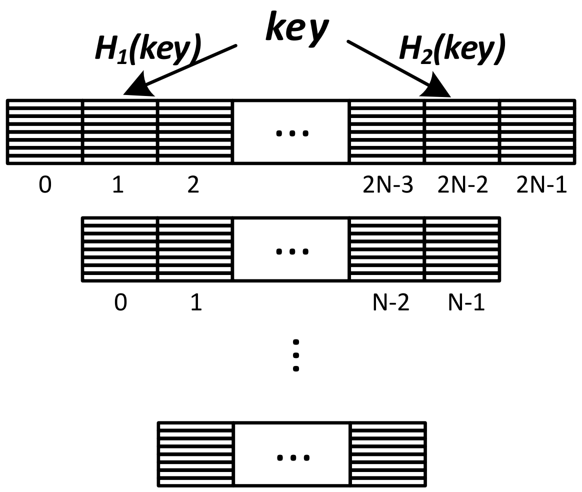


No extra writes for insertion ➡ ***Write-optimal***

Components

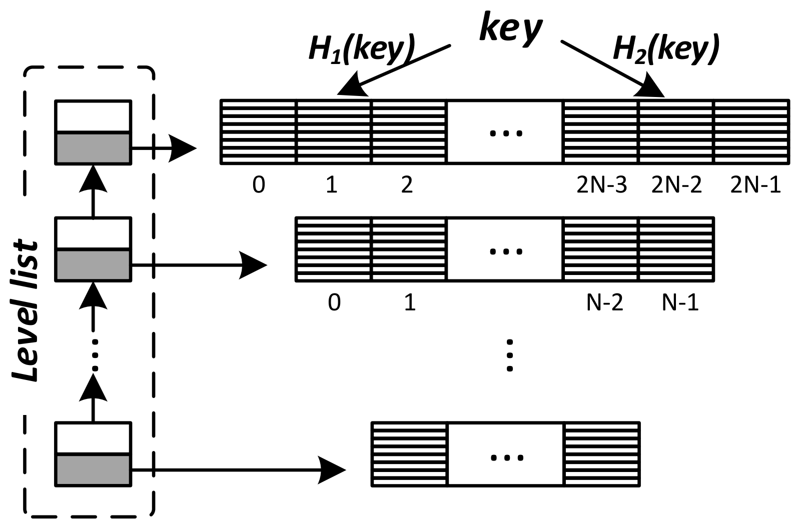
- Dynamic Multi-level Structure
- Non-blocking Resizing
- Lock-free Concurrency Control

The Support for Concurrent Resizing



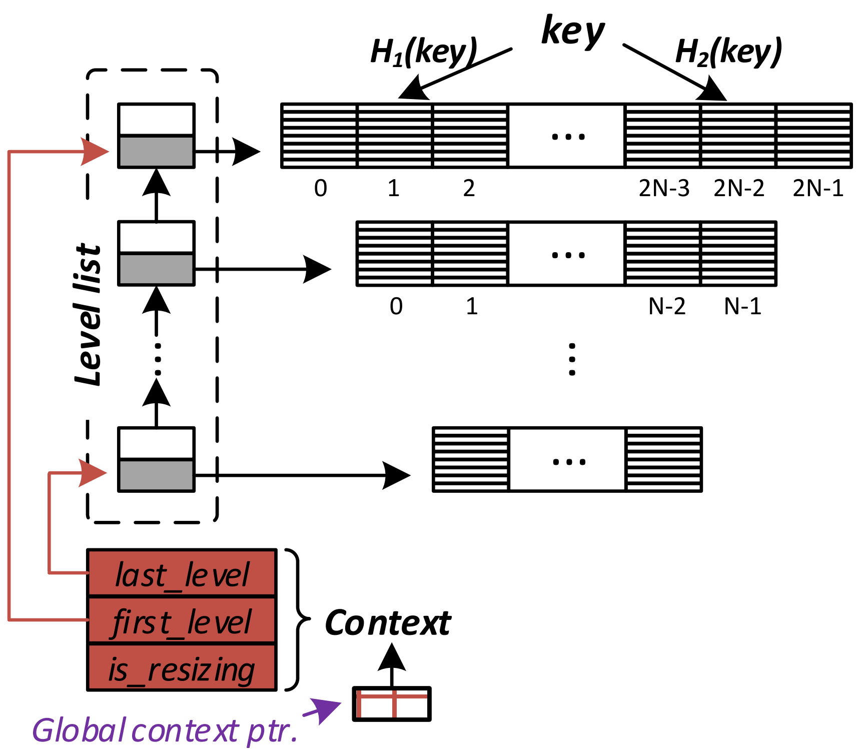
The Support for Concurrent Resizing

- Level list
  - A linked list to associate levels



The Support for Concurrent Resizing

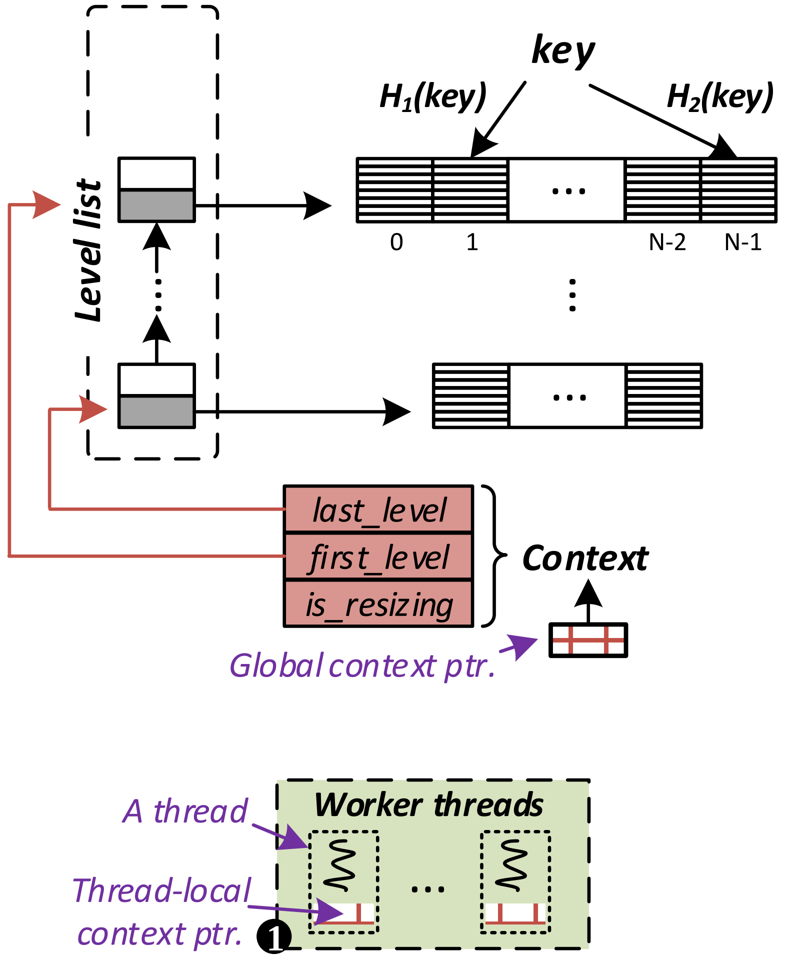
- Level list
  - A linked list to associate levels
- Context
  - A metadata structure including:
    - *first_level* (the largest level)
    - *last_level*
    - *is_resizing*



Non-blocking Resizing

➢ Resizing steps

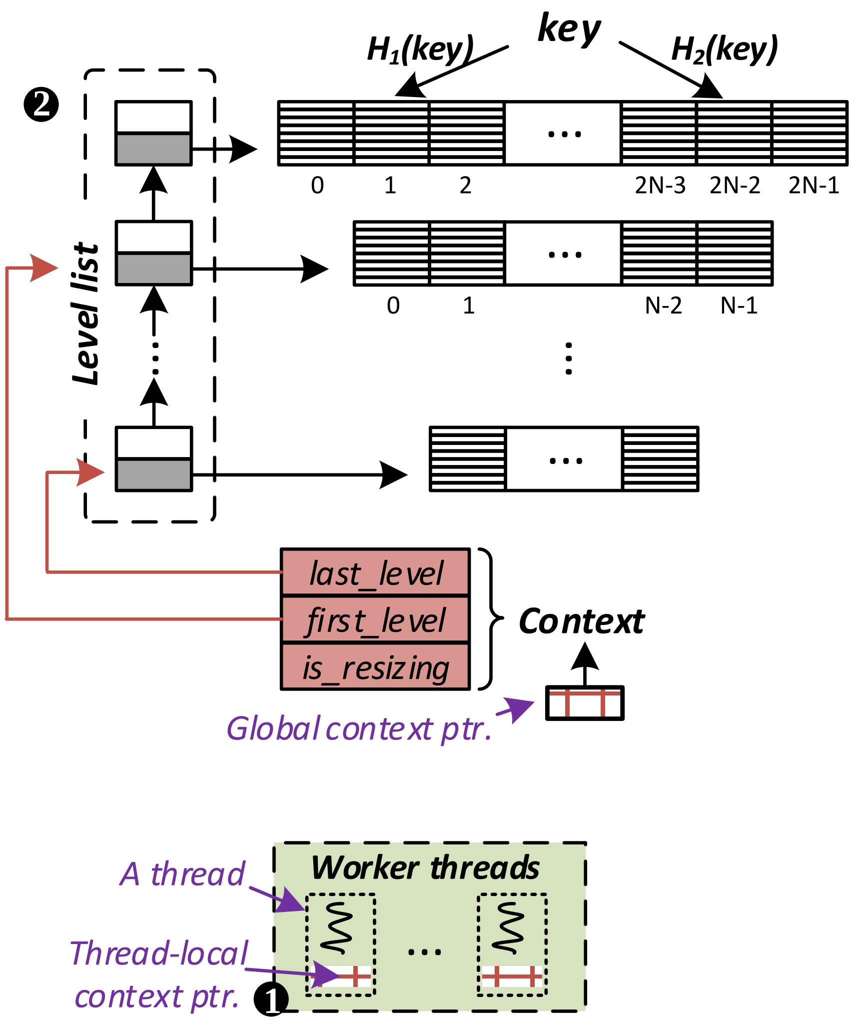
1. Make a local copy of the global context pointer



Non-blocking Resizing

➢ Resizing steps

1. Make a local copy of the global context pointer
2. CAS to append a new level



Non-blocking Resizing

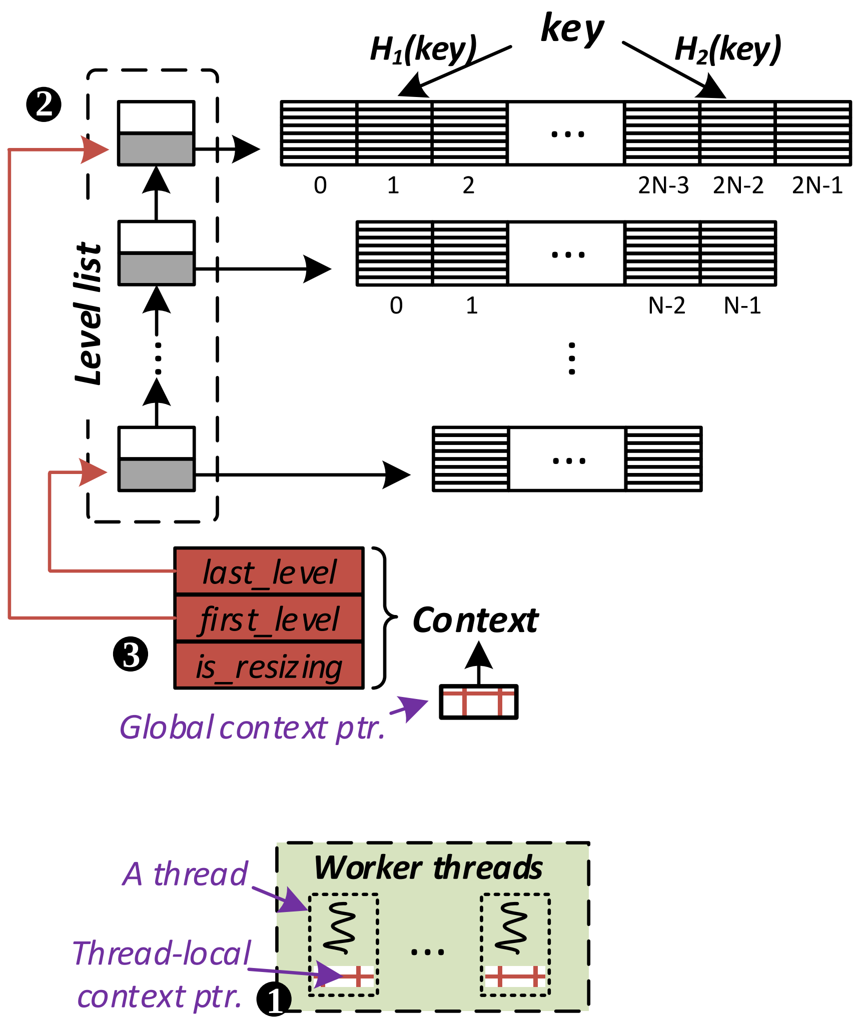
➢ Resizing steps

1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*

last_level	8 B
first_level	8 B
is_resizing	1 B

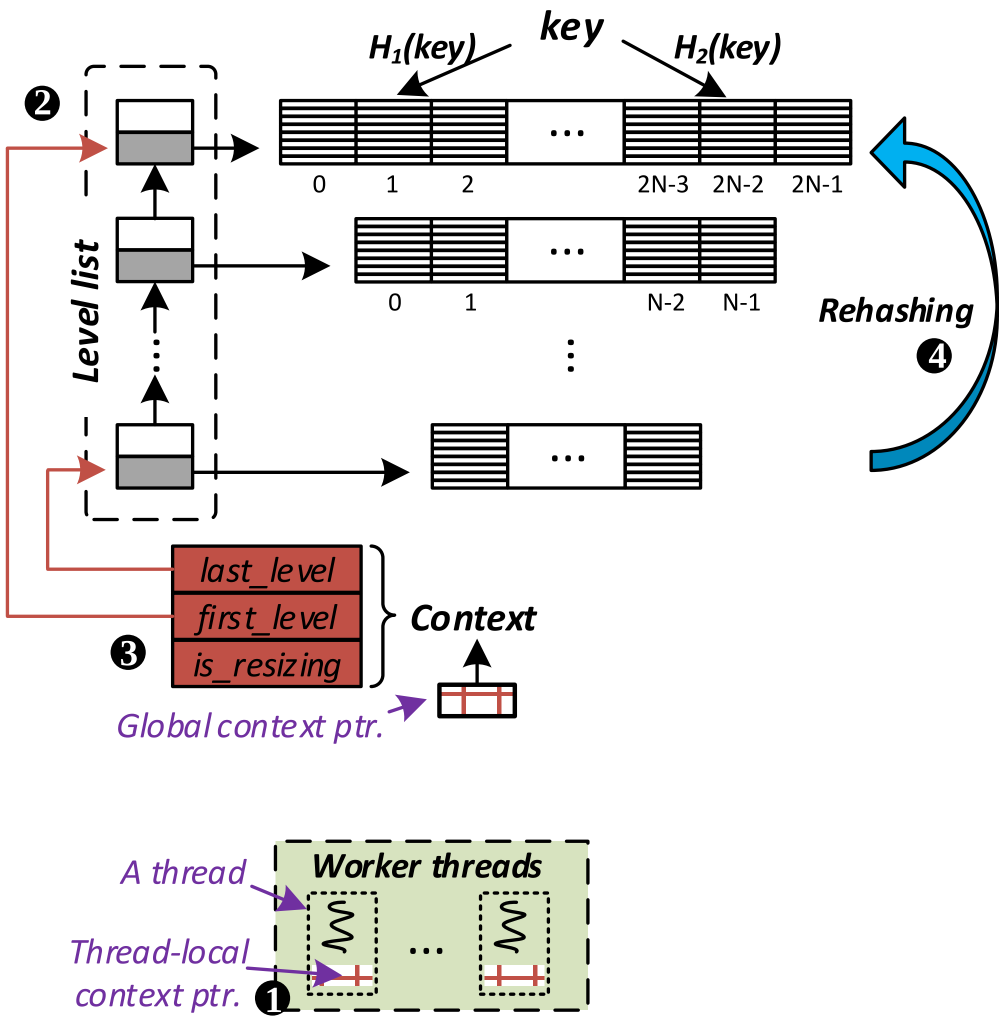
Context size: 17 bytes

Lightweight CoW



Non-blocking Resizing

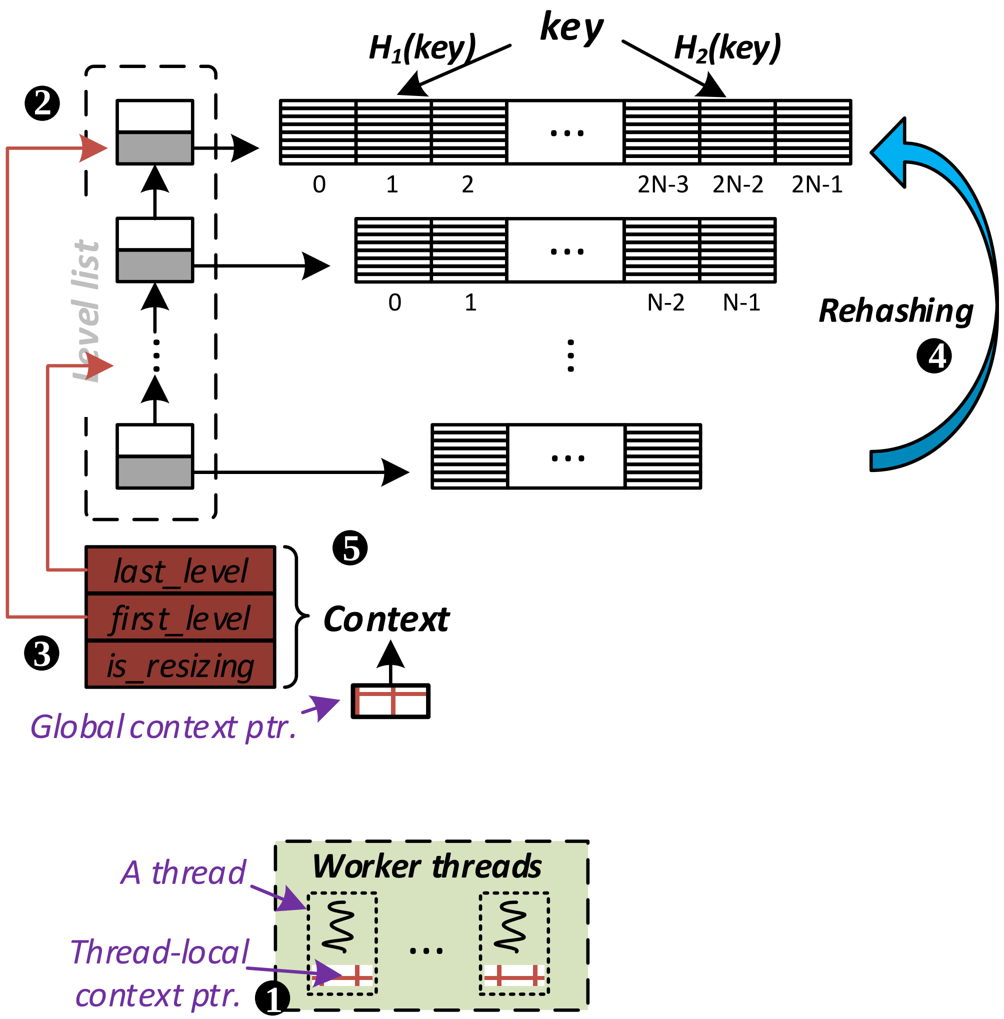
- Resizing steps
  1. Make a local copy of the global context pointer
  2. CAS to append a new level
  3. CoW + CAS to update the *first_level*
  4. Rehash items in the last level



Non-blocking Resizing

➢ Resizing steps

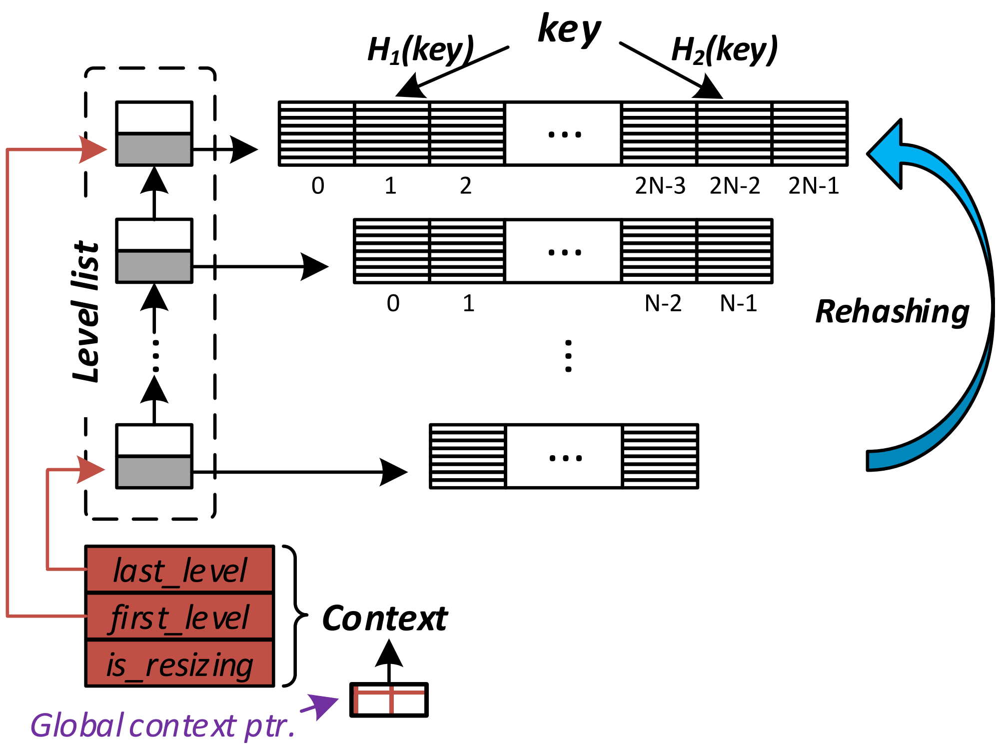
1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*
4. Rehash items in the last level
5. CoW + CAS to update the *last_level*



Non-blocking Resizing

➢ Resizing steps

1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*
4. Rehash items in the last level
5. CoW + CAS to update the *last_level*

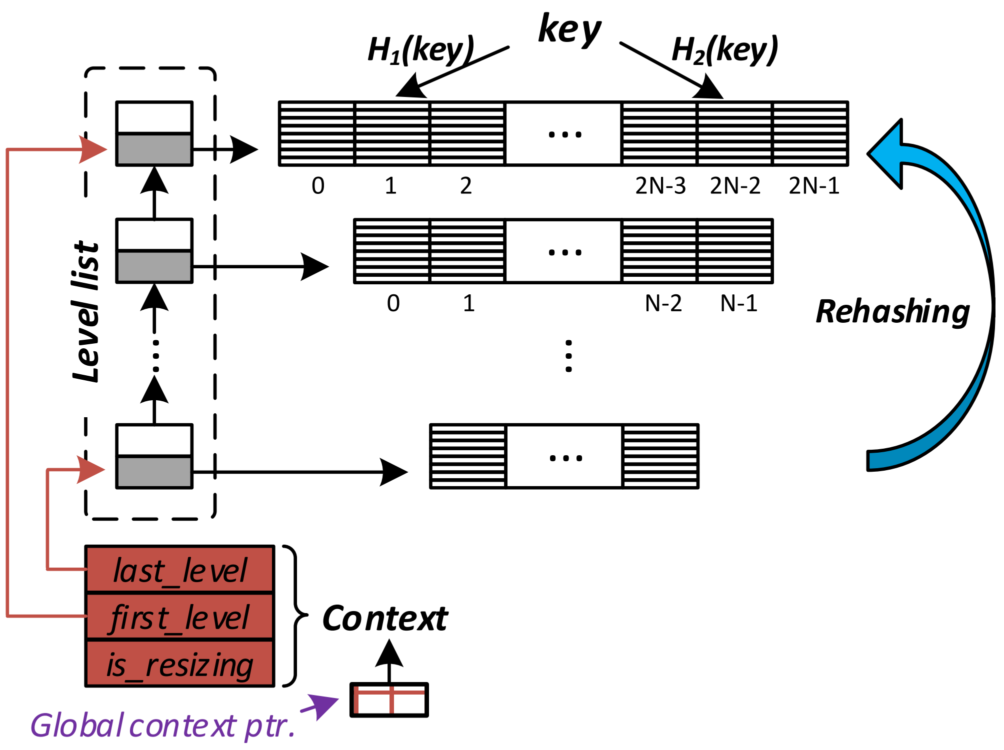


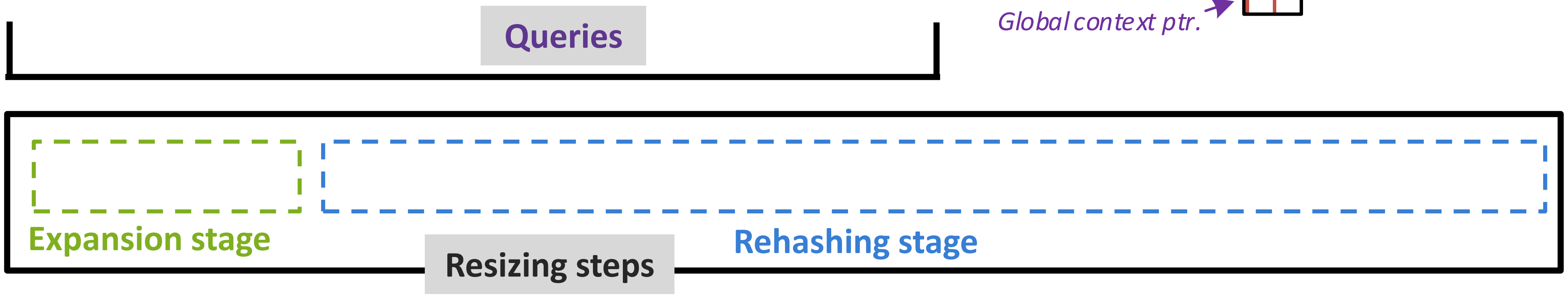
Non-blocking Resizing

- Resizing steps

Expansion stage
1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*

Rehashing stage
4. Rehash items in the last level
5. CoW + CAS to update the *last_level*





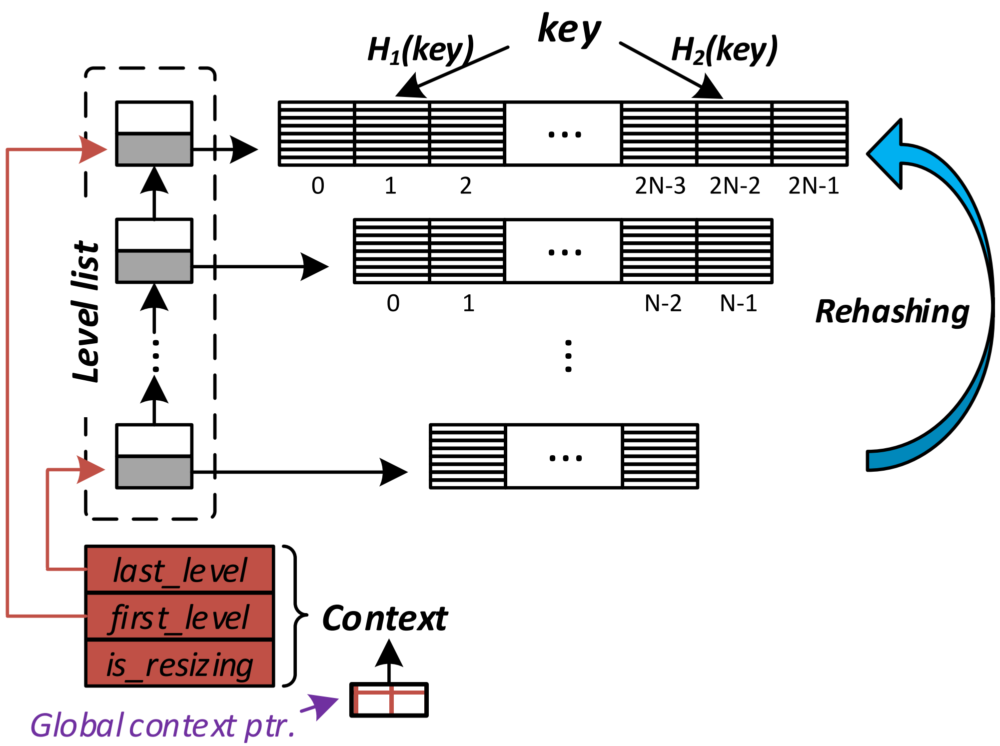
Non-blocking Resizing

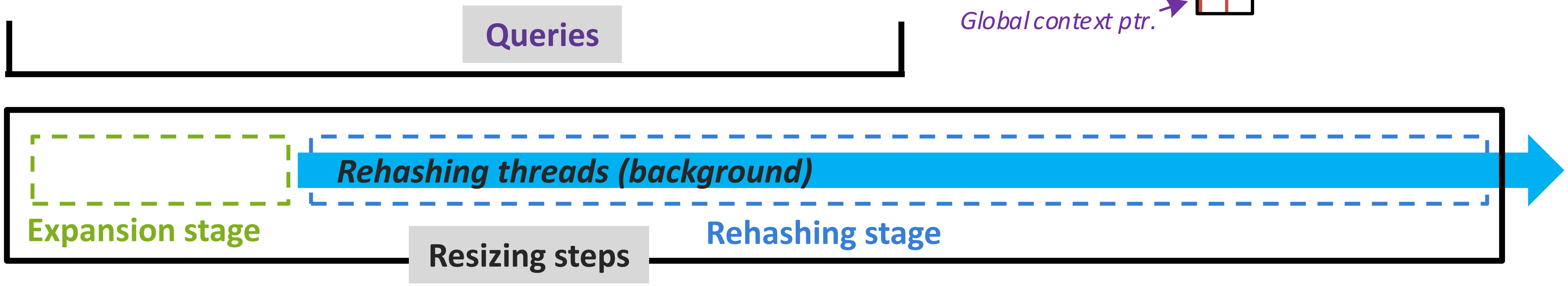
- Resizing steps

Expansion stage
1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*

Rehashing stage
4. Rehash items in the last level
5. CoW + CAS to update the *last_level*

- Non-blocking resizing scheme
  - Rehashing threads: rehash until there are 2 levels left





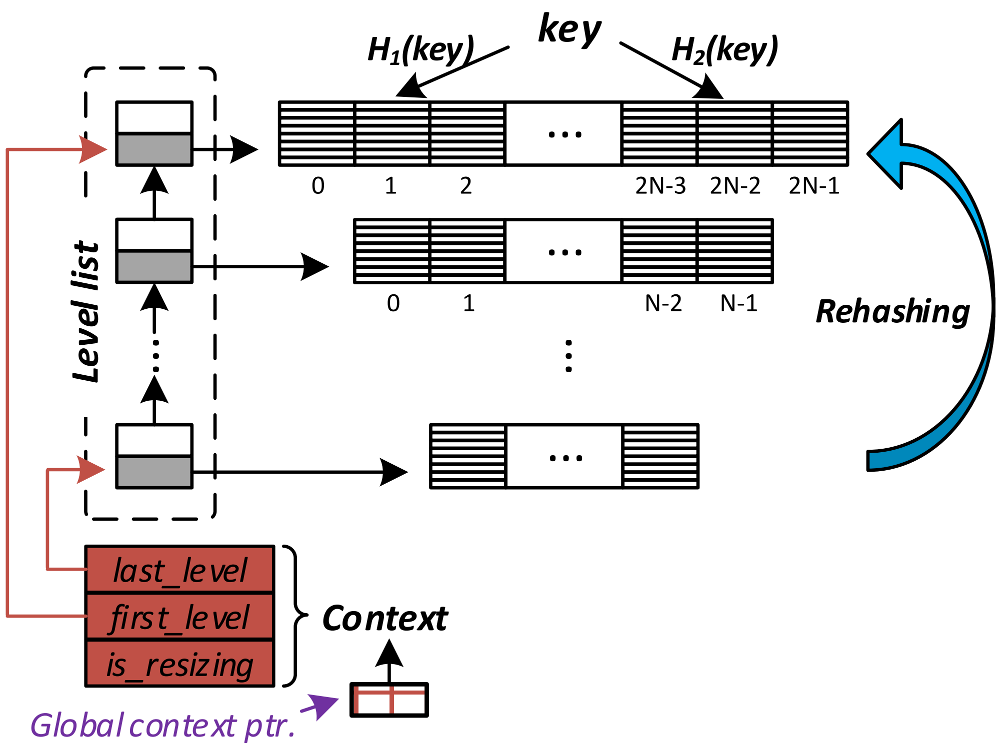
Non-blocking Resizing

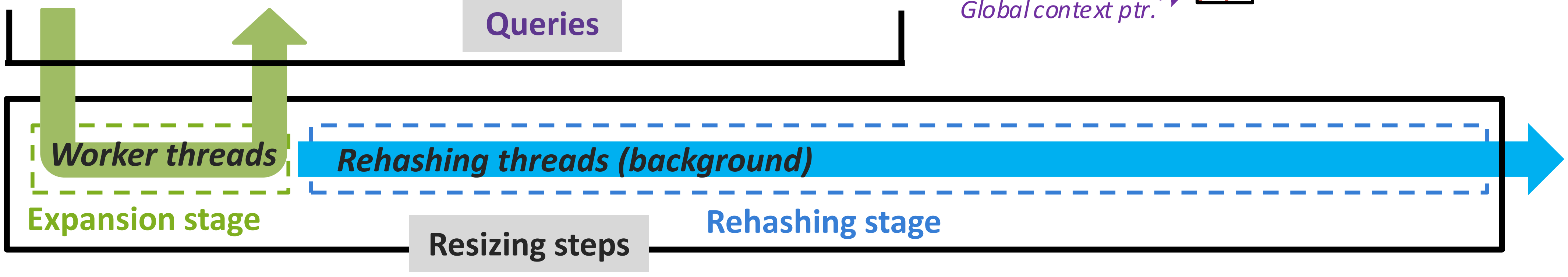
- Resizing steps

Expansion stage
1. Make a local copy of the global context pointer
2. CAS to append a new level
3. CoW + CAS to update the *first_level*

Rehashing stage
4. Rehash items in the last level
5. CoW + CAS to update the *last_level*

- Non-blocking resizing scheme
  - Rehashing threads: rehash until there are 2 levels left





Components

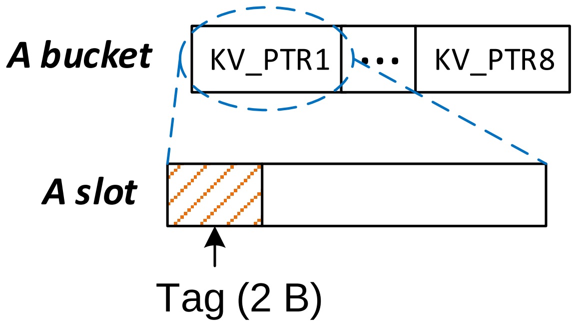
- Dynamic Multi-level Structure
- Non-blocking Resizing
- Lock-free Concurrency Control

Lock-free Search

- High latency for pointer dereference

Lock-free Search

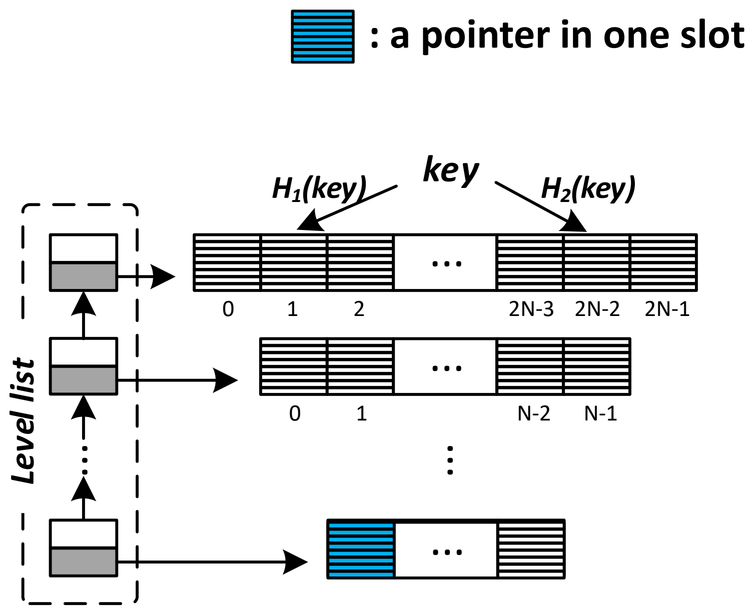
- High latency for pointer dereference
  - Summary tags
    - A tag is the summary for a key
    - Leverage the unused 16 highest bits of a pointer in x86_64 to store the tag



Update tag and pointer in an atomic manner

Lock-free Search

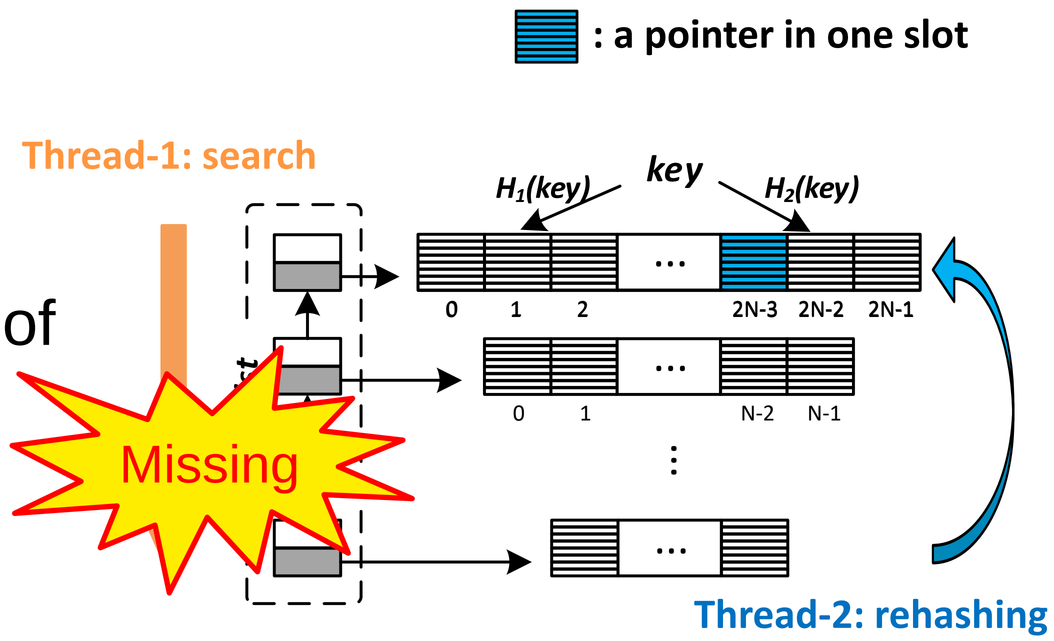
- High latency for pointer dereference
  - Summary tags
    - A tag is the summary for a key
    - Leverage the unused 16 highest bits of a pointer in x86_64 to store the tag



- Missing items due to rehashing

Lock-free Search

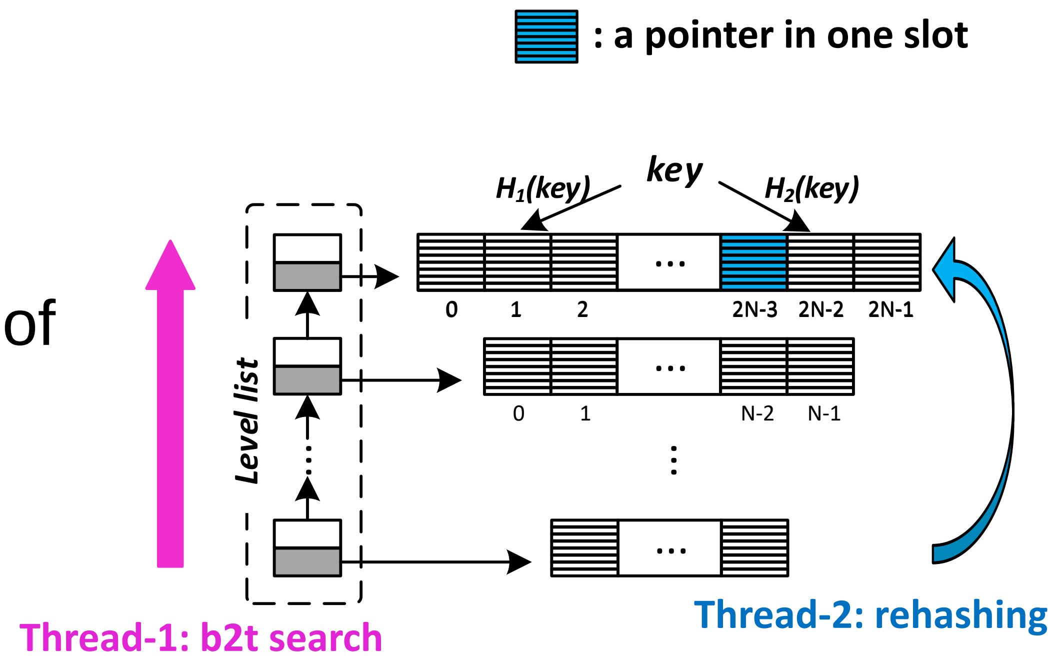
- ➢ High latency for pointer dereference
  - – Summary tags
    - A tag is the summary for a key
    - Leverage the unused 16 highest bits of a pointer in x86_64 to store the tag



- ➢ Missing items due to rehashing

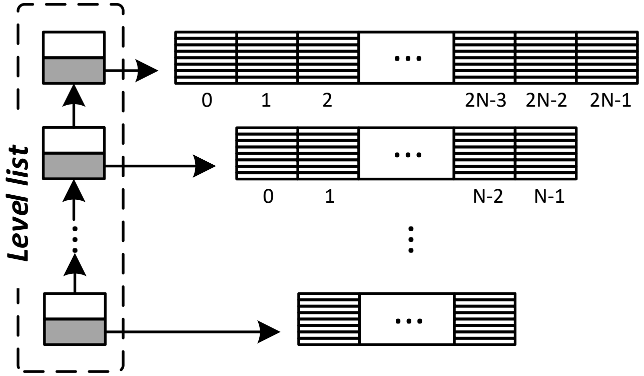
Lock-free Search

- ➢ High latency for pointer dereference
  - – Summary tags
    - • A tag is the summary for a key
    - • Leverage the unused 16 highest bits of a pointer in x86_64 to store the tag
- ➢ Missing items due to rehashing
  - – Bottom-to-top (b2t) search
    - • Search from the last level to the first level
    - • Redo the search when no item is found and the context changes



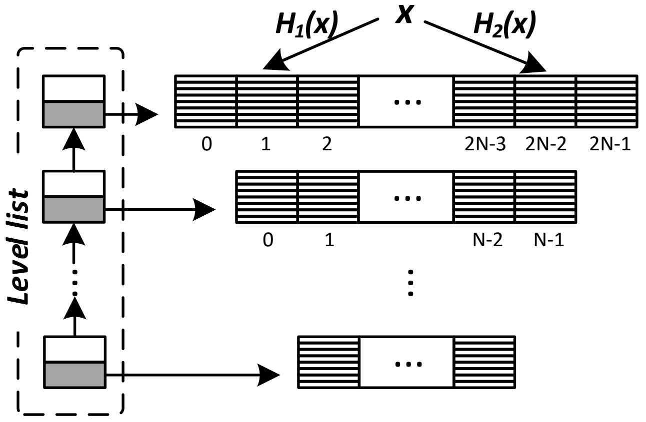
Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS



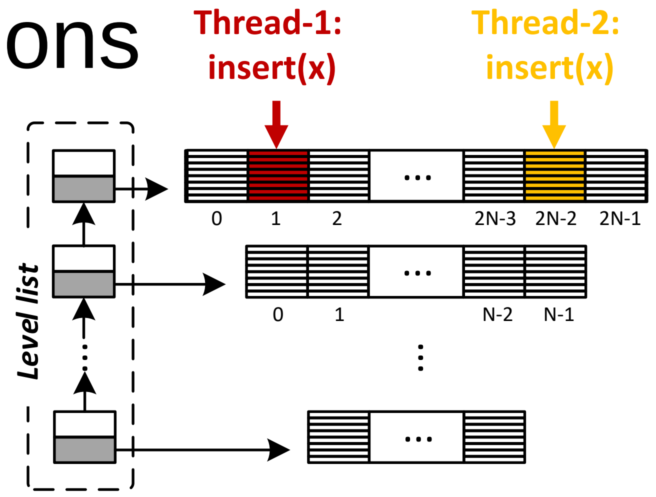
Lock-free Insertion

- ➢Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- ➢Duplicate items from concurrent insertions



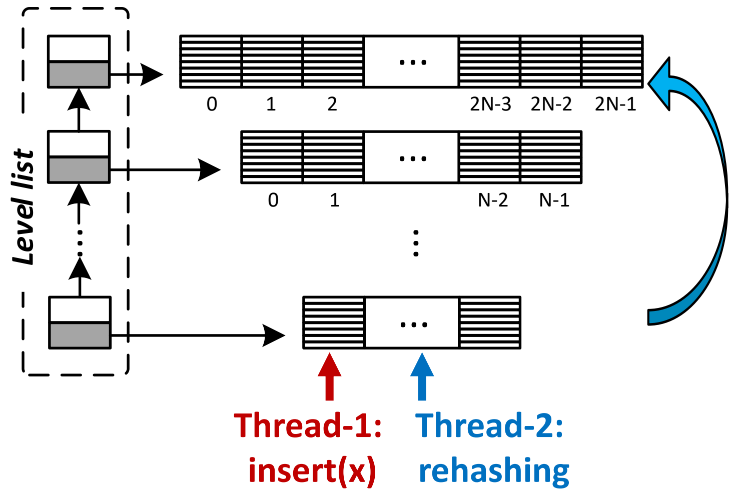
Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion



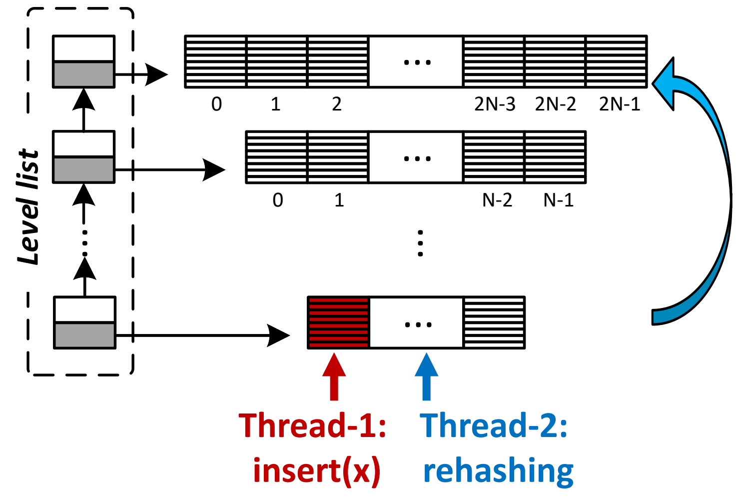
Lock-free Insertion

- ➢ Basic workflow
  - – Allocate the new item in PM
  - – B2t search to find duplicate keys
  - – Insert the pointer via CAS
- ➢ Duplicate items from concurrent insertions
  - – Both items are allowed for read
  - – Fix duplication in future update and deletion
- ➢ Loss of new items due to rehashing



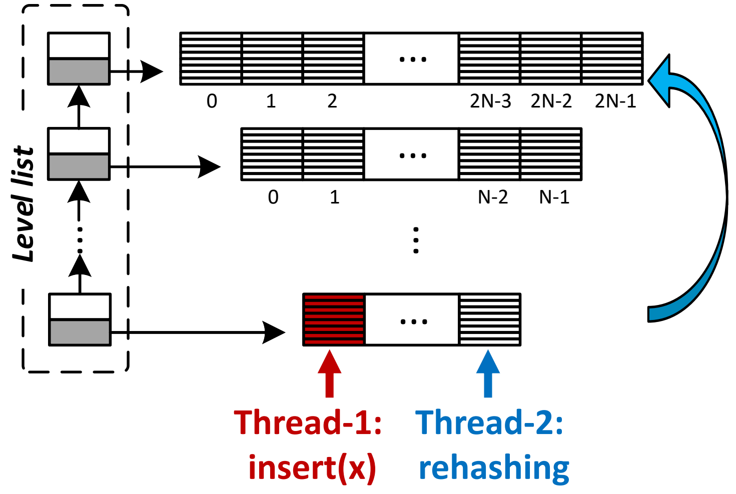
Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion
- Loss of new items due to rehashing



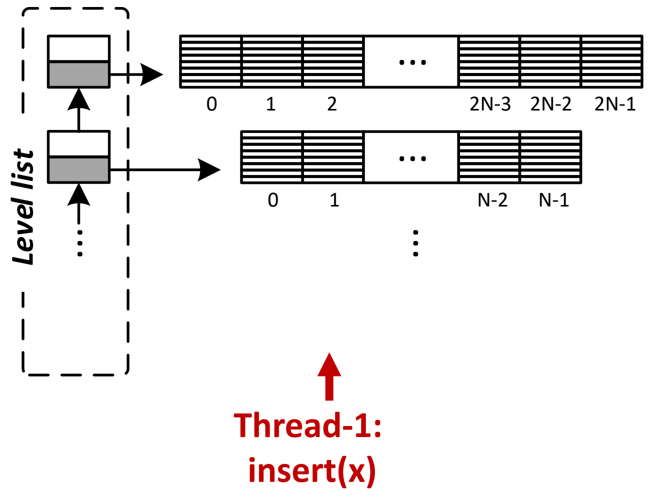
Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion
- Loss of new items due to rehashing



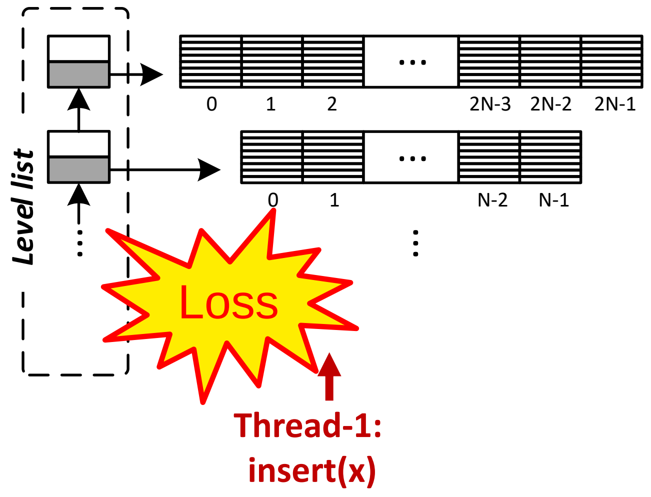
Lock-free Insertion

- ➢ Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- ➢ Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion
- ➢ Loss of new items due to rehashing



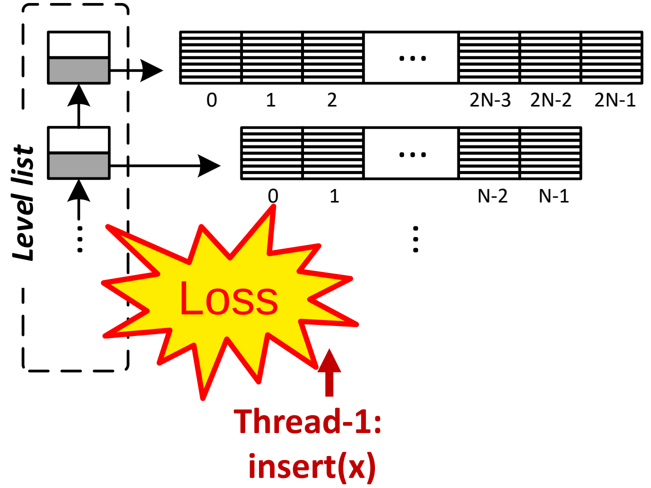
Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion
- Loss of new items due to rehashing



Lock-free Insertion

- Basic workflow
  - Allocate the new item in PM
  - B2t search to find duplicate keys
  - Insert the pointer via CAS
- Duplicate items from concurrent insertions
  - Both items are allowed for read
  - Fix duplication in future update and deletion
- Loss of new items due to rehashing
  - Context-aware insertion
    - Not inserted to the rehashed last level
    - Redo insertion for possible loss

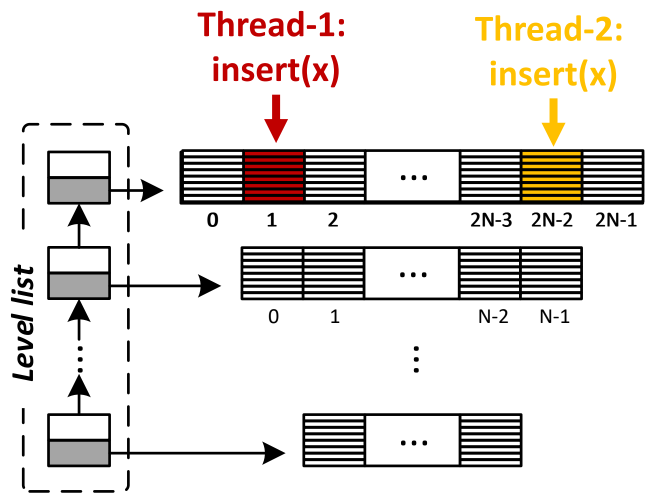


Lock-free Update

- Inconsistency for duplicate items

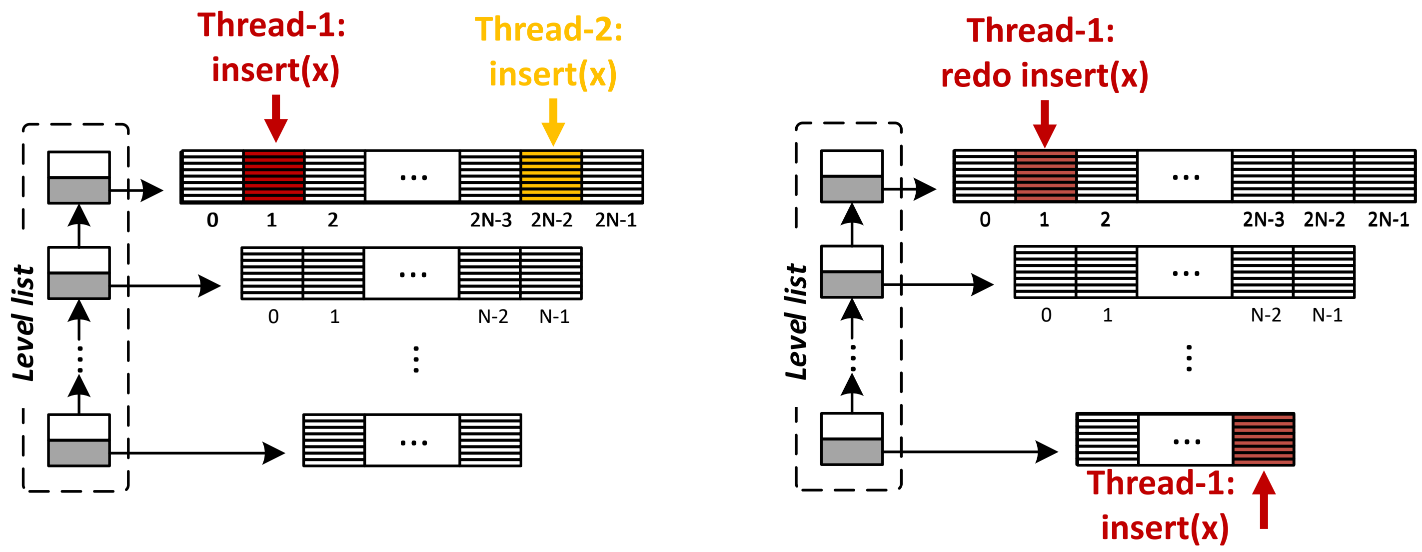
Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key



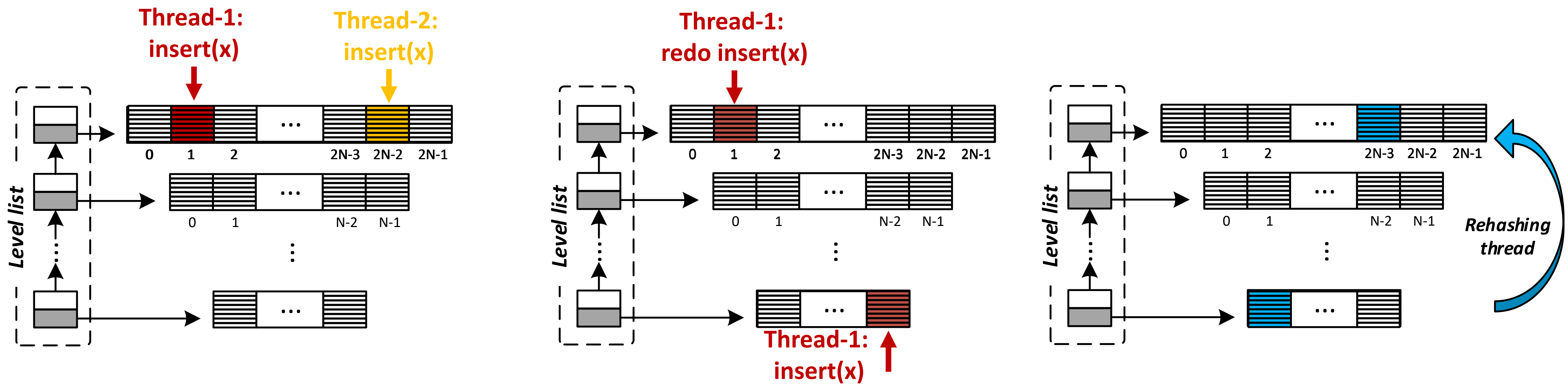
Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion



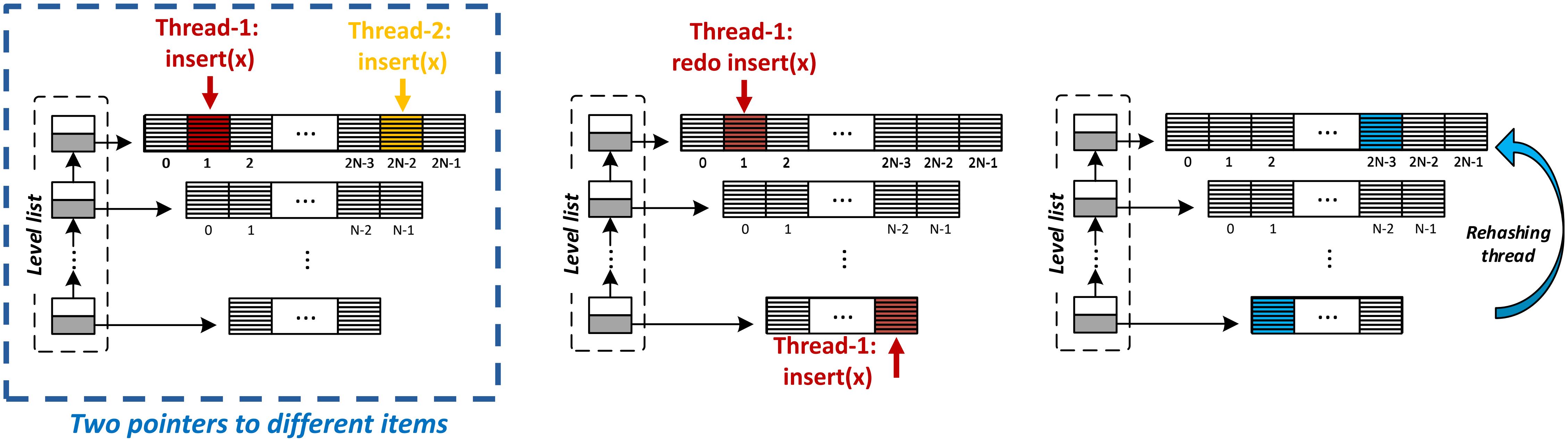
Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing



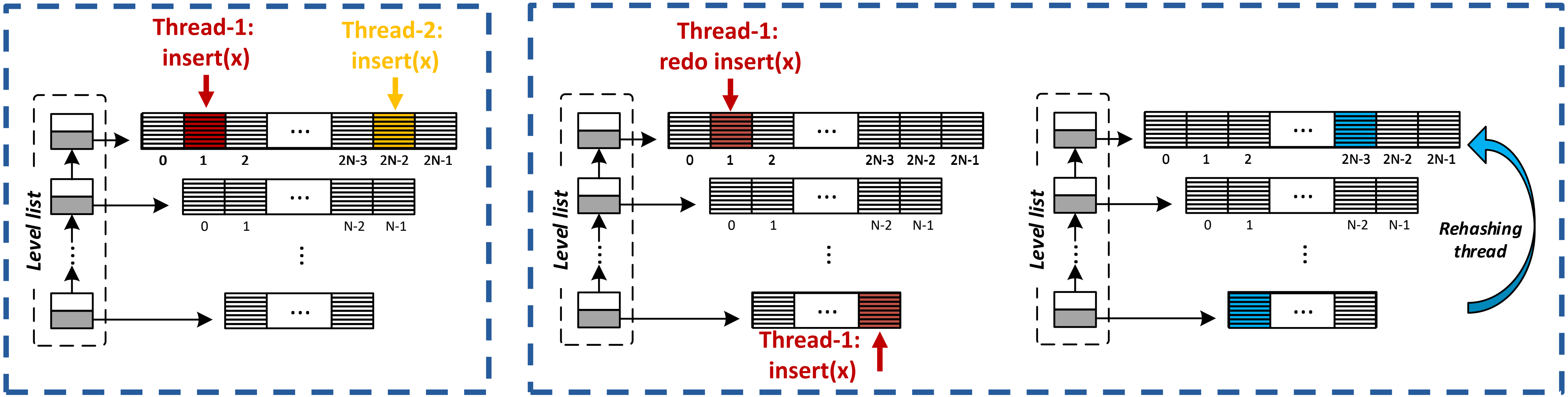
Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing



Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing

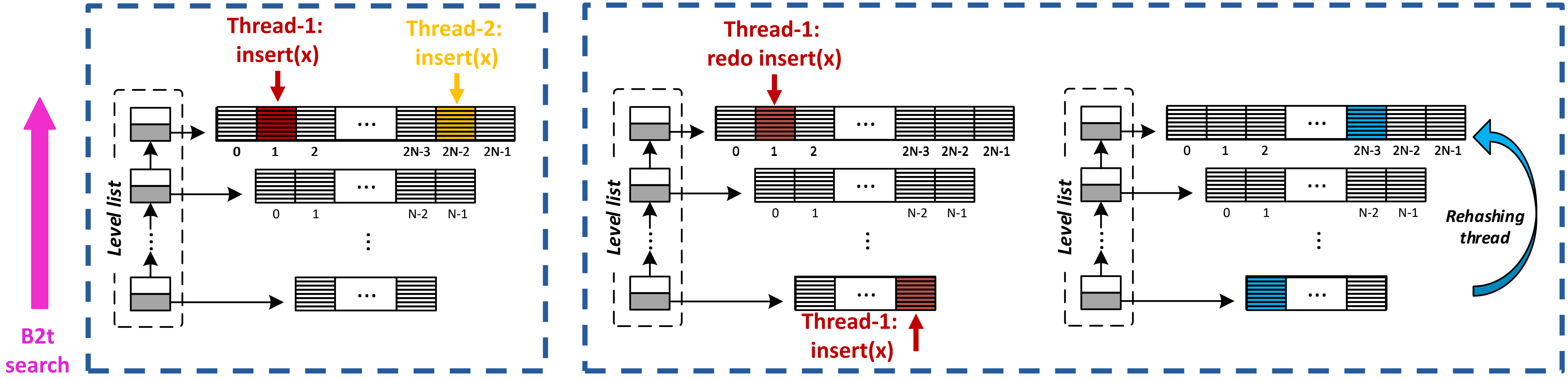


Two pointers to different items

Two pointers to the same item

Lock-free Update

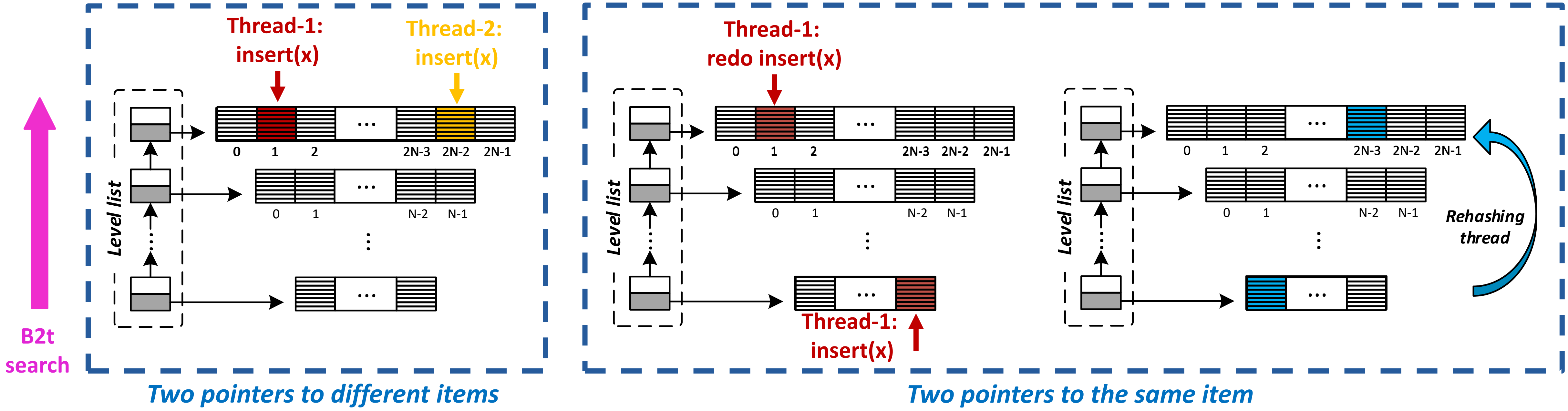
- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing
- Content-conscious Find
  - B2t search to find two pointers to duplicate items



Two pointers to different items　　*Two pointers to the same item*

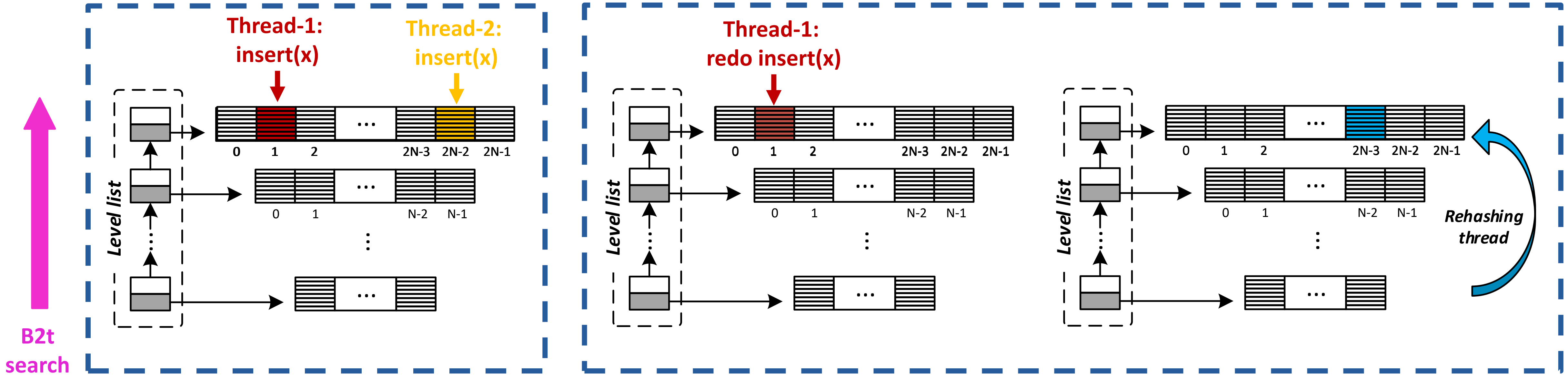
Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing
- Content-conscious Find
  - B2t search to find two pointers to duplicate items
  - Check if two pointers refer to the same item



Lock-free Update

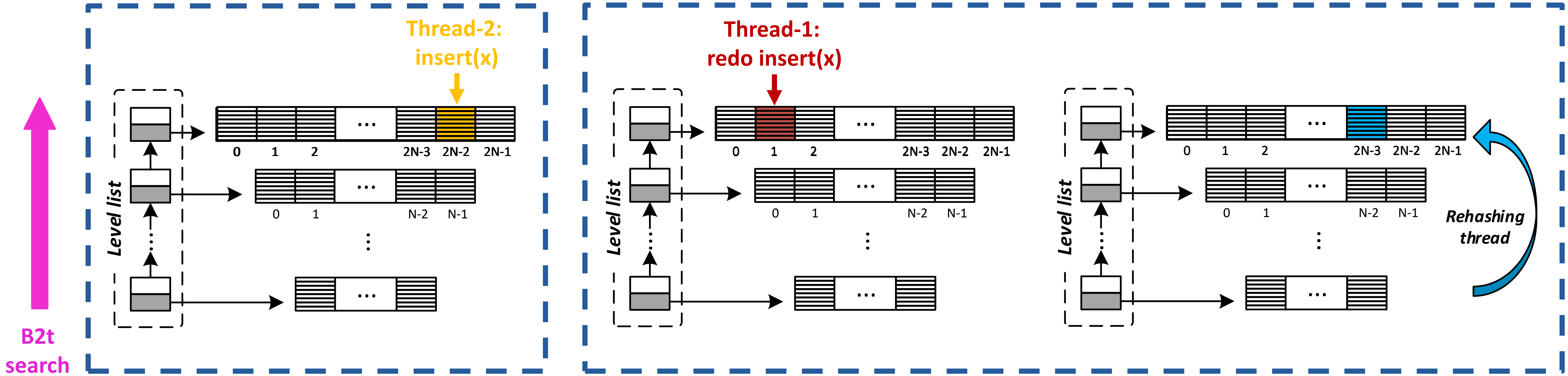
- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing
- Content-conscious Find
  - B2t search to find two pointers to duplicate items
  - Check if two pointers refer to the same item
  - **Yes**: delete the first pointer matching the key



Two pointers to different items *Two pointers to the same item*

Lock-free Update

- Inconsistency for duplicate items
  - Concurrent insertions with the same key
  - Retry of context-aware insertion
  - Data movement for rehashing
- Content-conscious Find
  - B2t search to find two pointers to duplicate items
  - Check if two pointers refer to the same item
    - **Yes**: delete the first pointer matching the key
    - **No**: delete the first pointer and corresponding item matching the key

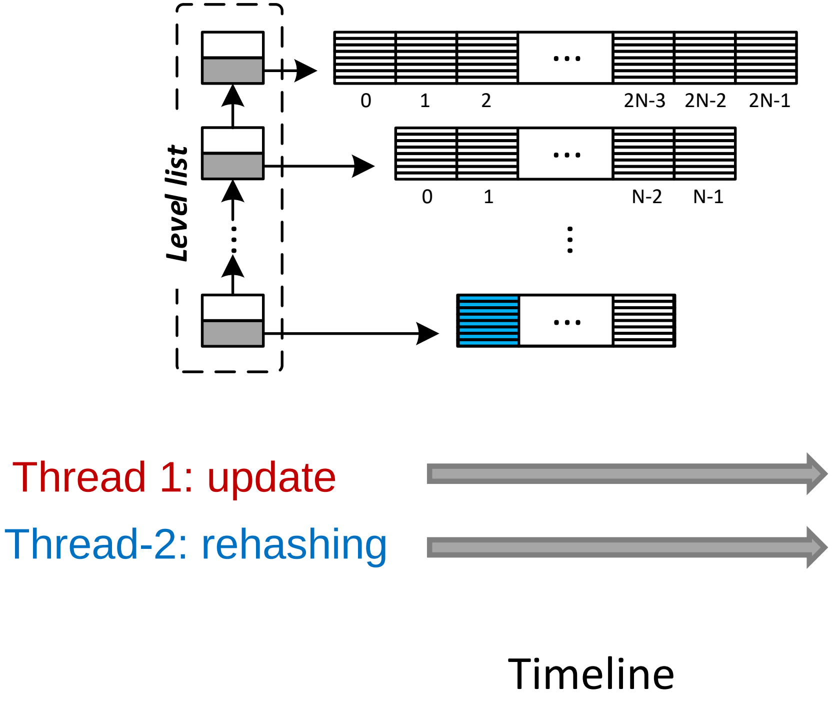


Two pointers to different items

Two pointers to the same item

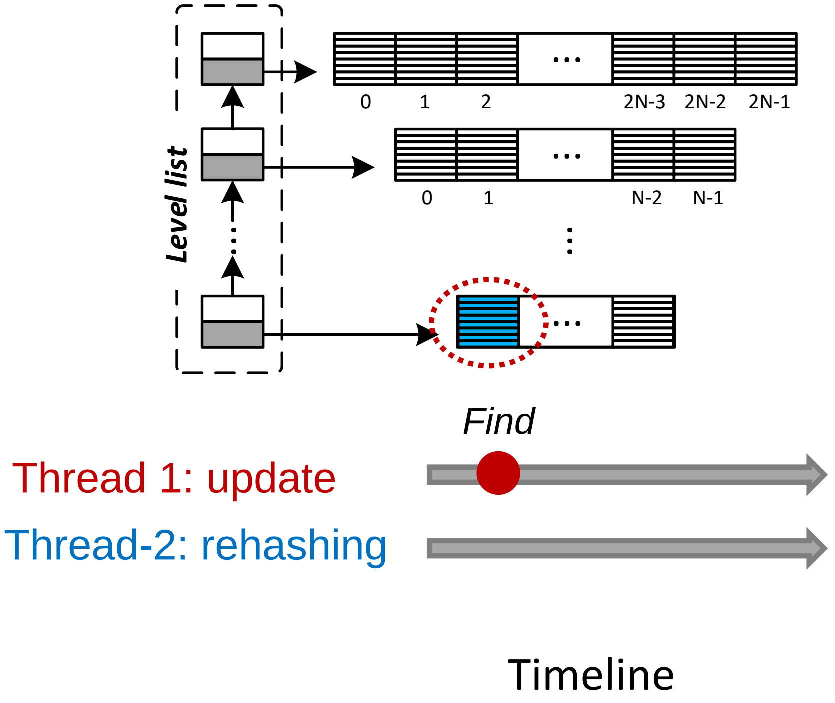
Failures of Lock-free Update

➢ Update failures due to interleaved update and rehashing



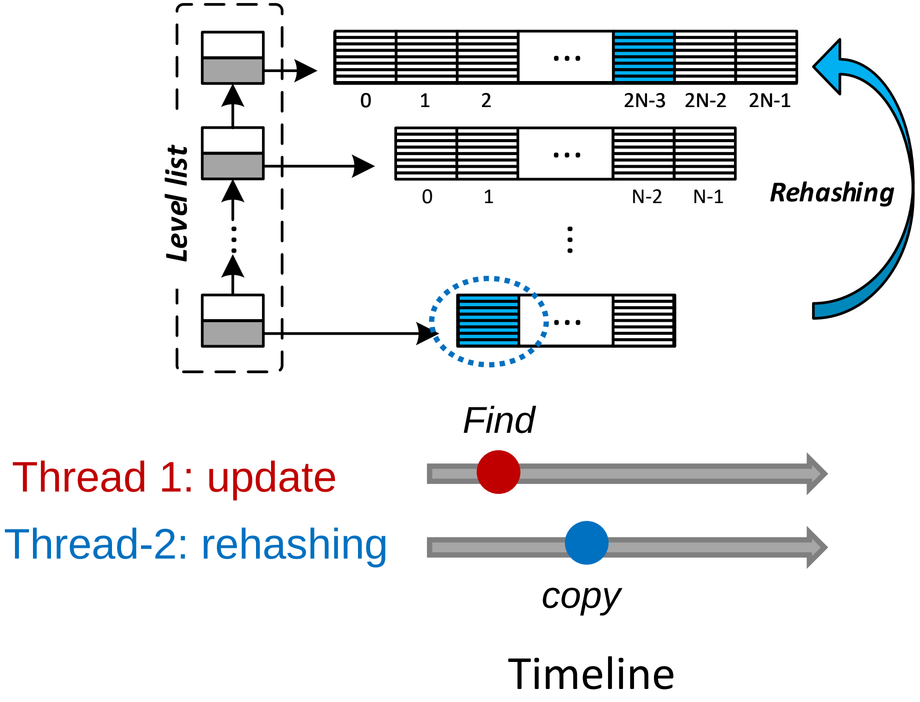
Failures of Lock-free Update

➢Update failures due to interleaved update and rehashing



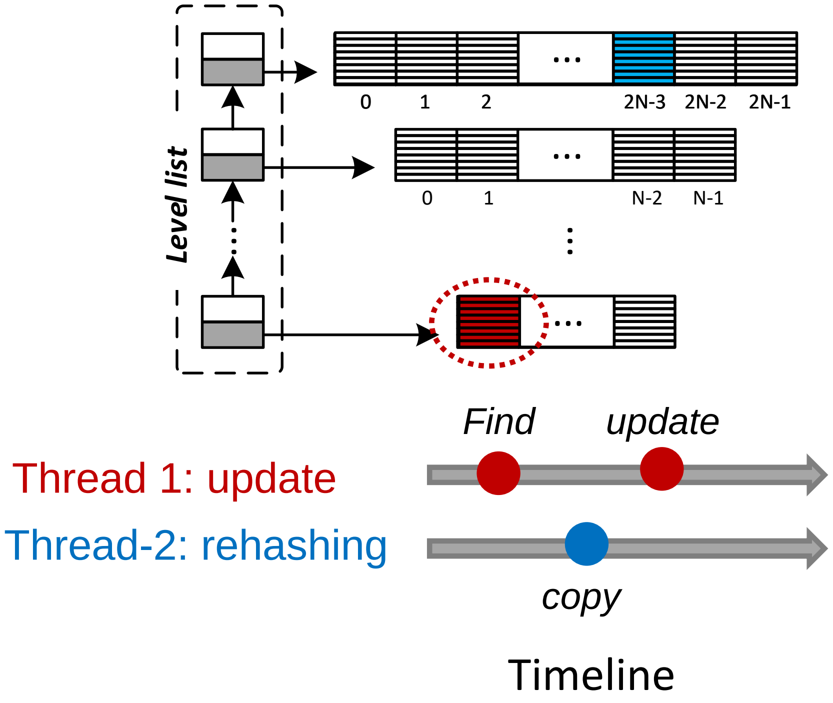
Failures of Lock-free Update

- Update failures due to interleaved update and rehashing



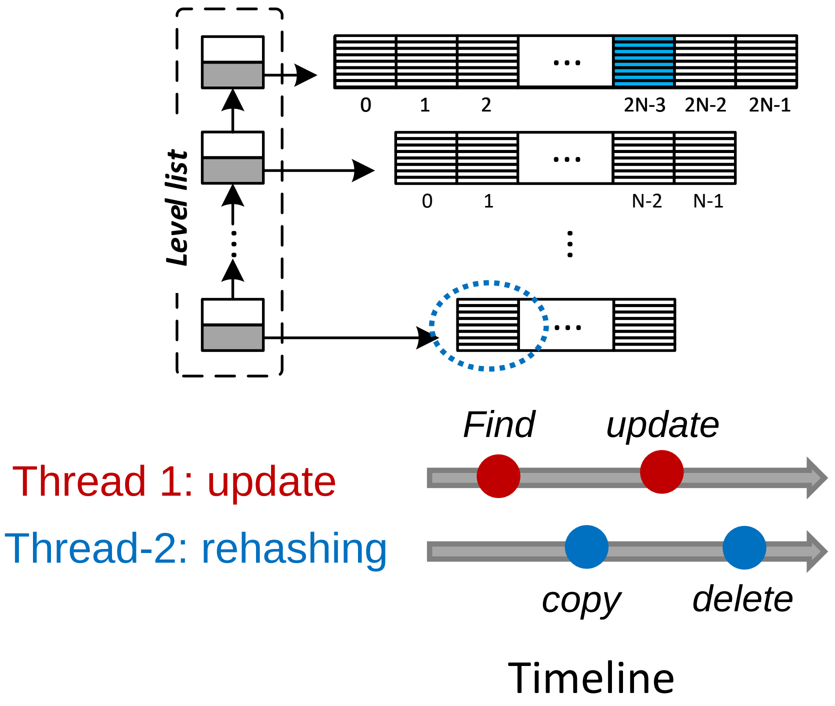
Failures of Lock-free Update

- ➢Update failures due to interleaved update and rehashing



Failures of Lock-free Update

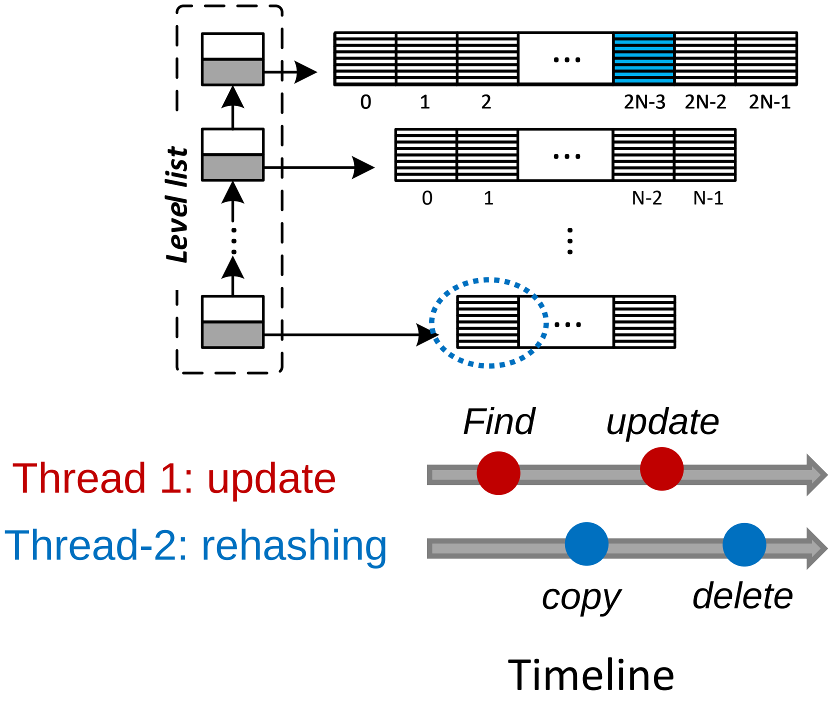
- Update failures due to interleaved update and rehashing



Failures of Lock-free Update

- Update failures due to interleaved update and rehashing

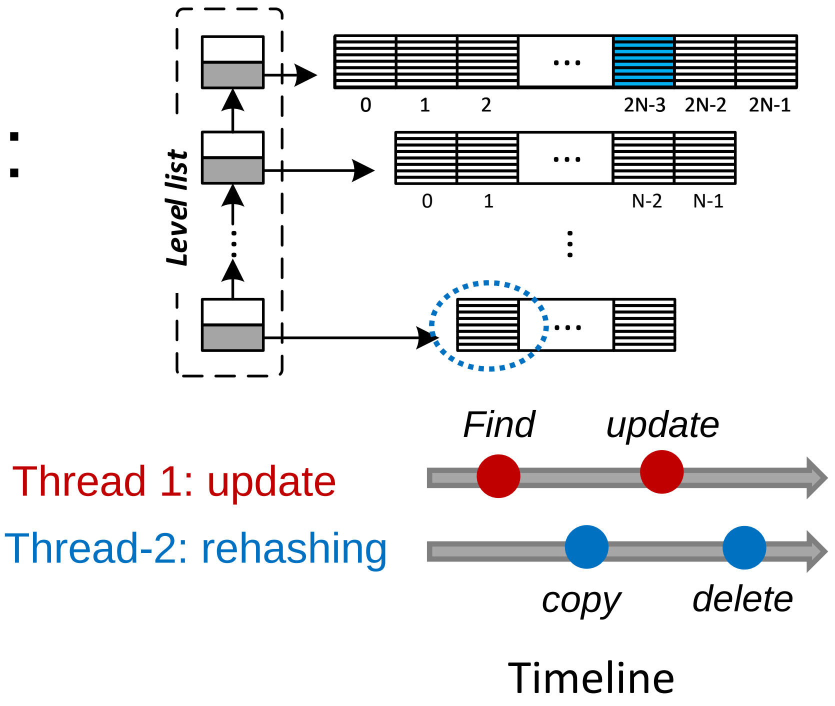
- **Baseline**: two-round Find for update



Failures of Lock-free Update

- ➢ Update failures due to interleaved update and rehashing

- ➢ **Baseline**: two-round Find for update
- ➢ Optimization: redo Find only when simultaneously satisfying:
  - – Table is resizing
  - – The updated bucket is in the last level
  - – The bucket index is in one of the processed regions of rehashing threads



Lock-free Deletion

- Delete matched pointers atomically via CAS

- Inconsistency due to duplicate items
  - Instead of Find, delete all matched items in b2t search

- Deletion failures due to interleaved deletion and rehashing
  - Similar optimizations to avoid frequent re-execution of deletion

Crash Recovery

- Crash consistency for lock-free Clevel hashing
  - Persist after PM writes
  - Persist dependent metadata after loading them

Atomic visibility* enables *low-overhead crash consistency

- Recovery
  - Rehashing resumes from the last processed bucket

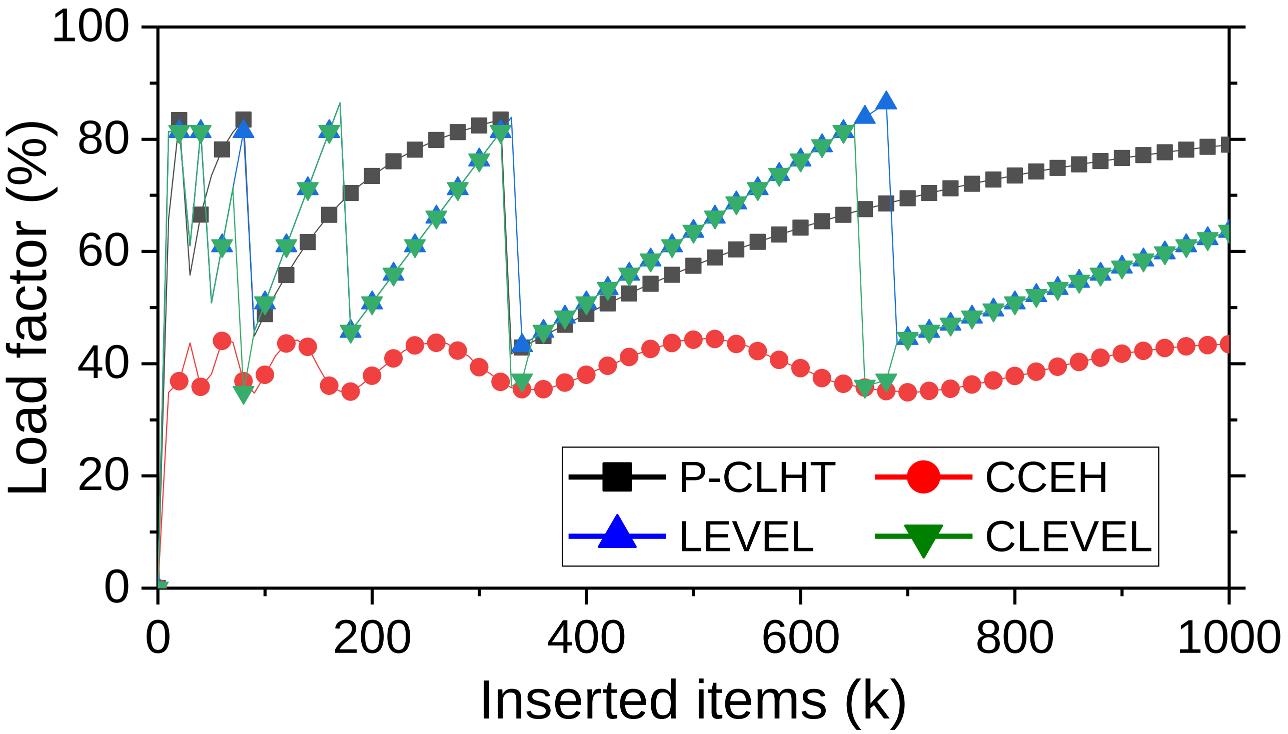
Experimental Setup

- Platform
  - Intel Optane DC PMM configured in *App Direct* mode
  - 36 threads in one NUMA node
  - PMDK

- Comparisons
  - **LEVEL**: original level hashing [OSDI ’18]
  - **CCEH**: lazy deletion version, default probing distance (16 slots) [FAST ’19]
  - **CMAP**: concurrent_hash_map engine from Intel pmemkv
  - **P-CLHT**: PM variant of CLHT converted by RECIPE [SOSP ’19]
  - **CLEVEL**: our Clevel hashing

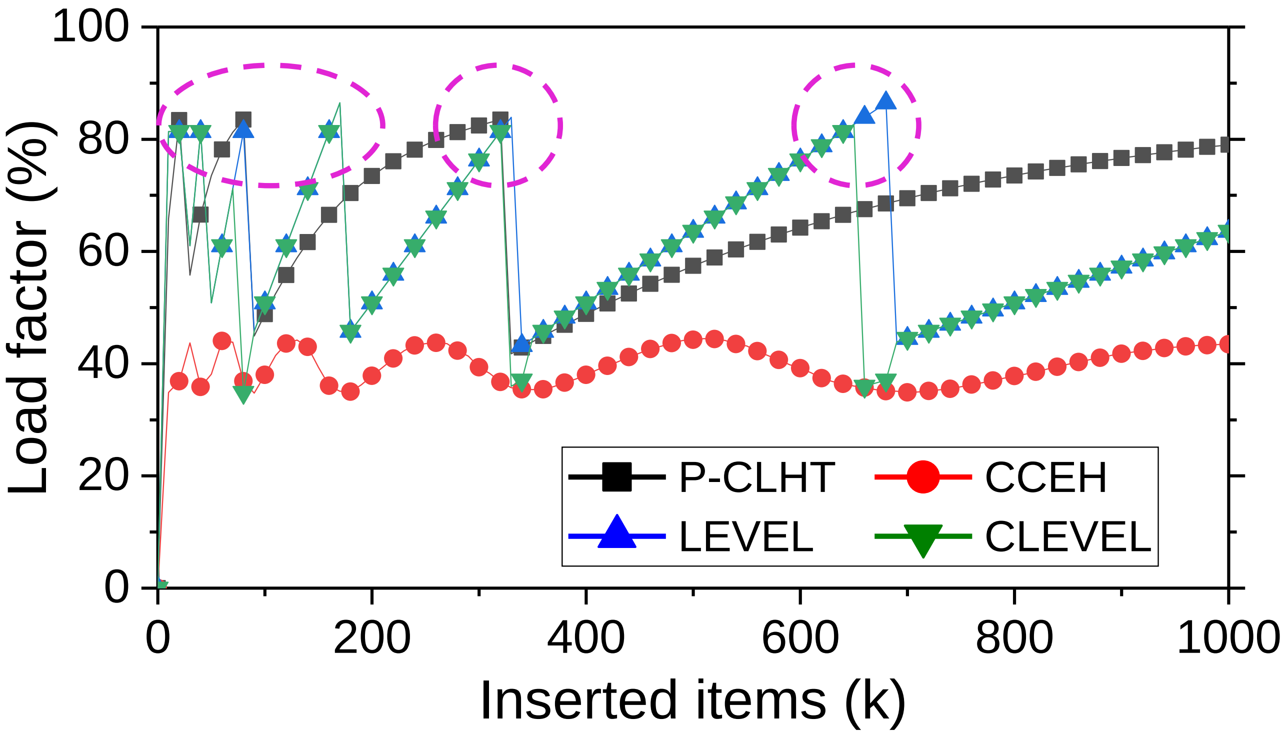
- Benchmark: YCSB

Load Factor



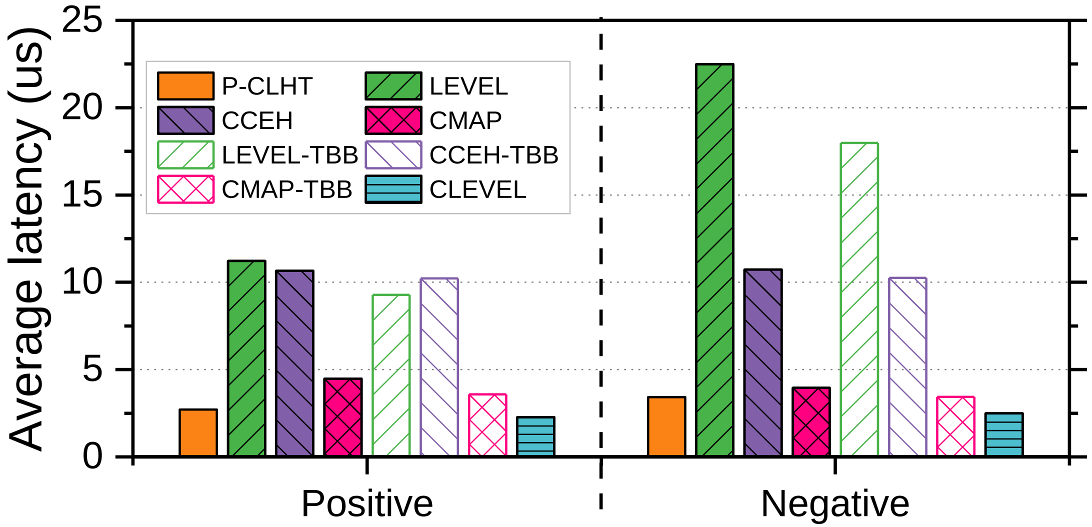
Load factor (%)
Inserted items (k)
P-CLHT
CCEH
LEVEL
CLEVEL
0
20
40
60
80
100
0
200
400
600
800
1000

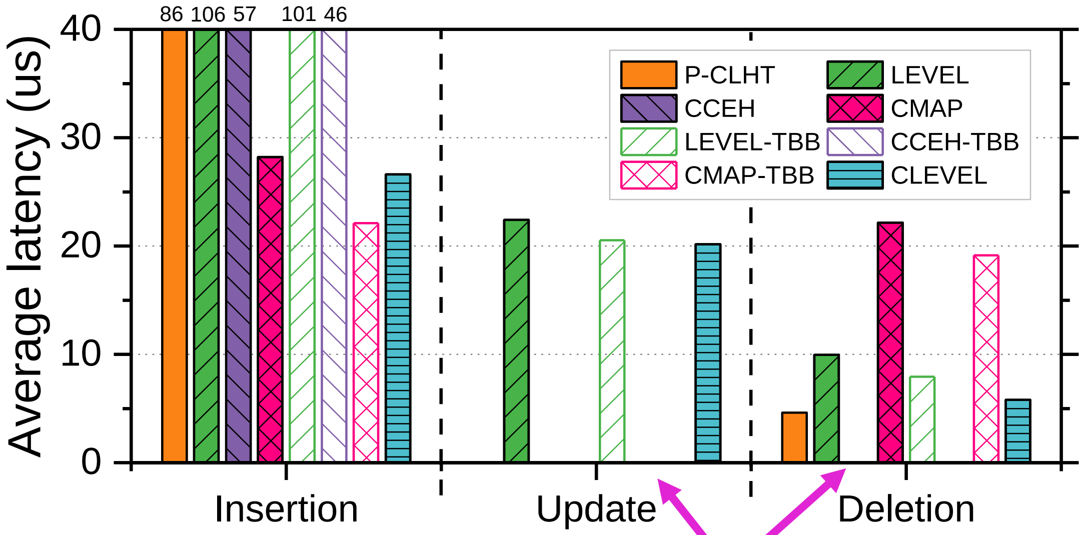
Load Factor



- Clevel hashing has comparable load factor with level hashing, i.e., 86%

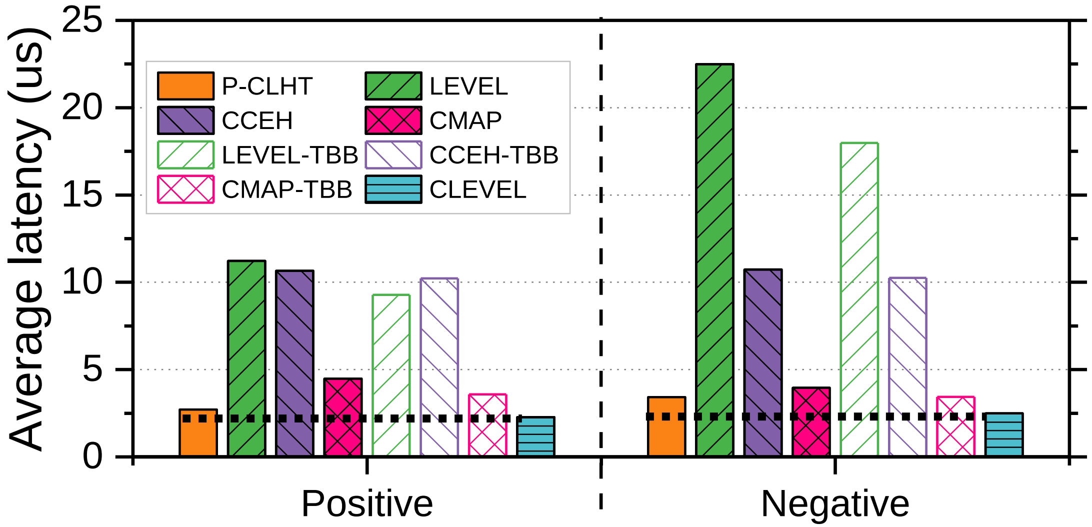
Micro-benchmarks

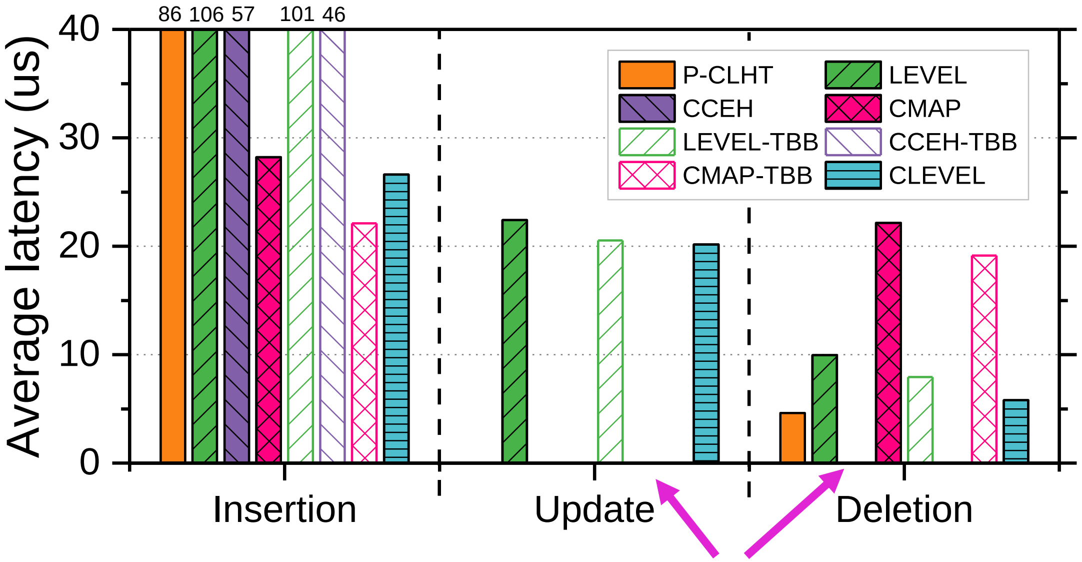




** Lack of implementation of update and deletion in open-source code*

Micro-benchmarks

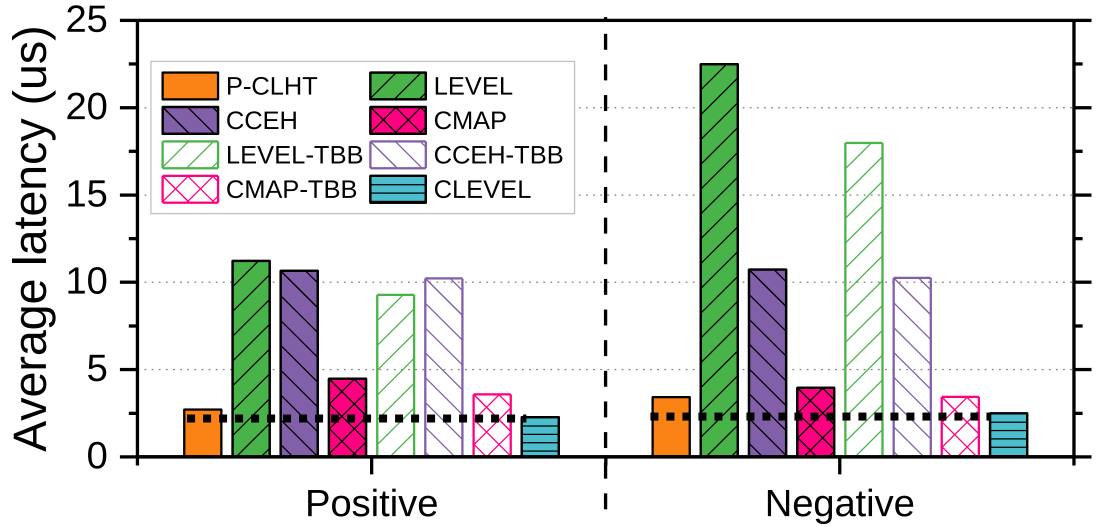


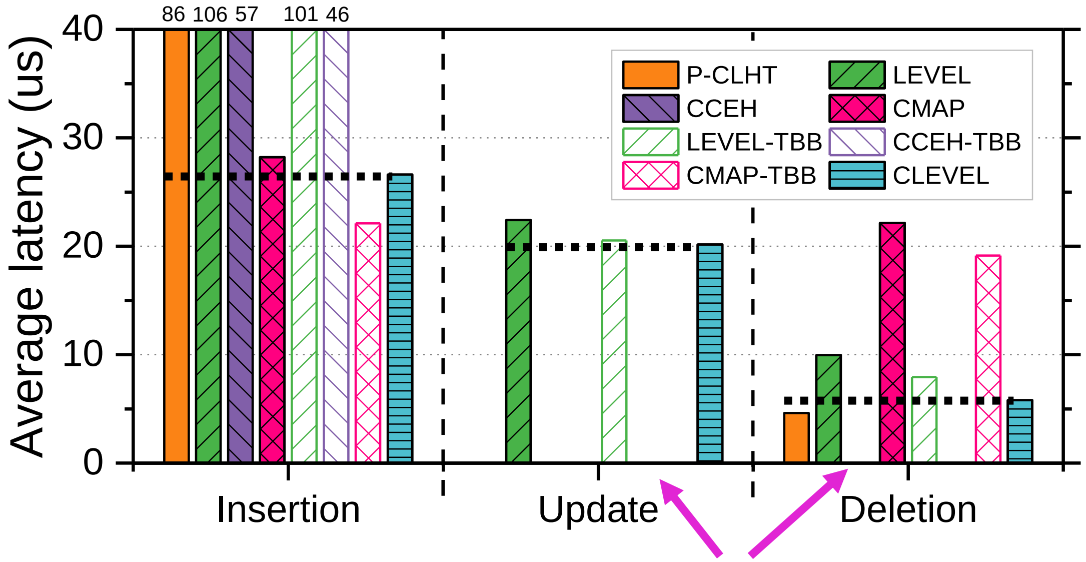


** Lack of implementation of update and deletion in open-source code*

- ➢ Due to using lock-free search and summary tags, Clevel hashing obtains
  - 1.2×−5.0× speedup for positive search
  - 1.4×−9.0× speedup for negative search

Micro-benchmarks



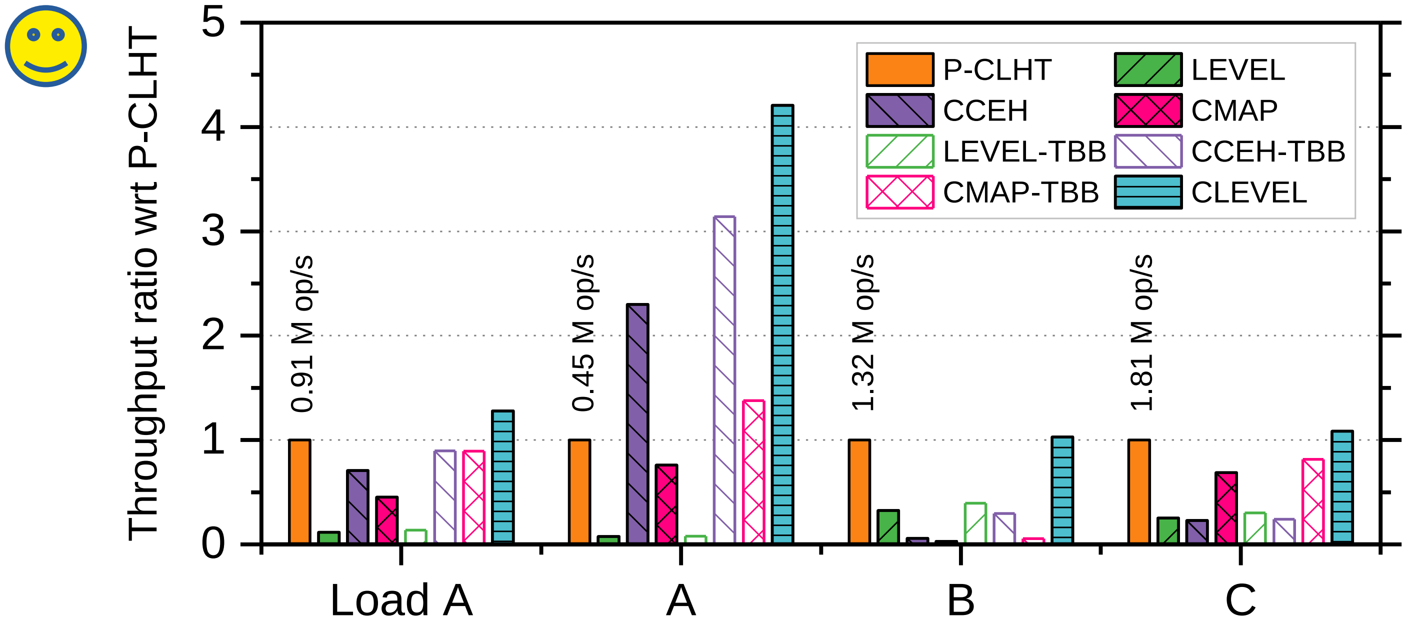


** Lack of implementation of update and deletion in open-source code*

- Due to using lock-free search and summary tags, Clevel hashing obtains
  - 1.2×−5.0× speedup for positive search
  - 1.4×−9.0× speedup for negative search

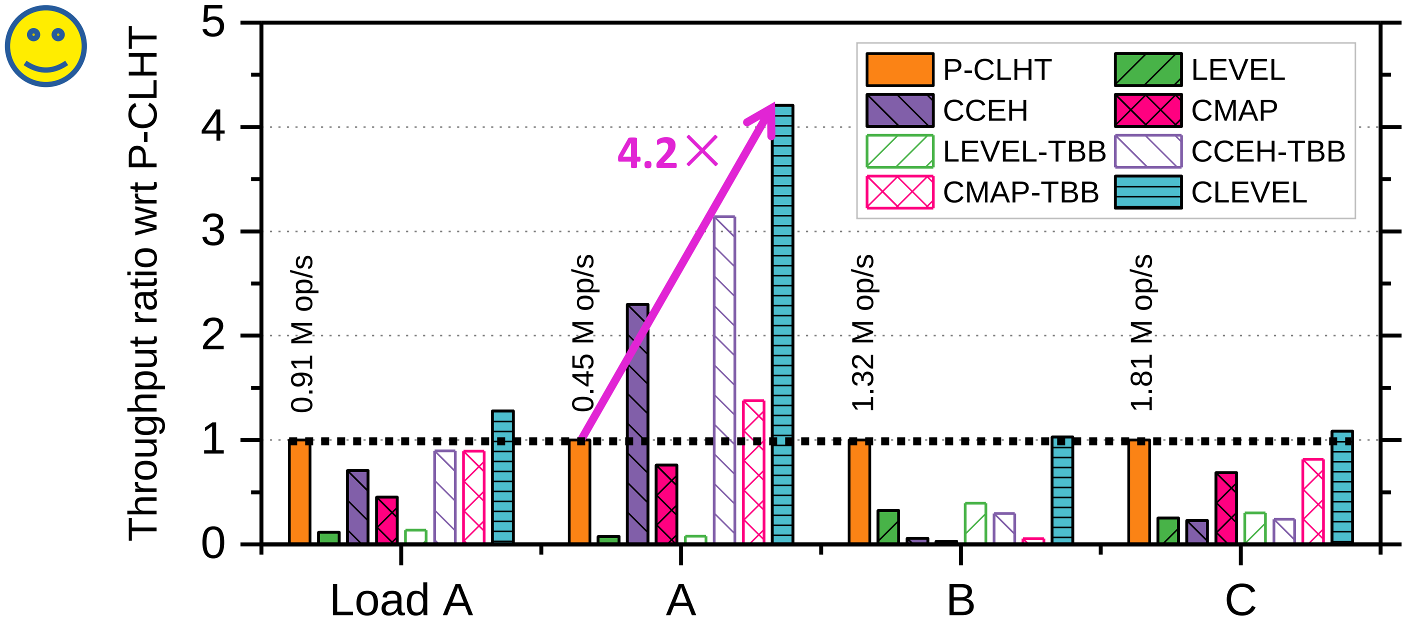
- Clevel hashing achieves low latency with correctness guarantee

Macro-benchmarks



Read ratio (%):	0	50	95	100
Write ratio (%):	100	50	5	0

Macro-benchmarks



	Load A	A	B	C
Read ratio (%):	0	50	95	100
Write ratio (%):	100	50	5	0

➢ Clevel hashing obtains up to 4.2× speedup than P-CLHT due to the lock-free concurrency control and non-blocking resizing

Conclusion

- Existing PM hashing indexes have limited considerations for concurrency
- Clevel hashing is PM-friendly
  - Write-optimal multi-level structure without extra writes for insertion
  - Crash consistency by enabling lock-free index to be persistent
- Clevel hashing achieves high concurrency
  - Non-blocking resizing without blocking queries
  - Lock-free concurrency control with correctness guarantee
- Clevel hashing achieves up to 4.2× speedup for throughput than P-CLHT

Thanks! Q&A

Email: chenzy@hust.edu.cn

Homepage: https://chenzhangyu.github.io

Open-source code: *https://github.com/chenzhangyu/Clevel-Hashing*