Προχωρημένη Κατανεμημένη Υπολογιστική

ΗΥ623

Διδάσκων –
Δημήτριος Κατσαρός

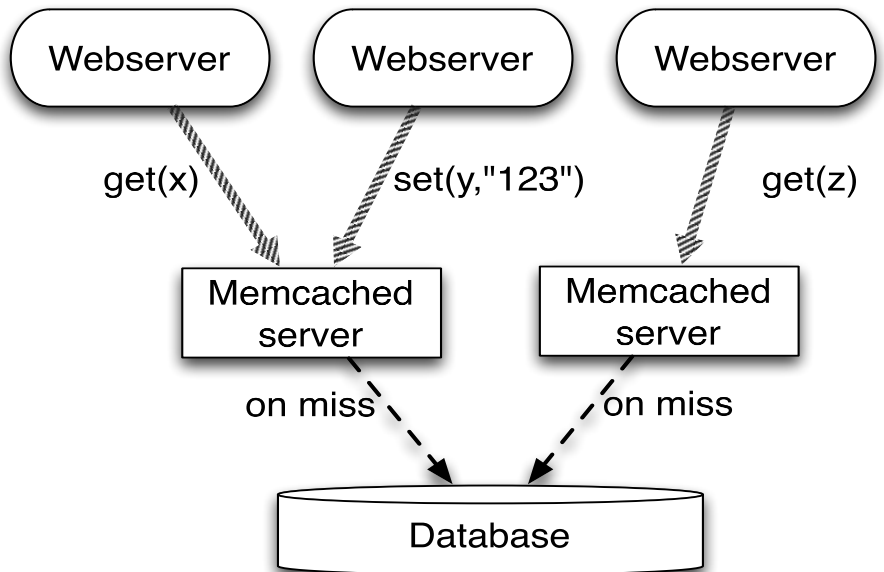
@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Background

Memcached Overview

- A DRAM-based key-value store
  - `GET(key)`
  - `SET(key, value)`

- LRU eviction for high hit rate
- Typical use:
  - Speed up webservers
  - Alleviate db load



Background

Typical Workloads

- Often used for small objects (Facebook[Atikoglu12])
  - 90% keys < 31 bytes
  - Some apps only use 2-byte values

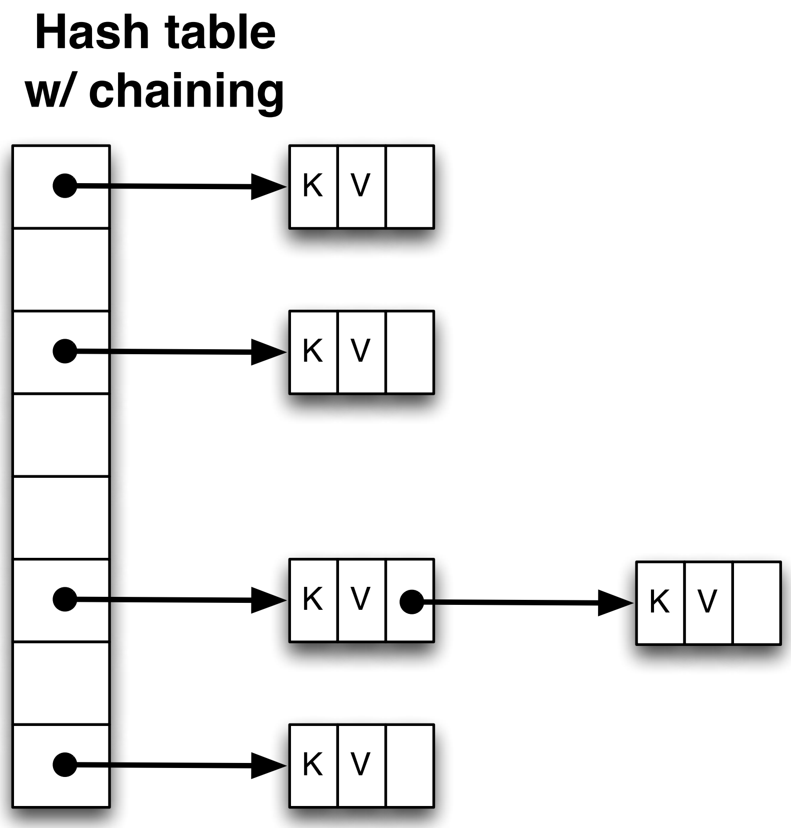
- Tens of millions of queries per second for large memcached clusters (Facebook[Nishtala13])

Small Objects, High Rate

Background

Memcached: Core Data Structures

- **Key-Value Index**:
  - Chaining hash table



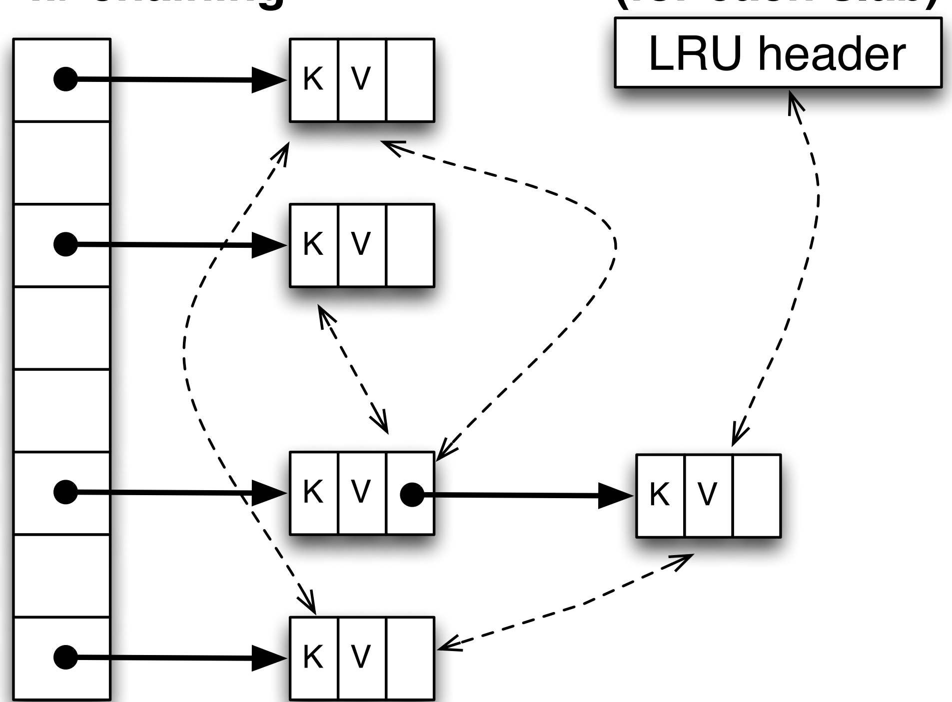
Background

Memcached: Core Data Structures

- **Key-Value Index**:
  - Chaining hash table
- **LRU Eviction**:
  - Doubly-linked lists

Hash table w/ chaining

Doubly-linked-list (for each slab)



Problems We Solve

- Single-node scalability
  - Accessing hash table and updating LRU are serialized
- Space overhead
  - 56-byte header per object
    - Including 3 pointers and 1 refcount
    - For a 100B object, overhead > 50%

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

CLOCK-based LRU eviction

- Better space efficiency and concurrency

Additional algo & tuning improvements

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

CLOCK-based LRU eviction

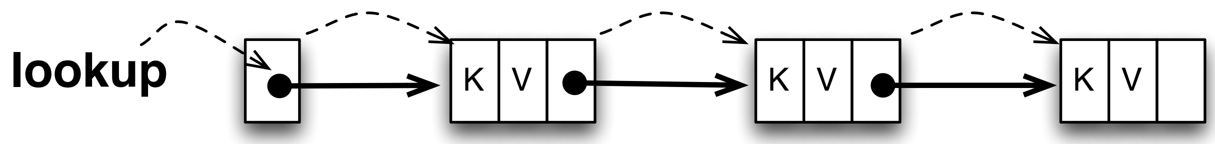
- Better space efficiency and concurrency

Additional algo & tuning improvements

Problem

Memcached Default Hash Table

- Chaining items hashed in same bucket:



Good: simple (Data Structure 101)

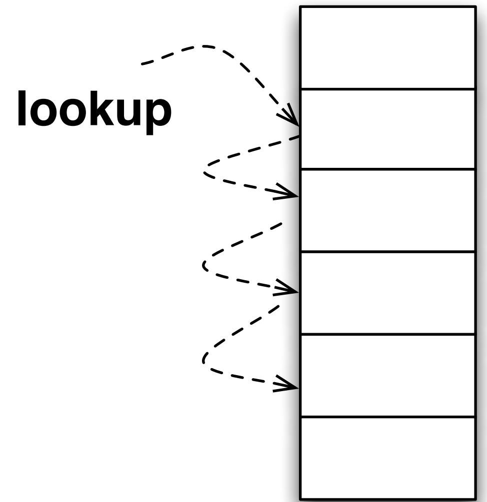
Bad: low cache locality:
(dependent pointer dereference)

Bad: pointer costs space

Problem

Linear Probing

- Probing consecutive buckets for vacancy



Good: simple

Good: cache friendly

Bad: poor memory efficiency:

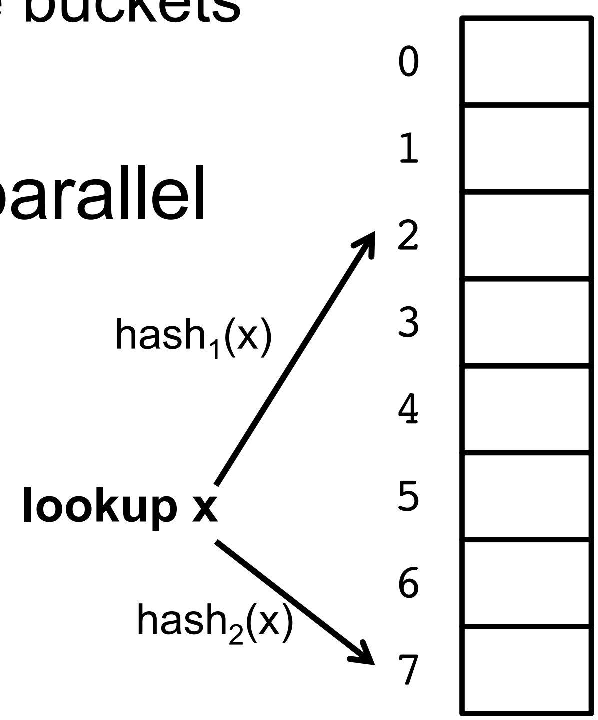
(if occupancy > 50%, lookup needs to search a long chain)

Solution

Cuckoo Hashing[Pagh04]

- Each key has two candidate buckets
  - Assigned by $hash_1$(key), $hash_2$(key)
  - Stored in one of its candidate buckets

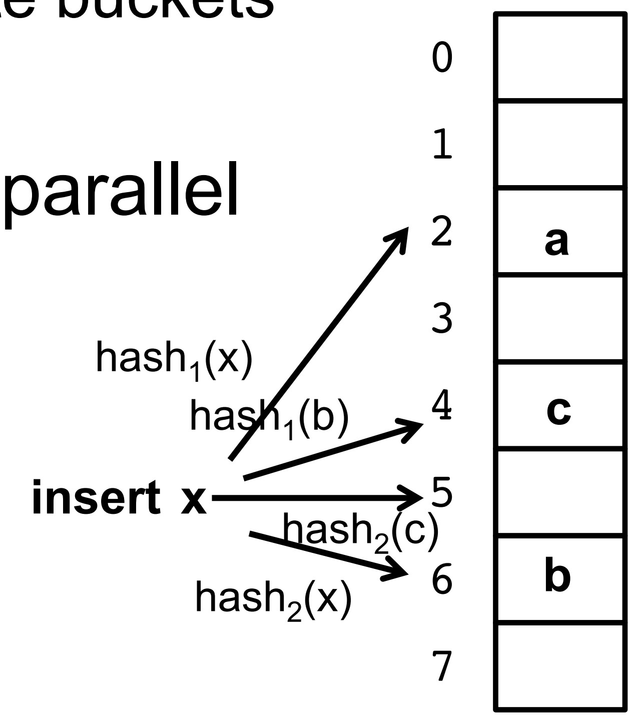
- Lookup: read 2 buckets in parallel



Solution

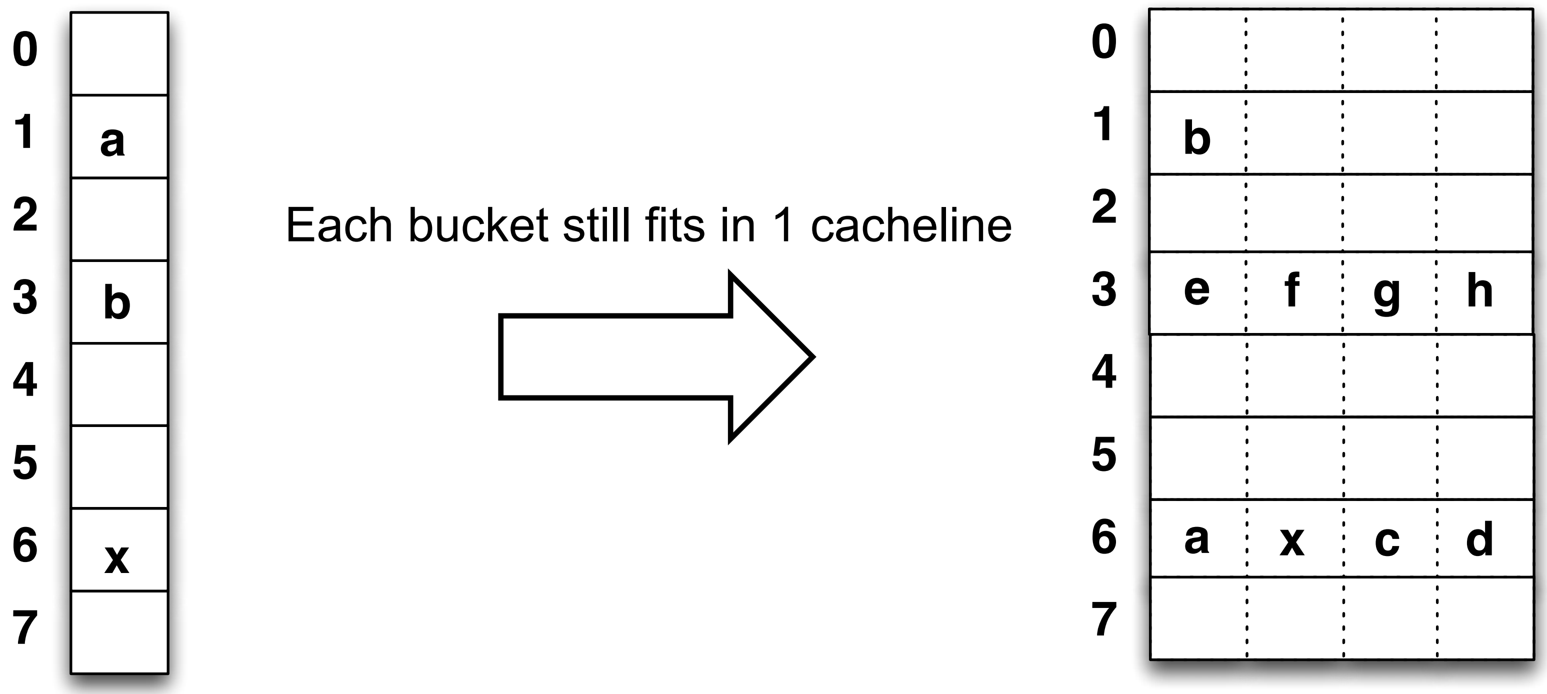
Cuckoo Hashing[Pagh04]

- Each key has two candidate buckets
  - Assigned by $hash_1$(key), $hash_2$(key)
  - Stored in one of its candidate buckets
- Lookup: read 2 buckets in parallel
- Insert:
  - Perform key displacement recursively
  - Still O(1) on average [Pagh04]



Solution

Increase Set-Associativity



- 2 cacheline-sized reads per lookup
- **50%** space utilized

- 2 cacheline-sized reads per lookup
- **95%** space utilized!

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

CLOCK-based LRU eviction

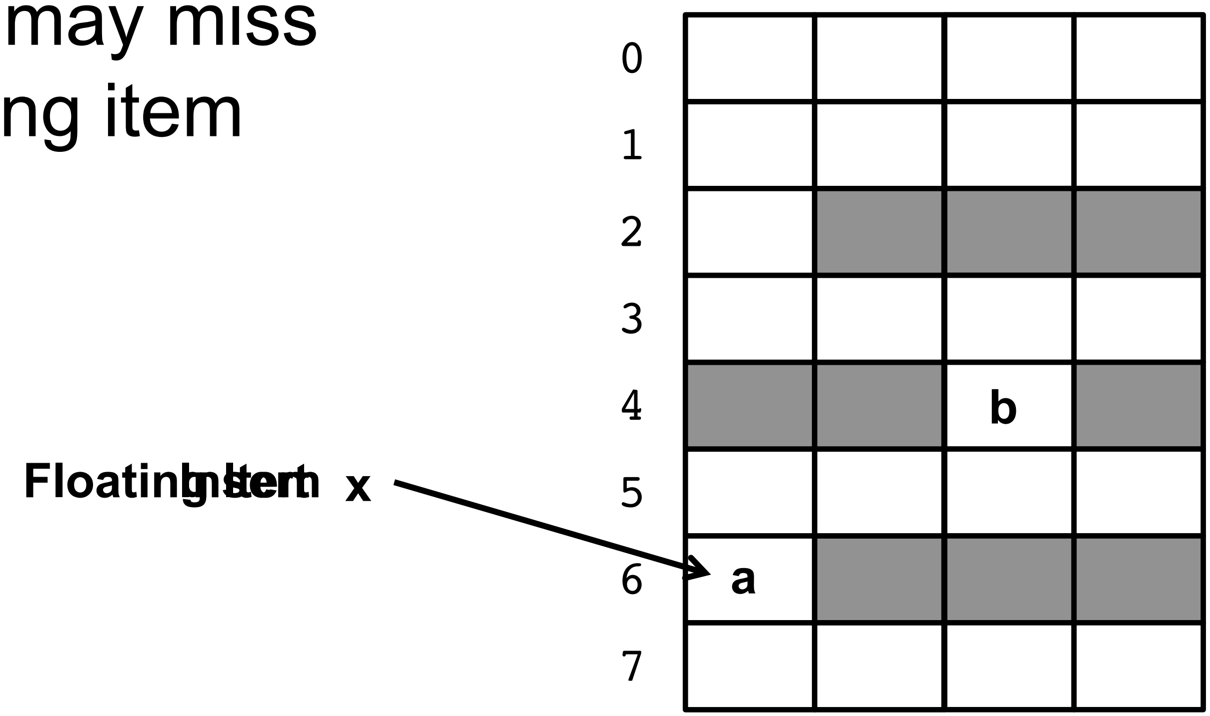
- Better space efficiency and concurrency

Additional algo & tuning improvements

Problem

False Miss Problem

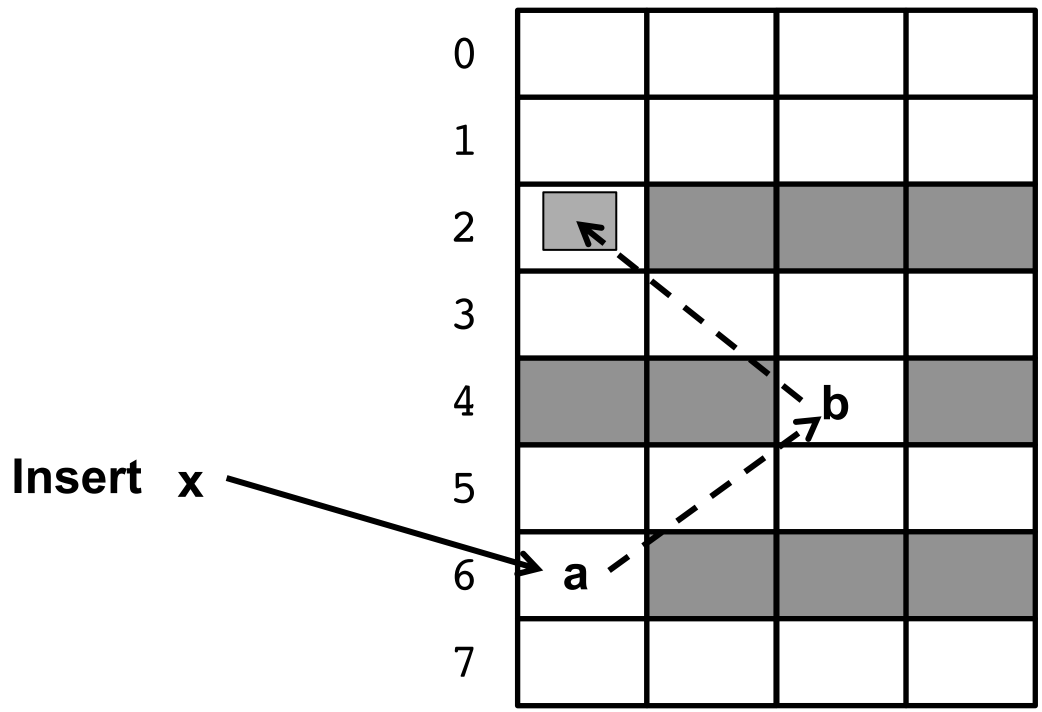
- During insertion:
  - always a “floating” item during insertion
  - a reader may miss this floating item



Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

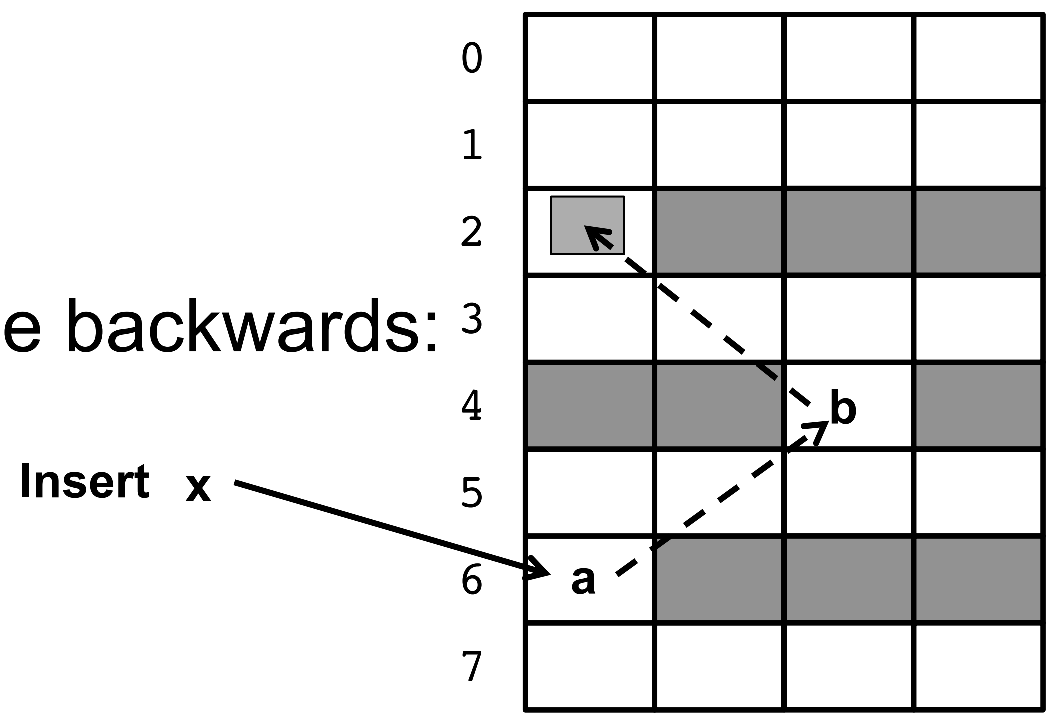


Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

- Step2: Move hole backwards:

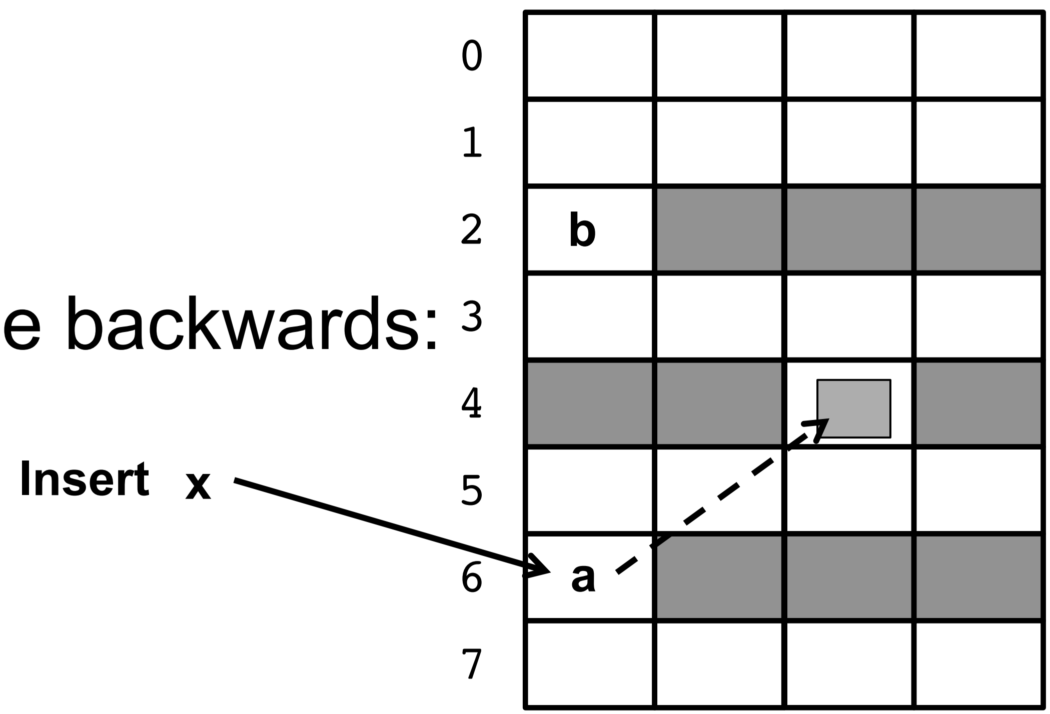


Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

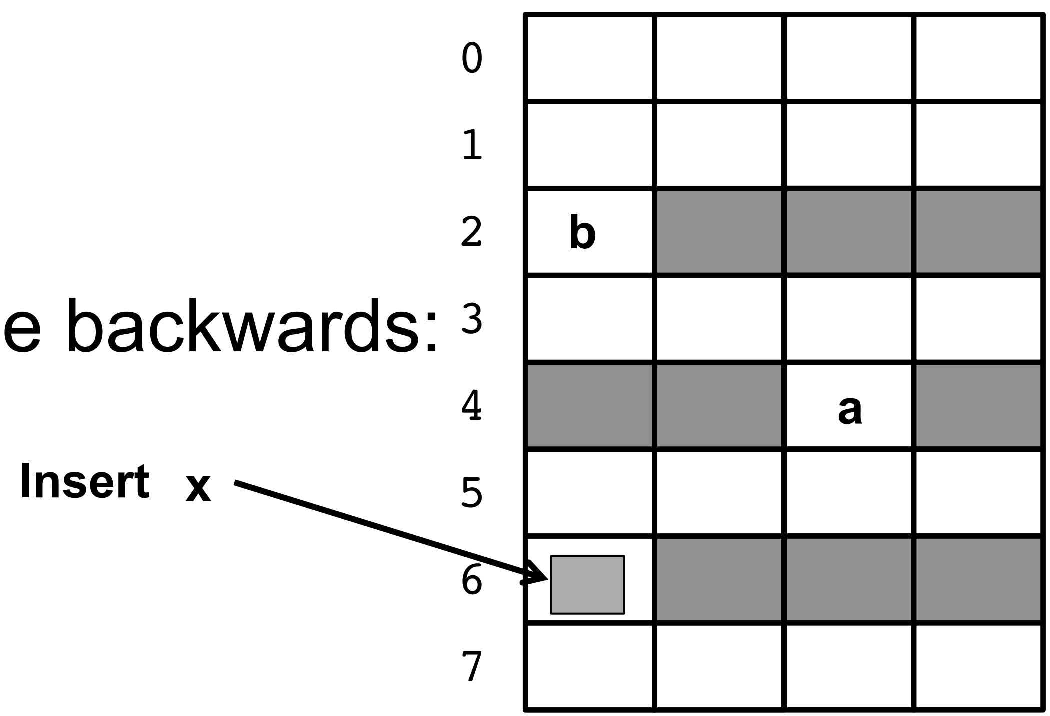
- Step2: Move hole backwards:



Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

- Step2: Move hole backwards:

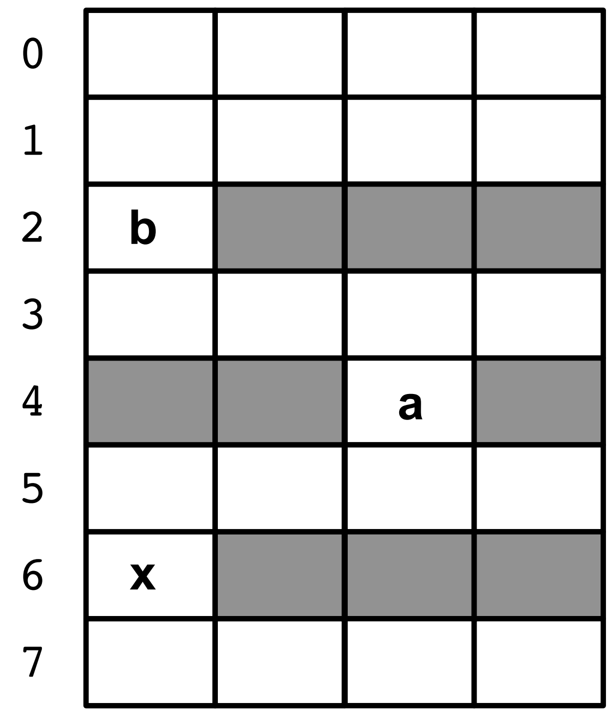


Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

- Step2: Move hole backwards:



Only need to ensure each move is atomic w.r.t. reader

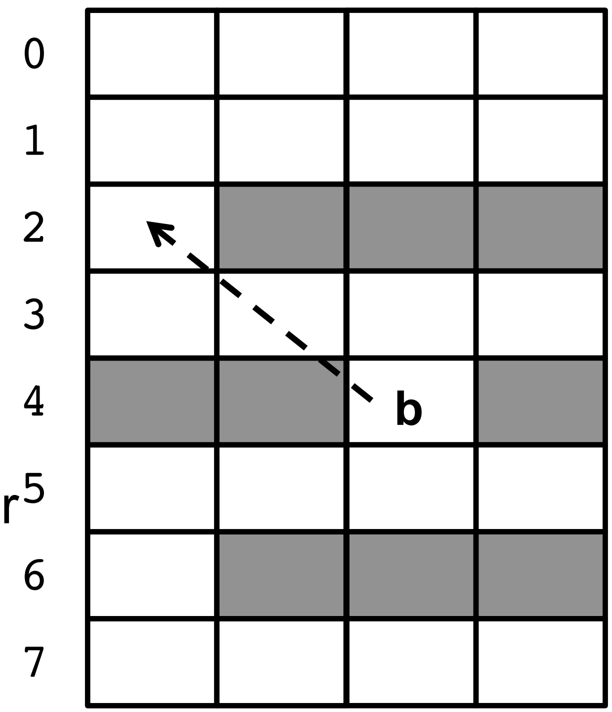
How to Ensure Atomic Move

- e.g., move key “b” from bucket 4 to bucket 2

- A simple implementation:

  Lock bucket 2 and 4
  Move key
  Unlock bucket 2 and 4

- Our approach: Optimistic locking
  - Optimized for read-heavy workloads
  - Each key mapped to a version counter
  - Reader detects version change (described in paper)



Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

CLOCK-based LRU eviction

- Better space efficiency and concurrency

2ptr/key => 1bit/key, concurrent update

Additional algo & tuning improvements

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

CLOCK-based LRU eviction

- Better space efficiency and concurrency

2ptr/key => 1bit/key, concurrent update

Additional algo & tuning improvements

Avoid unnecessary full-key comparisons on hash collision

Conclusion

- Optimistic cuckoo hashing
  - High space efficiency
  - Optimized for read-heavy workloads
  - Source Code available:
    github.com/efficient/libcuckoo

- MemC3 improves Memcached
  - 3x throughput, 30% more (small) objects
  - Optimistic Cuckoo Hashing, CLOCK, other system tuning

Exercise

Making LRU able to manage non equi-sized items

- Each item is characterized by its size S_i and its ΔT_i (# of references from now until the time that item i was last referenced)
- A different cache organization: all items of size 2^{i-1}-(2^i-1) are accommodated in the same <u>virtual</u> LRU queue
- We compare the quantity $S_i * \Delta T_i$ for the items that are in the tail of each LRU queue, and we evict the item with the largest such value
- *<u>Prove that</u>*: this decision is never two times worse than the optimal case, i.e., evicting the item with the largest $S_i * \Delta T_i$ among all items

Solution

- Let j be the item that was selected for eviction, and let m be the item with the largest $S*\Delta T$ value among all items, i.e., m=argmax{$S*\Delta T$}
- Let x be the least recently used item in the virtual LRU queue where item m belongs. Then $S_x \geq S_m/2$ (worst case scenario) and $\Delta T_x \geq \Delta T_m$ (x is at the tail of LRU queue), and thus $S_x * \Delta T_x \geq S_m * \Delta T_m /2$
- But, $S_j \Delta T_j \geq S_x \Delta T_x$ (because of our eviction policy)
- Therefore, $S_j \Delta T_j \geq S_m * \Delta T_m /2$ **O.E.Δ.**