SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems

Yuanyuan Sun, Yu Hua, Song Jiang*, Qiuyu Li, Shunde Cao, Pengfei Zuo

Huazhong University of Science and Technology
*University of Texas, Arlington

Presented in the USENIX ATC 2017

Indexing services in cloud storage

- Large amounts of data
  - From small hand-held devices to large-scale data centers
  - 44ZB in total, 5.2TB for each user in 2020 (IDC' 2014)

- Fast query services are important to both users and systems
  - Returning accurate results in a real-time manner
  - Improving system performance and storage efficiency

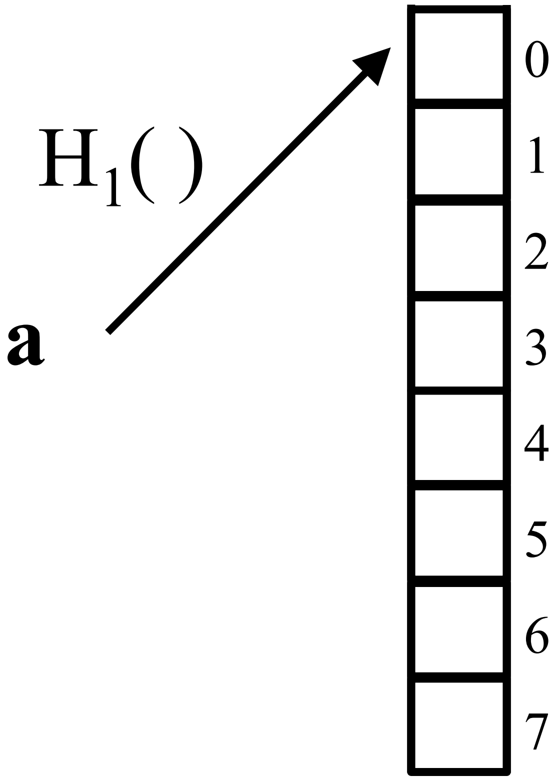
The importance of hash tables

- Hash tables are widely used in data stores and caches
  - Key-value stores, e.g., Memcached, Redis
  - Relational databases, e.g., MonetDB, HyPer
  - In-cache index (ICS 2014, MICRO 2015)
- Strengths:
  - Constant-scale addressing complexity ~O(1)
  - Fast query response
- Weakness:
  - Risk of high-latency for handling hashing collisions
- Cuckoo hashing

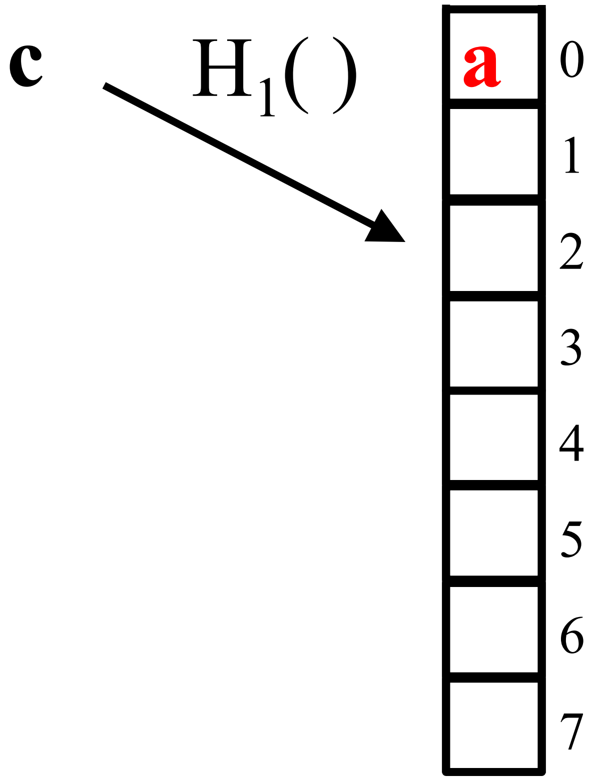
Cuckoo hashing

- Kick-out operations: like cuckoo birds
- Open addressing
- Supporting fast lookups: O(1) time complexity
- However, insertion latency can be very high and unpredictable, especially
  - **when an endless loop occurs!**

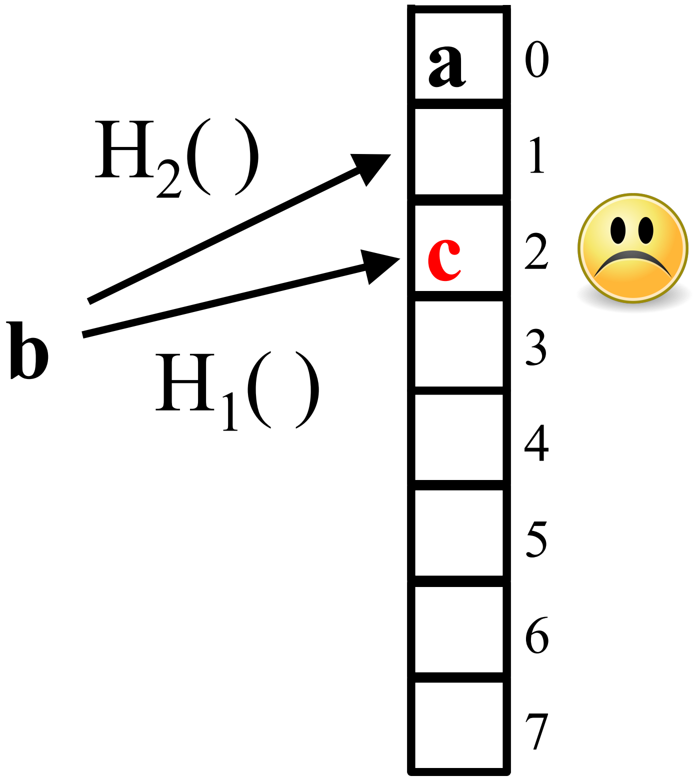
How is an endless loop formed?



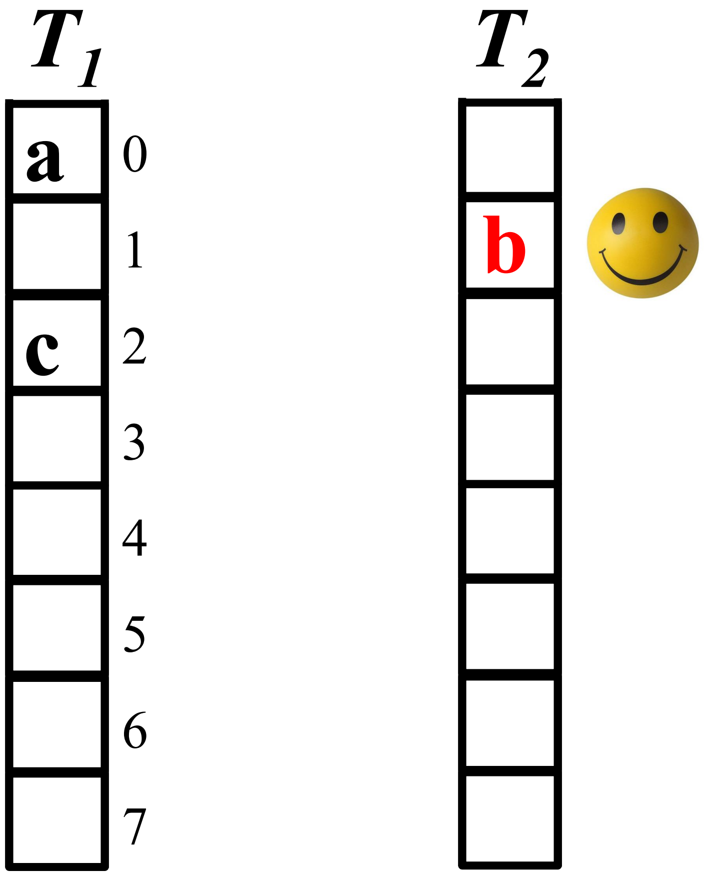
How is an endless loop formed?



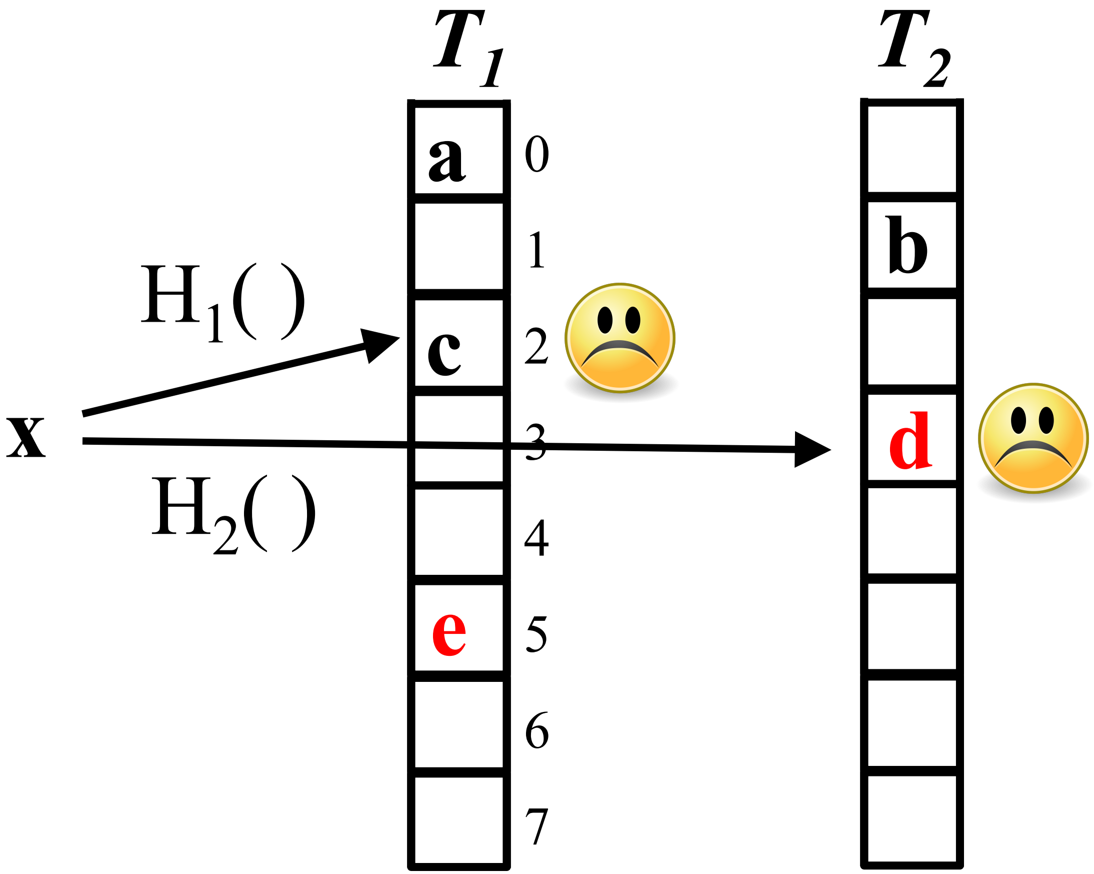
How is an endless loop formed?



How is an endless loop formed?

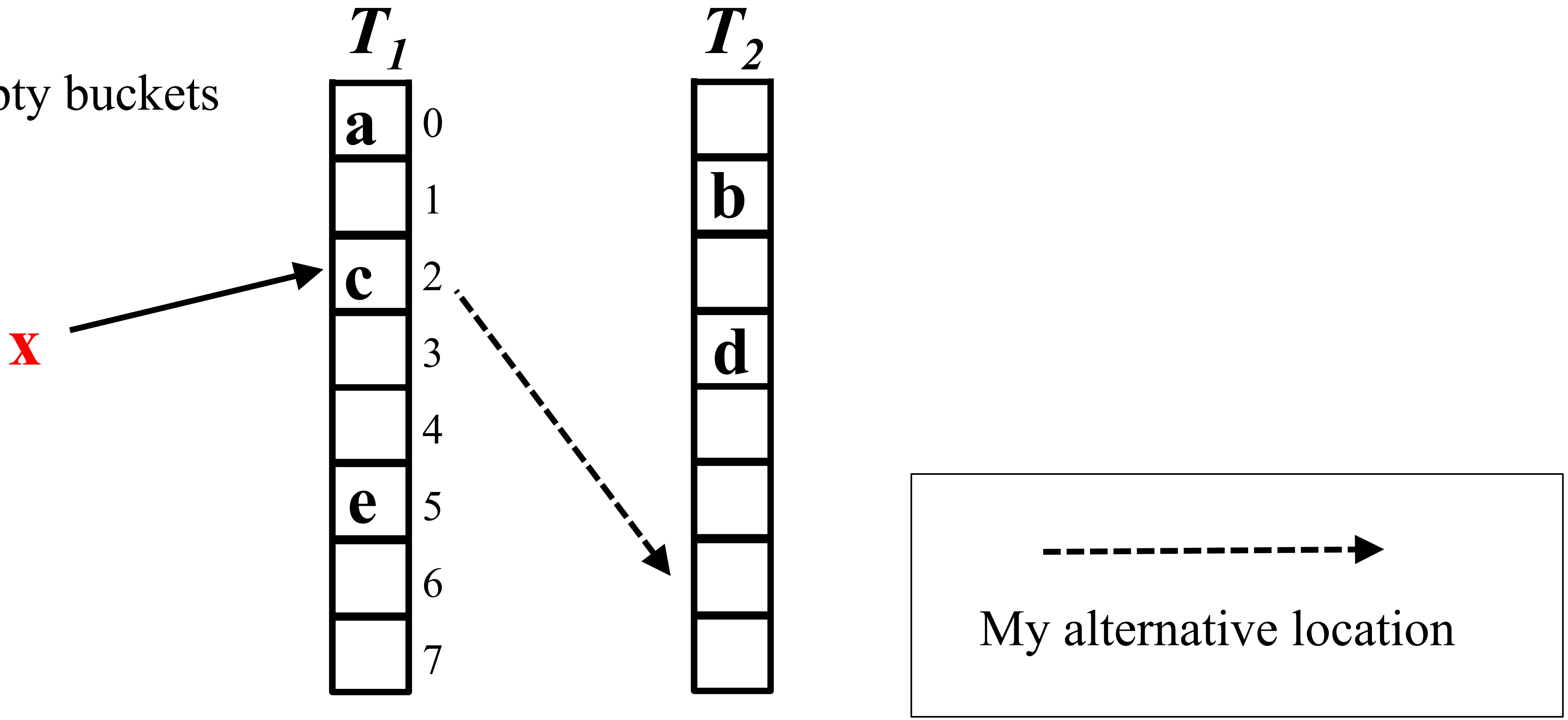


How is an endless loop formed?



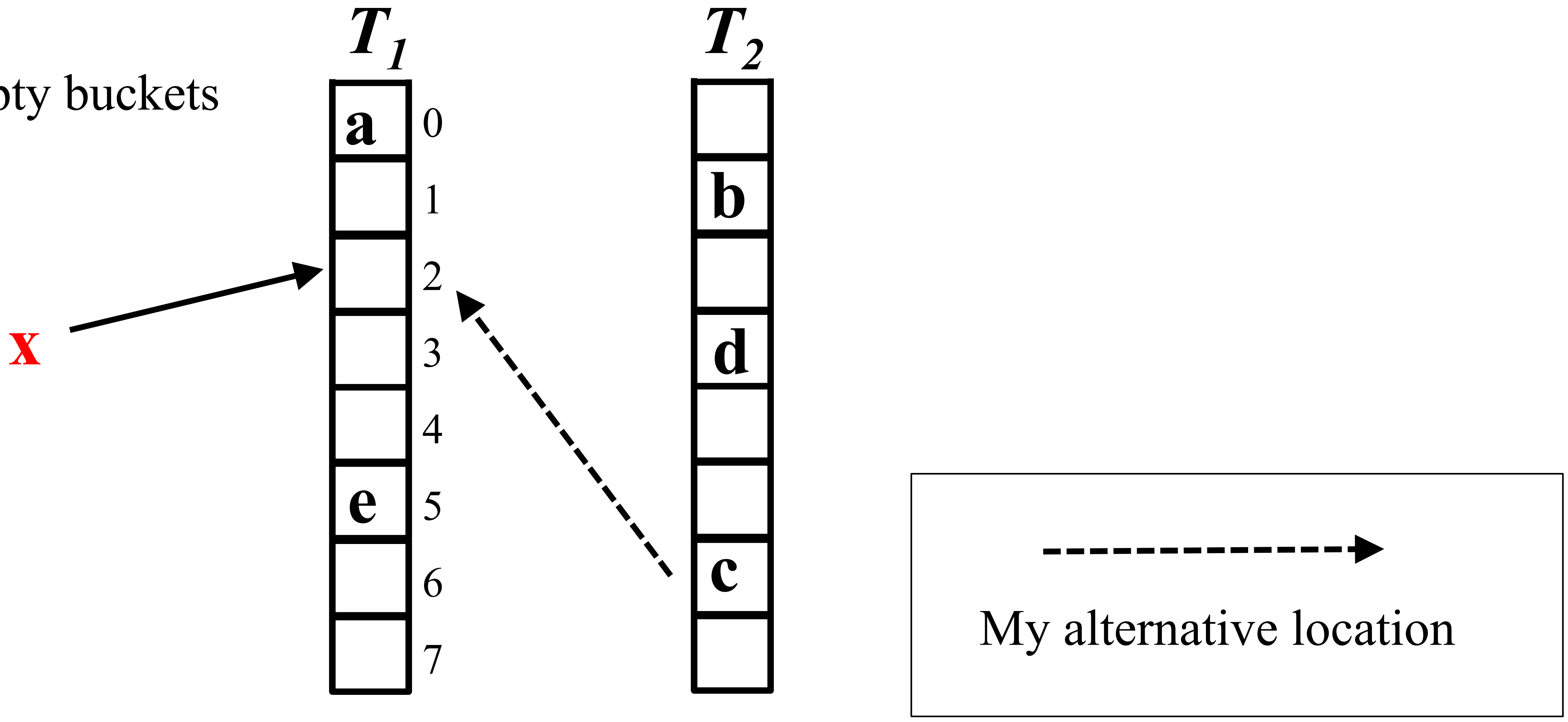
How is an endless loop formed?

Kickout for empty buckets



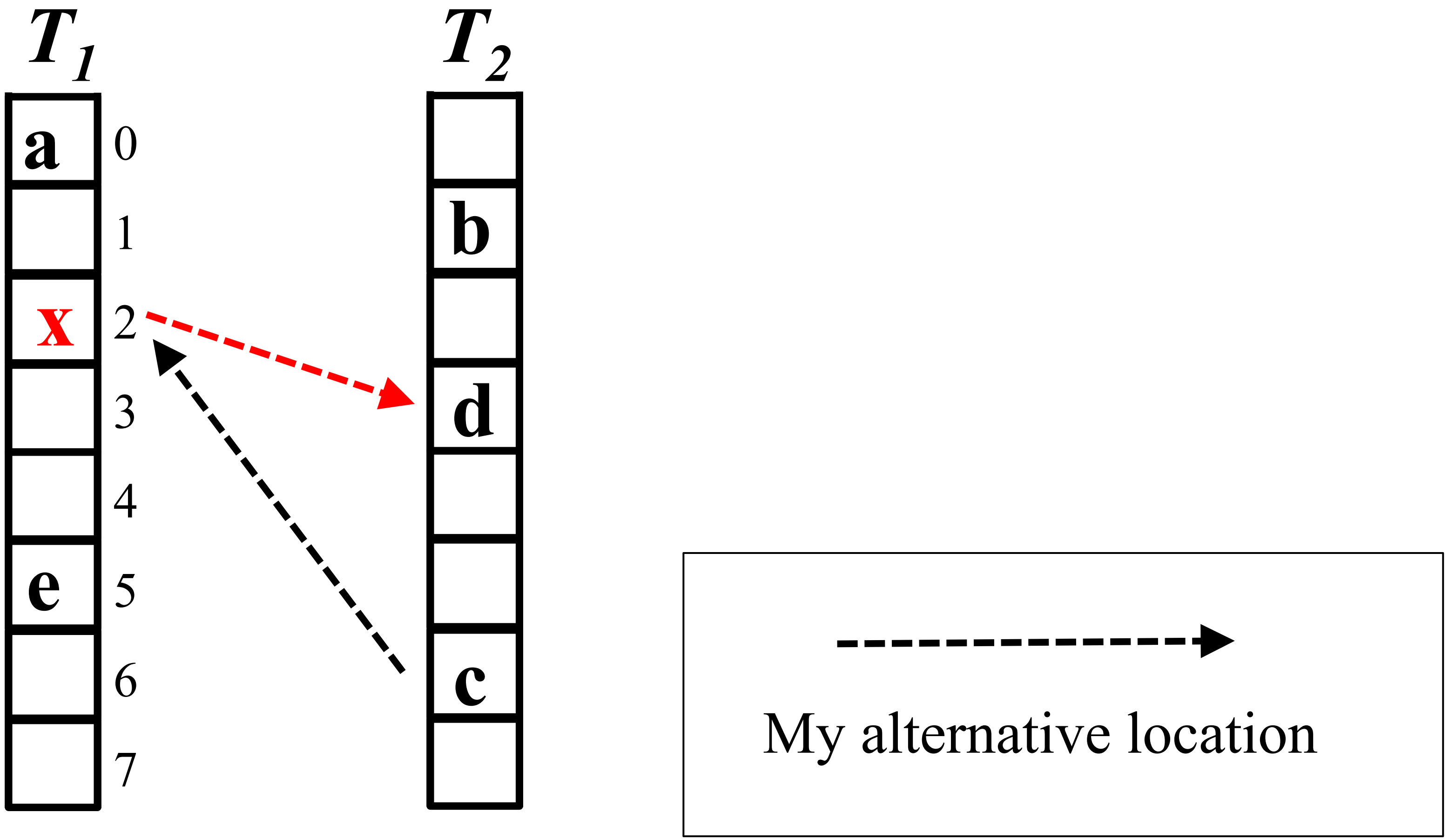
How is an endless loop formed?

Kickout for empty buckets

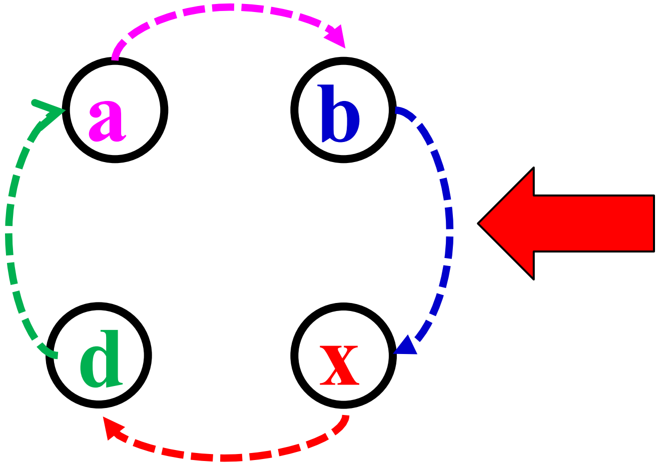


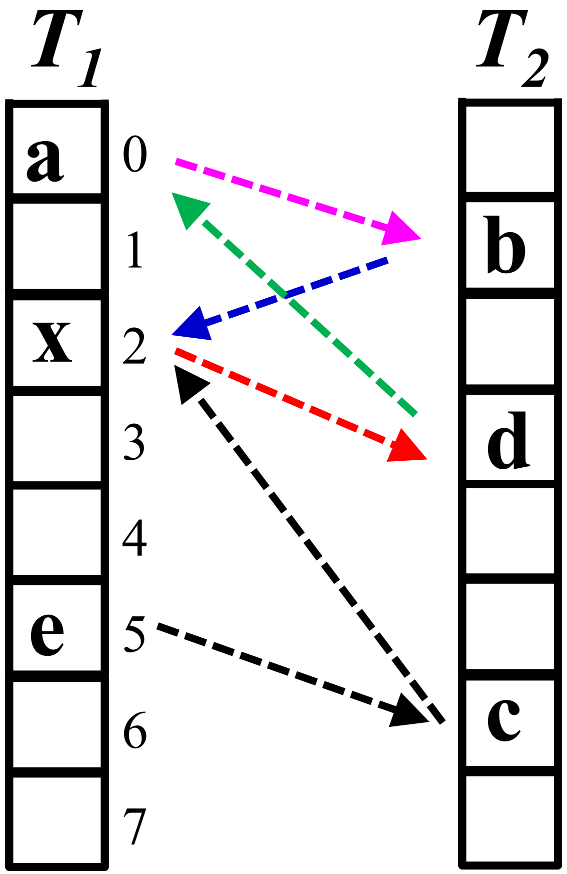
How is an endless loop formed?

Kickout for empty buckets

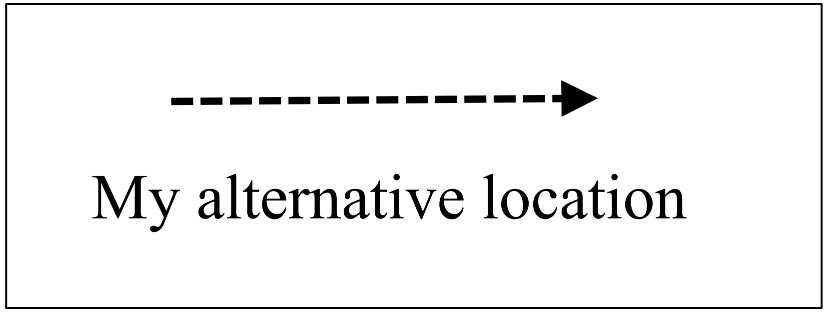


How is an endless loop formed?



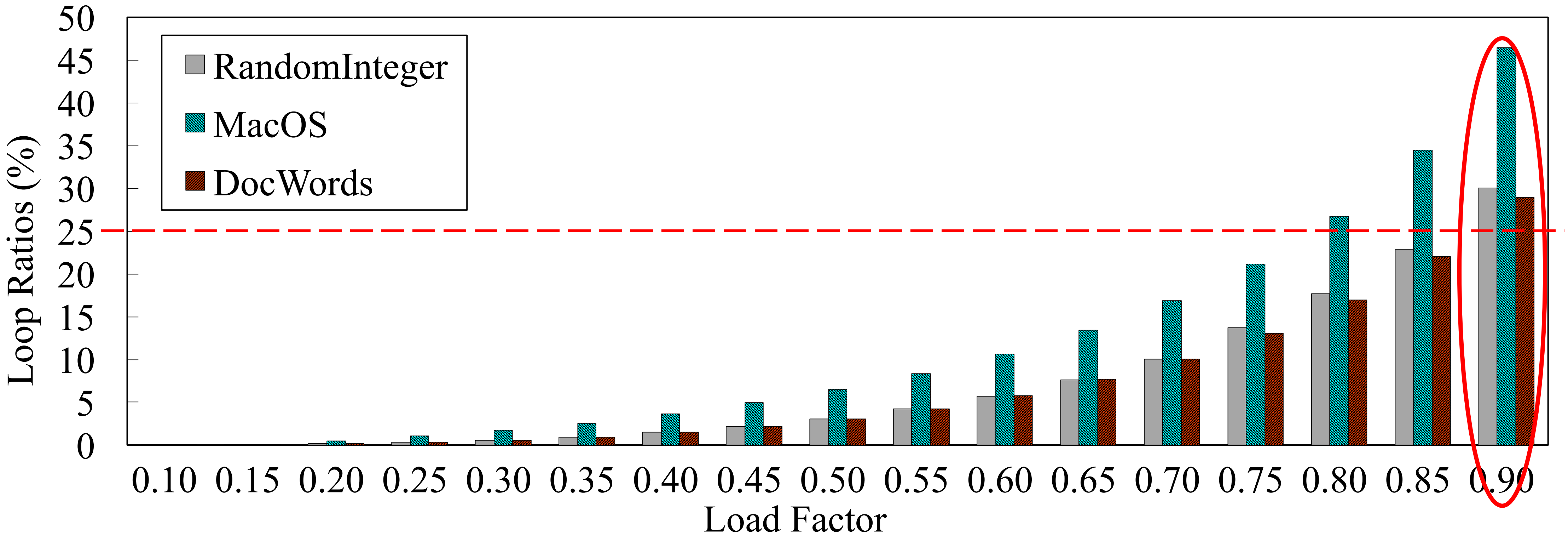


- An **endless loop** is formed.
- **Endless kickouts** for any insertion within the loop.



Observations

- Endless loops widely exist in the Cuckoo hashing structures.
  - More than 25% (cuckoo hashing with a stash)
- Loop ratio: the percentage of insertion failures due to loops



Existing works

- ChunkStash @USENIX ATC'10
  - Collisions: resursive strategy to relocate one of keys in candidates
  - Loops: an auxiliary linked list (or, hash table)

- MemC3 @NSDI'13
  - Collisions: random and repeat relocation (500 times)
  - Loops: an expansion process
  - Stand-alone implementation: libcuckoo @ EuroSys'14

- Horton tables @USENIX ATC'16
  - Recursively evicting keys within a certain search tree height

Motivations

- Due to endless loops:
  - Substantial resources consumption
    - A large number of step-by-step kick-out operations
  - Unbounded performance
    - Fruitless effort

- Design Goal:
  - Predetermining and avoiding occurrence of endless loops

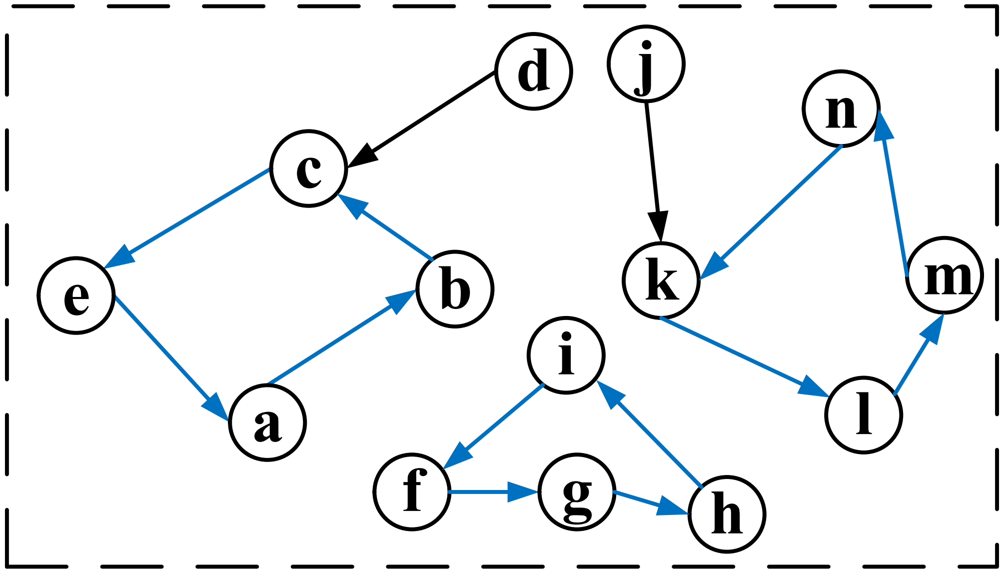
Our approach: SmartCuckoo

- Tracking item placements in the hash table
  - Representing the hashing relationship as a directed pseudoforest
  - Classifying item insertions into three cases
  - **Predetermining and avoiding loops** during insertion without any kick-out attempts.

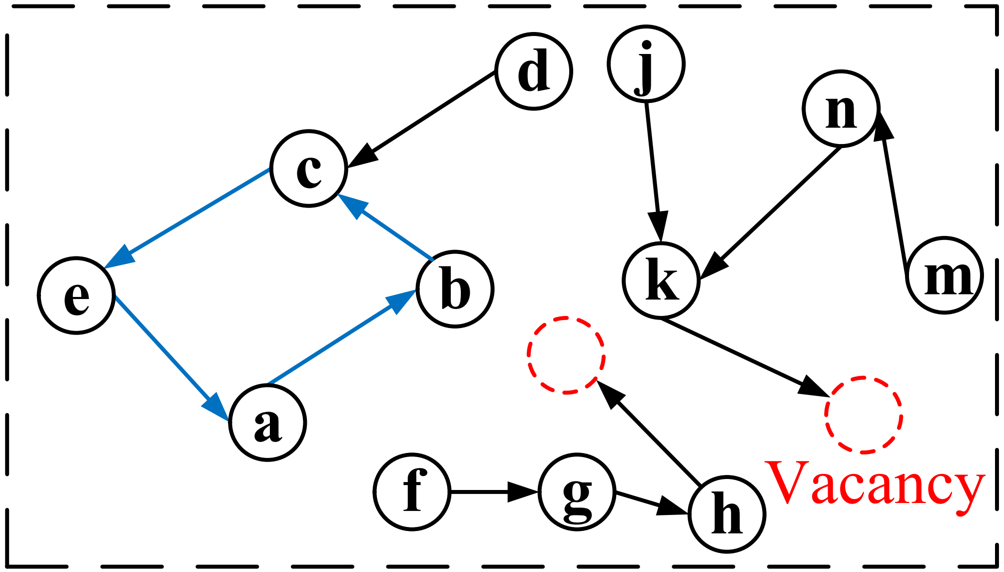
How to identify loop(s)?

- Pseudoforest:
  - A graph: each vertex has an outdegree of at most one
  - Each connected component (subgraph) has at most one cycle (loop)
  - In a subgraph:

Loop ⟷ #Vertices = #Edges **No loop ⟷ #Vertices = #Edges + 1**



Maximal



Non-maximal

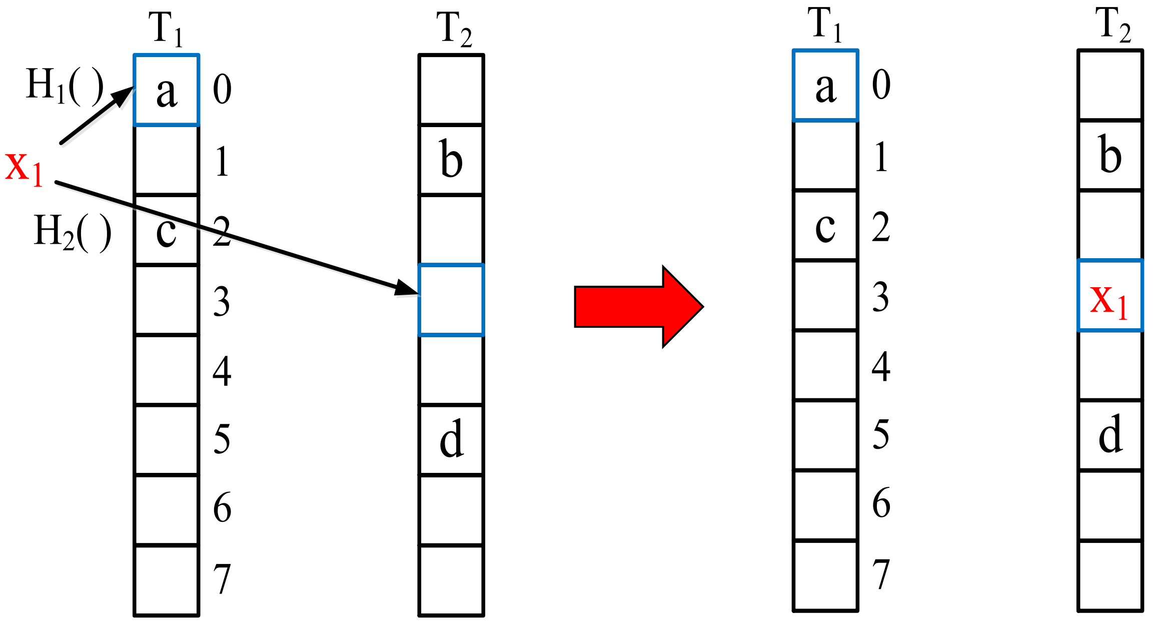
Classification and predetermination

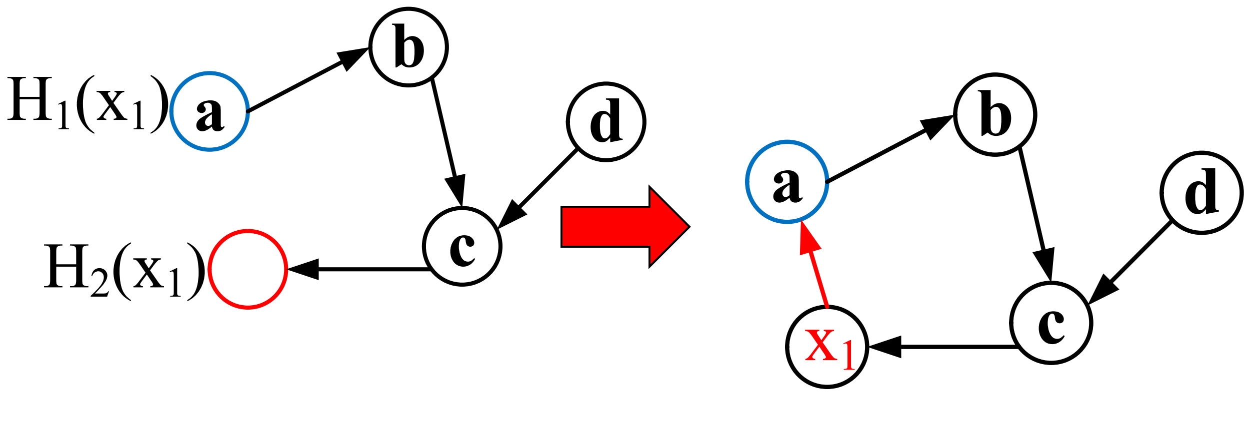
- Three cases depending on the number of vertices added to the graph
  - v+0, v+1, and v+2
  - v+0: 5 possible scenarios based on the status of corresponding subgraph(s)

Three cases	v+0					v+1	v+2
Two insert positions of a key	Same subgraph		Different subgraphs			A new one	Two new ones
Subgraph status	Non-maximal	Maximal	Both non-maximal	A maximal and a non-maximal	Both maximal	-	-
Scenarios	(a)	(e)	(b)	(c)	(d)	-	-

v+0: (a) One non-maximal subgraph

- One empty bucket
- Success!

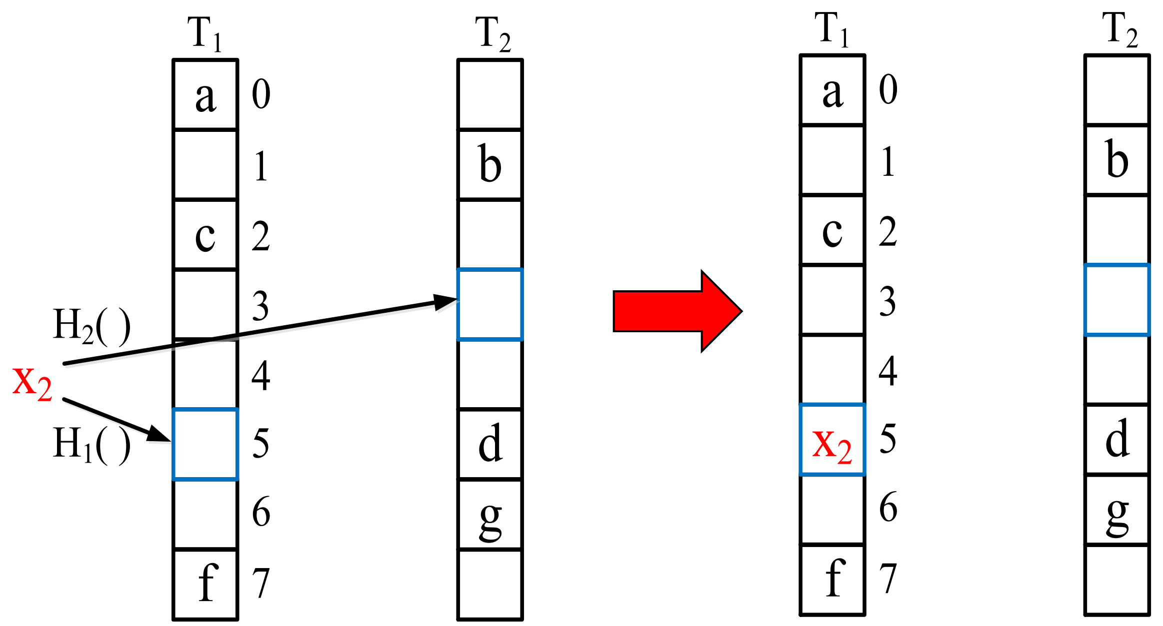


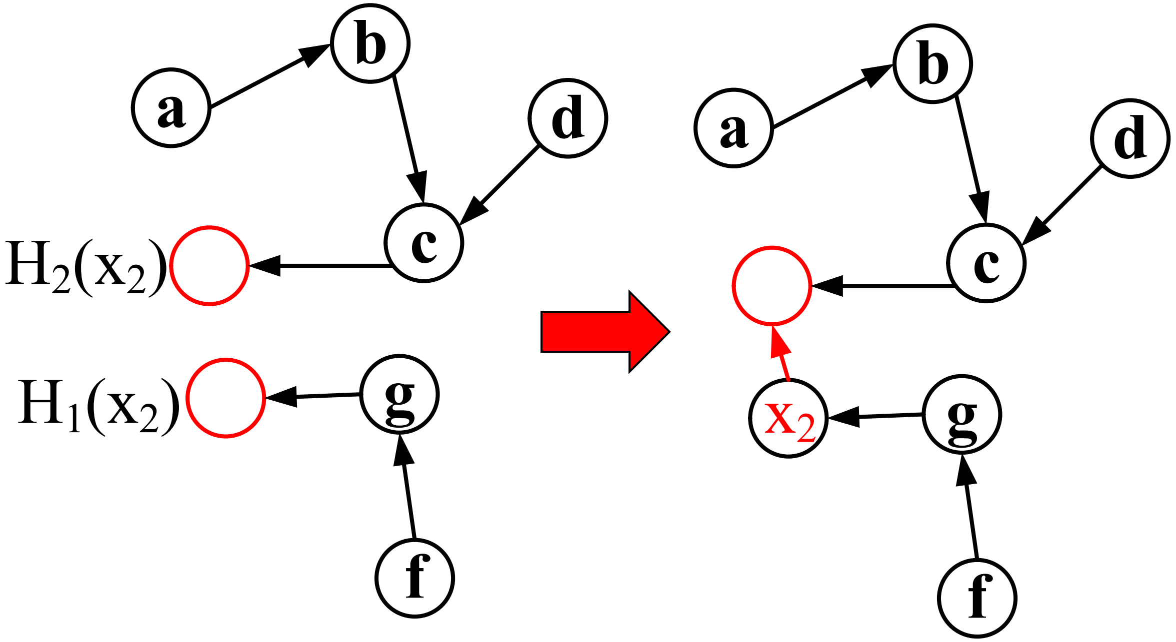


Pseudoforest

v+0: (b) Two non-maximal subgraphs

- Two empty buckets
- Success!

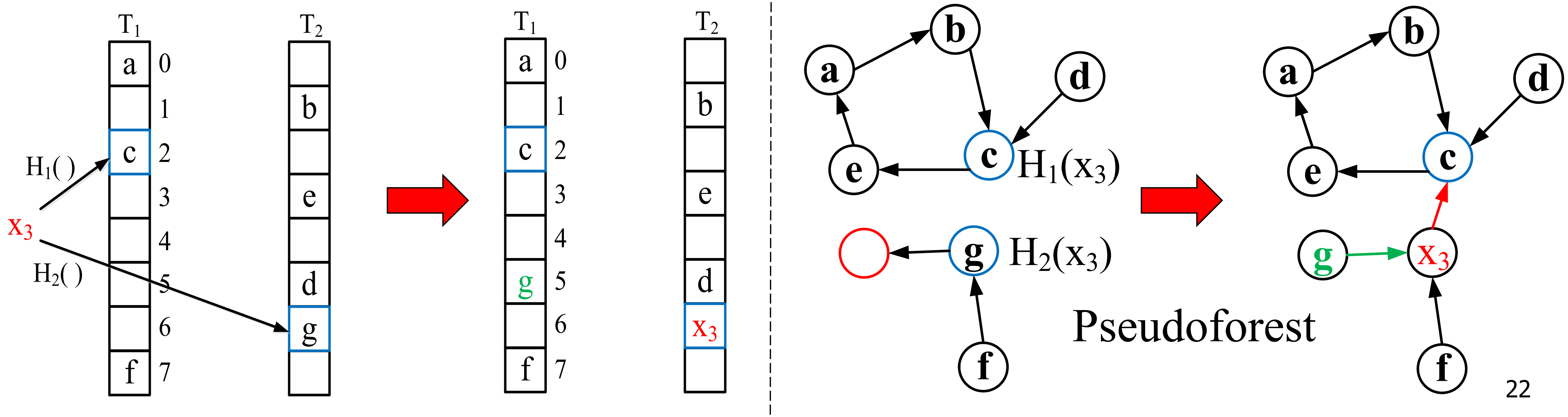




Pseudoforest

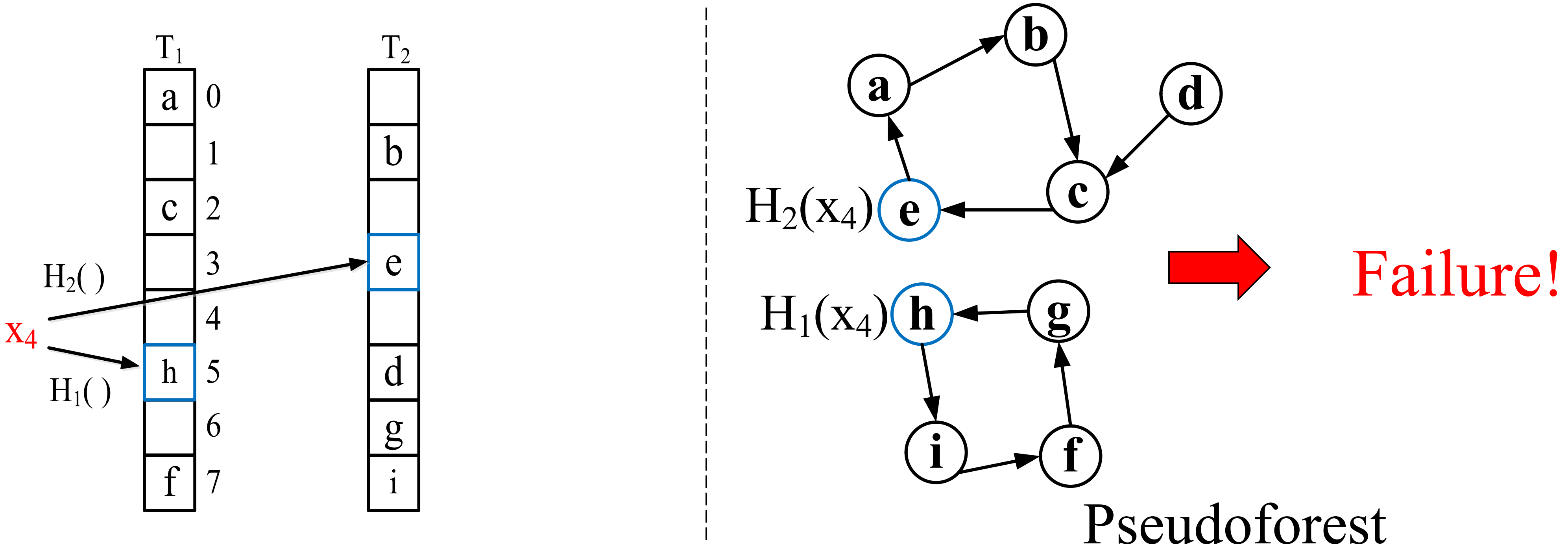
v+0: (c) One maximal and one non-maximal

- One loop and one empty bucket
- Conventional cuckoo hashing: taking a random walk
  - T_1: executing extra useless kick-out operations
  - T_2: making a success
  - SmartCuckoo: directly selecting to enter from T_2
- Success!



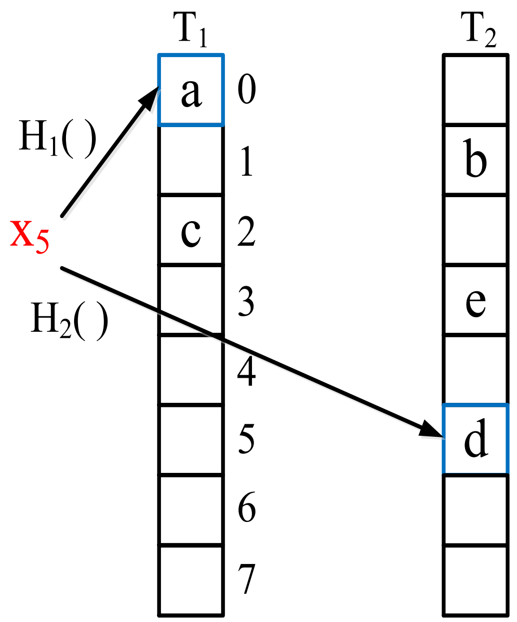
v+0: (d) Two maximal subgraphs

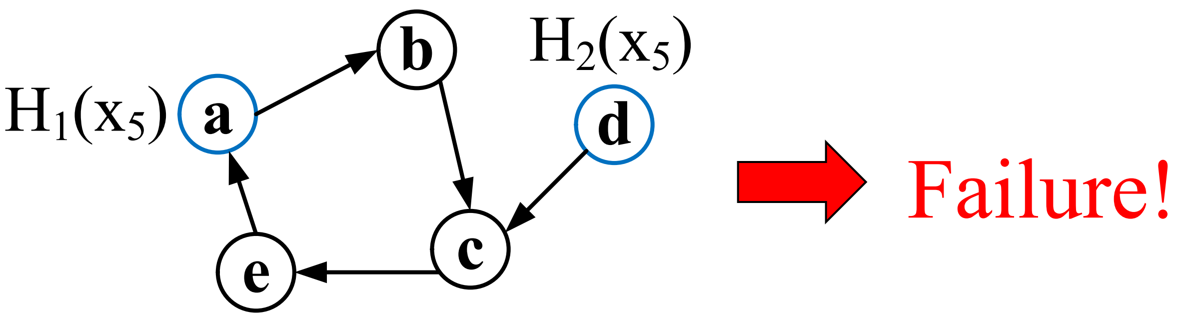
- Two loops!
- Execution:
  - Conventional cuckoo hashing: sufficient attempts, then reporting a failure
  - SmartCuckoo: reporting a failure without any kick-out operations.



v+0: (e) One maximal subgraph

- One loop!

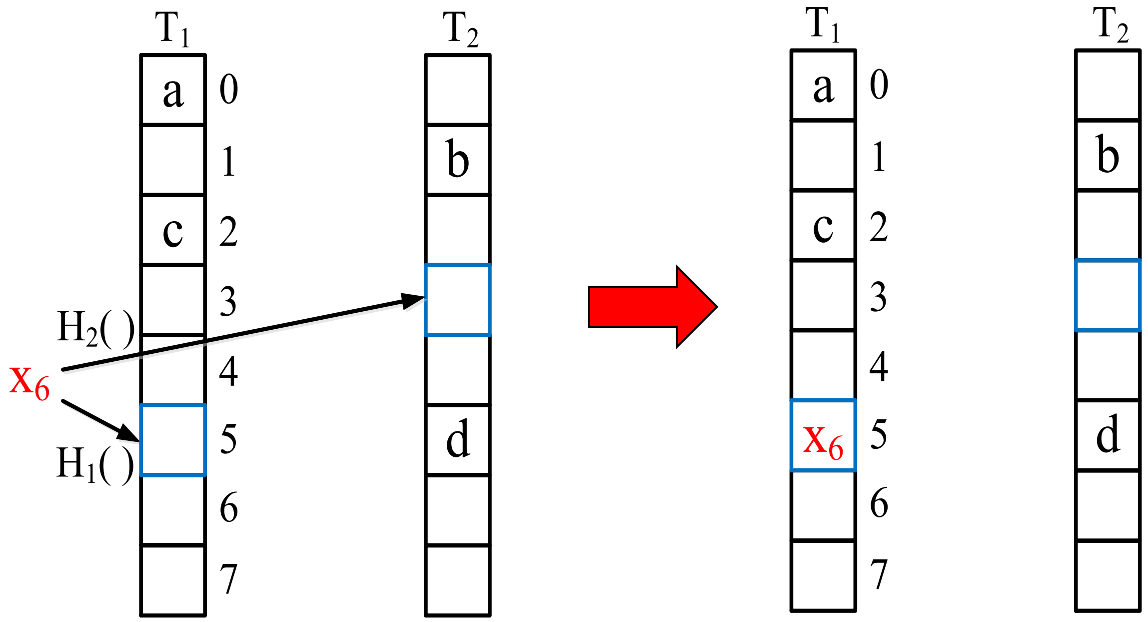


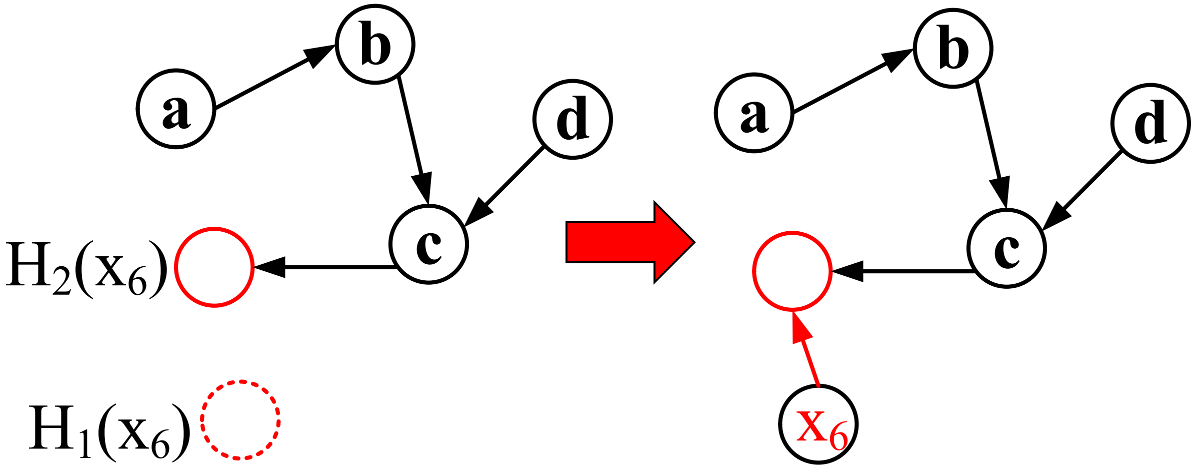


Pseudoforest

Case: v+1

- A new vertex after the item's insertion
- Success!

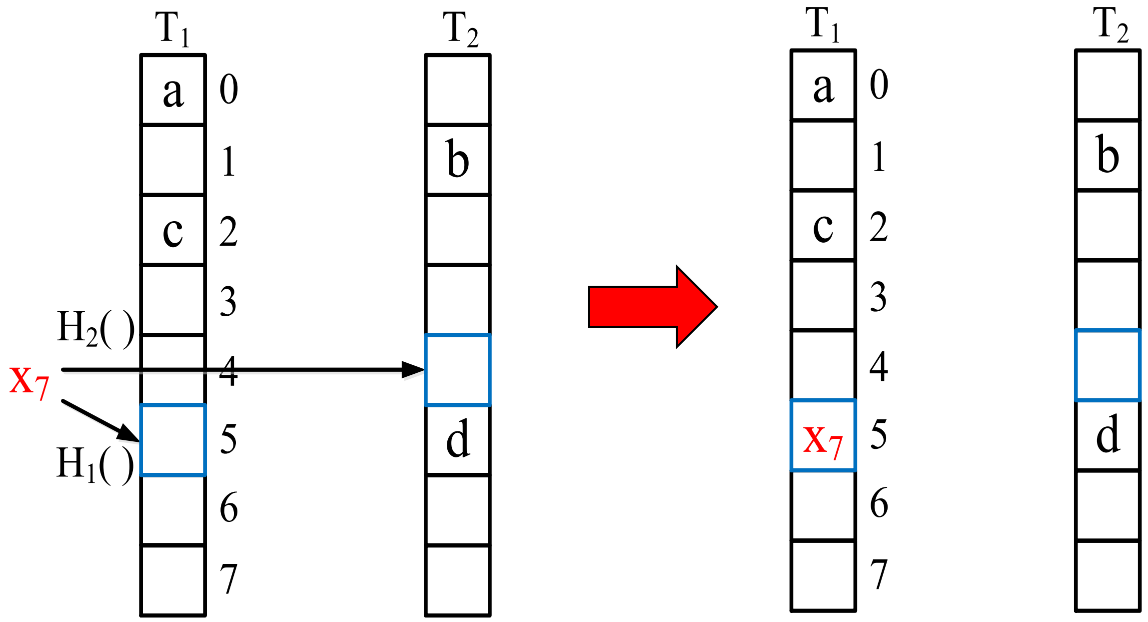


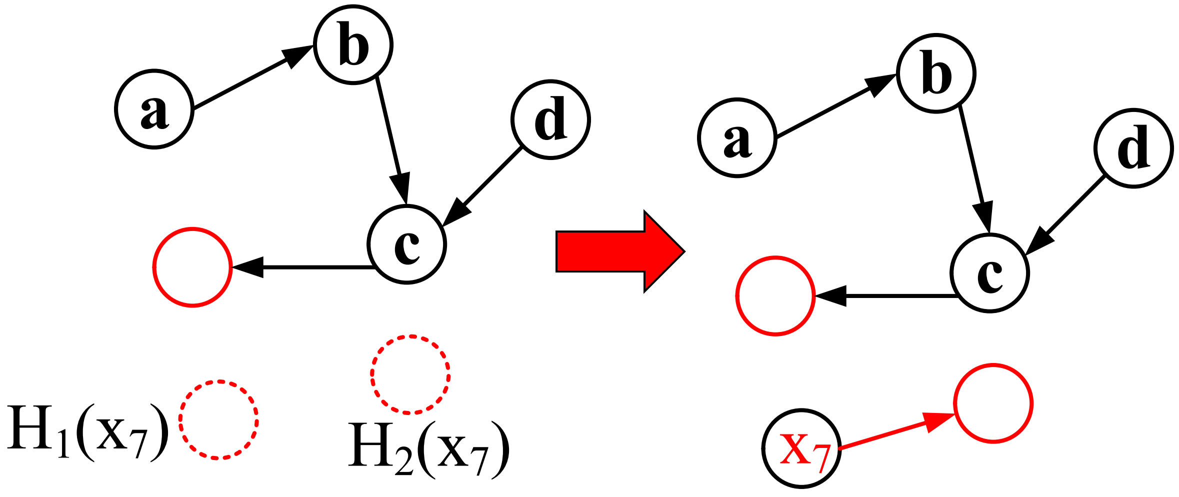


Pseudoforest

Case: v+2

- Two new vertices after the insertion
- Success!



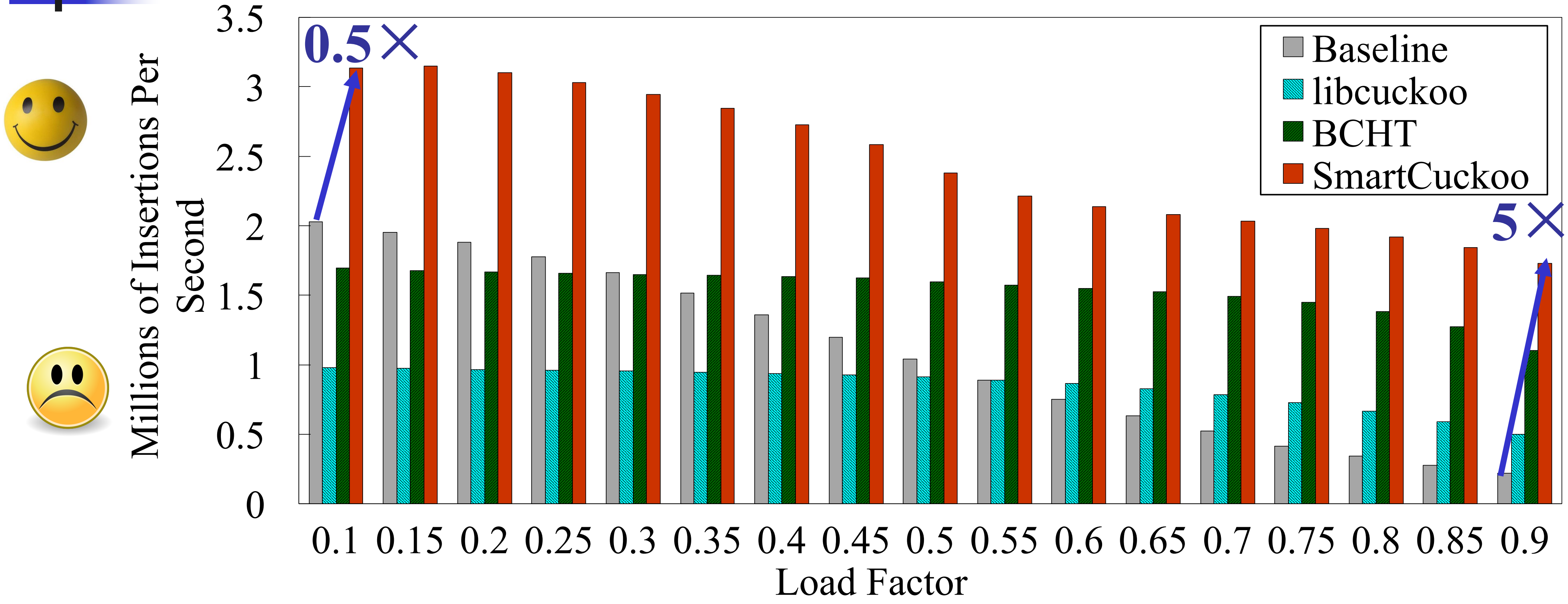


Pseudoforest

Evaluation methodology

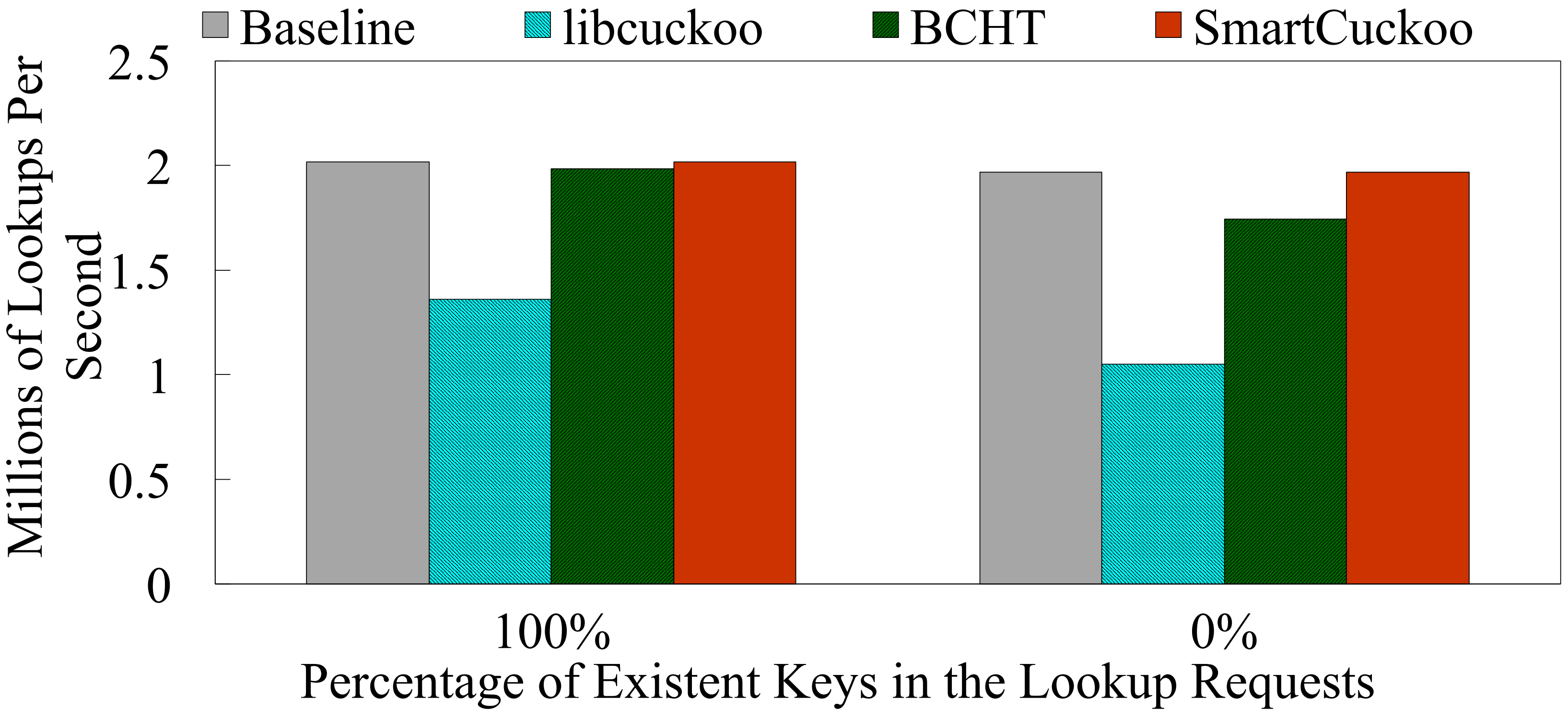
- Comparisons:
  - Baseline (Cuckoo hashing with a stash @ SIAM Journal on Computing'09)
  - libcuckoo @ EuroSys'14
  - BCHT (bucketized cuckoo hash table)
- Traces:
  - RandomInteger: random integer generator @ TOMACS'98
  - MacOS: http://tracer.filesystems.org
  - DocWords: http://archive.ics.uci.edu/ml/datasets/Bag+of+Words
  - YCSB: https://github.com/brianfrankcooper/YCSB @ SOCC'11
- Metrics: in millions of operations per second
  - Insertion throughput
  - Lookup throughput: positive/negative
  - Throughput of workload with mixed queries (YCSB)

Insertion throughput



- SmartCuckoo significantly increases insertion throughputs.
- 0.5× to 5× speedups compared to Baseline.

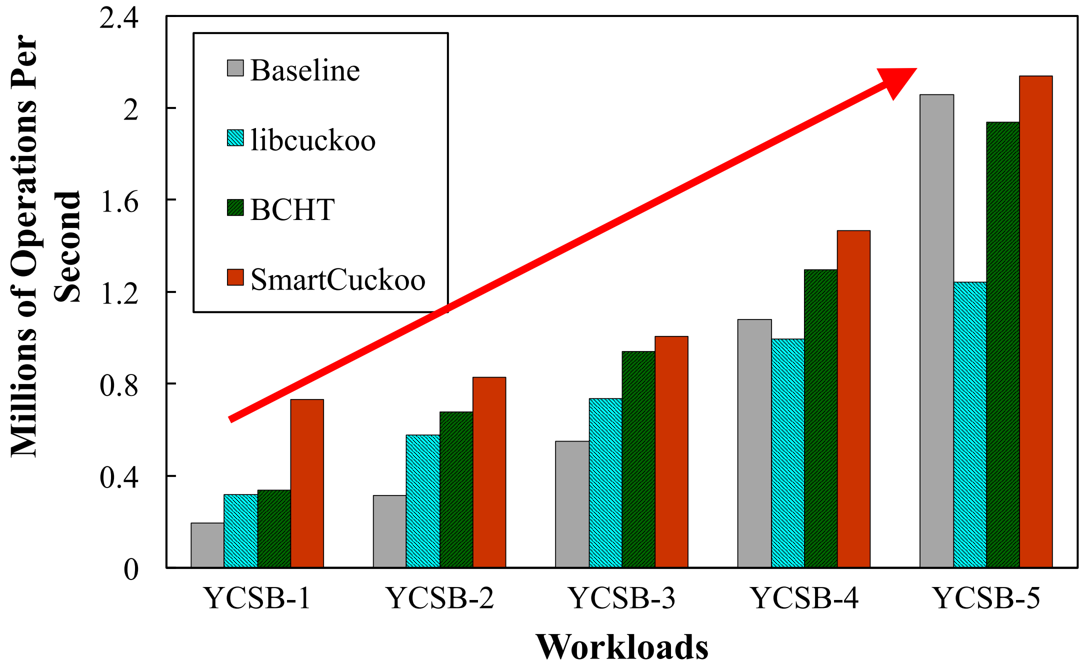
Lookup throughput



- 0%: all candidate positions for a key have to be accessed.
- Almost the same lookup throughput with Baseline.
- Significantly higher than libcuckoo and BCHT.

Throughput of workload with mixed queries

Workload	Insert	Lookup	Update
YCSB-1	100	0	0
YCSB-2	75	25	0
YCSB-3	50	50	0
YCSB-4	25	75	0
YCSB-5	0	95	5



- With the decrease of the percentage of insertions, all schemes increase the throughputs.
- In each workload, SmartCuckoo produces higher throughput than other three schemes.

Conclusion and future work

- Cuckoo hashing is cost-efficient to offer O(1) query performance.
- We address the problem of potential endless loops in item insertion.
- SmartCuckoo helps improve predictable performance in storage systems.

- To-do-list:
  - SmartCuckoo in hash tables with more than two hash functions;
  - The use of multiple slots in each bucket.

Thanks and questions?

Open-source code: https://github.com/syy804123097/SmartCuckoo