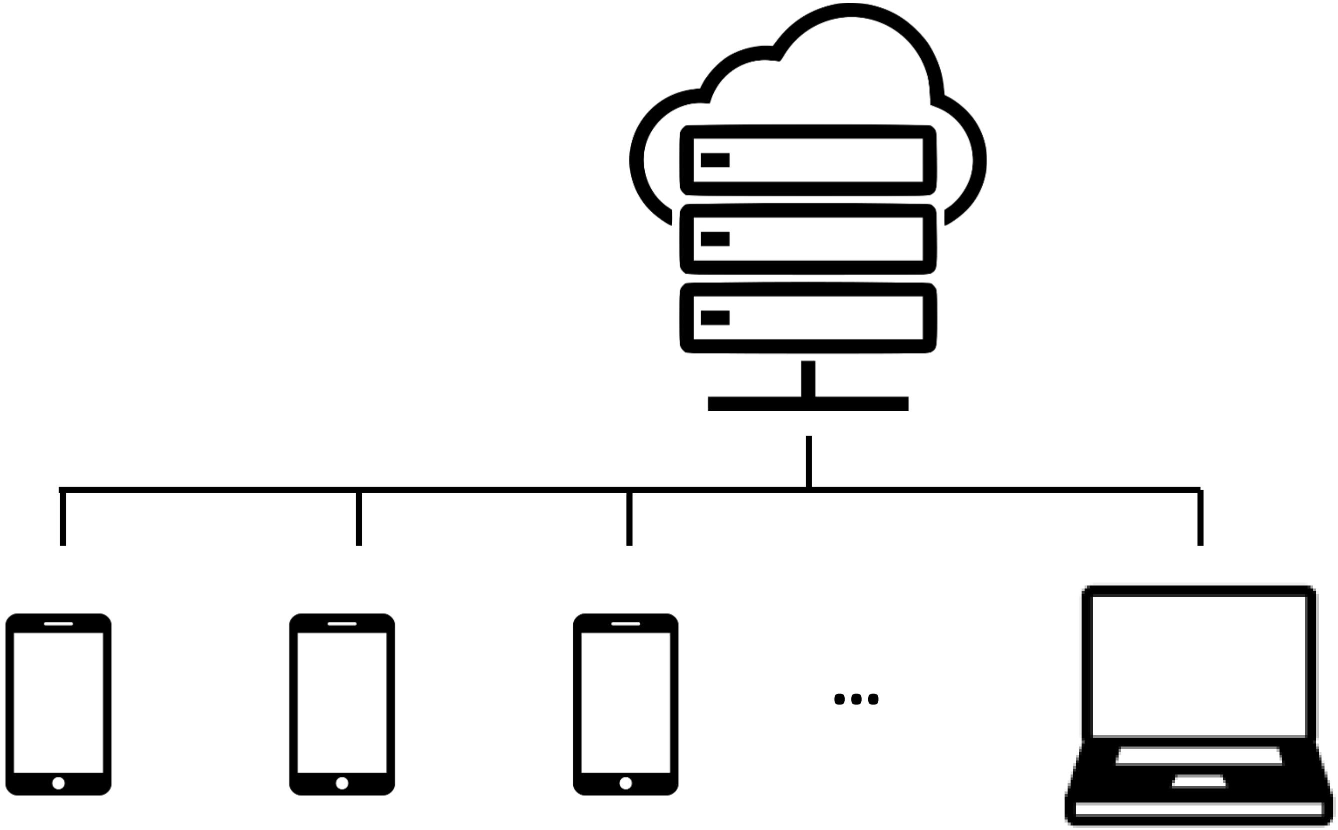
Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems

Yuanyuan Sun, Yu Hua, Zhangyu Chen, Yuncheng Guo
Huazhong University of Science and Technology

USENIX ATC 2019

Query Services in Cloud Storage Systems

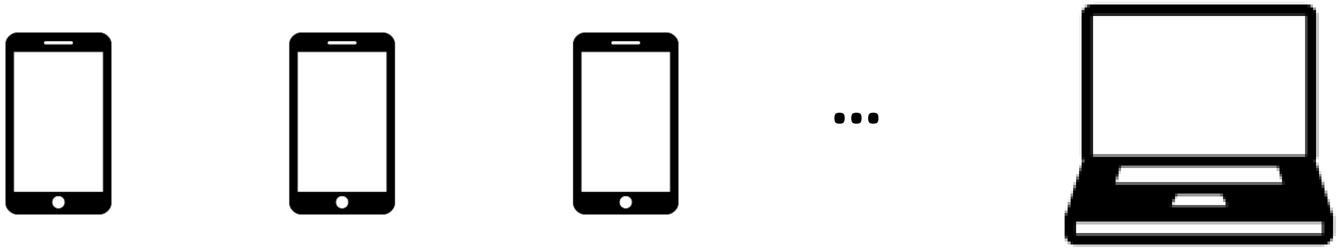
- **Large amounts of data**
  - **300 new profiles and more than 208 thousand photos per minute [September 2018@Facebook]**



Query Services in Cloud Storage Systems

- **Large amounts of data**
  - **300 new profiles and more than 208 thousand photos per minute [September 2018@Facebook]**

Demanding the support of low-latency and high-throughput queries



Hash structures

✓ **Constant-scale read performance**

- **Widely used in key-value stores and relational databases**

Hash structures

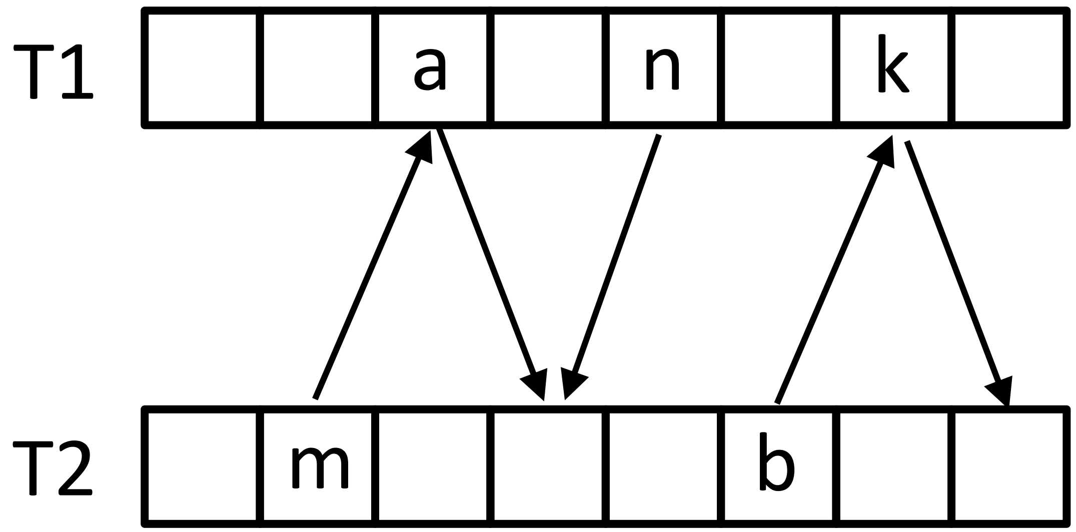
✓ **Constant-scale read performance**

- **Widely used in key-value stores and relational databases**

χ **High latency for handling hash collisions**

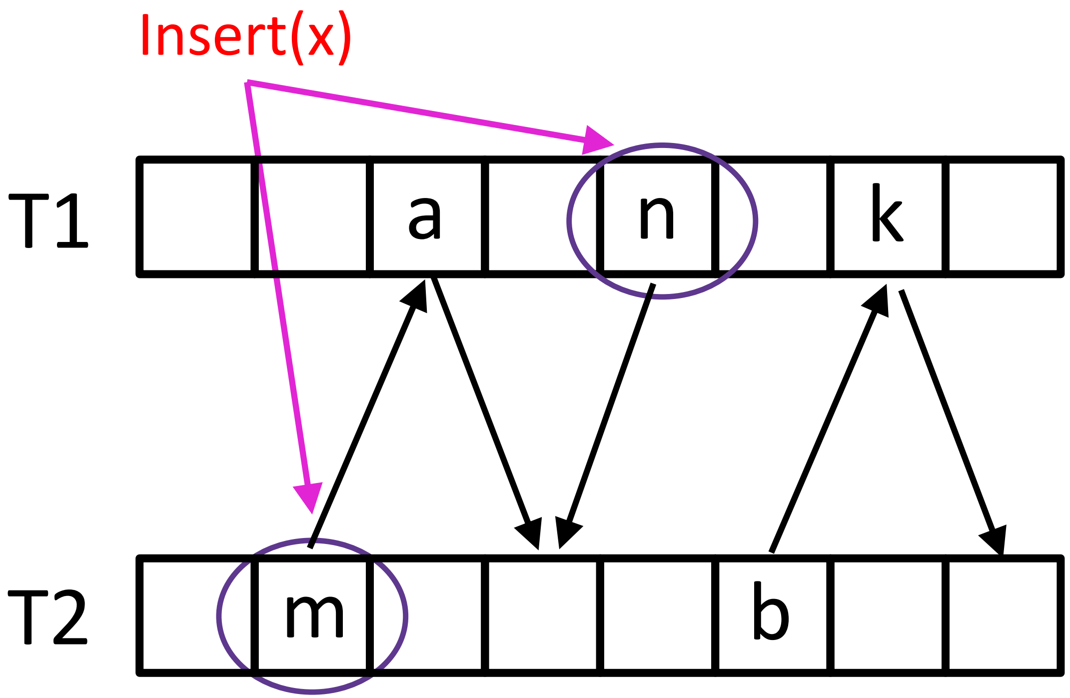
Cuckoo Hashing

- **Multi-choice hashing**
- **Handling hash collisions: kick-out operations**



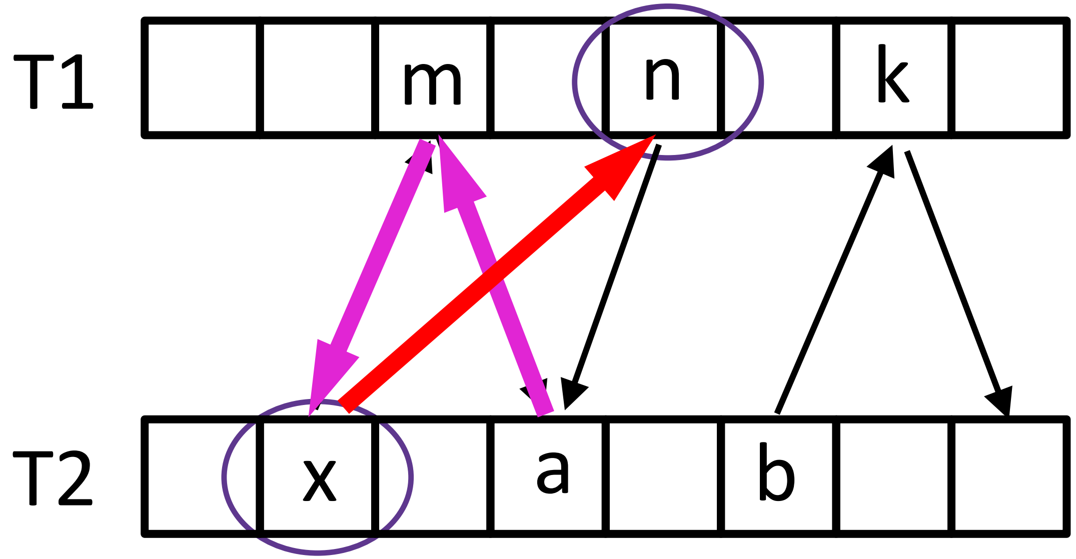
Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations



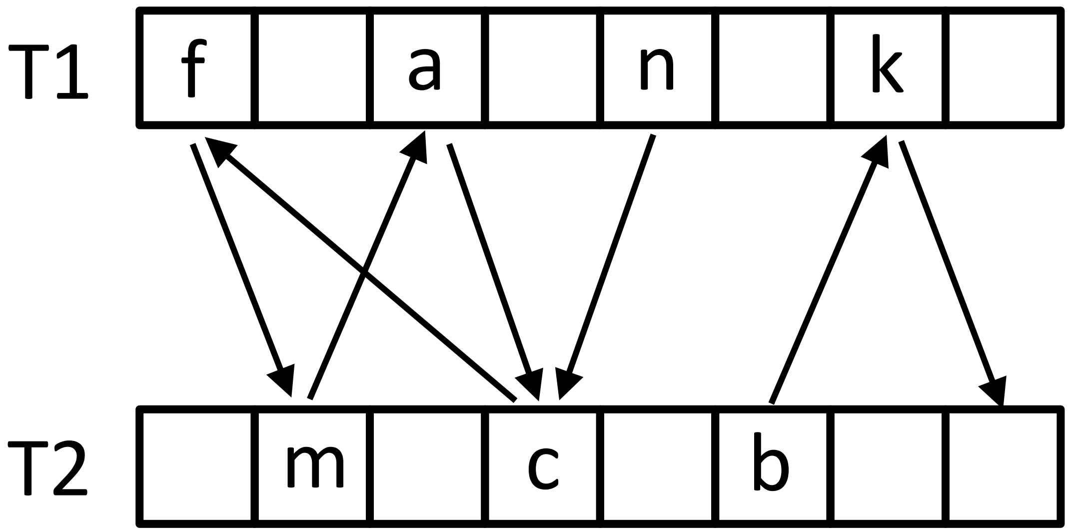
Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations



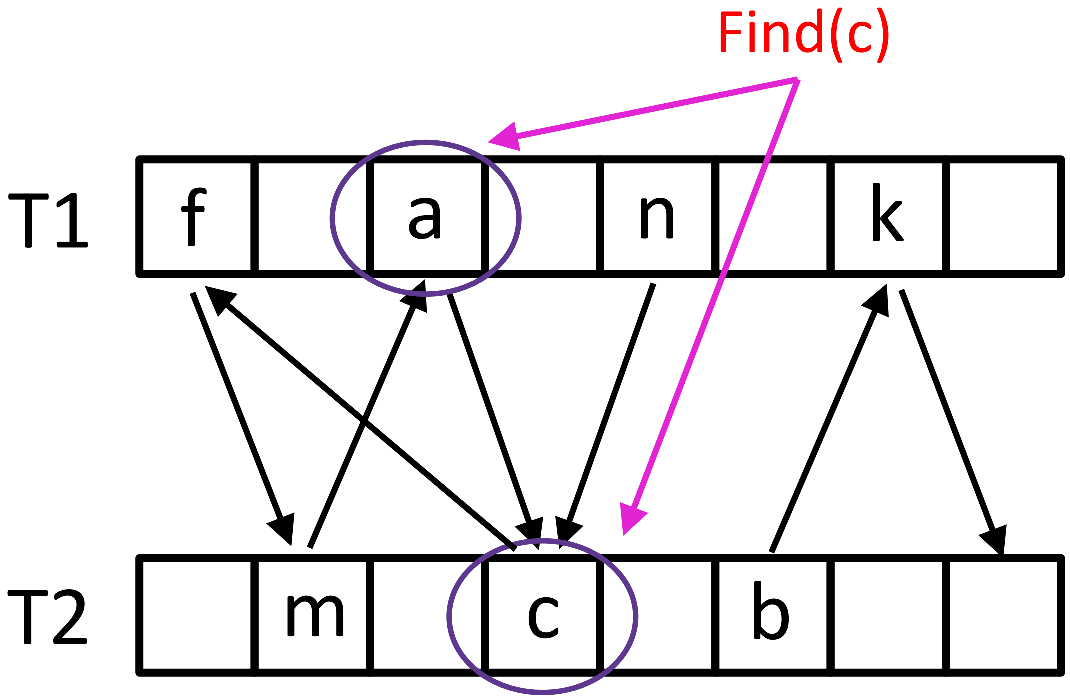
Cuckoo Hashing

- **Multi-choice hashing**
- **Handling hash collisions: kick-out operations**
- **For reads, only limited positions are probed => O(1) time complexity**



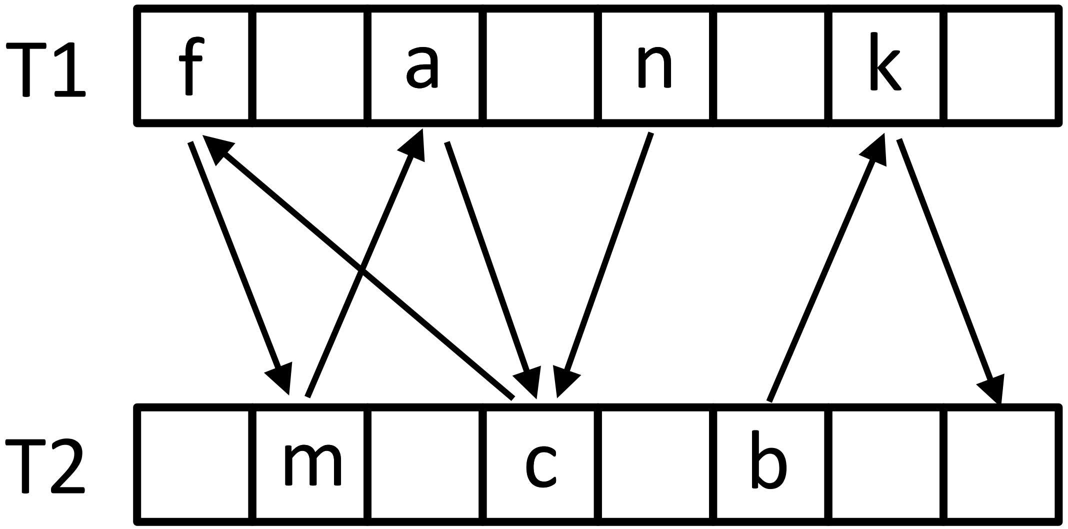
Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity



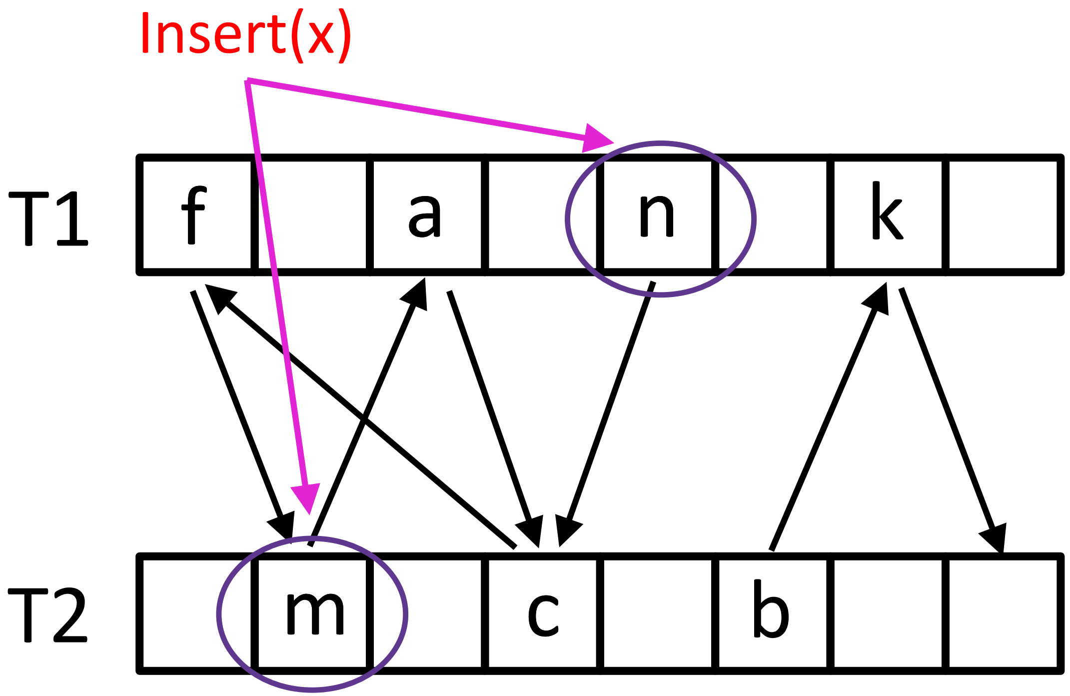
Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity
- For writes, endless loops may occur! => slow-write performance



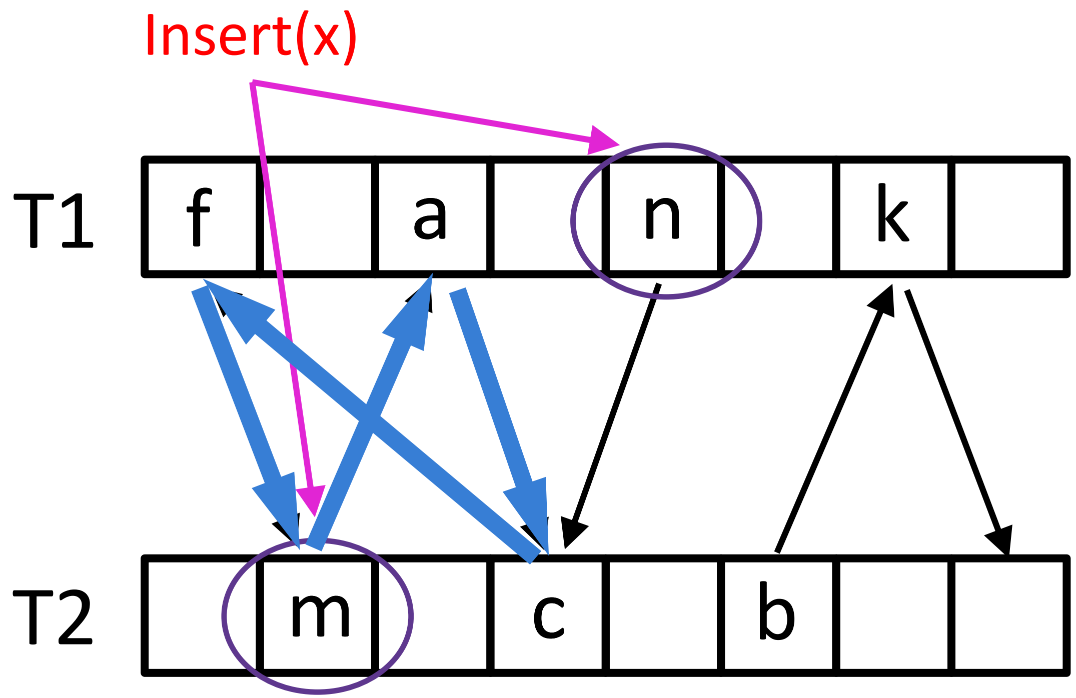
Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity
- For writes, endless loops may occur! => slow-write performance



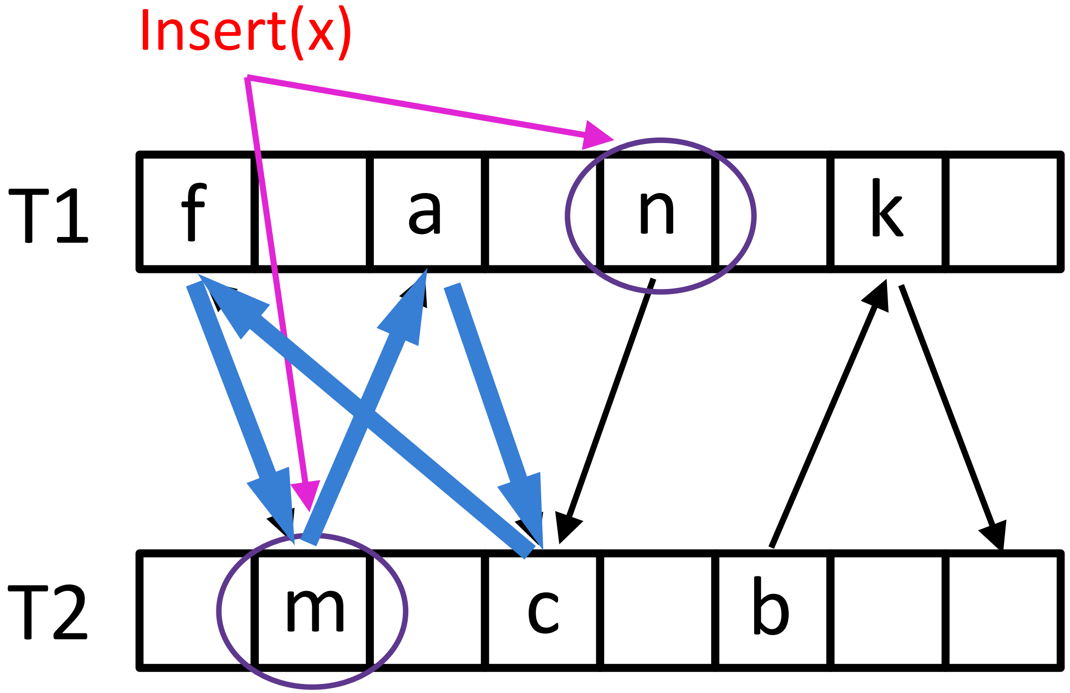
Cuckoo Hashing

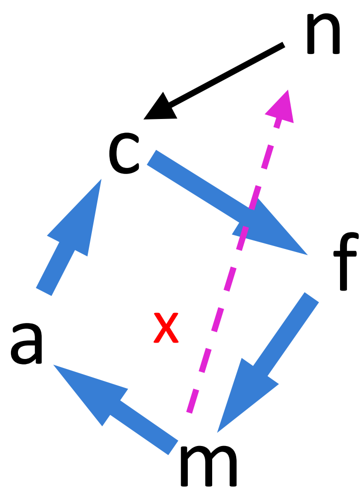
- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity
- For writes, endless loops may occur! => slow-write performance



Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity
- For writes, endless loops may occur! => slow-write performance

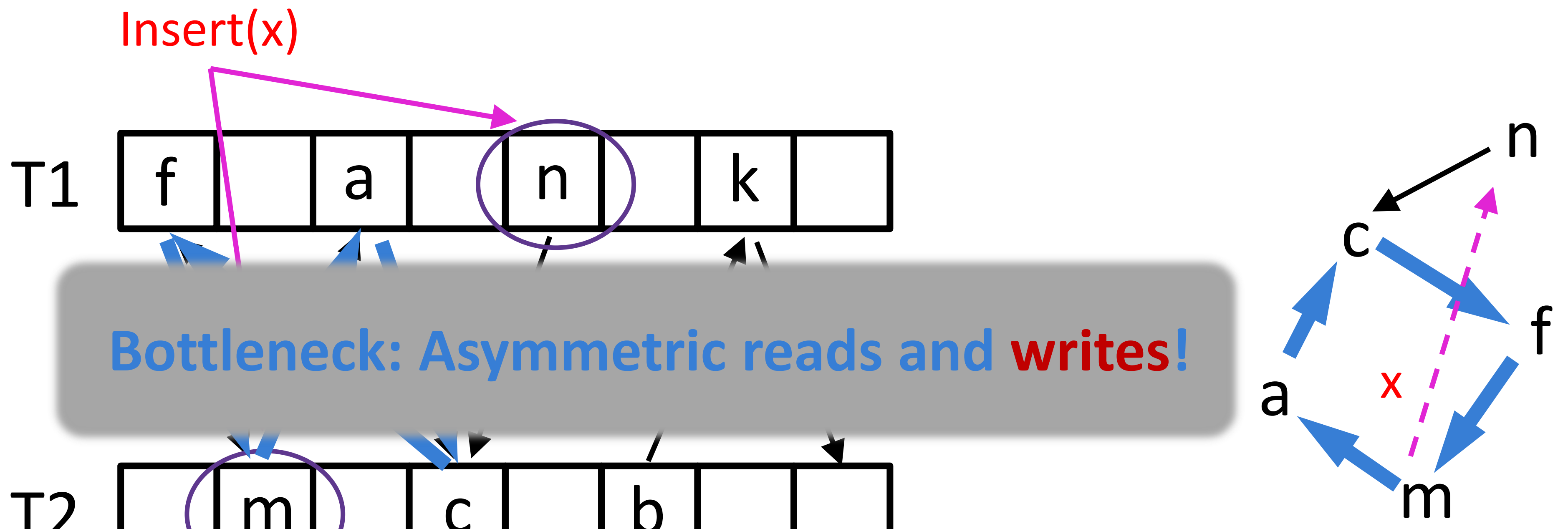




An endless loop occurs!

Cuckoo Hashing

- Multi-choice hashing
- Handling hash collisions: kick-out operations
- For reads, only limited positions are probed => O(1) time complexity
- For writes, endless loops may occur! => slow-write performance



An endless loop occurs!

Concurrency in Multi-core Systems

- **Existing concurrency strategy for cuckoo hashing**
  - **Lock two buckets before each kick-out operation (libcuckoo@EuroSys'14)**

Concurrency in Multi-core Systems

- **Existing concurrency strategy for cuckoo hashing**
  - **Lock two buckets before each kick-out operation (libcuckoo@EuroSys'14)**

- **Challenges:**
  - **Inefficient insertion performance**
  - **Limited scalability**

Concurrency in Multi-core Systems

- **Existing concurrency strategy for cuckoo hashing**
  - **Lock two buckets before each kick-out operation (libcuckoo@EuroSys'14)**

- **Challenges:**
  - **Inefficient insertion performance**
  - **Limited scalability**

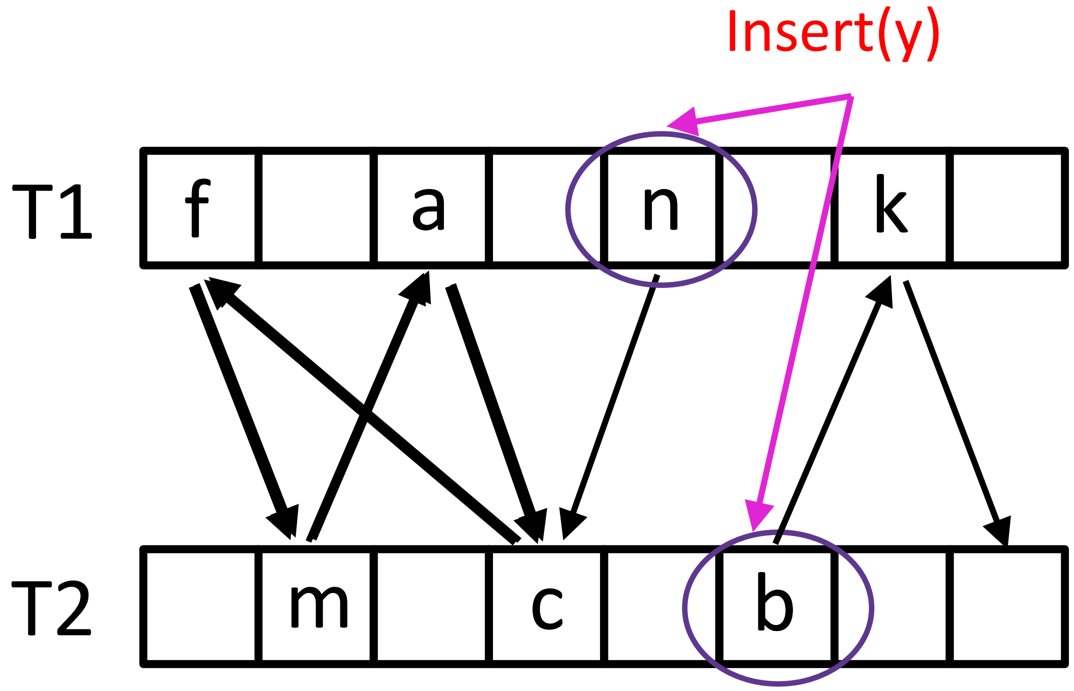
- **Design goal:**
  - **A high-throughput and concurrency-friendly cuckoo hash table**

Our Approach: CoCuckoo

- **Pseudoforests to predetermine endless loops**
- **Efficient concurrency strategy**
  - **A graph-grained locking mechanism**
  - **Concurrency optimization to reduce the length of critical path**
- **Higher throughput than state-of-the-art scheme, i.e., libcuckoo**

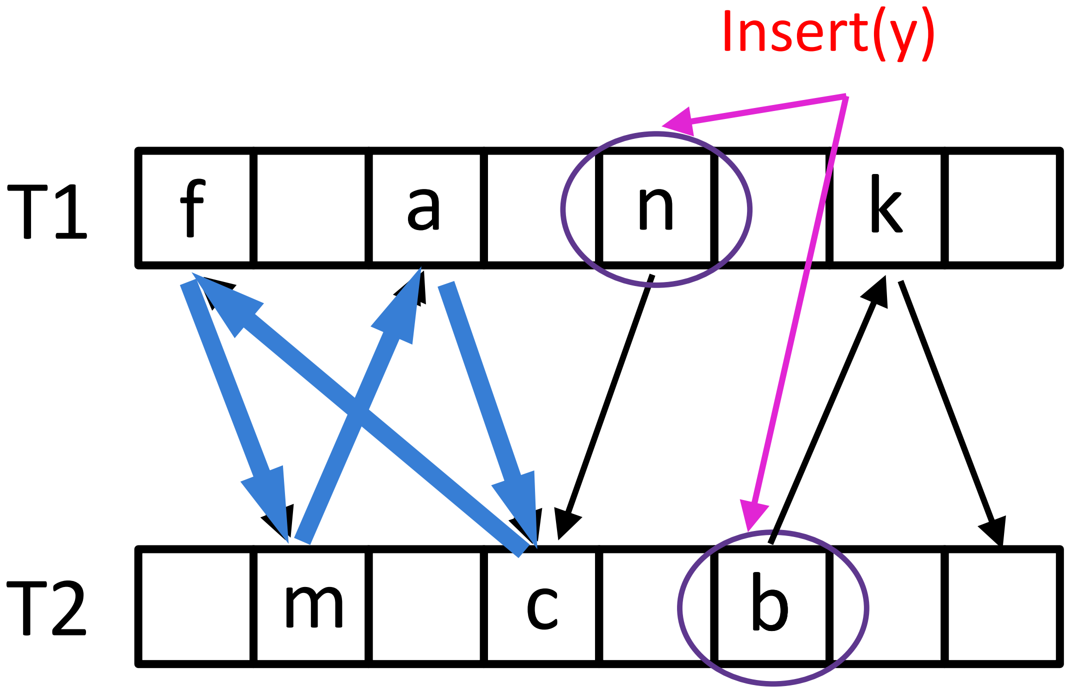
Pseudoforest

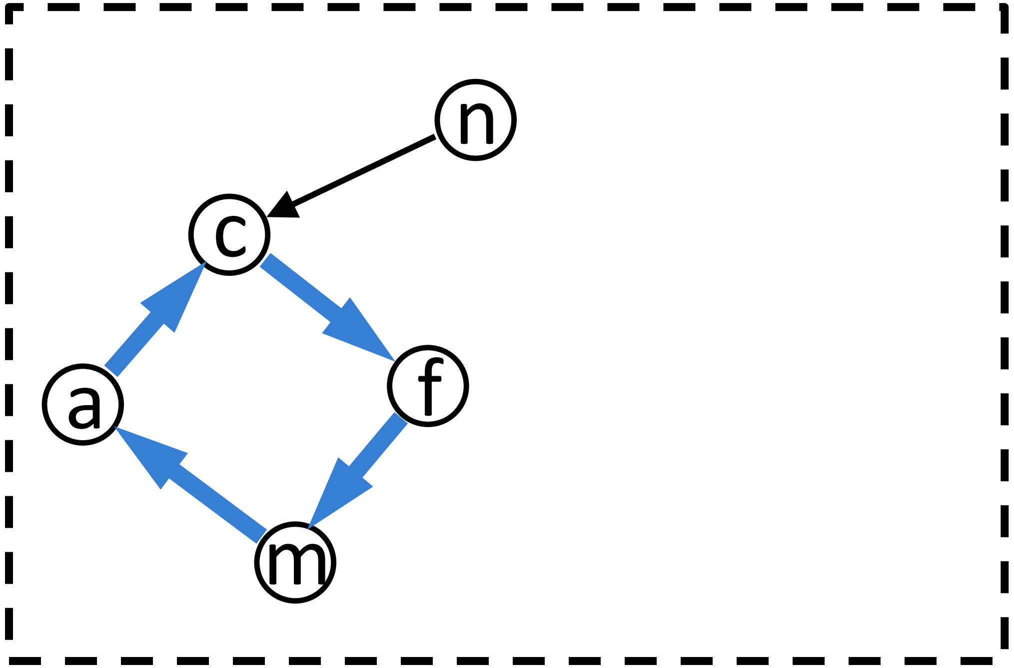
- **Vertex: a bucket**
- **Edge: an inserted item from the storage vertex to its backup vertex**
- **Identify endless loops: #Vertices = #Edges (called maximal)**



Pseudoforest

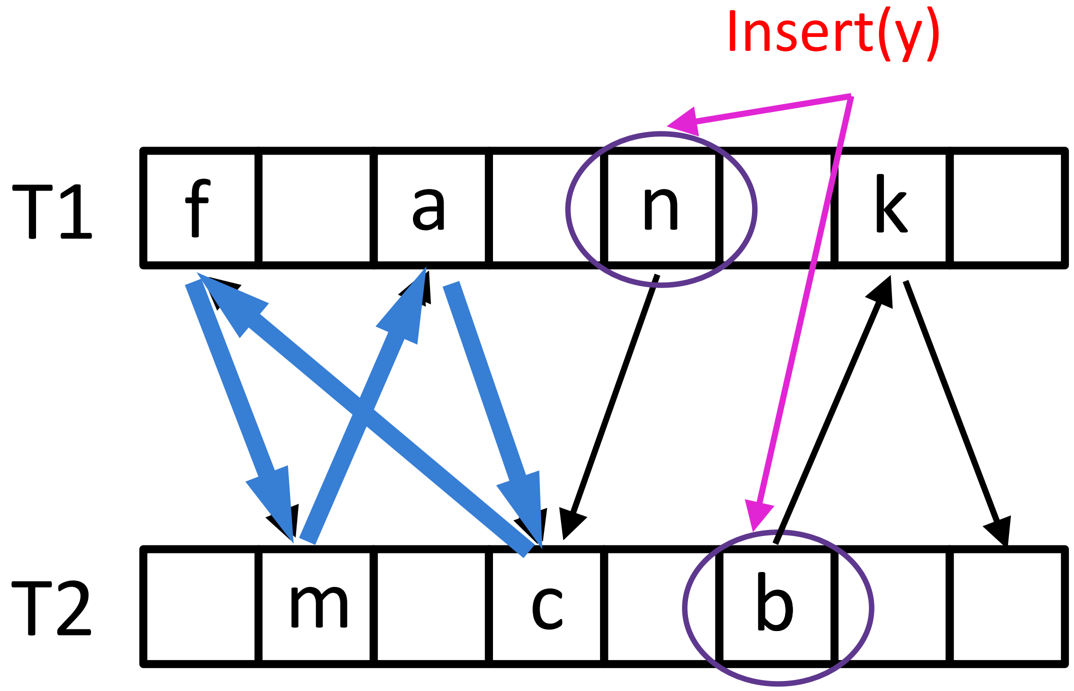
- **Vertex: a bucket**
- **Edge: an inserted item from the storage vertex to its backup vertex**
- **Identify endless loops: #Vertices = #Edges (called maximal)**

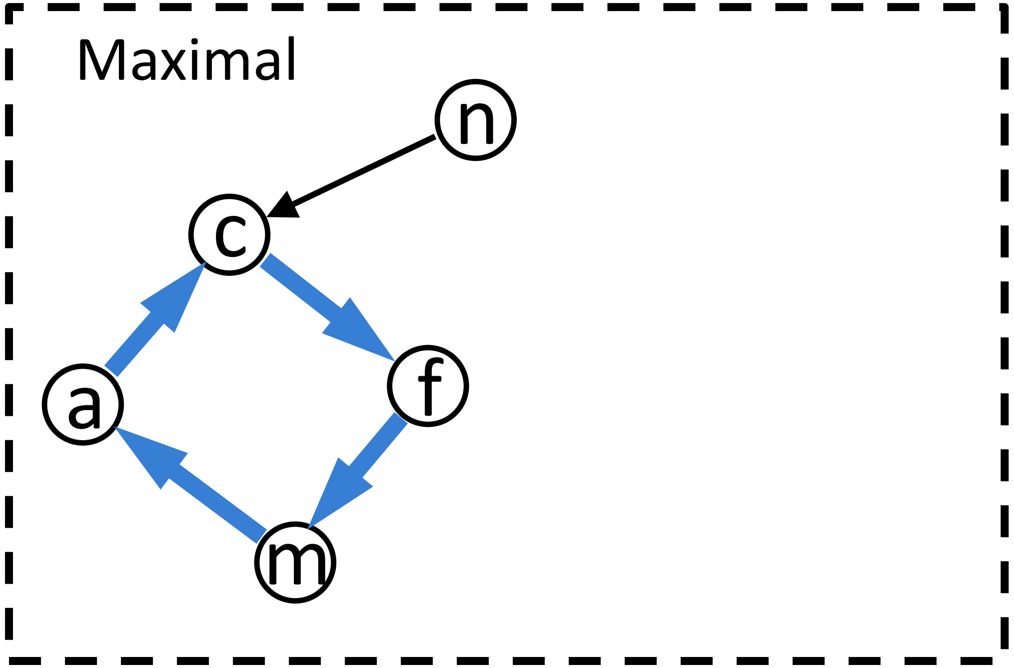




Pseudoforest

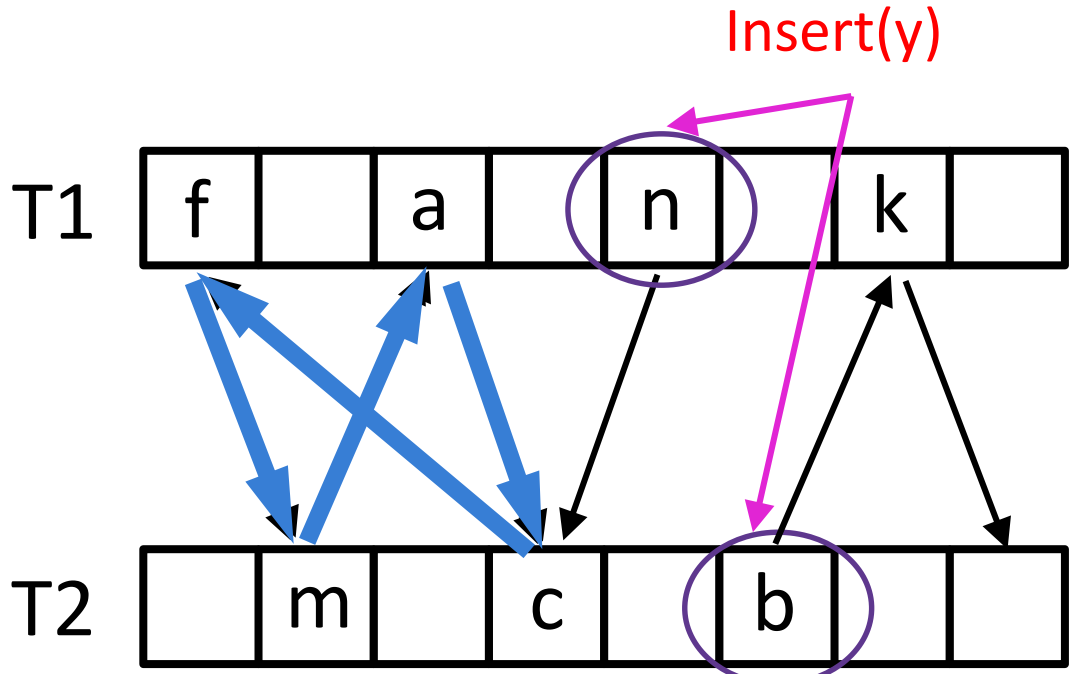
- Vertex: a bucket
- Edge: an inserted item from the storage vertex to its backup vertex
- Identify endless loops: #Vertices = #Edges (called **maximal**)

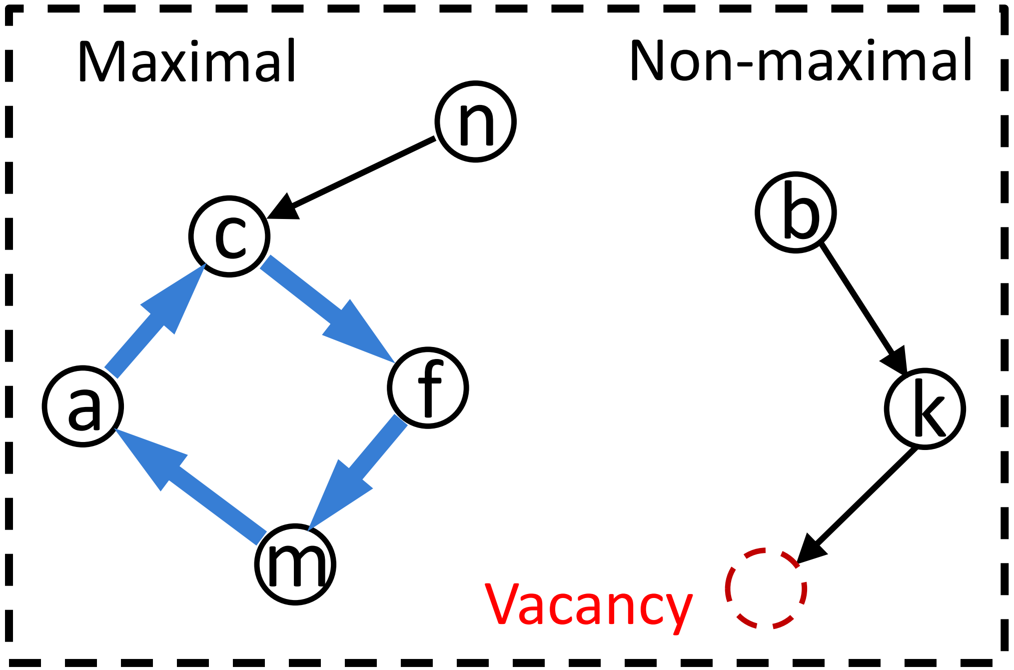




Pseudoforest

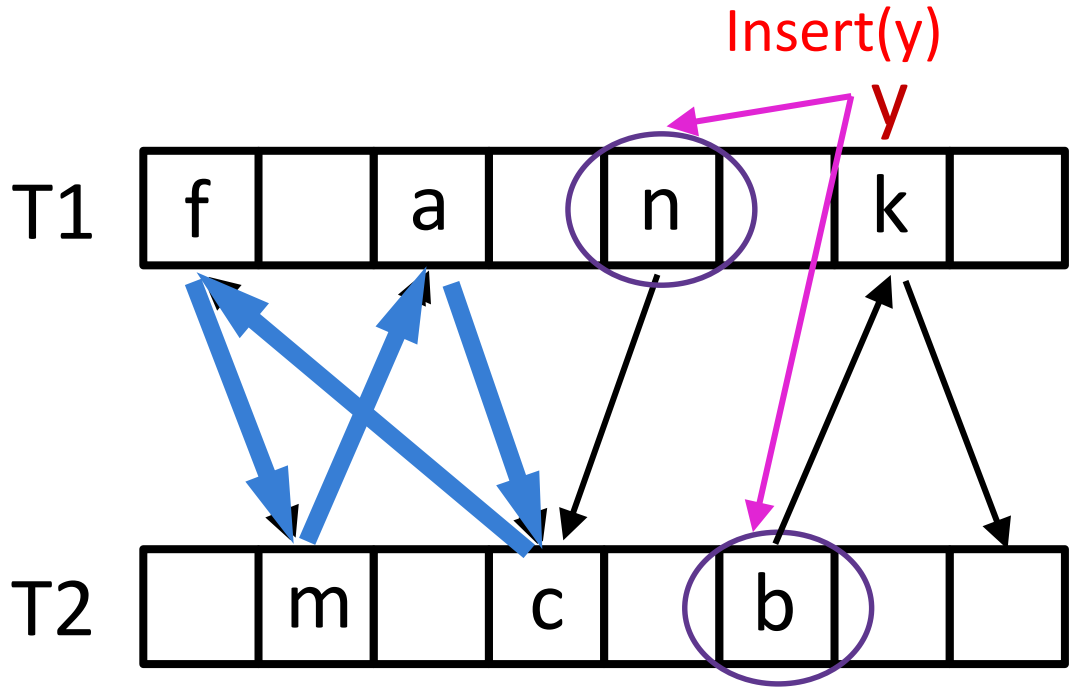
- **Vertex: a bucket**
- **Edge: an inserted item from the storage vertex to its backup vertex**
- **Identify endless loops: #Vertices = #Edges (called maximal)**

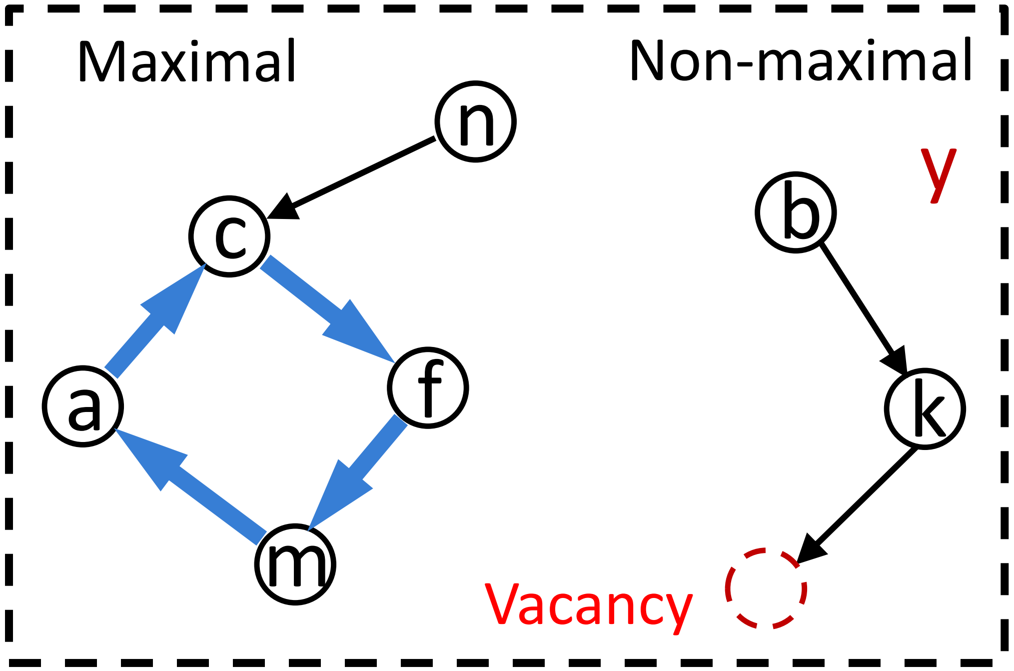




Pseudoforest

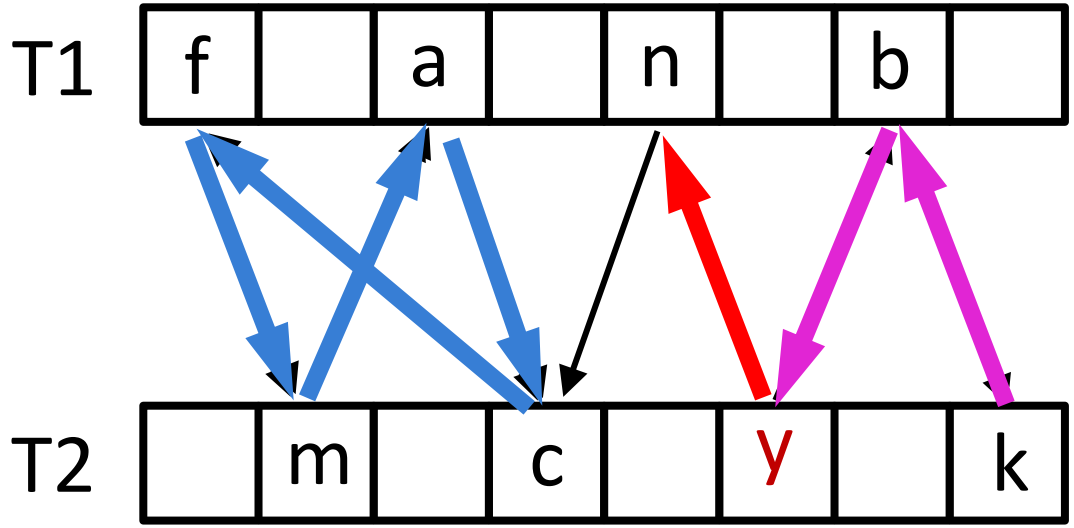
- Vertex: a bucket
- Edge: an inserted item from the storage vertex to its backup vertex
- Identify endless loops: #Vertices = #Edges (called **maximal**)

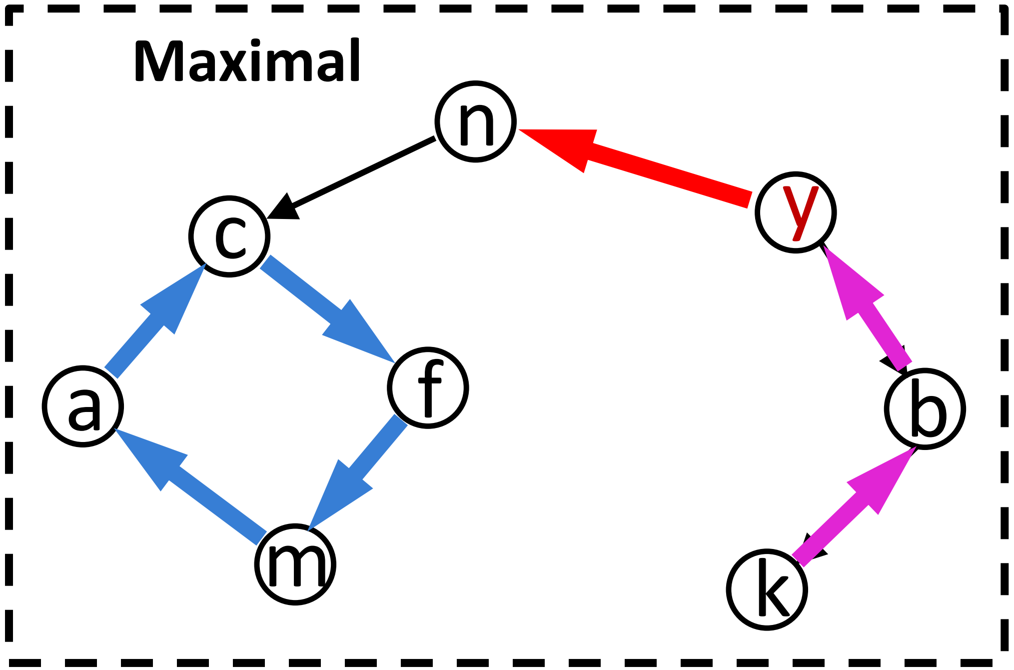




Pseudoforest

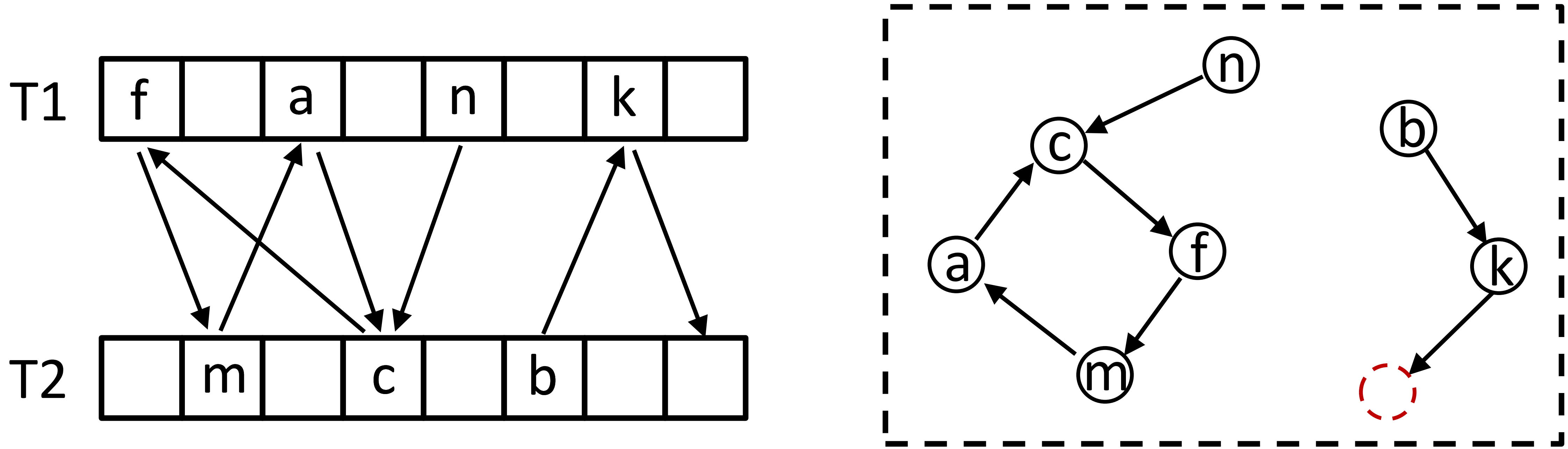
- **Vertex: a bucket**
- **Edge: an inserted item from the storage vertex to its backup vertex**
- **Identify endless loops: #Vertices = #Edges (called maximal)**





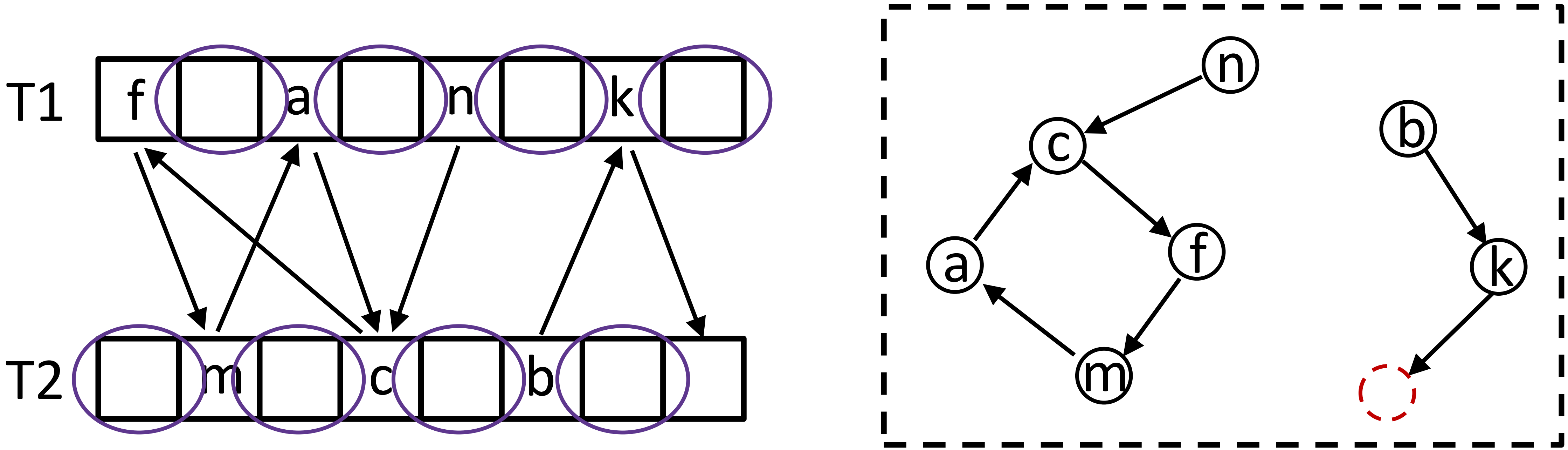
Graph-grained Locking

- **EMPTY subgraph**: buckets not represented in pseudoforest



Graph-grained Locking

- **EMPTY subgraph**: buckets not represented in pseudoforest



Graph-grained Locking

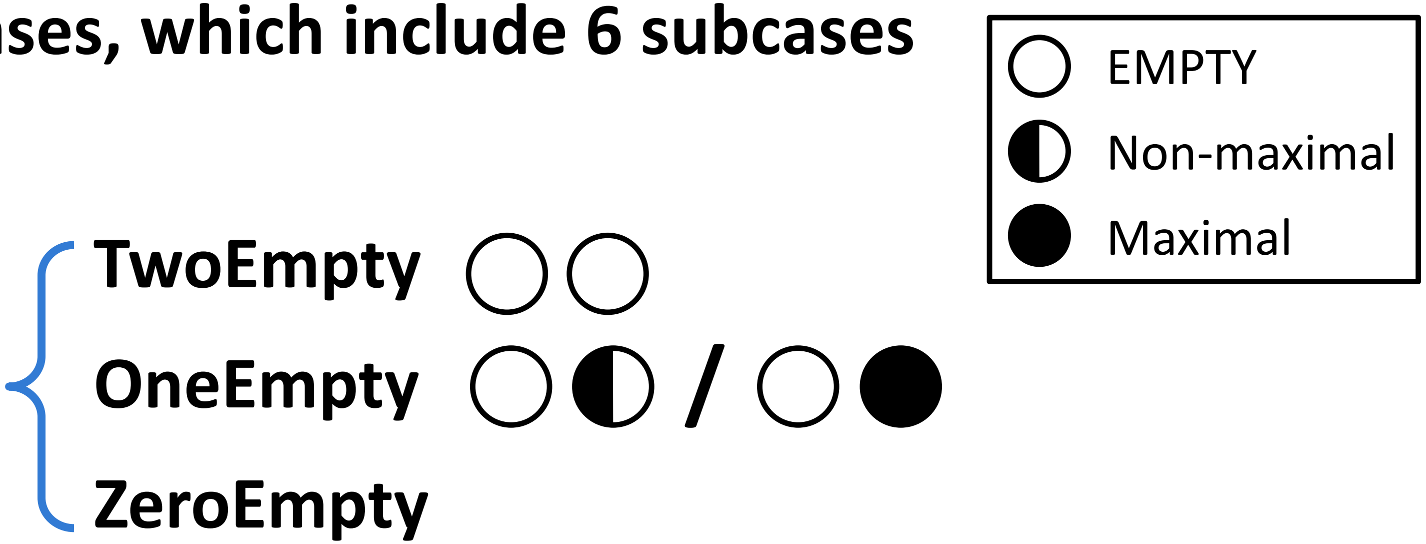
- **EMPTY subgraph: buckets not represented in pseudoforest**
- **Classify insertions into 3 cases, which include 6 subcases**

○ EMPTY
◐ Non-maximal
● Maximal

Graph-grained Locking

- **EMPTY subgraph: buckets not represented in pseudoforest**
- **Classify insertions into 3 cases, which include 6 subcases**

According to the number of corresponding EMPTY subgraphs



Graph-grained Locking

- **EMPTY subgraph: buckets not represented in pseudoforest**
- **Classify insertions into 3 cases, which include 6 subcases**

○	EMPTY
◐	Non-maximal
●	Maximal

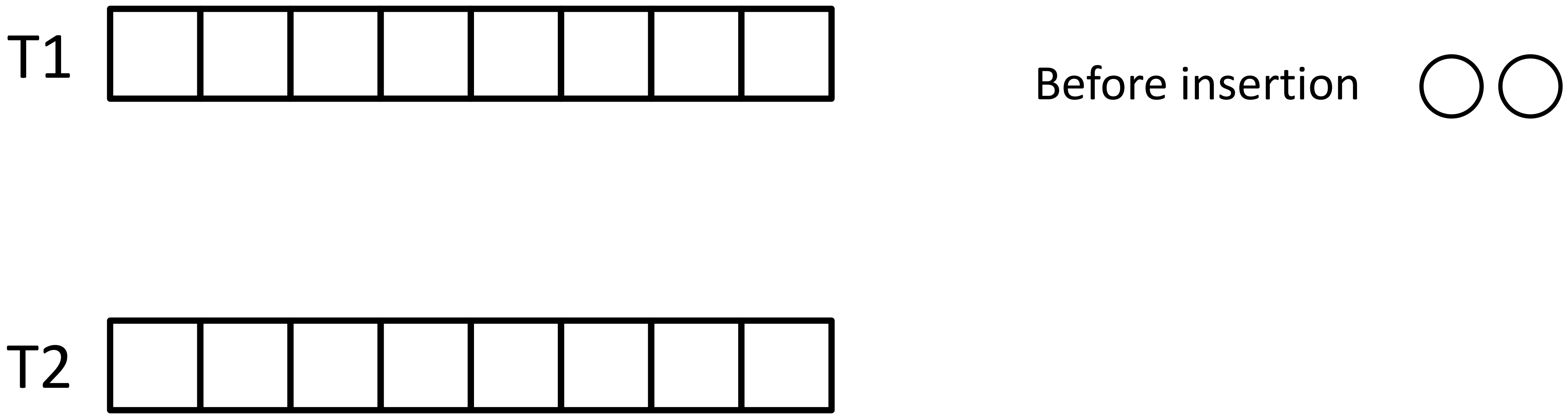
According to the number of corresponding EMPTY subgraphs

- **TwoEmpty** ○ ○
- **OneEmpty** ○ ◐ / ○ ●
- **ZeroEmpty**
  - **Diff_non_non** ◐ ◐
  - **Same_non** ◐
  - **Diff_non_max** ◐ ●
  - **Max** ● ● / ●

According to the states and the number of subgraphs

TwoEmpty

- **Two EMPTY subgraphs**

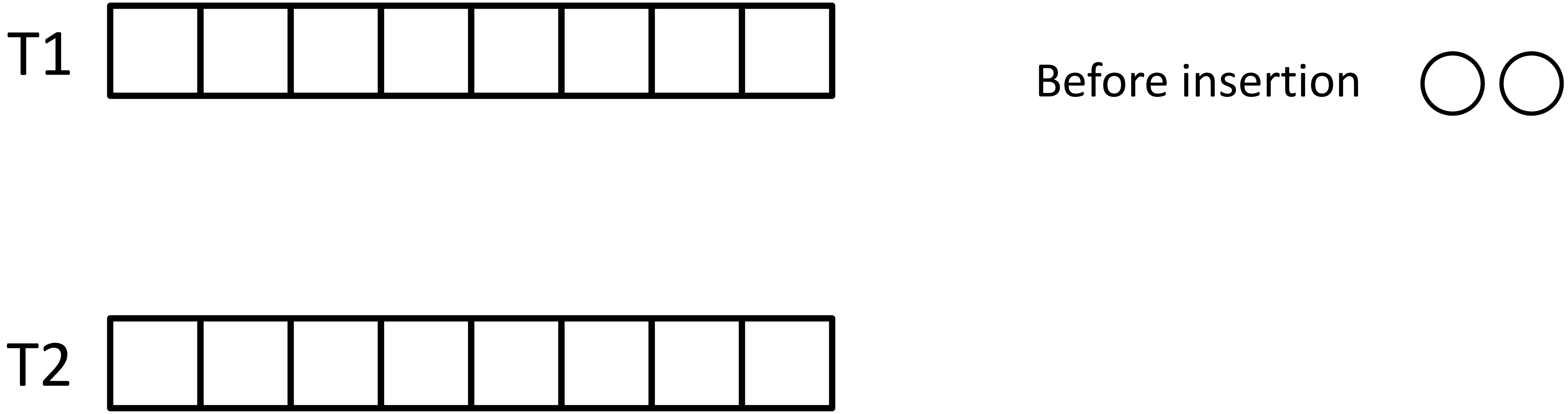


TwoEmpty

- **Two EMPTY subgraphs**
- **Insertion algorithm:**
  - **Atomically assign allocated subgraph number to two buckets**
  - **Insert item**
  - **Mark the subgraph as non-maximal**

critical path

With graph-grained lock(s)
Out of the critical path



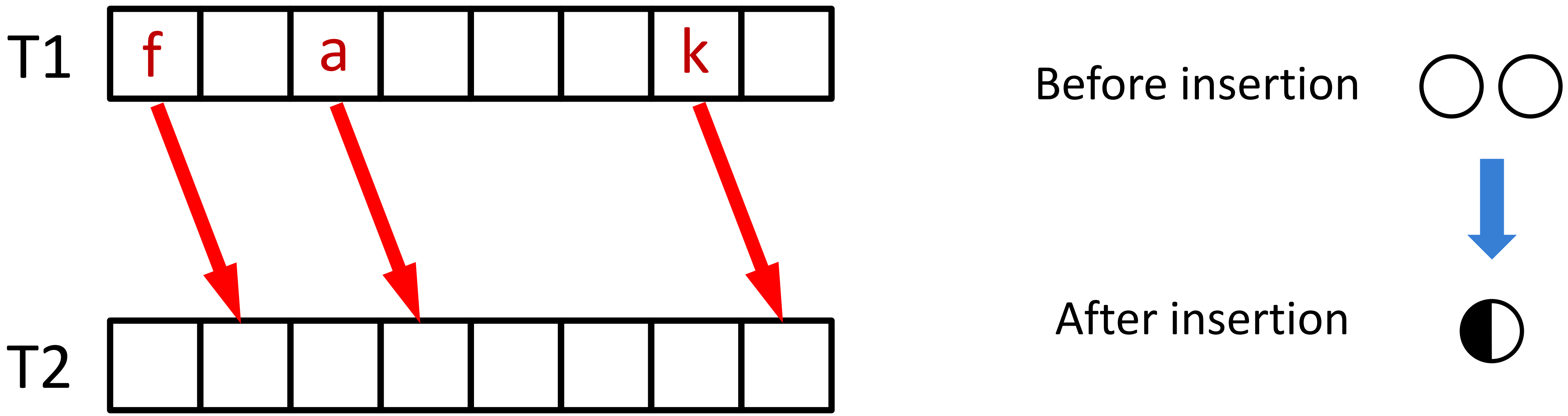
TwoEmpty

With graph-grained lock(s)
✓ Out of the critical path

- **Two EMPTY subgraphs**
- **Insertion algorithm:**

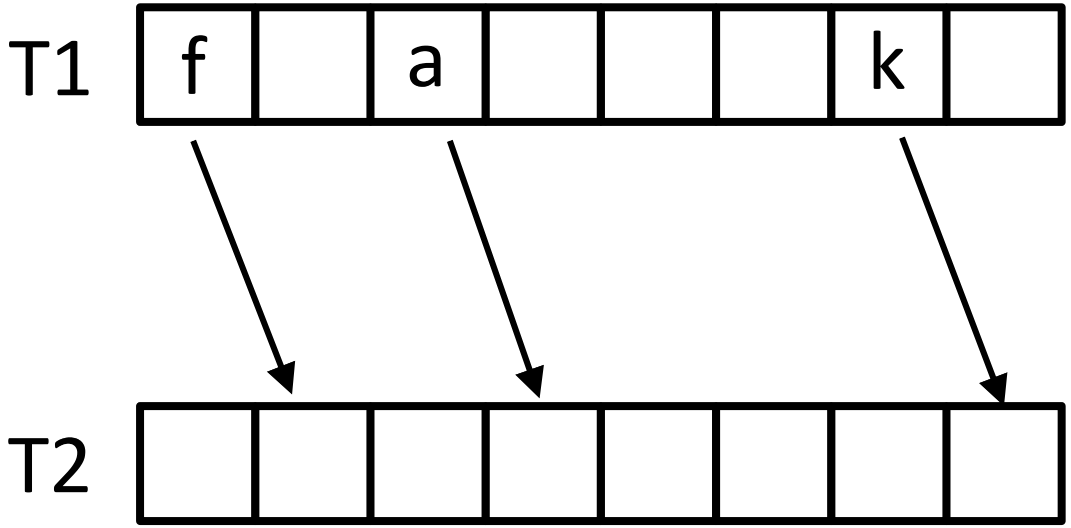
critical path

- **Atomically assign allocated subgraph number to two buckets**
- **Insert item**
- **Mark the subgraph as non-maximal**



OneEmpty ○◐ / ○●

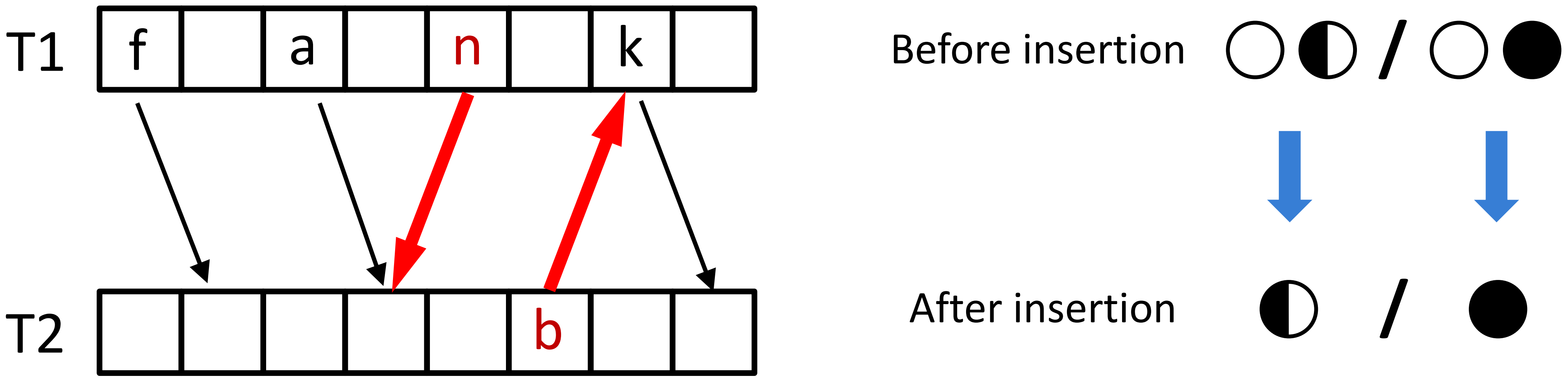
➢ **One EMPTY subgraph (the other is non-maximal/maximal)**



Before insertion ○◐ / ○●

OneEmpty ○◐ / ○●

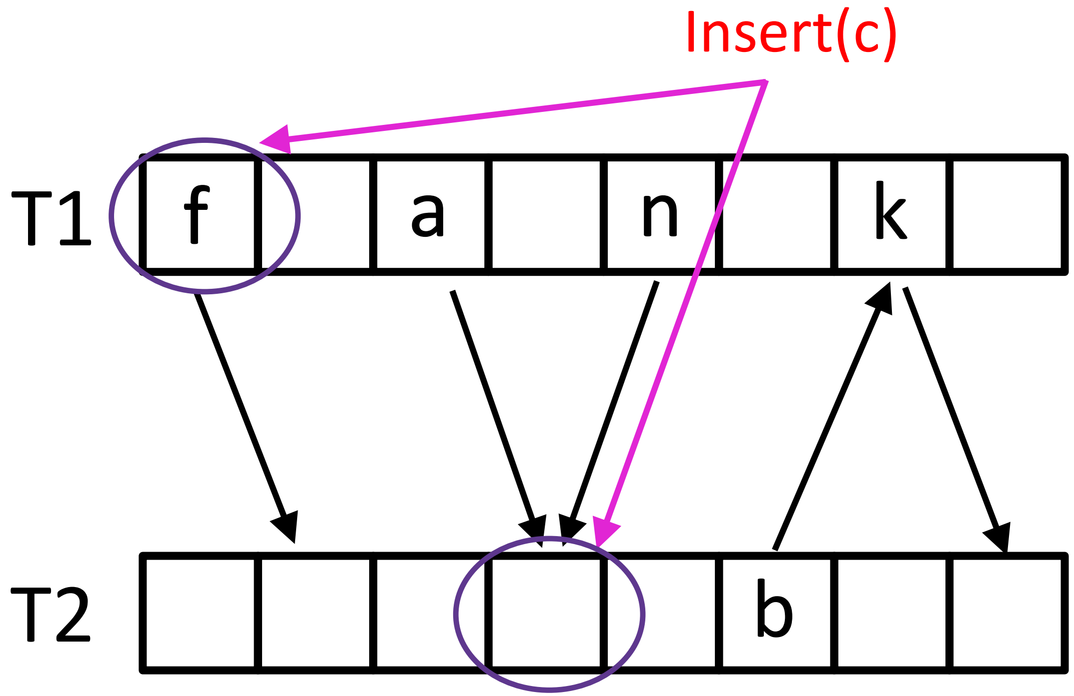
- **One EMPTY subgraph (the other is non-maximal/maximal)**
- **Insertion algorithm:**
  - ✓ **Two atomic operations without locks**
    - **Assign the existing subgraph number to the new vertex**
    - **Insert the item into the new vertex**



ZeroEmpty (Diff_non_non) ◐◐

- **Two different non-maximal subgraphs**
- **Insertion algorithm:**
  - **Kick-out (with item insertion)**
  - **Merge two subgraphs**

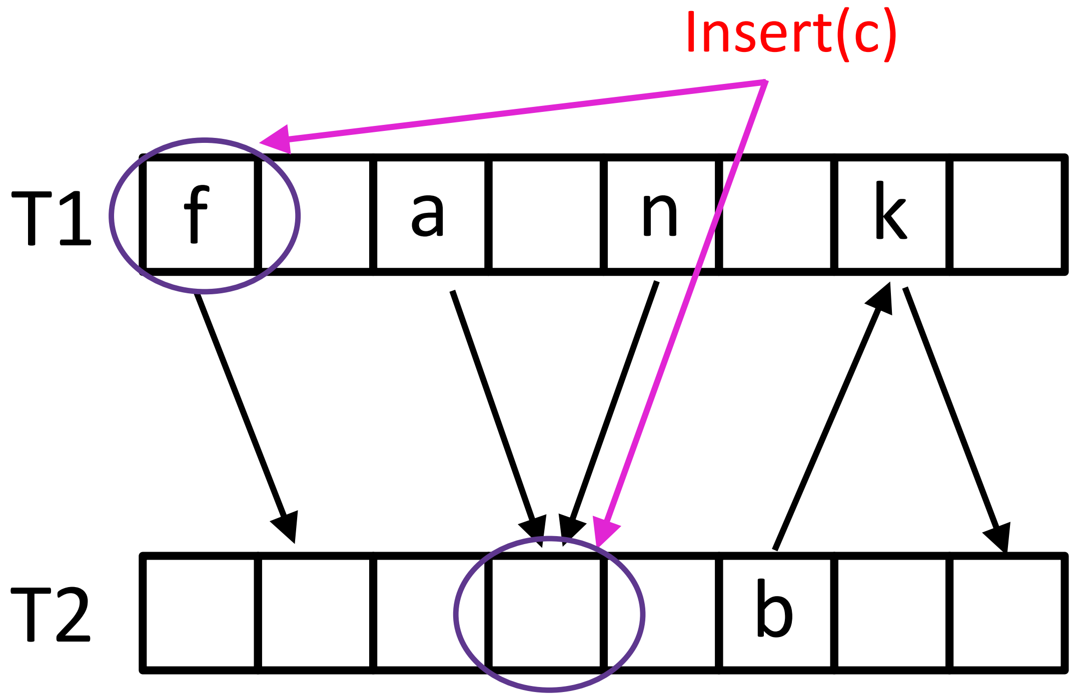
Before insertion ◐◐

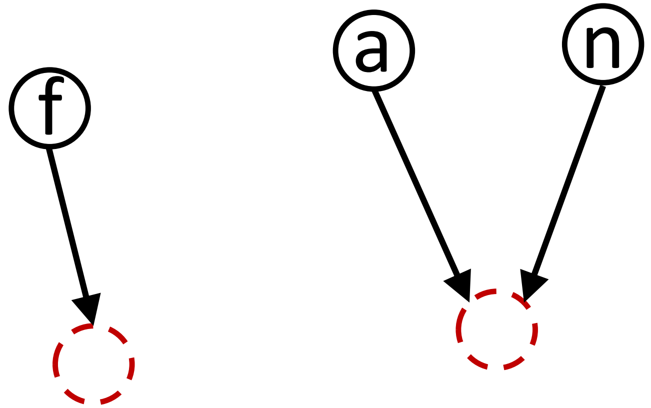


ZeroEmpty (Diff_non_non) ◐◐

- **Two different non-maximal subgraphs**
- **Insertion algorithm:**
  - **Kick-out (with item insertion)**
  - **Merge two subgraphs**

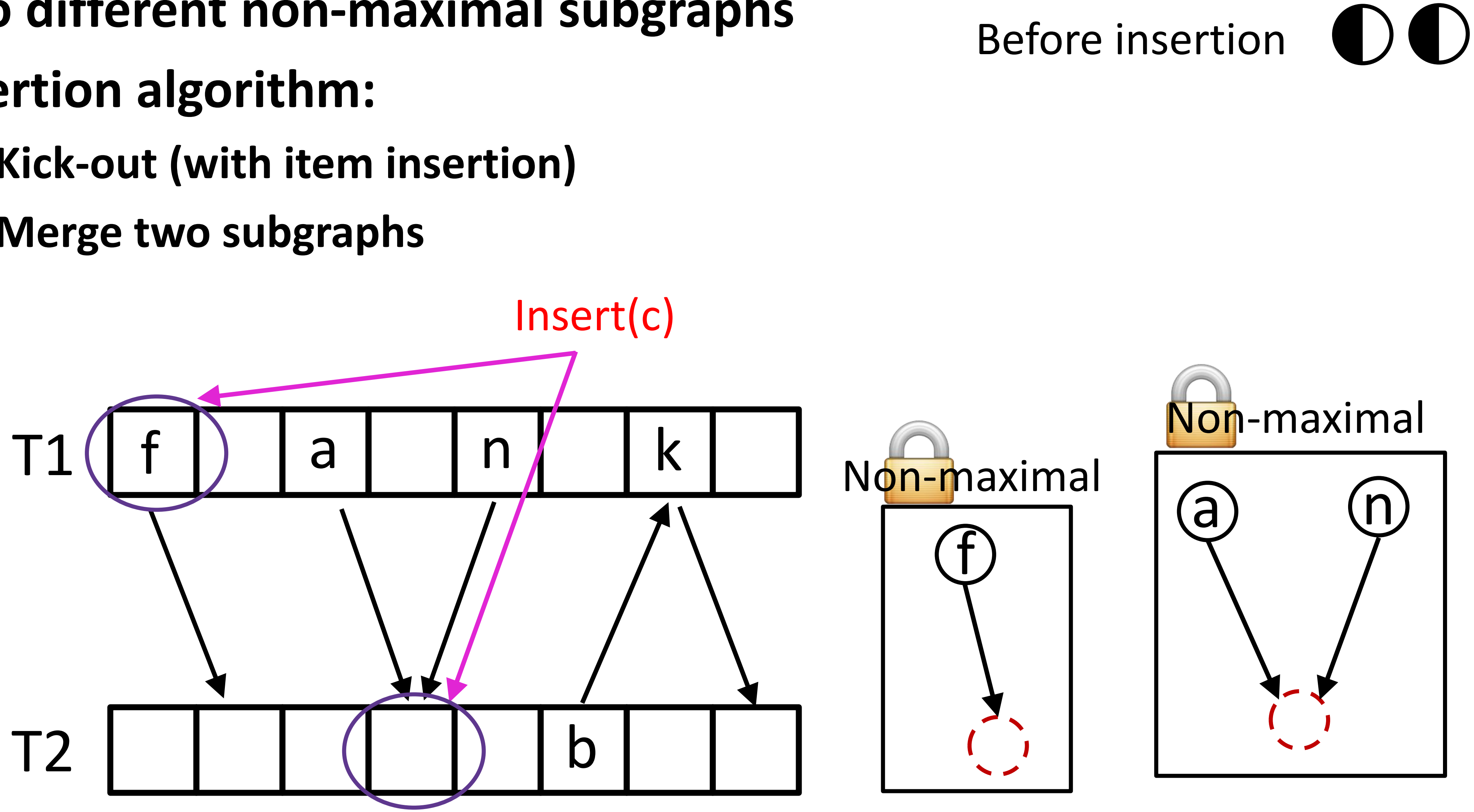
Before insertion ◐◐





ZeroEmpty (Diff_non_non) ◐◐

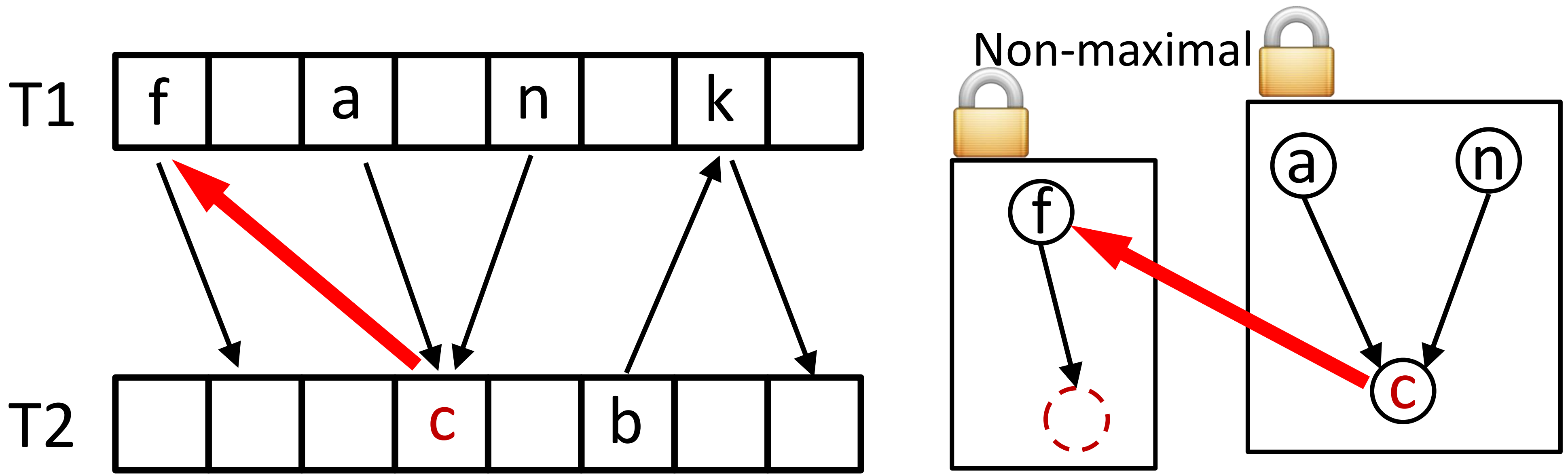
- **Two different non-maximal subgraphs**
- **Insertion algorithm:**
  - **Kick-out (with item insertion)**
  - **Merge two subgraphs**



ZeroEmpty (Diff_non_non) ◐◐

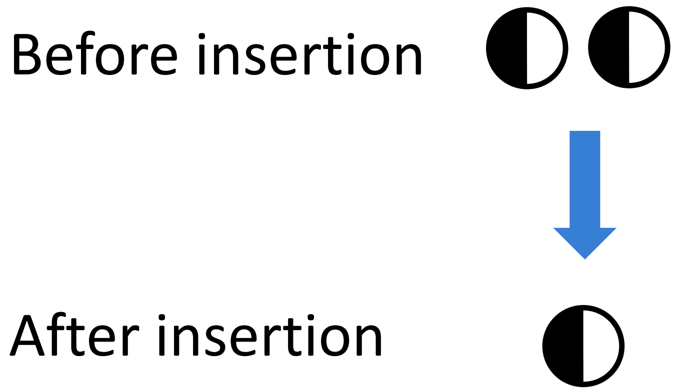
- **Two different non-maximal subgraphs**
- **Insertion algorithm:**
  - 🔒 **Kick-out (with item insertion)**
  - 🔒 **Merge two subgraphs**

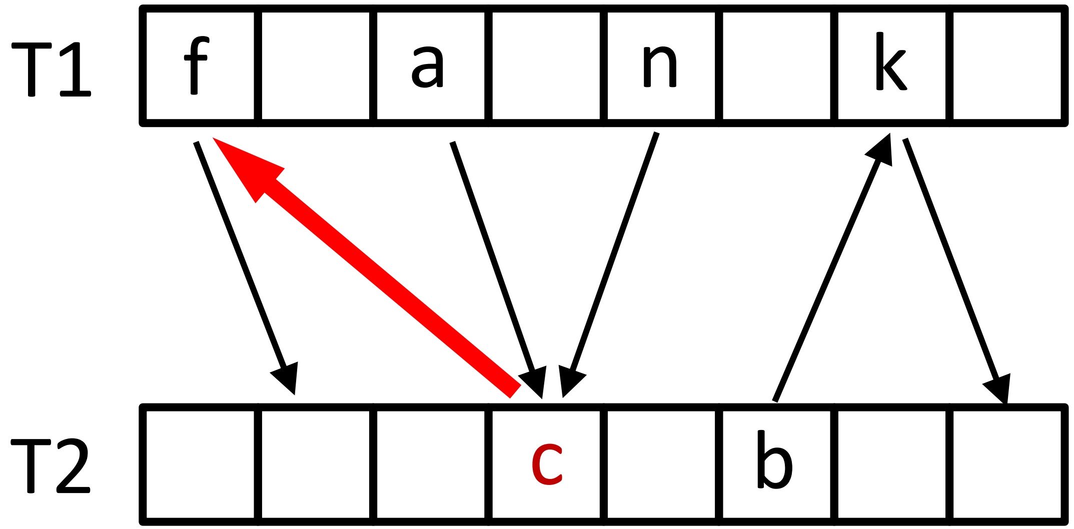
Before insertion ◐◐



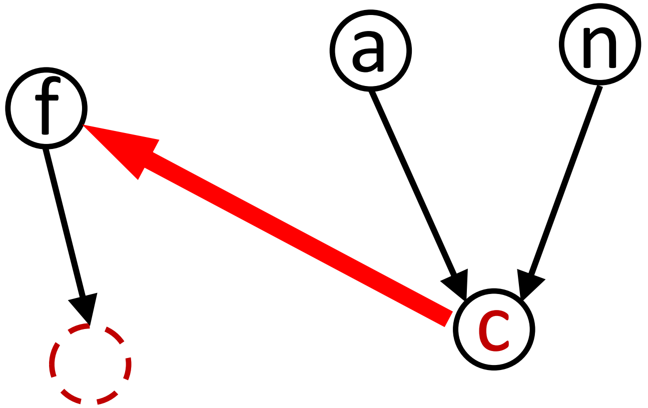
ZeroEmpty (Diff_non_non) ◐◐

- **Two different non-maximal subgraphs**
- **Insertion algorithm:**
  - **Kick-out (with item insertion)**
  - **Merge two subgraphs**





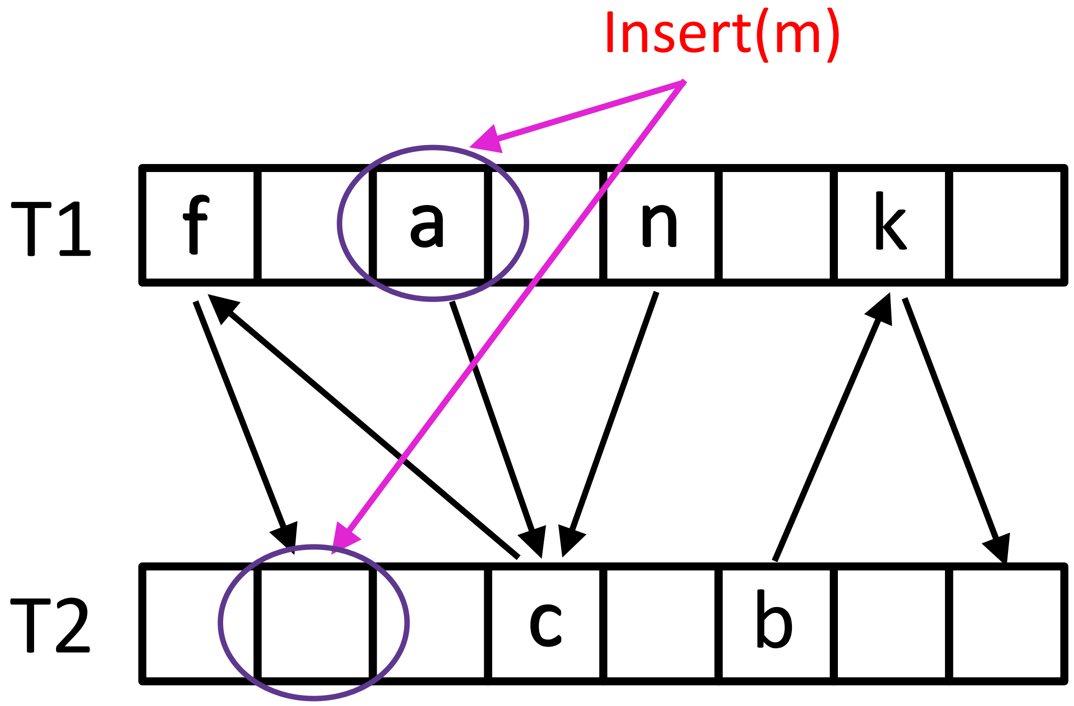
Non-maximal



ZeroEmpty (Same_non) ◐

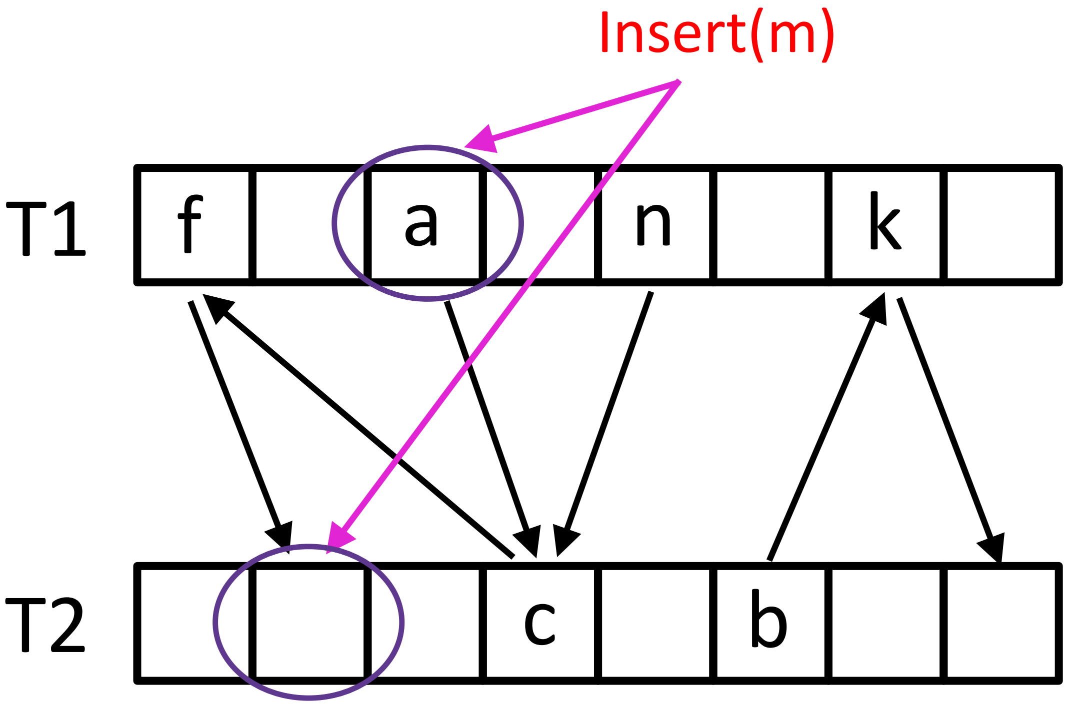
- The same non-maximal subgraph
- Insertion algorithm:
  - 🔒 Mark as maximal
  - ✓ Kick-out (with item insertion)

Before insertion ◐

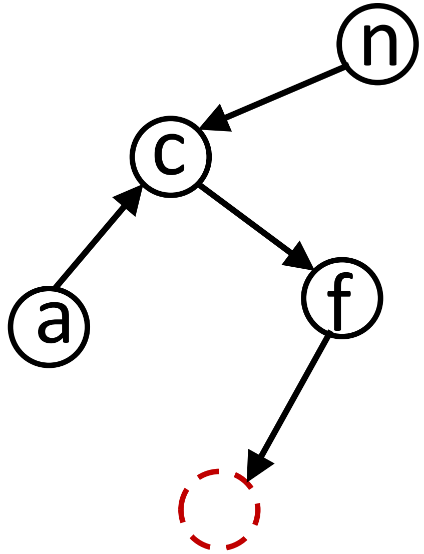


ZeroEmpty (Same_non) ◐

- ➢ **The same non-maximal subgraph**
- ➢ **Insertion algorithm:**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**

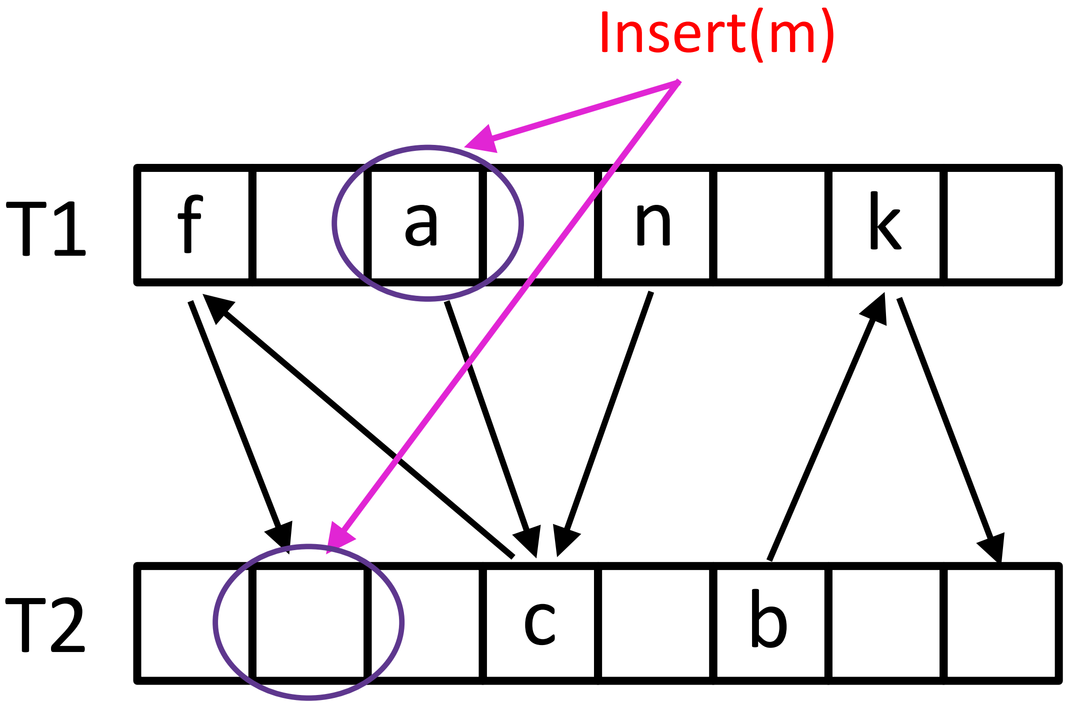


Before insertion ◐

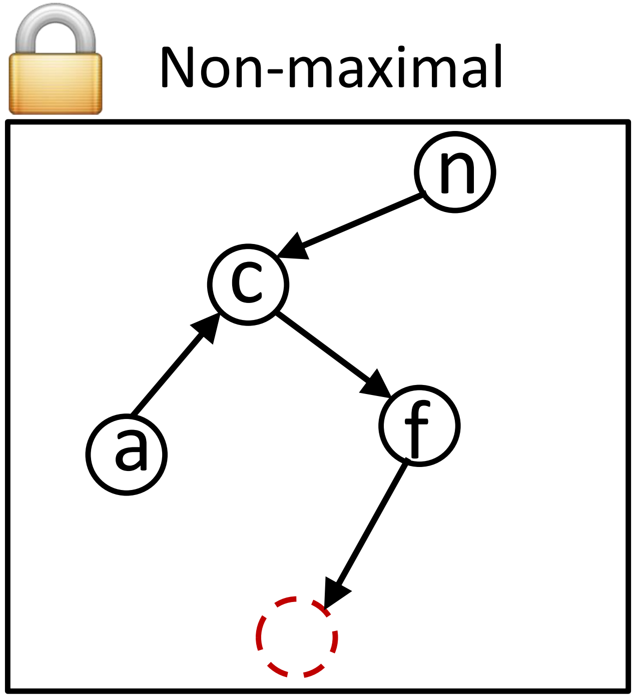


ZeroEmpty (Same_non) ◐

- **The same non-maximal subgraph**
- **Insertion algorithm:**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**



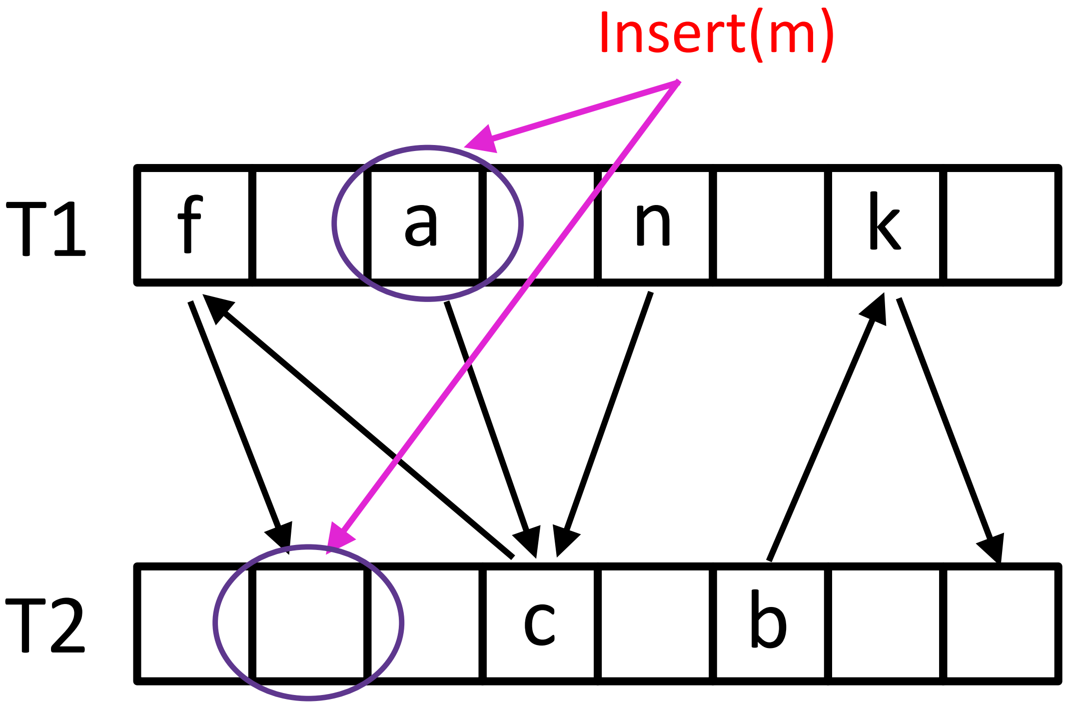
Before insertion ◐

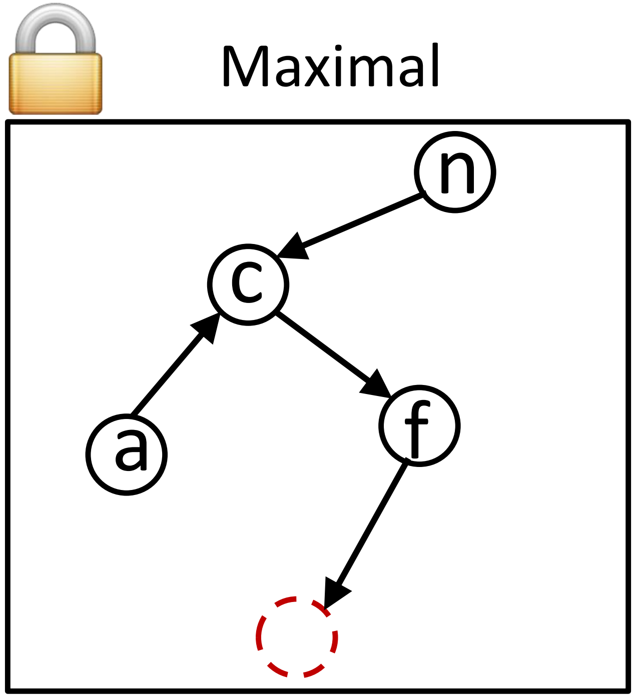


ZeroEmpty (Same_non) ◐

- ➢ **The same non-maximal subgraph**
- ➢ **Insertion algorithm:**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**

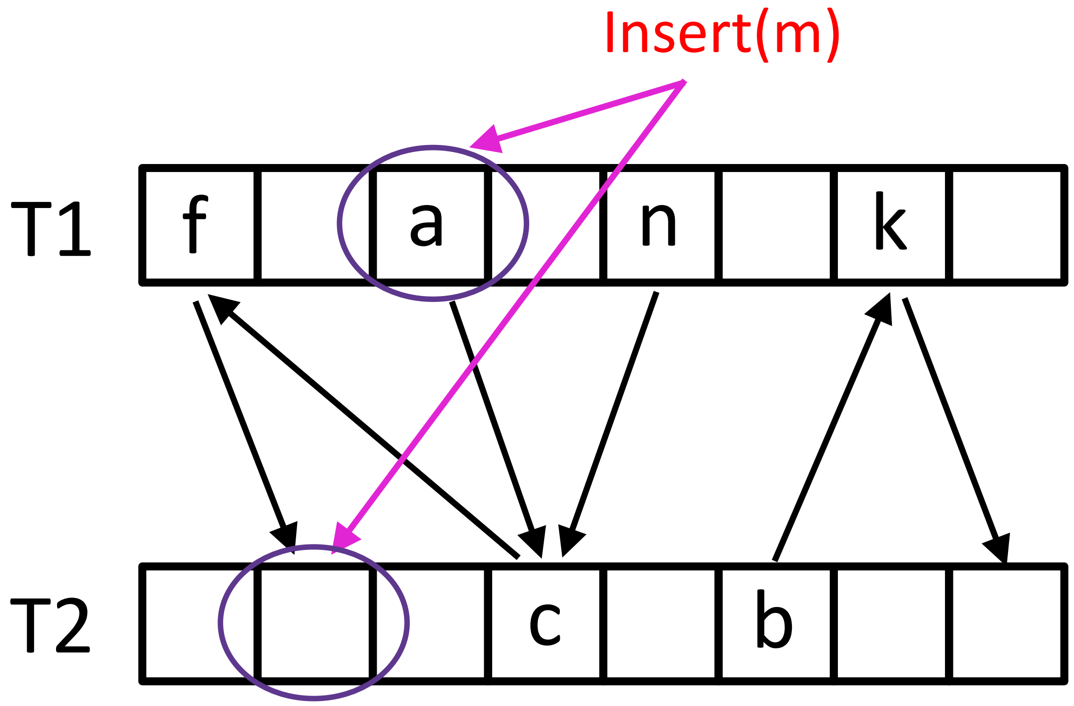
Before insertion ◐



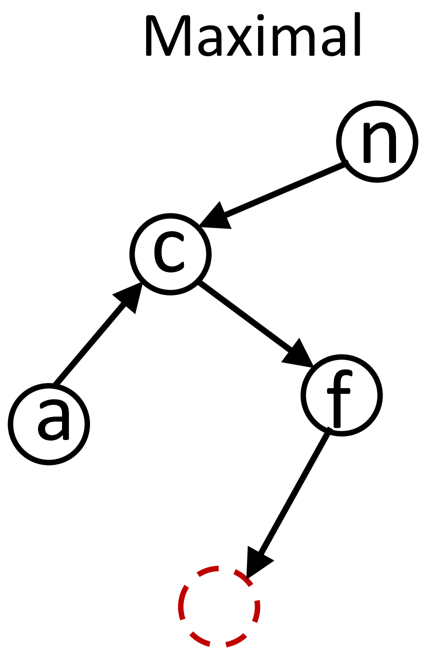


ZeroEmpty (Same_non) ◐

- **The same non-maximal subgraph**
- **Insertion algorithm:**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**



Before insertion ◐

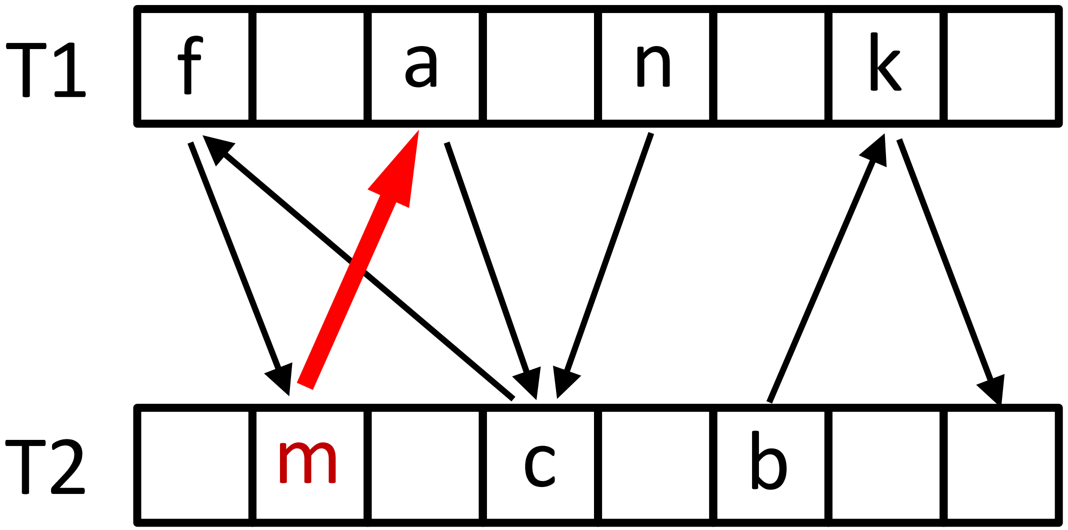


ZeroEmpty (Same_non) ◐

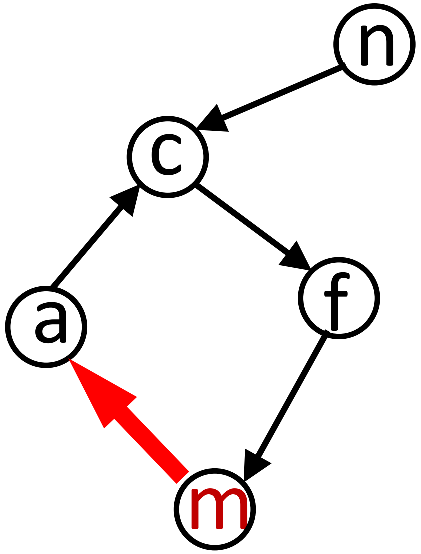
- **The same non-maximal subgraph**
- **Insertion algorithm:**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**

Before insertion ◐

After insertion ●

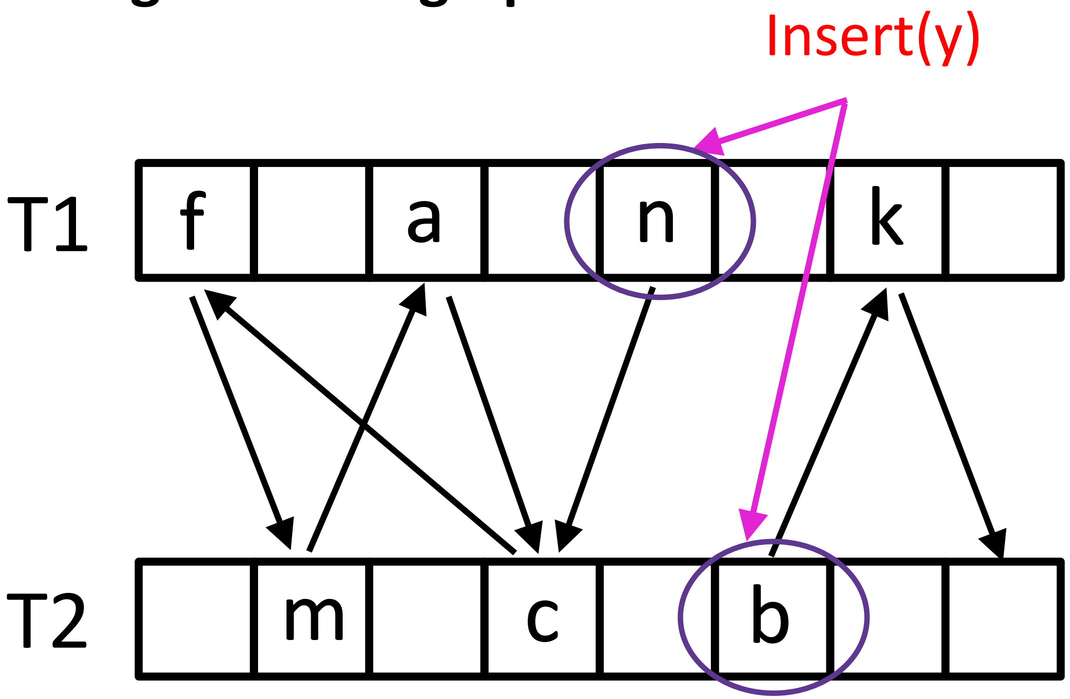


Maximal



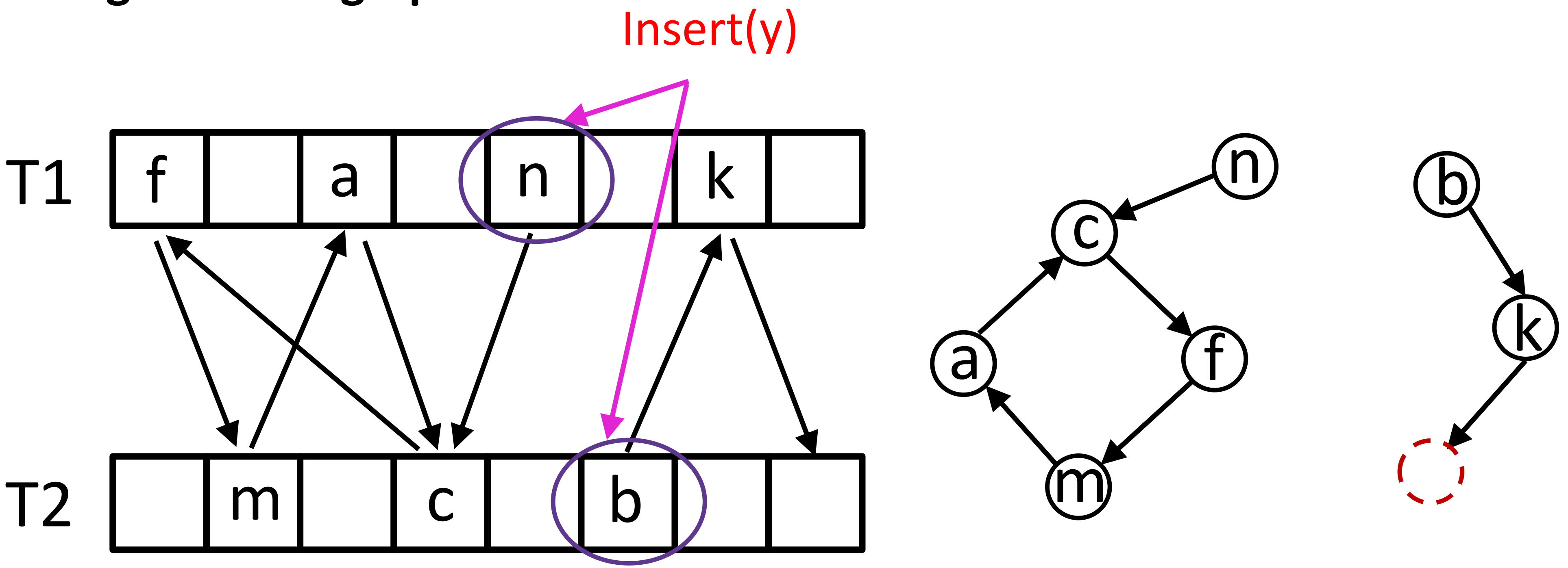
ZeroEmpty (Diff_non_max) ◐ ●

- ➢ **One non-maximal subgraph and one maximal subgraph** ◐ ●
- ➢ **Insertion algorithm (similar to same_non):**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**
  - ✓ **Merge two subgraphs**



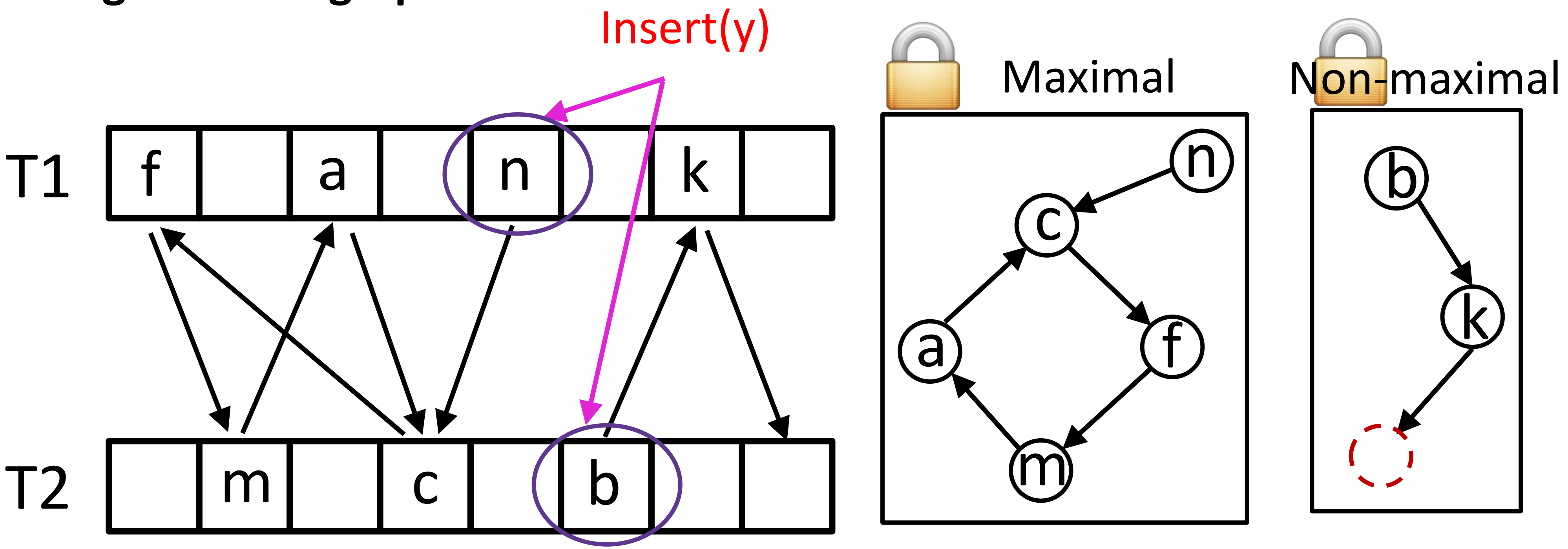
ZeroEmpty (Diff_non_max) ◐ ●

- One non-maximal subgraph and one maximal subgraph ◐ ●
- Insertion algorithm (similar to same_non):
  - 🔒 Mark as maximal
  - ✓ Kick-out (with item insertion)
  - ✓ Merge two subgraphs



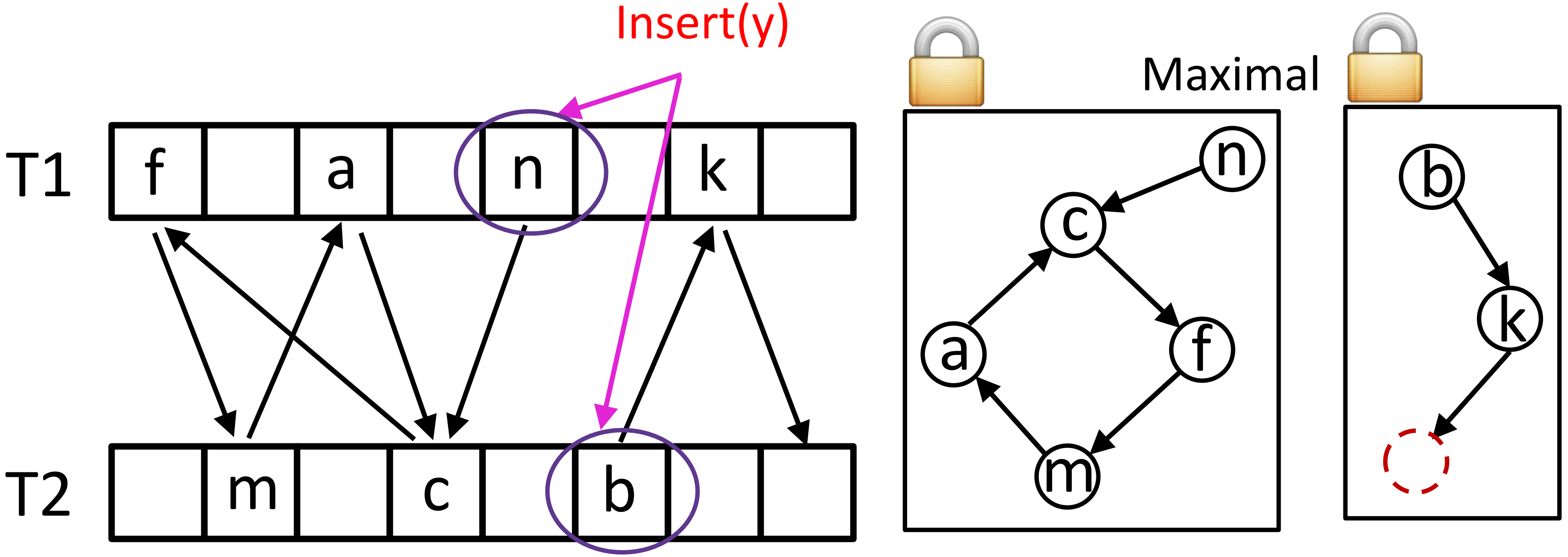
ZeroEmpty (Diff_non_max) ◐ ●

- One non-maximal subgraph and one maximal subgraph ◐ ●
- Insertion algorithm (similar to same_non):
  - 🔒 Mark as maximal
  - ✓ Kick-out (with item insertion)
  - ✓ Merge two subgraphs



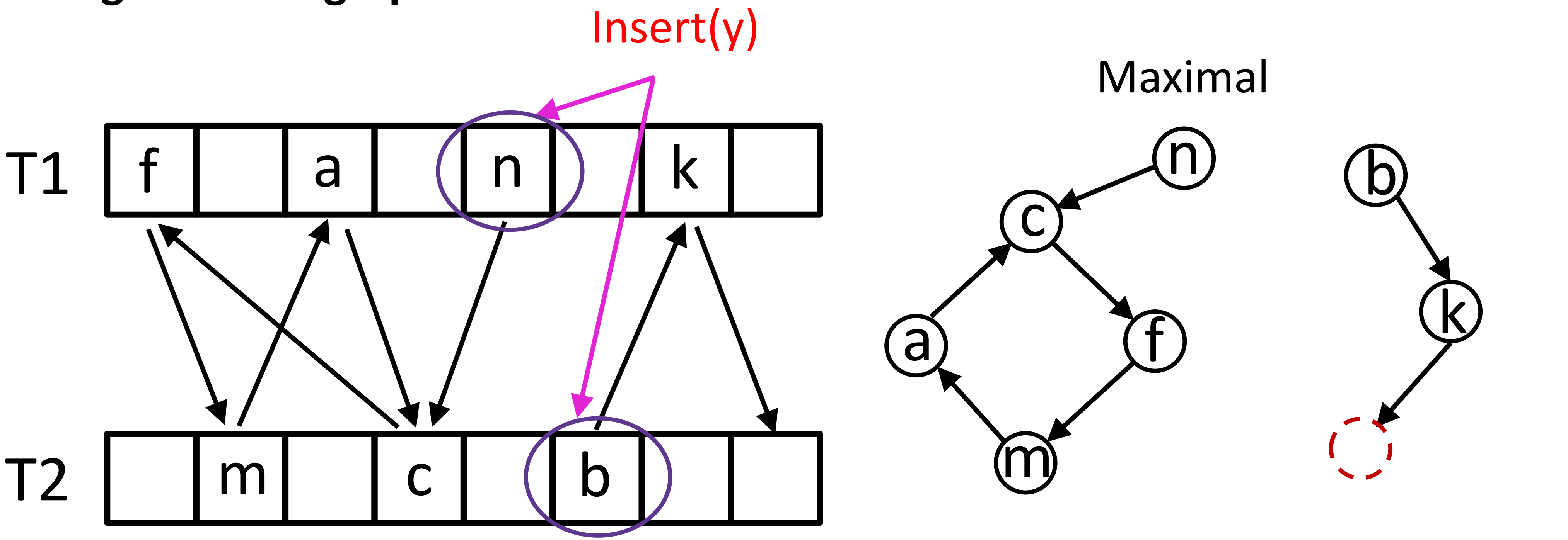
ZeroEmpty (Diff_non_max) ◐ ●

- One non-maximal subgraph and one maximal subgraph ◐ ●
- Insertion algorithm (similar to same_non):
  - 🔒 Mark as maximal
  - ✓ Kick-out (with item insertion)
  - ✓ Merge two subgraphs



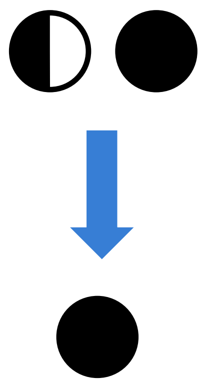
ZeroEmpty (Diff_non_max) ◐ ●

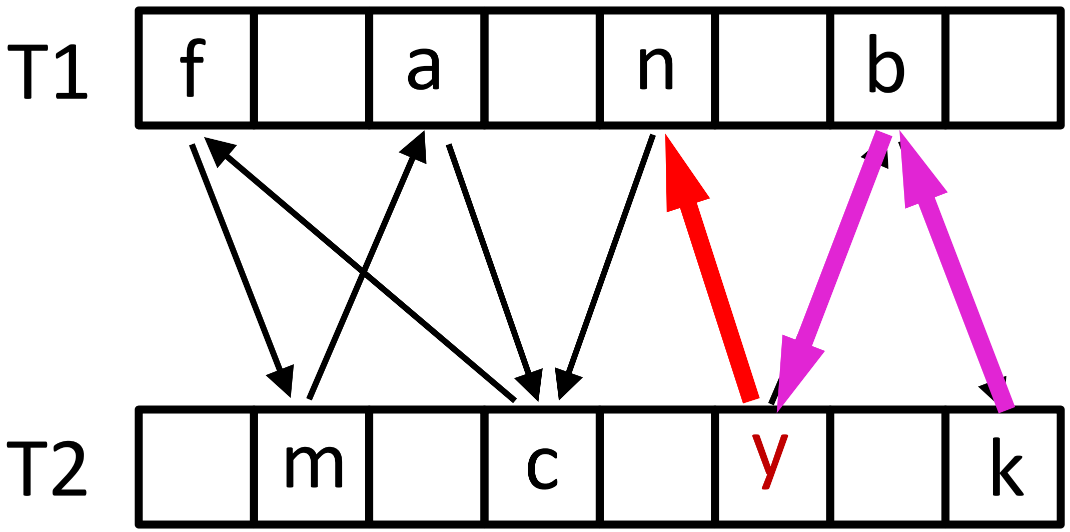
- **One non-maximal subgraph and one maximal subgraph** ◐ ●
- **Insertion algorithm (similar to same_non):**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**
  - ✓ **Merge two subgraphs**

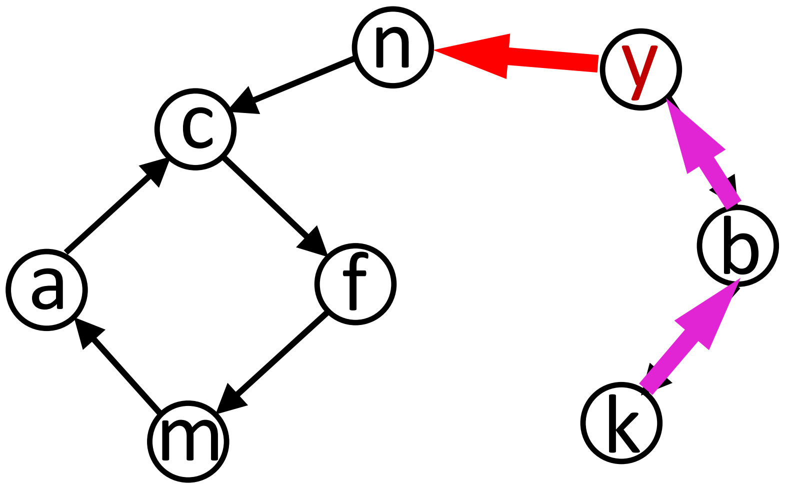


ZeroEmpty (Diff_non_max) ◐ ●

- **One non-maximal subgraph and one maximal subgraph**
- **Insertion algorithm (similar to same_non):**
  - 🔒 **Mark as maximal**
  - ✓ **Kick-out (with item insertion)**
  - ✓ **Merge two subgraphs**

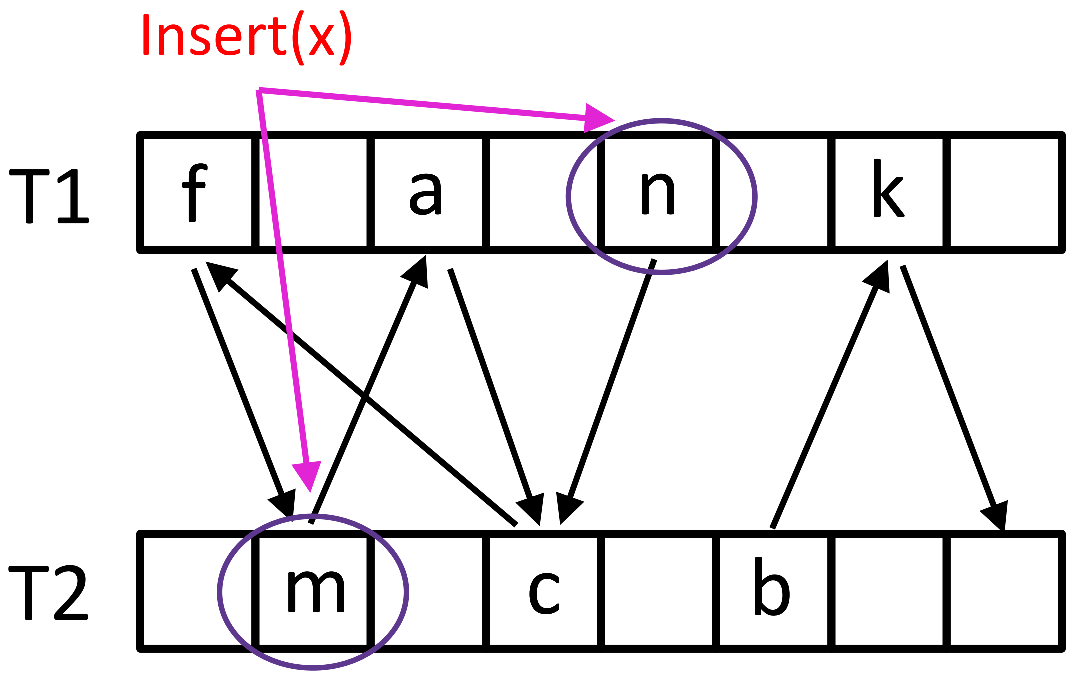






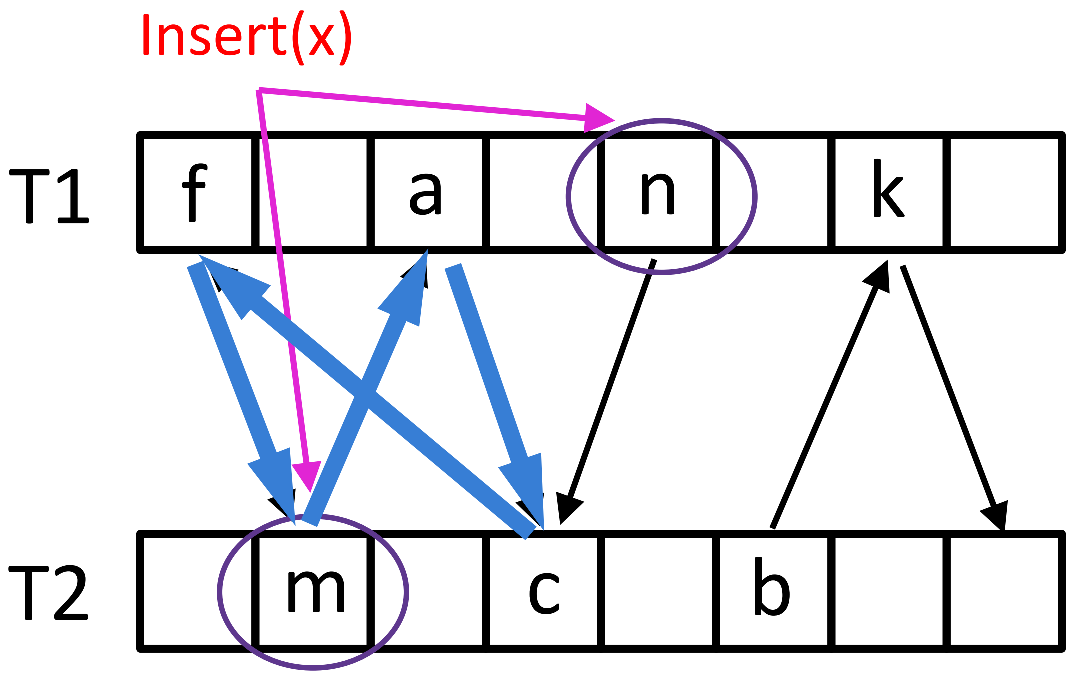
ZeroEmpty (Max) ●● / ●

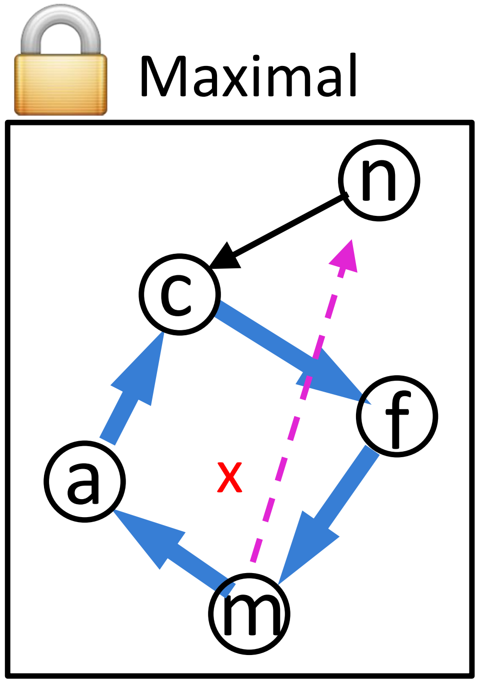
- **Two maximal subgraphs or the same maximal subgraph**
- **Always walking into a loop and predetermined to be a failure**
- **Insertion algorithm:**
  - ✓ **Do nothing**



ZeroEmpty (Max) ●●/●

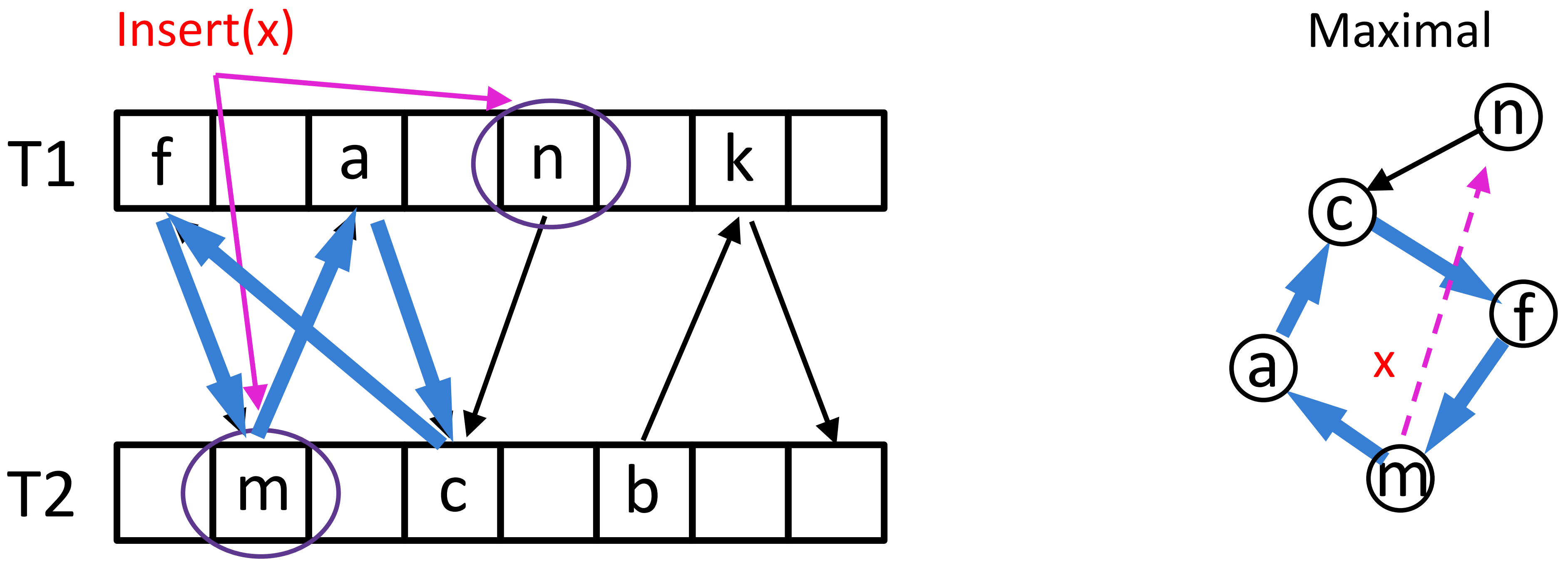
- **Two maximal subgraphs or the same maximal subgraph**
- **Always walking into a loop and predetermined to be a failure**
- **Insertion algorithm:**
  - ✓ **Do nothing**





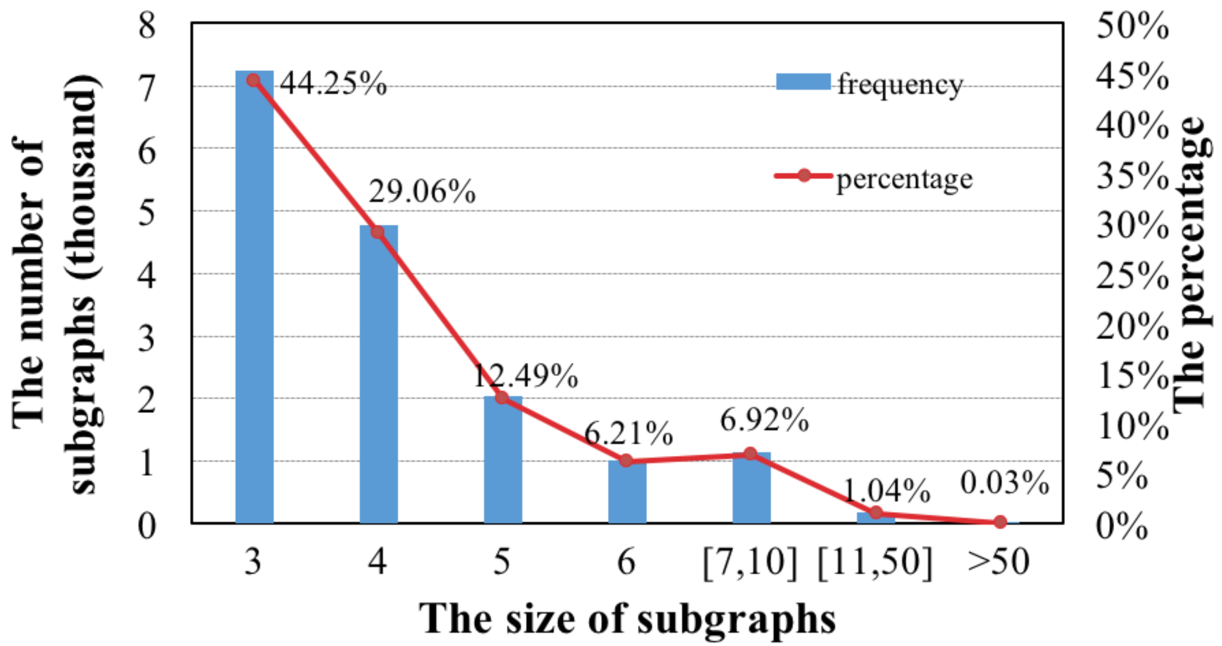
ZeroEmpty (Max) ●● / ●

- **Two maximal subgraphs or the same maximal subgraph**
- **Always walking into a loop and predetermined to be a failure**
- **Insertion algorithm:**
  - ✓ **Do nothing**



Lock Granularity

- **Most subgraphs are small ➡ the granularity of graph-grained locks is acceptable**
  - **Only constraining a very small number of buckets**
  - **3 vertices (44.25% subgraphs)**
  - **No more than 10 vertices (99% subgraphs)**



Subgraph Management

- **Subgraph number allocation**
  - **Subgraph number: identifying a unique subgraph**
  - **Unique without the need of continuity**

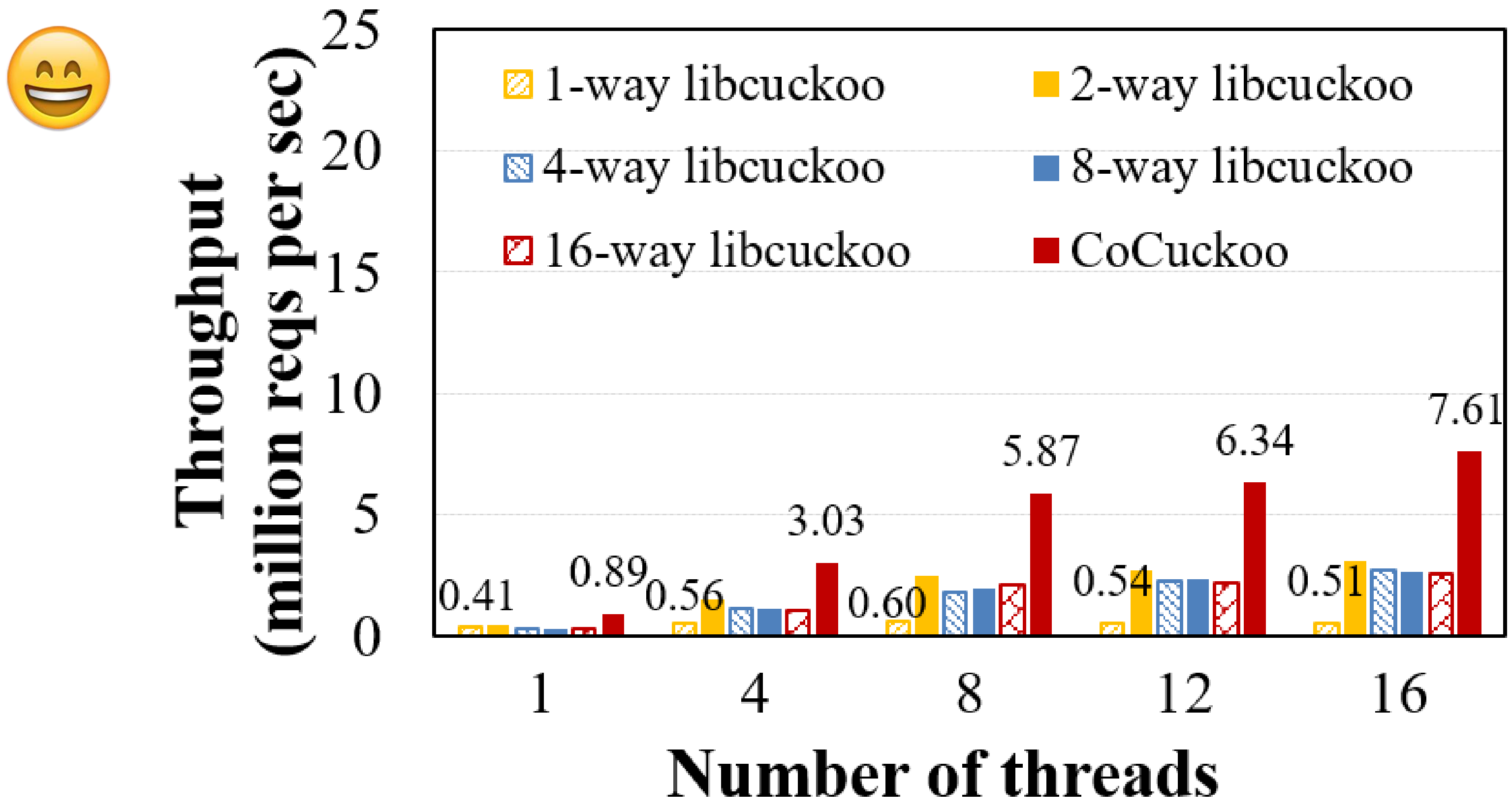
- **Subgraph number generator: a simple modular function**
  - **Modulus: the total number of threads *p***
  - **Remainder: the number of each thread *r***
  - ***n = kp+r , e.g.,* 8-thread CoCuckoo, Thread 2, n=2,10,18,…**

Performance Evaluation

- **Comparison:**
  - **libcuckoo@EuroSys'14**
  - **Slot numbers: 1, 2, 4, 8, 16**
- **Workloads:**
  - **YCSB: https://github.com/brianfrankcooper/YCSB @SOCC'11**
  - **2 million key-value pairs per workload**
- **Threads: 1, 4, 8, 12, 16**
- **Metrics:**
  - **Throughput**
  - **Predetermination for insertion**
  - **Extra space overhead**

Workload	Insert	Lookup
Insert-only (INS)	100%	0%
Insert-heavy (IH)	75%	25%
Insert-lookup balance (ILB)	50%	50%
Lookup-heavy (LH)	25%	75%
Lookup-only (LO)	0%	100%

Average Insertion Throughput



- ➢ **CoCuckoo significantly increases average throughputs.**
- ➢ **75%-150% improvements compared to 2-way libcuckoo.**

Predetermination for Insertion

Workloads	TwoEmpty	OneEmpty	Same_non	Max	Diff_non_non	Diff_non_max
Insert-only	25.673%	37.9628%	0.0003%	13.9802%	13.1447%	9.239%
Insert-heavy	32.9343%	40.4907%	0.0004%	3.5921%	16.7513%	6.2312%
Insert-lookup balance	44.675%	39.6011%	0.0002%	0%	15.7235%	0.0002%
Lookup-heavy	64.4448%	30.1658%	0%	0%	5.3894%	0%

Predetermination for Insertion

Workloads	TwoEmpty	OneEmpty	Same_non	Max	Diff_non_non	Diff_non_max
Insert-only	25.673%	37.9628%	0.0003%	13.9802%	13.1447%	9.239%
Insert-heavy	32.9343%	40.4907%	0.0004%	3.5921%	16.7513%	6.2312%
Insert-lookup balance	44.675%	39.6011%	0.0002%	0%	15.7235%	0.0002%
Lookup-heavy	64.4448%	30.1658%	0%	0%	5.3894%	0%

- **TwoEmpty and OneEmpty account for a large proportion**
  - **Short-term or no locks for the shared buckets**

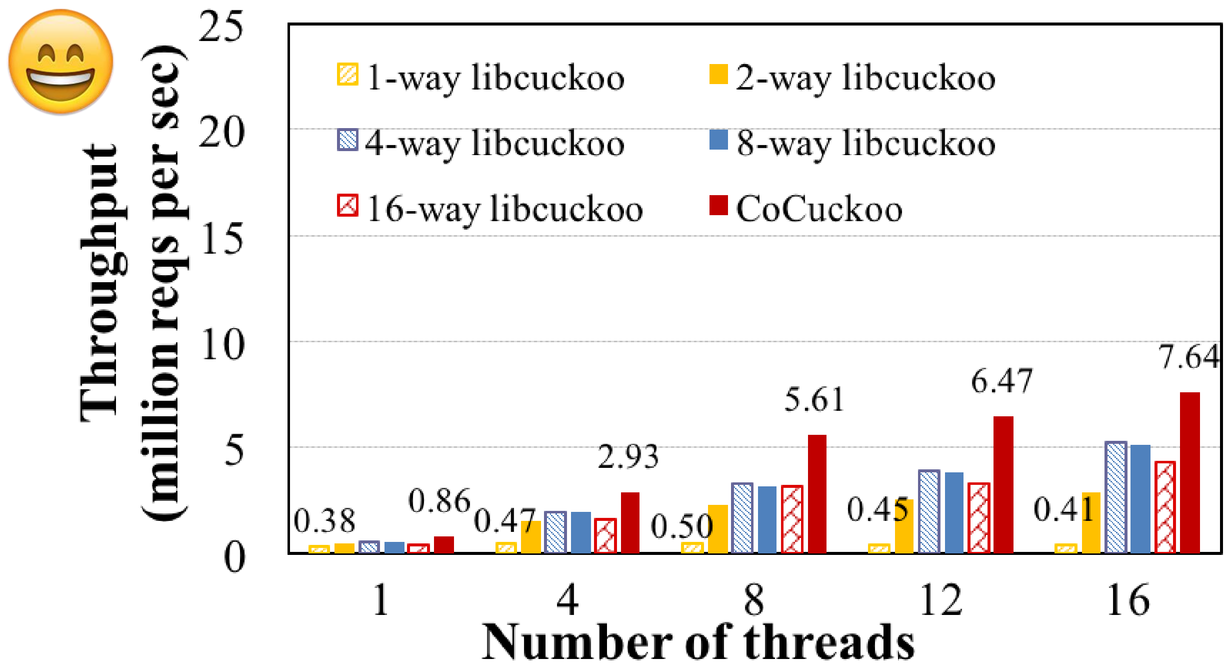
Predetermination for Insertion

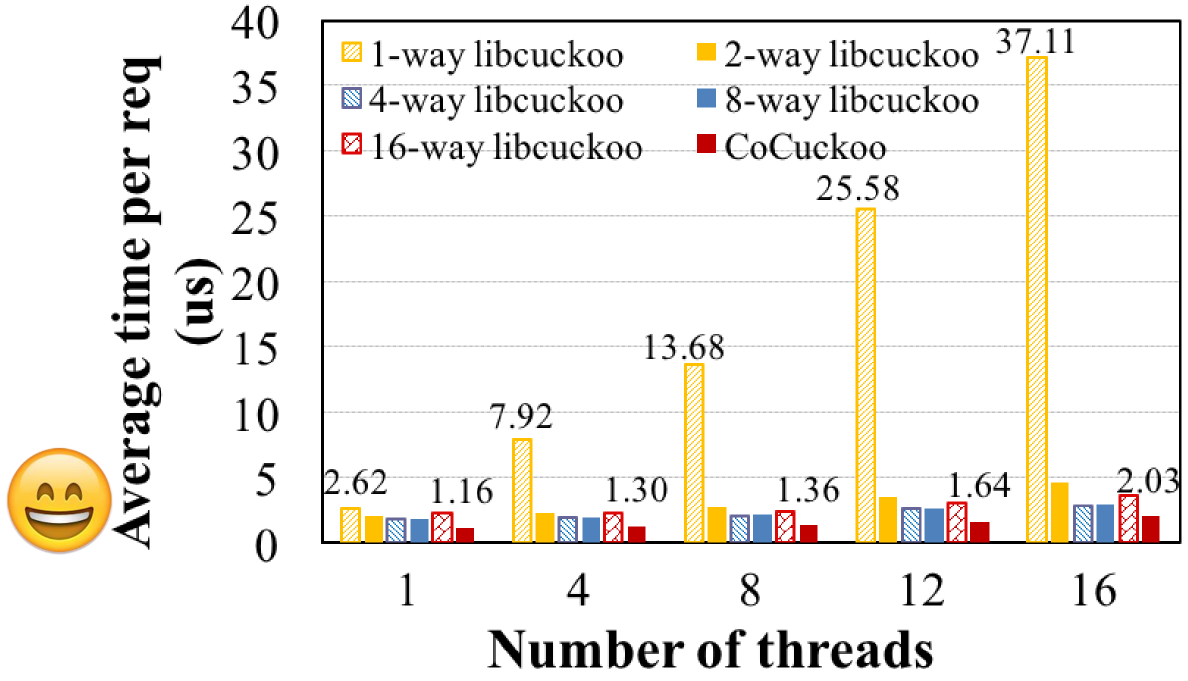
Workloads	TwoEmpty	OneEmpty	Same_non	Max	Diff_non_non	Diff_non_max
Insert-only	25.673%	37.9628%	0.0003%	13.9802%	13.1447%	9.239%
Insert-heavy	32.9343%	40.4907%	0.0004%	3.5921%	16.7513%	6.2312%
Insert-lookup balance	44.675%	39.6011%	0.0002%	0%	15.7235%	0.0002%
Lookup-heavy	64.4448%	30.1658%	0%	0%	5.3894%	0%

- **TwoEmpty and OneEmpty account for a large proportion**
  - **Short-term or no locks for the shared buckets**

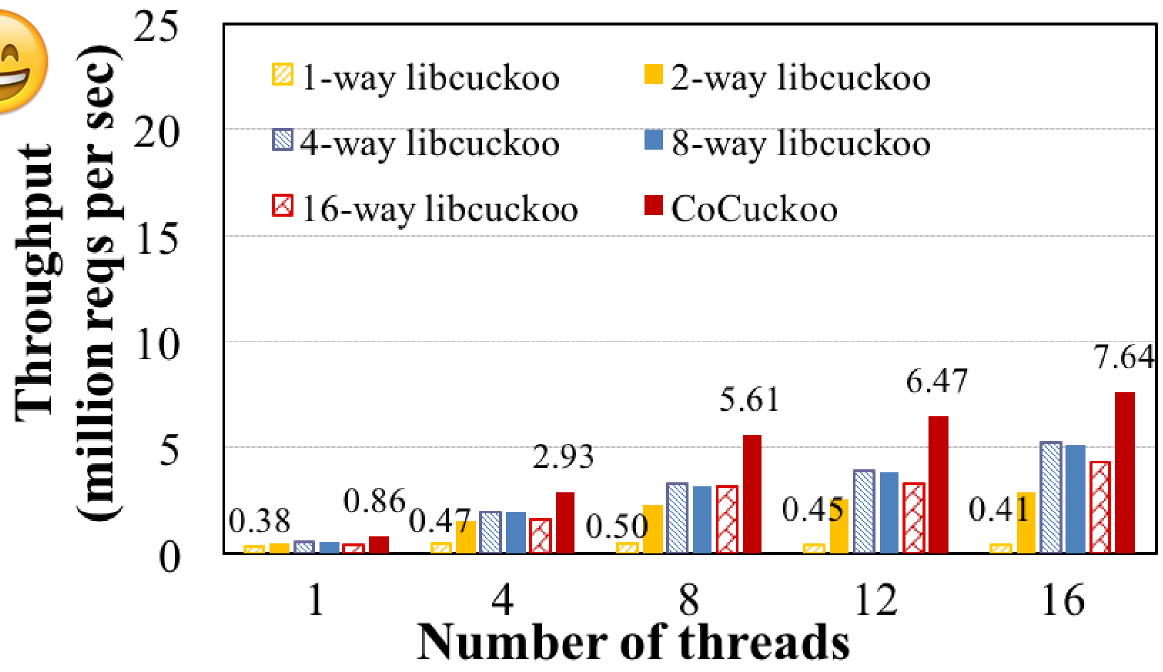
- **Max:**
  - **Predetermine insertion failures and release locks without any kick-out operations**

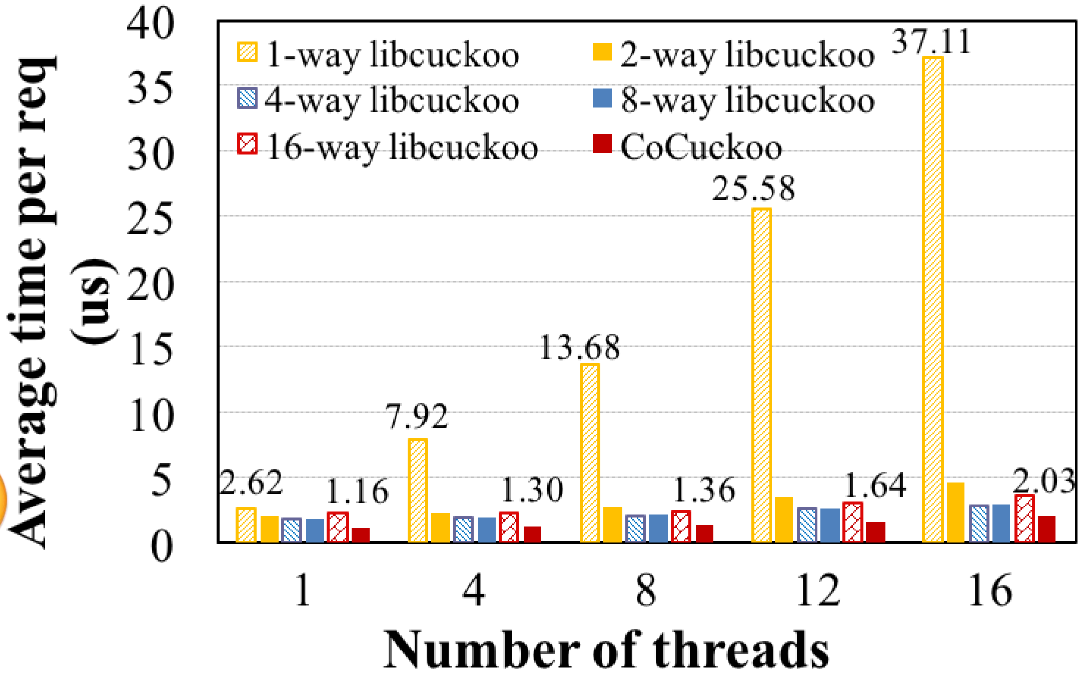
Extra Space Overhead





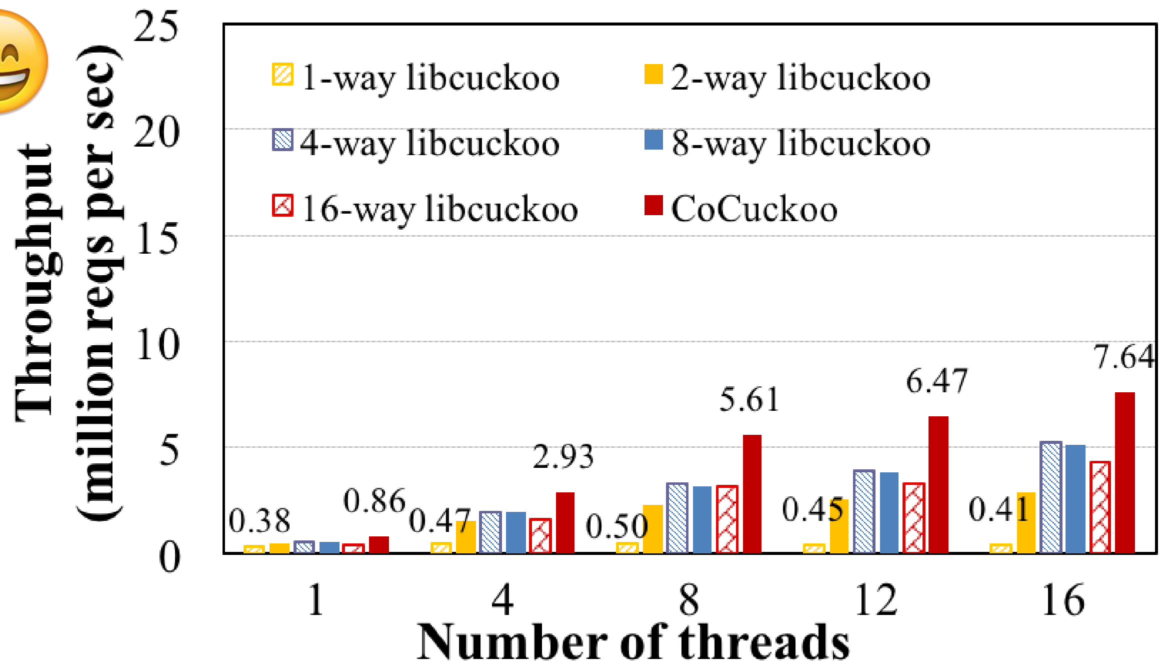
Extra Space Overhead

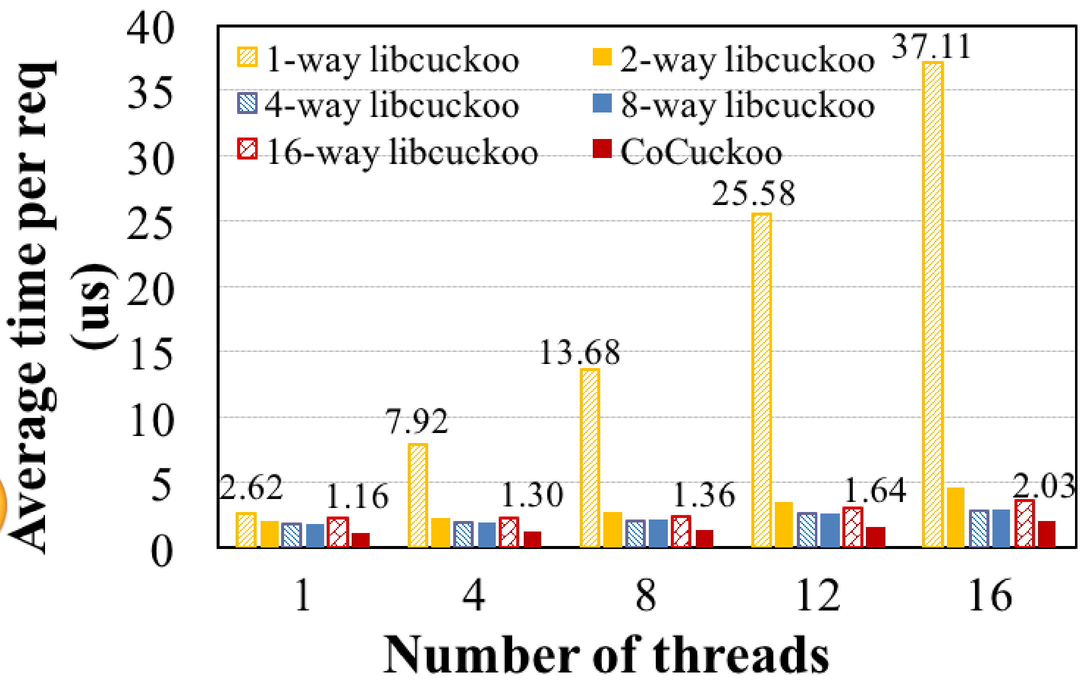




- **The same space available for both libcuckoo and CoCuckoo**
  - **CoCuckoo increases the throughput over 2-way libcuckoo by 73% - 159%.**
  - **CoCuckoo significantly decreases the average execution time per request.**

Extra Space Overhead





- **The same space available for both libcuckoo and CoCuckoo**
  - **CoCuckoo increases the throughput over 2-way libcuckoo by 73% - 159%.**
  - **CoCuckoo significantly decreases the average execution time per request.**
- **The extra space overhead is small**

Conclusion

- **CoCuckoo mitigates the asymmetric read and write costs in cuckoo hashing via**
  - **A pseudoforest to predetermine and avoid occurrence of endless loops**
  - **Graph-grained locking mechanism and concurrency optimization**

- **CoCuckoo achieves 75%-150% write throughput improvements compared with 2-way libcuckoo.**

Q&A

Homepage:: https://csunyy.github.io/