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
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

- High latency for handling hash collisions
Cuckoo Hashing

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

![Diagram of Cuckoo Hashing]

T1

T2

T1: a, n, k

T2: m, b

6
Cuckoo Hashing

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

Insert(x)
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 Diagram]

11
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

**Bottleneck:** Asymmetric reads and writes!

An endless loop occurs!
Concurrent cuckoo hashing strategy:

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

![Diagram of Pseudoforest](image)

*Insert(y)*
Pseudoforest

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

```
T1
  f  a  n  k

T2
  m  c  b
```

```
T1
  f
  a
  n
  k

T2
  m
  c
  b
```

```
Insert(y)
```

```
C
  a
  f
  m
```

```
C
  a
  f
  m
  n
```

21
- Vertex: a bucket
- Edge: an inserted item from the storage vertex to its backup vertex
- Identify endless loops: \#Vertices = \#Edges (called maximal)

\[ T1 \quad T2 \]

\[ \begin{array}{cccc}
\text{f} & \text{a} & \text{n} & \text{k} \\
\text{m} & \text{c} & \text{b} \\
\end{array} \]

\[ \text{Insert}(y) \]

\[ \text{Maximal} \]

\[ \begin{array}{ccc}
\text{c} & \text{n} \\
\text{a} & \text{f} \\
\text{m} \\
\end{array} \]
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**

- Insert(y)

**Diagram**:
- **T1**: bucket structure with vertices f, a, n, k
- **T2**: bucket structure with vertices m, c, b

**Graph**:
- **Maximal**:
  - Vertices: a, f, m
- **Non-maximal**:
  - Vertices: b, k
  - Vacancy (empty vertex)
- **Vertex**: a bucket
- **Edge**: an inserted item from the storage vertex to its backup vertex
- **Identify endless loops**: $\#\text{Vertices} = \#\text{Edges}$ (called maximal)

![Diagram](image-url)
Graph-grained Locking

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

![Diagram](https://via.placeholder.com/150)

- T1: f, a, n, k
- T2: m, c, b

- Pseudoforest graph:
  - a → c → f
  - m → c
  - n
  - b
  - k
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

![Locking Classification]

- 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

- TwoEmpty
- OneEmpty
- ZeroEmpty
Graph-grained Locking

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

![Diagram](image)

- **According to the number of corresponding EMPTY subgraphs**
  - TwoEmpty
  - OneEmpty
  - ZeroEmpty

- **According to the states and the number of subgraphs**
  - Diff_non_non
  - Same_non
  - Diff_non_max
  - Max
Two Empty subgraphs

Before insertion

T1

T2
Two Empty

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

Before insertion

T1

T2

With graph-grained lock(s)

Out of the critical path
TwoEmpty

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

Before insertion: ○ ○ ○

After insertion: ○ ○ ○

With graph-grained lock(s)

Out of the critical path
One Empty

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

Before insertion

T1: f a k
T2:


- 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

![Diagram showing subgraphs T1 and T2 with item insertion](image)

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

![Diagram](image)

Insert(c)
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

After insertion

Non-maximal

Before insertion

After insertion

Non-maximal
ZeroEmpty (Same_non)  ❌

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

Before insertion

Insert(m)

T1: f a n k
T2: c b
The same non-maximal subgraph

Insertion algorithm:

- Mark as maximal
- Kick-out (with item insertion)

Before insertion

```
Insert(m)
```

![Diagram](image)
ZeroEmpty (Same\_non)

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

Before insertion

- Insert(m)

```
T1: f a n k
T2: c b
Non-maximal: c → n
```
The same non-maximal subgraph

Insertion algorithm:
- Mark as maximal
- Kick-out (with item insertion)

Before insertion

Maximal

Insert(m)
ZeroEmpty (Same_non)

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

Before insertion

Before insertion

Maximal

T1

<table>
<thead>
<tr>
<th>f</th>
<th>a</th>
<th>n</th>
<th>k</th>
</tr>
</thead>
</table>

T2

| c | b |

Insert(m)
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

```
Insert(y)
```

```
T1
f a n k

T2
m c b
```
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

Insert(y)

T1

f  a  n  k

T2

m  c  b

Diff_non_max

T1

f  a  n  k

T2

m  c  b

n

b

k
ZeroEmpty (Diff_non_max) 〇 ●

- 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

Diagram showing the process of inserting an item into a graph and identifying non-maximal and maximal 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

![Diagram of subgraphs T1 and T2 with marked nodes and arrows for insertions.]

Insert(y)
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
Two maximal subgraphs or the same maximal subgraph
Always walking into a loop and predetermined to be a failure
Insertion algorithm:
Do nothing
- 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)  ● ● / ●

- Insert(x)
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

![Diagram](image-url)
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](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

<table>
<thead>
<tr>
<th>Workload</th>
<th>Insert</th>
<th>Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert-only (INS)</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Insert-heavy (IH)</td>
<td>75%</td>
<td>25%</td>
</tr>
<tr>
<td>Insert-lookup balance (ILB)</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>Lookup-heavy (LH)</td>
<td>25%</td>
<td>75%</td>
</tr>
<tr>
<td>Lookup-only (LO)</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>
CoCuckoo significantly increases average throughputs.
75%-150% improvements compared to 2-way libcuckoo.
## Predetermination for Insertion

<table>
<thead>
<tr>
<th>Workloads</th>
<th>TwoEmpty</th>
<th>OneEmpty</th>
<th>Same_non</th>
<th>Max</th>
<th>Diff_non_non</th>
<th>Diff_non_max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert-only</td>
<td>25.673%</td>
<td>37.9628%</td>
<td>0.0003%</td>
<td>13.9802%</td>
<td>13.1447%</td>
<td>9.239%</td>
</tr>
<tr>
<td>Insert-heavy</td>
<td>32.9343%</td>
<td>40.4907%</td>
<td>0.0004%</td>
<td>3.5921%</td>
<td>16.7513%</td>
<td>6.2312%</td>
</tr>
<tr>
<td>Insert-lookup balance</td>
<td>44.675%</td>
<td>39.6011%</td>
<td>0.0002%</td>
<td>0%</td>
<td>15.7235%</td>
<td>0.0002%</td>
</tr>
<tr>
<td>Lookup-heavy</td>
<td>64.4448%</td>
<td>30.1658%</td>
<td>0%</td>
<td>0%</td>
<td>5.3894%</td>
<td>0%</td>
</tr>
</tbody>
</table>
### Predetermination for Insertion

<table>
<thead>
<tr>
<th>Workloads</th>
<th>TwoEmpty</th>
<th>OneEmpty</th>
<th>Same_non</th>
<th>Max</th>
<th>Diff_non_non</th>
<th>Diff_non_max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert-only</td>
<td>25.673%</td>
<td>37.9628%</td>
<td>0.0003%</td>
<td>13.9802%</td>
<td>13.1447%</td>
<td>9.239%</td>
</tr>
<tr>
<td>Insert-heavy</td>
<td>32.9343%</td>
<td>40.4907%</td>
<td>0.0004%</td>
<td>3.5921%</td>
<td>16.7513%</td>
<td>6.2312%</td>
</tr>
<tr>
<td>Insert-lookup balance</td>
<td>44.675%</td>
<td>39.6011%</td>
<td>0.0002%</td>
<td>0%</td>
<td>15.7235%</td>
<td>0.0002%</td>
</tr>
<tr>
<td>Lookup-heavy</td>
<td>64.4448%</td>
<td>30.1658%</td>
<td>0%</td>
<td>0%</td>
<td>5.3894%</td>
<td>0%</td>
</tr>
</tbody>
</table>

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

Throughput (million reqs per sec)

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1-way libcuckoo</th>
<th>2-way libcuckoo</th>
<th>4-way libcuckoo</th>
<th>8-way libcuckoo</th>
<th>16-way libcuckoo</th>
<th>CoCuckoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.38</td>
<td>0.86</td>
<td>0.47</td>
<td>0.50</td>
<td>0.45</td>
<td>0.41</td>
</tr>
<tr>
<td>4</td>
<td>2.93</td>
<td>5.61</td>
<td>0.45</td>
<td>0.41</td>
<td>6.47</td>
<td>7.64</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average time per req (us)

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1-way libcuckoo</th>
<th>2-way libcuckoo</th>
<th>4-way libcuckoo</th>
<th>8-way libcuckoo</th>
<th>16-way libcuckoo</th>
<th>CoCuckoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.62</td>
<td>1.16</td>
<td>1.30</td>
<td>1.36</td>
<td>1.64</td>
<td>2.03</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 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
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/