A tutorial on efficient hash table algorithms

Yebangyu(bangyu.yby@alipay.com)

OceanBase

http://www.yebangyu.org

2015.12.29

Outline

- Why Separate Chaining & Open Addressing are not enough
- Cuckoo Hashing
- Hopscotch Hashing

Outline

- Why Separate Chaining & Open Addressing are not enough
- Cuckoo Hashing
- Hopscotch Hashing

Disadvantages of Separate Chaining

- Poor cache performance
- O(n) worst case behavior

Disadvantages of Open Addressing

- Performance degrades as the table fills up
- Clustering of keys cause a large variance in performance

Outline

- Why Separate Chaining & Linear Probe are not enough
- Cuckoo Hashing
- Hopscotch Hashing

Cuckoo Hashing

- Motivation
- Intuition
- How it works
- Implementation details

Authors of cuckoo hashing

- **Rasmus Pagh,** Professor of IT University of Copenhagen
- Research in algorithms for massive data, database performance
- describe cuckoo hashing as a Ph.D student in 2001

Motivation of Cuckoo Hashing

- Traditional methods O(n) worst case behavior
- Suppose that all data can fit in to memory
- Can search/find/contains O(1) worst case ?
- How to accelerate the search/find/contains operations ?

Intuition of Cuckoo Hashing

- Each item could hash to K positions /buckets
- When no position vacant, just rearrange the elements

How Cuckoo Hashing Works

- K hash functions and K tables. (Usually K = 2)
- Insert(x)

  index1 = hashfunc1(x) index2 = hashfunc2(x)

  if (table1[index1] vacant) put x there

  else if (table2[index2] vacant) put x there

  else rearrange

Insert(99) in Cuckoo Hashing

index	key
0	12
1	34
2	56
3	vacant
4	78

Index	key
0	11
1	22
2	vacant
3	33
4	44

Hashfunc1(99) = 2
Hashfunc2(99) = 3

Insert(99) in Cuckoo Hashing

index	key
0	12
1	34
2	56
3	vacant
4	78

Index	key
0	11
1	22
2	vacant
3	99
4	44

33

Hashfunc1(99) = 2
Hashfunc2(99) = 3
Insert 99 to table2

Insert(99) in Cuckoo Hashing

index	key
0	12
1	34
2	56
3	vacant
4	33

Index	key
0	11
1	22
2	vacant
3	99
4	44

78

Hashfunc1(99) = 2
Hashfunc2(99) = 3
Insert 99 to table2
Hashfunc1(33) = 4
Insert 33 to table1

Insert(99) in Cuckoo Hashing

index	key
0	12
1	34
2	56
3	vacant
4	33

Index	key
0	11
1	22
2	78
3	99
4	44

Hashfunc1(99) = 2
Hashfunc2(99) = 3
Insert 99 to table2
Hashfunc1(33) = 4
Insert 33 to table1
Hashfunc2(78) = 2
Insert 78 to table2

Why rearrangement may fail

Why rearrangement may fail

- Table is full
- Cycle

What to do when insertion failed

- Change hash functions OR Resize the tables
- When K =2 and load factor < 50%

  the probability of a cycle is very low

  expected number of displacements is a small constant
- What about K =3 or larger ? Open Question !

Efficiency of Cuckoo Hashing

- Insert: Amortized (Expected) const time.
- Find: O(1) worst case
- Delete: O(1) worst case
- 50% space efficient for K =2

Implementation of Cuckoo Hashing

- Two tables is not a must. It just made the analysis much easier
- Only one table & two hash functions is enough

Implementation of Cuckoo Hashing

- Two tables is not a must. It just made the analysis much easier
- In practice, we can use K other than 2 hash functions to be more space efficient

Trick1 of implementation of Cuckoo hashing : d-ary cuckoo hashing

- K tables and K hash functions
- K possible positions for each key
- 91%+ space efficient
- Well known as d-ary cuckoo hashing

Trick2 of implementation of Cuckoo hashing : bucketized cuckoo hashing

- Each location is a bucket capable of holding B entries

index	K1	K2	K3
0	12	44	vacant
1	33	30	66
2	56	40	vacant
3	11	15	37

Trick3 of implementation of Cuckoo hashing : d-ary + bucketized cuckoo hashing

- Each key is hashed according to K hash functions, leading to K possible locations for that key
- Each location is a bucket capable of holding B entries

index	K1	K2	K3
0	12	44	vacant
1	33	30	66
2	56	40	vacant
3	11	15	37

Trick3 of implementation of Cuckoo hashing : d-ary + bucketized cuckoo hashing

- How to choose K and B ?
- In practice , K =3 or 4 will be enough
- B * sizeof(Key) can fit in to a cacheline

Trick4 of implementation of Cuckoo hashing : d-ary + bucketized + vectorized cuckoo hashing

- Each entry consists of key and payload
- B Keys are stored contiguously , followed by B contiguous payloads
- Use SIMD to compare keys

index	K1	K2	K3	K4	P1	P2	P3	P4
0	11	44	77	vacant	Bob	Alice	Marry	vacant
1	22	55	vacant	vacant	Lucy	Jack	vacant	vacant
2	33	66	88	vacant	Jeff	Jure	David	vacant

Why named Cuckoo Hashing

Outline

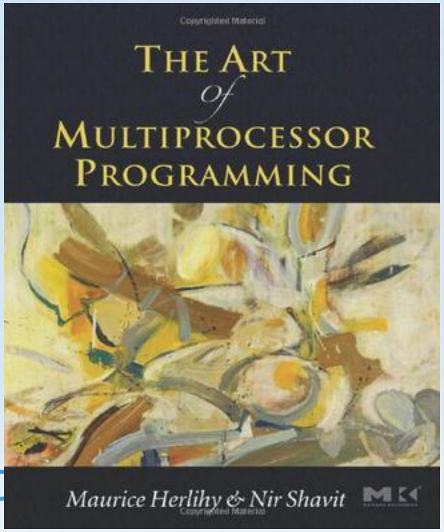
- Why Separate Chaining & Linear Probe are not enough
- Cuckoo Hashing
- Hopscotch Hashing

Hopscotch Hashing

- Motivation
- Intuition
- How it works
- Quiz

Authors of Hopscotch Hashing

- **Nir Shavit,** Professor of MIT
- describe Hopscotch Hashing in 2008
- Co-author of The art of Multicore Programming



Motivation Of Hopscotch Hashing
Disadvantages of Cuckoo Hashing

- need to access sequences of unrelated locations on different cache lines
- need to keep the load factor smaller than 50%

Intuition Of Hopscotch Hashing

- Each key can be located in H positions, whose elements can be fit in to a cacheline
- Element will be evicted to the new location whose distance from the original position is at most H

Hopscotch Hashing

- Motivation
- Intuition
- How it works
- Quiz

How it works

- How to add item x where $h(x) = i$:
  - Starting at i, use linear probing to find an empty entry at index k.
  - If the empty entry's index k is within $H - 1$ of i, place x there and return.
  - Otherwise, find a closer position via HopInfo

What is HopInfo

- Each position i contains HopInfo which tells whether the item in the alternate position is occupied by an element that hashes to i
- Hop[8] = 0010 indicates that only position 10 currently contains items whose hash value is 8, while positions 8, 9, and 11 do not.

index	item	Hop
6	G	1000
7	H	1100
8	M	0010

How to find a closer position

- Otherwise, find a closer position via HopInfo

To create an empty entry closer to i, find an item y whose hash value m lies between i and k.

m satisfies that $k - H < m < k$

Displacing y to k creates a new empty slot closer to i. Repeat. If no such item exists, or if the bucket already i contains H items, resize and rehash the table.

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14	vacant	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

H = 4

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14	vacant	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

Linear Probing suggests 14

14 – 6 >=H

TOO FAR !

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14	vacant	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

Linear Probing suggests 14

14 – 6 >=H

TOO FAR !

Consult HopInfo[11]
Move G Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12	F	1000
13	vacant	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

Linear Probing suggests 14

14 – 6 >=H

TOO FAR !

Consult HopInfo[11]
Move G Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12	F	1000
13	vacant	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

13 – 6 >=H
Still TOO FAR !
Consult HopInfo[10] Does Not Help
Consult HopInfo[11] Does Not Help
Consult HopInfo[12] Move F Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12	vacant	0100
13	F	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

13 – 6 >=H
Still TOO FAR !
Consult HopInfo[10] Does Not Help
Consult HopInfo[11] Does Not Help
Consult HopInfo[12] Move F Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12	vacant	0100
13	F	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

12 – 6 >=H
Still TOO FAR
Consult HopInfo[9] Move B Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	vacant	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

12 – 6 >=H
Still TOO FAR
Consult HopInfo[9] Move B Down

Insert(K)

index	item	HopInfo
6	C	1000
7	A	1100
8	D	0010
9	vacant	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

9 – 6 <H
GOOD ! Insert K there

Insert(K)

index	item	HopInfo
6	C	1001
7	A	1100
8	D	0010
9	K	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000

A:7 B:9 C:6
D:7 E:8 F: 12
G: 11 H: 9 K: 6

9 – 6 <H
GOOD ! Insert K there

Find(K) and Del(K)

- index = hash(K). Just check positions range from index to index+H-1 to locate or / and del it .
- Both Find and Del cost O(1) time
- Insert costs expected const time

Hopscotch Hashing

- Motivation
- Intuition
- How it works
- Quiz

Quiz

- What's the difference between CH and HH
- Is the sequence of displacements possible be cyclic in HH
- When we have to resize in HH
- Will HH work without hopinfo
- What data structures can be used to store hopinfo

Next Episode Preview

- BloomFilter
- HyperLogLog

Thank You All