Count-min sketch to Infinity:

Using Probabilistic Data Structures to Solve Presence, Counting, and Distinct Count Problems in .NET

presented by: Steve Lorello - Developer Advocate @Redis

redis

Steve Lorello
Developer Advocate
@Redis

@slorello
github.com/slorello89
twitch.tv/redisinc

redis

Repo For This Talk

Agenda

- What are Probabilistic Data Structures?
- Brief History of Probabilistic Data Structures in Redis
- Set Membership problems - Bloom Filters
- Counting problems - Count-min Sketch
- Cardinality problems - HyperLogLog
- Heavy Hitter problems - Heavy Keeper

What are Probabilistic Data Structures?

- Class of specialized data structures
- Tackle specific problems
- Use probability approximate
- Some implicit trade-off for performance

Probabilistic Data Structures Examples

Name	Problem Solved	Optimization
Bloom Filter	Presence	Space, Insertion Time, Lookup Time
Quotient Filter	Presence	Space, Insertion Time, Lookup Time
Skip List	Ordering and Searching	Insertion Time, Search time
HyperLogLog	Set Cardinality	Space, Insertion Time, Lookup Time
Count-min-sketch	Counting occurrences on large sets	Space, Insertion Time, Lookup Time
Cuckoo Filter	Presence	Space, Insertion Time, Lookup Time
Heavy Keeper	Keep track of top records	Space, Insertion Time, Lookup Time

A Brief History of Probabilistic Data Structures in Redis

Probabilistic Data Structures in Redis OSS

- Piqued interest of Salvatore and Community
- Redis was always:
  - Memory-first
  - Performance First
- Added HyperLogLog to Redis in v2.8.9 (2014)

Enter Module API

- Module API comes on the scene in Redis 4.0 (2017)
- Modules allow developers to:
    - Extend Redis
    - Develop Custom Data Structures in Redis
    - Add Custom Commands for Redis

Redis Inc Modules - Redis Bloom

- Redis Releases Redis Bloom (2017)
  - Brings Bloom Filters to Redis
  - Licenced under Redis Source Available License (RSAL)
- Redis adds Count-Mink Sketch and TopK in 2019
- Redis Consolidates all modules into Redis Stack 2022

Set Membership

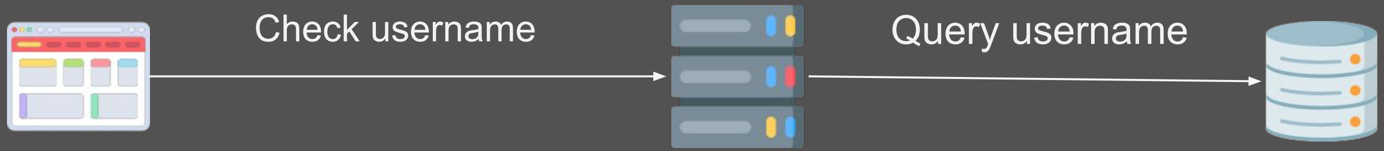
redis

Set Membership Problems

- Has a given element been inserted?
- e.g. Unique username for registration

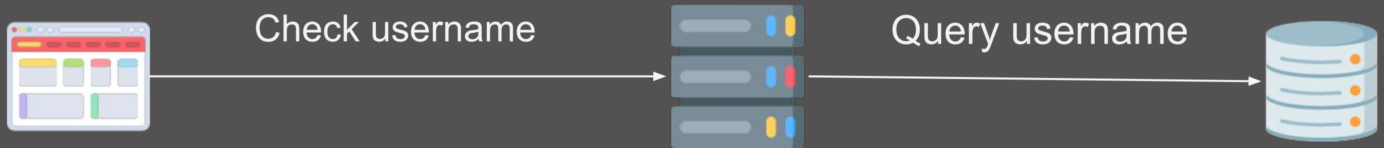
Presence Problem Naive Approach 1

- Store User Info in table ‘users’ and Query



Presence Problem Naive Approach SQL

```
SELECT COUNT(*)
FROM users
WHERE username = 'selected_username'
```



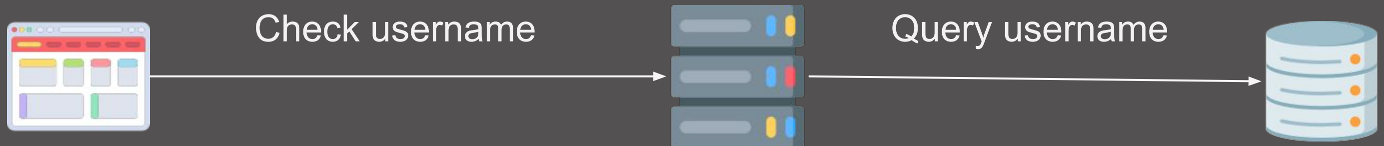
Summary

Access Type	Disk
Lookup Time	$O(n)$
Extra Space (beyond storing user info)	$O(1)$

redis

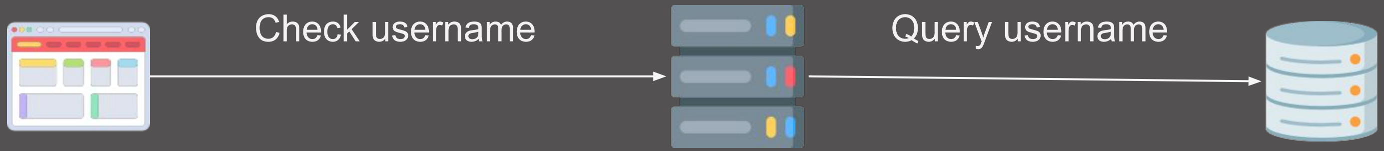
Presence Problem Naive Approach SQL Indexed

- Store User Info in table 'users'
- **Index username**



Presence Problem Naive Approach SQL Indexed

```
SELECT COUNT(*)
FROM users
WHERE username = 'selected_username'
```



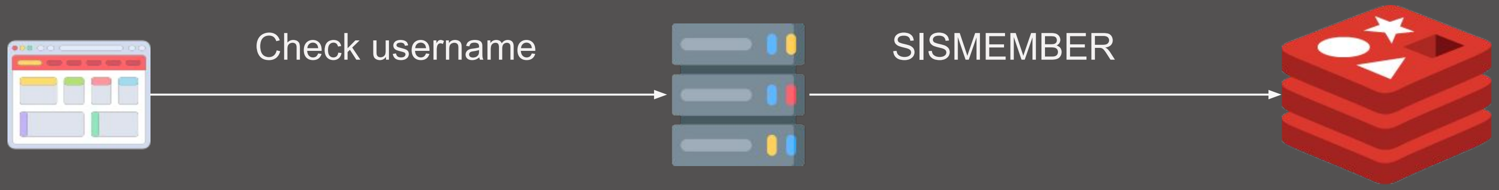
Summary

Access Type	Disk
Lookup Time	O(log(n))
Extra Space (beyond storing user info)	O(n)

redis

Presence Problem Naive Approach Redis

- Store usernames in Redis cache
- SADD usernames selected_username
- SISMEMBER usernames selected_username



Summary

Access Type	Memory
Lookup Time	O(1)
Extra Space (beyond storing user info)	O(n)

Bloom Filters

Bloom Filter

- Specialized ‘Probabilistic’ Data Structure for presence checks
- Can say if element has ***definitely*** not been added
- Can say if element has ***probably*** been added
- Uses constant K-hashes scheme
- Represented as a 1D array of bits
- All operations O(1) complexity
- Space complexity O(n) - bits

Insert:

For i = 0->K:

    FILTER[H[i](key)] = 1

QUERY:

```
For i = 0 -> K:
    If FILTER[H[ i ](key)] == 0:
        Return False
Return true
```

Complexities

Type	Worst Case
Space	O(n) - BITS
Insert	O(1)
Lookup	O(1)
Delete	Not Available

redis

Example Initial State

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	0	0	0	0	0	0	0	0

Example Insert username ‘razzle’

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	0	0	0	0	0	0	0	0

redis

Example Insert username ‘razzle’

- H1(razzle) = 2

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	0	0	0	0	0

redis

Example Insert username ‘razzle’

- H1(razzle) = 2
- H2(razzle) = 5

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	1	0	0	0	0

redis

Example Insert username ‘razzle’

- H1(razzle) = 2
- H2(razzle) = 5
- H3(razzle) = 8

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	1	0	0	1	0

Example Query username ‘fizzle’

H1(fizzle) = 8 - bit 8 is set—maybe?

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	1	0	0	1	0

Example Query username ‘fizzle’

H1(fizzle) = 8 - bit 8 is set—maybe?
H2(fizzle) = 2 - bit 2 is set—maybe?

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	1	0	0	1	0

Example Query username ‘fizzle’

H1(fizzle) = 8 - bit 8 is set—maybe?
H2(fizzle) = 2 - bit 2 is set—maybe?
H3(fizzle) = 4 - bit 4 is not set—definitely not.

Bloom Filter k = 3										
bit	0	1	2	3	4	5	6	7	8	9
state	0	0	1	0	0	1	0	0	1	0

False Positives and Tuning

- This algorithm will never give you false negatives, but it is possible to report false positives
- Optimize False Positives when Reserving Filter
- BF.RESERVE takes:
  - Error Rate: Probability of error
  - Capacity: Expected Cardinality of Filter

Counting Problems

What’s a Counting Problem?

- How many times does an individual occur in a stream
- Easy to do on small-mid size streams of data
- Very hard to scale to enormous data sets
- e.g. Counting Views on YouTube

Naive Approach: Hash Table

- Hash Table of Counters
- Lookup name in Hash table, instead of storing record, store an integer
- On insert, increment the integer
- On query, check the integer

redis

Pros

- Straight Forward
- Guaranteed accuracy (if storing whole object)

Cons

- O(n) Space Complexity in the best case
- Scales poorly (think billions of unique records)

Naive Approach Relational DB

- Issue a Query to a traditional Relational Database searching for a count of record where some condition occurs

SELECT COUNT(*) FROM views

WHERE name=”Gangnam Style”

Linear Time Complexity O(n)

Linear Space Complexity O(n)

redis

What’s the problem with a Billion Unique Records?

- Each unique record needs its own space in a Hash Table or row in a RDBMS (perhaps several rows across multiple tables)
- Taxing on memory for Hash Table
  - 8 bit integer? 1GB
  - 16 bit? 2GB
  - 32 bit? 4GB
  - 64 bit? 8GB
- Maintaining such large data structures in a typical program’s memory isn’t feasible
- In a relational database, it’s stored on disk

Count-min Sketch

Count-Min Sketch

- Specialized data structure for keeping count on very large streams of data
- Similar to Bloom filter in Concept - multi-hashed record
- 2D array of counters
- Sublinear Space Complexity (possibly even constant!)
- Constant Time complexity
- Never undercounts, sometimes over counts

Increment:

```
For i = 0 -> k:
    Table[ H(i) ][ i ]  += 1
```

Query:

```
minimum = infinity
For i = 0 -> k:
    minimum = min(minimum,Table[H(i)][i])
return minimum
```

Video Views Sketch 10 x 3

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	0	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	0	0	0	0	0	0
H3	0	0	0	0	0	0	0	0	0	0

redis

Increment Gangnam Style

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	0	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	0	0	0	0	0	0
H3	0	0	0	0	0	0	0	0	0	0

Increment Gangnam Style

- H1(Gangnam Style) = 0

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	1	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	0	0	0	0	0	0
H3	0	0	0	0	0	0	0	0	0	0

Increment Gangnam Style

- H1(Gangnam Style) = 0
- H2(Gangnam Style) = 4

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	1	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	0	0	0	0	0
H3	0	0	0	0	0	0	0	0	0	0

Increment Gangnam Style

- H1(Gangnam Style) = 0
- H2(Gangnam Style) = 4
- H3(Gangnam Style) = 6

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	1	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	0	0	0	0	0
H3	0	0	0	0	0	0	1	0	0	0

Increment Baby Shark

- H1(Baby Shark) = 0
- H2(Baby Shark) = 5
- H3(Baby Shark) = 6

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	1	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	0	0	0	0	0
H3	0	0	0	0	0	0	1	0	0	0

redis

Increment Baby Shark

- H1(Baby Shark) = 0

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	0	0	0	0	0
H3	0	0	0	0	0	0	1	0	0	0

redis

Increment Baby Shark

- H1(Baby Shark) = 0
- H2(Baby Shark) = 5

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	1	0	0	0	0
H3	0	0	0	0	0	0	1	0	0	0

redis

Increment Baby Shark

- H1(Baby Shark) = 0
- H2(Baby Shark) = 5
- H3(Baby Shark) = 6

Count Min Sketch

position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	1	0	0	0	0
H3	0	0	0	0	0	0	2	0	0	0

Query Gangnam Style

- H1(Gangnam Style) = 0

- MIN (2)

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	1	0	0	0	0
H3	0	0	0	0	0	0	2	0	0	0

Query Gangnam Style

- H1(Gangnam Style) = 0
- H2(Gangnam Style) = 4

- MIN (2, 1)

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	1	0	0	0	0
H3	0	0	0	0	0	0	2	0	0	0

Query Gangnam Style

- H1(Gangnam Style) = 0
- H2(Gangnam Style) = 4
- H3(Gangnam Style) = 6
- MIN (2, 1, 2) = 1

Count Min Sketch										
position	0	1	2	3	4	5	6	7	8	9
H1	2	0	0	0	0	0	0	0	0	0
H2	0	0	0	0	1	1	0	0	0	0
H3	0	0	0	0	0	0	2	0	0	0

Complexities

Type	Worst Case
Space	Sublinear
Increment	O(1)
Query	O(1)
Delete	Not Available

CMS Pros

- Extremely Fast - O(1)
- Super compact - sublinear
- Impossible to undercount

CMS Cons

- incidence of overcounting - all results are approximations

When to Use a Count Min Sketch?

- Counting many unique instances
- When Approximation is fine
- When counts are likely to be skewed (think YouTube video views)

Set Cardinality

redis

Set Cardinality

- Counting distinct elements inserted into set
- Easier on smaller data sets
- For exact counts - must preserve all unique elements
- Scales very poorly

Naive Approach - SQL

```
SELECT COUNT (DISTINCT id)
FROM views
```

Complexities

Space - Unindexed	O(1)
Query - Unindexed	O(n * log(n))
Space - Indexed	O(n)
Query - Indexed	O(n)
Insert	O(1)

redis

Naive Approach Redis

- Store all Values in Sorted Set or Set
- Use ZCARD or SCARD

Complexities

Space	O(n)
Query	O(1)
Insert	O(log(n)) or O(1)

HyperLogLog

redis

HyperLogLog

- Probabilistic Data Structure to Count Distinct Elements
- Space Complexity is O(1)
- Time Complexity O(1)
- Can handle billions of elements with a few kB of memory

redis

HyperLogLog Walkthrough

- Initialize an array of registers of size 2^P (where P is some constant, usually around 16-18)
- When an Item is inserted
  - Hash the Item
  - Determine the register to update: i - from the left P bits of the item’s hash
  - Set registers[i] to the index of the rightmost 1 in the binary representation of the hash
- When Querying
  - Compute harmonic mean of the registers that have been set
  - Multiply by a constant determined by size of P

Example: Insert Username ‘bar’ P = 16

H(bar) = 3103595182

- 1011 1000 1111 1101 0001 1010 1010 1110
- Take first 16 bits -> 1011 1000 1111 1101 -> 47357 = register index
- Index of rightmost 1 = 1
- registers[47357] = 1

Get Cardinality

- Calculate harmonic mean of only set registers.

Only 1 set register: 47357 -> 1

ceiling(.673 * 1 * 1/(2^1)) = 1

Cardinality = 1

Complexities

Space	O(1)
Query	O(1)
Insert	O(1)

redis

Top Elements

Top Element Flows

- Most Frequent Elements in Stream
- Mission critical for detecting heavy network flows

Naive Approach SQL

```
SELECT id
FROM views
GROUP BY id
ORDER BY count(id)
DESC LIMIT 10
```

Naive Approach Redis

- Store all counts in Sorted Set
  - ZINCRBY views 1 id
- ZREVRANGE to get top elements

Heavy Keeper

Heavy Keeper

- Multi-hash Strategy
- Multiple-arrays with Multiple-counters
- Decay’s smaller flows, promotes large flows
- Min-heap to maintain top-elements

redis

Counter

5-foo

Top-K Empty

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(2)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Gangnam Style

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(2)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Gangnam Style

- H1 = 3 - 0-null

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(2)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Gangnam Style

- H1 = 3 - 0-null
  - It’s null - so increment

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null
A(2)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Gangnam Style

- H2 = 5 - 0-null
  - It’s null - so increment

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Gangnam Style

- H3 = 1 - 0-null
  - It's null - so increment

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Baby Shark

- H1 = 3 - 1-GS
  - Ooooo what do we do?

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Baby Shark

- H1 = 3 - 1-GS
  - TRY to decrement it by 1 with decay probability
  - 1 - (1 * decayActivate())

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

redis

Top-K Insert Baby Shark

- H1 = 3 - 1-GS
  - TRY to decrement it by 1 with decay probability
  - 1 - (1 * decayActivate())
  - Success! Decrement, since 0, replace

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	0-null	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Baby Shark

- H2 = 4 - 0-null
  - Increment

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	1-BS	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	0-null	0-null	0-null	0-null	0-null

Top-K Insert Baby Shark

- H3 = 5 - 0-null
  - Increment

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	1-BS	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null

Top-K Query Gangnam Style

- H1 = 3 - 1-BS
- H2 = 5 - 1-GS
- H3 = 1 - 1-GS

Return Max Where Node is set to Gangnam Style (1)

	B(0)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)
A(0)	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null	0-null	0-null
A(1)	0-null	0-null	0-null	0-null	1-BS	1-GS	0-null	0-null	0-null	0-null
A(2)	0-null	1-GS	0-null	0-null	0-null	1-BS	0-null	0-null	0-null	0-null

Min-Heap Maintenance

- At Insertion, check count
- If Min-Heap contains element, update count
- If Min-Heap does not contain element:
  - Check if count greater than count of root element.
    - Replace if true

Redis Stack, the go to for Probabilistic Data Structures

Repo For This Talk

Resources

Redis
https://redis.io

Source Code For Demo:
https://github.com/slorello89/ProbabilisticDataStructures

C# Implementation Bloom Filter, HyperLogLog, and Count-Min Sketch:
https://github.com/TheAlgorithms/C-Sharp/tree/master/DataStructures/Probabilistic

Slides:
https://www.slideshare.net/StephenLorello/countmin-sketch-to-infinitypdf

Steve Lorello
Developer Advocate
@Redis

@slorello
github.com/slorello89
twitch.tv/redisinc

redis

Come Check Us Out!

Redis University:
https://university.redis.com

Discord:
https://discord.com/invite/redis

redis

Baby Name Freq hash table

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

redis

H(Liam) = 4

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

redis

H(Liam) = 4

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0

H(Sophia) = 8

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0

redis

H(Sophia) = 8

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	1	0

H(Liam) = 4

0	1	2	3	4	5	6	7	8	9
0	0	0	0	2	0	0	0	1	0

Baby Name existence table k = 3

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

H1(Liam)=0 H2(Liam) = 4 H3(Liam) = 6

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

redis

H1(Liam)=0 H2(Liam) = 4 H3(Liam) = 6

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	0	1	0	0	0

H1(Susan)=0 H2(Susan) = 5 H3(Susan) = 6

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	0	1	0	0	0

redis

H1(Susan)=0 H2(Susan) = 5 H3(Susan) = 6

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	1	1	0	0	0

redis

Does Tom Exist? H1(Tom)=1 H2(Tom)=4 H3(Tom) = 5

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	1	1	0	0	0

Does Tom Exist? H1(Tom)=1 H2(Tom)=4 H3(Tom) = 5

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	1	1	0	0	0

A hash of Tom = 0, so no Tom does not exist!

redis

Does Liam Exist? H1(Liam)=0 H2(Liam) = 4 H3(Liam) = 6

0	1	2	3	4	5	6	7	8	9
1	0	0	0	1	1	1	0	0	0

All Hashes of Liam = 1, so we repot YES

2-Choice Hashing

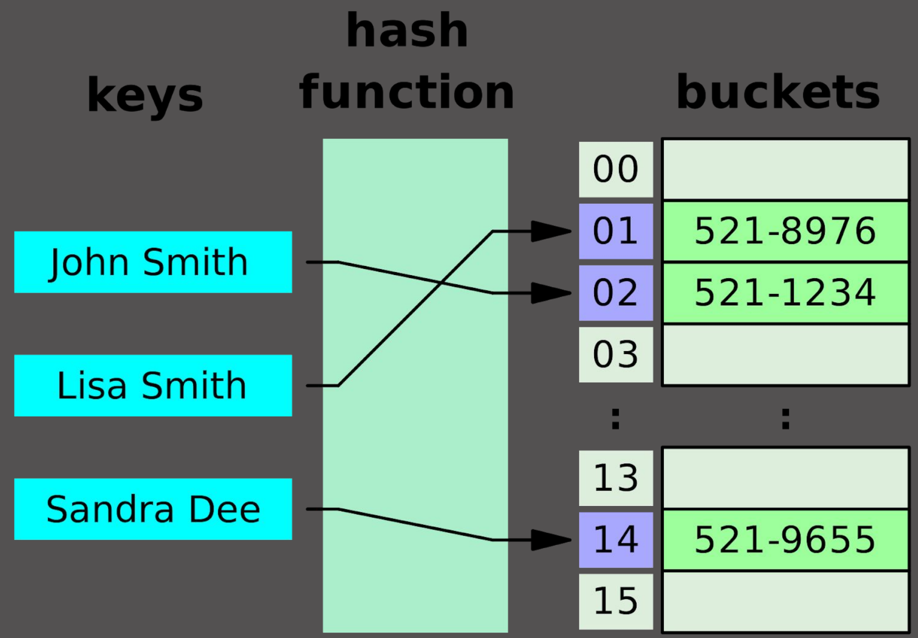
- Use two Hash Functions instead of one
- Store @ index with Lowest Load (smallest linked list)
- Time Complexity goes from log(n) in traditional chain hash table -> log(log(n)) with high probability, so nearly constant
- Benefit stops at 2 hashes, additional hashes don't help
- Still O(n) space complexity

Hash Tables

redis

Hash Table

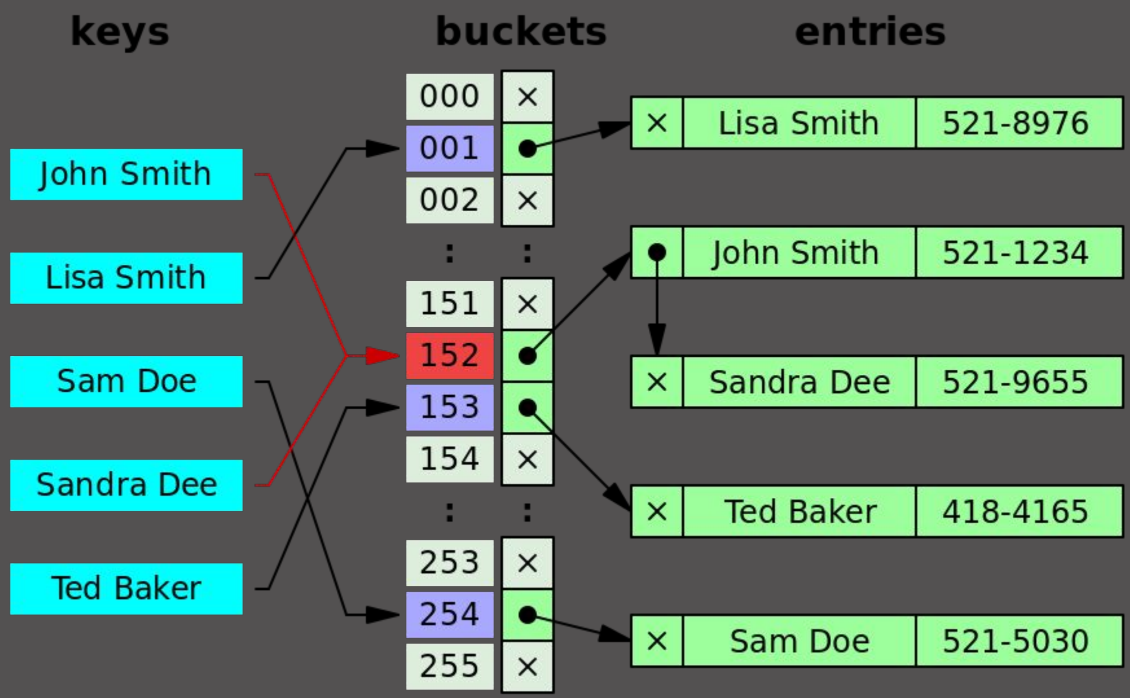
- Ubiquitous data structure for storing associated data. E.g. Map, Dictionary, Dict
- Set of Keys associated with array of values
- Run hash function on key to find position in array to store value



Source: wikipedia

Hash Collisions

- Hash Functions can produce the same output for different keys - creates collision
- Collision Resolution either sequentially of with linked-list



redis

Hash Table Complexity - with chain hashing

Type	Amortized	Worst Case
Space	O(n)	O(n)
Insert	O(1)	O(n)
Lookup	O(1)	O(n)
Delete	O(1)	O(n)

redis