Probabilistic Data Structures and Fingerprinting

Paolo Giaccone

Notes for the class on “Computer aided simulations labs”

Politecnico di Torino

October 2022

Outline

Section 1

Applications

Big Data and probabilistic data structures

3 V's of Big Data

- Volume (amount of data)
- Velocity (speed at which data is arriving and is processed)
- Variety (types of data)

Main efficiency metrics for data structures

- space
- time to write, to update, to read, to delete

Probabilistic data structures

- based on different hashing techniques
- approximated answers, but reliable estimation of the error
- typically, low memory, constant query time, high scaling

Probabilistic data structures

Membership

- answer approximate membership queries
- e.g., Bloom filter, counting Bloom filter, quotient filter, Cuckoo filter

Cardinality

- estimate the number of unique elements in a dataset.
- e.g., linear counting, probabilistic counting, LogLog and HyperLogLog

Frequency

- in streaming applications, find the frequency of some element, filter the most frequent elements in the stream, detect the trending elements, etc.
- e.g., majority algorithm, frequent algorithm, count sketch, count–min sketch

Probabilistic data structures

Rank

- estimate quantiles and percentiles in a data stream using only one pass through the data
- e.g., random sampling, q-digest, t-digest

Similarity

- find the nearest neighbor for large datasets using sub-linear in time solutions.
- e.g., locality-sensitive hashing, MinHash, SimHash

Section 2

Theoretical background

"Bins and Balls" model for load balancing

- n balls, n bins
- for any dropping policy, average occupancy 1 ball/bin
- a deterministic algorithm, by comparing the occupancy of all n bins, is able to achieve a maximum occupancy 1 ball/bin
  - not always feasible to track the size of $O(n)$ bins, when n is very large

Main question

Is it possible to achieve similar performance to the optimal load balancing (i.e., 1 ball/bin) with an algorithm that track $O(1)$ bins?

Randomized load balancing

Random Dropping policy

1. choose uniformly at random 1 bin

Maximum occupancy sharply concentrated on $\dfrac{\log n}{\log \log n}(1 + O(1))$ and upper bounded by $\dfrac{3 \log n}{\log \log n}$

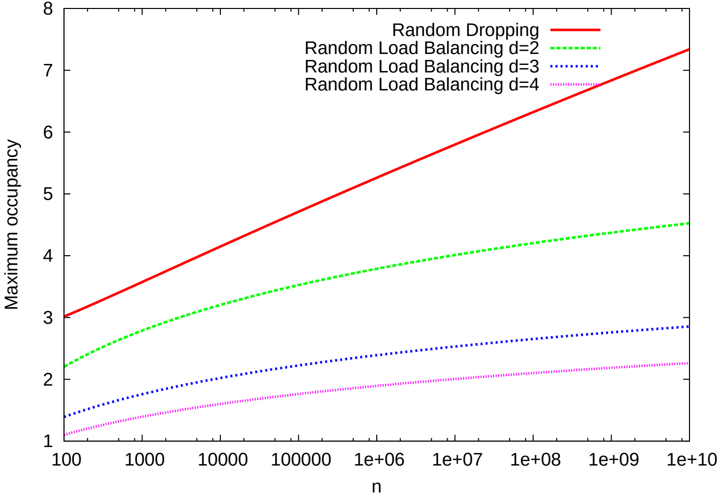
Random Load Balancing policy

1. choose uniformly at random d bins
2. place the ball in the least occupied one

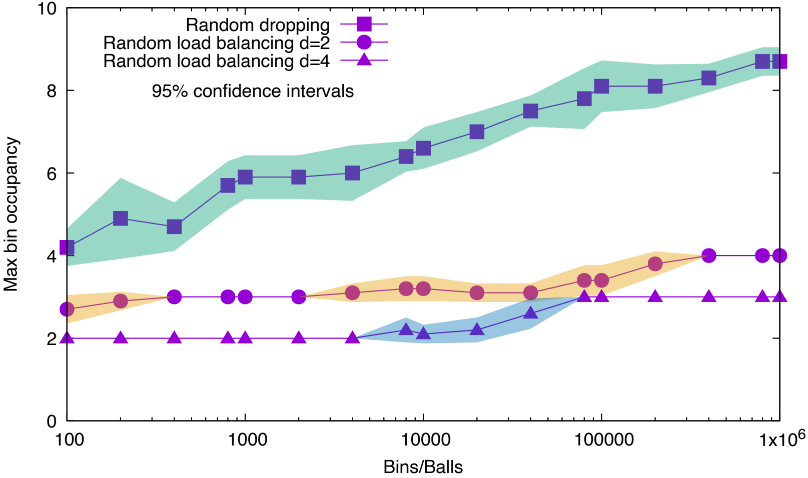
Maximum occupancy sharply concentrated on $\dfrac{\log \log n}{\log d} + O(1)$

Very famous result: "the power of 2 or d random choices"

Theoretical results for randomized dropping policies



Simulation results for randomized dropping policies



The birthday problem

Exact question

What is the minimum number of students in a class to be *sure* that at least 2 of them have the same birthday?

- 366 students (we neglect leap years for simplicity)

Approximated question

What is the minimum number of students in a class such that at least 2 of them have the same birthday with some given probability?

- 23 students to get probability > 0.5
- 41 students to get probability > 0.9

Exact analysis of birthday problem

Assume a class of m students, with $m < 366$. Let $\bar{p}$ be the probability that all m students have distinct birthdays:

$$\bar{p} = \frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{365 - m + 1}{365} = \prod_{i=0}^{m-1} \frac{365 - i}{365} = \prod_{i=0}^{m-1} \left(1 - \frac{i}{365}\right)$$

Now the probability p at least two students have the same birthday (i.e., birthday collision event) is

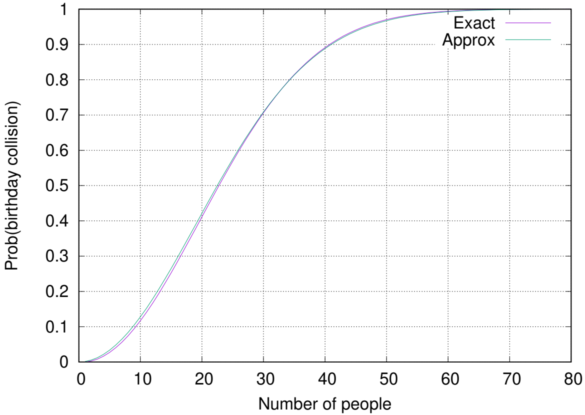
$$p = 1 - \bar{p} = 1 - \prod_{i=0}^{m-1} \left(1 - \frac{i}{365}\right) = 1 - \frac{365!}{(365 - m)! \times 365^m}$$

If $m = 23$ then $p = 0.507$. If $m = 41$, then $p = 0.903$.

Approximated analysis of birthday problem

Now observe that $e^{-ax} \approx 1 - ax$ if x is small

$$p \approx 1 - \prod_{i=0}^{m-1} e^{-\frac{i}{365}} = 1 - e^{-\frac{\sum_{i=0}^{m-1} i}{365}} = 1 - e^{-\frac{m(m-1)}{2\times 365}} \approx 1 - e^{-\frac{m^2}{2\times 365}}$$



Generalized version of birthday problem

Probability of collision

Consider m elements chosen uniformly at random, with repetition, from a set of cardinality n, with $m < n$. The probability $p(n)$ that at least one pair of equal elements has been chosen (i.e., a "collision" has occurred) is

$$p(n) \approx 1 - e^{-\frac{m^2}{2n}} \tag{1}$$

Proof: this is just a generalization of birthday problem with a generic number n of days in a year.

Generalized version of birthday problem

Number of elements for a collision

Consider m elements chosen uniformly at random, with repetition, from a set of cardinality n, with $m < n$. To observe at least one collision with probability p, it must hold:

$$m \approx \sqrt{2n \log\left(\frac{1}{1-p}\right)}$$

Proof: From (1), we can write:

$$1 - p = e^{-\frac{m^2}{2n}} \Rightarrow \log(1-p) = -\frac{m^2}{2n} \Rightarrow m = \sqrt{2n \log\left(\frac{1}{1-p}\right)}$$

Generalized version of birthday problem

Typical number of elements for a collision

Assume $p = 0.5$. Then,

$$m \approx 1.17\sqrt{n}$$

It can be shown that the typical number of elements is sharply concentrated around its average, when $n \to \infty$:

$$E[m] = \sqrt{\frac{\pi}{2} \times n} \approx 1.25\sqrt{n}$$

For example, for $n = 365$, $m \approx 22.3$ and $E[m] = 23.9$.

Section 3

Tables

Table

Definition

- alternative names: associative array, map, symbol table, dictionary
- abstract data type composed of a collection of $(key, value)$ pairs
- each possible key appears at most once

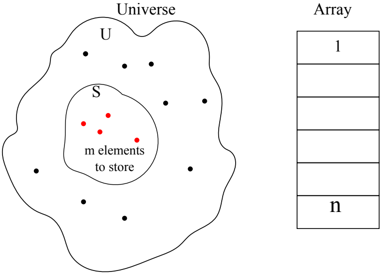
Example:

- in a router, measure the traffic destined to each TCP port
- table with
  - key is the destination TCP port
  - value is the number of bytes for the corresponding TCP packets
- e.g., $T = \{(80, 948567), (25, 5342), (21, 73888), (5436, 234)\}$

Table

Assumptions

- large universe U of possible keys
- small subset S of m keys to store ($S \subset U$, $m = |S|$)
  - $m \ll |U|$
- n array size



Example for measuring traffic for each possible TCP flow

- each TCP flow is identified by (source IP, destination IP, source port, destination port)
- $|U| = 2^{32+32+16+16} = 2^{96} \approx 8\ 10^{28}$
- storage available for $n = 10^6$ entries

Table implementation

Challenges

- fast lookup/update
- space efficiency

Possible solutions

1. direct access arrays
2. traditional hash tables
3. multiple-choice hash tables
4. cuckoo hash tables

Reference: Andrii Gakhov, "Probabilistic Data Structures and Algorithms for Big Data Applications", 2019

Direct access arrays

- one entry for each element in the universe
- array size $n = |U|$
  - unfeasible in many contexts
- if $|U| \gg m$, very inefficient in terms of space

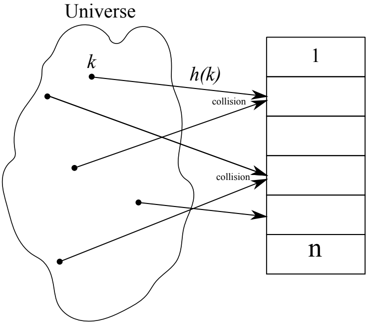
Performance

- average lookup $O(1)$
- worst-case lookup $O(1)$

Hash function

Hash function

- $h : U \to [1, n]$ is a deterministic function that looks random
  - $\Pr(h(k) = j) = 1/n$ (uniform)
  - $\forall k_1 \neq k_2,\ \Pr(h(k_1) = h(k_2)) = 1/n$ (independence between two values)



Hash collision

Hash collision

If two elements have the same hash, i.e., $h(u) = h(u')$, for $u, u' \in U$, then a collision event is experienced.

General birthday problem

An hash collision event for m hashed elements can be modeled as a birthday collision, for the general case of m elements chosen at random, with repetition, from a set of n elements. Thanks to the result in slide 17, we know that hash collisions are likely to occur when $m > 1.25\sqrt{n}$.

n	$1.25\sqrt{n}$	Fingerprint	Example
1024	40	10 bit	-
10^6	1280	20 bit	
$3.4\ 10^{38}$	$2.2\ 10^{19}$	128 bit	MD5
$1.4\ 10^{48}$	$1.5\ 10^{24}$	160 bit	SHA-1
$1.2\ 10^{77}$	$4.2\ 10^{38}$	256 bit	SHA-256

Hash function families

Cryptographic hash functions

- one way, collision resistant, i.e., hard to find keys that collide
- large number of bits to make brute force inversion hard
- slower than non-cryptographic hash functions

Examples of cryptographic hash functions

Name	Bits	Note
MD5 (Message-Digest algorithm)	128	now vulnerable
SHA-1 (Secure Hash Algorithm)	160	now vulnerable
SHA-2	224/256/384/512	

Hash function families

Non-cryptographic hash functions

- not designed to be collision resistant
- guarantee a low probability of collision
- fast to compute

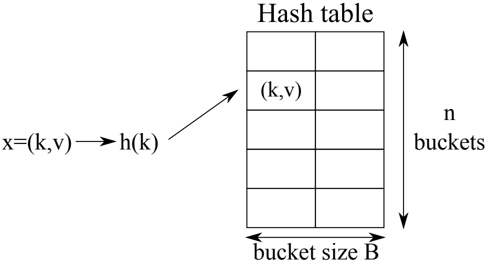
Examples of non-cryptographic hash functions

Name	Bits	Some usage
FNV (Fowler/Noll/Vo)	32/64/ 128/256/ 512/1024	DNS servers, IPv6, Twitter, data base indexing, web search engines
MurmurHash	32/64	Apache Hadoop/Cassandra Elasticsearch, libstc++, nginx load balancer/web server
CityHash	32/64/128/256	For strings
FarmHash	32/64/128	For strings

Traditional hash tables

Hash table

- array of n *buckets*
- store $x = (k, v)$ in bucket $h(k)$ of the table
- if two or more elements are mapped to the same entry (e.g., $h(k_1) = h(k_2)$), use either
  - a variable-size linked list of entries
  - a fixed-size array of B entries



Example of hash table

Hash table H of size $n = 3$. $H[i]$ is the linked list in bucket i.

Data insertion of $m = 6$ elements $x_i = (k_i, \text{value}_i)$

x	k	$h(k)$	$H[1]$	$H[2]$	$H[3]$
x_1	k_1	2	-	x_1	-
x_2	k_2	1	x_2	x_1	-
x_3	k_3	2	x_2	$x_1 \to x_3$	-
x_4	k_4	1	$x_2 \to x_4$	$x_1 \to x_3$	-
x_5	k_5	2	$x_2 \to x_4$	$x_1 \to x_3 \to x_5$	-
x_6	k_6	2	$x_2 \to x_4$	$x_1 \to x_3 \to x_5 \to x_6$	-

Performance

Expected lookup time

- average size of each linked list is $\frac{m}{n}$
- if k is in the table, it takes on average $\frac{1}{2}\frac{m}{n}$ steps
- if k is not in the table, it takes on average $\frac{m}{n}$ steps
- by choosing $m = n$ the average lookup time is $O(1)$

Worst-case lookup time

- when $m = n$, the maximum size of the linked list is as the maximum load in the bins-and-balls model for the random dropping policy
- it takes approximatively $\frac{\log n}{\log \log n}$ steps

Performance

- average lookup $O(1)$
- worst-case lookup $O(\frac{\log n}{\log \log n})$

Implementation

Closed addressing

- store collided elements in a secondary data structure
- when separate chaining is adopted, a linked list is adopted

Open addressing

- store collided elements in buckets other than their preferred positions
- when linear probing is adopted, collided elements are placed in the next empty bucket
- when deleting an element, the bucket must be flagged in order to preserve the chain of stored elements

Python implementation

Dictionary

- array implemented as continuous block of memory for easier indexing
- start with 8 elements and resize (by a factor 2 or 4) every time it is 2/3 full
- open addressing using pseudorandom sequences (with period equal to the array size)
- if interested in the details, enjoy the video "The Mighty Dictionary" by Brandon Craig Rhodes

  `https://archive.org/details/pyvideo_276___the-mighty-dictionary-55`

Compare: `python3.9 -m timeit -s "table ={ 3:1,3+8:2,3+16:3,3+24:4,3+32:5}" "table[3]"`
with: `python3.9 -m timeit -s "table ={ 3:1,3+8:2,3+16:3,3+24:4,3+32:5}" "table[35]"`

Balanced binary tree

Assume $m = n$

Performance

- average lookup $\theta(\log n)$
- worst-case lookup $\theta(\log n)$

Comparison

- better space efficiency than hash table
- average lookup worst than hash table
- worst-case lookup worst than hash table

Multiple-choice hash tables

Policy

- two independent hash functions h_1 and h_2
- if k is not yet available in the table, insert $x = (k, \text{value})$ in the least occupied bucket among $h_1(k)$ and $h_2(k)$

Performance

- average lookup $O(1)$
- worst-case lookup $O(\log \log n)$
  - thanks to "power of two" result for balls and bins

- easy to implement in parallel
- can be generalized to d random choices
  - worst-case lookup time $(\log \log n)/\log d + \theta(1)$

Example

Hash table H of size $n = 3$.

Data insertion of $m = 6$ elements $x_i = (k_i, v_i)$

x	k	$h_1(k)$	$h_2(k)$	$H[1]$	$H[2]$	$H[3]$
x_1	k_1	2	3	-	x_1	-
x_2	k_2	1	2	x_2	x_1	-
x_3	k_3	2	3	x_2	x_1	x_3
x_4	k_4	1	3	x_2	x_1	$x_3 \rightarrow x_4$
x_5	k_5	2	3	x_2	$x_1 \rightarrow x_5$	$x_3 \rightarrow x_4$
x_6	k_6	2	1	$x_2 \rightarrow x_6$	$x_1 \rightarrow x_5$	$x_3 \rightarrow x_4$