Load balancing: power-of-2-choice

When a ball comes in, pick two bins and place the ball in the bin with smaller number of balls.

Turns out with just **checking two bins** maximum number of balls drops to **O(log log n)**!

=> called "power-of-2-choices"

Intuition: Ideas?

Even though max loaded bins has O($\frac{\log N}{\log \log N}$) balls, most bins have far fewer balls.

Load balancing: power-of-2-choice

Proof (Intuition):

For a ball b, let

height(b) = number of balls in its bin after placing b

Probability of an incoming ball getting height 3 is at most ?

- Q: What needs to happen for this?
- Q: Fraction of bins that can have ≥ 2 balls?
  - at most ½ (since there are only N balls)

½ * ½ = ¼

So expected number of bins with 3 balls is at most = N/4

Load balancing: power-of-2-choice

Proof (Intuition) cont.:

(For a ball b, let height(b) = number of balls in its bin after placing b)

Probability of an incoming ball getting height 4 is at most ?

$\frac{1}{4} * \frac{1}{4} = 1/16 = \dfrac{1}{2^{2^{4-2}}}$

Probability of an incoming ball getting height h is at most ?

$$\frac{1}{2^{2^{h-2}}}$$

Choosing h = O(log log N)+2 gives probability 1/N.

Load balancing: power-of-d-choice

When a ball comes in, **pick d bins** and place the ball in the bin with smallest number of balls.

Theorem:

For any d>=2 the d-choice process gives a maximum load of

$$\frac{\log \log N}{\log d} \pm O(1)$$

with probability at least 1 – O(1/N)

Observations:

Just looking at two bins gives huge improvement.

Diminishing returns for looking at more than 2 bins.

15-750: Graduate Algorithms

Hashing:

- Hash function basics and some constructions
- Hash tables
- Bloom filters
- Load balancing (balls and bins)
- Data streaming model

Data streaming model

- Different computational model: elements going past in a "stream"
- Limited storage space: Insufficient to store all the elements
- Example applications:
  - Switch or a router where packets are passing through.
  - Big data

Notation:

- Denote the elements of the stream as $a_1, a_2, \ldots$
- Each element is from an alphabet U
- Each element takes b bits to represent
  - E.g. 32-bit IP addresses
- The question: what functions of input stream can we compute with what time and space overhead.

Data streaming model

- Functions of interest:
  - Sum of all elements seen (easy)
  - Max of the elements seen (easy)
  - Median (tricky to do with small space)
  - **Heavy-hitters, i.e., element(s) that have appeared most often)**
  - Number of distinct elements seen

Sampling vs. Hashing

Sampling is a natural option (since it helps reduce the amount of data)

But can lead to incorrect answers if not done correctly.

Example from [1]:

Suppose we want to figure out

#“uniques” = elements that occur exactly once.

Consider this sampling approach:

- Sample 10% of the stream by picking each element with probability 0.1.
- Count uniques and scale up the answer by 10

1. “Mining of Massive Datasets” book from Stanford: http://infolab.stanford.edu/~ullman/mmds/book.pdf

Sampling vs. Hashing

This will lead to incorrect answer:

Suppose stream length is n and n/2 are uniques and n/4 appear twice.

Q: Correct answer is? n/2

In the sampled stream,

Expected length = n/10

#uniques = 0.1*n/2 + n/4 (2*0.1 – 0.1^2)

(approx.) n/10

So our estimate of #uniques = n (incorrect)

This is in expectation, but will hold with high probability as n gets large (by Chernoff bound)

Sampling vs. Hashing

Q: What was the problem here?

Sampling decision was being made independently on each element of the stream.

Q: What we should have done?

If an element is sampled, all its copies are also sampled

Q: How can we achieve this via hashing?

Hash the elements to the range [10] and take elements that map to one value, say 0.

If we have at least 1-wise independence then we get 1/10 fraction of the stream along with duplicates.

Streams as vectors

Useful abstraction: viewing streams as vectors (in high dimensional space)

Stream at time t as a vector $x^t \in Z^{|U|}$

$$x^t = (x^t{}_1, x^t{}_2, \ldots, x^t{}_{|U|})$$

Element i =

number of times i^{th} element of U has been seen until time t

If next element is j, then x_j is incremented by 1

Streams as vectors

Leads to an extension of the model where each element of the stream is either

(1) A new element or (2) old element departing (i.e. deletions).

That is, updates to the stream looks like (add e) or (del e).

Assumption: #deletes for any element <= #additions.

=> running count for each element is non-zero

E.g.: U = {A, B, C}

add(A), add(B), add(A), del(B), del(A), add(C), . . .

(0, 0, 0), (1, 0, 0), (1, 1, 0), (2, 1, 0), (2, 0, 0), (1, 0, 0), (1, 0, 1), .

Streams as vectors

This vector notation makes it easy to to formulate some of the data stream problems:

- Heavy hitters = estimate "large" entries in the vector x
- Total number of elements seen = Sum of the elements of x (easy one)
- #distinct elements = #non-zero entries in x

Heavy hitters

Many ways to formalize the heavy hitters problem.

ε-heavy-hitters: Indices i such that $x_i > \varepsilon \| x \|_1$

Let us consider a simpler problem first.

Count-Query:

At any time t, given an index i, output the value of x^t_i with an error of at most $\varepsilon\|x^t\|_1$. I.e., output an estimate

$$y_i \in x_i \pm \varepsilon \| x \|_1$$

Q: Given an algorithm for Count-Query, how to get heavy hitters?

To first order: we can look for i's s.t. $y_i > 0$

(at least a good first step)

Heavy hitters

Q: Would sampling work for Count-query?

No. Example: N copies of A arrives and then they all depart.
Then sqrt(N) copies of B arrives.

At the end, heavy hitter = only B

But if we sample the elements with any prob. less that sqrt(N), we don't expect to see any B.

Next:

Hashing-based solution: Count-Min Sketch

By Cormode and Muthukrishnan.

Hashing-based solution: Count-Min Sketch

A hashing based solution (Step 1)

Let h: U -> [M] be a hash function

Let a A[1...M] be an array capable of storing non-negative integers.

When update a_t arrives

```
        If (a_t == (add, i))
                then A[h(i)]++;
else // a_t == (del, i)
                A[h(i)]--;
```

Hashing-based solution: Count-Min Sketch

Estimate for x^t_i: $y_i = A[h(i)]$

Q: What does y_i include?

Count for ith element + for all other elements that has a hash collision with it

<Analysis of expected error for universal hash families>

This is in expectation. Now we want to “boost” the probability that we are close to expectation.

Hashing-based solution: Count-Min Sketch

Estimate for x^t_i: y_i = A[h(i)]

Q: What does y_i include?

Count for ith element + for all other elements that has a hash collision with it

Hashing-based solution: Count-Min Sketch

A hashing based solution (Step 2)

Amplify the probability that we are close to the expectation: Independent repetitions!

ℓ hash functions: $h_1, h_2, .., h_\ell$: U -> [M]

ℓ arrays $A_1, \dots A_\ell$

(one for each hash function)

Same approach as before applied independently on each of the ℓ arrays using the associated hash function.

What should be the new estimate for the count query?