CSE 521: Design & Analysis of Algorithms I

Some Useful Hashing Data Structures

Paul Beame

Some Random Data Structure Ideas

- Bloom Filters
  - Quick certification of non-membership in a set
- The power of two random choices
  - Better load balancing
- Cuckoo hashing
  - Using two choices and data movement for a simple efficient dynamic dictionary data structure

Bloom Filters

- Given a set $\mathbf{S} = \{x_1, x_2, x_3, \ldots, x_n\}$ on a universe **U**, want to answer queries of the form:

$$\textit{Is } \boldsymbol{y} \in \boldsymbol{S} \textit{ ?}$$

- Bloom filter provides an answer in
  - "Constant" time (to hash).
  - Small amount of space.
  - But with small probability of a false positive
    - Useful when the answer is usually NO

Exact Computation based on Universal Hash Function Families

- Family of functions $\mathcal{H}$
  - Each $\mathbf{H}\in\mathcal{H}$ satisfies $\mathbf{H} : \mathbf{U} \rightarrow \{\mathbf{0},\ldots,\mathbf{m-1}\}$
  - Assume that **H** is chosen from $\mathcal{H}$ at random independent of the elements of **S**
- Universal Hash Function Family
  - For any $\mathbf{x}\neq\mathbf{y}\in\mathbf{U}$, $\mathbf{Pr}_{\mathbf{H}\in\mathcal{H}}[\mathbf{H(x)=H(y)}]=\mathbf{1/m}$
- Example Universal Family: $\mathcal{H}$
  - $\mathbf{U}=\{\mathbf{0},\ldots,\mathbf{2^N-1}\}$, $\mathbf{m}=\mathbf{2^M}$
  - each function specified by pair **(a,b)** where **a** is an **(M+N)**-bit integer and $\mathbf{b}\in\{\mathbf{0},\ldots,\mathbf{m-1}\}$
  - $\mathbf{H_{(a,b)}(x)}$=middle **M** bits of **ax+b** (which is **M+2N** bits long)

Exact Computation based on Universal Hash Function Families

- Hash the elements of **U**
- Collisions:
  - Open hashing
    - Place them nearby in the table
  - Separate chaining
    - Extra pointers to follow
  - Double hashing
    - Additional hash table for set of elements that within each table entry
    - Can be made into a perfect hash function with low failure probability but is complex

Bloom Filters

*Start with an **m** bit array, filled with 0s.*

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

*Hash each item x_j in **S** **k** times. If $H_i(x_j) = a$, set $B[a] = 1$.*

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

*To check if **y** is in **S**, check **B** at $H_i(y)$. All **k** values must be **1**.*

B	0	1	0	**0**	1	0	**1**	0	0	1	**1**	1	0	1	1	0

*Possible to have false positive; all **k** values are **1**, but **y** is not in **S**.*

B	0	1	0	0	**1**	0	**1**	0	0	1	**1**	1	0	1	1	0

***n** items* | ***m** = **cn** bits* | ***k** hash functions*

Truly Random Hash Functions

- Instead of using hash function families indexed by a small set like the set of **(a,b)** pairs let $\mathcal{H}$ be the set of all possible functions from **U** to **{0,...,m-1}**

- Then for any set of **s** distinct elements $x_1,\ldots,x_s$ of **U**:

  $\Pr_{H\in\mathcal{H}}[\,H(x_1)=a_1,\ldots,H(x_s)=a_s] = 1/m^s$

- Universal families don't achieve this for large **s**
  - In reality analysis is approximate since we don't usue truly random functions
  - Effectiveness in practice relies on data not being adversarial

False Positive Probability

- **Pr**(specific bit of filter is 0) is

$$p' \equiv (1-1/m)^{kn} \approx e^{-kn/m} \equiv p \qquad (p' \le p)$$

- If β is fraction of **0** bits in the filter then false positive probability for a new element is

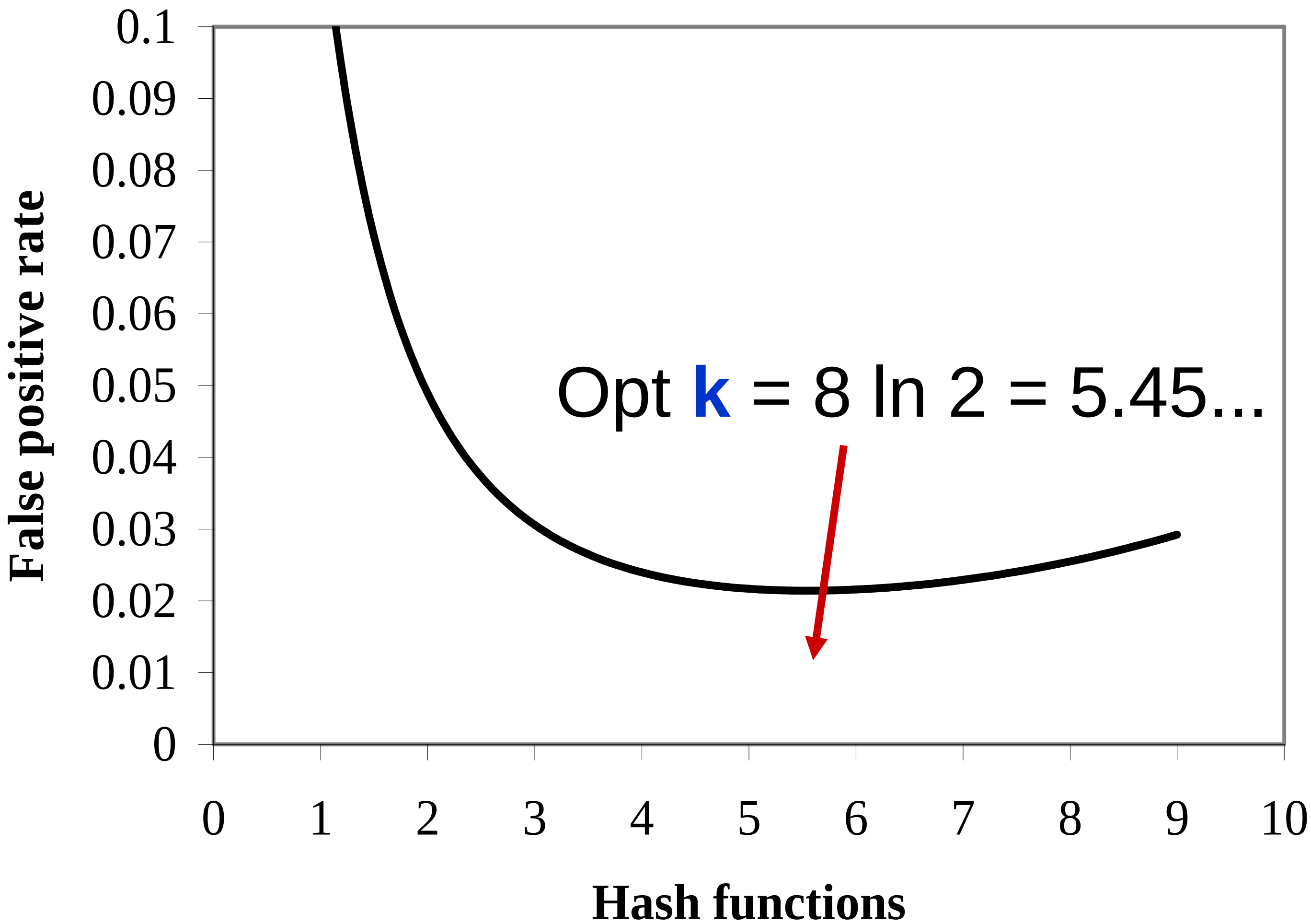
$$(1-\beta)^k \approx (1-p')^k \approx (1-p')^k = (1-e^{-kn/m})^k$$

- Approximations are almost exact since β is concentrated around $E[\beta]$.

- Find optimal at $k = (\ln 2)\, m/n$ by calculus.
  - So optimal false positive prob is about $(0.6185)^{m/n}$

n items ***m* = *cn*** bits ***k*** hash functions

Graph of $(1-e^{-k/c})^k$ for c=8



m/n = 8

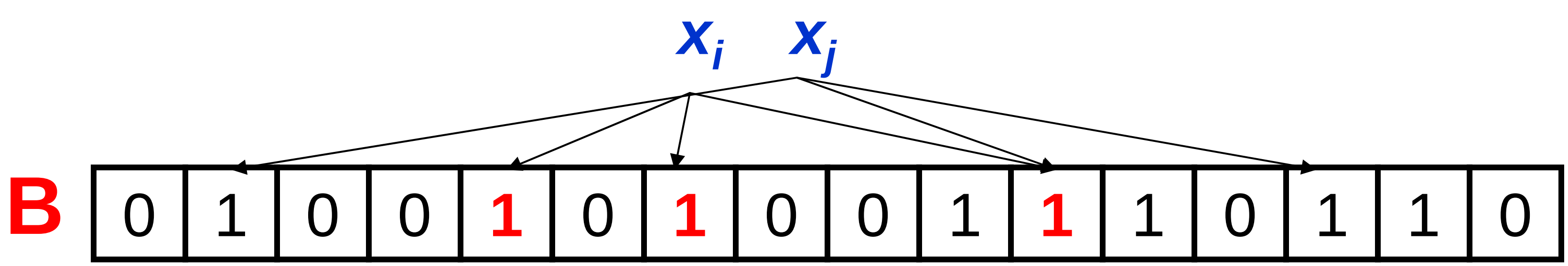
n items *m = cn* bits *k* hash functions

Application Example

- Google BigTable uses Bloom filters to reduce the disk lookups for non-existent rows or columns.
  - Avoiding costly disk lookups considerably increases the performance of a database query operation

Handling Deletions

- Bloom filters can handle insertions, but not deletions.



- If deleting x_i means resetting 1’s to 0’s, then deleting x_i will “delete” x_j.

Counting Bloom Filters

*Start with an **m** bit array, filled with 0s.*

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

*Hash each item x_j in **S** **k** times. If $H_i(x_j) = a$, add 1 to $B[a]$.*

B	0	3	0	0	1	0	2	0	0	3	2	1	0	2	1	0

To delete x_j decrement the corresponding counters.

B	0	**2**	0	0	**0**	0	2	0	0	3	2	1	0	**1**	1	0

Can obtain a corresponding Bloom filter by reducing to 0/1.

B	0	1	0	0	0	0	1	0	0	1	1	1	0	1	1	0

Counting Bloom Filters: Overflow

- Must choose counters large enough to avoid overflow
  - e.g. for **$c=8$** choose **4** bits per counter
  - Average load using **$k = (\ln 2)\, m/n$** counters is $\ln 2$.
  - Probability a counter has load at least **16** is $e^{-\ln 2} (\ln 2)^{16}/16!$ which is roughly **6.78×10^{-17}**

Bloom filter variety

- There are alternative ways to design Bloom filter style data structures that are more effective for some variations, applications

Random Load Balancing

- Assigning tasks to servers
  - Distributed/parallel environment
    - No central control
  - Tasks generated by processes anywhere
    - Indistinguishable
  - Goal: Assign tasks to servers in constant time keeping load balanced
- Simple approach
  - assign each task to a random server
- Case for analysis
  - **n** servers
  - **n** tasks (average load 1)

Random Load Balancing: Tossing Balls into Bins

- tasks ≡ balls, servers ≡ bins
- **Pr** [ball **i** in bin **j**] =**1/n**
- **Pr** [≥ **k** balls in bin **j**] ≤ (**n** choose **k**) n^{-k}
  ≤ $(n^k/k!)\, n^{-k}$
  $= 1/k! \approx 1/k^{\Theta(k)}$
- **Pr**[∃ bin with ≥ **k** balls] ≤ $n/k^{\Theta(k)}$
- In order for this to be small we need
  $k = \Omega(\log n / \log\log n)$
- Imbalance:
  - Some bin will have $\Omega(\log n/\log\log n)$ balls

Random Load Balancing: The Power of Two Choices

- Extra assumption:
  - Process can detect current load of server prior to assignment
- Power of two choices algorithm: [Azar-Broder-Karlin-Upfal]
  - For each task/ball choose **2** servers/bins uniformly at random
  - Assign task/ball to less loaded server/bin
  - More generally: make **d** random choices and assign to least loaded bin

Random Load Balancing: The Power of Two Choices

- Theorem [ABKU] With 2 random choices and assignment to the least loaded bin the no bin contains more than $\log \log \mathbf{n} + \mathbf{O}(\mathbf{1})$ balls almost certainly
  - With $\mathbf{d}$ choices the load goes down to $\log\log \mathbf{n} / \log \mathbf{d} + \mathbf{O}(\mathbf{1})$
- Proof idea:
  - For $\mathbf{i}=\mathbf{0},\mathbf{1},\ldots$ let β_i be the fraction of bins with load at least $\mathbf{i}$.

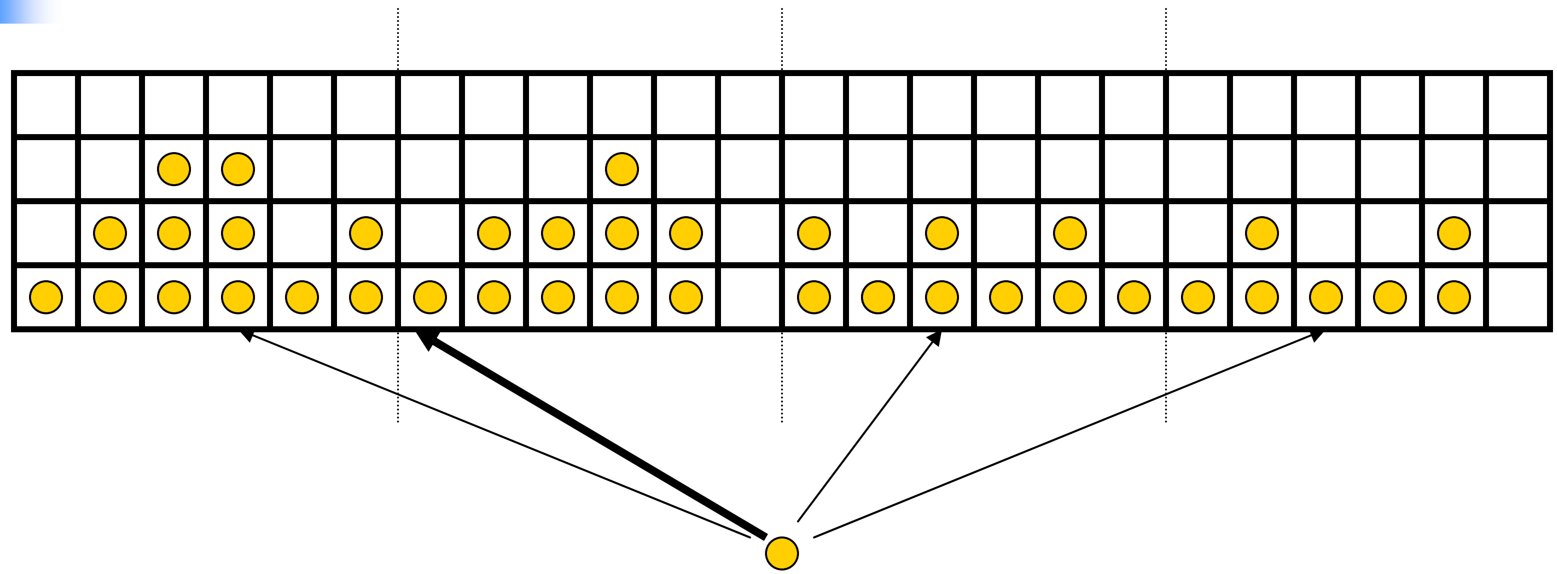
Power of 2 choices rough analysis

- Imagine assigning the balls sequentially
  - Let $\beta_i(\mathbf{t}) \le \beta_i$ denote the fraction of bins with load at least **i** after **t** balls
  - $\beta_0(\mathbf{t})=1$
  - Clearly β_2 is $\le$ ½ since there only **n** balls
  - For $t+1^{st}$ ball to create a bin with load $\ge i+1 \ge 3$, all of its d bin choices must have load $\ge i$.
    - Probability is at most $[\beta_i\,(\mathbf{t})]^d \le \beta_i{}^d$
  - Associate each bin of load $\ge i+1$ with the ball inserted that created that load
  - Expected total # of bins contributing to β_{i+1} is $\le n\,\beta_i{}^d$
  - Roughly implies that $\beta_{i+1} \le \beta_i{}^d$

Power of 2 choices rough analysis

- Since $\beta_2 \le ½$ and $\beta_{i+1} \le \beta_i^{\,d}$ we have $\beta_k \le (½)^{d^{k-2}}$
- Now the expected # of bins of load $\ge$ **k** is $n\,\beta_k \le n\,(½)^{d^{k-2}}$
- This is less than 1 when $\mathbf{n}\,(½)^{d^{k-2}} \le \mathbf{1}$ i.e. when $\log \mathbf{n} \le \mathbf{d}^{k-2}$, that is when $\log\log \mathbf{n} \le (\mathbf{k}-\mathbf{2}) \log \mathbf{d}$ equivalently when $\mathbf{k} \ge \log\log \mathbf{n} / \log \mathbf{d} + \mathbf{2}$
- This is just expected size but can show that with a small change in constant this holds with high probability, though proof is tricky

Extension: *d*-left Hashing



- Split hash table into *d* equal subtables.
- To insert, choose a bucket uniformly for each subtable.
- Place item in a cell in the least loaded bucket, breaking ties to the left.

Property of *d*-left Hashing

- [Vocking] Having **d**-separate tables of size **n/d** and tiebreaking to the left as in random **d**-left hashing is at least as good as independent choices.
  - Almost surely the most loaded bin has load at most $\log\log \mathbf{n}/(\mathbf{d}\Phi_d)+\mathbf{O}(\mathbf{1})$ where $\Phi_d \leq 2$

Cuckoo Hashing

- Simple dynamic perfect hashing using power of 2 choices
  - Use 2 random hash functions h_0 and h_1 to 2 tables of size $(1+\varepsilon)n$
  - To insert x
    - If bin $h_0(x)$ is full then check $h_1(x)$.
    - if both full then bin $h_0(x)$ contains some y with $h_0(y)=h_0(x)$ so set $b=1$ and repeat:
      - kick y out of its nest (as cuckoos do) and insert it in its unique alternative place $h_b(y)$, kicking out whatever z is already there
      - $y \leftarrow z;\quad b \leftarrow 1 - b$
  - It is possible that a cycle is created. To handle this add a max # of iterations through the loop and then rebuild the table using new random hash functions