Hashing

The most important techniques behind Yahoo! are hashing, hashing and hashing! – Udi Manber
Example 1: data bases

- Maintain a set of employees (students, messenger users, ...), each identified by a social security number (student ID, phone number, ...).
Example 2: deduplication

- In a programming language compiler, how to store user-declared identifiers?

- Search in a large search space (e.g. in a large graph)

- Index \textit{k-mer}s in a genomic sequence or words in a text
Hash tables: supported operations

- **Goal**: maintain a (possibly evolving) set of objects belonging to a large “universe” (e.g. configurations, ID numbers, words, etc.)

- **INSERT**: add a new object
- **DELETE**: delete existing object
- **LOOKUP**: retrieve an object

- possibly specified by a *key* (*associative array*)
- generalization of arrays (*direct addressing*)

(dictionary data structure)
Naive solutions

- **Bit array (bitmap)**
  - still too big for huge applications
  - does not support access to objects
  - BUT … (cf Bloom filters at the end of this lecture)

- **Linked list**
  - look-up too slow

- **Search trees**
  - better but still slow and memory demanding
Hash tables

- **Notation**
  - $U$: universe of all possible keys (Ex: strings, IP addresses, game configurations, …)
  - $K$: subset of keys (actually stored in the dictionary), $|K| \ll |U|$
  - $|K| = n$

- **Use a hash table** $T[0..m-1]$
  - hash function maps keys to entries of the hash table (buckets or slots)
  - we lose the direct-addressing ability: key $k$ maps or hashes to bucket $T[h[k]]$
Hashing and collisions

\[ h(k_1) = h(k_5) \]

\[ h(k_2) = h(k_3) \]

\[ U \text{ (universe of keys)} \]

\[ K \text{ (actual keys)} \]

Collision

\[ 0 \]

\[ h(k_1) \]

\[ h(k_4) \]

\[ h(k_2) = h(k_5) \]

\[ h(k_3) \]

\[ m-1 \]
Collisions: birthday "paradox"

- What is the probability that two people from a class of 40 students have their birthday the same day?
Collisions: birthday "paradox"

- What is the probability that two people from a class of 40 students have their birthday the same day?
- Answer: $\approx 0.89$

- Birthday paradox: in a group of 23 people, there is about 50% chance that two people have the same birthday

- Conclusion: collisions are frequent
I. Collision Resolution by Chaining

Universe of keys: $U$

Actual keys: $K$

Collision resolution by chaining:

$h(k_1) = h(k_4)$

$h(k_2) = h(k_5) = h(k_6)$

$h(k_3) = h(k_7)$

$h(k_8)$

$m-1$
Collision Resolution by Chaining

$U$ (universe of keys)

$K$ (actual keys)

$k_1$, $k_2$, $k_3$, $k_4$, $k_5$, $k_6$, $k_7$, $k_8$
Hashing with chaining

- INSERT($T \cdot k$) : $O(1)$
- DELETE($T \cdot k$), LOOKUP($k$): $O$ (list length)

⇒ a good hash function should distribute keys into buckets as uniformly as possible

⇒ uniform hashing ⇒ expected list length is $\alpha = n/m$ (load factor)

- the average time of DELETE and LOOKUP is $O(1 + \alpha)$ ⇒ $O(1)$ if $n = O(m)$ (practical case)
Good hash functions

- Hash function should be easy to compute
- Designing good hash functions is tricky. It is easy to design a bad hash function

*Examples*: Phone numbers. Benford's law (e.g. prices, population sizes, …)

- Keys are usually considered as natural numbers
- *Example*: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, \(\text{CLRS} = 67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141,764,947\)
Division Method

- Map a key $k$ into one of the $m$ slots by taking the remainder of $k$ divided by $m$. That is,
  \[ h(k) = k \mod m \]

- **Example:** $m = 31$ and $k = 78 \Rightarrow h(k) = 16$

- **Advantage:** Fast, since requires just one division operation

- **Disadvantage:** Have to avoid certain values of $m$.
  - Don’t pick certain values, such as $m = 2^p$ (as the hash won’t depend on all bits of $k$)

- **Good choice for $m$:**
  - Primes, not too close to power of 2 (or 10) are good.
Multiplication Method

- If \(0 < A < 1\), \(h(k) = \lfloor m \cdot (kA \mod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor\) where \((kA \mod 1) = kA - \lfloor kA \rfloor\): the fractional part of \(kA\)

- **Disadvantage:** Slower than the division method.

- **Advantage:** Value of \(m\) is not critical.
  - Typically chosen as a power of 2, i.e., \(m = 2^p\), which makes the implementation easy

- **Example:** \(m = 1000, k = 123, A = 0.6180339887\ldots\)
  \[ h(k) = \lfloor 1000(123 \cdot 0.6180339887 \mod 1) \rfloor = \lfloor 1000 \cdot 0.018169 \ldots \rfloor = 18 \]
II. Collision Resolution by Open Addressing

- All elements are stored in the hash table itself
  \[ \Rightarrow n \leq m, \text{ no pointers} \]

- hash function \( h(k, i) \) where \( i = 0, 1, 2, \ldots, m - 1 \), and \( < h(k, 0), h(k, 1), \ldots, h(k, m - 1) > \) is a permutation

- when inserting/looking up \( k \), probe \( h(k, 0), h(k, 1), \ldots \) (probe sequence) until
  - we find \( k \), or
  - the bucket contains \( \text{nil} \), or
  - \( m \) buckets have been unsuccessfully probed

- deletion is complicated, needs a special key "deleted", time may not be dependent on the load factor
Open Addressing: Linear probing

- The colliding item is placed in a different cell of the table

- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell $h(k, i) = (h'(k) + i) \mod m$

- Each table cell inspected is referred to as a “probe”

- Colliding items clump together, causing future collisions to cause a longer sequence of probes

**Example:**

- $h'(k) = k \mod 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order
Example (cont.)

$h'(k) = k \mod 13$
Quadratic probing

- \( h(k, j) = (h'(k) + c_1 \cdot j + c_2 \cdot j^2) \mod m \)

- for example, \( h(k, j) = (h'(k) + \frac{j(j+1)}{2}) \mod m \)

  \( h(k, 0), h(k, 1), ..., h(k, m - 1) \) is a permutation if \( m \) is a power of 2

  **Probing:**
  
  \( j = 0; i = h'(k) \)

  while bucket \( j \) is not empty AND \( j < m - 1 \)
  
  \( j = j + 1; i = i + j \)

- quadratic probing works better than linear probing (less clumping)
Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  \[ h(k, j) = (h(k) + jd(k)) \mod m \]
  for $j = 0, 1, \ldots, m-1$

- The secondary hash function $d(k)$ cannot have zero values

- $m$ should be relatively prime to $d(k)$, e.g. $m = 2^q$ and $d(k)$ odd, or $m$ is prime and $d(k) < m$

- Double hashing is usually more efficient than linear and quadratic probing
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - \( m = 13 \)
  - \( h(k) = k \mod 13 \)
  - \( d(k) = 7 - k \mod 7 \)
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

<table>
<thead>
<tr>
<th>( k )</th>
<th>( h(k) )</th>
<th>( d(k) )</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Linear probing vs Double hashing

comparison of average number of operations

successful search

unsuccessful search
Assuming that $< h(k, 0), h(k, 1), \ldots, h(k, m - 1) >$ is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

$$\frac{1}{1 - \alpha},$$

where $\alpha = n/m$ the load factor.

The expected number of probes for a successful search is

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1 - \frac{i}{m}} \leq \frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right)$$
Historical remarks

Hashing by open addressing: Analysed by Donald Knuth in the 60th (invention attributed to Andrei Ershov)
Hashing: some conclusions

- **Chaining:**
  - easy implementation
  - fast in practice
  - uses more memory

- **Open addressing:**
  - uses less memory
  - more complex removals

- Implemented in standard libraries, e.g. `std::unordered_map` in C++
Universal hashing

(Variant of hashing with chaining)
Universal hashing (with chaining)

- *Motivation*: avoid pathological ("adversary") datasets
  - practical applications in avoiding DDoS attacks
Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications’ data structures. Frequently used data structures have “average-case” expected running time that’s far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was dropping as much as 71% of its traffic and consuming all of its CPU. We show how modern universal hashing techniques can yield performance comparable to commonplace hash functions while being provably secure against these attacks.

Assume O(n) time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take O(n^2) time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in common with other low-bandwidth DoS attacks, such as stack smashing [2] or the ping-of-death [4], wherein a relatively short message causes an Internet server to crash or misbehave. While a variety of techniques can be used to address these DoS attacks, com-
DISCLAIMER: Do not use any of the material presented here to cause harm. I will find out where you live, I will surprise you in your sleep and I will tickle you so hard that you will promise to behave until the end of your days.

The story so far

Last year, at 28c3, Alexander Klink and Julian Wälde presented a way to run a denial-of-service attack against web applications.

One of the most impressing demonstrations of the attack was sending crafted data to a web application. The web application would dutifully parse that data into a hash table, not knowing that the data was carefully chosen in a way so that each key being sent would cause a collision in the hash table. The result is that the malicious data sent to the web application elicits worst case performance behavior of the hash table implementation. Instead of amortized constant time for an insertion, every insertion now causes a collision and degrades our hash table to nothing more than a fancy linked list, requiring linear time in the table size for each consecutive insertion. Details can be found in the Wikipedia article.
Universal hashing (with chaining)

- introduced by Carter&Wegman (1979)

- **Definition:** A family $H$ of hash functions is called *universal* iff for any pair of keys $k, l$,

  $$P_{h \in H}[h(k) = h(l)] \leq 1/m$$

  Equivalently, the number of hash functions $h$ with $h(k) = h(l)$ is $\leq |H|/m$

- **Theorem:** the expected time (over $h \in H$) of INSERT, DELETE, LOOKUP is $O(1 + \alpha)$

  **NB:** no assumption on the distribution of keys
Universal class of hash functions

- Choose a large prime $p$ (larger than the maximum key)
- Let $0 \leq a, b \leq p - 1, a \neq 0$
- for a key $k$, define $h_{a,b}(k) = ((ak + b) \mod p) \mod m$
- $H_{p,m} = \{h_{a,b}\}$ is a universal family of hash functions

**Proof idea:**
- given $a, b$, values $((ak + b) \mod p)$ are distinct for different $k$'s
- for $k_1 \neq k_2$ distinct pairs $(a, b)$ yield distinct pairs $((ak_1 + b) \mod p, (ak_2 + b) \mod p)$
- $P[((ak_1 + b) \mod p = \mod_m (ak_2 + b) \mod p)] \leq 1/m$

- In practice $p$ is often set to $2^{31} - 1$ for 32-bit numbers and to $2^{61} - 1$ for 64-bit numbers (Mersenne primes)
Example of universal hashing

- Assume we are hashing IP addresses $x_1.x_2.x_3.x_4$ with $0 \leq x_i \leq 255$
- Choose $m$ a prime number
- Consider quadruples $a = (a_1, a_2, a_3, a_4)$ with $0 \leq a_i \leq m - 1$
- Define
  $$h_a(x_1.x_2.x_3.x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \mod m$$
- $H = \{h_a\}$ is a universal family (can be proved)
Remarks

Other (more) efficient universal hashing schemes exist, such as Multiply-shift \[\text{Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem. J. Algorithms, 25:19–51, 1997}\]

\[h_a: [0..2^w - 1] \rightarrow [0..2^l - 1]\] defined by
\[h_a(x) = \left\lfloor (ax \mod 2^w)/2^{w-l} \right\rfloor\]
for a random \(w\)-bit odd integer \(a\)

for details see [M.Thorup, High Speed Hashing for Integers and Strings, arxiv:1504.06804, May 2019]
Perfect hashing
Motivation

- Can we guarantee a worst-case $O(1)$ time for hash table operations?
Motivation

- Can we guarantee a **worst-case** $O(1)$ time for hash table operations?

- **Yes** if the set of keys is static
Motivation

- Can we guarantee a worst-case $O(1)$ time for hash table operations?

- Yes if the set of keys is static

Naive solution: enumerate keys $\Rightarrow$ sorting!

- construction $O(n \log n)$
- storage (of the function) $O(n)$
- function computation $O(\log n)$
Motivation

- Can we guarantee a **worst-case** $O(1)$ time for hash table operations?

- **Yes** if the set of keys is **static**

- However, the construction time is **expected** $O(n)$ (Las Vegas algorithm)
Collisions: analysis

- What is the expected number of collisions?
  - i.e. number of pairs \( (k, l), k \neq l \) and \( h(k) = h(l) \)
  - \( X_{kl} = 1 \) iff \( h(k) = h(l) \)
  - \( E[\sum_{k \neq l} X_{kl}] = \sum_{k \neq l} E[X_{kl}] = \binom{n}{2} \frac{1}{m} \approx \frac{n^2}{2m} \)

- Remarks:
  - if \( m \approx n^2 \), then we have \( \frac{1}{2} \) expected collisions \( \Rightarrow \)
    \[ P[\text{collision}] \leq \frac{1}{2} \] (cf birthday paradox)
  - by iterating, we can build a hash table with NO collision after \( O(1) \) trials \textit{in expectation}
Perfect hashing

- Fredman, Komlós, Szemerédi (1984)
- Guarantees $O(1)$ worst-case time of LOOKUP for a static set of keys. Solution uses universal hashing.
- 2-level hash scheme:

  - Primary hash table of size $n$
  - LOOKUP: worst-case $O(1)$
Why $\sum n_i^2$ can be maid $\leq 2n$

- $\sum n_i^2 = n + \lbrack\text{nb of pairs which collide}\rbrack$
- $E[\text{nb of pairs which collide}] = 2 \cdot \frac{n^2}{2n} = n$

$\Rightarrow E[\sum n_i^2] = 2n \Rightarrow P[\sum n_i^2 > 4n] < 1/2$

by Markov inequality

Algorithm (sketch)

- hash to primary table of size $O(n)$ using a universal h.f. $h$
- if $\sum n_i^2 < 4n$ then hash each non-empty bucket to a table of size $n_i^2$; else rehash
- using a universal h.f. $h_i$; if collision, rehash until there is none (expected $O(1)$ time by birthday paradox)
Perfect hashing is practical!

- practical implementations exist, e.g. gperf in C++

- *dynamic* perfect hashing has also been studied, e.g. [Arbitman, Naor, Segev, FOCS 2010]
Minimal Perfect Hash Functions (MPHF)

- MPHF: bijective perfect hash function, i.e. $h: S \rightarrow [1..|S|]$
- enumeration of keys of $S$
- efficient construction algorithms of MPFH exist, see e.g. [https://blog.gopheracademy.com/advent-2017/mphf/](https://blog.gopheracademy.com/advent-2017/mphf/)
- space efficiency: a few bits per key
- lower bound $\approx 1.44$ bits/key, cf [Mehlhorn 82], [Belazzougui, Botelho, Dietzfelbinger 09]
Cuckoo hashing

(prefect hashing with open addressing)
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  
  ```
  pos ← h_1(x)
  loop
  if $T[pos]$ is empty then
    $T[pos] ← x$; return
  swap values of $x$ and $T[pos]$
  pos ← alternative position for $x$
  ```
Cuckoo hashing

- Introduced by Pagh & Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  
  pos $\leftarrow h_1(x)$
  loop
    if $T[pos]$ is empty then
      $T[pos] \leftarrow x$; return
    swap values of $x$ and $T[pos]$
    pos $\leftarrow$ alternative position for $x$
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  
  $pos \leftarrow h_1(x)$

  loop
  
  if $T[pos]$ is empty then
    
    $T[pos] \leftarrow x$; return
  
  swap values of $x$ and $T[pos]$

  $pos \leftarrow$ alternative position for $x$
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  
  ```
  pos ← h_1(x)
  loop
  \textbf{if} $T[pos]$ is empty \textbf{then}
  \quad $T[pos] ← x; \text{return}$
  \textbf{swap} values of $x$ and $T[pos]$
  pos ← alternative position for $x$
  ```
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  \[
  \text{pos} \leftarrow h_1(x)
  \]
  loop
    if $T[\text{pos}]$ is empty then
      $T[\text{pos}] \leftarrow x$; return
    swap values of $x$ and $T[\text{pos}]$
  pos $\leftarrow$ alternative position for $x$
Cuckoo hashing

- Introduced by Pagh & Rodler in 2001
- Uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  \[
  \text{pos} \leftarrow h_1(x)
  \]
  \[
  \text{loop}
  \]
  \[
  \text{if } T[\text{pos}] \text{ is empty then}
  \]
  \[
  T[\text{pos}] \leftarrow x; \text{return}
  \]
  \[
  \text{swap values of } x \text{ and } T[\text{pos}]
  \]
  \[
  \text{pos} \leftarrow \text{alternative position for } x
  \]
Cuckoo hashing

- Introduced by Pagh & Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  
  $pos \leftarrow h_1(x)$
  
  loop
    
    if $T[pos]$ is empty then
      $T[pos] \leftarrow x$; return
    
    swap values of $x$ and $T[pos]$

  $pos \leftarrow$ alternative position for $x$
Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions $h_1$ and $h_2$
- LOOKUP($x$): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT($x$):
  \[
  \text{pos} \leftarrow h_1(x)
  \]
  loop $n$ times
  \[
  \text{if } T[\text{pos}] \text{ is empty then}
  \]
  \[
  T[\text{pos}] \leftarrow x; \text{return}
  \]
  swap values of $x$ and $T[\text{pos}]
  \]
  \[
  \text{pos} \leftarrow \text{alternative position for } x
  \]
  rehash from scratch
Cost of insertions: How many iterations?

- Random graphs:
  - nodes = buckets \(m\)
  - edges = inserted elements \(n\)

- If \(n < m/2\), the graph contains very small connected components without cycles w.h.p.

- *Theorem*: if \(n < \frac{m}{2(1+\delta)}\), then \(P[\text{shortest path from } i \text{ to } j \text{ is of length } d] \leq \frac{1}{m} (1 + \delta)^{-d}\)

\[ \Rightarrow P[j \text{ is accessible from } i] = O\left(\frac{1}{m}\right) \]

\[ \Rightarrow \text{number of accessible nodes is } n \cdot O\left(\frac{1}{m}\right) = O(1) \]
Proof of the Theorem

- $d = 1$
  - $P[\text{an edge connects } i \text{ and } j] \leq \frac{2}{m^2} \Rightarrow P[\text{exists an edge between } i \text{ and } j] \leq \frac{2n}{m^2} < \frac{1}{m} (1 + \delta)^{-1}$

- induction: $d \Rightarrow d + 1$
  - for a fixed $k$, $\frac{1}{m} (1 + \delta)^{-d} \cdot \frac{1}{m} (1 + \delta)^{-1} = \frac{1}{m^2} (1 + \delta)^{-(d+1)}$
  - summing over $m$ possibilities for $k$, we obtain $\frac{1}{m} (1 + \delta)^{-(d+1)}$
Cost of rehashing: how likely is a cycle?

- If \( n < \frac{m}{2(1+\delta)} \), careful analysis shows that the probability of a rehash is \( O \left( \frac{1}{n^2} \right) \).
- A rehash involves \( n \) insertions, each taking expected \( O(1) \) time \( \Rightarrow \) amortized cost of rehashing is \( O \left( \frac{1}{n} \right) \) time per insertion.
- For more details see
Cuckoo hashing: summary

- **LOOKUP, DELETE:** worst-case $O(1)$ (two probes)
- **INSERT:** expected $O(1)$

- deletions supported
- no dynamic memory allocation (as in chaining)
- reasonable memory use, but load factor $< 1/2$
- generalization: $(H, b)$-cuckoo hashing
  - $H$ hash functions (instead of 2), each bucket carries $b$ items
  - admits a much higher load factor, e.g. (3,4)-tables admits load factor of over 99.9% [Walzer, ICALP 2018]
Load for $h > 2$ or $b > 1$

$b = 1$

<table>
<thead>
<tr>
<th>$h$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>0.5</td>
<td>0.918</td>
<td>0.976</td>
<td>0.992</td>
<td>0.997</td>
<td>0.999</td>
</tr>
</tbody>
</table>

$h = 2$

<table>
<thead>
<tr>
<th>$b$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>0.5</td>
<td>0.897</td>
<td>0.959</td>
<td>0.980</td>
<td>0.989</td>
<td>0.997</td>
<td>0.999</td>
</tr>
</tbody>
</table>
Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes

by Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia

11 Jan 2018

There's recently been a lot of excitement about a new proposal from authors at Google: to replace conventional indexing data structures like B-trees and hash maps by instead fitting a neural network to the dataset. The paper compares such learned indexes against several standard data structures and reports promising results. For range searches, the authors report up 3.2x speedups over B-trees while using 9x less memory, and for point lookups, the authors report up to 80% reduction of hash table memory overhead while maintaining a similar query time.

While learned indexes are an exciting idea for many reasons (e.g., they could enable self-tuning databases), there is a long literature of other optimized data structures to consider, so naturally researchers have been trying to see whether these can do better. For example, Thomas Neumann posted about using spline interpolation in a B-tree for range search and showed that this easy-to-implement strategy can be competitive with learned indexes. In this post, we examine a second use case in the paper: memory-efficient hash tables. We show that for this problem, a simple and beautiful data structure, the cuckoo hash, can achieve 5-20x less space overhead than learned indexes, and that it can be surprisingly fast on modern hardware, running nearly 2x faster. These results are interesting because the cuckoo hash is asymptotically better than simpler hash tables at load balancing, and thus makes optimizing the hash function using machine learning less important: it's always great to see cases where beautiful theory produces great results in practice.

Going Cuckoo for Fast Hash Tables

Let's start by understanding the hashing use case in the learned indexes paper. A typical hash function distributes keys randomly across the slots in a hash table, causing some slots to be empty, while others have collisions, which require some form of chaining of items. If lots of memory is available, this is not a problem: simply create many more slots than there are keys in the table (say, 2x) and collisions will be rare. However, if memory is scarce, heavily loaded tables will result in slower lookups due to more chaining. The authors show that, by learning a hash function that spreads the input keys more evenly throughout
Bloom filters

Approximate membership data structures
Bloom filters: generalities

- Bloom (1970)
- generalizes the bitmap representation of sets
- *approximate membership data structure*: supports INSERT and LOOKUP
- LOOKUP only checks for the presence, no satellite data
- produces false positives (with low probability)
- cannot iterate over the elements of the set
- DELETE is not supported (in the basic variant)
- very space efficient, keys themselves are *not* stored
- *Example*: forbidden passwords
Bloom filter: how it works

- $U$: universe of possible elements
- $K$: subset of elements, $|K| = n$
- $m$: size of allocated bit array
- Define $d$ hash functions $h_1, \ldots, h_d: U \rightarrow \{0, \ldots, m - 1\}$
- $\text{INSERT}(k)$: set $h_i(k) = 1$ for all $i$
- $\text{LOOKUP}(k)$: check $h_i(k) = 1$ for all $i$
- false positives but no false negatives
Bloom filters: analysis

- $P[\text{specific bit of filter is 0}] = (1 - 1/m)^{dn} \approx e^{-dn/m} \equiv p$

- $P[\text{false positive}] = (1 - p)^d = (1 - e^{-dn/m})^d$

- Optimal number $d$ of hash functions: $d = \ln 2 \cdot \frac{m}{n} \approx 0.693 \cdot \frac{m}{n}$

- Therefore, for the optimal number of hash functions,

  $$P[\text{false positive}] = 2^{-\ln 2 \cdot \frac{m}{n}} \approx 0.6185 \frac{m}{n}$$

- E.g. with 10 bits per element, $P[\text{false positive}]$ is less than 1%

- To insure the FP rate $\varepsilon$: $m = \log_2 e \cdot n \cdot \log_2 \frac{1}{\varepsilon} \approx 1.44 \cdot n \cdot \log_2 \frac{1}{\varepsilon}$
Dependence on the number of hash functions

$$\text{Opt } d = 8 \ln 2 = 5.45 \ldots$$

- $m/n = 8$
- $n$ elements
- $m$ bits
- $d$ hash functions
Lower bound on the size of approximate membership data structures (AMD)

- Bloom filter takes $1.44 \cdot \log \frac{1}{\varepsilon}$ bits per key, is this optimal?

- How many AMDs are there to store all sets of size $n$ drawn from universe $U$ with FPP $\varepsilon$?

- Each AMD specifies a set of size $\varepsilon|U|$ (assuming $|U|$ large) containing a set of size $n$

- Any set of size $n$ should be covered, and the number of such sets is $\geq \left( \frac{|U|}{n} \right) / \left( \frac{\varepsilon|U|}{n} \right) \approx \left( \frac{1}{\varepsilon} \right)^n$ (cf Erdős&Spencer 74, Rödl 85)

- $\Rightarrow$ each FPP must take $\geq n \cdot \log \frac{1}{\varepsilon}$ bits
Bloom filter: properties/operations

- For the optimal number of hash function, about a half of the bits is 1 [immediate from the formula]
- The Bloom filter for the union is the OR of the Bloom filters
- Is similar true for the intersection? [explain]
- If a Bloom filter is sparse, it is easy to halve its size
Bloom filters: applications

- Bloom filters are very easy to implement
- Used e.g. for
  - spell-checkers (in early UNIX-systems)
  - unsuitable passwords, "approximate" unsuitable passwords (Manber&Wu 1994)
  - online applications (traffic monitoring, …)
  - distributed databases
  - malicious sites in Google Chrome
  - read articles in publishing systems (Medium)
  - Google Bigtable, Apache HBase, Bitcoin, bioinformatics, …
- Sometimes (when the set of possible queries is limited) it is possible to store the set of false positives in a separate data structure
Cuckoo filters

filters via Cuckoo hashing
Filters via MPHF

- Given a set \( K \subset U \), build an MPHF \( h: K \rightarrow [1..n] \)
- Given \( \varepsilon \), pick a hash function \( f \) mapping keys of \( K \) into fingerprints of \( \log \frac{1}{\varepsilon} \) bits
- Build an array \( F \) of fingerprints: \( F[h(k)] = f(k) \)

\[
P[f(x) = f(y)] = \frac{1}{2^{1 \log \frac{1}{\varepsilon}}} = \varepsilon \text{ (false positive proba)}
\]

- Space: \( n \cdot \log \frac{1}{\varepsilon} + \text{<size of MPFR>} \)
- lower bound: \( \text{size of MPFR} \geq 1.44n \)
- \( K \) must be static, does not support insertions/deletions
Cuckoo filter: ideas

- Use Cuckoo hash table (e.g. (2,4)-table) instead of MPHF

**Problem**: How to move a fingerprint? i.e. how to know its alternative bucket?
Cuckoo filter: ideas

- Use Cuckoo hash table (e.g. (2,4)-table) instead of MPHF

- **Problem:** How to move a fingerprint? i.e. how to know its alternative bucket?

  \[
  h_1: K \rightarrow 2^{\log |T|}, \quad h_2: 2^{\log_{1/\varepsilon}^{1}} \rightarrow 2^{\log |T|}
  \]

  location 1: \( h_1(k) \)

  location 2: \( h_1(k) \oplus h_2(f(k)) \)

- Alternative location of a fingerprint \( \alpha \) at location \( i \) is \( i \oplus h_2(\alpha) \)
Remarks

- Two locations of a key are not fully independent. E.g. two keys sharing the same bucket and the same fingerprint have the same alternative location. ($\Rightarrow$ store multisets in $b$-element buckets)

- Practical: Cuckoo vs. Bloom: for small false positive rate ($< 3\%$) and $b = 4$, Cuckoo filter achieves the same performance as Bloom with smaller space

![Figure 4: False positive rate vs. space cost per element. For low false positive rates ($< 3\%$), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.][Fan et al. Cuckoo filter: practically better than Bloom, CoNEXT 2014]