An algorithms engineering view of Hashing Schemes and Related Topics

Andrei Broder
Systems Research Center, Palo Alto
broder@pa.dec.com
November 19, 1997

digital

Engineering

Engineering is the professional art of applying science to the optimum conversion of the resources of nature to the uses of humankind.

[Britannica]

digital

Engineering

An engineer is a man who can do for a dime what any damn fool can do for a dollar.

[Nevil Shute]

Algorithms Engineering

The art and science of crafting cost-efficient algorithms.

Some good books...

- Textbook: R. Sedgewick, Algorithms in C, 3rd ed, 1997.
- More C: D. Hanson, C Interfaces and Implementations, 1997.
- Math bottom line + references & timings: R. Baeza-Yates & G. Gonnet, Handbook of algorithms and Data Structures, 2nd ed, 1991.
- THE BOOK on analysis of algorithms: Knuth, Art of Computer Programming. Vol 1, 3rd ed, 1997, Vol 3, 1973.

Analysis vs. engineering

Analysis of algorithms

- Abstract models
  - RAM, Turing machine, Finite Automatons, Pointer machines
- Cost models
  - Comparison model, random bits, competitive ratio, memory
- Asymptotic analysis
- Worst case/Average case
- Optimal results

Algorithms engineering

- Real life components.
  - virtual memory, cache structure, branch prediction ...
- Dollars and cents
  - Reliability, safety factors, recovery, usability,...
- Given range of inputs
- Input specs
- Performance specs

Plan

- Introduction
- Standard hashing schemes
- Choosing the hash function
  - Universal hashing
- Fingerprinting
- Bloom filters
- Perfect hashing
- Real life considerations

Dictionaries (Symbol tables)

- Dictionaries are data structures for manipulating sets of data items of the form

```
item = [key, info]
```

- Usually (but not always) the keys are unique.

Some examples of dictionaries

- Rolodex
  - Hash function: first letter
  - Supports insertions, deletions, adaptation.
- Spelling dictionary
  - System word list is fixed.
  - Personal word list allows additions.
  - Issues: Average case must be very fast, errors allowed, nearest neighbor searches.

More examples

- Router
  - Translate destination into wire number.
  - Insertions and deletions are rare.
  - Strict limit on the worst case.
- Visited pages
  - Frequent insertions.
  - Deletions on expiration date (batch).
  - Keys are long.
- Compiler symbol table
  - Fixed list of reserved words used very often.
  - Variable names added & deleted.
  - Must support scope rules.

Basic operations

`item = [key, info];` Given the `item` the `key` can be extracted or computed.

- `Insert(item);`
- `Delete(item);`
- `Search(key);` (returns `item`)

More operations

- `Init(…);`
- `InitFromData(…);`
- `Exists(key);` (returns `Boolean`)
- `List(…); Sort(…); Iterate(…);` (return the entire list unordered/ordered/one-at-a-time);
- `Join(…);` (combine two structures);
- `Nearest(key);` (returns `item`)

For our examples

- Rolodex
  - `Insert; Delete; Search;`
  - ? `Exists; List; Iterate; Join; Nearest;`
- Spelling dictionary (system)
  - `Exists`
- Router
  - `Insert; Delete; Search;`
- Visited pages
  - `Insert; Delete; Search;`
  - ? `Exists; List; Join;`
- Compiler symbol table (reserved words)
  - `Search; Exists;`

Implementing dictionaries

- Schemes based on key comparison - keys viewed as elements of arbitrary total order
  - Ordered list
  - Binary search trees
- Schemes based on arithmetic on keys - there is a key $\Rightarrow$ address map.
  - Hashing
  - Bloom filters
- Hybrid schemes
  - Interpolation search.
  - Radix search.

Hashing schemes - basics

We want to store N items in a table of size M,

at a location computed from the key K.

Two main aspects:

- Hash function
  - Method for computing table index from key
- Collision resolution strategy
  - How to handle two keys that hash to the same index

Hash functions

- Simple choice:
  - Table size `M`
  - Hash function `h(K) = K mod M;`
- Works fine if keys are random integers.
- Example: 20 random keys in [1..100]

```
[56, 82, 87, 39, 98, 86, 69, 22, 99, 61,
 64, 50, 77, 75, 8, 62, 17, 10, 71, 58]
```

- …hashed in a table of size 20

```
[16, 2, 7, 19, 18, 6, 9, 2, 19, 1, 4, 10,
 17, 15, 8, 2, 17, 10, 11, 18]
```


Why do collisions happen?

- Birthday paradox: expected number of random insertions until the first collision is only

  sqrt(π M)

- Examples:

M = 100	sqrt(π M) ~ 12
M = 1000	sqrt(π M) ~ 40
M = 10000	sqrt(π M) ~ 125

Separate chaining

- Basic method: keep a linked list for each table slot.
- Advantages:
  - Simple, widely used (maintainability)
- Disadvantages:
  - Wastes space, must deal with memory allocation.

Example

- Input:

  `[56, 82, 87, 39, 98, 86, 69, 22, 99, 61, 64, 50, 77, 75, 8, 62, 17, 10, 71, 58]`

- Hash table:

```
0: [50, 10]         5: [75]
1: [61, 71]         6: [56, 86]
2: [82, 22, 62]     7: [87, 77, 17]
3: []               8: [98, 8, 58]
4: [64]             9: [39, 69, 99]
```


Performance

- Insert cost: `1`
- Average search cost (hit): `1+N/(2 M)`
- Average search cost (miss): `1+N/M`
- Worst case search cost: `N+1`
- Expected worst case search cost (`n=m`):

  `~log n/log log n`
- Space requirements:
  - `(N + M) * |link| + N*|key| + N*|info|`
- Deletions: easy
- Adaptation (new hash function): easy

Embellishments

- Keep lists sorted:
  - Average insert cost: `1+N/(2 M)`
  - Average search cost (hit): `1+N/(2 M)`
  - Average search cost (miss): `1+N/(2 M)`
- Move-to-front / transpose
  - Last item accessed in a list becomes the first or moves one closer (Self adjusting hashing)
- Store lists as a binary search tree:
  - Improves expected worst case
- Assume key can be reconstructed (in part) from hash value -- don't store full key!

Saving space

- Example: store only `K div 10` rather than `K`
- Input:
  `[56, 82, 87, 39, 98, 86, 69, 22, 99, 61, 64, 50, 77, 75, 8, 62, 17, 10, 71, 58]`
- Hash table:

`0: [50, 10]`	`5: [75]`
`1: [61, 71]`	`6: [56, 86]`
`2: [82, 22, 62]`	`7: [87, 77, 17]`
`3: []`	`8: [98, 08, 58]`
`4: [64]`	`9: [39, 69, 99]`

Performance

- Old space requirements:
  `(N + M) * |link| + N*|key| + N*|info|`
- New space requirements:
  `(N + M) * |link| + N*|key| + N*|info| - N*log(M)`
  `~ M * |link| + N*|key| + N*|info|`
- Wasted space: `M * |link|`

Class problem

- We need to maintain a very large hash table via separate chaining.
- Sometimes, after many insertions and deletions we get into a bad configuration and must change hash function.
- The table is so large that
  - it cannot be duplicated
  - we must allow searches and inserts even while the hash function is being changed.
  - ...but we can disable searches and inserts for short periods.

Question

- How can we modify the basic structure to allow for hash function change?
- Explain how insertions and searches can be done during the change.
- What is the new cost of insertion, deletions, searches? (At regular times and during the change.)
- If you can think of more than one way to do this, discuss tradeoffs.
- Extra credit: can you deal with deletions during change?

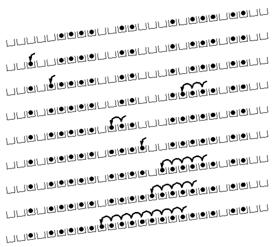
Open addressing

- No links, all keys are in the table.
- When searching for `K`, check locations $r_1(K), r_2(K), r_3(K), \ldots$ until either
  - `K` is found; or
  - we find an empty location (`K` not present)
- Various flavors of open addressing differ in which probe sequence they use.
- Random probing -- each r_i is random. (Impractical)

Linear probing

- When searching for `K`, check locations `h(K), h(K)+1, h(K)+2,` … until either
  - `K` is found; or
  - we find an empty location (`K` not present)
- If table is very sparse, almost like separate chaining.
- When table starts filling, we get clustering but still constant average search time.
- Full table $\Rightarrow$ infinite loop.

Linear probing -- clustering



[R. Sedgewick]

Performance

- Load α = M/N
- Average search cost (hit) ~
$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$
- Average search cost (miss) ~
$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$
- Very delicate math analysis.
- Don’t use α above 0.8 .

Performance

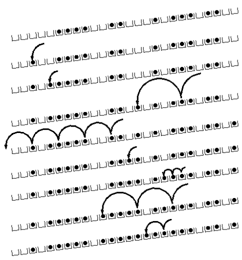
- Expected worst case search cost:
  `O(log n)`
- Space requirements:
  - **`M*(|key| + |info|)`**
- Deletions:
  - By marking
  - By deleting the item and reinserting all items in the chain.

Double hashing

- When searching for **K**, check locations **h_1(K), h_1(K)+ h_2(K), h_1(K)+2*h_2(K),…** until either
  - **K** is found; or
  - we find an empty location (**K** not present)
- Must be careful about **h_2(K)**
  - Not 0.
  - Not a divisor of **M**.
- Almost as good as random probing.
- Very difficult analysis.

Hashing schemes
Andrei Broder, 11/19/97

Double hashing



[R. Sedgewick]

Hashing schemes
Andrei Broder, 11/19/97

Performance

- Load α = M/N
- Average cost (hit) ~

$$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

- Average cost (miss/insert) ~

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

- Don't use α above 0.95 .

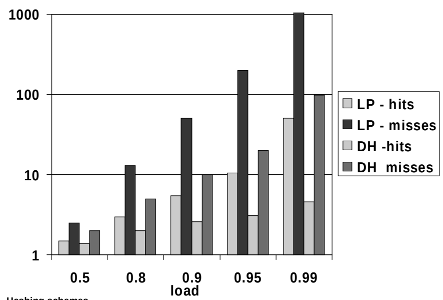
Hashing schemes
Andrei Broder, 11/19/97

Performance

- Expected worst case search cost:

  `O(log n)`
- Space requirements:
  - `M*(|key| + |info|)`
- Deletions:
  - Only by marking.

Open addressing performance



Rules of thumb

- Separate chaining is idiot-proof but wastes space…
- Linear probing uses space better, is fast when tables are sparse, interacts well with paging, …
- Double hashing is very space efficient, quite fast (get initial hash and increment at the same time), needs careful implementation, ...

Choosing the hash function

- Want function to seem random.
- Choice depends on key type and acceptable cost.
- Often machine dependent.
- Sometime deliberate overflow.
- Key = small integer
  - For **M** prime **h(K) = K mod M**;
  - For M non-prime

```
h(K) = floor(0.616161 * K) mod M;
h(K) = (16161*K mod 2^32) mod M;
```

Hashing schemes
Andrei Broder, 11/19/97
37

More hash functions

- Key = real in [0,1]
  - For any M

```
h(K) = floor(K*M);
```

- Key = string - Horner's rule

```
int h(char *s)
  {int j;
   for (j=0; *s; s++) j=131*j + *s;
   return (j % M)
  }
```

Hashing schemes
Andrei Broder, 11/19/97
38

Other

- Key = string of bits
  - Polynomial arithmetic. View **K** as polynomial **K(x)** with coefficients in Z^2
  - Compute

$$K(x)*x^d \bmod P(x)$$

    for irreducible **P** of degree **d.**
  - Very fast in hardware.
  - In software

```
(1 xor + 1 index)/byte
```

Hashing schemes
Andrei Broder, 11/19/97
39

Caveats

- Hash functions are very often the cause of performance bugs.
- Hash functions often make the code not portable.
- Sometime a poor HF distribution-wise is faster overall.
- Always check where the time goes.

Universal hashing

- Don't use a fixed hash function; for every run choose a function from a small family.
- Example:

  `h(K) = (a*K + b) mod M`

`a` and **`b`** chosen u.a.r. in **`[1..M]`** and **`M`** prime

- Main property

  `Pr(h(K1)=h(K2)) = 1/M`

Properties

- Theory:
  - We make no assumptions about input. All proofs are valid wrt our random choices.
- Practice:
  - If one choice of **a** and **b** turns out to be bad, make a new choice.
  - Must use hash schemes that allow re-hashing.
  - Useful in critical applications.

Fingerprinting

- Fingerprints are short tags for larger objects.
- Notations

  $\Omega =$ The set of all objects

  $k =$ The lenght of the fingerprint

  $f : \Omega \rightarrow \{0,1\}^k$ A fingerprinting function
- Properties

$$f(A) \neq f(B) \Rightarrow A \neq B$$

$$\Pr\left(f(A) = f(B) \middle| A \neq B\right) \approx \frac{1}{2^k}$$

Why fingerprint?

- Probability is wrt our choice of a fpr scheme.
  - Don't need assumption about input.
- Keys are long or there are no keys.
  - In AltaVista 100M urls @ 90 bytes/url = 9GB
    100M fprs @ 8 byte/fpr = 0.8GB
  - Find duplicate pages -- two pages are the same if they have the same fpr.

Fingerprinting schemes

- Cryptographically secure:
  - MD2, MD4, MD5, SHS, etc
  - relatively slow
- Rabin's scheme
  - Based on polynomial arithmetic
  - Very fast (1 table lookup + 1 xor + 1 shift) /byte
  - Nice extra-properties

Rabin's scheme

- View each string A as a polynomial over Z_2:
  `A = 1 0 0 1 1` $\Rightarrow$ `A(x) =` x^4 `+ x + 1`
- Let `P(t)` be an irreducible polynomial of degree `k` chosen uar
- The fingerprint of `A` is
  `f(A) = A(t) mod P(t)`
- The probability of collision among `n` strings of average length `t` is about
  `n^2 t / 2^k`

Nice extra properties

- Let ♦ = catenation. Then
  f(a ♦ b) = f(f(a) ♦ b)
- Can compute extensions of strings easily.
- Can compute fprs of sliding windows.
- Other applications:
  - 2-D pattern matching.
  - Type-checking in Modula-3
  - Test set equality

Bloom filters

- Want to check only existence of key (e.g. spelling dictionary, stolen credit cards, etc)
- Small probability of error is OK.
- Simple solution:
  - Keep bit-table **B**
  - For each **K** turn **B(h(K))** on;
  - Say K is in iff **B(h(K))** is on;
  - Works if there are no collisions! Must have
    N = O(sqrt(M))
  - Collisions generate false drops

Better solution:

- Use r hash functions.
  - For each `K` turn on
    `B(h`$_1$`(K), B(h`$_2$`(K)),…,B(h`$_r$`(K))`
  - Say `K` is in iff all hash bits are on.
  - Probability of false drop is

$$(1-\exp(-r\alpha))^r$$

  - Optimum choice for r is

$$r = \alpha \ln(2)$$

  - With this choice, probability of false drop

$$0.6185^{1/\alpha}$$

Hashing schemes
Andrei Broder, 11/19/97
49

Example

- **`/usr/dict/words`** -- about **`210KB`**, **`25K`** words
- Use **`30KB`** table
- Load = **`25/(30*8) ~ 0.104`**
- Optimum **`r = 7`**
- Probability of false drop
  - **`1%`** for **`r =7`**
  - **`1.3%`** for **`r=4`**

Hashing schemes
Andrei Broder, 11/19/97
50

Perfect hashing

- The set of keys is given and never changes.
- Find “simple” hash function so that there are no collisions.
- Example: reserved words in a compiler.
- Hard to do but can be very useful.
- Example **`(M ~ 6N)`**
  `(a K mod b) mod M`
- Takes time **`O(`**n^3 **`log n)`** to compute.

Hashing schemes
Andrei Broder, 11/19/97
51

Other real life considerations

- Interaction with memory system: caching or (shudder!) paging.
- Branch prediction.
- Pipeline stalls.
- For ultimate performance:
  - Figure out the math
  - Implement carefully
  - Use measurement tools
    - gprof, pixie, dcpi, atom, etc
  - Repeat

Summary

- Basic hashing schemes
  - Separate chaining
  - Open adressing
    - Linear probing
    - Double hashing
- Choosing the hash function
- Fingerprinting
- Bloom filters

Conclusion

As always...

Understand the problem, know the possibilities, and choose the right solution: it will make a difference.

Digital Equipment Corporation
© 1997