CLOUDFLARE™

Rolling and Extendable Hashes

December 18, 2013

John Graham-Cumming

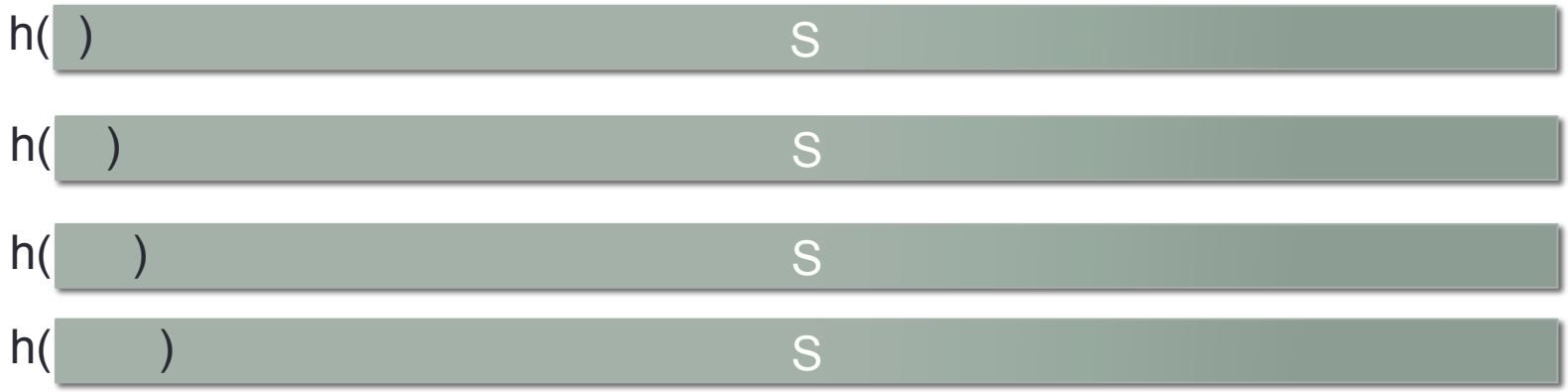
CRC

Cyclic Redundancy Check

- Widely used as hash function
  - e.g. common to hash keys uses a CRC for load balancing
  - memcached, nginx, ...
  - crc(key) % #servers
- Not designed as a hash function!
- Choice of polynomial can change collisions in real world
- CRCs have no internal state and they are fast
  - Shifts, XOR, small table lookup

Extending a CRC

Compute the hash (a CRC) by extending the string to the right



Typical CRC allows this because when adding a bit/byte/word the CRC is calculated from the previous CRC and the added data.

CRC64

- Typical implementation (t is lookup table)

```
BIT64 crc = 0;

while ( *s ) {
  BYTE c = *s++;
  crc = (crc >> 8) ^ t[(crc & 0xff) ^ c];
}
```

- Some common polynomials:
  - CRC64-ISO (0x800000000000000D) $x^{64}+x^4+x^3+x+1$
  - CRC64-ECMA (0xA17870F5D4F51B49)

$$x^{64}+x^{63}+x^{61}+x^{59}+x^{56}+x^{55}+x^{52}+x^{49}+x^{48}+x^{47}+x^{46}+x^{44}+$$
$$x^{41}+x^{37}+x^{36}+x^{34}+x^{32}+x^{31}+x^{28}+x^{26}+x^{23}+x^{22}+x^{19}+x^{16}+$$
$$x^{13}+x^{12}+x^{10}+x^8+x^7+x^5+x^3+x^1$$

One use: spam filter tokens

- Common to see in open source spam filters

```
From: Prince of Nigeria
Subject: I beg pardon for interrupting your fine day

[...]
```

- Generates tokens:

```
from:prince from:nigeria subject:pardon subject:interrupting
subject:fine subject:day
```

- And from the message body

```
body:html:font:red body:html:image:size:0
```

- Repeated fixed parts (from:, subject:, html:image:size:) can be CRCed once

RABIN/KARP

Rabin/Karp String Searching

"Efficient randomized pattern-matching algorithms"
Rabin/Karp, IBM Journal of Research and Development
March 1987

Compute and compare len(S)-len(P)+2 hashes

When hash matches verify that substrings actually match

Two Expensive Operations

- The h() function itself
  - Need a very fast hash algorithm
  - Average complexity is O(len(S))
- Substring comparison on hash matches
  - Need minimal hash collisions
  - Worst case complexity is O(len(S) * len(P))
- Trivial to make work with multiple patterns
  - Just compute their hashes
  - Use Bloom filter (or other hash table) to lookup patterns for each hash calculated
  - Average complexity is O(len(S) + #patterns)

Rabin/Karp Rolling Hash

- Basic on arithmetic in the ring Z_p where p is prime
- Treats a substring s_0 .. s_n as a number in a chosen base b.

$$h(s_0 \ldots s_n) = \sum_{i=0}^{n} s_i b^{n-i} \bmod p$$

- This hash can be updated when sliding across a string

Rabin/Karp Efficient Update

$$h(s_1 \ldots s_{n+1})$$

$$= \sum_{i=1}^{n+1} s_i b^{n+1-i} \bmod p$$

$$= b \sum_{i=1}^{n+1} s_i b^{n-i} \bmod p$$

$$= \left(b \sum_{i=0}^{n} s_i b^{n-i} - s_0 b^n + s_{n+1} \right) \bmod p$$

Previous hash

Precomputed

$$= \left(b h(s_0 \ldots s_{n-1}) - s_0 b^n + s_{n+1} \right) \bmod p$$

$$= \left(b \left(h(s_0 \ldots s_{n-1}) - s_0 b^{n-1} \right) + s_{n+1} \right) \bmod p$$

Rabin/Karp base and modulus values

- Use a prime base
  - When operating on bytes a suitable b is 257
- Choose a prime modulus closest to the desired hash output size
  - e.g. for 32-bit hash value use 4294967291
- Various tricks can speed up the calculation
  - Don't calculate b^{n-1}, calculate b^{n-1} mod p instead
  - Choose a p such that bp fits in a convenient bit size
  - Memoize x (b^{n-1} mod p) for all x byte values
  - Choose p is a power of two (instead of a prime) can use & instead of %
  - Result: update requires one *, one +, one –, one &

Bentley/McIlroy Compression

"Data Compression Using Long Common Strings"
Jon Bentley, Douglas McIlroy
Proceedings of the IEEE Data Compression Conference, 1999

"Dictionary" to compress against (could be self)
Broken into blocks, each block hashed

Slide Rabin/Karp hash across string to compress
Look for matches in dictionary
Extend matches backwards block size bytes, forwards indefinitely
O(len(S))

Never gonna give you up

```
We're no strangers to love
You know the rules and so do I
A full commitment's what I'm thinking of
You wouldn't get this from any other guy
I just wanna tell you how I'm feeling
Gotta make you understand
Never gonna give you up<204,13>let you down<204,13>run around<45,5>desert you<20
4,13><184,9>cry<204,13>say goodbye<204,13>tell a lie<45,5>hu<285,7>
We've<30,5>n each<129,7>for so long
Your heart's been aching but
You're too shy to say it
Inside we both<30,6>wha<422,9>going on
We<30,10>game<45,5>we're<210,7>play it
And if you ask me<161,17>Don't tell m<220,5>'re too blind to see
<340,13><217,161><229,12><634,161>(Ooh,<633,12>)<967,24>)<340,13>give, n<230,11>
give
(G<635,10><1004,57><377,11><389,160><139,239><229,12><634,333>
```

- Open source implementation from CloudFlare in Go
- Bentley/McIlroy then gzip very effective

RSYNC PROTOCOL

RSYNC hashing

"Efficient Algorithms for Sorting and Synchronization"
Andrew Tridgell
Doctoral Thesis, February 1999

File on remote machine
Broken into blocks, each block hashed twice (rolling hash plus MD5)
Hash sent to local machine

Slide rolling hash across source file looking for match
Check MD5 hash to eliminate rolling hash collisions
Send bytes form S to remote host, or token saying which hash matched

The RSYNC rolling hash

- Block size is b, offset into file is k, the file bytes are s, M is an arbitrary modulus

$$r_1(k,b) = \left(\sum_{i=0}^{b-1} a_{i+k} \right) \bmod M$$

$$r_2(k,b) = \left(\sum_{i=0}^{b-1} (b-i) a_{i+k} \right) \bmod M$$

$$r(k,b) = r_1(k,b) + M r_2(k,b)$$

- In practice, M = 2^{16}

Hash updates are fast

$$r_1(k+1,b) = \left(r_1(k,b) - a_k + a_{k+b}\right) \bmod M$$

$$r_2(k+1,b) = \left(r_2(k,b) - ba_k + r_1(k+1,b)\right) \bmod M$$

(* proof is left as an exercise for the reader)