GigaHash: Scalable Minimal Perfect Hashing for Billions of URLs

Kumar Chellapilla
Microsoft Live Labs
One Microsoft Way
Redmond, WA 98052 USA
kumarc@microsoft.com

Anton Mityagin
Microsoft Live Labs
One Microsoft Way
Redmond, WA 98052 USA
mityagin@microsoft.com

Denis Charles
Microsoft Live Labs
One Microsoft Way
Redmond, WA 98052 USA
cdx@microsoft.com

ABSTRACT

A minimal perfect function maps a static set of n keys on to the range of integers $\{0,1,2,\dots,n-1\}$. We present a scalable high performance algorithm based on random graphs for constructing minimal perfect hash functions (MPHFs). For a set of n keys, our algorithm outputs a description of h in expected time $O(n)$. The evaluation of $h(x)$ requires three memory accesses for any key x and the description of h takes up $0.89n$ bytes ($7.13n$ bits). This is the best (most space efficient) known result to date. Using a simple heuristic and Huffman coding, the space requirement is further reduced to $0.79n$ bytes ($6.86n$ bits). We present a high performance architecture that is easy to parallelize and scales well to very large data sets encountered in internet search applications. Experimental results on a one billion URL dataset obtained from Live Search crawl data, show that the proposed algorithm (a) finds an MPHF for one billion URLs in less than 4 minutes, and (b) requires only 6.86 bits/key for the description of h.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Experimentation, Performance.

Keywords

Minimal perfect hashing, perfect hash function, Web search engine, space efficient hash table

1. INTRODUCTION

A hash function maps elements from an input space to a finite range of integers. Typically, the range of the hash function is much smaller than the input space. Thus a hash function is not *injective*. However, for a subset of the input space that is smaller than the range, the hash function has few collisions. In particular, if a hash function is drawn from a 2-universal family of hash functions that map to a set of size m, then with high probability a hash function will be collision free if it is used to map $\leq \sqrt{m}$ keys (see [1]).

A *perfect* hash function maps a *static* set of n keys into a set of m integer numbers without collisions, where $m \geq n$. If $m = n$, the hash function is called *minimal*. Usually, the range of the minimal perfect hash function (MPHF) is the contiguous set of integers $\{0,1,2,\dots,n-1\}$. One point to note is that while general hash functions deal with a *dynamic* set of keys, perfect hash functions always work with a *static* set of keys.

MPHFs have many applications (information retrieval systems, database systems, hypertext, hypermedia, language translation systems, electronic commerce systems, compilers, operating systems, among others) but we focus on their use in web search engines. Large scale web applications typically work with several billion URLs. For example, commercial search crawlers[1] encounter several tens of billions of URLs during crawling and index them for search and retrieval. However, URLs are of variable length and not suitable for efficient processing. So, rather than process URLs directly, they are first hashed to a fixed size data structure. All subsequent processing deals with the hashes. The hashing scheme has to be carefully designed to avoid collisions. Given 1 billion URLs, one can uniquely represent each URL using a 30-bit number ($2^{30} \approx 1$ billion). A simple universal hash would require twice as many bits to have a low probability of collision (see birthday paradox problem [1]), which puts the required number of bits per URL at 60. A 60- or 64-bit hash is a perfect hash with high probability. Some search related applications require the generated hashes to be contiguous i.e., the perfect hash function also needs to be minimal. For example, computing the PageRank for a web graph requires a mapping from URL space to a contiguous sequence of integers that represent rows and columns of the web graph adjacency matrix.

In this paper, we propose using MPHFs for very large static web datasets such as a static set of all URLs seen by crawler and indexed by a search engine. The evaluation of $h(x)$ requires three memory accesses for any key x and the description of h takes less than one byte per key.

2. MINIMAL HASHING ALGORITHM

2.1 Notation and Terminology

We use the following notation and terminology in this paper:

- U: universe of keys of size $|U| = u$.
- S: the static set of keys to be hashed. $S \subset U$.
- n: the number of static keys $|S| = n \ll u$.
- $h: U \to M$ is a hash function that maps keys from the universe U into a given range of integers $M = \{0,1,2,\dots,m-1\}$
- h is a *perfect* hash function if it is one-to-one on S, i.e., if $h(k_1) \neq h(k_2)$ for all $k_1 \neq k_2$, k_1, $k_2 \in S$. h is a *minimal* perfect hash function if it is one-to-one on S and $n = m$.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

[1] such as GoogleBot (Google), Yahoo! Slurp (Yahoo), MSNBot (Live), Ask.com/Teoma

2.2 The Minimal Hashing Scheme

Perfect hashing and minimal perfect hashing have enjoyed a rich history of development ever since the early days of computer science when hashing was introduced. However, our construction improves upon recent work of Czech et al. [2] and Botelho et al. [3]. These MPH algorithms are based on random graphs and have the following stages:

- *Mapping*: transform the key set, S, from the original universe U to a new universe U'.
- *Ordering*: place the keys in a sequence that determines the order in which hash values are assigned to keys.
- *Searching*: assign hash values to the keys in order.

The universe is mapped to the edges of a random acyclic graph $G = (V, E)$ with $|E| = |S|$: key k is mapped to edge $e = (h_1(k), h_2(k))$, where $h_1, h_2 : U \rightarrow V$ are 2-universal hash functions. Next, they pick an arbitrary isomorphism $i : E \rightarrow \{0, \dots, n-1\}$ and find a function $g : V \rightarrow \{0, \dots, n-1\}$ such that $i(e) = \{g(v_1) + g(v_2)\} \bmod n$. . Then the MPH is simply $H(k) := \{g(h_1(k)) + g(h_2(k))\} \bmod n$. Czech et al. [2], showed that if $|V| = c|E|$ with $c \geq 2.09$ then G obtained in this way is acyclic with sufficiently high probability. Since the graph is acyclic the set of equations for the g values can be solved very easily. The description of the hash function requires the table of g values to be stored and is proportional to c. The algorithm of Botelho et al. sets c to be < 2.09, and consequently the graph G is cyclic. They show how one can order the assignment of the hash values to the cyclic portion, the so-called 2-core, of the graph to be able to construct the MPH (see [3] for the details). Their method works as long as the 2-core is $\leq \frac{1}{2}|E|$. The size of the 2-core is known to be $\leq \frac{1}{2}|E|$as long as $c \geq 1.15$ thus leading to improvements.

Our algorithm begins with the observation that the distribution the g values produced is skewed. In particular, we analyze the probability that $g(v) = 0$. We use this to compress the storage requirements of the table of g values further. Theoretically, we prove that the compression allows us to reach a lower effective value of $c \geq 0.94$, which is 23% better. Empirically, on a one billion URL dataset, we show that c can be taken as low as 0.81, an improvement of 42% over $c = 1.15$.

2.3 Key Improvements

The following proposition allows us to analyze the proportion of vertices with $g(v) = 0$.

Proposition 1. The expected proportion of vertices v that have $g(v) = 0$ owing to the fact that they belong to a connected components of size ≤ 2 in G is

$$p_0 = \left(1 - \frac{2}{m}\right)^n + \frac{n}{m}\left(1 - \frac{2}{m}\right)^{2(n-1)}$$

One can use an n bit-vector B to identify unused/zero $g(v)$ values. Using a bit-vector the space required for the MPH drops from $cn \log n$ bits to $c(1 - p_0)n \log_2 n + p_0 n$ bits. For $n = 10^9$ and $m \approx 1.15n$, we have $p_0 \approx 0.211$, thus the effective $c = 0.94$. Empirically, one can reduce m down to $0.93n$, wherein $p_0 \approx 0.12$ and the effective c drops to 0.81. One can also exploit the non-uniform distribution of g values using Huffman coding.

2.4 Scalable High Performance Architecture

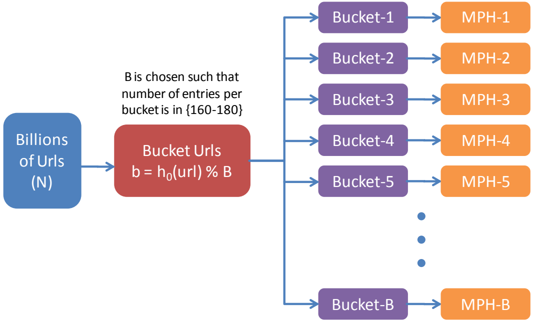

Figure 1. Architecture of our minimal perfect hash function

Figure 1 shows the high level organization of our minimal perfect hash (MPH) function for billions of URLs. Generation of a MPHF consists of the following steps:

1. Read the input data from a hard drive or network.
2. Convert URLs to fingerprint strings and distribute them into buckets. Text data is parsed and segmented into individual URLs. For each URL we compute 64-bit fingerprints using Jenkin's Hash. Based on the fingerprint, we determine a bucket ID and place the URL in that bucket. Bucket sizes are chosen so that they contain 160-180 URLs on average.
3. Create minimal perfect hash function (over the fingerprints) for each of the buckets. For each of the buckets we construct a minimal perfect hash function as described in Section 2.2 and 2.3. Each of the buckets is processed in parallel.
4. The MPHs for the buckets are "stitched" into a global MPH using a table of offsets for each bucket.

2.5 Results

In the segmentation step, one billion URLs were segmented into 6.25 million buckets. Each of the buckets was independently processed to obtain local MPHFs. Table 1 shows the space gains using a bit-vector and Hufman coding of the g table. The time to create the table was 3.9 min on an AMD Opteron 285 (dual processor) 64-bit machine with 16GB RAM. Hash lookups required roughly 0.025×10^{-6}sec, completing 1 billion lookups in 23.7 seconds.

Table 1. Space gains using zero/unused $g[.]$ bitvector and Huffman coding for hashing 1 billion URLs.

C	MPHF size (GB)	zero/ unused $g[.]$ values	Huffman Coded Size (GB)	Space gain (%)	Bits per URL	Bits per URL (Huff)
1.15	1.267	22.99 %	1.011	20.12	8.10	8.09
1.00	1.114	18.52 %	0.929	14.88	7.59	7.43
0.95	1.067	16.26 %	0.899	13.43	7.39	7.19
0.93	1.048	15.35 %	0.885	12.76	7.31	7.08
0.90	1.017	14.02 %	0.862	11.92	7.17	6.90
0.89	1.008	13.54 %	0.858	11.60	7.13	6.86

3. REFERENCES

[1] D. E. Knuth. The Art of Computer Programming: Sorting and Searching, volume 3. Addison-Wesley, 1973.

[2] Z. Czech, G. Havas, and B. Majewski. An optimal algorithm for generating minimal perfect hash functions. Information Processing Letters, 43(5):257–264, 1992.

[3] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. A Practical Minimal Perfect Hashing Method. 4th Intl. Workshop on Efficient and Experimental Algorithms (WEA05), Springer-Verlag, vol. 3505, 488-500, 2005.