Information Systems 38 (2013) 108–131

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Practical perfect hashing in nearly optimal space ☆

Fabiano C. Botelho [a,*], Rasmus Pagh [c], Nivio Ziviani [b]

[a] Data Domain an EMC Company, Santa Clara, USA
[b] Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte, Brazil
[c] IT University of Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 13 April 2010
Received in revised form
25 May 2012
Accepted 4 June 2012
Recommended by: K.A. Ross
Available online 18 June 2012

Keywords:
Perfect hash functions
Randomized algorithms
Random graphs
Large key sets

ABSTRACT

A *hash function* is a mapping from a key universe U to a range of integers, i.e., $h: U \mapsto \{0,1,\ldots,m-1\}$, where m is the range's size. A *perfect hash function* for some set $S \subseteq U$ is a hash function that is one-to-one on S, where $m \geq |S|$. A *minimal perfect hash function* for some set $S \subseteq U$ is a perfect hash function with a range of minimum size, i.e., $m = |S|$. This paper presents a construction for (minimal) perfect hash functions that combines theoretical analysis, practical performance, expected linear construction time and nearly optimal space consumption for the data structure. For n keys and $m = n$ the space consumption ranges from $2.62n + o(n)$ to $3.3n + o(n)$ bits, and for $m = 1.23n$ it ranges from $1.95n + o(n)$ to $2.7n + o(n)$ bits. This is within a small constant factor from the theoretical lower bounds of $1.44n$ bits for $m = n$ and $0.89n$ bits for $m = 1.23n$. We combine several theoretical results into a practical solution that has turned perfect hashing into a very compact data structure to solve the membership problem when the key set S is static and known in advance. By taking into account the memory hierarchy we can construct (minimal) perfect hash functions for over a billion keys in 46 min using a commodity PC. An open source implementation of the algorithms is available at http://cmph.sf.net under the GNU Lesser General Public License (LGPL).

1. Introduction

Perfect hashing is an elementary problem in computer science. The goal is to find a collision free hash function for a given static key set. Perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, item sets in data mining techniques [13,14], routing tables [43], sparse spatial data [35], and large web maps [18]. Perfect hashing methods can be used to construct a data structure to compactly store a static key set that supports queries to locate keys in one probe. For applications with only successful searches,[1] a key is simply represented by the value of a perfect hash function and the key set is not needed to locate information related with the key. For applications with unsuccessful searches, the key set has to be represented somehow to handle collisions.

There are many applications where the search space is restricted to keys with successful searches. One good example can be found in the deduplication of objects in a file system, which maintains an index that maps each unique object to a disk location of a block that holds it. At a given point in time, the file system knows all object

☆ This work was performed while the first author was an associated professor at the Department of Computer Engineering of the Federal Center for Technological Education of Minas Gerais, Belo Horizonte, Brazil, and an associated researcher at the Department of Computer Science of the Federal University of Minas Gerais, Belo Horizonte, Brazil.

* Corresponding author. Tel.: +1 408 368 7892.
E-mail addresses: fbotelho@datadomain.com (F.C. Botelho), pagh@itu.dk (R. Pagh), nivio@dcc.ufmg.br (N. Ziviani).

0306-4379/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2012.06.002

[1] A *successful search* happens when the queried key is found in the key set and an *unsuccessful search* happens otherwise.

identifiers in the system. Therefore, a perfect hash function can be used to locate the objects on disk without the need to keep object identifiers in main memory.

In a garbage collector system, it first marks all objects that can be possibly reached; second, it frees all unreferenced objects that have not been marked. A deduplicated file system, like the Data Domain[2] File System [49] (DDFS), stores tens of billions of objects, each one identified by a hash value of at least 20 bytes. For, say, 100 billion objects, we need approximately 2000 gigabytes of internal memory to keep track of the objects. However, by leveraging the index DDFS maintains, which has the key space a perfect hash function needs to be built for, we can build a more compact data structure. Such a data structure is composed of two parts: (i) the perfect hash function; and (ii) a bitmap used to indicate whether a given object is being referenced. To store such a data structure we need to store both the function and the bitmap. The bitmap size depends on the function range. A perfect hash function, like the one we describe in this paper, plays a fundamental role in terms of bringing down the memory requirements. For n keys, we are able to build functions that have a range of size $m=1.23n$. The space consumption for the functions ranges from 1.95 to 2.7 bits per key for large n. The bitmap would require 1.23 bits per key. Hence it is possible to bring the space requirements for the garbage collector from 2000 gigabytes to anywhere between 37 and 46 gigabytes. The important observation here is the fact that the index has the entire key space and therefore by having an one-to-one mapping one does not need to keep the keys in memory.

1.1. Notation and lower bounds

In this paper, a *key* is a bit string of maximum length L bits. A *key set* S is a subset of a *key universe* $U=\{0,1\}^L$ of size $u=2^L$. A *hash function* is a mapping from a key universe U to a range of integers, i.e., $h: U\mapsto\{0,1,\ldots,m-1\}$, where m is the range's size. A *perfect hash function* (PHF), for some set $S\subseteq U$, is a hash function that is one-to-one on S, where $m\geq|S|$. A *minimal perfect hash function* (MPHF), for some set $S\subseteq U$, is a perfect hash function with a range of minimum size, i.e., $m=|S|$. We present in Appendix A some of the symbols and acronyms used throughout the paper.

The theoretical lower bound for a perfect hash function description was first studied in [27,37] and a simpler proof was later given in [44]. Consider Mehlhorn's Theorem III.2.3.6 (a) presented in [37] as a starting point to derive theoretical lower bounds for the space consumption of the PHFs and MPHFs' description.

Theorem 1.1 (*Mehlhorn [37] Theorem III.2.3.6 (a)*). *Let u,m,n be non-negative integers. Given a key universe U of size u, a class $\mathcal{H}$ of functions $h: U\mapsto\{0,\ldots,m-1\}$ is called (u,m,n)-perfect if for every $S\subseteq U$, $|S|=n$, there is $h\in\mathcal{H}$ such that h is perfect for S. Then*

$$|\mathcal{H}|\geq\frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n\binom{m}{n}}.$$

Our focus in this paper is the case where $m<3n$. For this constraint, applying Stirling's approximation $x!\approx x^xe^{-x}\sqrt{2\pi x}$ to $\log|\mathcal{H}|$[3] yields an information theoretical lower bound for a PHF $(m=1.23n)$ of $(m-n+\frac{1}{2})\log(1-n/m)-(u-n+\frac{1}{2})\log(1-n/u)$ and for an MPHF $(m=n)$ of $(n-u-\frac{1}{2})\log(1-n/u)-\frac{1}{2}\log(2\pi n)$. Considering $u\gg n$, this gives a value of approximately $0.89n$ bits for PHFs and approximately $1.44n$ bits for MPHFs.

1.2. Contributions

In our algorithms we use the well-known idea of partitioning the input key set into small buckets. When the key set fits entirely in the internal memory there is no need for partitioning and we treat it as a single bucket. This leads to an algorithm that operates on internal random access memory, which is referred to as *RAM algorithm* from now on. When the key set does not fit in the internal memory we have to do the partitioning and optimize our algorithm for IO operations. This leads to an external memory algorithm, which is referred to as *EM algorithm* from now on.

The RAM and EM algorithms combine practical performance, expected linear construction time and nearly optimal space consumption for the resulting data structure. The engineering to combine several theoretical results into a practical solution has turned perfect hashing into a very compact data structure to solve the membership problem when the key universe is static and known in advance. Perfect hashing is the data structure that provides the best trade-off between space usage and lookup time when compared with other open addressing and chaining hash schemes too index static key sets [7].

The space consumption of our algorithms to store the resulting functions depends on the relation between m and n. For $m=1.23n$, the space consumption is approximately $1.95n+o(n)$ bits for the RAM algorithm and $2.7n+o(n)$ bits for the EM algorithm. For $m=n$, the space consumption is approximately $2.62n+o(n)$ bits for the RAM algorithm and $3.3n+o(n)$ bits for the EM algorithm. We remark that although the EM algorithm generates functions whose space consumption is $O(n)$ bits, the hidden constant in the asymptotic notation requires that n be in the order of hundreds of millions to achieve the space consumption described above. In practice this is not a limitation because for smaller sets the RAM algorithm should be used rather than the EM algorithm which is designed for large sets that cannot be processed in internal memory.

[2] Data Domain develops a deduplicated file system tailored for a backup load. It was acquired by EMC[2] in July 2009.

[3] Throughout this paper we denote $\log_2 x$ as $\log x$.