Indexing Internal Memory with Minimal Perfect Hash Functions

Fabiano C. Botelho[1,2], **Hendrickson R. Langbehn**[1], **Guilherme Vale Menezes**[1], **and Nivio Ziviani**[1]

[1]Department of Computer Science, Federal Univ. of Minas Gerais, Belo Horizonte, Brazil

[2]Computing Department, Federal Center for Technological Education, Belo Horizonte, Brazil

{fbotelho,reiter,gmenezes,nivio}@dcc.ufmg.br

Abstract. *A perfect hash function (PHF) is an injective function that maps keys from a set S to unique values, which are in turn used to index a hash table. Since no collisions occur, each key can be retrieved from the table with a single probe. A minimal perfect hash function (MPHF) is a PHF with the smallest possible range, that is, the hash table size is exactly the number of keys in S. MPHFs are widely used for memory efficient storage and fast retrieval of items from static sets. Differently from other hashing schemes, MPHFs completely avoid the problem of wasted space and wasted time to deal with collisions. In the past, the amount of space to store an MPHF description was* $O(\log n)$ *bits per key and therefore similar to the overhead of space of other hashing schemes. Recent results on MPHFs by [Botelho et al. 2007] changed this scenario: in their work the space overhead of an MPHF is approximately 2.6 bits per key. The objective of this paper is to show that MPHFs are a good option to index internal memory when static key sets are involved and both successful and unsuccessful searches are allowed. We have shown that MPHFs provide the best tradeoff between space usage and lookup time when compared with linear hashing, quadratic hashing, double hashing, dense hashing, cuckoo hashing and sparse hashing. For example, MPHFs outperforms linear hashing, quadratic hashing and double hashing when these methods have a hash table occupancy of 75% or higher (if the MPHF fits in the CPU cache the same happens for hash table occupancies greater than or equal to 55%). Furthermore, MPHFs also have a better performance in all measured aspects when compared to sparse hashing, which has been designed specifically for efficient memory usage.*

1. Introduction

Some types of databases are updated only rarely, typically by periodic batch updates. This is true, for example, for most data warehousing applications (see [Seltzer 2005] for more examples and discussion). Another interesting phenomenon was the web popularization, which created various new challenges related to the huge growth of data volume and the need to process it in order to get useful information. Search engines are responsible for collecting, representing, processing and disseminating this useful information according to the information need of their users. Besides the quality of the information provided, it is essential to satisfy the user information need in an efficient way despite the huge amount of data to be processed and the huge amount of users issuing queries all the time. Therefore, in such scenarios, it is necessary to construct data structures that permit to

represent this data volume as compactly as possible, and to process queries in an efficient way over it.

In this paper we study data structures that are suitable for indexing internal memory in an efficient way in terms of both space and lookup time, specially when memory intensive applications are involved. Indeed, we are interested in applications where a key set is fixed for a long period of time and each key is associated with satellite data. For example, this happens in On-Line Analytical Processing (OLAP) and search engine applications, which use extensive preprocessing of data to allow very fast evaluation of certain types of queries. More formally, given a *static* key set $S \subseteq U$ of size n from a key universe U of size u, where each key is associated with satellite data, the question we are interested in is: what are the data structures that provide the best tradeoff between space usage and lookup time?

An efficient way to represent a vocabulary in terms of lookup time is using a table indexed by a hash function. A *hash function* $h : U \rightarrow M$ is a function that maps the keys from U to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$. Considering $S \subseteq U$ and given a key $x \in S$, the hash function h computes an integer in $[0, m-1]$ for the storage or retrieval of x in a *hash table*. Hashing methods for *non-static key sets* can be used to construct data structures storing S and supporting membership queries of the type "$x \in S$?" in expected time $O(1)$. However, they involve a certain amount of wasted space owing to unused locations in the table and wasted time to resolve collisions when two keys are hashed to the same table location.

In the aforementioned scenarios, the key set is fixed for a long period of time (i.e., might be considered *static key sets*), hence it is possible to compute a function as part of the preprocessing phase to find any key in a table in one probe; such hash functions are called *perfect*. Given a key set S, we shall say that a hash function $h : U \rightarrow M$ is a *perfect hash function* (PHF) for S if h is an injection on S, that is, there are no *collisions* among the keys in S: if x and y are in S and $x \neq y$, then $h(x) \neq h(y)$. Since no collisions occur, each key can be retrieved from the table with a single probe. If $m = n$, that is, the table has the same size as S, then h is a minimal perfect hash function (MPHF). MPHFs completely avoid the problem of wasted space and time. Better still, it was observed in [Manegold et al. 2000] that MPHFs also avoid cache misses that arise due to collision resolution schemes like open addressing and chaining [Knuth 1973].

The objective of this paper is to show that MPHFs provide the best tradeoff between space usage and lookup time when compared to other hashing schemes. It was not the case in the past because the space overhead to store MPHFs was $O(\log n)$ bits per key for practical algorithms [Czech et al. 1992, Majewski et al. 1996]. Therefore, a better performace in terms of time and space was obtained by using a single hash function and resolving collisions with linear probing [Ho 1994, Knuth 1973]. However, new results on MPHFs by Botelho, Pagh and Ziviani [Botelho et al. 2007] have motivated this work. In their work, MPHFs require approximately 2.6 bits per key of space overhead and can be evaluated in $O(1)$ time.

We obtained interesting results in two scenarios: (i) when the MPHF description fits in the CPU cache and (ii) when it cannot be entirely placed in the CPU cache. In the first scenario we show that the other hashing schemes cannot outperform minimal perfect

hashing, even when the hash table occupancy is lower than 55%. An MPHF requiring just 2.6 bits per key of storage space permits to store sets on the order of 10 million keys in a 4 MB CPU cache, which is enough for a large range of applications. In the second scenario, other hashing schemes require a hash table occupancy lower than 75% to obtain the same performance attained by minimal perfect hashing. For both scenarios, the space overhead of minimal perfect hashing is within a factor of $O(\log n)$ bits lower than other hashing schemes.

2. The Algorithms

In this section we describe the hashing methods we used to compare minimal perfect hashing with, namely, linear hashing, quadratic hashing, double hashing, dense hashing, cuckoo hashing and sparse hashing. The hash table entries store items, and each item is composed by a key and possibly some data, i.e., a pair $< k, d >$. All the methods analyzed use collision resolution by open addressing, that is, they look at various positions of the hash table one by one until it either finds the key k being searched for or it finds an empty position [Knuth 1973]. In contrast, collision resolution could also be made by chaining, in which a linked list is used to store items that collided in the same table position. Open addressing is preferred over chaining if we are interested in lookup time, since it has a better locality of reference and reduces the number of cache misses.

The hash table structure used by linear hashing, quadratic hashing, double hashing, dense hashing and cuckoo hashing is shown in Figure 1. Every table position has a pointer, initially pointing to an empty value. When an item is inserted in the table, the pointer of the corresponding position starts to refer to it. The hash table structures for sparse hashing and minimal perfect hashing are presented in Sections 2.5 and 2.6, respectively.

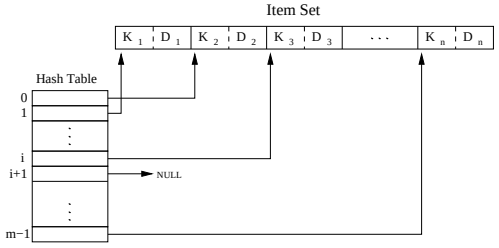


Figure 1. Hash table used for linear hashing, quadratic hashing, double hashing, dense hashing and cuckoo hashing.

Note that we should not insert the item itself in the table, since the allocated empty positions would cause an expressive waste of memory space, especially if the item occupies several bytes. Hence, the wasted space is reduced by using only one pointer per empty position. If we define p as the pointer size in bits, the space overhead for methods that use the structure in Figure 1 is $p \times m$ bits for a hash table of size m. For a 64 bits architecture, $p = 64$ bits.

Throughout this section we shall use $\oplus_m$ as a notation for an addition modulus m. For instance, we may describe the operation $(a + b) \bmod m$ as $a \oplus_m b$.

2.1. Linear Hashing

Linear hashing is considered one of the simplest open addressing schemes available. It uses a hash function $h : S \rightarrow [0, m-1]$ and tests positions $h(k), h(k) \oplus_m 1, h(k) \oplus_m 2, ...$ sequentially until it finds the term k being searched. Otherwise, if it finds an empty position, or if the sequential search reaches position $h(k)$ after running over all other positions, the item being searched does not exist in the hash table [Knuth 1973].

The pseudocode shown below represents how this method works:

1. Calculate $i = h(k)$.
2. If the i-th position is empty or $h(k)$ is reached again after running over all table positions, then the search is concluded and the item relative to k is not in the hash table.
3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.
4. Else, $i = i \oplus_m 1$. Go to step 2.

The efficiency of a search for a given key $k \in S$ in the linear hashing method depends on the number of probes performed during the search. This is highly sensitive to the hash table load factor $\alpha = n/m$ (i.e., the ratio between the number of items and the number of entries in the hash table.) The higher is α, the larger is the number of probes. According to Knuth [Knuth 1973] the expected number of probes performed for successful and unsuccessful searches are $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ and $\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$, respectively. The main problem with this method is that it degenerates in a sequential search when the number of terms n gets closer to the table size m, which causes a waste of time. Another issue is the waste of space caused by empty positions in the hash table.

2.2. Quadratic Hashing

Quadratic hashing is very similar to linear hashing, however, it uses two additional parameters, r and q, besides the hash function $h(k) : S \rightarrow [0, m-1]$. Parameter r indicates how many positions ahead the current position the next search for the term k will be performed, and parameter q indicates the value which parameter r will be added to after each iteration. Quadratic hashing is expected to have a better performance when compared to linear hashing for higher load factors, since it prevents the production of clusters which delay the search for items. However, this method shares some problems found in linear hashing, e.g., the waste of space due to empty positions and the waste of time due to successive collisions when n gets closer to m [Hopgood and Davenport 1972]. The quadratic hashing method may also have a smaller locality of reference when compared to linear hashing, as the pace r may become much larger than one.

The period of search is defined as the number of entries that appear in a sequence from a particular initial position before an entry is encountered twice. The period of search should preferably be the same as the table size m or, at least, as large as possible. Otherwise, the table may appear to be full when there is still space available. If m is a prime number then the period of search for the quadratic hash method is $m/2$.

The pseudocode shown below represents how this method works:

1. Calculate $i = h(k)$.

2. If the i-th position is empty or $h(k)$ is reached again after running over all reachable positions, then the search is concluded and the item relative to k is not in the hash table.
3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.
4. Else, $i = i \oplus_m r$, $r = r \oplus_m q$. Go to step 2.

Given a hash table load factor $\alpha = n/m$, the expected number of probes in quadratic hashing is $1 - \ln(1-\alpha) - \frac{\alpha}{2}$ for successful searches and $\frac{1}{1-\alpha} - \ln(1-\alpha) - \alpha$ for unsuccessful searches, according to [Knuth 1973]. Furthermore, in [Knuth 1973] it was proposed a variation of quadratic hashing, which was also compared with perfect hashing in our experiments. We used an implementation available in [Silverstein 2007], which is called dense hashing.

2.3. Double Hashing

Double hashing also works in a way very similar to linear hashing, but with the difference that, instead of one function, it uses two: $h_1(k)$ and $h_2(k)$. The first one produces values in the range $[0, m-1]$, mapping the term into its position in the hash table, the same way as the hash function in linear hashing does. The additional function $h_2(k)$ produces values in the range $[1, m-1]$, which are used as steps in the process of finding empty positions. Values produced by $h_2(k)$ are relatively primes to the table size m. This is necessary to ensure that the period of search will be of the same size as table size m, which guarantees that any given item can be inserted in any table position (see, e.g., [Knuth 1973]). Furthermore, we can check if the table is full by counting the number of collisions, since m successive collisions indicates a full structure.

This method tests positions using a distance $h_2(k)$, i.e., it tests positions $h_1(k), h_1(k) \oplus_m h_2(k), h_1(k) \oplus_m 2h_2(k), ...$, until it finds an empty position or until it finds the term k being searched for.

The method is described bellow:

1. Calculate $i = h_1(k)$, $d = h_2(k)$.
2. If the i-th position is empty or $h_1(k)$ is reached again after running over all table positions, then the search is concluded and the item relative to k is not in the table.
3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.
4. Else, $i = i \oplus_m d$. Go to step 2.

Double hashing reduces the problem of clustering in a better way than quadratic hashing does. This is because function $h_2(k)$ provides a different step d for each key k, and the multiple step sizes produce a more uniform distribution of the used positions. This method still shares some problems with previously cited methods, such as the waste of space due to unused positions and the possibility of successive collisions when the structure is almost full. Knuth [Knuth 1973] estimated the expected number of successful probes in searches for double hashing as $-\left(\frac{1}{\alpha}\ln(1-\alpha)\right)$, and the number of unsuccessful probes in searches as $\frac{1}{1-\alpha}$.

2.4. Cuckoo Hashing

Cuckoo hashing uses two hash functions, $h_1(k)$ and $h_2(k)$, to get two possible table positions for a given term. When a term x has to be inserted in the structure, one of the two

possible positions ($h_1(x)$ or $h_2(x)$) is chosen. If the chosen position is already occupied, the term y contained there will be removed from the structure, yielding an empty position to the term x being inserted. Term y, in turn, has two possible positions, given by $h_1(y)$ and $h_2(y)$. Consequently, y can be inserted in a position different from its former one. However, that position can be occupied too. Thus, this process must continue until all terms are inserted in one of their possible positions, or until some item can not be inserted [Zukowski et al. 2006, Pagh and Rodler 2004].

In case we need to search for a term k, the two possible positions for k (namely $h_1(k)$ and $h_2(k)$) are checked. If neither one contains the term, then it is not in the structure. Insertion in cuckoo hashing is better described bellow:

1. Calculate $i = h_1(k)$
2. If the i-th position is empty, insert the term k in that position
3. Else,
   Swap the term k with the term x contained in the i-th position
   If $h_1(x) == i$, then $i = h_2(x)$
   Else, $i = h_1(x)$
   Go to step 2

A problem with this method is that it is possible that it gets into an infinite loop during the insertion of a term, since it can cause a sequence of items to be expelled indefinitely in a cyclical manner. We may prevent this by allowing only a maximum amount of iterations during term insertion. Notwithstanding, cuckoo hashing still will not be able to insert the term with the same hash function values, and the table needs to be rebuilt with different functions if the term is to be inserted.

2.5. Sparse Hashing

Sparse hashing is based on a sparse array structure which uses little memory space. It is implemented as an array of groups A, where the number of groups in a sparse array of m entries is calculated as $G = \lceil m/M \rceil$. Each group stored in $A[g]$, $0 \leq g < G$, is responsible for M indexes of the hash table, i.e., $A[0]$ is responsible for the items in the range $[0, M-1]$, $A[1]$ for the items in the range $[M, 2M-1]$, and so on. Each group g contains an array R_g that stores the actual items and a bitmap B_g of size M. The bitmap B_g indicates the assigned indexes in the range $[0, M-1]$. If $B_g[f] = 1$, $0 \leq f < M$, then index f has a corresponding value stored in R_g. Note that an item in group g with an offset f is not necessarily placed in position f of R_g, but in the position $R_g[j]$, where j is the number of bits set from $B_g[0]$ to $B_g[f-1]$. Therefore, the array R_g is dynamically reallocated when new items are inserted in it. Thus, the size of R_g can differ among groups. Figure 2 illustrates these data structures.

A lookup for an item with key k is performed by first calculating its position $i = h(k)$, in which $h(k) : S \rightarrow [0, m-1]$. The group g to which the item belongs is defined as $g = \lfloor i/M \rfloor$, and its offset inside g is $f = i \bmod M$. In this way, we need to check the value of $B_g[f]$. If it is set to 0, then the item is not present in the hash table. Otherwise, it is possibly present in group g and we need to check if there is a collision. This can be done by checking if the item with key k is present in R_g. The position j of the item in this array is calculated by counting the number of bits set between $B_g[0]$ and $B_g[f-1]$. If the item in position j is not the one with key k, then there is a collision, which will be resolved by quadratic probing on i (see Section 2.2).

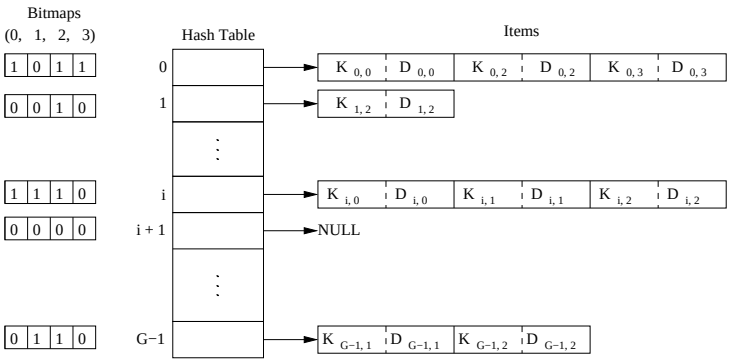


Figure 2. Hash table used in the sparse hashing method.

Insertion is performed in a similar fashion. First, we must check if the item is present with a lookup. If not, we shall insert the item in R_g in the position calculated by counting the number of bits set between $B_g[0]$ and $B_g[f-1]$, in the same way it is done in the lookup. An insertion may require the displacement of all items with internal offset j such that $j \geq f$. Let us take Figure 2 as an example. Suppose we want to insert a certain item with key k for which $g = 0$ and $f = 1$. Then the item must be inserted in position 1 of group 0, but that position is already occupied. To solve this, we need to move the items with key $K_{0,2}$ and $K_{0,3}$ one position ahead of their current position. The item with $K_{0,3}$ will be moved to the position allocated for the new term, i.e., the forth position. The item with key $K_{0,2}$ will be moved to the position just left of the item with key $K_{0,3}$, i.e., the third position. Finally, the position calculated for the item with key k will be free and we can place the new item there. Figure 3 shows the situation of group 0 after the insertion of the item with key $K_{0,1}$.

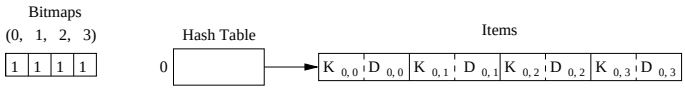


Figure 3. Group 0 after an insertion.

This method differs from the others in the sense that it prioritizes efficient memory usage. It allocates as little space as possible to represent unassigned positions, and the arrays containing the actual items grow only when it is needed. If each pointer has a size of p bits, the space overhead of sparse hashing for a hash table of size m and G groups is $m + G \times p$. That is, m bits to represent the bitmaps, and G pointers, one for each group.

Although being very efficient in memory usage, sparse hashing is not designed to be efficient in time: each lookup needs to perform a sequential search through B_g to find the position of an item R_g.

2.6. Minimal Perfect Hashing

The hash table structure used by minimal perfect hashing is shown in Figure 4. In this structure there is no need for pointers, i.e., all the items are inserted directly in the table. This is only possible because there are no empty entries in the hash table, and therefore we will not lose any space if we increase the capacity of the table entries to fit the items themselves. This is not the case for the other methods, in which any increase in the

capacity of the table entries would cause even more space to be wasted. Moreover, the minimal perfect hashing avoids the use of memory space to keep the pointers, which is an additional advantage. However, there is still the need to store the MPHF representation in main memory, and the space overhead for this method is approximately $2.62n$ bits for a set of n keys, as can be seen in [Botelho et al. 2007].

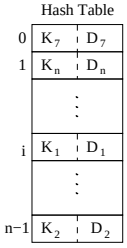


Figure 4. Hash table used in the perfect hashing method.

The minimal perfect hash function $h : S \rightarrow [0, n-1]$ used to index the hash table presented in Figure 4 is taken from the family of MPHFs proposed in [Botelho et al. 2007]. Their MPHFs are generated based on random r-partite hypergraphs where each edge connects $r \geq 2$ vertices[1]. In our experiments we used a version that employs hypergraphs with $r = 3$, since it generates the fastest and most compact MPHFs. However, for simplicity of exposition, we will now illustrate the MPHF construction when $r = 2$.

Figure 5 gives an overview of the MPHF construction for $r = 2$, taking as input a key set $S \subseteq U$ containing the first four month names abbreviated to the first three letters, i.e., $S = \{\text{jan}, \text{feb}, \text{mar}, \text{apr}\}$. The mapping step in Figure 5(a) assumes that it is possible to find $r = 2$ hash functions, h_0 and h_1, with independent values uniformly distributed in the intervals [0,3] and [4,7], respectively. These functions are used to assign each key in S to an edge of an acyclic random bipartite graph $G = (V, E)$[2], such that $|V| = m = 8$ and $|E| = n = 4$. In our example, January is mapped to edge $\{h_0(jan), h_1(jan)\} = \{2, 5\}$, February is mapped to $\{h_0(jan), h_1(jan)\} = \{2, 6\}$, and so on.

The assigning step in Figure 5(b) builds an array g representing a function $g : [0, m-1] \rightarrow \{0, 1, 2\}$, which is used to uniquely assign an edge with key k to one of its $r = 2$ incident vertices. The value r is used to represent unassigned vertices. Note that a vertex for a key k is either given by $h_0(k)$ or $h_1(k)$. The decision of which function $h_i(k)$ to be used for k is made by calculating $i = (g[h_0(k)] + g[h_1(k)]) \bmod 2$. In our example, January is mapped to 2 because $(g[2] + g[5]) \bmod 2 = 0$ and $h_0(jan) = 2$. Similarly, February is mapped to 6 because $(g[2] + g[6]) \bmod 2 = 1$ and $h_1(feb) = 6$, and so on.

The ranking step builds a data structure used to compute a function $rank(v)$, which counts in $O(1)$ time the number of assigned positions in g before a given position $v \in [0, m-1]$. This is a well-studied primitive in succinct data structures [Pagh 2001, Okanohara and Sadakane 2007, Raman et al. 2002]. To illustrate, $rank(7) = 3$ means that there are three positions assigned before position 7 in g, namely $0, 2$ and 6.

[1]A hypergraph is the generalization of a standard undirected graph where each edge connects $r \geq 2$ vertices.

[2]See [Botelho et al. 2007] for details on how to obtain such a graph with high probability.

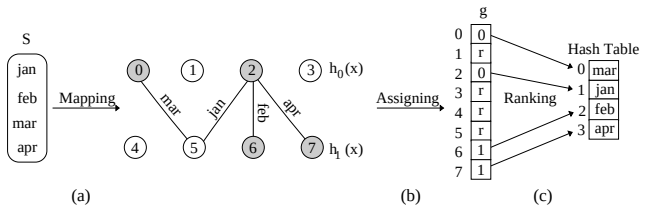


Figure 5. (a) The mapping step generates an acyclic bipartite random 2-graph. (b) The assigning step builds an array g so that each edge is uniquely assigned to a vertex. (c) The ranking step builds the data structure used to compute function $rank : V \rightarrow [0, n-1]$ in $O(1)$ time.

In our experiments, the MPHF is constructed based on hypergraphs with $r = 3$, and we use three hash functions $h_i : S \rightarrow [i\frac{m}{3}, (i+1)\frac{m}{3} - 1]$, in which $0 \leq i < 3$ and $m = 1.23n$. The value $1.23n$ is required to generate an acyclic random 3-partite hypergraph with high probability [Botelho et al. 2007]. Here again, the functions are assumed to have independent values uniformly distributed. The MPHF has the following form: $h(k) = rank(phf(k))$, where $phf : S \rightarrow [0, 1.23n - 1]$ is a perfect hash function defined as $phf(k) = h_i(k)$ and $i = (g[h_0(k)] + g[h_1(k)] + g[h_2(k)]) \bmod 3$. The array g is now representing a function $g : V \rightarrow \{0, 1, 2, 3\}$, and $rank : V \rightarrow [0, n-1]$ is now the cardinality of $\{u \in V \mid u < v \wedge g[u] \neq 3\}$. Notice that a vertex u is assigned if $g[u] \neq 3$.

3. Experimental Results

In this section we present the key sets used in the experiments and the results of the comparative study. All experiments were carried out on a computer running Linux version 2.6, with a 1.86 gigahertz Intel Core 2 64 bits processor, 4 gigabytes of main memory and 4 megabytes of L2 cache. All results presented are averages on 50 trials and were statistically validated with a confidence level of 95%. Table 1 summarizes the symbols and acronyms used throughout this section.

Symbol	Meaning
α	Load factor.
n	Number of keys in a key set.
N	Number of keys used in the lookup step.
Probes/N	Average number of probes per key during the lookup.
T(s)	Average time (in seconds) spent during the lookup of N keys.
$\mathcal{S}_o$(bits/key)	Space Overhead in bits per key.
LH	Linear Hashing.
QH	Quadratic Hashing.
DH	Double Hashing.
CH	Cuckoo Hashing.
SH	Sparse Hashing.
DeH	Dense Hashing.
MPH	Minimal Perfect Hashing.

Table 1. Symbols and acronyms used throughout this section.

The linear hashing, quadratic hashing, double hashing, cuckoo hashing and minimal perfect hashing structures were all implemented using the C language. We used the CMPH library available at `http://cmph.sf.net` to generate the MPHFs used in the minimal perfect hashing structure. For sparse hashing and dense hashing we used the original implementation available in [Silverstein 2007].

It is important to notice that we are interested in the performance of lookups and therefore we do not present results concerning the time to build the data structures. Nevertheless, it is important to stress that the MPHF construction is very fast, as can be seen in [Botelho et al. 2007]. As an illustration, for a set of 37,294,116 keys, the construction of the MPHF takes 1 minute and 38 seconds. We consider two situations: (i) when only successful lookups are performed (i.e., the key is always found in the hash table) and (ii) when only unsuccessful lookups are involved (i.e., a key is never found in the hash table). The results are evaluated for each data structure in terms of the average number of lookups, the average lookup time and the space overhead.

The experimental results are presented in three distinct subsections. First, in Section 3.2, we compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing structures. Second, in Section 3.3, we compare it with sparse hashing and dense hashing structures. Finally, in Section 3.4, we compare it with cuckoo hashing structure. The three sets of experiments use the key sets described in Section 3.1.

3.1. Key Sets

In our experiments we used three key sets: (i) a key set of $5,424,923$ unique query terms extracted from the AllTheWeb[3] query log, referred to as AllTheWeb key set; (ii) a key set of $37,294,116$ unique URLs collected from the Brazilian Web by the TodoBr[4] search engine, referred to as URLs-37 key set; and (iii) a smaller key set of 10 million URLs randomly selected from the URLs-37 key set, which is referred to as URLs-10 key set. Table 2 shows the main characteristics of each key set, namely the shortest key size, the largest key size and the average key size in bytes.

Key Set	n	Shortest Key	Largest Key	Average Size of the Keys
AllTheWeb	5,424,923	2	31	17.46
URLs-10	10,000,000	8	494	58.36
URLs-37	37,294,116	8	496	58.77

Table 2. Characteristics of the key sets used for the experiments.

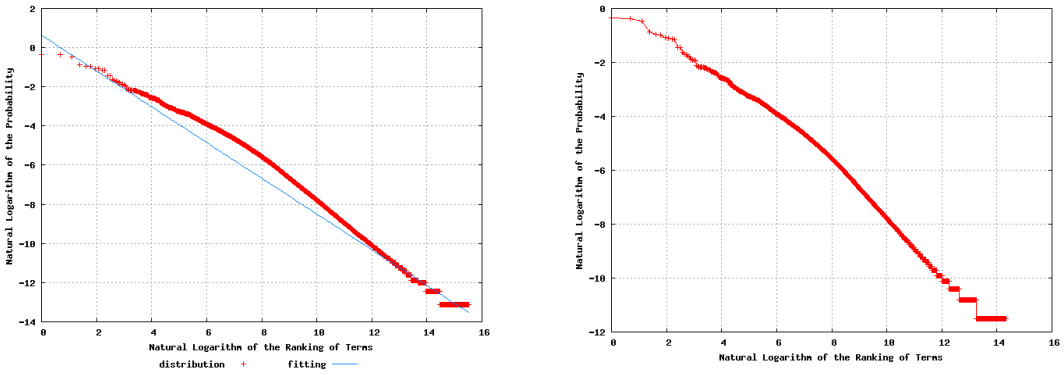
In order to test the lookup performance of the considered hash structures in a real world environment, we need to look up keys in a way similar to the real access patterns of actual applications. In the case of the AllTheWeb key set, the probability distribution of query term lookups was extracted from the AllTheWeb query log. Similarly, the distribution of URL lookups must be equivalent to the access pattern performed by a web crawler that needs to check whether a URL extracted from a web page is new, i.e., whether it has not been collected before. Therefore, we decided to use an automatic generator to simulate these lookup patterns found in search engines.

The probability distribution of query term lookups for the AllTheWeb key set is shown in Figure 6 (a). It is plotted in a log-log scale, constituting a power law distribution with inclination -0.91. This same distribution was used to simulate the lookup stream submitted to the hashing data structures in order to evaluate their performance, as can be

[3]AllTheWeb (`www.alltheweb.com`) is a trademark of Fast Search & Transfer company, which was acquired by Overture Inc. in February 2003. In March 2004 Overture itself was taken over by Yahoo!.

[4]TodoBr (www.todobr.com.br) is a trademark of Akwan Information Technologies, which was acquired by Google Inc. in July 2005.

seen in Figure 6 (b). We generated 10 million keys to be looked up in a hashing data structure storing the AllTheWeb key set.



(a) Extracted from AllTheWeb query log. (b) Generated automatically.

Figure 6. Probability distribution of query term lookups

Pages arriving in a crawling system are known to have a few very popular URLs and many not so popular URLs, which also constitutes a power law behavior [Broder et al. 2000]. Consequently, we employed the same distribution found for query terms to describe the probability of arrival of a URL in a crawler. We generated 250 million and 20 million URLs to be looked up in the hashing data structures that store the URLs-37 key set and the URLs-10 key set, respectively.

So far we have described how to generate key sets to perform successful searches in hashing data structures. In order to test the performance of the data structures when unsuccessful searches are involved, we have randomly generated three additional key sets: (i) 10 million keys of average size equal to 17.46 bytes to be looked up when the structures are storing the AlltheWeb key set, (ii) 20 million keys of average size equal to 58.36 bytes to be looked up when the structures are storing the URLs-10 key set, and (iii) 250 million keys of average size equal to 58.77 bytes to be looked up when the structures are storing the URLs-37 key set. They were created based on the average key sizes presented in Table 2. The successful and unsuccessful searches correspond to a lower bound and to an upper bound of the execution time, respectively.

In our experiments we used an 8-byte fingerprint of the key instead of the key itself. The use of fingerprints was motivated by two reasons: (i) to guarantee that all keys have the same size, since in this way we can allocate a fixed size for each key without waste of space; and (ii) to reduce the amount of memory used to store each key, as the average key size in all key sets used is greater than 8 bytes. A point worth noting is that each key set was stored entirely in main memory, but the set of automatically generated keys is too big to be stored in the same way, and had to be kept in disk.

3.2. Minimal Perfect Hashing Versus Linear Hashing, Quadratic Hashing and Double Hashing

In this section we compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing. Linear hashing, quadratic hashing and double hashing methods were tested with different load factors, ranging from 50 to 90% (because

of lack of space we present results for load factors of 50%, 75% and 85%). We considered both successful and unsuccessful searches to measure the average number of probes and the amount of time spent (on average) to look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

The results for successful and unsuccessful searches are presented in Tables 3 and 4, respectively. As expected, quadratic hashing and double hashing perform better than linear hashing for high load factors, since they avoid the creation of clusters in this case. Furthermore, we can see that double hashing always has a smaller number of collisions per key when compared to quadratic hashing and linear hashing, but it is slower since it needs to compute two hash functions instead of one. The average number of probes measured for both successful and unsuccessful searches are very close to the expected according to the equations presented in Sections 2.1, 2.2 and 2.3 (this is not shown in the tables).

		LH		QH		DH	
Key Set	α	Probes/N	T(s)	Probes/N	T(s)	Probes/N	T(s)
AllTheWeb	85%	3.78	5.67	2.40	5.27	2.17	5.42
	75%	2.47	5.04	1.93	4.97	1.90	5.11
	50%	1.48	4.34	1.42	4.40	1.40	4.56
URLs-10	85%	3.63	18.98	2.27	17.87	2.16	18.36
	75%	2.37	17.69	1.87	17.29	1.83	17.69
	50%	1.51	16.33	1.39	16.19	1.35	16.39
URLs-37	85%	3.94	269.19	2.37	253.18	2.29	263.80
	75%	2.46	247.95	1.89	242.51	1.83	250.60
	50%	1.55	229.62	1.43	228.92	1.37	233.79

Table 3. Load factor influence on the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

		LH		QH		DH	
Key Set	α	Probes/N	T(s)	Probes/N	T(s)	Probes/N	T(s)
AllTheWeb	85%	22.80	14.82	7.54	8.60	6.67	8.89
	75%	8.44	8.17	4.43	6.67	4.00	6.95
	50%	2.50	5.19	2.13	5.14	2.00	5.25
URLs-10	85%	22.61	34.81	7.54	22.68	7.25	23.71
	75%	8.49	21.93	4.43	18.77	4.00	19.27
	50%	2.50	15.59	2.13	15.57	2.00	15.63
URLs-37	85%	22.53	526.05	7.55	333.49	6.67	379.17
	75%	8.51	318.94	4.43	270.53	4.00	296.62
	50%	2.50	220.64	2.13	217.66	2.00	222.92

Table 4. Load factor influence on the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

We now compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing. Tables 5 and 6 show two remarkable results. First, when the MPHF description fits in the L2 cache, which is the case for the AllTheWeb key set and URLs-10 key set, the minimal perfect hashing structure outperforms the others in terms of lookup time for load factors greater than 55% for both successful and unsuccessful searches. Second, in the converse situation in which the MPHF description does not fit in the L2 cache, which is the case for the URLs-37 key set, the same thing happens for load factors greater than or equal to 75% and 65% for successful and unsuccessful searches, respectively. Therefore, as can be seen, the use of MPHFs saves a significant amount of space with almost no loss in the lookup time.

Data Structure	AllTheWeb			URLs-10			URLs-37		
	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62	100	250.36	2.62
LH	55	4.46	116.36	55	16.34	116.36	75	247.95	85.33
QH	55	4.52	116.36	55	16.33	116.36	80	247.48	80
DH	50	4.56	128	50	16.39	128	75	250.60	85.33

Table 5. Comparison of MPH with LH, QH and DH, considering the space overhead and the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

Data Structure	AllTheWeb			URLs-10			URLs-37		
	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	5.33	2.62	100	16.38	2.62	100	252.65	2.62
LH	55	5.48	116.36	60	16.91	106.67	65	258.15	98.46
QH	55	5.36	116.36	60	16.57	106.67	70	253.64	91.43
DH	55	5.48	116.36	60	16.70	106.67	65	257.75	98.46

Table 6. Comparison of MPH with LH, QH and DH, considering the space overhead and the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

3.3. Minimal Perfect Hashing Versus Sparse Hashing and Dense Hashing

Sparse hashing and dense hashing were tested with their default load factor only, which is 80%. Table 7 shows the time spent to execute the lookup step for each method for successful searches only. As expected, sparse hashing had the worst performance in lookup time when compared to the other methods, as it is designed to be efficient in space but not in execution time. The same is true for unsuccessful searches, and we omit the results to save space. It is important to note that perfect hashing has clearly outperformed the other methods in all aspects. Experiments were performed using only the AllTheWeb and URLs-10 key sets. We decided not to use the URLs-37 key set since we did not expect any improvements on the results.

Data Structure	AllTheWeb			URLs-10		
	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62
SH	80	11.47	2,92	80	35.76	2,92
DeH	80	6.51	80	80	27.48	80

Table 7. Comparison of MPH with DeH and SH, considering the space overhead and the time to successfully look up 10 and 20 million keys in the AllTheWeb and URLs-10 key sets, respectively.

3.4. Minimal Perfect Hashing Versus Cuckoo Hashing

Cuckoo hashing has a different behavior when compared to any of the methods analyzed, as it cannot work if the load factor is high, i.e., at most 50%. Therefore, we decided to show the comparison between this method and perfect hashing in this separated subsection. Cuckoo hashing was tested with load factors ranging from 20% to the maximum load factor with which it works.

Table 8 shows the average number of probes and the average lookup time to successfully search for 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively. We can see that cuckoo hashing performs slightly faster for all key sets used, but the space overhead for the minimum perfect hashing structure is much

lower than for cuckoo hashing in all experiments. The same happens for unsuccessful searches, as we can see in Table 9.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62	100	250.36	2.62
CH	20	4.08	320	20	15.99	320	20	222.40	320
CH	30	4.13	213	30	16.05	213	30	224.98	213
CH	40	4.28	160	40	16.22	160	40	228.76	160
CH	50	4.38	128	50	16.34	128	50	233.89	128

Table 8. Comparison of MPH with CH, considering the space overhead and the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)	$\alpha(\%)$	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	5.33	2.62	100	16.38	2.62	100	252.65	2.62
CH	20	5.06	320	20	15.79	320	20	222.46	320
CH	30	5.10	213	30	15.92	213	30	227.21	213
CH	40	5.30	160	40	16.07	160	40	229.58	160
CH	50	5.34	128	50	16.17	128	50	231.26	128

Table 9. Comparison of MPH with CH, considering the space overhead and the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

4. Conclusions

In this paper we have presented a thorough study of data structures that are suitable for indexing internal memory in an efficient way in terms of both space and lookup time when we have a key set that is fixed for a long period of time (i.e., a static key set) and each key is associated with a satellite data. This is widely used in data warehousing and search engine applications (see [Seltzer 2005] for other examples).

It is well known that an efficient way to represent a key set in terms of lookup time is by using a table indexed by a hash function. For static key sets it is possible to pay the price of pre-computing a MPHF to find any key in a table in one single probe. We have shown that minimal perfect hashing has a clear advantage in memory usage when compared to other hashing methods, since there are no empty entries in its hash table and thus space overhead is greatly reduced by avoiding the use of pointers. This implies in a gain of $O(\log n)$ bits.

In our study, we compared MPHFs with linear hashing, quadratic hashing, double hashing, dense hashing, cuckoo hashing and sparse hashing. We have shown that MPHFs provide the best tradeoff between space usage and lookup time among these hashing schemes. As an example, minimal perfect hashing have a better performance in all measured aspects when compared to sparse hashing, which has been designed specifically for efficient memory usage. Furthermore, if the MPHF can be stored in cache, minimal perfect hashing outperforms linear hashing, quadratic hashing and double hashing in all aspects when these methods have a hash table occupancy of 55% or higher. The same happens for hash table occupancies greater than or equal to 75% if the MPHF does not fit in cache. This implies in a significant memory overhead due to a great number of unused positions in the hash table.

5. Acknowledgements

We thank the partial support given by INFOWEB Project Grant MCT/CNPq/CT-INFO 550874/2007-0, and CNPq Grant 30.5237/02-0 (Nivio Ziviani).

References

Botelho, F., Pagh, R., and Ziviani, N. (2007). Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADs'07)*, pages 139–150. Springer LNCS vol. 4619.

Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. (2000). Graph structure in the web. *Computer Networks*, 33(1):309–320.

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

Ho, Y. (1994). Application of minimal perfect hashing in main memory indexing. Technical report, Cambridge, MA, USA.

Hopgood, F. and Davenport, J. (1972). The quadratic hash method when the table size is a power of 2. *The Computer Journal*, 15(4):314–315.

Knuth, D. E. (1973). *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, second edition.

Majewski, B., Wormald, N., Havas, G., and Czech, Z. (1996). A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554.

Manegold, S., Boncz, P. A., and Kersten, M. L. (2000). Optimizing database architecture for the new bottleneck: Memory access. *The VLDB journal*, 9:231–246.

Okanohara, D. and Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX'07)*.

Pagh, R. (2001). Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363.

Pagh, R. and Rodler, F. F. (2004). Cuckoo hashing. *J. Algorithms*, 51(2):122–144.

Raman, R., Raman, V., and Rao, S. S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, pages 233–242, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Seltzer, M. (2005). Beyond relational databases. *ACM Queue*, 3(3).

Silverstein, C. (2007). An extremely memory-efficient hash_map implementation (google-sparsehash). `http://code.google.com/p/google-sparsehash`.

Zukowski, M., Héman, S., and Boncz, P. (2006). Architecture-conscious hashing. In *Second DAMON workshop (SIGMOD 2006)*, Chicago, USA.