External Perfect Hashing for Very Large Key Sets

Fabiano C. Botelho
Dept. of Computer Science
Federal Univ. of Minas Gerais
Belo Horizonte, Brazil
fbotelho@dcc.ufmg.br

Nivio Ziviani
Dept. of Computer Science
Federal Univ. of Minas Gerais
Belo Horizonte, Brazil
nivio@dcc.ufmg.br

ABSTRACT

We present a simple and efficient external perfect hashing scheme (referred to as EPH algorithm) for very large static key sets. We use a number of techniques from the literature to obtain a novel scheme that is theoretically well-understood and at the same time achieves an order-of-magnitude increase in the size of the problem to be solved compared to previous "practical" methods. We demonstrate the scalability of our algorithm by constructing minimum perfect hash functions for a set of 1.024 billion URLs from the World Wide Web of average length 64 characters in approximately 62 minutes, using a commodity PC. Our scheme produces minimal perfect hash functions using approximately 3.8 bits per key. For perfect hash functions in the range $\{0, \ldots, 2n-1\}$ the space usage drops to approximately 2.7 bits per key. The main contribution is the first algorithm that has experimentally proven practicality for sets in the order of billions of keys and has time and space usage carefully analyzed without unrealistic assumptions.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks; E.2 [**Data Storage Representations**]: Hash-table representations

General Terms

Algorithms, Design, Performance, Theory

Keywords

minimal, perfect, hash, functions, large, key sets

1. INTRODUCTION

Perfect hashing is a space-efficient way of creating compact representation for a static set S of n keys. For applications with successful searches, the representation of a key $x \in S$ is simply the value $h(x)$, where h is a perfect hash function for the set S of values considered. A *perfect hash function* (PHF) maps the elements of S to unique values. A *minimal perfect hash function* (MPHF) produces values that are integers in the range $[0, n-1]$, which is the smallest possible range. More formally, a PHF maps a static key set $S \subseteq U$ of size $|S| = n$ to unique values in the range $[0, m-1]$, where $m \geq n$ and U is a key universe of size u. If $m = n$ we have a MPHF.

MPHFs 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, universal resource locations (URLs) in web search engines [7], item sets in data mining techniques [8, 9], sparse spatial data [22], graph compression [3], routing tables and other network applications [28].

Some types of databases are updated only rarely, typically by periodic batch updates. This is true, for example, for most data warehousing applications (see [31] for more examples and discussion). In such scenarios it is possible to improve query performance by creating very compact representations of keys by MPHFs. In applications where the set of keys is fixed for a long period of time the construction of a MPHF can be done as part of the preprocessing phase.

A PHF can also be used to implement a data structure with the same functionality as a Bloom filter [26]. In many applications where a set S of elements is to be stored, it is acceptable to include in the set some false positives[1] with a small probability by storing a signature for each perfect hash value. This data structure requires around 30% less space usage when compared with Bloom filters, plus the space for the PHF. Bloom filters have applications in distributed databases and data mining [8, 9].

Perfect hash functions have also been used to speed up the partitioned hash-join algorithm presented in [24]. By using a PHF to reduce the targeted hash-bucket size from 4 tuples to just 1 tuple they have avoided following the bucket-chain during hash-lookups that causes too many cache and translation lookaside buffer (TLB) misses.

Though there has been considerable work on how to construct PHFs, there is a gap between theory and practice among all previous methods on minimal perfect hashing. On one side, there are good theoretical results without experimentally proven practicality for large key sets. On the other side, there are the theoretically analyzed time and space usage algorithms that assume that truly random hash functions are available for free, which is an unrealistic assumption.

[1]False positives are elements that appear to be in S but are not.

CIKM'07, November 6–8, 2007, Lisboa, Portugal.

In this paper we attempt to bridge this gap between theory and practice, using a number of techniques from the literature to obtain a novel external memory based perfect hashing scheme, referred to as *EPH algorithm*. The EPH algorithm increases one order of magnitude in the size of the greatest key set for which a MPHF was obtained in the literature [4]. This improvement comes from a combination of a novel, theoretically optimal perfect hashing scheme that greatly simplifies previous methods, and the fact that our algorithm is designed to make good use of the memory hierarchy (see Section 4.2 for details).

The minimum amount of space to represent a MPHF for a given set S is known to be approximately 1.4427 bits per key. We present a scalable algorithm that produces MPHFs using approximately 3.8 bits per key. Also, for applications that can allow values in the range $\{0, \ldots, 2n-1\}$, the space usage drops to approximately 2.7 bits per key. We demonstrate the scalability of the EPH algorithm by considering a set of 1.024 billion strings (URLs from the world wide web of average length 64), for which we construct a MPHF on a commodity PC in approximately 62 minutes. If we use the range $\{0, \ldots, 2n-1\}$, the space for the PHF is less than 324 MB, and we still get hash values that can be represented in a 32 bit word. Thus we believe our MPHF method might be quite useful for a number of current and practical data management problems.

2. RELATED WORK

There is a gap between theory and practice among minimal perfect hashing methods. On one side, there are good theoretical results without experimentally proven practicality for large key sets. We will argue below that these methods are indeed not practical. On the other side, there are two categories of practical algorithms: the theoretically analyzed time and space usage algorithms that assume truly random hash functions for their methods, which is an unrealistic assumption because each truly random hash function $h : U \rightarrow [0, m-1]$ require at least $u \log m$ bits of storage space, and the algorithms that present only empirical evidences. The aim of this section is to discuss the existent gap among these three types of algorithms available in the literature.

2.1 Theoretical results

In this section we review some of the most important theoretical results on minimal perfect hashing. For a complete survey until 1997 refer to Czech, Havas and Majewski [11].

Fredman, Komlós and Szemerédi [15] proved, using a counting argument, that at least $n \log e + \log\log u - O(\log n)$ bits are required to represent a MPHF[2], provided that $u \geq n^{2+\jmath}$ for some $\jmath > 0$ (an easier proof was given by Radhakrishnan [29]). Mehlhorn [25] has made this bound almost tight by providing an algorithm that constructs a MPHF that can be represented with at most $n \log e + \log\log u + O(\log n)$ bits. However, his algorithm is far away from practice because its construction and evaluation time is exponential on n (i.e., $n^{\theta(ne^n u \log u)}$).

Schmidt and Siegel [30] have proposed the first algorithm for constructing a MPHF with constant evaluation time and description size $O(n + \log\log u)$ bits. Nevertheless, the scheme is hard to implement and the constants associated with the MPHF storage are prohibitive. For a set of n keys, at least $29n$ bits are used, which means that the space usage is similar in practice to schemes using $O(n \log n)$ bits.

One important theoretical result was proposed by Hagerup and Tholey [17]. The MPHF obtained can be evaluated in $O(1)$ time and stored in $n \log e + \log\log u + O(n(\log\log n)^2 / \log n + \log\log\log u)$ bits. The construction time is $O(n + \log\log u)$ using $O(n)$ computer words of the Fredman, Komlós and Szemerédi [16] model of computation. In this model, also called the *Word RAM* model, an element of the universe U fits into one machine word, and arithmetic operations and memory accesses have unit cost. The Hagerup and Tholey [17] algorithm emphasizes asymptotic space complexity only. As discussed in [5], for $n < 2^{150}$ the scheme is not even defined, as it relies on splitting the key set into buckets of size $\hat{n} \leq \log n/(21 \log\log n)$. By letting the bucket size be at least 1, then for $n < 2^{300}$ buckets of size one will be used, which means that the space usage will be at least $(3 \log\log n + \log 7)\, n$ bits. For a set of a billion keys, this is more than 17 bits per element. In addition, the algorithm is also very complicated to implement, but we will not go into that. Thus, it is safe to conclude that the Hagerup-Tholey algorithm will not be space efficient in practical situations.

2.2 Practical results assuming full randomness

Let us now describe the main practical results analyzed with the unrealistic assumption that truly random hash functions are available for free.

The family of algorithms proposed by Czech et al [23] uses random hypergraphs to construct order preserving MPHFs. A PHF h is *order preserving* if the keys in S are arranged in some given order and h preserves this order in the hash table. One of the methods uses two truly random hash functions $h_1(x) : S \rightarrow [0, cn-1]$ and $h_2(x) : S \rightarrow [0, cn-1]$ to generate MPHFs in the following form: $h(x) = (g[h_1(x)] + g[h_2(x)] \bmod n$ where $c > 2$. The resulting MPHFs can be evaluated in $O(1)$ time and stored in $O(n \log n)$ bits (that is optimal for an order preserving MPHF). The resulting MPHF is generated in expected $O(n)$ time. Botelho, Kohayakawa and Ziviani [4] improved the space requirement at the expense of generating functions in the same form that are not order preserving. Their algorithm is also linear on n, but runs faster than the ones by Czech et al [23] and the resulting MPHF are stored using half of the space because $c \in [0.93, 1.15]$. However, the resulting MPHFs still need $O(n \log n)$ bits to be stored.

Since the space requirements for truly random hash functions makes them unsuitable for implementation, one has to settle for a more realistic setup. The first step in this direction was given by Pagh [27]. He proposed a family of randomized algorithms for constructing MPHFs of the form $h(x) = (f(x) + d[g(x)]) \bmod n$, where f and g are chosen from a family of universal hash functions and d is a set of displacement values to resolve collisions that are caused by the function f. Pagh identified a set of conditions concerning f and g and showed that if these conditions are satisfied, then a MPHF can be computed in expected time $O(n)$ and stored in $(2+\epsilon)n \log n$ bits.

Dietzfelbinger and Hagerup [12] improved the algorithm proposed in [27], reducing from $(2+\epsilon)n \log n$ to $(1+\epsilon)n \log n$ the number of bits required to store the function, but in

[2]Logarithms are in base 2.

their approach f and g must be chosen from a class of hash functions that meet additional requirements. Woelfel [33] has shown how to decrease the space usage to $O(n \log\log n)$ bits asymptotically. However, there is no empirical evidence on the practicality of this scheme.

Botelho, Pagh and Ziviani [5] presented a family $\mathcal{F}$ of practical algorithms for construction and evaluation of PHFs of a given set that uses $O(n)$ bits to be stored, runs in linear time and the evaluation of a function requires constant time. The algorithms in F use r-uniform random hypergraphs given by function values of r hash functions on the keys of S. For $r = 2$ they obtained a space usage of $(3+\epsilon)n$ bits for a MPHF, for any constant $\epsilon > 0$. For $r = 3$ they obtained a space usage of approximately $2.62n$ bits for a MPHF, which is within a factor of 2 from the information theoretical lower bound of approximately $1.4427n$ bits. For $m = 1.23n$ they obtained a space usage of $1.95n$ bits, which is slightly more than two times the information theoretical lower bound of approximately $0.89n$ bits.

2.3 Empirical results

In this section we discuss results that present only empirical evidences for specific applications. Fox et al. [14] created the first scheme with good average-case performance for large data sets, i.e., $n \approx 10^6$. They have designed two algorithms, the first one generates a MPHF that can be evaluated in $O(1)$ time and stored in $O(n \log n)$ bits. The second algorithm uses quadratic hashing and adds branching based on a table of binary values to get a MPHF that can be evaluated in $O(1)$ time and stored in $c(n + 1/\log n)$ bits. They argued that c would be typically lower than 5, however, it is clear from their experimentation that c grows with n and they did not discuss this. They claimed that their algorithms would run in linear time, but, it is shown in [11, Section 6.7] that the algorithms have exponential running times in the worst case, although the worst case has small probability of occurring.

Fox, Chen and Heath [13] improved the result of [14] to get a function that can be stored in cn bits. The method uses four truly random hash functions to construct a MPHF. Again c is only established for small values of n. It could very well be that c grows with n. So, the limitation of the three algorithms from [14] and [13] is that no guarantee on the size of the resulting MPHF is provided.

Lefebvre and Hoppe [22] have recently designed MPHFs that require up to 7 bits per key to be stored and are tailored to represent sparse spatial data. In the same trend, Chang, Lin and Chou [8, 9] have designed MPHFs tailored for mining association rules and traversal patterns in data mining techniques.

2.4 Differences between the EPH algorithm and previous results

In this work we propose a practical algorithm that is theoretically well-understood and performs efficiently for very large key sets. To the best of our knowledge the EPH algorithm is the first one that demonstrates the capability of generating MPHFs for sets in the order of billions of keys, and the generated functions require less than 4 bits per key to be stored. This increases one order of magnitude in the size of the greatest key set for which a MPHF was obtained in the literature [4], mainly because the EPH algorithm is designed to make good use of the memory hierarchy, as discussed in Section 4.2. We need $O(N)$ computer words, where $N \ll n$, for the construction process. Notice that both space usage for representing the MPHF and the construction time are carefully proven. Additionally, our scheme is much simpler than previous theoretical well-founded schemes.

3. THE EPH ALGORITHM

Our algorithm uses the well-known idea of partitioning the key set into a number of small sets[3] (called "buckets") using a hash function h_0. Let $B_i = \{x \in S \mid h_0(x) = i\}$ denote the ith bucket. If we define $\mathit{offset}[i] = \sum_{j=0}^{i-1} |B_i|$ and let p_i denote a MPHF for B_i then clearly

$$p(x) = p_i(x) + \mathit{offset}[h_0(x)] \tag{1}$$

is a MPHF for the whole set S. Thus, the problem is reduced to computing and storing the offset array, as well as the MPHF for each bucket.

The EPH algorithm is essentially a two-phase multi-way merge sort with some nuances to make it work in linear time. Figure 1 illustrates the two steps of the EPH algorithm: *the partitioning* step and *the searching step*. The partitioning step takes a key set S and uses a hash function h_0 to partition S into 2^b buckets. The searching step generates a MPHF p_i for each bucket i, $0 \le i \le 2^b - 1$ and computes the offset array. To compute the MPHF of each bucket we used one algorithm from the family of algorithms proposed by Botelho, Pagh and Ziviani [5] (see Section 3.3 for details on the algorithm).

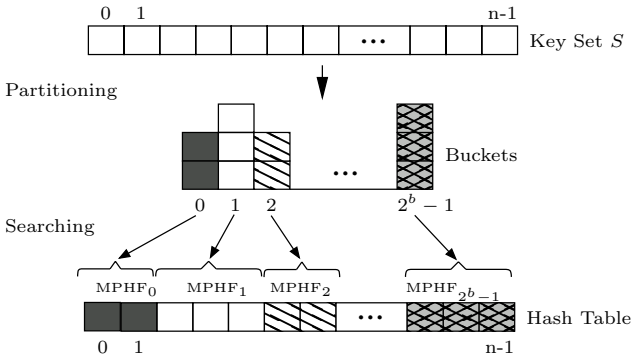

Figure 1: Main steps of the EPH algorithm

We will choose h_0 such that it has values in $\{0,1\}^b$, for some integer b. Since the offset array holds 2^b entries of at least $\log n$ bits we want 2^b to be less than around $n/\log n$, making the space used for the offset array negligible. On the other hand, to allow efficient implementation of the functions p_i we impose an upper bound ℓ on the size of any bucket. We will describe later how to choose h_0 such that this upper bound holds.

To create the MPHFs p_i we could choose from a number of alternatives, emphasizing either space usage, construction time, or evaluation time. We show that all methods based on the assumption of truly random hash functions can be made to work, with explicit and provably good hash functions. For the experiments we have implemented the

[3]Used in e.g. the perfect hash function constructions of Schmidt and Siegel [30] and Hagerup and Tholey [17], for suitable definition of "small".

algorithm presented in [5] (see Section 3.3 for more details). Since this computation is done on a small set, we can expect nearly all memory accesses to be "cache hits". We believe that this is the main reason why our method performs better than previous ones that access memory in a more "random" fashion.

We consider the situation in which the set of all keys may not fit in the internal memory and has to be written on disk. The EPH algorithm first scans the list of keys and computes the hash function values that will be needed later on in the algorithm. These values will (with high probability) distinguish all keys, so we can discard the original keys. It is well known that hash values of at least $2 \log n$ bits are required to make this work. Thus, for sets of a billion keys or more we cannot expect the list of hash values to fit in the internal memory of a standard PC.

To form the buckets we sort the hash values of the keys according to the value of h_0. Since we are interested in scalability to large key sets, this is done using an implementation of an external memory mergesort [21]. If the merge sort works in two phases, which is the case for all reasonable parameters, the total work on the disk consists of reading the keys, plus writing and reading the hash function values once. Since the h_0 hash values are relatively small (less than 15 decimal digits) we can use radix sort to do the internal memory sorting.

We have designed two versions of the EPH algorithm. The first one uses the linear hash function described in Section 3.4.2 that are slower and require more storage space, but guarantee that the EPH algorithm can be made to work for every key set. The second one uses faster and more compact pseudo random hash functions proposed by Jenkins [19]. This version is, from now on, referred to as *heuristic EPH algorithm* because it is not guaranteed that it can be made to work for every key set. However, empirical studies show that limited randomness properties are often as good as total randomness in practice and, the heuristic EPH has worked for all key sets we have applied it to.

The detailed description of the partitioning and searching steps are presented in Sections 3.1 and 3.2, respectively. The internal algorithm used to compute the MPHF of each bucket is from [5] and is presented in Section 3.3. The internal algorithm uses two hash functions h_{i1} and h_{i2} to compute a MPHF p_i. These hash functions as well as the hash function h_0 used in the partitioning step of the EPH algorithm are described in Section 3.4.

3.1 The Partitioning Step

The partitioning step performs two important tasks. First, the variable-length keys are mapped to 128-bit strings by using the linear hash function h' presented in Section 3.4. That is, the variable-length key set S is mapped to a fixed-length key set F. Second, the set S of n keys is partitioned into 2^b buckets, where b is a suitable parameter chosen to guarantee that each bucket has at most $\ell = 256$ keys with high probability (see Section 3.4). We have two reasons for choosing $\ell = 256$. The first one is to keep the buckets size small enough to be represented by 8-bit integers. The second one is to allow the memory accesses during the MPHF evaluation to be done in the cache most of the time. Figure 2 presents the partitioning step algorithm.

The critical point in Figure 2 that allows the partitioning step to work in linear time is the internal sorting algorithm.

- ▶ Let β be the size in bytes of the fixed-length key set F
- ▶ Let μ be the size in bytes of an a priori reserved internal memory area
- ▶ Let $N = \lceil \beta/\mu \rceil$ be the number of key blocks that will be read from disk into an internal memory area

1. **for** $j = 1$ **to** N **do**
   - 1.1 Read a key block S_j from disk (one at a time) and store $h'(x)$, for each $x \in S_j$, into $\mathcal{B}_j$, where $|\mathcal{B}_j| = \mu$
   - 1.2 Cluster $\mathcal{B}_j$ into 2^b buckets using an indirect radix sort algorithm that takes $h_0(x)$ for $x \in S_j$ as sorting key(i.e, the b most significant bits of $h'(x)$)
   - 1.3 Dump $\mathcal{B}_j$ to the disk into File j

Figure 2: Partitioning step

We have two reasons to choose radix sort. First, it sorts each key block $\mathcal{B}_j$ in linear time, since keys are short integer numbers (less than 15 decimal digits). Second, it just needs $O(|\mathcal{B}_j|)$ words of extra memory so that we can control the memory usage independently of the number of keys in S.

At this point one could ask: why not to use the well known replacement selection algorithm to build files larger than the internal memory area size? The reason is that the radix sort algorithm sorts a block $\mathcal{B}_j$ in time $O(|\mathcal{B}_j|)$ while the replacement selection algorithm requires $O(|\mathcal{B}_j| \log |\mathcal{B}_j|)$. We have tried out both versions and the one using the radix sort algorithm outperforms the other. A worthwhile optimization we have used is the last run optimization proposed by Larson and Graefe [21]. That is, the last block is kept in memory instead of dumping it to disk to be read again in the second step of the algorithm.

Figure 3(a) shows a *logical* view of the 2^b buckets generated in the partitioning step. In reality, the 128-bit strings belonging to each bucket are distributed among many files, as depicted in Figure 3(b). In the example of Figure 3(b), the 128-bit strings in bucket 0 appear in files 1 and N, the 128-bit strings in bucket 1 appear in files 1, 2 and N, and so on.

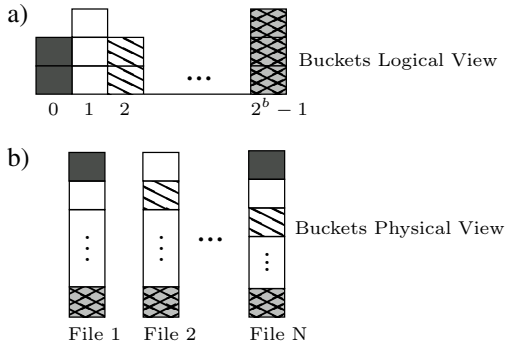

Figure 3: Situation of the buckets at the end of the partitioning step: (a) Logical view (b) Physical view

This scattering of the 128-bit strings in the buckets could generate a performance problem because of the potential number of seeks needed to read the 128-bit strings in each bucket from the N files on disk during the second step. But, as we show later on in Section 4.3, the number of seeks can be kept small using buffering techniques.

3.2 The Searching Step

The searching step is responsible for generating a MPHF for each bucket and for computing the offset array. Figure 4 presents the searching step algorithm. Statement 1 of Figure 4 constructs the heap H of size N. This is well known to be linear on N. The order relation in H is given by the bucket address i (i.e., the b most significant bits of $x \in F$). Statement 2 has two important steps. In statement 2.1, a bucket is read from disk, as described below. In statement 2.2, a MPHF is generated for each bucket B_i using the internal memory based algorithm presented in Section 3.3. In statement 2.3, the next entry of the offset array is computed. Finally, statement 2.4 writes the description of MPHF$_i$ and *offset*$[i]$ to disk. Note that to compute *offset*$[i+1]$ we just need the current bucket size and *offset*$[i]$. So, we just need to keep two entries of vector *offset* in memory all the time.

- ▶ Let H be a minimum heap of size N, where the order relation in H is given by $i = x[96, 127] >> (32 - b)$ for $x \in F$
1. **for** $j = 1$ **to** N **do** { Heap construction }
   1.1 Read the first 128-bit string x from File j on disk
   1.2 Insert (i, j, x) in H
2. **for** $i = 0$ **to** $2^b - 1$ **do**
   2.1 Read bucket B_i from disk driven by heap H
   2.2 Generate a MPHF for bucket B_i
   2.3 *offset*$[i+1]$ = *offset*$[i] + |B_i|$
   2.4 Write the description of MPHF$_i$ and *offset*$[i]$ to the disk

Figure 4: Searching step

The algorithm to read bucket B_i from disk is presented in Figure 5. Bucket B_i is distributed among many files and the heap H is used to drive a multiway merge operation. Statement 1.1 extracts and removes triple (i, j, x) from H, where i is a minimum value in H. Statement 1.2 inserts x in bucket B_i. Statement 1.3 performs a seek operation in File j on disk for the first read operation and reads sequentially all 128-bit strings $x \in F$ that have the same index i and inserts them all in bucket B_i. Finally, statement 1.4 inserts in H the triple (i', j, x'), where $x' \in F$ is the first 128-bit string read from File j (in statement 1.3) that does not have the same bucket address as the previous keys.

1. **while** bucket B_i is not full **do**
   1.1 Remove (i, j, x) from H
   1.2 Insert x into bucket B_i
   1.3 Read sequentially all 128-bit strings from File j that have the same i and insert them into B_i
   1.4 Insert the triple (i', j, x') in H, where x' is the first 128-bit string read from File j that does not have the same bucket index i

Figure 5: Reading a bucket

It is not difficult to see from this presentation of the searching step that it runs in linear time. To achieve this conclusion we use $O(N)$ computer words to allow the merge operation to be performed in one pass through each file. In addition, it is also important to observe that:

1. $2^b < \frac{n}{\log n}$ (see Section 3.4),
2. $N \ll n$ (e.g., see Table 5 in Section 4.3) and
3. the algorithm for the buckets runs in linear time, as shown in Section 3.3.

In conclusion, our algorithm takes $O(n)$ time because both the partitioning and the searching steps run in $O(n)$ time. The space required for constructing the resulting MPHF is $O(N)$ computer words because the memory usage in the partitioning step does not depend on the number of keys in S and, in the searching step, the algorithm used for the buckets is applied to problems of size up to 256. All together makes our algorithm the first one that demonstrates the capability of generating MPHFs for sets in the order of billions of keys.

3.3 The algorithm used for the buckets

For the buckets we decided to use one of the algorithms from the family $\mathcal{F}$ of algorithms[4] presented by Botelho, Pagh and Ziviani [5], because it outperforms the algorithms presented in Section 2 and also is a simple and near space-optimal way of constructing a minimal perfect hash function for a set S of n elements. They assume that it is possible to create and access two truly random hash functions $f_0 : U \rightarrow [0, \frac{m}{2} - 1]$ and $f_1 : U \rightarrow [\frac{m}{2}, m - 1]$, where $m = cn$ for $c > 2$. The Functions f_0 and f_1 are used to map the keys in S to a bipartite graph $G = (V, E)$, where $V = [0, m-1]$ and $E = \{\{f_0(x), f_1(x)\} \mid x \in S\}$ (i.e, $|E| = |S| = n$). Hence, each key in S is associated with only one edge from E. Figure 6(a) illustrates this step, referred to as *mapping step*, for a set S with three keys.

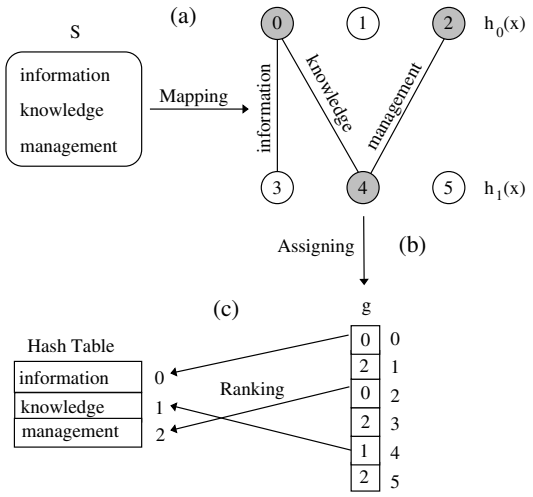

Figure 6: (a) Mapping step generates a bipartite graph (b) Assigning step generates a labeling g so that each edge is uniquely associated with one of its vertices (c) Ranking step builds a function rank : $V \rightarrow [0, n-1]$

In the following step, referred to as *assigning step*, a function $g : V \rightarrow \{0, 1, 2\}$ is computed so that each edge is uniquely represented by one of its vertices. For instance,

[4]The algorithms in $\mathcal{F}$ use r-uniform random hypergraphs given by function values of r hash functions on the keys of S. An r-graph is a generalization of a standard graph where each edge connects $r \geq 2$ vertices. For the buckets in the EPH algorithm we used the 2-uniform hypergraph instance.

in Figure 6(b), edge $\{0,3\}$ is associated with vertex 0, edge $\{0,4\}$ with vertex 4 and edge $\{2,4\}$ with vertex 2. Then, a function $\phi : S \rightarrow V$ defined as $\phi(x) = f_i(x)$, where $i = i(x) = (g(f_0(x)) + g(f_1(x))) \bmod 2$ is a perfect hash function on S.

The assigning step splits the set of vertices into two subsets: (i) the *assigned ones* and (ii) the *unassigned ones*. A vertex v is defined as assigned if $g(v) \neq 2$ and unassigned otherwise. Also, the number of assigned vertices is guaranteed by construction to be equal to $|S| = n$. Therefore, a function rank : $V \rightarrow [0, n-1]$ that counts how many vertices are assigned before a given assigned vertex $v \in V$ is a MPHF on V. For example, in Figure 6(c), rank(0) = 0, rank(2) = 1 and rank(4) = 2, which means that there is no assigned vertex before vertex 0, one before vertex 2, and two before vertex 4. This implies that a function $h : S \rightarrow [0, n-1]$ defined as $h(x) = \text{rank}(\phi(x))$ is a MPHF on S. The last step of the algorithm, referred to as *ranking step*, is responsible for computing the data structures used to compute function rank in time $O(1)$.

Botelho, Pagh and Ziviani [5] have shown that g can be generated in linear time if the bipartite graph $G = G(f_0, f_1)$ is acyclic. When a cyclic graph is generated a new pair (f_0, f_1) is randomly selected so that the values of f_0 and f_1 are truly random and independent. Hence the number of iterations to get an acyclic random graph must be bounded by a constant to finish the algorithm in linear time. They have shown that if $|V| = m = cn$, for $c > 2$, the probability of generating an acyclic bipartite random graph is $Pr_a = \sqrt{1-(2/c)^2}$ and the number of iterations is on average $N_i = 1/Pr_a$. For $c = 2.09$ we have $Pr_a \approx 0.29$ and $N_i \approx 3.4$. Finally, they have shown how to compress g and the data structures used to compute function rank in time $O(1)$ so that the resulting MPHFs are stored in $(3+\epsilon)n$ bits for $\epsilon > 0$.

3.4 Hash functions used by the EPH algorithm

The aim of this section is threefold. First, in Section 3.4.1, we define the hash functions h_{i1} and h_{i2} used by the algorithm from [5] to generate the MPHF of each bucket, where $0 \leq i \leq 2^b - 1$. Second, in Section 3.4.2, we show how to efficiently implement the hash functions h_{i1}, h_{i2}, and h_0, which is used to split the key set S into 2^b buckets. Third, in Section 3.4.3, we show the conditions that parameter b must meet so that no bucket with more than ℓ keys is created by h_0. We also show that h_{i1} and h_{i2} are truly random hash functions for the buckets. This section was mostly derived from the technical report Botelho, Pagh and Ziviani [6].

3.4.1 Hash functions used in the buckets

The hash functions h_{i1} and h_{i2} will be defined based on the linear hash functions over Galois field 2 (or simply GF(2)) analyzed by Alon, Dietzfelbinger, Miltersen and Petrank [1]. For that, a key is defined as a binary vector of fixed length L. This is not a restriction as any variable-length key can be padded with zeros at the end because the ascii character 0 does not appear in any string.

We define h_{i1} and h_{i2} as

$$\begin{aligned} h_{i1}(x) &= \rho(x, s_i, 0) \bmod |B_i| \\ h_{i2}(x) &= \rho(x, s_i, 1) \bmod |B_i| \end{aligned} \quad (2)$$

where

$$\rho(x,s,\Delta) = \left(\sum_{j=1}^{k} t_j[y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j[y_{j-k}(x) \oplus \Delta] \right) \bmod p.$$

The functions $y_1, \ldots, y_k$ are hash functions from $\{0,1\}^L$ to $\{0,1\}^{r-1}0$, where $2^r \gg \ell$ and k are parameters chosen to make the algorithm work with high probability (see Section 3.4.3). Note that the range is the set of r-bit strings ending with a 0. The purpose of the last 0 is to ensure that we can have no collision between $y_j(x_1)$ and $y_j(x_2) \oplus 1$, $1 \leq j \leq k$, for any pair of elements x_1 and x_2. The tables $t_1, \ldots, t_{2k}$ contain 2^r random values from $\{0, \ldots, p-1\}$, where p is a prime number much larger than $|B_i|$ (i.e., the size of the desired range of h_{i1} and h_{i2}). The variable s is a random seed number and the symbol $\oplus$ denotes exclusive-or. The variable s_i is specific to bucket i. The algorithm randomly selects s_i from $\{1, \ldots, p-1\}$ until the functions h_{i1} and h_{i2} work with the algorithm of Section 3.3, which is used to generate MPHFs for the buckets. We will sketch a proof in Section 3.4.3 that a constant fraction of the set of all functions works.

3.4.2 Implementation of the hash functions

The family of linear hash functions proposed by Alon, Dietzfelbinger, Miltersen and Petrank [1] enable us to implement the functions $h_0, y_1, y_2, y_3, \ldots, y_k$ to be computed at once. We use a function h' from that family that has the following form: $h'(x) = Ax$, where $x \in S$ and A is a $\gamma \times L$ matrix in which the elements are randomly chosen from $\{0,1\}$. The output is a bit string of an a priori defined size γ. In our implementation $\gamma = 128$ bits. It is important to realize that this is a matrix multiplication over GF (2). The implementation can be done using a bitwise-and operator (&) and a function $f : \{0,1\}^\gamma \rightarrow \{0,1\}$ to compute parity instead of multiplying numbers. The parity function $f(a)$ produces 1 as a result if $a \in \{0,1\}^\gamma$ has an odd number of bits set to 1, otherwise the result is 0. For example, let us consider $L = 3$ bits, $\gamma = 3$ bits, $x = 110$ and

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

The number of rows gives the required number of bits in the output, i.e., $\gamma = 3$. The number of columns corresponds to the value of L. Then,

$$h'(x) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

where $b_1 = f(101 \ \& \ 110) = 1$, $b_2 = f(001 \ \& \ 110) = 0$ and $b_3 = f(110 \ \& \ 110) = 0$.

To get a fast evaluation time we use a tabulation idea from [2], where we can get evaluation time $O(L/\log\sigma)$ by using space $O(L\sigma)$ for $\sigma > 0$. Note that if x is short, e.g. 8 bits, we can simply tabulate all the function values and compute $h'(x)$ by looking up the value $h'[x]$ in an array h'. To make the same thing work for longer keys, split the matrix A into parts of 8 columns each: $A = A_1|A_2|\ldots|A_{\lceil L/8 \rceil}$, and create a lookup table h'_i for each submatrix. Similarly split x into parts of 8 bits, $x = x_1 x_2 \ldots x_{\lceil L/8 \rceil}$. Now $h'(x)$ is the exclusive-or of $h'_i[x_i]$, for $i = 1, \ldots, \lceil L/8 \rceil$. Therefore, we have set σ to 256 so that keys of size L can be processed

in chunks of $\log \sigma = 8$ bits. In our URL collection the largest key has 65 bytes, i.e., $L = 520$ bits.

The 32 most significant bits of $h'(x)$, where $x \in S$, are used to compute the bucket address of x, i.e., $h_0(x) = h'(x)[96, 127] >> (32 - b)$. We use the symbol $>>$ to denote the right shift of bits. The other 96 bits correspond to $y_1(x), y_2(x), \ldots y_6(x)$, taking $k = 6$. This would give $r = 16$, however, to save space for storing the tables used for computing h_{i1} and h_{i2}, we hard coded the linear hash function to make the most and the least significant bit of each chunk of 16 bits equal to zero. Therefore, $r = 15$. This setup enable us to solving problems of up to 500 billion keys, which is plenty of for all the applications we know of. If our algorithm fails in any phase, we just restart it. As the parameters are chosen to have success with high probability, the number of times that our algorithm is restarted is $O(1)$.

Finally, the last parameter related to the hash functions we need to talk about is the prime number p. As p must be much larger than the range of h_{i1} and h_{i2}, then we set it to the largest 32-bit integer that is a prime, i.e, $p = 4294967291$.

3.4.3 Analyzes of the hash functions

In this section we show that the EPH algorithm can be made to work with high probability. For that we need to analyze the following points:

1. The function h' discussed in Section 3.4.2 that was used to implement $h_0, y_1, y_2, \ldots, y_k$ maps the variable-length keys from S to a fixed-length key set F, where each key contains $b + kr$ bits. As the keys in S are assumed to be all distinct, then all keys in F should be distinct as well. We will show that this is the case with high probability.

   Proof sketch: as the function h' comes from a family of universal hash functions [1], the probability that there exist two keys that have the same values under all functions is at most $\binom{n}{2}/2^{b+kr}$. This probability can be made negligible by choosing k and r appropriately.

2. We have imposed an upper bound ℓ on the size of the largest bucket created by h_0. Therefore, we will use a result from [1] to argue that if

   $$b \leq \log n - \log(\ell / \log \ell) + O(1), \qquad (3)$$

   then the maximum bucket size is bounded by ℓ with high probability (some constant close to 1). For the implementation, we will experimentally determine the smallest possible choices of b.

   Proof sketch: a direct consequence of Theorem 5 in [1] is that, assuming $b \leq \log n - \log \log n$, the expected size of the largest bucket is $O(n \log b / 2^b)$, i.e., a factor $O(\log b)$ from the average bucket size. So, by imposing the requirement that $\ell \geq \log n \log \log n$ we justify the choice of b in Eq. (3).

3. we will now analyze the probability (over the choice of $y_1, \ldots, y_k$) that $x \mapsto \rho(x, s_i, 0)$ and $x \mapsto \rho(x, s_i, 1)$ map the elements of B_i uniformly and independently to $\{0, \ldots, p-1\}$, for any choice of the random seed s_i.

   Proof sketch: a sufficient criterion for this is that the sums $\sum_{j=1}^{k} t_j[y_j(x) \oplus \Delta]$ and $\sum_{j=k+1}^{2k} t_j[y_{j-k}(x) \oplus \Delta]$, $\Delta \in \{0, 1\}$, have values that are uniform in $\{0, \ldots, p-1\}$ and independent. This is the case if for every $x \in B_i$ there exists an index j_x such that neither y_{j_x} nor $y_{j_x} \oplus 1$ belongs to $y_{j_x}(B_i - \{x\})$. Since $y_1, \ldots, y_k$ are universal hash functions, the probability that this is not the case for a given element $x \in B_i$ is bounded by $(|B_i|/2^r)^k \leq (\ell/2^r)^k$. If we choose, for example $r = \lceil \log(\sqrt[3]{n}\ell) \rceil$ and $k = 6$ we have that this probability is $o(1/n)$. Hence, the probability that this happens for *any* key in S is $o(1)$.

4. Finally, we need to show that it is possible to obtain, with constant probability, a value of s_i such that the pair (h_{i1}, h_{i2}) works for the MPHF of the bucket i. That is, the functions h_{i1} and h_{i2} should be truly random.

   Proof sketch: as argued above, the functions $x \mapsto \rho(x, s_i, 0)$ and $x \mapsto \rho(x, s_i, 1)$ are random and independent on each bucket, for every value of s_i. Then, for a given bucket and a given value of s_i there is a probability $\Omega(1)$ that the pair of hash functions work for that bucket. Therefore, for any $\Delta \in \{0, 1\}$ and $s_i \neq s_j$, the functions $x \mapsto \rho(x, s_i, \Delta)$ and $x \mapsto \rho(x, s_j, \Delta)$ are independent. Thus, by Chebychev's inequality the probability that less than a constant fraction of the values of s_i work for a given bucket is $O(1/p)$. So with probability $1 - o(1)$ there is a constant fraction of "good" choices of s_i in every bucket, which means that trying an expected constant number of random values for s_i is sufficient in each bucket.

4. EXPERIMENTAL RESULTS

In this section we present the experimental results. We start presenting the experimental setup. We then present the performance of our algorithm considering construction time, storage space and evaluation time as metrics for the resulting functions. Finally, we discuss how the amount of internal memory available affects the runtime of our two-step external memory based algorithm.

4.1 The data and the experimental setup

The EPH algorithm was implemented in the C language and is available at `http://cmph.sf.net` under the GNU Lesser General Public License (LGPL). All experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1 gigahertz AMD Athlon 64 Processor 3200+ and 1 gigabyte of main memory.

Our data consists of a collection of 1.024 billion URLs collected from the Web, each URL 64 characters long on average. The collection is stored on disk in 60.5 gigabytes of space.

4.2 Performance of the EPH algorithm

We are firstly interested in verifying the claim that the EPH algorithm runs in linear time. Therefore, we run the algorithm for several numbers n of keys in S. The values chosen for n were 32, 128, 512 and 1024 million. We limited the main memory in 512 megabytes for the experiments in order to show that the algorithm does not need much internal memory to generate MPHFs. The size μ of the a priori reserved internal memory area was set to 200 megabytes. In Section 4.3 we show how μ affects the runtime of the algorithm. The parameter b (see Eq. (3)) was set to the minimum value that gives us a maximum bucket size lower

than $\ell = 256$. For each value chosen for n, the respective values for b are $18, 20, 22$ and 23 bits.

In order to estimate the number of trials for each value of n we use a statistical method for determining a suitable sample size (see, e.g., [18, Chapter 13]). We got that just one trial for each n would be enough with a confidence level of 95%. However, we made 10 trials. This number of trials seems rather small, but, as shown below, the behavior of the EPH algorithm is very stable and its runtime is almost deterministic (i.e., the standard deviation is very small) because it is a random variable that follows a (highly concentrated) normal distribution.

Table 1 presents the runtime average for each n, the respective standard deviations, and the respective confidence intervals given by the average time $\pm$ the distance from average time considering a confidence level of 95%. Observing the runtime averages we noticed that the algorithm runs in expected linear time, as we have claimed. Better still, it outputs the resulting MPHF faster than all practical algorithms we know of, because of the following reasons. First, we make good use of the memory hierarchy because the memory accesses during the generation of a MPHF for a given bucket cause cache hits, once the problem was broken down into problems of size up to 256. Second, at searching step we are dealing with 16-byte (128-bit) strings instead of 64-byte URLs. The percentages of the total time spent in the partitioning step and in the searching are approximately 49% and 51%, respectively.

n (millions)	32	128	512	1024
EPH	2.5 ± 0.02	10.1 ± 0.1	39.7 ± 0.4	83.0 ± 0.9
Heuristic EPH	1.9 ± 0.05	7.3 ± 0.1	30.9 ± 0.5	64.3 ± 1.1

Table 1: EPH algorithm: average time in minutes for constructing a MPHF with confidence level of 95% in a PC using 200 megabytes of internal memory.

In Table 1 we see that the runtime of the algorithm is almost deterministic. A given bucket i, $0 \leq i < 2^b$, is a small set of keys (at most 256 keys) and, the runtime of the building block algorithm is a random variable X_i with high fluctuation (it follows a geometric distribution with mean $1/Pr \approx 3.4$). However, the runtime Y of the searching step of the EPH algorithm is given by $Y = \sum_{0 \leq i < 2^b} X_i$. Under the hypothesis that the X_i are independent and bounded, the *law of large numbers* (see, e.g., [18]) implies that the random variable $Y/2^b$ converges to a constant as $n \to \infty$. This explains why the runtime is almost deterministic.

The next important metric on MPHFs is the space required to store the functions. In order to apply the algorithm used for the buckets to larger sets we randomly choose f_0 and f_1 from the family of universal hash functions proposed by Thorup [32]. Botelho, Pagh and Ziviani [5] have analyzed that algorithm under the full randomness assumption so that universal hashing is not enough to guarantee that it works for every key set. But it has been the case for every key set we have applied it to. Then, we refer to this version as *heuristic BPZ algorithm*.

The EPH algorithm is designed to be used when the key set does not fit in main memory. Table 2 shows that it can be used for constructing PHFs and MPHFs that require on average 2.7 and 3.8 bits per key to be stored, respectively.

The lookup tables used by the hash functions of the EPH algorithm require a fixed storage cost of 1,847,424 bytes. To avoid the space needed for lookup tables we have implemented the heuristic EPH algorithm. It uses the pseudo random hash function proposed by Jenkins [19] to replace the hash functions described in Section 3.4. The Jenkins function just loops around the key doing bitwise operations over chunks of 12 bytes. Then, it returns the last chunk. Thus, in the mapping step, the key set S is mapped to F, which now contains 12-byte long strings instead of 16-byte long strings.

n	b	Bits/key	
		PHF	MPHF
10^4	6	2.93	3.71
10^5	9	2.73	3.57
10^6	13	2.65	3.82
10^7	16	2.51	3.70
10^8	20	2.80	4.02
10^9	23	2.65	3.83

Table 2: EPH algorithm: space usage to respectively store the resulting PHFs and MPHFs.

The Jenkins function needs just one random seed of 32 bits to be stored instead of quite long lookup tables, a great improvement from the 1,847,424 bytes necessary to implement truly random hash functions. Therefore, there is no fixed cost to store the resulting MPHFs, but two random seeds of 32 bits are required to describe the functions h_{i1} and h_{i2} of each bucket. As a consequence, the MPHFs generation and the MPHFs efficiency at retrieval time are faster (see Table 3 and 4). The reasons are twofold. First, we are dealing with 12-byte strings computed by the Jenkins function instead of 16-byte strings of the truly random functions presented in Section 3.4. Second, there are no large lookup tables to cause *cache misses*. For example, the construction time for a set of 1024 million keys has dropped down to 64.3 minutes in the same setup. The disadvantage of using the Jenkins function is that there is no formal proof that it works for every key set. That is why the hash functions we have designed in this paper are required, even being slower. In the implementation available, the hash functions to be used can be chosen by the user.

Table 3 presents a comparison of our algorithm with the ones proposed by Botelho, Pagh and Ziviani [5] (BPZ), by Pagh [27] (Hash-displace), by Botelho, Kohayakawa and Ziviani [4] (BKZ), by Czech, Havas and Majewski [10] (CHM), and by Fox, Chen and Heath [13] (FCH), considering construction time and storage space as metrics. Notice that they are the most important practical results on MPHFs known in the literature. Observing the results, the EPH algorithm is the fastest one at construction time and the heuristic BPZ algorithm builds slightly more compact functions.

Finally, we show how efficient are the resulting MPHFs at retrieval time for the methods aforementioned, which is as important as construction time and storage space. Table 4 presents the time, in seconds, to evaluate 2×10^6 keys. We group the BKZ and CHM methods together because the resulting MPHFs have the same form. The MPHFs generated by the EPH algorithm are slower. Nevertheless, the difference is not so expressive (each key can be evaluated in few microseconds) and the EPH algorithm is the first efficient option for sets that do not fit in main memory.

Time in seconds to construct a MPHF for 2×10^6 keys			
Algorithms	Function type	Construction time (seconds)	bits/key
EPH Algorithm	PHF	6.92 ± 0.04	2.64
	MPHF	6.98 ± 0.01	3.85
Heuristic EPH Algorithm	MPHF	4.75 ± 0.02	3.7
Heuristic BPZ Algorithm	PHF	12.99 ± 1.01	2.09
	MPHF	13.94 ± 1.06	3.35
Hash-displace	MPHF	46.18 ± 1.06	64.00
BKZ	MPHF	8.57 ± 0.31	32.00
CHM	MPHF	13.80 ± 0.70	66.88
FCH	MPHF	758.66 ± 126.72	5.84

Table 3: Construction time and storage space without considering the fixed cost to store lookup tables.

Time in seconds to evaluate 2×10^6 keys						
key length (bytes)	Function type	8	16	32	64	128
EPH Algorithm	PHF	2.05	2.31	2.84	3.99	7.22
	MPHF	2.55	2.83	3.38	4.63	8.18
Heuristic EPH Algorithm	MPHF	1.19	1.35	1.59	2.11	3.34
Heuristic BPZ Algorithm	PHF	0.41	0.55	0.79	1.29	2.39
	MPHF	0.85	0.99	1.23	1.73	2.74
Hash-displace	MPHF	0.56	0.69	0.93	1.44	2.54
BKZ/CHM	MPHF	0.61	0.74	0.98	1.48	2.58
FCH	MPHF	0.58	0.72	0.96	1.46	2.56

Table 4: Evaluation time.

It is important to emphasize that the BPZ, BKZ, CHM and FCH methods were analyzed under the full randomness assumption. Therefore, the EPH algorithm is the first one that has experimentally proven practicality for large key sets and has both space usage for representing the resulting functions and the construction time carefully proven. Additionally, it is the fastest algorithm for constructing the functions and the resulting functions are much simpler than the ones generated by previous theoretical well-founded schemes so that they can be used in practice. Also, it considerably improves the first step given by Pagh with his hash and displace method [27].

4.3 Controlling disk accesses

In order to bring down the number of seek operations on disk we benefit from the fact that the EPH algorithm leaves almost all main memory available to be used as disk I/O buffer. In this section we evaluate how much the parameter μ affects the runtime of the EPH algorithm. For that we fixed n in 1.024 billion of URLs, set the main memory of the machine used for the experiments to 1 gigabyte and used μ equal to 100, 200, 300, 400 and 500 megabytes.

In order to amortize the number of seeks performed we use a buffering technique [20]. We create a buffer j of size $\mathbb{B} = \mu/N$ for each file j, where $1 \leq j \leq N$. Every time a read operation is requested to file j and the data is not found in the jth buffer, $\mathbb{B}$ bytes are read from file j to buffer j. Hence, the number of seeks performed in the worst case is given by $\beta/\mathbb{B}$ (remember that β is the size in bytes of the fixed-length key set F). For that we have made the pessimistic assumption that one seek happens every time buffer j is filled in. Thus, the number of seeks performed in the worst case is $16n/\mathbb{B}$, since after the partitioning step we are dealing with 128-bit (16-byte) strings instead of 64-byte URLs, on average. Therefore, the number of seeks is linear on n and amortized by $\mathbb{B}$. It is important to emphasize that the operating system uses techniques to diminish the number of seeks and the average seek time. This makes the amortization factor to be greater than $\mathbb{B}$ in practice.

Table 5 presents the number of files N, the buffer size used for all files, the number of seeks in the worst case considering the pessimistic assumption aforementioned, and the time to generate a PHF or a MPHF for 1.024 billion of keys as a function of the amount of internal memory available. Observing Table 5 we noticed that the time spent in the construction decreases as the value of μ increases. However, for $\mu > 400$, the variation on the time is not as significant as for $\mu \leq 400$. This can be explained by the fact that the kernel 2.6 I/O scheduler of Linux has smart policies for avoiding seeks and diminishing the average seek time (see `http://www.linuxjournal.com/article/6931`).

μ (MB)	100	200	300	400	500
N (files)	245	99	63	46	36
$\mathbb{B}$ (in KB)	418	2,069	4,877	8,905	14,223
$\beta/\mathbb{B}$	151,768	30,662	13,008	7,124	4,461
EPH (time)	94.8	82.2	79.8	79.2	79.2
Heuristic EPH (time)	71.0	63.2	62.9	62.4	62.4

Table 5: Influence of the internal memory area size (μ) in the EPH algorithm runtime to construct PHFs or MPHFs for 1.024 billion keys (time in minutes).

5. CONCLUDING REMARKS

This paper has presented a novel external memory based algorithm for constructing PHFs and MPHFs. The algorithm can be used with provably good hash functions or with heuristic hash functions that are faster to compute.

The EPH algorithm contains, as a component, a provably good implementation of the BPZ algorithm [5]. This means that the two hash functions h_{i1} and h_{i2} (see Eq. (3)) used instead of f_0 and f_1 behave as truly random hash functions (see Section 3.4.3). The resulting PHFs and MPHFs require approximately 2.7 and 3.8 bits per key to be stored and are generated faster than the ones generated by all previous methods. The EPH algorithm is the first one that has experimentally proven practicality for sets in the order of billions of keys and has time and space usage carefully analyzed without unrealistic assumptions. As a consequence, the EPH algorithm will work for every key set.

The resulting functions of the EPH algorithm are approximately four times slower than the ones generated by all previous practical methods (see Table 4). The reason is that to compute the involved hash functions we need to access lookup tables that do not fit in the cache. To overcome this problem, at the expense of losing the guarantee that it works for every key set, we have proposed a heuristic version of the EPH algorithm that uses a very efficient pseudo random hash function proposed by Jenkins [19]. The resulting functions require the same storage space, are now less than twice slower to be computed and are still faster to be generated.

6. ACKNOWLEDGMENTS

We thank Rasmus Pagh for helping us with the analysis of the EPH algorithm, which was previously published in Botelho, Pagh and Ziviani [6]. We also thank the partial support given by GERINDO Project–grant MCT/CNPq/CT-INFO 552.087/02-5, and CNPq Grants 30.5237/02-0 (Nivio Ziviani) and 142786/2006-3 (Fabiano C. Botelho).

7. REFERENCES

[1] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *J. of the ACM*, 46(5):667–683, 1999.

[2] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[3] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proc. of the 13th Intl. World Wide Web Conference*, pages 595–602, 2004.

[4] F. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In *Proc. of the 4th Intl. Workshop on Efficient and Experimental Algorithms*, pages 488–500. Springer LNCS, 2005.

[5] F. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*, pages 139–150. Springer LNCS, 2007.

[6] F. C. Botelho, R. Pagh, and N. Ziviani. Perfect hashing for data management applications. Technical Report TR002/07, Federal University of Minas Gerais, 2007. Available at http://arxiv.org/pdf/cs/0702159.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th Intl. World Wide Web Conference*, pages 107–117, April 1998.

[8] C.-C. Chang and C.-Y. Lin. A perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.

[9] C.-C. Chang, C.-Y. Lin, and H. Chou. Perfect hashing schemes for mining traversal patterns. *J. of Fundamenta Informaticae*, 70(3):185–202, 2006.

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

[11] Z. Czech, G. Havas, and B. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[12] M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *Proc. of the 9th European Symposium on Algorithms*, pages 109–120. Springer LNCS vol. 2161, 2001.

[13] E. Fox, Q. Chen, and L. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proc. of the 15th Intl. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 266–273, 1992.

[14] E. Fox, L. S. Heath, Q. Chen, and A. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[15] M. L. Fredman, J. Komlós, and E. Szemerédi. On the size of separating systems and families of perfect hashing functions. *SIAM J. on Algebraic and Discrete Methods*, 5:61–68, 1984.

[16] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *J. of the ACM*, 31(3):538–544, July 1984.

[17] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. of the 18th Symposium on Theoretical Aspects of Computer Science*, pages 317–326. Springer LNCS vol. 2010, 2001.

[18] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley, first edition, 1991.

[19] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's J. of Software Tools*, 22(9), 1997.

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

[21] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proc. of the 1998 ACM SIGMOD intl. conference on Management of data*, pages 472–483. ACM Press, 1998.

[22] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, 2006.

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

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

[25] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.

[26] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Proc. of the 16th ACM-SIAM symposium on Discrete algorithms*, pages 823–829, 2005.

[27] R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures*, pages 49–54, 1999.

[28] B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In *Proc. of the IEEE International Symposium on Information Theory*. IEEE Press, 2006.

[29] J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41:203–207, 1992.

[30] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. on Computing*, 19(5):775–786, October 1990.

[31] M. Seltzer. Beyond relational databases. *ACM Queue*, 3(3), April 2005.

[32] M. Thorup. Even strongly universal hashing is pretty fast. In *Proc. of the 11th ACM-SIAM symposium on Discrete algorithms*, pages 496–497, 2000.

[33] P. Woelfel. Maintaining external memory efficient hash tables. In *Proc. of the 10th International Workshop on Randomization and Computation (RANDOM'06)*, pages 508–519. Springer LNCS vol. 4110, 2006.