BRICS

Basic Research in Computer Science



# Lossy Dictionaries

**Rasmus Pagh**
**Flemming Friche Rodler**

**BRICS Report Series** **RS-01-33**

**ISSN 0909-0878** **August 2001**

**Copyright © 2001, Rasmus Pagh & Flemming Friche Rodler.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
**This document in subdirectory** `RS/01/33/`

# Lossy Dictionaries

Rasmus Pagh*and Flemming Friche Rodler
**BRICS**†
Department of Computer Science
University of Aarhus, Denmark
{pagh,ffr}@brics.dk

August, 2001

**Abstract**

Bloom filtering is an important technique for space efficient storage of a conservative approximation of a set $S$. The set stored may have up to some specified number of "false positive" members, but all elements of $S$ are included. In this paper we consider *lossy dictionaries* that are also allowed to have "false negatives", i.e., leave out elements of $S$. The aim is to maximize the weight of included keys within a given space constraint. This relaxation allows a very fast and simple data structure making almost optimal use of memory. Being more time efficient than Bloom filters, we believe our data structure to be well suited for replacing Bloom filters in some applications. Also, the fact that our data structure supports information associated to keys paves the way for new uses, as illustrated by an application in lossy image compression.

## 1 Introduction

Dictionaries are part of many algorithms and data structures. A dictionary provides access to information indexed by a set $S$ of *keys*: Given a key, it returns the associated information or reports that the key is not in the set. In this paper we will not be concerned with updates, i.e., we consider the *static* dictionary problem. The main parameters of interest are of course the space used by the dictionary and the time for looking up information. We will assume keys as well as the information associated with keys to have a fixed size.

A large literature has grown around the problem of constructing efficient dictionaries, and theoretically satisfying solutions have been found. Often a slightly easier problem has been considered, namely the *membership* problem, which is the dictionary problem without associated information. It is usually easy to derive a dictionary from a solution to the membership problem, using

*Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

extra space corresponding to the associated information. In this paper we are particularly interested in dictionary and membership schemes using little memory. Let $n$ denote the size of the key set $S$. It has been shown that when keys are $w$-bit machine words, lookups can be performed in constant time in a membership data structure occupying $B + o(B)$ bits of memory, where $B = \log \binom{2^w}{n}$ is the minimum amount of memory needed to be able to represent any subset of size $n$ [2] (logarithms in this paper are base 2). However, constant factors in the lower order term and lookup time make this and similar schemes less than one could hope for from an applied point of view. Also, difficulty of implementation is an obstacle to practical use. In total, current schemes with asymptotically optimal space usage appear to be mainly of theoretical interest.

If one relaxes the requirements to the membership data structure, allowing it to store a slightly different key set than intended, new possibilities arise. A technique finding many applications in practice is *Bloom filtering* [1]. This technique allows space-efficient storage of a superset $S'$ of the key set $S$, such that $S' \setminus S$ is no more than an $\epsilon$ fraction of $\{0,1\}^w$. For $n \ll 2^w$, about $\log(1/\epsilon)$ bits per key in $S$ are necessary and sufficient for this [4]. This is a significant savings compared to a membership data structure using $B \approx n \log(\frac{2^w e}{n})$ bits. Lookup of a key using Bloom filtering requires $O(\log(1/\epsilon))$ memory accesses, and is thus relatively slow compared to other hashing schemes when $\epsilon$ is small. Also, Bloom filtering differs from most other hashing techniques in that it does *not* yield a solution to the dictionary problem.

### 1.1 This Paper

In this paper we introduce the concept of *lossy dictionaries* that can have not only false positives (like Bloom filters), but also false negatives. That is, some keys in $S$ (with associated information) are thrown away when constructing the dictionary. For false positives there is no guarantee on the associated information returned. We let each key in $S$ have a weight, and try to maximize the sum of weights of keys in the dictionary under a given space constraint.

We study this problem on a unit cost RAM, in the case where keys are machine words of $w$ bits, examining a very simple and efficient data structure from a theoretical as well as an experimental point of view. Experimentally, we find that our data structure has surprisingly good behavior with respect to keeping the keys of largest weight. The experimental results are partially explained by our theoretical considerations, under strong assumptions on the hash functions involved. Specifically, we assume that in our RAM model, for a number of random functions, arbitrary function values can be returned in constant time by an oracle. We also show that our data structure is nearly optimal with respect to space usage.

### 1.2 Applications

A *cache* can be seen as a dictionary that stores a small subset of a large key set, plus associated information. It is thus inherently lossy. A lossy dictionary allowed to discard a small fraction of a key set may thus in many cases be a quite

acceptable implementation. If no wrong information is to be returned, we can allow no false positives. Our lossy dictionary seems best suited for aplications where the cache only changes periodically, as for example in Web caching.

*Web cache sharing* [7] is a technique for implementing cooperating caches, for example Web proxies. When a request arrives at a proxy, it first checks whether it can answer the request. If not, it can forward the request to other proxies in the network. However, this increases traffic and is rather expensive. In cooperative caching each proxy keeps a summary of the content of all relevant proxies available to it. To reduce space requirements, this summary is stored with a small fraction of error using Bloom filtering. Often this reduces network traffic dramatically, since there is no more than a small chance that an expensive request forwarding is performed in vain. Lossy dictionaries with two-sided error could be used as a summary rather than a Bloom filter, since a small fraction of false negatives (cache misses) is tolerable.

In fact, the general idea of using in-memory summaries to reduce the number of expensive operations, such as I/O's, is well known in the database community. It dates at least back to [18], which uses Bloom filtering for efficient management of different versions of databases.

Recently, interest in lossy (volume) data compression with fast random access to decoded data has arisen [9, 11, 16, 17]. In [17] we show that lossy dictionaries are well suited for this purpose, providing lossy storage of the coefficients of wavelet transformed data. Compared to the previously best methods in the literature [9, 16], our lossy dictionary based scheme typically performs 80% better in terms of compression ratio, while significantly reducing the random access time.

### 1.3 Related Work

Most previous work on static dictionaries has considered the membership problem on a unit cost RAM with word size $w$. The first membership data structure with worst case constant lookup time using $O(n)$ words of space was constructed by Fredman et al. [8]. For constant $\delta > 0$, the space usage is $O(B)$ when $2^w > n^{1+\delta}$, but in general the data structure may use $\Omega(Bw)$ bits of space. The space usage has been lowered to $B + o(B)$ bits by Brodnik and Munro [2]. The lower order term was subsequently improved to $o(n) + O(\log w)$ bits by the first author [13]. The main concept used in the latter paper is that of a *quotient function* $q$ of a hash function $h$, defined to be a function such that the mapping $k \mapsto (h(k), q(k))$ is injective.

The membership problem with false positives was first considered by Bloom [1]. He described a technique, now known as *Bloom filtering*, where lookups return the conjunction of a number of bits from a bit vector. The locations of the bit probes are the values of a series of hash functions on the element to be looked up. Apart from Bloom filtering the paper presents a less space efficient data structure that is readily turned into a lossy dictionary with only false positives. However, the space usage of the derived lossy dictionary is not optimal. Carter et al. [4] provided a lower bound of $n \log(1/\epsilon)$ bits on the space needed to solve membership with an $\epsilon$ fraction false positives, for $n \ll 2^w$, and gave data

structures with various lookup times matching or nearly matching this bound. Though none of their membership data structures have constant lookup time, such a data structure follows by plugging the abovementioned results on space optimal membership data structures [2, 13] into a general reduction provided in [4]. In fact, the dictionary of [13] can be easily modified to a lossy dictionary with false positives, thus also supporting associated information, using $O(n + \log w)$ bits more than the lower bound.

Another relaxation of the membership problem was recently considered by Buhrman et al. [3]. They store the set $S$ exactly, but allow the lookup procedure to use randomization and to have some probability of error. For two-sided error $\epsilon$ they show that there exists a data structure of $O(nw/\epsilon^2)$ bits in which lookups can be done using just *one* bit probe. It was shown that to do the same without false negatives, $O(n^2w/\epsilon^2)$ bits suffice, and that this is essentially optimal. Schemes using more bit probes and less space were also investigated. If one fixes the random bits of the lookup procedure appropriately, the result is a lossy dictionary with error $\epsilon$. However, it is not clear how to efficiently guarantee the $\epsilon$ fraction of false positives in a reasonable model of computation, so this does not immediately give rise to a lossy dictionary.

## 2 Lossy Dictionaries

Consider a set $S$ containing keys $k_1, \ldots, k_n$ with associated information $a_1, \ldots, a_n$ and positive weights $v_1, \ldots, v_n$. Suppose we are given an upper bound $m$ on available space and an error parameter $\epsilon > 0$. The *lossy dictionary problem* for $\epsilon = 0$ is to store a subset of the keys in $S$ and corresponding associated information in a data structure of $m$ bits, trying to optimize the sum of weights of included keys. For general $\epsilon$ we also allow the dictionary to contain $2^w\epsilon$ keys from the complement of $S$. In this section we show the following theorem.

**Theorem 1** *Let a sequence of keys $k_1, \ldots, k_n \in \{0,1\}^w$, associated information $a_1, \ldots, a_n \in \{0,1\}^l$, and weights $v_1 \geq \cdots \geq v_n > 0$ be given. Let $r > 0$ be an even integer, and $b \geq 0$ an integer. Suppose we have oracle access to random functions $h_1, h_2 : \{0,1\}^w \to \{1, \ldots, r/2\}$ and corresponding quotient functions $q_1, q_2 : \{0,1\}^w \to \{0,1\}^s \setminus 0^s$. There is a lossy dictionary with the following properties:*

1. *The space usage is $r(s - b + l)$ bits (two tables with $r/2$ cells of $s - b + l$ bits).*
2. *The fraction of false positives is bounded by $\epsilon \leq (2^b - 1)r/2^w$.*
3. *The expected weight of the keys in the set stored is $\sum_{i=1}^{n} p_{r,i}\, v_i$ where*

$$p_{r,i} \geq \begin{cases} 1 - 52\, r^{-1}/(\frac{r}{i} - 2), & \text{for } i < r/2 \\ 2\,(1 - 2/r)^{i-1} - (1 - 2/r)^{2(i-1)}, & \text{for } i \geq r/2 \end{cases}$$

*is the probability that $k_i$ is included in the set (which is independent of $v_i$).*

4. *Lookups are done using at most two (independent) accesses to the tables.*

5. *The construction time is* $O(n \log^* n + rl/w)$.

As discussed in Section 2.1 there exist quotient functions for $s = w - \log(r/2) + O(1)$ if the hash functions map approximately the same number of elements to each value in $\{1, \ldots, r/2\}$. The inequality in item 2 is satisfied for $b = \lfloor \log(2^w \epsilon / r + 1) \rfloor$, so for $s = w - \log r + O(1)$ an $\epsilon$ fraction of false positives can be achieved using space $r\,(\log(\frac{1}{\epsilon + r/2^w}) + l + O(1))$. As can be seen from item 3, almost all of the keys $\{k_1, \ldots, k_{r/2}\}$ are expected to be included in the set represented by the lossy dictionary. For $i \geq r/2$ our bound on $p_{i,r}$ is shown in Figure 5 of Section 3, together with experimentally observed probabilities. If $n \geq r$ and if $r$ is large enough, it can be shown by integration that, in the expected sense, more than 70% of the keys from $\{k_1, \ldots, k_r\}$ are included in the set (our experiments indicate 84%). We show in Section 2.5 that the amount of space that we use to achieve this is within a small constant factor of optimal.

Note that by setting $b = 0$ we obtain a lossy dictionary with no false positives. Another point is that given a desired maximum space usage $m$ and false positive fraction $\epsilon$, the largest possible size $r$ of the tables can be usually be chosen efficiently. Assume, for example, that we have quotient function with range $\lceil \log(2^{w+1}/r) \rceil$ and consider the case $b = 0$. The memory usage is $r(\lceil \log(2^{w+1}/r) \rceil + l)$. Whenever $r$ is doubled, the number of bits per cell becomes one less. This means that the memory usage increases piecewise linearly in $r$, with jumps when $r$ is a power of two. By setting $r = 2^i$ for $i = 1, 2, 3, \ldots$ we find the $i$ for which $m$ is first exceeded. The correct value of $r$ can now easily be found in the interval $2^{i-1} < r < 2^i$. For general $b$ this becomes more complicated, as we need to investigate more intervals, but finding $r$ is still implementable in $O(\log m)$ time.

## 2.1 Preliminaries

The starting point for the design of our data structure is a static dictionary recently described in [14]. In this dictionary, two hash tables $T_1$ and $T_2$ are used together with two hash functions $h_1, h_2 : \{0,1\}^w \to \{1, \ldots, r/2\}$, where $r$ denotes the combined size of the hash tables, assumed to be even. A key $x \in S$ is stored in either cell $h_1(x)$ of $T_1$ or cell $h_2(x)$ of $T_2$. It was shown that if $r \geq (2 + \delta)\, n$, for constant $\delta > 0$, and $h_1, h_2$ are random functions, there exists a way of arranging the keys in the tables according to the hash functions with probability at least $1 - \frac{52}{\delta r}$. For small $\delta$ this gives a dictionary utilizing about 50% of the hash table cells. The arrangement of keys was shown to be computable in expected linear time.

Another central concept is that of *quotient functions*. Recall that a quotient function $q$ of a hash function $h$ is a function such that the mapping $k \mapsto (h(k), q(k))$ is injective [13]. When storing a key $k$ in cell $h(k)$ of a hash table, it is sufficient to store $q(k)$ to uniquely identify $k$ among all other elements hashing to $h(k)$. To mark empty cells, one needs a bit string not mapped to by the quotient function, e.g. $0^s$ for the quotient functions of Theorem 1. The

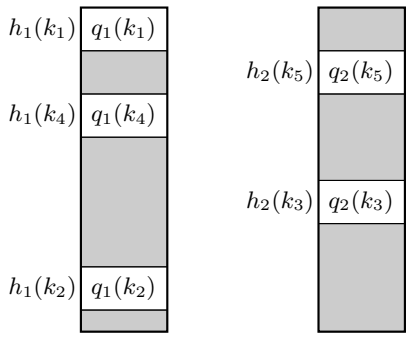


**Figure 1:** Example of our data structure.

idea of using quotient functions is that storing $q(k)$ may require fewer bits than storing $k$ itself. If a fraction $O(1/r)$ of all possible keys hashes to each of $r$ hash table cells, there is a quotient function whose function values can be stored in $w - \log r + O(1)$ bits. This approach was used in [13] to construct a dictionary using space close to the information theoretical minimum.

**Example** We consider the hash function family from [6] mapping from $\{0,1\}^w$ to $\{0,1\}^t$, i.e., with $r = 2^t$. It contains functions of the form $h_a(k) = (ak \mod 2^w) \operatorname{div} 2^{w-t}$ for $a$ odd and $0 < a < 2^w$. A corresponding family of quotient functions is given by $q_a(k) = (ak \mod 2^w) \bmod 2^{w-t}$, whose function values can be stored in $w - \log r$ bits.

## 2.2 Our Data Structure

The idea behind our lossy dictionary, compared to the static dictionary of [14] described above, is to try to fill the hash tables almost completely, working with key sets of size similar to or larger than $r$. Each key has two hash table cells to which it can be matched.

Thus, given a pair of hash functions, the problem of finding a maximum weight subset of $S$ that can be arranged into the hash tables is a maximum weight matching problem that can be solved in polynomial time, see e.g. [5]. In Section 2.3 we will present an algorithm that finds such an optimal solution in time $O(n \log^* n)$. The term $O(rl/w)$ in the time bound of Theorem 1 is the time needed to copy associated information to the tables. Assume for now that we know which keys are to be represented in which hash table cells.

For $b = 0$ we simply store quotient function values in nonempty hash table cells and the value $0^s$ in empty hash table cells, using $s$ bits per cell, as shown in Figure 1. For general, $b$ we store only the first $s - b$ bits. Observe that no more than $2^b$ keys with the same hash function value can share the first $s - b$ bits of the quotient function value. This means that there are at most $2^b - 1$ false positives for each nonempty cell. Since $0^s$ is not in the range, this is also true for empty cells. In addition to the $s - b$ bits, we use $l$ bits per cell to store associated information.

We now proceed to fill in the remaining details on items 3 and 5 of Theorem 1.

### 2.3 Construction Algorithm

Recall that the task of constructing our data structure boils down to finding the largest weight arrangement of keys in the tables. Given hash functions $h_1$ and $h_2$ and a key set $K$, we define the bipartite graph $G(K)$ with vertex set $\{1,2\} \times \{1,\ldots,r/2\}$, corresponding in a natural way to hash table cells, and the multiset of edges $\{\{(1,h_1(k)),(2,h_2(k))\} \mid k \in K\}$, corresponding to keys. Note that there may be parallel edges if several keys have the same pair of hash function values. We will use the terms keys/edges and cells/vertices synonymously. A connected component of $G(K)$ is defined to be *saturated* if the number of edges is greater than or equal to the number of vertices, i.e., if it is not a tree. We have the following characterization of the key sets that can be placed in the tables according to the given hash functions.

**Lemma 1** *The key set $K$ can be placed in the tables if and only if each connected component of $G(K)$ is a tree, plus possibly an extra edge.*

*Proof.* By Hall's theorem, $K$ can be placed in the tables if and only if every subset $K' \subseteq K$ satisfies $|h_1(K')| + |h_2(K')| \geq |K'|$. This is true if and only if every subset $K'$ of edges in $G(K)$ covers at least $|K'|$ vertices. Since it is equivalent to quantify only over subsets of edges within a connected component, the lemma follows. □

By an *optimal solution* for a key set $K$ we will understand a maximum weight subset of $K$ that can be placed in the tables.

**Lemma 2** *There is an optimal solution for $\{k_1,\ldots,k_i\}$ including key $k_i$ if and only if for any optimal solution $K'$ for $\{k_1,\ldots,k_{i-1}\}$, the set $K' \cup \{k_i\}$ can be placed in the tables.*

*Proof.* If $K' \cup \{k_i\}$ can be placed in the tables for some solution $K'$ optimal for $\{k_1,\ldots,k_{i-1}\}$, then $K' \cup \{k_i\}$ must be optimal for $\{k_1,\ldots,k_i\}$.

On the other hand, suppose that for some $K \subseteq \{k_1,\ldots,k_{i-1}\}$, the key set $K \cup \{k_i\}$ can be placed in the tables and has optimal weight, and let $K'$ be an optimal solution for $\{k_1,\ldots,k_{i-1}\}$. Consider the connected components of $(1,h_1(k_i))$ and $(2,h_2(k_i))$ in $G(K)$. By Lemma 1 and since $K \cup \{k_i\}$ can be placed in the tables, at least one of the (possibly identical) connected components must be a tree, without loss of generality the component of $(1,h_1(k_i))$. Since $K \cup \{k_i\}$ is optimal, the connected component of $(1,h_1(k_i))$ in $G(\{k_1,\ldots,k_{i-1}\})$ must also be a tree. (If there was a cycle, a key of higher weight could be substituted for $k_i$, contradicting the optimality of $K \cup \{k_i\}$.) In particular, the connected component of $(1,h_1(k_i))$ in $G(K')$ is a tree. Thus, by Lemma 1 the set $K' \cup \{k_i\}$ can be placed in the tables. □

The lemma implies that the following greedy algorithm finds an optimal key set $S'$ given keys sorted according to nonincreasing weight.

1. Initialize a union-find data structure for the cells of the hash tables.

2. For each equivalence class, set a "saturated" flag to `false`.

3. For $i = 1, \ldots, n$:

   (a) Find the equivalence classes $c_1$ of cell $h_1(k_i)$ in $T_1$, and $c_2$ of cell $h_2(k_i)$ in $T_2$.

   (b) If $c_1$ or $c_2$ is not saturated:

      i. Include $k_i$ in the solution.

      ii. Join $c_1$ and $c_2$ to form an equivalence class $c$.

      iii. Set the saturated flag of $c$ if $c_1 = c_2$, or if the saturated flag is set for $c_1$ or $c_2$.

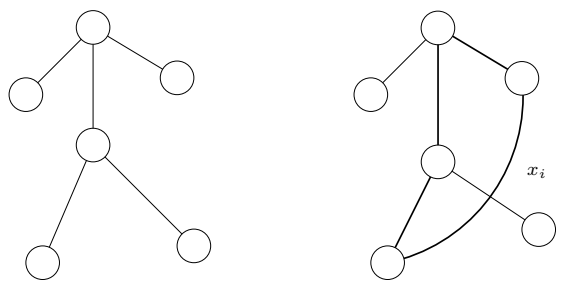


**Figure 2:** The case where $c_1 = c_2$ and the component is nonsaturated. The component becomes saturated.

**Figure 3:** The case with one saturated and one nonsaturated component. The new component becomes saturated.

**Figure 4:** The case with two saturated components. The new element is not included.

In the loop, equivalence classes correspond to the connected components of the graph $G(\{k_1, \ldots, k_{i-1}\})$. There is a simple implementation of a union-find data structure for which operations take $O(\log^* n)$ amortized time; see [19] which actually gives an even better time bound. Figures 2 to 4 show three possible cases in step 3b of the algorithm.

What remains is arranging the optimal key set $S'$ in the tables. Consider a vertex in $G(S')$ of degree one. It is clear that there must be an arrangement such that the corresponding cell contains the key of the incident edge. Thus, one can

iteratively handle edges incident to vertices of degree one and (conceptually) delete them. As we remove the same number of edges and vertices from each connected component, the remaining graph consists of connected components with no more edges than vertices and no vertices of degree one, i.e., cycles. The arrangement of edges in a cycle follows as soon as one key has been put (arbitrarily) into one of the tables. The above steps are easily implemented to run in linear time. This establishes item 5 of Theorem 1.

### 2.4 Quality of Solution

We now turn to the problem of estimating the quality of the solution. Note that the optimal key set returned by our algorithm does not depend on the actual weights, but only on the sequence of hash function values. Thus, the expected weight of our optimal solution is $\sum_{i=1}^{n} p_{r,i}\, v_i$, where $p_{r,i}$ is the probability that the $i$th key is included in the returned optimal set of keys, which is *independent* of the weights.

Our algorithm includes all keys $\{k_1, \ldots, k_i\}$ in the optimal solution returned if they can all be accommodated under the given hash functions. Using the result of [14] mentioned in Section 2.1 on $\{k_1, \ldots, k_i\}$ with $\delta = r/i - 2$, we have that for $i < r/2$ this happens with probability at least $1 - 52\, r^{-1}/(r/i - 2)$. In particular, $p_{r,i}$ is at least this big.

For $i \geq r/2$ we derive a lower bound on $p_{r,i}$ as follows. If one of the vertices $(1, h_1(k_i))$ and $(2, h_2(k_i))$ in $G(\{k_1, \ldots, k_{i-1}\})$ is isolated, then $k_i$ is in the optimal solution returned. The randomness assumption on our hash functions implies that $G(\{k_1, \ldots, k_{i-1}\})$ has $i-1$ randomly and independently chosen edges. Thus, we have the bound $p_{r,i} \geq 1 - (1 - (1 - 2/r)^{i-1}))^2 = 2(1 - 2/r)^{i-1} - (1 - 2/r)^{2(i-1)} \approx 2e^{-i/r} - e^{-2i/r}$. This establishes item 3 of Theorem 1 and concludes the proof.

### 2.5 A Lower Bound

This section gives a lower bound on the amount of memory needed by a lossy dictionary with an $\epsilon$ fraction of false positives and $\gamma n$ false negatives. Our proof technique is similar to that used for the lower bound in [4] for the case $\gamma = 0$.

**Proposition 1** *For $0 < \epsilon < 1/2$ and $0 < \gamma < 1$, a lossy dictionary representing a set $S \subseteq \{0,1\}^w$ of $n$ keys, $120 < n \leq 2^{w-1}$, with at most $2^w\epsilon$ false positives and at most $\gamma n$ false negatives must use space of at least*

$$(1-\gamma)\, n \log\left(\frac{1}{\epsilon + n/2^w}\right) - \tfrac{5}{2} n \text{ bits.}$$

*Proof.* We can assume without loss of generality that $\gamma n$ is integer (this only gives a stronger space lower bound), and that $2^w\epsilon$ is integer. Consider the set of all data structures used for the various subsets of $n$ elements from $\{0,1\}^w$. Any of these data structures must represent a set of at most $2^w\epsilon + n$ keys, in order to meet the requirement on the number of false positives. Thus, the number of $n$-element sets having up to $\gamma n$ keys outside the set represented by a given data

structure is at most $\sum_{i=0}^{\gamma n} \binom{2^w\epsilon+n}{n-i}\binom{2^w}{i}$. Since $\epsilon < 1/2$ and $n \le 2^{w-1}$ we have $2^w\epsilon + n \le 2^w$, and so the largest term in the summation is $\binom{2^w\epsilon+n}{n-\gamma n}\binom{2^w}{\gamma n}$. Thus we have the upper bound $n\binom{2^w\epsilon+n}{n-\gamma n}\binom{2^w}{\gamma n}$.

We will use the inequalities $(\frac{a}{b})^b \le \binom{a}{b} < (\frac{ae}{b})^b$, see e.g. [10, Proposition 1.3]. By the upper bound on the number of sets representable by each data structure, we need, in order to represent all $\binom{2^w}{n}$ key sets, space at least

$$\begin{aligned}
&\log\binom{2^w}{n} - \log\left(n\binom{2^w\epsilon+n}{(1-\gamma)n}\binom{2^w}{\gamma n}\right) \\
&\qquad \ge \log\left(\frac{2^w}{n}\right)^n - \log\left(n\left(\frac{(2^w\epsilon+n)e}{(1-\gamma)n}\right)^{(1-\gamma)n}\left(\frac{2^we}{\gamma n}\right)^{\gamma n}\right) \\
&\qquad = n\log\left(\frac{2^w}{n}\frac{(1-\gamma)n}{(2^w\epsilon+n)e}\right) - \gamma n\log\left(\frac{(1-\gamma)n}{(2^w\epsilon+n)e}\frac{2^we}{\gamma n}\right) - \log n \\
&\qquad = n\log\left(\frac{(1-\gamma)/e}{\epsilon+n/2^w}\right) - \gamma n\log\left(\frac{1-\gamma}{\gamma(\epsilon+n/2^w)}\right) - \log n \\
&\qquad = (1-\gamma)\,n\log\left(\frac{1}{\epsilon+n/2^w}\right) - (H(\gamma)+\log e)\,n - \log n
\end{aligned}$$

where $H(\gamma) = -\gamma\log\gamma - (1-\gamma)\log(1-\gamma) \le 1$ is the binary entropy function. For $n > 120$ the sum of the last two terms is bigger than $\frac{5}{2}n$. □

In the discussion following Theorem 1 we noted that if there are quotient functions with optimal range, the space usage of our scheme is $n\log(\frac{1}{\epsilon+n/2^w}) + O(n)$ when tables of combined size $n$ are used. The expected fraction $\gamma$ of false negatives is less than 3/10 by Theorem 1. This means that our data structure uses within $O(n)$ bits of 10/7 times the lower bound. The experiments described in Section 3 indicate that the true factor is less than 6/5.

### 2.6 Using More Tables

We now briefly look at a generalization of the two-table scheme to schemes with more tables. Unfortunately the algorithm described in Section 2.3 does not seem to generalize to more than two tables. An optimal solution can again be found using maximum weight matching, but the time complexity of this solution is not attractive. Instead we can use a variant of the *cuckoo* scheme described by the authors in [15], greedily attempting to accommodate keys in order $k_1, \ldots, k_n$.

For two tables an insertion attempt for $k_i$ works as follows: We store $k_i$ in cell $h_1(k_i)$ of $T_1$, pushing the previous occupant, if any, away and thus making it *nestless*. If cell $h_1(k_i)$ was free we are done. Otherwise we insert the nestless element in $T_2$, possibly pushing out another element. This continues until we either find a free cell or loop around unable to find a free cell, in which case $k_i$ is discarded. It follows from [15] and the analysis in Section 2.3 that this algorithm finds an optimal solution, though, not as efficiently as the algorithm given in Section 2.2. When using three or more tables it is not obvious in which of the tables one should attempt placing the "nestless" key. One heuristic that works well is to simply pick one of the two possible tables at random. It is

interesting to compare this heuristic to a random walk on an expander graph, which will provably cross any large subset of the vertices with high probability.

The main drawback of using three tables is, of course, that another memory probe is needed for lookups. Furthermore, as the range of the hash functions must be smaller than when using two tables, the smallest possible range of quotient functions is larger, so more space may be needed for each cell.

# 3 Experiments

An important performance parameter of our lossy dictionaries is the ability to store many keys with high weight. We tested this ability for lossy dictionaries using two and three tables. For comparison, we also tested the simple one-table scheme that stores in each cell the key of greatest weight hashing to it. The tests were done using truly random hash function values, obtained from a high quality collection of random bits freely available on the Internet [12]. Figure 5 shows experimentally determined values of $p_{r,\alpha r}$, the probability that the key with index $i = \alpha r$ is stored in the dictionary, determined from $10^4$ trials. For the experiments with one and two tables we used table size $r = 2048$ while for the experiment with three tables we used $r = 1536$. We also tried various other table sizes, but the graphs were almost indistinguishable from the ones shown. From Figure 5 we see the significant improvement of moving from one to more tables. As predicted, nearly all of the $r/2$ heaviest keys are stored when using two tables. For three tables this number increases to about $.88r$. Of the $r$ heaviest keys, about 84% are stored when using two tables, and 95% are stored when using three tables.

Apart from asymptotically vanishing differences around the point where the curves start falling from 1, the graphs of Figure 5 seem independent of $r$. For two tables the observed value of $p_{r,\alpha r}$ for $\alpha > 1/2$ is approximately $3.5/9.6^{\alpha}$ and for three tables it is approximately $8/33^{\alpha}$ for $\alpha > 0.95$.

The gap to the two-table lower bound of Theorem 1 can be explained by the fact that this lower bound considers only two cells of the hash tables, whereas opportunities for storing keys may appear when considering more cells.

# 4 Conclusion

We have introduced the concept of lossy dictionaries and presented a simple and efficient data structure implementing a lossy dictionary. Our data structure combines very efficient lookups and near-optimal space utilization, and thus seems a promising alternative to previously known data structures when a small percentage of false negatives is tolerable, such as the examples in Section 1.2.

Though simple and efficient hash functions seem to work well in practice with our data structure, the challenge of finding such families that provably work well remains. Furthermore, the last two graphs in Figure 5 are not completely understood. The same is true for the insertion heuristic for three or more tables.

**Acknowledgment** We thank Stephen Alstrup and Theis Rauhe for helpful discussions on the construction of our two table data structure.

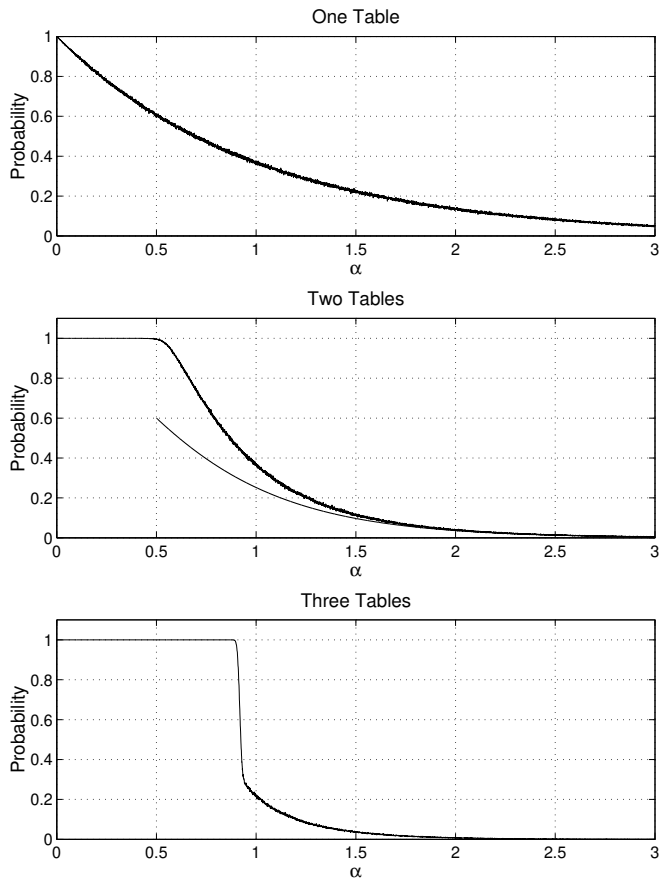


**Figure 5:** The observed probability that the element with $(\alpha r)$th highest weight is stored when using one, two and three tables. For two tables our lower bound is shown.

# References


[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640 (electronic), 1999.

[3] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, New York, 2000.

[4] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, New York, 1978.

[5] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial optimization.* John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.

[6] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[8] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[9] Insung Ihm and Sanghun Park. Wavelet-based 3D compression scheme for very large volume data. *Graphics Interface*, pages 107–116, 1998.

[10] Stasys Jukna. *Extremal Combinatorics with Applications in Computer Science.* Springer-Verlag, 2000.

[11] Tae-Young Kim and Yeong Gil Shin. An efficient wavelet-based compression method for volume rendering. In *Seventh Pacific Conference on Computer Graphics and Applications*, pages 147–156, 1999.

[12] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[13] Rasmus Pagh. Low Redundancy in Static Dictionaries with $O(1)$ Lookup Time. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604. Springer-Verlag, Berlin, 1999.

[14] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01).* ACM Press, New York, 2001.

[15] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. To appear in Proceedings of ESA 2001, 2001.

[16] Flemming Friche Rodler. Wavelet based 3D compression with fast random access for very large volume data. In *Seventh Pacific Conference on Computer Graphics and Applications*, pages 108–117, Seoul, Korea, 1999.

[17] Flemming Friche Rodler and Rasmus Pagh. Fast random access to wavelet compressed volumetric data using hashing. Manuscript.

[18] Dennis S. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large data bases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

[19] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

# Recent BRICS Report Series Publications

**RS-01-33** Rasmus Pagh and Flemming Friche Rodler. *Lossy Dictionaries*. August 2001. 14 pp. Short version appears in Meyer auf der Heide, editor, *9th Annual European Symposiumon on Algorithms*, ESA '01 Proceedings, LNCS 2161, 2001, pages 300–311.

**RS-01-32** Rasmus Pagh and Flemming Friche Rodler. *Cuckoo Hashing*. August 2001. 21 pp. Short version appears in Meyer auf der Heide, editor, *9th Annual European Symposiumon on Algorithms*, ESA '01 Proceedings, LNCS 2161, 2001, pages 121–133.

**RS-01-31** Olivier Danvy and Lasse R. Nielsen. *Syntactic Theories in Practice*. July 2001. 37 pp. Extended version of an article to appear in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001 (Firenze, Italy, September 4, 2001).

**RS-01-30** Lasse R. Nielsen. *A Selective CPS Transformation*. July 2001. 24 pp. To appear in Brookes and Mislove, editors, *27th Annual Conference on the Mathematical Foundations of Programming Semantics*, MFPS '01 Proceedings, ENTCS 45, 2000. A preliminary version appeared in Brookes and Mislove, editors, *Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01,* (Aarhus, Denmark, May 24–27*, 2001)*, BRICS Notes Series NS-01-2, 2001, pages 201–222.

**RS-01-29** Olivier Danvy, Bernd Grobauer, and Morten Rhiger. *A Unifying Approach to Goal-Directed Evaluation*. July 2001. 23 pp. To appear in *New Generation Computing*, 20(1), November 2001. A preliminary version appeared in Taha, editor, *2nd International Workshop on Semantics, Applications, and Implementation of Program Generation*, SAIG '01 Proceedings, LNCS 2196, 2001, pages 108–125.

**RS-01-28** Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. *A Fully Equational Proof of Parikh's Theorem*. June 2001.