UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Perfect Hashing for External Files

M.V. Ramakrishna

CS-86-25

1986

Perfect Hashing for External Files

M.V. Ramakrishna

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

ABSTRACT

Perfect hashing refers to hashing with no overflows. The use of perfect hashing functions has previously been studied only for small static sets stored in main memory. All known systematic methods of finding perfect hashing functions are conceptually complicated, and most methods need exponential time to determine a perfect hashing function for a given set of keys.

In this thesis we propose and analyze a perfect hashing scheme for large external files. The scheme guarantees retrieval of any record in a single disk access. Insertions and deletions are simple, and the file size may vary considerably without adversely affecting the performance. A variant of the basic scheme gives a completely dynamic file organization which also supports efficient range searching. These advantages are achieved at the cost of a small amount of supplemental internal storage and increased cost of insertions.

An ordinary hashing function is used to divide the records of the file into a number of groups. The records in each group are then hashed over a number of contiguous pages of external memory by a perfect hashing function. A perfect hashing function for a group can be found by repeated random selection from a suitable class of hashing functions. We analyze the probability of a randomly chosen function (from the set of all functions) being perfect. We then describe a policy that limits the cost of finding perfect hashing functions. The resulting tradeoff between the storage utilization and the cost of finding perfect hashing functions is investigated. Results of experiments with a simple and practical class of hashing functions are reported. They indicate that the relative frequency of perfect hashing functions within the class is statistically the same as predicted by the theoretical analysis for the set of all functions. The performance of the new scheme is also compared with other hashing schemes. We conclude that the proposed perfect hashing scheme is a practical and competitive technique for organizing external files.

It's not easy being perfect ...
But somebody has to do it!

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Per-Åke Larson, for his valuable guidance, encouragement and friendship during the course of this research. Research under his supervision was more fun than work. Special thanks are also due to Gordon Cormack for many fruitful discussions I had with him, especially at the early stages of the research. The other members of my thesis committee, Walter Burkhard, David Matthews and Derick Wood provided helpful comments.

I would like to thank Andre Trudel for his careful reading of the thesis and for his comments to improve its readability. Also Derek Kirkland and Scott Barthelmy for providing the test files for experiments, and Yvonne Fink for typesetting most of the thesis.

My deepest gratitude is to my wife Shobha, for her love and support. We wish to thank our families in India for their constant support, our son Vivek and many friends in Waterloo who were responsible for our pleasant stay here.

Finally, I wish to gratefully acknowledge the financial support from the Canadian Commonwealth Scholarship and Fellowship Authority, and the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

Chapter 1

Introduction

Hashing is an efficient and popular technique for organizing internal tables and external files. The fundamental idea behind hashing is to perform some transformation on the key associated with a record to produce an address within the range of the table where the record is to be stored. If the address is already full, we say that a collision has occurred. If each address is capable of holding more than one record, as typically is the case with external files, *overflow* is a more appropriate word for *collision*. A key concern of hashing is the presence of overflow records. Various schemes have been proposed to handle the overflow problem. This thesis takes a different approach by trying to eliminate overflow records altogether.

A hashing function is said to be perfect (for a given set of keys) if it causes no overflow records. In this thesis, we propose and analyze a perfect hashing scheme for large external files and show that the scheme is indeed practical. A key feature of the scheme is that any record can be retrieved in a single disk access.

A significant amount of the research on hashing is dedicated to the problem of handling overflows. The basic approach has been to accept the inevitability of overflow records and try to handle them as efficiently as possible. The quotations below illustrate this point:

> "In order to use a hash table, a programmer **must** make two almost independent decisions: He **must** choose a hash function $h(k)$, and he **must** select a method for collision resolution". [KN74]
>
> "One of the key **concerns** when designing hashing schemes and hash files is the choice of a method for handling overflow records". [LR80]
>
> "When hashing is used for organizing files and tables some method for solving the problem of overflow records **must** be devised". [LR79]
>
> "Handling collisions is the **central issue** in hashing and the subject of this thesis". [CP86]

Knuth explains the reasons for this traditional approach with the following example[KN74]. Consider storing the 31 most common English words in a hash table having 41 locations. There are a total of $\approx 10^{50}$ possible functions and only $\approx 10^{43}$ of them will give distinct addresses for the 31 words, i.e. no overflows; thus only one in 10 million functions will be perfect.

Perfect hashing functions are rare, even when the table is fairly large compared to the number of keys. The famous "birthday paradox" asserts that if we select a random function which maps 23 keys into a table of size 365, the probability of the function being perfect is less than one-half (0.4927).

For external files stored on disk, typically, each addressable unit of memory is capable of holding more than one record. Each addressable unit is called a page or bucket. (In the above examples, each address is capable of holding at

most one record; the page size is 1.) It is easier to find a hashing function causing no overflows when keys are hashed into larger pages. This fact is exploited in this thesis to obtain a practical perfect hashing scheme for external files. The proposed scheme is an extension of a scheme proposed by Cormack, Horspool and Kaiserswerth [CH85] to external files .

The research reported in this thesis is the first attempt at applying perfect hashing to external files. Some early results were reported in [LR85]. We wish to emphasize that the main goal of the research was to determine whether the idea of using perfect hashing to organize large external files is *practical*. The next section outlines what we consider *practical*, given current technology.

1.1. Definitions and assumptions

Consider a hash table (or hash file) consisting of m buckets (pages), each with a capacity of b keys (records). A set $I = \{x_1, x_2, \ldots, x_n\}$, of n keys, $n \leq mb$, are to be stored in the table. A hashing function h, $h: I \rightarrow [0, m-1]$, assigns each key an address in the range $0, 1, \ldots, m-1$. The hashing function h is said to be a **perfect hashing function** if no address receives more than b records. It is a **minimal perfect hashing** function if the table is of minimal size, $m = \lceil n/b \rceil$.

We make the assumption that the keys are integers. An implicit assumption in all research on hashing is that the hashing function is simple, both in terms of the space required to store the parameters of the hashing function and the cost of computing the hash address for a key. Using the RAM model of analysis[AH74], let a key occupy a unit of memory and the cost of a simple arithmetic operation involving integers of unit length be unity. Under this model, it is reasonable to require that a hashing function use O(1) storage for its parameters and the cost of computing a hash address is O(1).

The term *practical* range is frequently used in this thesis. It is difficult to give an absolute definition of *practical*. The following outlines what we consider practical with respect to various parameters.

1. Permanent internal memory: A few bytes per hundred records stored in the file. For example, 5 to 50 kilobytes for a file of one million records.
2. Temporary space: The temporary use of a fairly large amount of internal storage does not cause any serious problems in modern computer systems. We consider temporary buffer space (used during an insert or delete operation) for a few hundred records reasonable.
3. Page size: Given current disk technology, using pages capable of storing 10 to 50 records seems reasonable.
4. Hash address computation: Computing a hash address should require a small, constant number of operations, independent of the file size. However, the number of operations may depend on the key length.
5. Insertion and deletion costs: A few hundred hash address computations, on the average, is considered reasonable. The minimum disk I/O cost is 2.0 accesses for any file organization. An average of up to 2-3 times the minimum I/O cost seems acceptable.

6. File size: We are mainly interested in files having a few thousand to a few million records.

Although the techniques and the analysis presented are general and applicable to any parameter range, we concentrate on the above ranges when presenting numerical results.

We use the terms *disk* and *secondary storage* synonymously. It is implicitly assumed that accessing secondary storage is much slower than accessing internal memory. This is true in practice for all mass storage devices.

1.2. Background

Greniewski and Turski were the first to describe a method for constructing perfect hashing functions [GT63]. Their method is not general, and the resulting hashing functions are complicated. Systematic methods for finding perfect hashing functions were first reported by Sprugnoli in 1977 [SP77]. Since then, several other researchers have reported new methods for finding perfect hashing functions. The various methods can be grouped into two classes:

a) Direct perfect hashing

b) Composite perfect hashing (which need some form of a lookup table)

All the methods for finding direct perfect hashing functions found in the literature deal only with small static sets of 10 to 15 elements. Those which can handle larger sets are all composite.

1.2.1. Direct perfect hashing

Sprugnoli dealt only with small static sets (10-15 elements, no insertions or deletions allowed)[SP77]. He proved the existence of a perfect hashing function of the form $h(x) = \lfloor (x+s)/N \rfloor$, where s and N are constants, for any given set of keys $I = \{x_1, x_2, \cdots, x_n\}$. He described an algorithm, called the **Quotient Reduction Method**, for finding s and N. The algorithm involves sorting, and finding the $(n-1)$st differences of the elements of the set I, $n = |I|$. The complexity of the algorithm is at least $O(n^3)$ with a large constant. The space utilization may be poor, especially if the keys are not uniformly distributed. This is illustrated by a simple example: given the key set $I = \{0, 9, 10, 11\}$, the best perfect hashing function is $h(x) = \lfloor (x + 0)/1 \rfloor$, yielding a storage utilization of only 33%.

When the keys have a nonuniform distribution, Sprugnoli reports that hashing functions of the following form yield better results:

$$h(x) = \lfloor ((xq+d) \bmod m) / N \rfloor .$$

He described a method, called the **Remainder Reduction Method**, for finding the constants q, d, m and N so that h is a perfect hashing function for the given set of keys. However, he was unable to prove that the method always succeeds in finding a perfect hashing function.

Jaeschke [JS81] proposed **Reciprocal Hashing**, which he claimed to be superior to Sprugnoli's methods. The hashing functions are of the form:

$$h(x) = \lfloor C / (Dx + E) \rfloor \ mod\ n,$$

where C, D and E are constants. D and E are chosen so that for all pairs $x_i, x_j \in I, i \neq j, (Dx_i + E)$ and $(Dx_j + E)$ are relatively prime. Jaeschke gave constructive proofs for the existence of C, D and E for any given set of keys. However, the values of the constants obtained are too large to be useful in practice. The choice $D = 1, E = 0$ was reported to be satisfactory in most cases; C can be calculated by a trial and error search. The theoretical upper bound for the complexity of the search procedure is x_{max}^{n+1} where x_{max} is such that $\forall x_i \in I, x_i \leq x_{max}$. However, Jaeschke reports that experimentally the average complexity is 1.82^n for $n \leq 15$ and $I \subset \{1, 2, 3, \ldots, 1000000\}$. For $n \leq 15$ the time complexity is roughly the same as that of Sprugnoli's methods, but Reciprocal Hashing yields minimum perfect hashing functions. When $n > 15$, neither method is practical because of the prohibitively high cost of finding perfect hashing functions.

Chang [CH84] proposed a perfect hashing scheme based on the Chinese Remainder Theorem. The space required to store the hashing function is $O(n)$ and given a key the cost of evaluating its hash address is $O(n)$. Moreover, his method works only for small key values (less than 100) and hence we will not go into further details.

1.2.2. Composite perfect hashing

It is impractical to find direct perfect hashing functions for sets larger than 10 to 15 elements. Sprugnoli suggested the use of segmentation to handle larger sets. Use a function h_0, to divide the given set into a number of segments: $S_i = \{ x \mid h_0(x) = i \text{ and } x \in I \}$. h_0 can be an ordinary hashing function. If each segment has no more than 10-15 elements, direct perfect hashing can be used to store the segments separately. The parameters of the perfect hashing functions must be stored in a table and hence retrieval involves two levels of access: one to obtain the parameters of the perfect hashing function for a segment and the other to get to the key. This idea is the basis for most of the recent research on perfect hashing, including the research reported in this thesis. We call such a two-level hashing function, a composite perfect hashing function.

Fredman, Komlos and Szemeredi [FK82] proposed a scheme which implements Sprugnoli's idea of segmentation. The hashing functions are of the form $h(x) = (kx\ mod\ p)\ mod\ m$, where p is a prime number such that $\forall x \in I, x < p$ and k is a constant, $k < p$. For any given set J, $r = |J|$, they show that there always exists a perfect hashing function of the form $h(x) = (kx\ mod\ p)\ mod\ m$, where $m = r^2$.

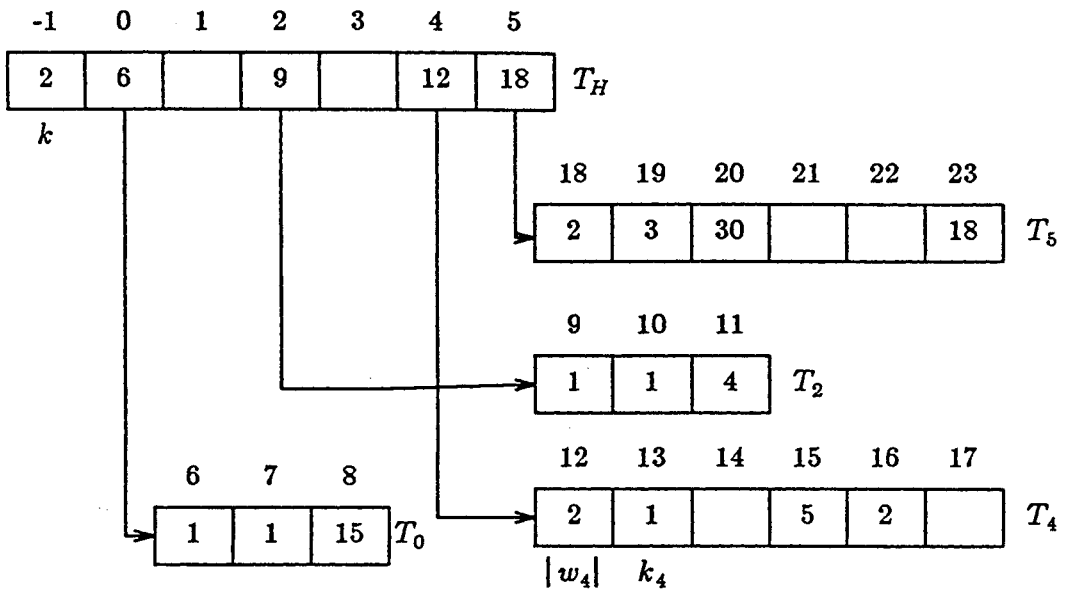


Figure 1.2.1 Illustration of the hash table construction

Figure 1.2.1 illustrates their construction for $I = \{2,4,5,15,18,30\}$, $n = 6$, $p = 31$. $T_H[-1 .. n-1]$ is an array of length $(n+1)$ which stores the first-level structure. $T_H[-1]$ contains the value of k used for the segmentation function H: in this case $H(x) = ((2x \ mod \ 31) \ mod \ 6)$. All the keys hashing to location i under H form the set w_i. The entry $T_H[i]$ points to an array $T_i[-2 .. \lfloor w_i \rfloor^2 - 1]$ of length $\lfloor w_i \rfloor^2 + 2$ in which the set w_i is stored. $h_i(x) = (k_i \ x \ mod \ p) \ mod \ \lfloor w_i \rfloor^2$ is a perfect hashing function for the set w_i. k_i is found by trial-and-error. $\lfloor w_i \rfloor$ and k_i are stored in $T_i[-2]$ and $T_i[-1]$, respectively. The elements of w_i are stored in T_i at the addresses given by h_i. This scheme requires five accesses for retrieval and uses $O(n)$ storage. The table can be constructed in random expected time $O(n)$.

Du, Hsieh, Jea and Shieh [DH83] proposed another composite perfect hashing scheme. Their first-level structure is called a Hash Indicator Table (HIT). A perfect hashing function is constructed as the composition of a set of hashing functions $H = \{h_1, h_2, \ldots, h_s\}$. The algorithms for constructing the table given the set I and for retrieving a given key x are given below.

```
HIT, T : array 1 .. m of integer;

procedure store (I);
    I : set of integer;
begin
for j := 1 to s do
    for all elements x in I do
        d:= h_j(x);
        if HIT [d] = 0 for one and only one x ∈ I then
            begin HIT [d] := j; T[d] := x; I := I − {x}; end
        else no perfect hashing function can be found ;
    endloop;
endloop;
end;

function Retrieve (x): integer;
    x: integer;
begin
for i := 1 to s do
    d := h_i(x) ;
    if (HIT[d] = i) then return (T[d]);
endloop;
end;
```

This scheme needs up to s address calculations and accesses to the HIT to retrieve a key. If both the HIT and the data array are in the same storage medium, this can hardly be called a perfect hashing scheme. One may as well use any other scheme, such as double hashing, which needs s or less accesses for each retrieval.

Yang and Du [YD84] extended this scheme to handle dynamic key sets stored in external memory. Their scheme resembles Larson's dynamic hashing [LR78]. They report that about 1.6 kilobytes of internal memory is required to store 1,000 records. The internal space requirement is much higher than other schemes which guarantee single access retrieval [GL82, LR78].

Retaining the same basic structure, Cormack, Horspool and Kaiserswerth [CH85] extended the scheme of Fredman et. al. [FK82] to handle insertions and deletions. They called the resulting scheme "practical perfect hashing". The external perfect hashing scheme presented in chapter 2 of this thesis is an extension of their "practical perfect hashing" to external files. We will not go into further details of the scheme here.

Cichelli[CC80] described a simple method of computing perfect hashing functions. The first-level structure he used is basically a mapping function as described below. In his scheme keys are treated as strings over an alphabet, rather than as integers. The hashing functions used are of the form:

$$h(\text{key}) = \text{length(key)} + g(\text{first letter of key}) + g(\text{last letter of key})$$

$$g : \{\text{set of first and last letters of keys}\} \rightarrow \{\text{set of natural numbers}\} .$$

The function g is stored in a table and constructed so as to make h a perfect hashing function for the given set of keys. Cichelli described a backtracking method based on trial-and-error to construct the function g. In the worst case, this search procedure takes exponential time. It worked well for a few examples given in [CC80]. There are infinitely many trivial cases for which this method fails, for example, when the key set contains both the keys FOR and FAR. Others have worked on this scheme, improving its performance to some extent by taking into account additional letters of the key [CK83, CB85, SG85]. Depending on the size of the table for the function g, sets of up to a few hundred keys can be handled.

1.2.3. Single access retrieval from external files

The dynamic hashing scheme proposed by Larson is one of the earliest hashing schemes which uses an index and avoids overflows [LR78]; it may be regarded as a perfect hashing scheme for external files if the full index is stored in internal memory. Similarly, extendible hashing [FN79] guarantees single access retrieval, if the directory structure is stored in internal memory. We have already mentioned another perfect hashing scheme for external files using a Hash Indicator Table [YD84]. All these methods use too much internal memory to be of any interest in practice.

An external hashing scheme developed by Gonnet and Larson [GL82] achieves the goal of retrieval in a single disk access. However, it is not a perfect hashing scheme; it does allow overflows. There is an index table stored internally, with one entry per (external) bucket. Each entry contains a separator of length k bits: for a full bucket, the maximum of the hash *signatures*† of the keys stored in the bucket; for non-full buckets, the value 2^k-1. The number of bits per entry determines the maximum load factor possible for a given bucket size. Normal double hashing is used to store the keys. To retrieve a record, the primary hash address and the signature of the key are computed. If the computed signature is less than or equal to the separator of the bucket, the required bucket has been found. Otherwise, proceed by calculating the next address and signature of the key until a bucket whose separator is greater than the signature of the key is found. Insertion into non-full buckets is straightforward. When an insertion causes a bucket to overflow, the keys are partitioned according to their signatures. A separator s is determined so that the number of keys having signatures less than or equal to s is as close to the bucket size as possible: those records are retained in the bucket. Keys whose signatures are strictly greater than s are moved to their subsequent secondary addresses. In essence, external probing has been replaced by internal probing and the separator table contains enough information to terminate the probing at the correct address. The results reported for this method are quite impressive: with just 2 bits of internal memory per bucket, about 84% storage utilization can be attained with a bucket size of

---

† A hash *signature* of length k, is the first k bits produced by a secondary hashing function operating on the key.

20 records. Larson has extended this scheme to handle dynamic files [LR84].

1.3. Thesis outline

In this chapter we have discussed several perfect hashing schemes. All the methods for finding perfect hashing functions involve some form of trial-and-error searching. Unless the set is very small (10-15 elements), some form of intermediate table is essential. All the methods are intended for internally stored tables, and none consider perfect hashing for external files where each address is capable of storing more than one record.

In chapter 2 the proposed perfect hashing scheme for large external files is described. It is based on Sprugnoli's idea of segmentation: divide the file into smaller groups and use perfect hashing to store each group separately. The basic scheme uses hashing to achieve segmentation; an extension using a B-tree to achieve segmentation is also described. There are several questions related to the practicality of the proposed scheme. The rest of the thesis is devoted to answering these questions.

In chapter 3 we propose a method for finding perfect hashing functions and analyze its performance. The method simply consists of making trials by choosing functions at random from the set of all possible functions. The main contribution of the chapter is an analysis of the cost of the trial-and-error approach of finding perfect hashing functions when each bucket can hold more than one record. We give a simple recurrence relation for computing the probability of a randomly chosen function being perfect. Previously known methods of computing these probabilities are several orders of magnitude slower.

The results presented in chapter 3 show that there is a sharp trade-off between the cost of finding perfect hashing functions and the required load factor. In chapter 4 we consider policies for balancing the trade-offs. We define an optimal policy which attempts to limit the cost of finding perfect hashing functions. Formulation and solution of a nonlinear integer optimization problem to obtain the optimal policy is then discussed.

In chapter 5 we study the effect of the optimal policy on the load factor of a group and a file. Numerical results are given for different file sizes. The two main costs of external perfect hashing schemes (cost of insertions and amount of internal memory space required) are also analyzed.

The analysis in chapters 3-5 assumes that perfect hashing functions are found by trial and error, choosing functions at random from the set of all functions. Choosing functions from the set of all functions is highly impractical and in chapter 6 we propose and study a simple class of hashing functions from which functions can be chosen for trials. Experimental results using this class of functions are presented.

Chapter 7 deals with various implementation considerations. Since it is impractical to solve a nonlinear integer optimization problem as described in chapter 4, we give a simple heuristic to determine the rehashing policy. Several other practical aspects such as the organization of header table, etc. are discussed. We also show how the basic scheme can be extended to make it fully dynamic.

In chapter 8 we compare the performance of the proposed external perfect hashing scheme with other hashing schemes for external files. Several open problems arising from the thesis are also discussed.

Chapter 2

An external perfect hashing scheme

2.0. Chapter overview

In this chapter we describe a perfect hashing scheme for external files. It is an implementation of Sprugnoli's idea of segmentation: divide the given set of keys into small groups and use perfect hashing to store each group separately. The basic scheme uses a hashing function for segmentation. Procedures for retrieval, insertion, and deletion of a record are presented. A variant of the basic scheme using a B-tree for segmentation is also outlined along with its advantages. We conclude the chapter by stating the basic problems that must be solved to make the proposed scheme viable. The rest of the thesis is devoted to answering these questions.

2.1. Introduction

Sprugnoli suggested the idea of segmentation to store large static sets using perfect hashing: divide the given set into a number of small groups and store each group separately using perfect hashing. Both the schemes proposed by Fredman et. al. and by Cormack et. al. make use of this idea [FK82, CH85]. The basic external perfect hashing scheme proposed in this chapter is an extension of the two level data structure suggested by Cormack et. al.[CH85]. It uses their basic data structure but each group is stored in a number of contiguous pages on secondary storage.

The data structure consists of a header table stored in internal memory. Each entry in the header table corresponds to a group. Each group of records is stored separately on external storage by a perfect hashing function. The information stored in an entry of the header table is the corresponding group's starting location on external storage, its size, and the parameters of its perfect hashing function.

The set of keys must be partitioned into subsets or groups. The main requirement of the partitioning is that, given a key, it should be possible to easily identify the group to which the key belongs. There are various ways in which partitioning can be accomplished and correspondingly the header table can be organized in several different ways. The simplest way of partitioning a set of keys is by using a hashing function. This corresponds to organizing the header table as a hash table whose size is determined by the number of groups desired. However, by organizing the header table as a tree we can retain ordering between the keys in different groups. Obviously, the header table may be organized using various other data structures.

We first describe in more detail the external perfect hashing scheme assuming that the header table is organized as a hash table. Procedures for retrieval, insertion and deletion are given. A variant of the scheme in which the header table is organized as a B-tree is then outlined. The extra costs and advantages of using a B-tree are discussed.

2.2. The basic scheme

Figure 2.2.1 illustrates the basic external perfect hashing scheme. H is an ordinary hashing function mapping keys into a header table with s entries. Let key group t denote the set of keys $\{x \mid x \in$ the given key set and $t = H(x)\}$ for $0 \leq t \leq s-1$. Each entry in the header table is of the form (p, m, R), where p is a pointer to a group of m contiguous pages on secondary storage and R is the set of parameters defining the perfect hashing function used to store the group of keys. Let (p_t, m_t, R_t) be the header table entry for the key group t. Let page group t denote the set of m_t contiguous pages $p_t, p_t+1, p_t+2, \ldots, p_t + m_t-1$. The address of a record with key x belonging to group t, $t = H(x)$, is given by $p_t + h(x, R_t)$, where h is a perfect hashing function for group t with parameters R_t.

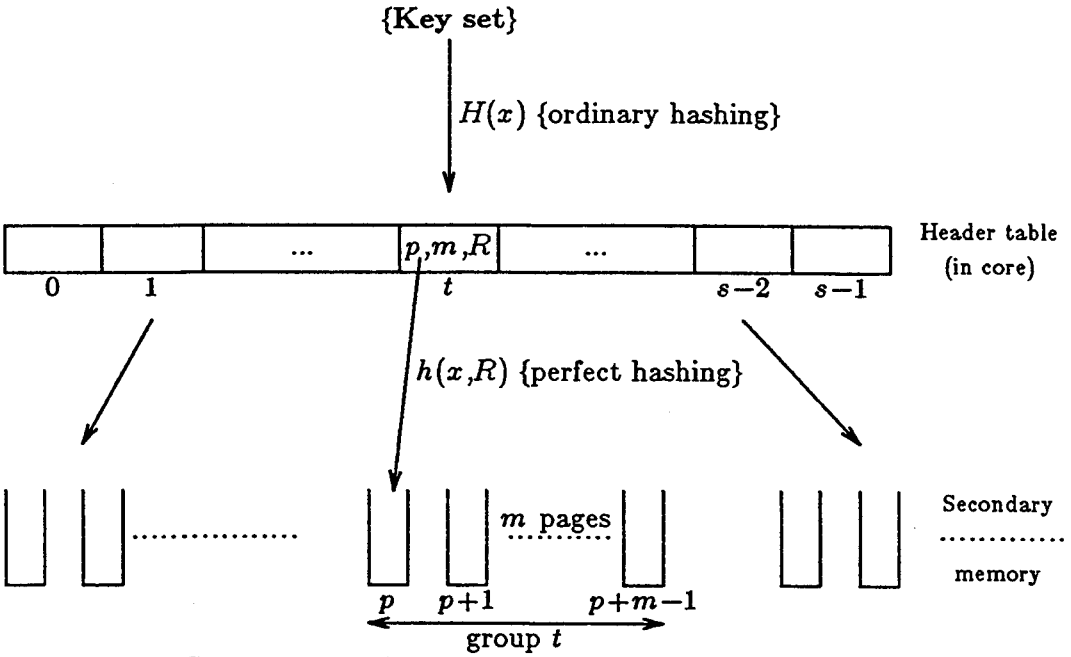


Figure 2.2.1 The external perfect hashing scheme

Algorithms for retrieval, insertion and deletion of a record with key x are outlined below:

Retrieval of a record with key x
Compute the group to which x belongs : $t := H(x)$.
Extract $< p_t, m_t, R_t >$ from entry t of the header table .
Compute the page address of x : $A_x := p_t + h(x, R_t)$.
Read in page A_x.
Search page A_x for key x (if key x is not found, x is not in the file).

Insertion of a record with key x
Compute A_x as above and read in page A_x.
If the page is not full then
    Insert the record into page A_x and write back the page.

else **Perform a rehash as follows:**
    Read in all the pages of the group t; i.e., pages $p_t, p_t+1 , \cdots , p_t + m_t-1$.
    Find a new perfect hashing function for all the keys in the group, including the new key x, using m_u pages (m_u not necessarily equal to m_t).
    Let R_u be the parameters of the new perfect hashing function.
    Find an address p_u on secondary storage having space for m_u contiguous pages (p_u may be equal to p_t).
    Redistribute all the keys (including x) using the new perfect hashing function and store the records on pages $p_u, p_u+1, p_u+2 , \ldots , p_u + m_u-1$.
    Update the header table entry of group t to (p_u, m_u, R_u).

Deletion of a record with key x
Compute A_x as above and read in the page.
Search for key x within page A_x.
If key x is found then delete the record and write back the page
else the desired record is not in the file.
(If there are too many deletions from a group, in order to maintain the storage utilization above a certain lower limit, we may want to rehash the group into fewer pages using a procedure similar to the one outlined for insertions.)

The external perfect hashing scheme described above guarantees retrieval of any record in a single access to secondary storage, provided that the header table is in internal memory. Deletions are straightforward and pose no problem (because there are no overflows). Since the header table is organized as a hash table, there is no ordering between the keys of adjacent groups. Range searches on primary keys cannot be efficiently carried out. A variant of the basic scheme in which the header table is organized as a B-tree has many of the advantages of a standard B-tree file organization, in addition to single access retrieval capability.

B-tree based scheme

Figure 2.2.2 illustrates a variant of the basic external perfect hashing scheme in which the header table is organized as a B-tree. The B-tree is similar to an ordinary B-tree index [KN74], except for the nodes at the lowest level (level 1 in figure 2.2.2) which have entries of the form (p, m, R). Let group t denote the set of keys $\{x \mid x_3 < x \leq x_4\}$ and the entry stored between x_3 and x_4 be (p_t, m_t, R_t). p_t is a pointer to a group of m_t contiguous pages on secondary memory in which the keys of group t are stored using a perfect hashing function with parameters R_t. The page address of a key x belonging to group t is given by $p_t + h(x, R_t)$.

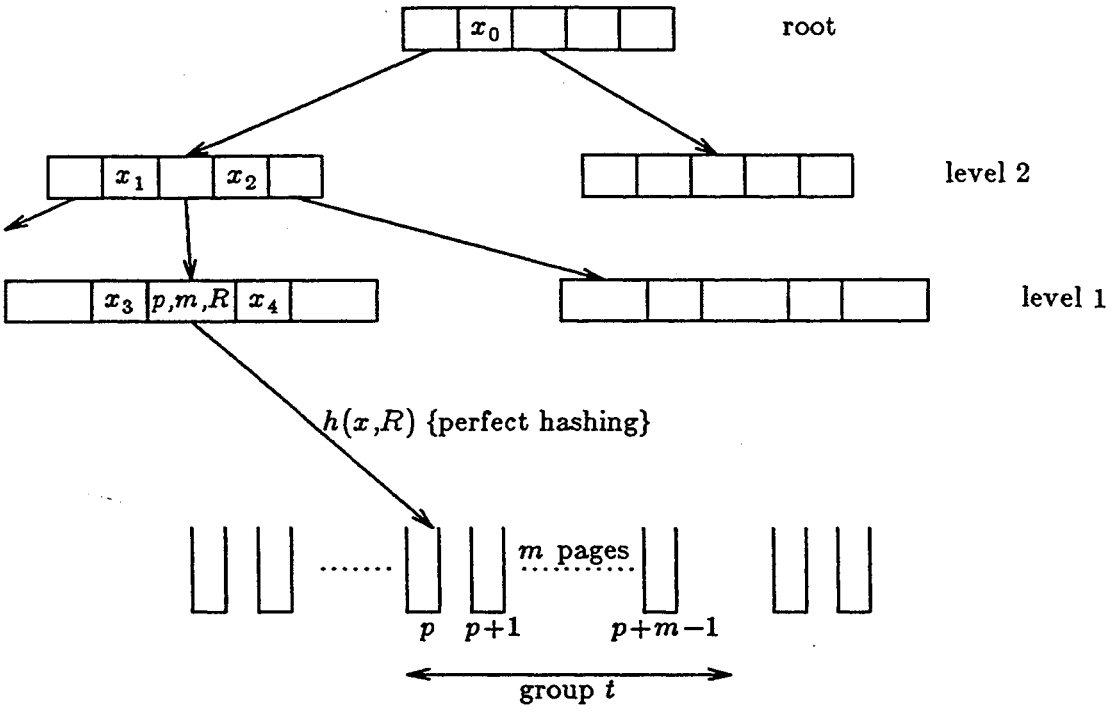


Figure 2.2.2 B-tree based external perfect hashing scheme

Given a key x, the group to which it belongs is found by using the standard B-tree search procedure. Once the group is found, retrieval, insertion, and deletions can be performed using the procedures outlined in the previous section.

There are no rigid bounds on the number of keys in a group. Hence the conditions for splitting a node on the lowest level, level 1, are flexible. One possibility is to fix the maximum number of keys per group, say n_{max}. A group is split into two only when an insertion into the group causes rehashing and the number of keys in the group exceeds n_{max}. The group is not split if the insertion does not cause a rehash. The group is split into two approximately equal groups with a separating key x_i, and the two groups are stored separately by two perfect hashing functions. The original (p, m, R) entry in the node of the header table is replaced by (p_1, m_1, R_1) x_i (p_2, m_2, R_2) increasing the number of entries in the node by one. If this causes the node to overflow, it is split as if it were a normal B-tree node.

A prefix B-tree could be used instead of an ordinary B-tree to reduce the space required to store the separating keys [BU77]. Other forms of B-trees can be used to further reduce the overall memory requirement. As the order of the tree increases, the amount of internal memory required to store the header table for a given number of groups decreases. A 2-3 tree may be used as one extreme case of a B-tree. At another extreme, a simple sorted table could be used.

In addition to guaranteeing single access retrieval of any record, the B-tree based scheme has almost all the advantages of a B-tree file organization. Although there is no ordering of keys within a group, the keys in adjacent groups are ordered. This implies that range searches are more efficient than with a

regular B-tree file structure. The reason is that the pages of a group are contiguous and hence can be read in a single access. However, any range search, no matter how narrow the range, involves at least searching an entire group.

The B-tree based external perfect hashing scheme is completely dynamic; it allows an unlimited number of insertions and deletions, and the file size may grow or shrink without any adverse effects. However, as we shall see later, even the hash table based scheme allows the file size to vary by a considerable amount without adverse effects on the performance. The advantages of the B-tree based scheme are achieved at the cost of increased header table size compared with the primary (hash table based) scheme.

2.3. Open problems

The external perfect hashing schemes outlined above are conceptually simple and straightforward to implement. However, their usefulness depends on the costs involved: computational cost and storage space. The major issues to be considered are:

1. How to find perfect hashing functions or rather, how difficult is it to find perfect hashing functions?
2. How often will it be necessary to rehash a group and what are the costs involved?
3. How large a header table is required? For example, how many kilobytes of internal memory is required to store a file of one million records?
4. What storage utilization (of secondary storage) can be achieved?

Although we have posed four different questions, it is obvious that they are all interrelated. The rest of the thesis attempts to answer these questions. The goal is to show that the external perfect hashing scheme is indeed practical, and that all the costs involved are within reasonable ranges.

The following typical performance result is included here to give a first indication of the performance level that can be achieved. Using a page size of 40 records, a file of 10^6 records can be stored using only 6 kilobytes of internal memory and achieving 80% storage utilization. The file organization allows any record to be retrieved in a single access to secondary storage, and also allows insertions and deletions. The file size may vary by a factor of about 4 without any significant degradation of performance. About 96% of all insertions involve minimum cost: one read followed by one write. The other 4% of insertions are more costly, involving computation of about 8,000 hash addresses of the form $h(x) = (c*x + d) \ mod \ p \ \ mod \ m$, and requiring internal buffer space to store about 800 records for the duration of the computation. In addition, one read access to about 30 contiguous pages followed by one write access is required.

Chapter 3

Finding perfect hashing functions

3.0. Chapter overview

In this chapter we propose a method for finding perfect hashing functions and analyze its performance. The method simply consists of making trials by selecting functions at random from a suitable class of hashing functions. We explain the motivation for choosing this method over other methods reported in the literature.

The main contribution of this chapter is an analysis of the performance of the trial-and-error approach for the case when each bucket can hold more than one record and functions for trials are chosen from the set of all functions. We give a simple recurrence relation for computing the probability of a randomly chosen function being perfect. Previously known methods of computing these probabilities are impractical and several orders of magnitude slower. Based on the numerical results obtained we claim that the cost of finding perfect hashing functions is within a practical range.

The final section deals with limits and approximations. The effect on the expected load factor is examined as the number of pages is increased. Approximate closed form expressions are obtained for the relevant probabilities.

3.1. Introduction

In the previous chapter we proposed a hashing scheme for external files based on perfect hashing. The key question to be answered was how to find perfect hashing functions and the costs involved. In this chapter we propose a solution and analyze its cost.

We make the following observations regarding methods for finding perfect hashing functions proposed in the literature:

(1) All methods proposed so far are rather complicated and their implementation is non-trivial.

(2) They all employ some form of trial-and-error search at one point or the other.

(3) Every method is closely tied to some particular class of functions. Sprugnoli[SP77] considers functions of the form $\lfloor (x+s)/N \rfloor$ and Jaeschke[JS81] considers $\lfloor C/(Dx+E) \rfloor$ mod n.

Can we somehow simplify and generalize the methods for finding perfect hashing functions? Why not choose a general class of functions, and use a simple trial and error approach? If the cost of a trial is low, it may not be worthwhile to spend much time trying to reduce the number of trials. These observations led us to consider the following trial-and-error procedure for finding perfect hashing functions:

> *Given n keys to be hashed into m buckets (pages), each capable of holding up to b keys, choose a function at random from the set of all functions mapping n objects to m objects. Hash all the keys using the chosen function. If none of the pages receive more than b keys then we have found a perfect hashing function. Otherwise, choose another function at random and repeat the process until a function which is perfect for the given set is found.*

Given a set of n objects to be mapped into a set of m objects, there are a total of m^n different mappings. We will refer to this set of functions as the *set of all functions*. In the rest of this chapter a trial refers to this process of choosing a function at random (from the set of all functions, unless otherwise stated) and hashing the keys using the chosen function to verify if it is perfect.

The rest of this chapter deals with the analysis of the trial-and-error method. It is not practical to choose functions from the set of all functions, because $n \log m$ bits would be required to represent a function from this class. In practice, we must restrict the choice to a smaller class of functions. However, there exist simple (to represent and evaluate) classes of functions which have the same behavior (with respect to the relative frequency of perfect hashing functions) as the set of all functions. One such class of functions and experimental results showing that it has the desired behavior are described in chapter 6.

3.2. Probability of a randomly chosen function being perfect

Let $F(n,m,b)$ denote the number of ways in which n keys can be distributed among m pages each having a capacity of b keys, so that none of the pages overflow and let $P(n,m,b)$ denote the probability of a trial succeeding. Then

$$P(n,m,b) = \frac{F(n,m,b)}{m^n}. \tag{3.0}$$

We will often express the number of records in terms of load factor α ($\alpha = 100n/mb$ percent).

The above situation can be modeled by a traditional urn model. Consider the case where n balls are to be distributed into m urns, each with a capacity of b balls. Let each ball be randomly tossed into an urn so that the probability of a ball falling into any particular urn is $1/m$ independent of the outcome of other tossings. The probability that such a random distribution does not cause any urn to overflow is precisely $P(n,m,b)$ defined above.

$P(n,m,b)$ is a measure of the performance of the proposed trial-and-error method for finding perfect hashing functions. For the sake of simplicity, let θ denote $P(n,m,b)$. The trial-and-error process is a sequence of Bernoulli trials, and the probability of the ith trial succeeding is given by $\theta(1-\theta)^{i-1}$. The expected number of trials required to find a perfect hashing function is $\sum_{i=1}^{\infty} i\theta(1-\theta)^{i-1} = \frac{1}{\theta}$. That is, the reciprocal of $P(n,m,b)$ gives the expected number of trials required to find a perfect hashing function.

We first analyze the simple case when $b = 1$, and compare the results with other approaches of direct perfect hashing. Later we analyze the more interesting case when b is greater than one.

Case $b = 1$

Consider the urn model. There are n balls and m urns; the first ball may occupy any one of the m urns. The second ball has $m-1$ choices and so on. The nth ball may occupy any one of the $m-n+1$ free urns. There are a total of m^n possible distributions of the n balls into m urns. $P(n,m,1)$ is then given by

$$P(n,m,1) = \frac{m(m-1)\cdots(m-n+1)}{m^n}. \tag{3.1}$$

Table 3.2.1 compares the performance of the trial-and-error method with that of Jaeschke's[JS81] reciprocal hashing method.

n	Expected number of trials; m			Reciprocal hashing number of trials; m	Load factor which can be obtained by the trial-and-error method for the same cost as that of reciprocal hashing
5	1.7; 20	3.3; 10	26; 5	21; 5	100%
10	3.4; 40	15.3; 20	649; 11	408; 10	87%
15	6.7; 60	71; 30	6322; 18	7710; 15	83%

Table 3.2.1 Comparison of the trial-and-error method with reciprocal hashing.

The first column represents the number of keys. In columns 2, 3 and 4 the first number represents the expected number of trials required to find a perfect hashing function by the trial-and-error method (computed using (3.1)). The second number gives the number of locations used (table size). The 5th column lists the number of trials required to find a perfect hashing function using Jaeschke's reciprocal hashing. Note that they are experimental averages as reported in [JS81]. The value of m in column 4 has been chosen so that the expected number of trials required is as close as possible to that of reciprocal hashing.

Jaeschke points out that for n greater than 10 to 15, all methods of finding direct perfect hashing functions are impractical because of the high cost. He also claims that for small n his method is faster than Sprugnoli's methods. We see from table 3.2.1 that the trial-and-error method compares well with reciprocal hashing, which leads to a rather surprising conclusion: The performance of the simple-minded trial-and-error method is close to that of other known methods for finding (direct) perfect hashing functions. A load factor of 83% to 100% can be obtained with the trial-and-error method when the computation cost is approximately equal to that of reciprocal hashing.

With reference to the proposed external perfect hashing scheme, our aim is to find perfect hashing functions for large sets of keys when each page is capable of holding several keys. Suppose that the group size is in the range of 500 to 1000 keys. Known methods of finding perfect hashing functions have a complexity of at least $O(n^3)$. A complexity of n^3 corresponds to n^2 trials with the trial-and-error method (because each trial requires $\Omega(n)$ computations), which translates into several hundred thousand trials. In view of this, intuitively it appears that one can expect a better performance of the trial-and-error method when the page size is greater than one.

Case $b > 1$

The analysis for the case $b > 1$ is more difficult. David and Barton give the following expression for $F(n,m,b)$ [BD59, DB62]:

$$F(n,m,b) = \sum_{0 \le f_i \le b} \left[n! \Big/ \prod_{i=1}^{m} f_i \right] \tag{3.2}$$

where the summation is over all possible combinations of f_i such that $\sum_{i=1}^{m} f_i = n$. An example makes the above expression clear. $F(4,3,2)$ denotes the number of

ways in which 4 balls can be distributed among 3 urns, each capable of holding at most 2 balls:

$$F(4,3,2) = \frac{4!}{2!1!1!} + \frac{4!}{2!2!0!} + \frac{4!}{0!2!2!} + \frac{4!}{1!2!1!} + \frac{4!}{2!0!2!} + \frac{4!}{1!1!2!} = 54$$

David and Barton point out "$F(N)$ [$= F(n,m,b)$] does not posses a simple form"[DB62, pp221]. The formula is clearly not suited for numerical evaluation of $F(n,m,b)$. However, there is a generating function which may be used to compute $F(n,m,b)$:

$$F(n,m,b) = \text{Coefficient of } (x^n/n!) \text{ in } \Big(G_b(x)\Big)^m, \quad \text{where}$$

$$G_b(x) = (1+x/1!+x^2/2!+ \cdots +x^b/b!).$$

The use of the above generating function requires routines capable of handling very large integers (of the order $b!^m$). Computation of $F(n,30,30)$ took several hours of CPU time on the MAPLE symbolic algebra system running on a VAX-780 [CG83]. For larger values of m and b, it is prohibitively expensive to compute $F(n,m,b)$ using the generating function. However, as explained below we were able to derive a very simple recurrence relation to compute the probabilities.

Recurrence relation for $P(n,m,b)$

Suppose that n balls have already been randomly distributed among m urns and no overflow has occurred. Let the next ball, the $(n+1)$st, be tossed into a randomly chosen urn. Let p_0 denote the conditional probability that the $(n+1)$st ball will not overflow. Then p_0 can be expressed as

$$p_0 = \frac{P(n+1,m,b)}{P(n,m,b)}. \tag{3.3}$$

The numerator represents the joint probability that none of the n balls have overflowed and the $(n+1)$st does not overflow. The denominator is the probability that none of the n balls have overflowed.

The $(n+1)$st ball will overflow if and only if it falls into an urn already full (i.e., one containing b balls). The probability of this event is the same as the probability of a rondomly chosen urn being full. This in turn is the same as the probability of an arbitrary but fixed urn being full (without loss of generality, we may consider the probability of urn number 1 being full). Hence, the probability of the (n+1)st ball overflowing, $(1-p_0)$, can be expressed as

$$1-p_0 = \frac{\binom{n}{b} F(n-b,m-1,b)}{F(n,m,b)}. \tag{3.4}$$

The numerator represents the total number of ways in which the fixed urn will be full (out of the total possible $F(n,m,b)$ ways). It consists of two terms. $\binom{n}{b}$ represents the number of ways in which b balls (those in the full urn) may be chosen from n balls. $F(n-b,m-1,b)$ is the number of ways in which the remaining $(n-b)$ balls may be distributed among the other $(m-1)$ urns so that none of the urns overflow. The denominator, $F(n,m,b)$, represents the total number of

ways in which n balls can be distributed among m urns so that none overflow. By combining equations (3.0), (3.3) and (3.4) we obtain

$$1 - \frac{P(n+1,m,b)}{P(n,m,b)} = \binom{n}{b} \frac{P(n-b,m-1,b)(m-1)^{n-b}}{P(n,m,b)m^n}$$

$$P(n+1,m,b) = P(n,m,b) - \binom{n}{b} P(n-b,m-1,b) \frac{(m-1)^{n-b}}{m^n} \qquad (3.5)$$

It is clear from equation (3.5) that to evaluate $P(n,m,b)$ we need to calculate $P(i,j,b)$ for $j = 1,2,\cdots,m$, and $i = 1,2,\cdots,n-(m-j)(b+1)$. Similar computations are required implicitly by the generating function approach. Although the above recurrence relation looks complicated, involving large numbers of the order of m^n, the computation can be organized so that the evaluation of each new value of $P(n,m,b)$ requires only 5 floating point operations. The following algorithm computes a table of $P(n,m,b)$ for a fixed b and $1 < m \leq mmax$, $1 \leq n \leq b * mmax$. The procedure is based on (3.5) and the following identity:

$$\binom{n}{b}\frac{(m-1)^{n-b}}{m^n} = \left(\frac{n}{n-b}\right)\left(\frac{m-1}{m}\right)\left\{\binom{n-1}{b}\frac{(m-1)^{n-b-1}}{m^{n-1}}\right\}$$

The iterations start with the initializations $P(n,m,b) = 1.0$, for $0 \leq n \leq b$, $m \leq mmax$.

Algorithm to compute a table of $P(n,m,b)$.

```
procedure pnmb (mmax, b, P)
b, mmax : integer ;
P: array[0..b * mmax, 1..mmax] of real;

begin
    m, n, deficit : integer;
    term, skew : real;

    for m := 1 to mmax do
        for n := 0 to b do
            P[n,m] := 1.0;

    for m := 2 to mmax do
        deficit := b;
        term := 1.0 ;
        skew := (m-1.0)/m;
        for n := b+1 to m * b do
            divide(term, m, deficit);
            P[n,m] := P[n-1,m] - term * P[n-b-1,m-1] ;
            term = term * skew * n/(n-b);
        endloop;
    endloop;
end;
```

```
procedure divide(term, m, deficit)
term : real;
m, deficit : integer;

begin
      while( deficit>0 and term>mcepsilon) do
            term := term/m;
            deficit := deficit - 1;
      endloop;
end;
```

The above procedure is a direct implementation of the recurrence relation (3.5). The initial value of *term* should actually be $1/m^b$. For large values of m and b, this may lead to *term* having a value too small to be represented by a floating-point number. However, as the iteration progresses, the value of *term* increases slowly. The procedure *divide* handles this problem by not allowing the value of *term* to go very much below machine epsilon. The incorrect value of *term* at the start of the iteration does not cause any error because in the main procedure the product of *term* and a probability is subtracted from another probability (P[n,m] := P[n–1,m] – term * P[n–b–1, m–1]). Considering the range of values of the probabilities at the beginning of the iteration (close to 1.0), if *term* is less than machine epsilon it is as good as being zero for subtractions.

The numbers involved are well scaled and the procedure is computationally stable. Roundoff errors do not cause any problems over the practical range of load factors. When the load factor approaches 1.0, roundoff errors become significant and $P(n,m,b)$ may become negative. This situation may easily be corrected by an iterative correction procedure. In practice this problem arises only when $P(n,m,b)$ is so small that it can be assumed to be zero.

Figures 3.2.1 and 3.2.2 plot the probabilities $P(n,m,b)$ against the load factor, computed using the procedure given above. The higher curves correspond to lower values of m. The graphs indicate that the probability of a trial succeeding drops very rapidly from almost 1.0 to almost 0.0 within a narrow load factor range. This critical region shifts slowly towards zero as the value of m increases.

The plots of the results are somewhat surprising. For example, $P(320,10,40) = 0.49$, meaning that we can expect to find a perfect hashing function to distribute 320 keys into 10 pages of size 40 (corresponding to a load factor of 80%) in little over two trials. However, if we want to find a perfect hashing function for 1280 keys at 80% load factor (corresponding to 40 pages), we need 30 trials on the average. For the range of n given in the examples, a cost of about 10 trials corresponds to a complexity of $n \log n$ hash address evaluations. These results indicate that finding perfect hashing functions is not too difficult and it appears that the proposed external perfect hashing scheme may be practical (of course, this is true only if the load factor is not too high).

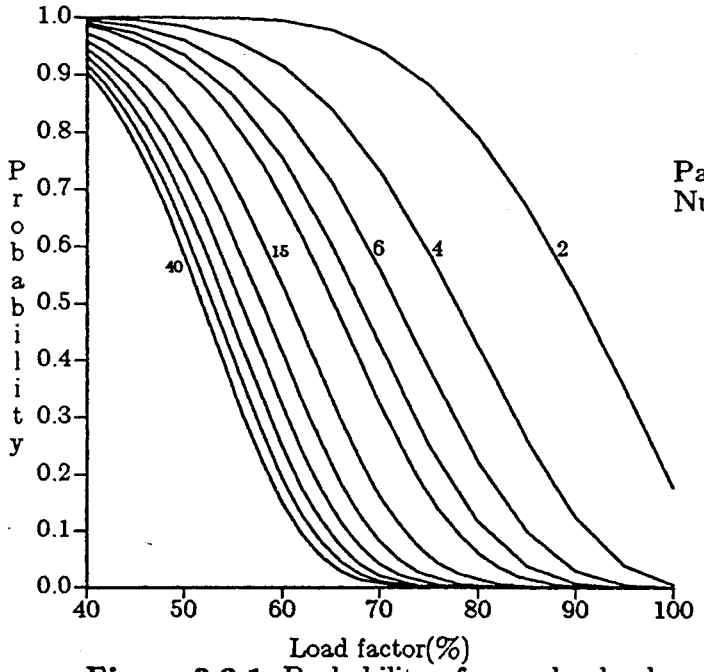


Figure 3.2.1 Probability of a randomly chosen function being perfect

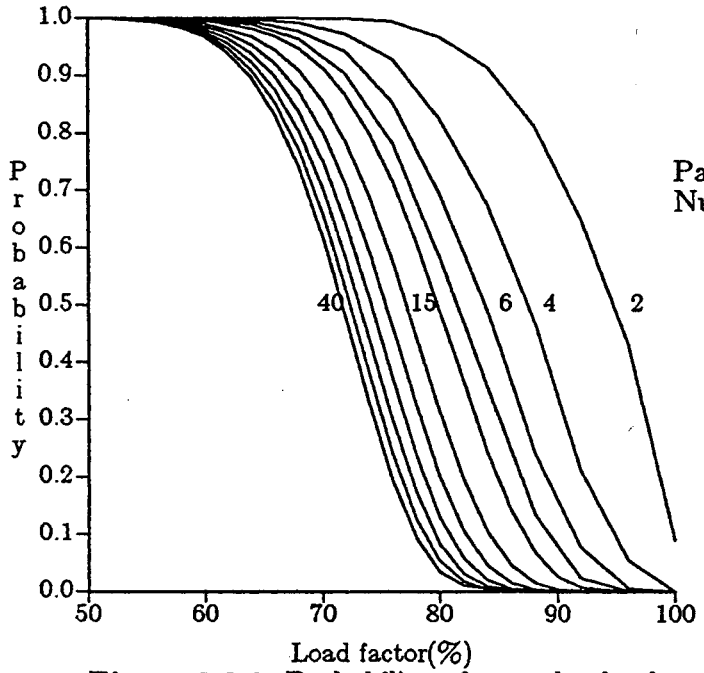


Figure 3.2.2 Probability of a randomly chosen function being perfect

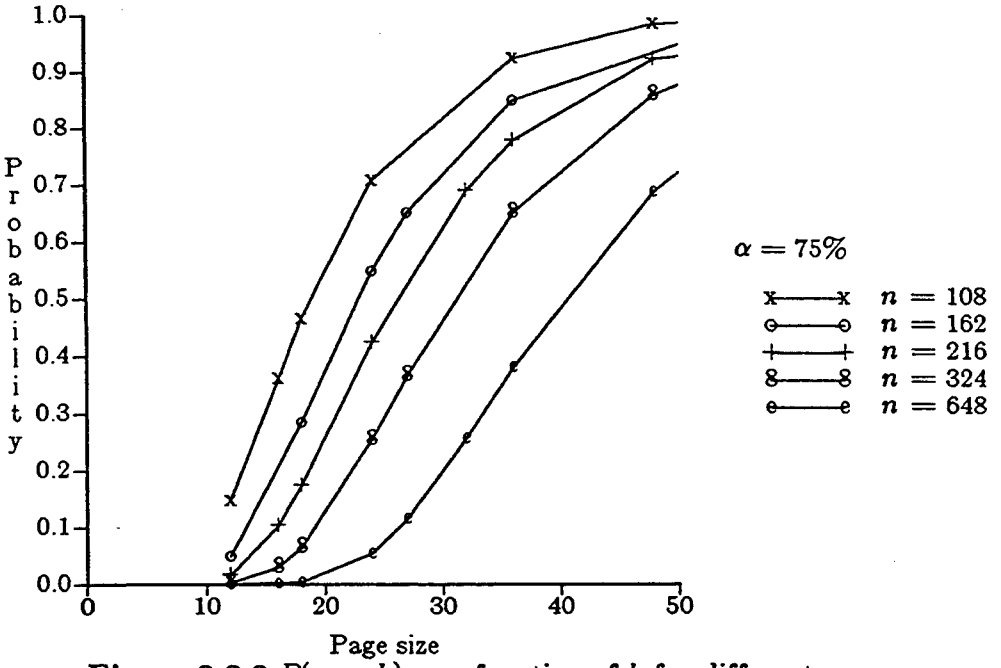


Figure 3.2.3 $P(n,m,b)$ as a function of b for different n

Figure 3.2.3 illustrates the effect on $P(n,m,b)$ of increasing the page size. Here the load factor is kept constant at 75% (m is varied so that n/mb=0.75) and the resulting $P(n,m,b)$ is plotted as a function of b. One curve is plotted for each value of n (= 108, 162, 216, 324 and 648). It is clear that, for a given set of keys, increasing the page size b while keeping the storage utilization constant, improves the probability of success of a trial. There is a critical range of b for any given key set. Too low a value of b results in a prohibitively high cost of finding perfect hashing functions. On the other hand, very large values of b do not result in a corresponding $P(n,m,b)$ advantage.

3.3. Approximate formulae

In figures 3.2.1 and 3.2.2, as the value of m is increased, the space between adjacent $P(n,m,b)$ curves narrows. One would expect that as m tends to infinity, the critical region would approach zero. We wish to study the movement of the critical region for increasingly large values of m. The practical significance of this study is that it helps us understand the behavior of the trial-and-error method for exceptionally large groups.

We first obtain an approximate, closed form expression for $P(n,m,b)$. The analysis is based on the urn model described at the beginning of section 3.2. The first approximation is to assume that pages overflow independently when keys are distributed randomly over the pages. In reality, for a given number of pages and keys, when one of the pages overflows there are fewer keys left for other pages and hence the probability of some other page overflowing is reduced. As we shall see, the error due to this simplification is significant only when the load factor is close to 1.0.

Let $Pov(n,m,b)$ denote the probability of an arbitrary but fixed page overflowing when n keys are randomly hashed into m pages, each having a capacity of b records. Pov may be expressed as the following sum:

$$Pov(n,m,b) = \sum_{i=b+1}^{n} \binom{n}{i} \left(\frac{1}{m}\right)^{i} \left(1 - \frac{1}{m}\right)^{n-i} \tag{3.6}$$

Under the assumption that pages overflow independently of each other, the probability that none of the pages overflow, $P(n,m,b)$, is given by

$$P(n,m,b) \approx \Big(1 - Pov(n,m,b)\Big)^{m}. \tag{3.7}$$

Although the above equations may be used to compute $P(n,m,b)$ approximately, they are rather complicated and do not help us to understand the behavior of $P(n,m,b)$ for increasingly large values of m. However, using the customary Poisson approximation of the binomial distribution we obtain the following (This is a good approximation for moderate values of $b\alpha$ - in our case $5 \leq b\alpha \leq 50$)[FL68]:

$$Pov(\alpha,b) \approx \sum_{i=b+1}^{n} \frac{e^{-b\alpha}(b\alpha)^{i}}{i\,!} \tag{3.8}$$

where $b\alpha = n/m$ is the average number of records per page, and $Pov(\alpha,b)$ is same as the probability $Pov(n,m,b)$ with the parameter $\alpha = n/mb$. For large values of b and α not too close to 1.0, the summation may be approximated as follows:

$$\begin{aligned} Pov(\alpha,b) &\approx \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\left\{1 + \frac{b\alpha}{b+2} + \frac{(b\alpha)^2}{(b+2)(b+3)} + \cdots\right\} \\ &\approx \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\left\{1 + \frac{b\alpha}{b+2} + \frac{(b\alpha)^2}{(b+2)^2} + \cdots\right\} \\ &= \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\left\{1 \,/\, (1 - \frac{b\alpha}{b+2})\right\} \\ Pov(\alpha,b) &\approx \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\left\{\frac{b+2}{b(1-\alpha)+2}\right\} \end{aligned} \tag{3.9}$$

For large values of m the expression $(1-Pov)^m$ can be approximated by e^{-mPov}. Since Pov is a function of α and b only, it is appropriate to express $P(n,m,b)$ as a function of α, m and b. Hence from (3.7) we have,

$$P(\alpha,m,b) \approx e^{-m\ Pov(\alpha,b)}. \tag{3.10}$$

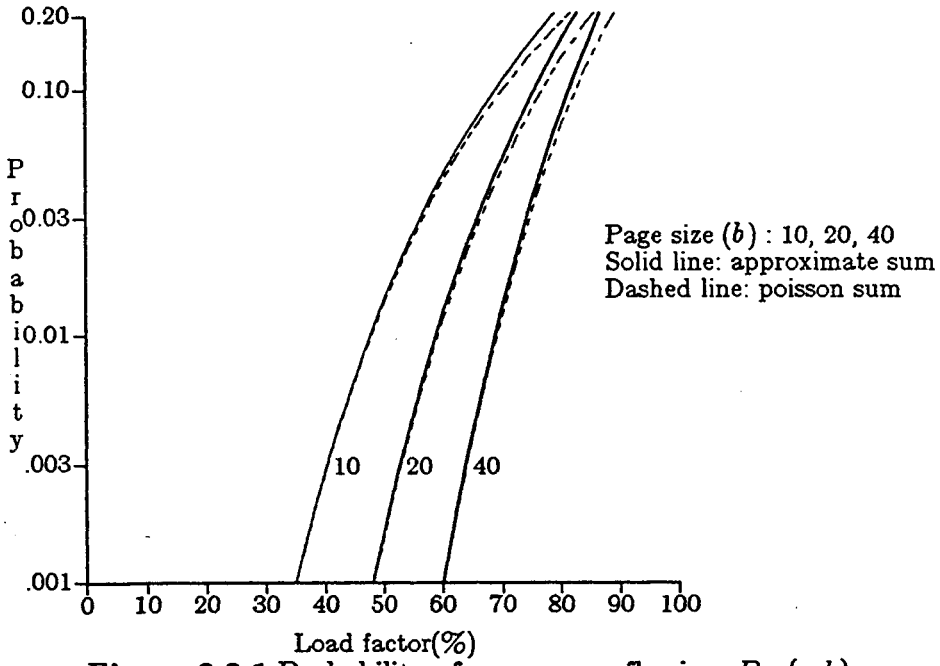


Figure 3.3.1 Probability of a page overflowing, $Pov(\alpha,b)$

These simple equations help us understand the behavior of $P(\alpha,m,b)$ for increasingly large values of m. Because of the number of assumptions involved, first we investigate how closely equations (3.9) and (3.10) approximate the correct values. Figure 3.3.1 plots $Pov(\alpha,b)$ for b = 10, 20 and 40. The solid lines correspond to Pov given by (3.9) and the dashed lines correspond to Pov given by (3.8). For a given load factor, a higher value of b increases the accuracy of (3.9). Later it will be clear that we are interested in (3.9) when $Pov(\alpha,b)$ values are low. As can be seen from the graph, (3.9) is a good approximation of (3.8) when the load factor is below a certain value. ((3.8) itself is a very good approximation to (3.6)). In figure 3.3.2, $P(\alpha,m,b)$ is plotted against the load factor α for different values of m. The solid lines correspond to the approximate probabilities computed using (3.9) and (3.10). The dashed lines represent the exact values obtained using the procedure given in section 3.2. The solid lines almost coincide with the dashed lines and hence we conclude that equations (3.9) and (3.10) are acceptable approximations (later it will be clear that we are especially interested in the case when m is over 50 and $P(\alpha,m,b)$ is around 0.05 to 0.20).

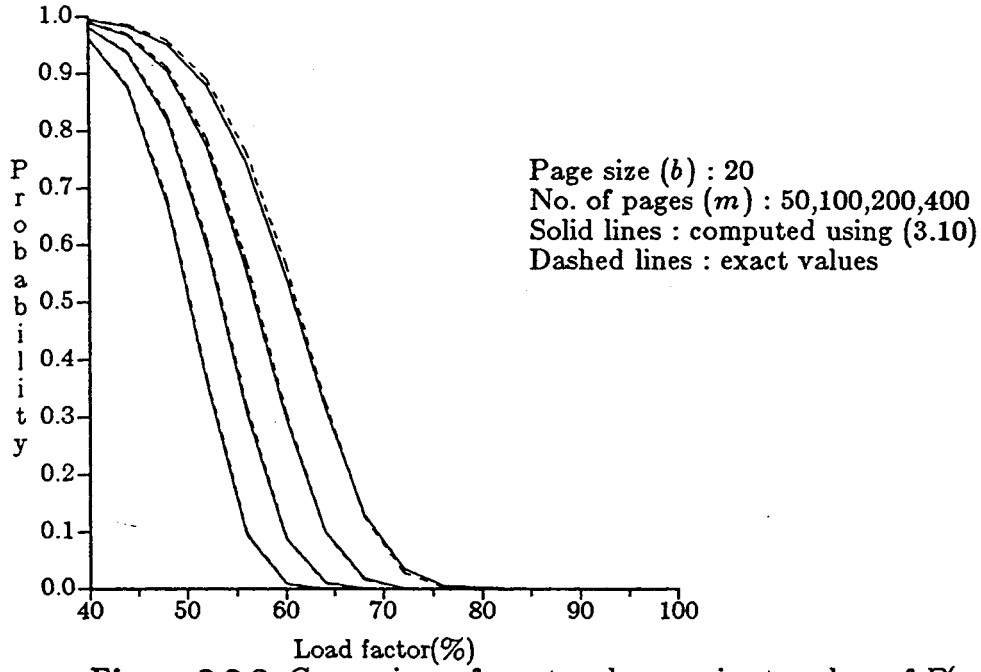


Figure 3.3.2 Comparison of exact and approximate values of $P(n,m,b)$

In figure 3.3.2, roughly every 4% drop in the load factor α allows a doubling of m while keeping the probability $P(\alpha,m,b)$ constant at 0.1. This implies that $Pov(\alpha,b)$ is halved for every 4% reduction in α when α is around 65% to 70%. Consider equation (3.9), and take its derivative:

$$Pov(\alpha,b) = \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\, \frac{b+2}{b+2-b\alpha}$$

$$\frac{d}{d\alpha}\, Pov(\alpha,b) = \frac{(b\alpha)^{b+1}}{(b+1)!}\, e^{-b\alpha}\, \frac{b+2}{b+2-b\alpha} \left\{ \frac{b+1}{\alpha} - b + \frac{b}{b+2-b\alpha} \right\}$$

$$= Pov(\alpha,b) \left[\frac{b+1}{\alpha} - b\left(\frac{b+1-b\alpha}{b+2-b\alpha}\right) \right]$$

$$\frac{d}{d\alpha}\, Pov(\alpha,b) \approx Pov(\alpha,b) \left[b\ (1/\alpha - 1) \right] \qquad \text{(for large } b \text{ and } \alpha \text{ not too near 1.0)}$$

The above derivative explains the behavior of $P(\alpha,m,b) = e^{-m\ Pov(\alpha,b)}$ for increasingly large values of m. A small change in α affects $Pov(\alpha,b)$ by a large factor of $b(1/\alpha - 1)$. For example, with $b = 20$ and $\alpha = 0.6$, a reduction of 4% in the value of the load factor ($\Delta\alpha = 0.04$) reduces $Pov(\alpha,b)$ to half its value, ($\Delta\ Pov(.6,20) \quad = Pov(0.6,20)*(20(1/0.6 - 1)) * 0.04 \quad = 0.5\ Pov(.6,20)$). This implies that, the effect of doubling the value of m on the probability of success of a trial can be compensated by reducing the load factor by only 4%. For increasingly larger values of m, α would be decreasing gradually (keeping $P(\alpha,m,b)$ constant) thereby increasing the value of the factor $b(1/\alpha - 1)$. Thus for each successive doubling of m, smaller and smaller sacrifices in the load factor will suffice to maintain the value of $P(\alpha,m,b)$ constant.

This result is useful in explaining the worst case behavior of the proposed external perfect hashing scheme. Suppose we have designed a scheme with a certain expected number of keys per group, but because of a skew in the main hashing function, some groups are very large (twice the expected size, say). We now claim that the load factor of the group would decrease very little. In other words, having a few very large groups does not cause serious problems.

3.4. Conclusions

In this chapter we proposed using a simple trial-and-error method for finding perfect hashing functions. The analysis indicate that the performance of the method is comparable to that of other methods for small static sets stored in internal memory. The main contribution of this chapter is the derivation of a recurrence relation for computing the probability of a random distribution of n balls into m urns resulting in no overflows (which is the probability of a trial succeeding in finding a perfect hashing function). The results indicate that, on the average, one can find a perfect hashing function to distribute several hundred keys within a few trials, provided that the load factor is not too high. We also derived an approximate closed form expression for the above probability. These approximate formulae are of general interest in themselves. They also help us understand the limiting behavior of the probabilities.

All the results were obtained assuming that hashing functions are chosen at random from the set of all functions. This is clearly impractical. In chapter 6 we show that there exist practical classes of hashing functions which behave as predicted by the theoretical analysis of this chapter.

Chapter 4

Space-time tradeoff and rehashing policies

4.0. Chapter overview

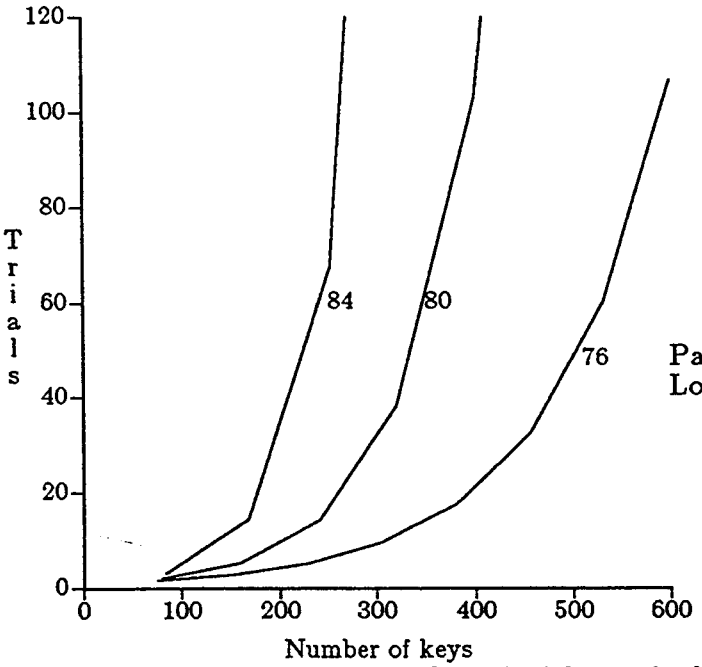
In the previous chapter we studied a method for finding perfect hashing functions and the relationship between its cost and the load factor of the group. In this chapter we consider policies for balancing this tradeoff. For a detailed study, we have chosen a policy that attempts to limit the cost of finding perfect hashing functions and the reasons for this choice are explained. Determining the optimal policy involves solving a nonlinear integer programming problem. In general, such problems are very expensive to solve. Thus, a heuristic procedure based on dynamic programming is introduced to give an approximate solution.

4.1. Difficulty of balancing the cost of finding a perfect hashing function

When a record is to be inserted into a file organized using the proposed external perfect hashing scheme, first the group to which it belongs is determined. Then the page address is computed using the perfect hashing function associated with the group. If that page is not full, the record is inserted into the page. In this case the current hashing function for the group remains perfect for the enlarged group. If the page is full, a new perfect hashing function must be determined for the enlarged group. The records are then redistributed using the new perfect hashing function and the group is relocated if necessary. We call this operation a **rehash**.

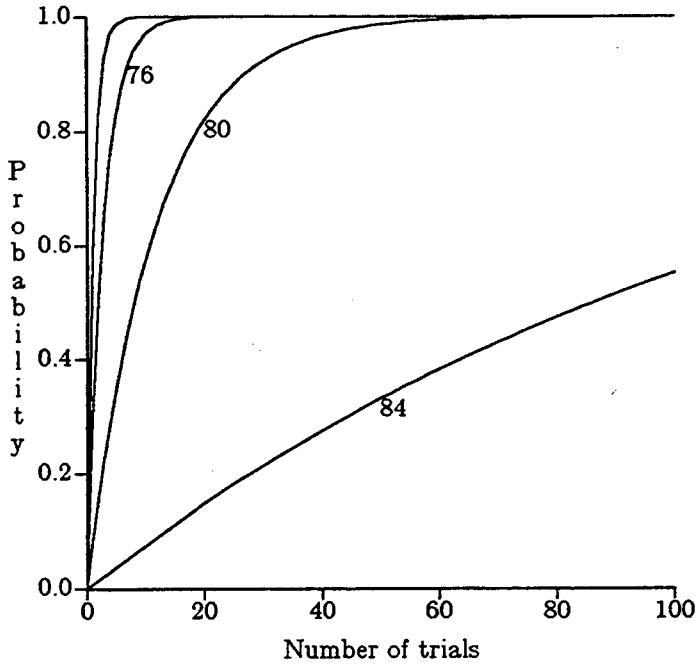
In the previous chapter we analyzed a trial-and-error method for finding perfect hashing functions. The analysis indicated that there is a sharp tradeoff between the cost of finding a perfect hashing function and the load factor (number of pages) of the group. There is a critical load factor range in which the probability of success of a trial falls rapidly. Generally, the aim is to achieve as high a load factor as possible. However, it is clear that we need to operate within the critical region: if the load factor is too high, the probability of success of a trial is close to zero and the cost of rehashing is prohibitively high. The question is then how to choose an operating point in practice? In other words, given a set of keys how does one choose the number of pages for making trials in finding a perfect hashing function? Since the critical region depends on the group size, this is a nontrivial problem.

Suppose we fix the load factor at some desired value. This has the consequence that rehashing becomes more and more expensive as the group size increases. As a matter of fact, the load factor cannot be fixed exactly but only restricted to a narrow range, because only an integral number of pages can be allocated to a group. This effect is well illustrated by Figure 4.1.1 in which the expected number of trials required to find a perfect hashing function is plotted as a function of the number of records in the group at a fixed load factor. (The expected number of trials, $1/P(n,m,b)$, is plotted as a function of n, m is varied so as to keep the load factor n/mb constant). The page size is 20, and there is one curve plotted for each load factor of 76%, 80% and 84%. At a fixed load factor, the cost of rehashing increases rapidly as the group size increases. For example, suppose we choose a load factor of 80%. The expected cost of finding a perfect hashing function is about 15 trials when the group size is 250. As the group size increases to 320, the expected cost raises to 40 trials. If the group size further increases to 400, the expected number of trials exceeds 100. Clearly it is necessary to reduce the load factor to keep the cost of rehashing within reasonable limits.



Page size (b): 20
Load factor : 76, 80, 84 %

Fig. 4.1.1 Expected number of trials required to find a perfect hashing function



Page size (b): 40
Number of pages (m): 30
Load factor(percentage):
72, 76, 80, 84

Fig. 4.1.2 Probability of success in finding a perfect hashing function

Another alternative is to fix the expected number of trials. (Here again we can only restrict the cost to a certain range but cannot fix it exactly.) A problem with this approach is the large variance of the cost of rehashing. This is illustrated by figure 4.1.2, which also shows the dramatic effect of load factor variations on the cost of rehashing. For the sake of simplicity, let θ denote $P(n,m,b)$. The probability of ith trial succeeding is given by $\theta(1-\theta)^{i-1}$ (refer to section 3.2 for details). The probability of finding a perfect hashing function within t trials is given by $\sum_{i=1}^{t}\theta(1-\theta)^{i-1}$. In figure 4.1.2, the probability of finding a perfect hashing function within t trials is plotted as a function of t. The page size is 40 and the group size is fixed at 30 pages. There is one curve corresponding to each of 72%, 76%, 80% and 84% load factors. Consider the curve corresponding to a load factor of 80%. $P(\ 80\%,\ 30,\ 40) = 0.082$, which corresponds to an expected cost of 12.19 trials. Assuming that the expected cost of rehashing is acceptable, we see that the variation in the number of trials is very high. For example, in 30% of the cases only 3 trials are required to find a perfect hashing function, whereas in 10% of the cases we may not have succeeded after 28 trials. Furthermore, in 5% of the cases more than 40 trials are required to find a perfect hashing function. In practice, this wide variation in the number of trials may not be acceptable. In an attempt to overcome the above mentioned problems, we suggest a policy for rehashing in the following section.

4.2. Rehashing policies

Consider the problem of distributing 180 keys into pages of size 20. At least 9 pages are required corresponding to a 100% load factor. The probability of a trial succeeding in finding a perfect hashing function at 100% load factor is extremely small and for all practical purposes equal to zero. With 18 pages, corresponding to a 50% load factor, the probability of a trial succeeding is close to 1.0. Table 4.2.1 gives the values of $P(180,\ m,\ 20)$ for values of m from 9 to 18 pages. To find a perfect hashing function for the given set of keys, it is not necessary to make all the trials with a fixed number of pages. The trials may be distributed over different group sizes; starting with a small number of pages and gradually increasing the number of pages as trials progress until a trial succeeds. This enables us to limit the cost of rehashing.

Number of pages (m)	10	11	12	13	14	15	16	17	18
Load Factor (%)	90	81.8	75	69.2	64.3	60	56.3	52.9	50
$P(180,\ m,\ 20)$	.005	.100	.331	.578	.758	.867	.929	.962	.980

Table 4.2.1 Probability of a trial succeeding

In general, given n, the number of keys, we suggest restricting the number of pages, m, in the group to some interval $m_l \leq m \leq m_h$. The lower bound m_l is $\lceil n/b \rceil$. The upper bound m_h is chosen so that $P(n,\ m_h,\ b)$ is close to 1.0. For the range of n and b we are interested in, this can be achieved by choosing m_h to correspond to a load factor of around 50%. Let t_{max} be an upper bound† on the number of trials we are willing to make. We distribute these trials among r

stages, where $r = m_h - m_l + 1$, that is, we partition t_{max} into $(t_1, t_2, \ldots, t_r)$ such that $t_1 + t_2 + \cdots t_r = t_{max}$.

To find a perfect hashing function make up to t_1 trials with m_l pages. If there is no success, try up to t_2 functions with $(m_l + 1)$ pages and so on. If there is no success even after making t_r trials with m_h pages, continue making trials with m_h pages until a perfect hashing function is found. When trying functions with m_h pages (corresponding to around a 50% load factor), the probability of a trial succeeding is close to 1.0 and hence we can expect to succeed almost immediately. However, there is no guarantee of success.

Every partitioning of t_{max} into $t_1, t_2, \ldots, t_r$ defines a policy of distributing trials; hereafter referred to as a **rehashing policy**. Making all trials with a fixed number of pages (constant load factor) corresponds to the partitioning $(0, 0, \ldots, t_{max}, 0, 0 \cdots, 0)$. The problems resulting from such a policy were discussed in section 4.1. Note that we use the terms *policy* and *partitioning* synonymously with *rehashing policy.*

Our aim is to define an optimal rehashing policy in the next section. However, we first discuss the general implications of a policy after introducing some additional notation in table 4.2.2.

i	1	2	···	r
Number of pages: $(m_l + i - 1)$	m_l	$m_l + 1$	···	$m_h = m_l + r - 1$
Probability of failure of a trial with $(m_l + i - 1)$ pages, $q_i = 1 - P(n, m_l + i - 1, b)$	q_1	q_2	···	q_r
Number of trials with $(m_l + i - 1)$ pages, t_i	t_1	t_2	···	t_r
Probability of a policy succeeding with $(m_l + i - 1)$ pages: $R_i(n)$	$R_1(n)$	$R_2(n)$	···	$R_r(n)$
Probability of accommodating the $(n + 1)$st key without a rehash: p_{i0}	p_{10}	p_{20}	···	p_{r0}

Table 4.2.2 Notation used

Rows 1 - 3 need no explanation. In row 4, $R_i(n)$ denotes the probability that the rehashing policy succeeds with $(m_l + i - 1)$ pages. We can obtain an expression for $R_i(n)$ as follows:

The probability of all the t_1 trials with m_l pages failing $= q_1^{t_1}$.
The probability of the policy resulting in m_l pages, $R_1(n) = (1 - q_1^{t_1})$.
The probability of all the t_2 trials with $m_l + 1$ pages failing $= q_2^{t_2}$.
The probability of the policy resulting in $(m_l + 1)$ pages, $R_2(n) = q_1^{t_1}(1 - q_2^{t_2})$

---

† t_{max} will be a "soft" bound in the sense that there is a certain probability of having to make more than t_{max} trials. However, the probability of this occurring can be made arbitrarily small.

Hence, we have $R_i(n) = {q_1}^{t_1}{q_2}^{t_2} \cdots {q_{i-1}}^{t_{i-1}}(1 - {q_i}^{t_i}),\ 1 \leq i \leq r-1$ (4.1)

and $R_r(n) = {q_1}^{t_1}{q_2}^{t_2} \cdots {q_{r-1}}^{t_{r-1}}$ (4.2)

Because we continue trials with m_r pages until success, the term $(1 - {q_r}^{t_r})$ is absent in the expression for $R_r(n)$.

When n keys have been distributed over $(m_l + i - 1)$ pages by some perfect hashing function, all the pages cannot be full (unless n is precisely equal to $b(m_l+i-1)$). If the $(n + 1)$st key is hashed using the current perfect hashing function, there is a certain probability that the new key will not overflow (i.e., the current perfect hashing function is also perfect for the enlarged set of $n + 1$ keys). This probability is denoted, in row 4, by p_{i0}, and can be computed as

$$p_{i0} = \frac{P(n + 1, m_l + i - 1, b)}{P(n\ , m_l + i - 1, b)}. \tag{4.3}$$

4.3. Implications of a rehashing policy

The main problem is how to partition $t_{\max}$. How to choose a rehashing policy? Before we answer this question in the next section, several implications of a rehashing policy are discussed below.

1. Probability of success in $t_{\max}$ trials

The probability of a rehashing policy $(t_1, t_2, \ldots, t_r)$ succeeding in $t_{\max}$ trials is $(1 - {q_1}^{t_1}{q_2}^{t_2} \cdots {q_r}^{t_r})$. When using a trial-and-error method of finding perfect hashing functions, there is no strict upper bound on the number of trials. However, the probability of a trial succeeding with m_h pages is nearly 1.0. This enables us to impose a bound on the maximum cost of rehashing. We can specify a minimum probability (say 0.99) of success in finding a perfect hashing function within $t_{\max}$ trials.

2. Expected number of trials

Every policy has associated with it a certain expected number of trials required for success, $E(t)$. A policy which favors trials with a higher number of pages has a lower expected number of trials. $E(t)$ is given by the following expression:

$$\begin{aligned} E(t) = {} & 1(1 - q_1) + 2q_1(1 - q_1) + \cdots + t_1{q_1}^{t_1-1}(1 - q_1) \\ & + (t_1 + 1){q_1}^{t_1}(1 - q_2) + \cdots + (t_1 + t_2){q_1}^{t_1}{q_2}^{t_2-1}(1 - q_2) \\ & + \cdots + (t_1 + t_2 + \cdots + t_r){q_1}^{t_1}{q_2}^{t_2} \cdots {q_r}^{t_r-1}(1 - q_r) \\ & + \sum_{j=1}^{\infty} (t_1 + t_2 + \cdots + t_r + j){q_1}^{t_1}{q_2}^{t_2} \cdots {q_r}^{t_r+j-1}(1 - q_r) \end{aligned} \tag{4.5}$$

3. Load factor

The expected number of pages, $E(m)$, resulting from a policy is given by

$$E(m) = m_l(1 - {q_1}^{t_1}) + (m_l + 1){q_1}^{t_1}(1 - {q_2}^{t_2}) + \cdots$$
$$+ (m_l + r - 2){q_1}^{t_1}{q_2}^{t_2} \cdots {q_{r-2}}^{t_{r-2}}(1 - {q_{r-1}}^{t_{r-1}})$$
$$+ m_h{q_1}^{t_1}{q_2}^{t_2} \cdots {q_{r-1}}^{t_{r-1}}(1 - {q_r}^{t_r}) + m_h{q_1}^{t_1}{q_2}^{t_2} \cdots {q_r}^{t_r}$$

The next to last term accounts for the result of t_r trials with m_h pages succeeding. The last term accounts for trials made beyond $t_{\max}$; ${q_1}^{t_1}{q_2}^{t_2} \cdots {q_r}^{t_r}$ is the probability of none of the $t_{\max}$ trials succeeding, in which case the trial-and-error method continues making trials with m_h pages until success. The last two terms can be combined and $E(m)$ may be written as follows:

$$E(m) = m_l(1 - {q_1}^{t_1}) + (m_l + 1){q_1}^{t_1}(1 - {q_2}^{t_2}) + \cdots$$
$$lineup + m_h{q_1}^{t_1}{q_2}^{t_2} \cdots {q_{r-1}}^{t_{r-1}} \tag{4.6}$$

The corresponding load factor is then given by $E(lf) = \dfrac{n}{E(m) * b}$

A policy favoring fewer pages does not necessarily yield a higher load factor, because the probability of success is also lower. It is not easy to see exactly what policy will maximize the resulting load factor.

4. Probability of the next insertion causing overflow

Suppose a key is to be inserted into a group for which a perfect hashing function was found using the policy $(t_1, t_2, \ldots, t_r)$. There is a certain probability that the new key will overflow and cause a rehash. This probability, denoted by $RH(n)$, is given by

$$RH(n) = 1 - \sum_{i=1}^{r} p_{i0} \cdot R_i(n). \tag{4.7}$$

The probability p_{i0} increases as i increases (number of pages $= m_l + i - 1$) and hence a policy which favors trials with larger number of pages is preferable.

4.4. Optimal rehashing policies

In the last section we considered the effects of a policy on various factors. In practice it is desirable to have as high a load factor as possible and at the same time limit the number of trials required to find a perfect hashing function. Accordingly we will define in this thesis an **optimal policy** as follows:

> *Given $t_{\max}$, P_s and a set of keys, we say that a partitioning $(t_1^*, t_2^*, \ldots, t_r^*)$ of $t_{\max}$ is optimal if the resulting number of pages is minimized subject to the condition that the probability of success in finding a perfect hashing function within $t_{\max}$ trials is at least P_s.*

Mathematically, the optimal rehashing policy is the solution of the following nonlinear optimization problem:

minimize

$$E(m) = m_l(1 - q_1^{t_1}) + (m_l + 1)q_1^{t_1}(1 - q_2^{t_2}) + \cdots + m_h q_1^{t_1} q_2^{t_2} \cdots q_{r-1}^{t_{r-1}}. \tag{4.6}$$

subject to the conditions:

$$t_1 + t_2 + \cdots + t_r = t_{\max} \tag{4.8}$$

$$q_1^{t_1} q_2^{t_2} \cdots q_r^{t_r} \leq 1 - P_e. \tag{4.9}$$

The solution $(t_1^*, t_2^*, \ldots, t_r^*)$ defines the following policy. Try up to t_1^* hashing functions chosen at random with m_l pages. If a perfect hashing function has not been found by then, try up to t_2^* hashing functions with $(m_l + 1)$ pages and so on. Within a total of $t_{\max}$ trials, a perfect hashing function will be found with probability P_e or higher. If there is no success in $t_{\max}$ trials, continue making trials with m_h pages until a perfect hashing function is eventually found. The expected number of trials required beyond $t_{\max}$ is very small. However, as already mentioned, there is no strict upper bound on the number of trials required.

The objective function (4.6) is a nonlinear function of the variables $t_1, t_2, \ldots, t_r$. The constraint represented by (4.8) is linear, and later we will show how (4.9) can also be converted into a linear constraint. The variables $t_1, t_2, \ldots, t_r$ represent the number of trials and hence have to be positive integers. Thus the optimization problem is a nonlinear integer programming problem. In general, such problems are very difficult to solve. However, before discussing the solution procedure, we need to examine the usefulness of the solution.

In practice, our aim of solving the optimization problem is to determine the optimal rehashing policy and then follow that policy in finding a perfect hashing function. The combined cost of finding the optimal rehashing policy and following it to find a perfect hashing function for a given $t_{\max}$ and P_e should not exceed the cost of following some other, easily computable, rehashing policy for a higher value of $t_{\max}$. The cost of a trial is quite small. A trial involves choosing a hashing function at random (for the moment assumed to be not too costly) and evaluating at most n hash addresses. If the value of n is around a few hundred and $t_{\max}$ at most 20, the cost of finding the optimal rehashing policy by solving the optimization problem will be much higher than the cost of a trial. To be useful, the procedure for finding a rehashing policy has to be extremely simple and cheap. This implies that we will eventually have to resort to some simple heuristic procedure for finding good rehashing policies. However, we want to compare the performance of the proposed external perfect hashing scheme under the optimal rehashing policy with that under policies obtained by heuristic procedures. It is with this objective in mind that we proceed to discuss the solution procedure for the optimization problem.

4.4.1. Solution of the optimization problem

The constraint (4.9), which appears nonlinear, can easily be converted to a linear constraint by taking the logarithm of both sides. All the constants are rounded to integers after multiplying both sides by a large negative integer scale factor(SF).‡ Then (4.9) may be written as

$$a_1t_1 + a_2t_2 + \cdots + a_rt_r \geq hlimit$$

where $hlimit = \text{round } (SF * \log(1-P_s))$

$a_i = \text{round } (SF * \log q_i)$ for $1 \leq i \leq r$.

The optimization problem may now be written as

minimize $E(m)$, where

$$E(m) = m_l(1 - q_1^{t_1}) + (m_l + 1)q_1^{t_1}(1 - q_2^{t_2}) + \cdots + m_h q_1^{t_1} \cdots q_{r-1}^{t_{r-1}}. \quad (4.10)$$

subject to the conditions

$$t_1 + t_2 + \cdots + t_r = t_{\max} \quad (4.11)$$

$$a_1t_1 + a_2t_2 + \cdots + a_rt_r \geq hlimit. \quad (4.12)$$

In general, integer programming problems are difficult to solve and problems encountered in practice are often NP-complete. Solution procedures for linear integer programming problems normally involve solving linear optimization problems treating integer variables as continuous. The solution so obtained is then rounded to the nearest integer and the result is an approximate solution for the given problem. Such an approximate solution may be far from the optimal solution and it may even be infeasible. The solution of the linear optimization (continuous variables) problem can be used to split a given problem into two subproblems with tighter bounds, each of which may be further split and so on. This solution technique is known as branch and bound and basically it enumerates a small fraction of all the possible solutions for the given problem. Another method of enumerating a fraction of the feasible solutions is the dynamic programming technique. This technique is not particularly efficient for most integer programming problems [Section 18.2, HL80].

Solving a nonlinear optimization problem is difficult in general. Hence, the methods mentioned above to solve linear integer programming problems cannot easily be adapted to solve nonlinear integer programming problems. There has been much less progress in developing algorithms for nonlinear integer programming [Section 18.6, HL80].

---

‡ This simplifies the solution procedure. However the magnitude of SF should be sufficiently large so that the errors introduced are negligible. $SF = -10{,}000$ was used in all the computations reported in this thesis.

In principle, dynamic programming does not distinguish between linear and nonlinear objective functions. Exploiting the special structure of the problem at hand, we have arrived at a method based on dynamic programming which gives an approximate solution that is close to the optimal solution. The cost of the solution procedure is dependent on the required proximity of the obtained solution to the optimal solution.

The optimal partitioning of t_{max} is basically a multistep allocation process. We first outline briefly, the basic features which characterize multistep allocation problems suitable for solution by dynamic programming [Section 7.2, HL80]. In parallel we describe how our problem fits or does not fit the requirements of the standard approach.

Characterization of multistep allocation problems suitable for solution by dynamic programming:

1. The problem can be divided into **stages**, with a policy decision required at each stage. Our problem consists of r stages, each stage corresponding to trials with a certain number of pages. The number of trials to be allocated to a particular stage is the policy decision required at that stage.
2. Each stage has a number of **states** associated with it. Here we face the main difficulty in adopting the standard dynamic programming approach to our problem. If we were to minimize the objective function taking into account constraint (4.11) only, then we would have a classical dynamic programming problem. The states associated with each step would have been the number of trials (out of a total of t_{max}) left for allocation to the current and further stages. The number of states for each stage would then be restricted to t_{max}. However, to take into account constraint (4.12), the state associated with a stage must include the amount contributed to the left-hand side of constraint (4.12) in addition to the *resources* (number of trials) used in further† stages. In other words, the state variable at any stage needs to be a tuple $< \tau, h_l >$, where τ is the number of trials available for distribution to further stages and h_l is the corresponding amount of the left-hand side of (4.12) to be realized from further stages in allocating τ trials. τ is bounded by t_{max} and h_l by *hlimit*, and the number of possible states at each stage is $t_{max} * hlimit$. This is a very large number to handle at each stage as we shall see later.
3. Given the current state, an optimal policy for the remaining stages is independent of the policy adapted in previous stages. In other words, the knowledge of the current state of the system conveys all the information about previous stages necessary for determining the optimal policy henceforth. This property is referred to as the principle of optimality. It is obvious, by (4.16) or from the expression for $E(m)$ given below, that the

† We need to clarify the use of *further* and *previous* in this context. The trial-and-error process starts at stage 1 and proceeds with stages 2,3, $\cdots, i, \cdots, r$. The dynamic programming solution procedure starts at stage r and proceeds with stages $r-1, \cdots, i, \cdots, 1$. At stage i, *further* refers to stages $i+1, \cdots, r$.

optimization problem satisfies this condition.

4. There exists a recursive relationship that identifies the optimal policy for each state at stage i, given the optimal policy for each state at stage $(i+1)$.

The recursive relationship can be identified by writing the objective function (4.10) in the following form:

$$E(m) = m_l(1 - q_1^{t_1}) + q_1^{t_1}((m_l+1)(1 - q_2^{t_2}) + q_2^{t_2}((m_l+2)(1 - q_3^{t_3}) + \cdots + m_h)) \cdots).$$

Let $f_1, f_2, \cdots f_i, \cdots f_r$ denote functions of 3 variables: $f_i(\tau, h_l, t_i)$. τ denotes the number of trials available for distribution to stages $i, i+1, \cdots, r$, $0 \leq \tau \leq t_{\max}$; h_l is the corresponding amount of the left-hand side of (4.12) realized, $0 \leq h_l \leq hlimit$; t_i is the number of trials allocated to stage i, $0 \leq t_i \leq \tau$. Let f_i^*, denote the minimum value of f_i in the following sense:

$$f_i^*(\tau, h_l) = \min_{0 \leq t_i \leq \tau} f_i(\tau, h_l, t_i). \tag{4.13}$$

We now identify the following recurrence relations:

$$f_r(\tau, h_l, t_r) = m_h \tag{4.14}$$

$$f_r^*(\tau, h_l) = m_h \tag{4.15}$$

$$f_i(\tau, h_l, t_i) = (m_l + i - 1)(1 - q_i^{t_i}) + q_i^{t_i} f^*_{i+1}(\tau - t_i, h_l - a_i t_i) \tag{4.16}$$

$$f_i^*(\tau, h_l) = \min_{0 \leq t_i \leq \tau} f_i(\tau, h_l, t_i), \qquad 1 \leq i < r.$$

The required minimum value of $E(m)$ can then be written as

$$f_1(\tau, h_l, t_1) = m_l(1 - q_1^{t_1}) + q_1^{t_1} f_2^*(\tau - t_1, h_l - a_1 t_1)$$

$$\text{minimum } E(m) = \min_{h_l \geq hlimit} f_1^*(t_{\max}, h_l)$$

Outline of the solution procedure

After identifying the states and the recurrence relations, the solution procedure is straightforward. The computation starts at the last stage by finding $f_r^*(\tau, h_l)$ for all possible states $<\tau, h_l>$ and the results are tabulated.

$$f_r^*(\tau, h_l) = \min_{t_r} f_r(\tau, h_l, t_r) = \min_{t_r}(m_h) = m_h$$

$<\tau, h_l>$	f_r^*	t_r^*
$<1, a_r>$	m_h	1
$<2, 2*a_r>$	m_h	2
...	...	...

The procedure continues by computing similar tables for stages $(r-1)$, $(r-2)$ etc. At stage i, $1 \leq i < r$, we have the following recurrence relation:

$$f_i(\tau, h_l, t_i) = (m_l + i - 1)(1 - q_i^{t_i}) + q_i^{t_i} f^*_{i+1}(\tau - t_i, h_l - a_i t_i)$$

$$f_i^*(\tau, h_l) = \min_{0 \le t_i \le \tau} f_i(\tau, h_l, t_i), \qquad 1 \le i < r.$$

Our goal is to construct a table of $f_i^*(\tau, h_l)$ for each possible state $< \tau, h_l >$. We have a table of f^*_{i+1} available from the $(i+1)$st stage. This involves optimizing the value of f_i for each of the states $< \tau, h_l >$, which can be accomplished by enumerating the value of f_i for all values of t_i, $0 \le t_i \le t_{max}$. At each stage the results are tabulated as follows:

$< \tau, h_l >$	f_i^*	t_i^*
$<1, \cdots>$	...	...
$<1, \cdots>$	...	...
$<..., \cdots>$	...	...
$<2, \cdots>$	...	...
$<..., \cdots>$	...	...
$<t_{max}, \cdots>$	...	...
$<..., \cdots>$	...	...
$<t_{max}, \cdots>$	...	...

The solution procedure ends with the computation of a table for f_1^*, from which the required minimum value of $E(m)$ can be obtained for the state $< t_{max}, hlimit >$.

Difficulty with using the basic solution procedure

The procedure outlined above is straightforward in principle but computationally expensive, because of the large number of states at each stage. The states are represented as tuples of the form $< \tau, h_l >$ where $0 \le \tau \le t_{max}$ and $0 \le h_l \le hlimit$, hence there are $t_{max} * hlimit$ different states at each stage. The scale factor SF used to obtain (4.12) from (4.9) has to be sufficiently large to limit the errors introduced during the conversion process. The value of *hlimit* will be at least several hundred, and hence the number of states may be several thousand at each stage. Not all the states are meaningful at a given stage: at the beginning of the computation, for stage r there are only t_{max} possible states. At the next stage, $(r-1)$st, $O(t^2_{max})$ states are possible and so on. The number of states grows exponentially. At stage i, we need to store $O(t_{max}^{r-i+1})$ states (of course this is bounded by $t_{max} * hlimit$). The storage space required and the computational cost at each stage are proportional to the number of states at each stage. We also need to store the intermediate partial policy $(t_i^*, t_{i+1}^*, \ldots, t_r^*)$ associated with each state of stage i. (The alternative is to store only t_i^* at stage i and retain all the tables up to stage r.) The large number of states is a serious problem from a computational point of view and using standard dynamic programming to find the optimal solution does not seem feasible. However, after various experiments aimed at understanding the structure of the problem we arrived at a simple modification to the basic solution procedure which enables us to obtain approximate solutions to the optimization problem. There is some degree of control over the accuracy and the cost depends on the accuracy required. The modification involves a heuristic to reduce the number of states at each stage. It should be emphasized that the heuristic is based on the particular structure of the problem and is not generally applicable to nonlinear integer

optimization problems.

4.4.2. A heuristic for approximate solution of the optimization problem

The computational cost of dynamic programming depends strongly on the structure and parameters of the problem. Before continuing, we present a typical problem.

Example:

Number of keys $(n) = 180$
Page size $(b) = 20$
Maximum number of trials $(t_{\max}) = 10$
Probability of success required in $t_{\max}$ trials $(P_s) = 0.99$
Minimum number of pages $m_l = 180/20 = 9$
Upper bound on number of pages $m_h \approx 180/(.5*20) \approx 17$
Number of stages $(r) = 17 - 9 + 1 = 9$
Chosen scale factor $(SF) = -10000$
$hlimit = -10000 * \log(1 - .99) \quad = 46052$

The values $q_i = 1 - P(180, 8 + i, 20)$ can be computed using the procedure presented in chapter 4. The results are tabulated below. Note that $a_i = SF * \log q_i$.

i	m	$P(n, m, b)$	q_i	a_i
1	9	.000	1.00	0
2	10	.005	.995	52
3	11	.100	.900	1046
4	12	.331	.669	4023
5	13	.578	.422	8629
6	14	.758	.242	14185
7	15	.867	.133	20223
8	16	.930	.070	26476
9	17	.963	.037	32797

Table 4.4.1 Values of the constants

We wish to determine $(t_1, t_2, \ldots, t_9)$ so as to minimize f_1 where,

$$f_1 = 9(1 - 1^{t_1}) + 10\times 1^{t_1}\times(1 - .995^{t_2}) + \cdots . 17\times 1^{t_1} \cdots \times 0.070^{t_8} \quad (4.17)$$

Subject to the constraints $t_1 + t_2 + \cdots + t_9 = 10$ (4.18)

$$0t_1 + 52t_2 + \cdots + 32797\, t_9 \geq 46052. \quad (4.19)$$

There are a large number of states $< \tau, h_l >$ possible for each value of τ, $0 \leq \tau \leq t_{\max}$. Most of these states are not realizable. For example, at stage r only $t_{\max}$ states of the form $< t_r, a_r * t_r >$, $0 \leq t_r \leq t_{\max}$ are meaningful i.e., only one state for each value of τ. The main idea behind the heuristic is to reduce the number of states at each stage of the computation. The problem is, how to decide which states to retain? The following heuristic has experimentally been found to perform satisfactorily.

Proposed Heuristic

At stage i, store only a few of the states for each value of τ. Out of the possible h_l states, the states corresponding to the best few values of f_i^* are chosen to be retained. Let "depth" denote the number of states retained for each value of τ.

As a consequence of the above heuristic, we will not be able to compute $f_i^*(\tau, h_l)$ for every required state $< \tau, h_l >$ because we do not have the values of f^*_{i+1} for all states. The recurrence relations (4.13) and (4.16) have to be rewritten as follows, so that f_i is computed only for those states for which the corresponding f^*_{i+1} is available:

$$f_i(\tau, h_l + a_i t_i, t_i) = (m_l + i - 1)(1 - q_i^{t_i}) + q_i^{t_i} f^*_{i+1}(\tau - t_i, h_l) \tag{4.20}$$

$$f_i^*(\tau, h_l + a_i t_i) = \min_{0 \le t_i \le \tau} f_i(\tau, h_l + a_i t_i, t_i). \tag{4.21}$$

The above equations enable us to compute f_i^* for a limited number of states as dictated by the availability of f^*_{i+1} from the table computed at stage $(i+1)$. Out of the values for f_i^* so computed the best *depth* values, and their corresponding states are kept. The table constructed at stage i, with $depth = d$, has the following format:

$< \tau, h_l >$	f_i^*	t_i^*
$< 1, h_{11} >$	...	...
$< 1, h_{12} >$	...	...
...	...	
$< 1, h_{1d} >$	..	
$< 2, h_{21} >$		
...		
$< 2, h_{2d} >$		
...		
$< t_{max}, h_{maxd} >$		

Because we do not have explicit control over the states for which we are computing f_i^*, it is possible that there may be no feasible solution by the time we reach the first stage (f_1). This problem may be overcome as follows. At any stage i, there are *depth* number of states $< \tau, h_l >$ with $\tau = t_{max}$. All these states correspond to allocating all the t_{max} trials to stages $i, i+1, \ldots, r$ and none to stages $1, 2, \ldots, i-1$. So all the states $< t_{max}, h_l >$ should have $h_l > hlimit$ for the states to be useful in subsequent stages. If this rule is enforced starting at stage r we are guaranteed at least *depth* number of feasible solutions at stage 1.

Appendix A shows a few steps of the solution procedure for the problem given by (4.17), (4.18) and (4.19).

4.4.3. Performance of the heuristic solution procedure

When using the heuristic described in the previous section, it is desirable from the point of view of accuracy to have *depth* as large as possible. However, the complexity of the computation at each stage is $O(t^2_{max} * depth)$ and hence *depth* should be small to keep the computational cost small. Reducing *depth* to 1 corresponds to ignoring constraint (4.19) completely. A number of experiments were performed to study the effect of the value of *depth* on the accuracy of the solution obtained. Each experiment consisted of finding the approximate optimal rehashing policy for given values of page size (b), number of keys (n), and *depth* using the heuristic solution procedure. The true optimal rehashing policy was computed by exhaustive enumeration and the two results were compared. The experiments were repeated with *depth* = 1, 3 and 5, and for n varying up to a certain value *Nmax*, which varied with the page size b. The results of these experiments are tabulated in table 4.4.2. All the experiments were performed with $t_{max} = 10$ and $P_e = 0.99$.

depth	b	10	20	30	40	50
	Nmax	80	160	285	400	500
depth = 1	No. of errors	30	11	7	2	0
	max error	1.34	.35	.07	.007	
depth = 3	No. of errors	5	4	2	0	0
	max error	.093	.17	.028		
depth = 5	No. of errors	3	0	0	0	0
	max error	.063				

Table 4.4.2 Performance of the heuristic solution procedure

In table 4.4.2, "No. of errors" corresponds to the number of cases in which the heuristic solution procedure failed to yield the optimal rehashing policy (as determined by exhaustive searching); "max error" is the maximum difference (expressed as percentage) between the optimal value of $E(m)$ and the obtained solution.

The results in column 3 correspond to a page size of 10. Experiments were performed for n varying from 11, $(b+1)$, to 80. Out of these 70 cases, only 30 failed to give the optimal solution with *depth* = 1 and the rest of the cases did result in optimal policies. Out of these 30 cases, the maximum error in the resulting $E(m)$ was only 1.34 percent. When *depth* was increased to 5, there were only 3 cases yielding non optimal solutions and the maximum error dropped to 0.063 percent.

As the page size was increased, *Nmax* was also increased to maintain the same number of stages(approximately) in the optimization problem. In spite of a larger number of experiments, the number of nonoptimal solutions is reduced as b is increased. This should not be interpreted as resulting from some relationship between the heuristic procedure and the page size . The reason is that $P(n, m, b)$ increases as b increases. With $b = 10$, all the 30 non-optimal solutions were obtained for $48 \leq n \leq 80$. All the solutions obtained by the heuristic procedure were optimal for $n \leq 48$. It may be noted that $P(n, m, b)$ is quite high for $n < 50$ even when $b = 10$. This leads us to conclude that the heuristic

procedure performs well even for low values of *depth* when $P(n, m, b)$ is sufficiently high.

If constraint (4.19) is eliminated, the heuristic with *depth* = 1 would always give the optimal solution. When the q_i's are low (i.e., $P(n, m, b)$ are high) (4.19) poses no real constraint to the optimization problem and hence the heuristic procedure works well.

The main observation drawn from table 4.4.2 is that for any b, as *depth* increases the performance of the heuristic improves and that even with *depth* = 1 the solution obtained is not far from the optimal solution. With *depth* = 3 the performance of the heuristic solution procedure seems quite satisfactory. Hence, *depth* = 3 was used to compute most of the results presented in this thesis.

For the example problem given by (4.17), (4.18) and (4.19), the policy obtained by the heuristic procedure with *depth* = 3 is (0,0,4,4,1,0,1,0,0) corresponding to an expected number of pages of 11.9152. The exact optimal solution is (0,0,3,5,1,1,0,0,0) corresponding to an expected number of pages of 11.8993. The error in the solution is only 0.134% (which translates into an error of 0.101 percentage points in the load factor).

4.5. Conclusions

The analysis presented in the previous chapter showed a sharp tradeoff between the cost of finding a perfect hashing function and the load factor. There are many possible ways of balancing this tradeoff. In this chapter we explained the difficulty of choosing the number of pages while rehashing a group in practice. We suggested distributing the trials over different load factors and defined an optimal rehashing policy which attempts to limit the number of trials required to find a perfect hashing function. Computation of the optimal rehashing policy involves the solution of a nonlinear integer programming problem. In general, such problems are very difficult to solve and we therefore proposed a heuristic solution procedure based on dynamic programming. We presented results of several experiments performed to study the performance of the heuristic. They indicate that solutions very close to the optimal solutions can be obtained at a reasonable expense.

Even so, it is impractical to solve an optimization problem every time a rehash is to be performed. In chapter 7 we propose a very simple heuristic to compute rehashing policies. The results of this chapter are useful for comparing the performance of this and other nonoptimal policies with that of the optimal policy.

Chapter 5

Performance under the optimal rehashing policy

5.0. Chapter overview

In the previous chapter we defined an optimal rehashing policy which attempts to limit the cost of finding perfect hashing functions. In this chapter we study the performance of the proposed external perfect hashing scheme under this policy. The load factor of a group under the optimal rehashing policy is computed as a function of the number of records in the group, and the load factor of a file is computed as a function of the average group size. Numerical results are given for various parameter combinations. After a discussion of the results obtained, we go on to consider the cost of perfect hashing. The two main cost components are: cost of an insertion and header table space. Insertion costs are affected by the probability of rehashing and the actual cost of performing a rehash (internal processing and disk I/O). The space requirements of the header table are estimated for a file of one million records assuming that the header table is organized as a hash table.

5.1. Load factor under the optimal rehashing policy

In this section we study the load factor (storage utilization) that can be achieved using the proposed external perfect hashing scheme. We are mainly interested in the load factor of the overall file. To compute the overall load factor we must first consider the load factor of a group as a function of the number of records in the group.

Consider a file organized using the external perfect hashing scheme. The file may be built in one of two ways:

1. Making one insertion at a time starting from an empty file. We call such a file an **incrementally** built file.
2. If all the records of the file are available, they are partitioned into groups and individual groups are stored by perfect hashing. We shall call this operation **initial loading** of the file.

5.1.1. Load factor of a group built incrementally

Consider an arbitrary but fixed group of an incrementally built file. Initially, when the number of records in the group is less than or equal to b, the group requires only one page to store the records. When the $(b + 1)$st record is inserted, a rehash becomes necessary. The group size will grow to two pages (possibly 3 depending on the values of b and t_{max}). In general, when inserting the $(n + 1)$st record into a group already containing n records, one of two events will occur: the record fits into the page to which it is assigned by the current perfect hashing function, or the page overflows if it already contains b records. In the first case the insertion is simple: read in the page, insert the record somewhere on the page and write it back. In the second case, a new perfect hashing function must be found for the $(n + 1)$ keys of the group and all the records must be redistributed using the new perfect hashing function.

In this section we compute the load factor of a group as a function of the number of records n. Numerical results are presented for the case when the optimal† distribution of trials is determined and followed whenever rehashing is required.

Figure 5.1.1 illustrates the state transitions caused by the insertion of a record into a group. When there are b keys in the group, and the page size is b they all fit on one page. Hence, with probability 1.0 the group consists of one page. (represented by the first row in the figure). When the $(b + 1)$st key is inserted, rehashing is necessary. There is no possibility of the group size remaining at one page, and with a very high probability the group size ends up being 2 pages. There is a small, but nonzero, probability that the group size will increase to 3 pages. In general, after n keys have been inserted into the group, the state of the group is represented by the 3rd row of Figure 5.1.1.

---

† All the numerical results reported in this thesis use the approximate solution of the optimization problem obtained using the procedure described in section 4.4 (with $depth = 3$).

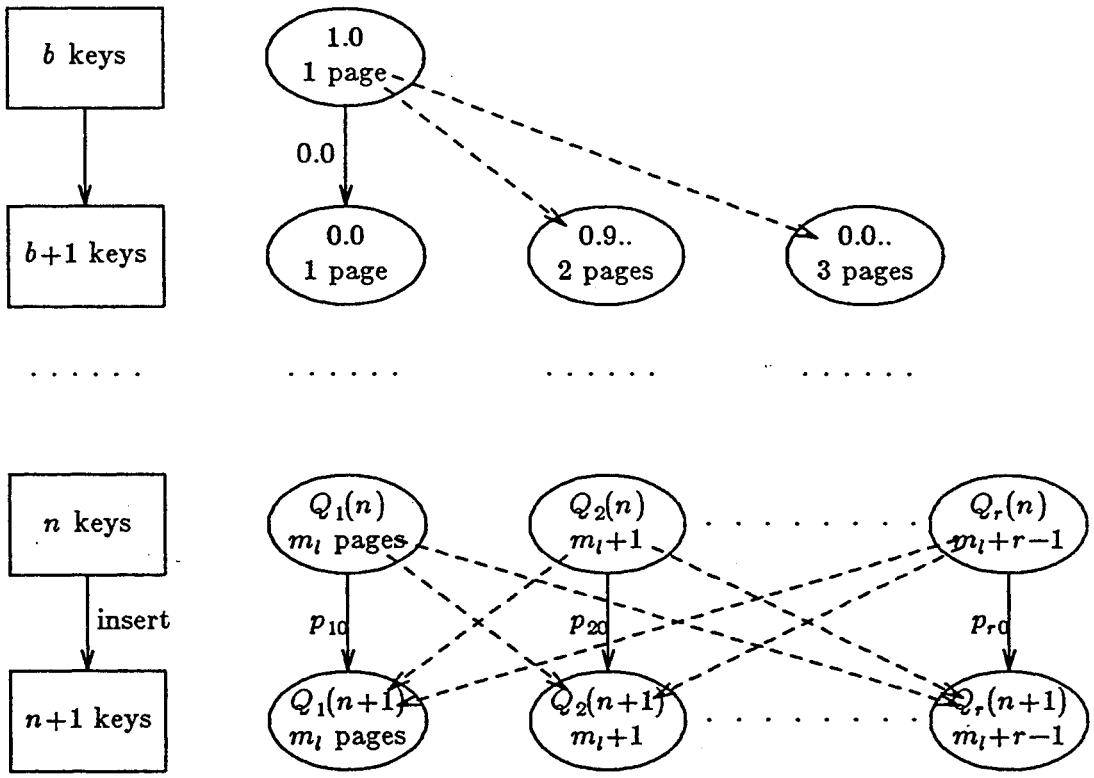


Figure 5.1.1 State transitions caused by an insertion into a group

The following notation is used:

m_l denotes the lower bound on number of pages

m_h denotes the upper bound on number of pages

$r =$ $m_h - m_l + 1$ is the number of different stages possible for trials

Although m_l, m_h *and* r are dependent on n, to simplify the notation we will not make it explicit. If the insertion of the $(n+1)$st record causes rehashing, m_l is chosen as $\lceil (n+1)/b \rceil$. An upper bound for m_h is such that $P(n,m_h,b) > P_s$, i.e., the probability of success of a single trial with m_h pages is greater than the required probability of success in $t_{\max}$ trials. For the range of n and b we are interested in, it is sufficient to choose m_h corresponding to a load factor of 50%, $m_h \approx (n+1)/(0.5*b)$.

$Q_i(n)$ denotes the probability of having $(m_l + i - 1)$ pages in a group with n keys. $Q_i(n) = 0$ *for* $i > r$

p_{i0} denotes the probability of a perfect hashing function for n keys and $(m_l + i - 1)$ pages remaining perfect for $(n + 1)$ keys

Given $Q_i(n)$ for $1 \leq i \leq r$, the aim is to determine the corresponding $Q_i(n+1)$. These can be computed as follows. Let $RH(n)$ be the probability of the $(n + 1)$st insertion causing a rehash. Then

$$p_{i0} = \frac{P(n+1,\, m_l + i - 1,\, b)}{P(n,\, m_l + i - 1,\, b)}, \tag{5.1}$$

$$RH(n) = 1 - \sum_{k=1}^{r} Q_k(n)\, p_{k0}. \tag{5.2}$$

If rehashing is necessary when inserting the $(n+1)$st record, an optimization problem represented by equations (4.10), (4.11) and (4.12) is solved with the following parameters (using specified values of $t_{\max}$ and P_s):

$$m_l = \lceil (n+1)/b \rceil\ ,\quad m_h \approx (n+1)/(0.5*b),\quad r = m_h - m_l + 1$$

$$q_i = 1 - P(n+1,\, m_l + i - 1,\, b) \quad \text{for } i = 1 \cdots r.$$

Let $(t_1, t_2, \ldots, t_r)$ be a rehashing policy (we are actually interested in the optimal rehashing policy, as defined in the previous chapter). Note that this means trying up to t_1 hashing functions with m_l pages. If not successful, try up to t_2 functions with $m_l + 1$ pages and so on. Finally, keep trying with m_h pages until success.

Let $R_i(n+1)$ denote the probability that the above rehashing procedure results in $(m_l + i - 1)$ pages. $R_i(n+1)$ can be computed as

$$R_i(n+1) = {q_1}^{t_1} {q_2}^{t_2} \cdots {q_{i-1}}^{t_{i-1}} (1 - {q_i}^{t_i}), \qquad \text{for } i = 1 \cdots (r-1)\ ,$$

$$R_r(n+1) = 1 - \sum_{i=1}^{r-1} R_i(n+1). \tag{5.3}$$

We can now give an expression for $Q_i(n+1)$. Recall that $Q_i(n+1)$ is the probability of having $(m_l + i - 1)$ pages after the $(n+1)$st key has been successfully inserted into the group. There are two ways a group can reach a state with $(m_l + i - 1)$ pages when the $(n+1)$st key is inserted: i) there were $(m_l + i - 1)$ pages in the group with n keys and the insertion did not cause rehashing and ii) the insertion caused a rehash which resulted in $(m_l + i - 1)$ pages. Hence we have

$$Q_i(n+1) = p_{i0} Q_i(n) + RH(n)\, R_i(n+1), \quad i = 1 \cdots (r-1)\ ,$$

$$Q_r(n+1) = 1 - \sum_{i=1}^{r-1} Q_k(n+1)\ . \tag{5.4}$$

This recurrence relation can be used to compute $Q_i(n)$ for any value of n, starting from $n = b$, $Q_1(b) = 1.0$ and $Q_i(b) = 0$ for $i > 1$. The expected number of pages in a group with $(n+1)$ keys, $E_{gnp}(n+1)$, and the corresponding load factor† of the group $E_{glf}(n+1)$ are then given by

† The load factor we are referring to is the average weighted load factor computed as (number of records) / b * (expected number of pages). For brevity, we simply use the terms "load factor".

$$E_{gnp}(n+1) = \sum_{i=1}^{r} (m_l + i - 1)\, Q_i(n+1) \tag{5.5}$$

$$E_{glf}(n+1) = \frac{n+1}{b\, E_{gnp}(n+1)}. \tag{5.6}$$

Table 5.1.1 illustrates the results of one step of the calculations. The expected number of pages in a group with 180 records is 11.629 (the corresponding load factor is 77.39%) and the probability of a rehash when inserting the 180th record is 0.094.

$n = 179, b = 20, t_{\max} = 10, P_s = 0.99$						
i	No. of pages $m_l + i - 1$	$Q_i(n)$	p_{i0}	policy	$R_i(n+1)$	$Q_i(n+1)$
1	9	.000	.111	0	.000	.000
2	10	.047	.750	0	.000	.035
3	11	.573	.886	4	.342	.540
4	12	.264	.943	4	.526	.298
5	13	.057	.970	1	.076	.062
6	14	.028	.984	0	.000	.028
7	15	.022	.991	1	.048	.026
8	16	.008	.995	0	.000	.008
9	17	.002	.997	0	.007	.002

Table 5.1.1 Results of one step of the calculations

Figure 5.1.2 shows the load factor of a group (computed using (5.6)) plotted as a function of the number of records in the group. The results were computed using the procedure described above. The oscillations at the beginning are due to fragmentation. When there are exactly b records in the group, one page is sufficient to store the records corresponding to 100% load factor. However, when there are $b + 1$ records in the group, at least 2 pages are needed yielding a load factor just over 50%. Similar situations develop when the number of records in the group is around $2b$, $3b$, . . . As the number of records increases, the effect of fragmentation diminishes and the oscillations die out. We will see later that fragmentation imposes a lower bound on the group size for achieving higher load factors.

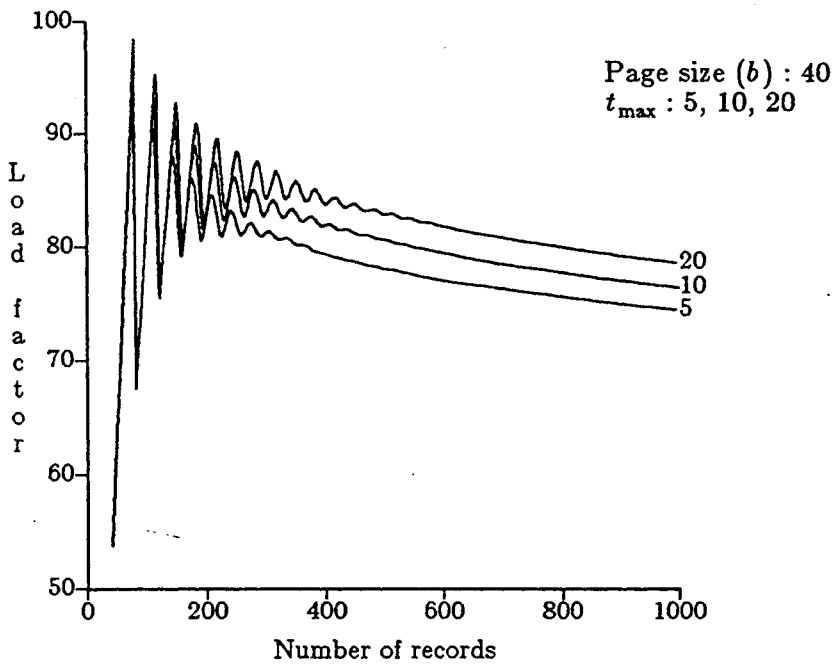


Fig. 5.1.2 Load factor of an incrementally built group

As the number of records in a group increases, the load factor of the group decreases slowly. Because the maximum number of trials is kept constant, this is to be expected. It is a result of the fact that $P(n, m, b)$ decreases with increasing m as explained in section 3.4. Note that the discussion in chapter 3 dealt with load factors and probability of success of a trial. The situation here is somewhat different: the maximum number of trials is fixed and the trials are distributed according to the optimal rehashing policy.

Increasing t_{max} increases the load factor; every doubling of t_{max} seems to increase the load factor by a constant amount. This holds for a certain range of t_{max}, but for increasingly larger values of t_{max} the effect of a doubling t_{max} gradually diminishes. This is again due to the behavior of $P(n, m, b)$ at very high load factors.

5.1.2. Load factor of a group built by initial loading

For an incrementally built group, the load factor(computed as $n/bE(m)$) is not only dependent on the rehashing policy but also on the state of the group . This may be regarded as a certain kind of memory, or inertia, in the system. In the case of a group built by initial loading, only rehashing (more specifically, the policy of finding perfect hashing functions) determines the load factor. For this case, all the equations obtained in the last section are applicable with the modification that p_{i0} should be set to zero in equation (5.2) and (5.4).

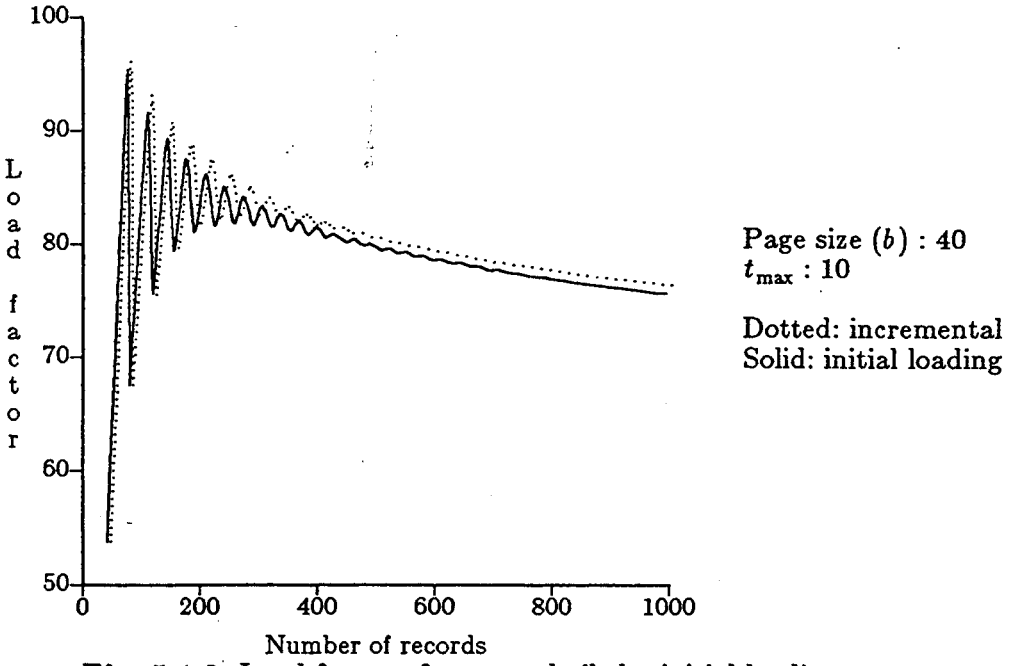


Fig. 5.1.3 Load factor of a group built by initial loading

In Figure 5.1.3 the solid line is a plot of the load factor of a group built by initial loading as a function of the number of records in the group. The page size is 40 and t_{max} is 10. The dashed line corresponds to the load factor of an incrementally built group, drawn for comparison purposes. The solid line lies below the dashed line because in an incrementally built group the load factor increases, when an insertion does not cause rehashing. If an insertion does cause rehashing, the resulting load factor is the same as that for a group built by initial loading. Hence the load factor of an incrementally built group is never less than that achieved by initial loading, and on average, slightly higher.

5.1.3. Load factor of a file

Given the distribution of the expected number of pages of a group for different group sizes, the overall load factor of a file depends on the distribution of the number of records in a group. This in turn depends on the way in which the header table is organized. The expected number of pages per group in a file, E_{fgp}, is given by the summation

$$E_{fgp} = \sum_{n=0}^{\infty} PGR(n)\, E_{gnp}(n) \qquad (5.7)$$

where $PGR(n)$ is the probability that a group in the file consists of n records, and $E_{gnp}(n)$ is the expected number of pages in a group having n records.

$PGR(n)$ depends on the header table organization. For the case when the header table is organized as a hash table, these probabilities can easily be computed. Let λ be the expected number of records per group in a file whose header table is organized as a hash table. Using the Poisson approximation of the binomial distribution, $PGR(n)$ is given by

$$PGR(n) = \frac{e^{-\lambda}\,\lambda^n}{n!}. \tag{5.8}$$

The expected number of pages per group in the file is then dependent on λ and not on the total number of records in the file and is given by

$$E_{fgp}(\lambda) = \sum_{n=0}^{\infty} \frac{e^{-\lambda}\,\lambda^n}{n!}\, E_{gnp}(n), \tag{5.9}$$

and the corresponding load factor of the file by

$$E_{flf}(\lambda) = \frac{\lambda}{bE_{fgp}(\lambda)}. \tag{5.9a}$$

Figure 5.1.4 is a plot of the load factor of an incrementally built file computed using (5.9a) and the results obtained in the last section. $t_{\max}$ is 20 in all the cases and there is one curve plotted for each value of b (10, 20, 30, 40, 50). The effect of the page size on the load factor of a file is very significant. Clearly, it is desirable to have as large a page size as possible for a given group size. Page sizes below 20 seem impractical because of the resulting low load factor. It may be noted that for a given file, the size of the header table determines the average group size.

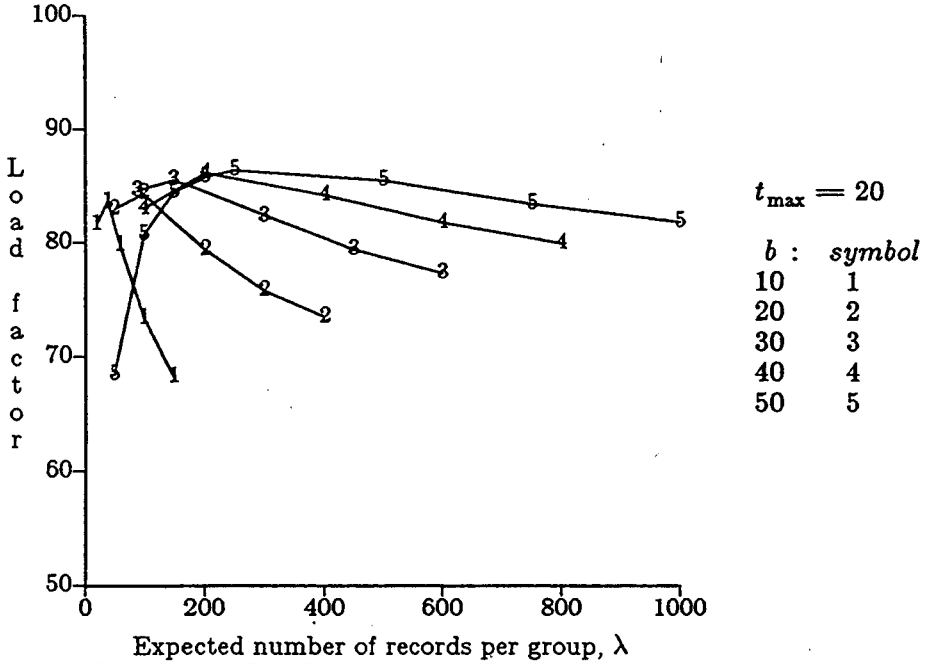


Fig. 5.1.4 Load factor of a file as a function of average group size

For a given page size, the load factor falls slowly as the group size increases. This is a direct consequence of the decreasing group load factor with increasing number of records in the group. The slowly decreasing load factor gives a significant advantage to the external perfect hashing scheme in practice. Suppose a file is designed with an expected group size of 500 records, page size of 40 and hence a load factor of 82%. If the file size shrinks to half its original size, the average group size falls to 250 and the load factor of the file improves to 85%. In most hashing schemes, shrinking files lead to wasted space and a

corresponding drop in the storage utilization, while on the other hand expanding files lead to deteriorating performance. In the above example, if the file size is doubled gradually, the group size gradually increases to 1000 records and the load factor falls to 80%.

As already noted in section 5.1.1 fragmentation is significant when the group size is small. Hence, the load factor of the file starts to fall when the group size decreases beyond a certain point (with $t_{max} = 20$, maximum load factor is achieved when $\lambda \approx 5b$). For a given page size, there is an upper limit on the load factor of the file for a given value of t_{max}. This upper limit is not very sensitive to the page size. This is because the fragmentation problem is more pronounced for larger pages and offsets the advantage of higher values of $P(n, m, b)$.

In summary, the load factors achievable by the proposed external perfect hashing scheme appear to be in a practically acceptable range. The scheme can also gracefully accommodate varying file sizes.

5.2. Cost of insertions

We have covered the performance and benefits of the external perfect hashing scheme in the last section. The advantages of the scheme compared with other hashing schemes are obtained at certain extra costs. In this section we study the costs associated with insertions: computational cost of finding perfect hashing functions and I/O cost for redistributing the records. Since deletions as such do not necessarily involve extra costs, we will not consider deletions here, and postpone further discussion to section 8.3.

5.2.1. Probability of an insertion causing a rehash

Insertion of a record may cause rehashing of a group. If the page assigned to the record by the current hashing function is already full, a new perfect hashing function must be found and the records of the group redistributed according to the new hashing function. The expected cost of an insertion depends on the relative frequency of rehashing and the cost of rearranging the records of a group.

The probability of the insertion of the $(n + 1)$st record into an incrementally built group causing a rehash was derived in section 5.1.1 and is given by

$$RH(n) = 1 - \sum_{i=1}^{r} Q_i(n)\, p_{i0}\,, \tag{5.2}$$

where $p_{i0} = \dfrac{P(n+1,\, m_l + i - 1,\, b)}{P(n,\, m_l + i - 1,\, b)}$ for $i = 1, \cdots, r$.

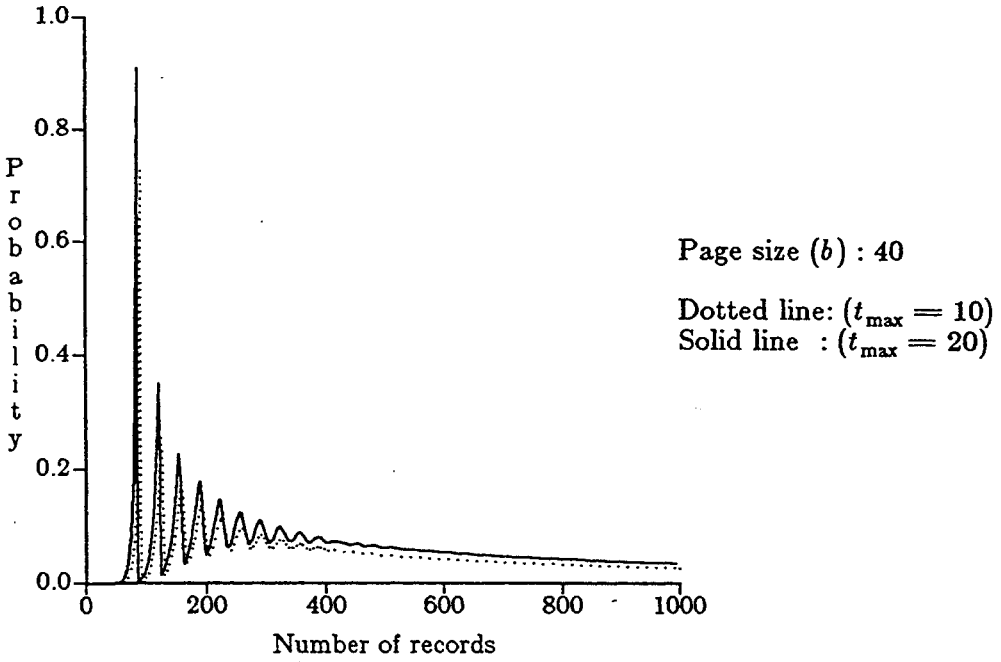


Fig. 5.2.1 Probability of an insertion causing rehashing of a group

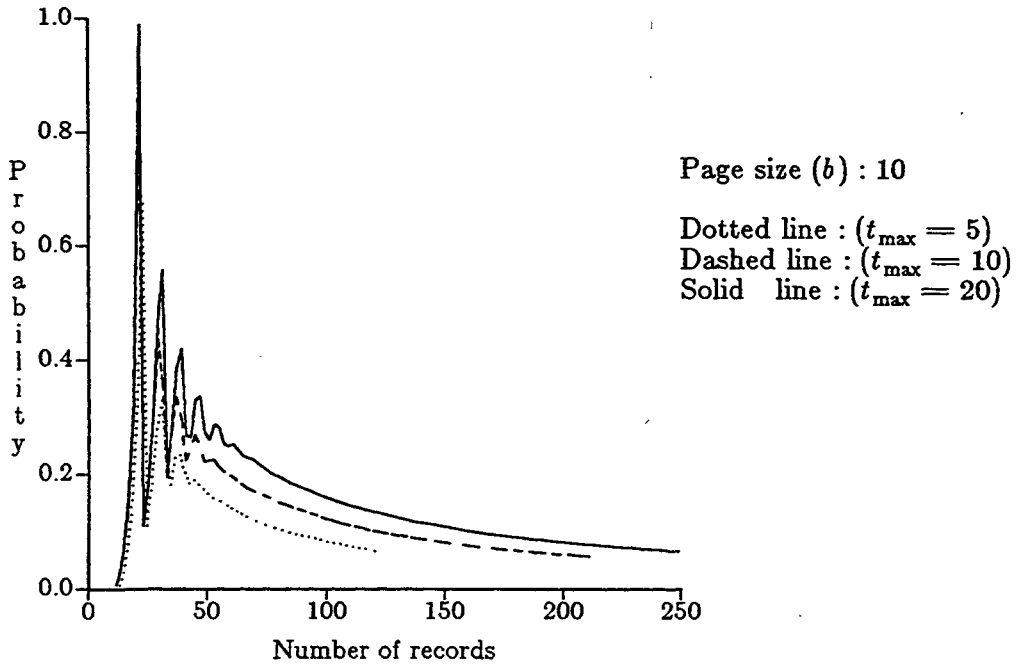


Fig. 5.2.2 Probability of an insertion causing rehashing of a group

Figures 5.2.1 and 5.2.2 are plots of the probability of an insertion causing a rehash as a function of the number of records in an incrementally built group. The oscillations at the beginning are due to fragmentation as already discussed in section 5.1.1. For practical ranges of the group size, the probability of rehashing is quite small. With $b = 40$ and $t_{max} = 10$, the probability of rehashing is only 0.0257 when the group size is 1000, i.e., approximately one in forty insertions triggers a rehash. The upper curve in figure 5.2.1 corresponds to $t_{max} = 20$ and

the lower one to $t_{max} = 10$, which appears counter intuitive. With $t_{max} = 20$, more work is spent on finding perfect hashing functions and why should this lead to doing so more often? The reason is that, on average, we find better hashing functions (higher load factor) with a higher value of t_{max}, and the probability of rehashing is higher when there are fewer non-full pages. This is well illustrated by figure 5.2.3 which is a plot of the probability of an insertion causing a rehash as a function of the load factor of the group. When t_{max} is increased, the load factor of a group increases which in turn increases the rehashing probability.

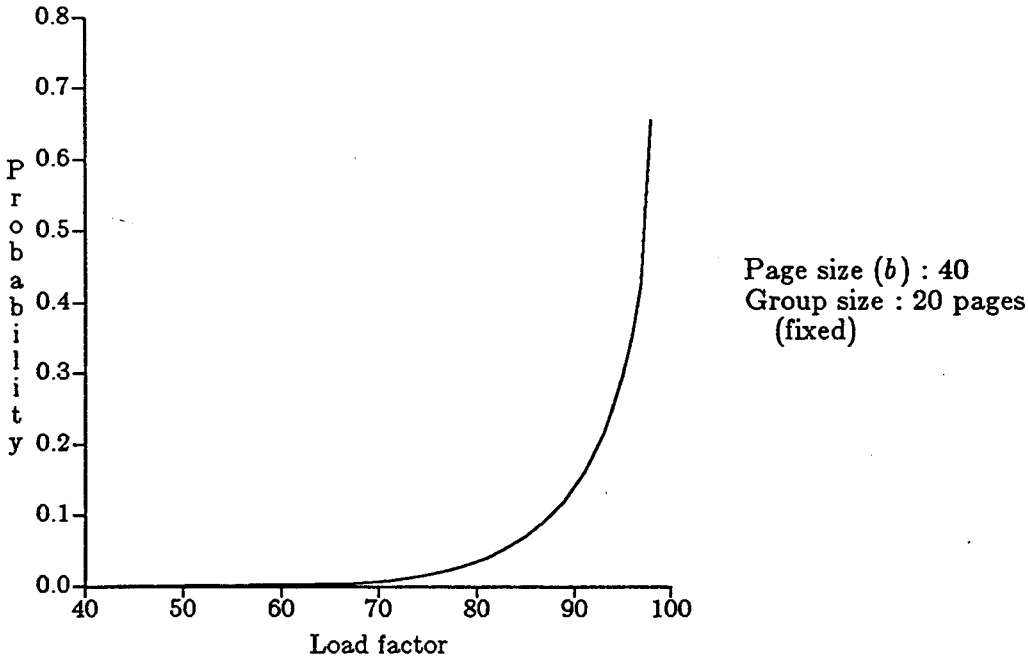


Fig. 5.2.3 Probability of an insertion causing rehashing

For a file whose header table has been organized as a hash table, the probability of an insertion into the file causing rehash of a group, *FRH*, is given by

$$FRH(\lambda) = \sum_{n=0}^{\infty} \frac{e^{-\lambda}\,\lambda^n}{n!} \cdot RH(n). \tag{5.10}$$

Figure 5.2.4 is a plot of the probability of an insertion into a file causing a rehash, plotted as a function of λ, the average group size. The page size is 40 and there is one curve for each t_{max} of 10 and 20. Because of the distribution of the group size in a file, the oscillations seen in Figure 5.2.1 are eliminated.

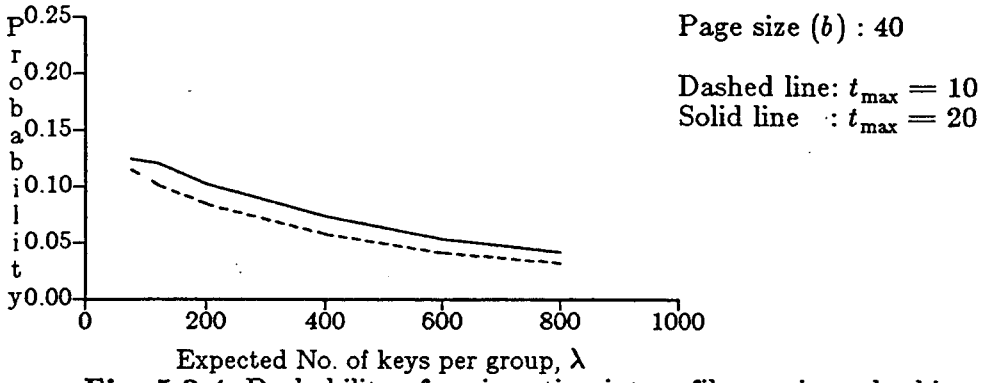


Fig. 5.2.4 Probability of an insertion into a file causing rehashing

5.2.2. Cost of rehashing

In the previous section we have seen that the probability of an insertion causing a rehash is quite small. However, should it become necessary to rehash, it is an expensive operation: all the pages of the group have to be read in, a new perfect hashing function found, and all the pages written back after redistributing the records. Thus rehashing involves considerable internal processing and I/O.

Expected number of trials

The optimal rehashing policy limits the cost of finding perfect hashing functions. Given a policy $(t_1, t_2, \ldots, t_r)$ the expected number of trials, $E(t)$, required to find a perfect hashing function is given by

$$\begin{aligned} E(t) = {} & 1(1-q_1) + 2q_1(1-q_1) + \cdots + t_1 {q_1}^{t_1-1}(1-q_1) \\ & + (t_1+1){q_1}^{t_1}(1-q_2) + \cdots + (t_1+t_2){q_1}^{t_1}{q_2}^{t_2-1}(1-q_2) \\ & + \cdots + (t_1+t_2+\cdots+t_r){q_1}^{t_1}{q_2}^{t_2}\cdots{q_r}^{t_r-1}(1-q_r) \\ & + \sum_{j=1}^{\infty}(t_1+t_2+\cdots+t_r+j){q_1}^{t_1}{q_2}^{t_2}\cdots{q_r}^{t_r+j-1}(1-q_r) \end{aligned} \qquad (5.11)$$

The above expression can be simplified but it is computationally convenient to use the above form. Figure 5.2.5 is a plot of the expected number of trials required to find a perfect hashing function in an incrementally built file, using an optimal rehashing policy. The page size is 40 and $t_{max} = 10$. The numerical results indicate that, roughly, 40% to 60% of t_{max} trials on average are required to find a perfect hashing function when the page size is in the range 10 to 50 records.

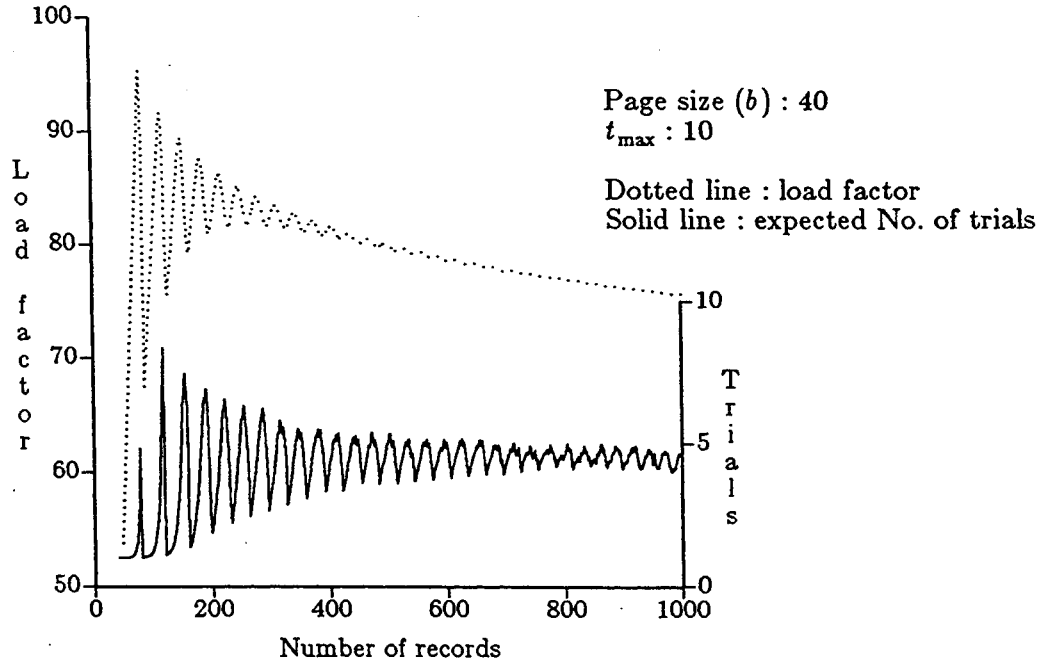


Fig. 5.2.5 Expected number of trials to find a perfect hashing function

Expected cost of a trial

Choosing a hashing function at random for a trial will be discussed in chapter 6. Once a hashing function has been chosen, a trial consists of computing all the hash addresses until one of the pages overflows. If the chosen hashing function is perfect all the n hash addresses have to be evaluated. If the trial is unsuccessful, we need not compute all the n hash addresses. Let $E_{ct}(n, m, b)$ denote the expected number of hash function evaluations required for verifying if a hashing function (to distribute n keys into m pages of size b) is perfect. E_{ct} is given by

$$E_{ct}(n, m, b) = \sum_{i=1}^{n} i * \left\{ \begin{array}{l} \text{Probability that the } i\text{th key} \\ \text{overflows and none of the} \\ \text{keys } 1, \cdots, (i-1) \text{ overflow} \end{array} \right\} + n * \left\{ \begin{array}{l} \text{Probability that none} \\ \text{of the keys } 1,2,\ldots,n \\ \text{overflow} \end{array} \right\} \qquad (5.12)$$

Let $\rho(i, m, b)$ denote the probability that the ith key is the first one to overflow when keys are being hashed one at a time. Recall that $P(i, m, b)$ is the probability of a random distribution of i keys causing no overflow. This means that $1 - P(i, m, b)$ is the probability that one of the keys 1, 2 ,... i overflows. Hence we have:

$$1 - P(i, m, b) = \sum_{k=1}^{i} \rho(k, m, b)$$

$$1 - P(i-1, m, b) = \sum_{k=1}^{i-1} \rho(k, m, b)$$

which by subtraction yields

$$P(i-1, m, b) - P(i, m, b) = \rho(i, m, b). \tag{5.13}$$

Eqn. (5.12) can now be written as,

$$\begin{aligned} E_{ct}(n, m, b) &= \sum_{i=1}^{n} i\ \rho(i, m, b) + n\ P(n, m, b) \\ &= \sum_{i=1}^{n} i\{P(i-1, m, b) - P(i, m, b)\} + nP(n, m, b) \\ &= P(0, m, b) - P(1, m, b) + 2\{P(1, m, b) - P(2, m, b)\} \\ &\qquad + 3\{P(2, m, b) - P(3, m, b)\} + \cdots + \\ &\qquad n\{P(n-1, m, b) - P(n, m, b)\} + n\ P(n, m, b) \\ &= P(0, m, b) + P(1, m, b) + \cdots + P(n-1, m, b) \end{aligned}$$

$$E_{ct}(n, m, b) = \sum_{i=0}^{n-1} P(i, m, b). \tag{5.14}$$

$E_{ct}(n, m, b)$ can be viewed as the area under the plot of $P(n, m, b)$ against n.

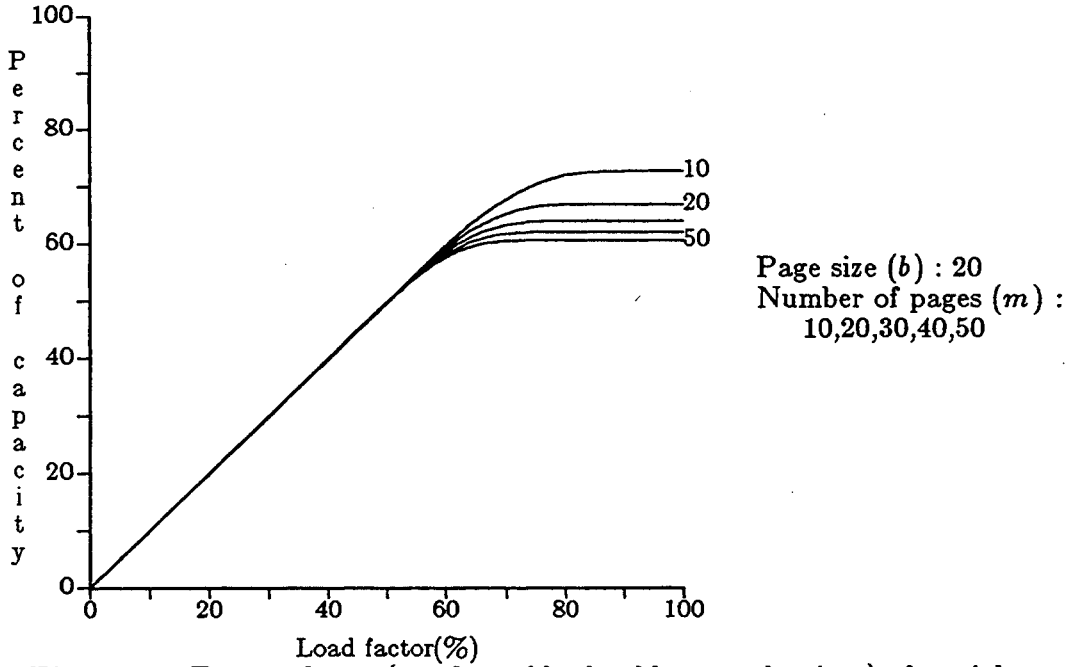


Fig. 5.2.6 Expected cost (number of hash address evaluations) of a trial

Figure 5.2.6 is a plot of $E_{ct}(n, m, b)$ as a function of n for $b = 20$ and m varying from 10 to 50. Both n and $E_{ct}(n, m, b)$ are expressed as a percentage of the full capacity of m pages, i.e., mb. The expected number of hash addresses to

be evaluated is almost constant (depending on b and m) when the load factor is above 70%. This is a direct consequence of the shape of the $P(n, m, b)$ curve.

Cost of successful and unsuccessful trials

Let $E_{cs}(n, m, b)$ denote the expected number of hash address evaluations of a successful trial. Since all the keys have to be hashed in a successful trial, E_{cs} is given by

$$E_{cs}(n, m, b) = n \tag{5.15}$$

Let $E_{cu}(n, m, b)$ denote the expected number of hash address evaluations of an unsuccessful trial. E_{cu} is readily obtained from (5.12). The first term in (5.12) corresponds to the product of E_{cu} and the probability of a trial being unsuccessful. (The second term is precisely $E_{cs}(n, m, b) * P(n, m, b)$). Thus from (5.14) we obtain:

$$\begin{aligned} E_{cu}(n, m, b) &= \frac{\sum_{i=0}^{n-1} P(i, m, b) - n\ P(n, m, b)}{1-P(n, m, b)} \\ &= \frac{\sum_{i=0}^{n-1} \{P(i, m, b) - P(n, m, b)\}}{1-P(n, m, b)} \end{aligned} \tag{5.16}$$

Expected cost of a rehashing policy

Let $E_{crp}(n, b)$ denote the expected number of hash address evaluations of a rehashing policy $(t_1, t_2, \cdots, t_r)$, that is, t_1 trials with m_l pages, t_2 trials with $m_l + 1$ pages and so on, until success with m_h pages. E_{crp} may be written as,

$$\begin{aligned} E_{crp}(n, b) &= (1-q_1)E_{cs}(n, m_l, b) + q_1(1-q_1)\Big(E_{cu}(n, m_l, b) + E_{cs}(n, m_l, b)\Big) \\ &\quad + \cdots + q_1^{t_1-1}(1-q_1)((t_1-1)E_{cu}(n, m_l, b) + E_{cs}(n, m_l, b)) \\ &+ q_1^{t_1}\{(1-q_2)\ \Big(t_1E_{cu}(n, m_l, b) + E_{cs}(n, m_l+1, b)\Big) \\ &+ q_2(1-q_2)\ \Big(t_1E_{cu}(n, m_l, b)+E_{cu}(n, m_l+1, b)+E_{cs}(n, m_l+1, b)\Big) + \cdots \\ &+ q_2^{t_2-1}(1-q_2)\ \Big(t_1E_{cu}(n, m_l, b) + (t_2-1)E_{cu}(n, m_l+1, b)+E_{cs}(n, m_l+1, b)\Big)\} \\ &+ \cdots + q_1^{t_1}q_2^{t_2}\cdots q_{r-1}^{t_{r-1}}\Big\{\cdots\Big\} \end{aligned} \tag{5.17}$$

The above expression can be simplified using the following identities for geometric series:

$$\begin{aligned} F_{us}(q,t) &= (1-q)q + 2(1-q)q^2 + \cdots + (t-1)(1-q)q^{t-1} \\ &= \left\{\frac{q(1-q^{t-1})}{(1-q)} - (t-1)q^t\right\}, \end{aligned} \tag{5.18}$$

$$F_{ss}(q,t) = (1-q) + (1-q)q + \cdots + (1-q)q^{t-1}$$
$$= (1-q^t)\,. \tag{5.19}$$

E_{crp} can now be written as:

$$\begin{aligned}
E_{crp}(n, b) = {} & F_{ss}(q_1,t_1)\, E_{cs}(n, m_l, b) + F_{us}(q_1,t_1)\, E_{cu}(n, m_l, b) \\
& + q_1^{t_1}\{F_{ss}(q_2,t_2)\, \left[t_1 E_{cu}(n, m_l, b) + E_{cs}(n, m_l+1, b)\right] \\
& \quad + F_{us}(q_2,t_2)\, E_{cu}(n, m_l+1, b)\} \\
& + q_1^{t_1} q_2^{t_2}\{F_{ss}(q_3,t_3)\, (t_1 E_{cu}(n, m_l, b) \\
& \quad + t_2 E_{cu}(n, m_l+1, b) + E_{cs}(n, m_l+2, b)\,) \\
& \quad + F_{us}(q_3,t_3)\, E_{cu}(n, m_l+2, b)\} + \cdots \\
& + q_1^{t_1} q_2^{t_2} \cdots q_{r-1}^{t_{r-1}}\{t_1\, E_{cu}(n, m_l, b) \\
& \quad + t_2\, E_{cu}(n, m_l+1, b) + \cdots + t_{r-1} E_{cu}(n, m_h-1, b) \\
& \quad + E_{cs}(n, m_h, b) + \frac{q_r}{(1-q_r)}\, E_{cu}(n, m_h, b)\}
\end{aligned} \tag{5.20}$$

It is very cumbersome to compute E_{crp} using (5.20). For practical purposes the following approximation seems adequate:

$$E_{crp} \approx E(t) * E_{cs}(n, em, b),$$

where $E(t)$ is the expected number of trials of the policy, and em is the integer nearest to the expected number of pages resulting from the policy.

I/O cost

Because the pages of a group are contiguous, a rehash does not necessarily involve a large number of disk accesses. The cost of accessing a group of contiguous pages is only slightly higher than the cost of accessing a single page (a characteristic of disks). If we can afford enough buffer space to store all the pages of a group in core, redistribution of the records can be accomplished in a single read access followed by a single write access. However, if buffer space is scarce, rehashing can be accomplished with as few as 3 page buffers, but at an additional I/O cost. Further discussion on this is postponed to chapter 7.

5.3. Internal storage requirements

In order to achieve one access retrieval, the header table must be stored in internal memory. The header table should be small enough to make this feasible. In this section we will consider only the number of entries in the header table rather than the exact memory requirement, because we have not yet considered the exact size of header table entries. The header table size depends on the file size. As an example we consider a file of one million records for comparison purposes.

Figure 5.2.7 plots the load factor of a file of one million records as a function of the number of entries in the header table. For a given file size, the average group size determines the number of groups and hence the header table size.

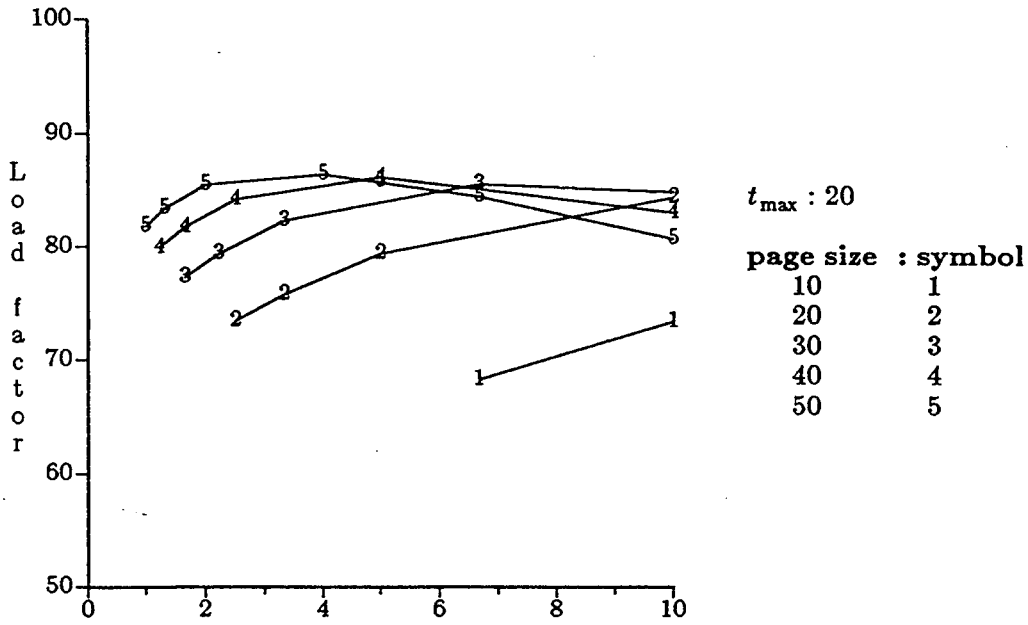


Fig. 5.2.7 Load factor of a file of 10^6 records as a function of header table size

(If a file has N records and λ is the average group size required, the size of the header table is N/λ.) Figure 5.2.7 is plotted using the same data as that for figure 5.1.3.

The page size affects the load factor more than the header table size. A header table of 2000 entries with a page size of 50 yields an 85% load factor. Doubling the header table to 4000 entries increases the load factor to only 86%. But if the header table size is kept at 2000 entries and the page size is reduced to 30, the resulting load factor drops to 77%. As we have already mentioned, page sizes below 20 appear to be impractical due to the large header table required to maintain a reasonable load factor.

Increasing the header table size beyond a certain point results in decreasing load factors due to fragmentation. Larger header tables correspond to smaller groups and cheaper individual rehashing. However, it is clear from Figure 5.2.2 that smaller groups require rehashing more often.

5.4. Conclusions

In this chapter we analyzed the performance of the external perfect hashing scheme under the optimal rehashing policy and reported numerical results for a few different page and group sizes. The results indicate that the external perfect hashing scheme is practical with reference to the resulting load factor and costs involved. The probability of rehashing is quite small and when rehashing is required, the costs involved are not prohibitively high. The scheme gracefully accommodates varying file sizes. For example, a file of one million records can be stored on a disk with 40 records per page at a load factor of over 82%. The header table then needs internal memory space to store 2000 entries. With this organization about 94% of the insertions can be handled at the minimum cost of

a single read access followed by a single write access. The other 6% of the insertions involve some extra computation and internal space: about 4,000 hash addresses need to be evaluated, and during this period about 500 records need to be stored in internal memory. However, there are several practical problems to be addressed before the scheme can be implemented in practice. The next two chapters address these problems.

Chapter 6

A practical class of hashing functions

6.0. Chapter overview

The analysis in chapters 3-5 assumed that hashing functions were chosen at random from the set of all functions. This is clearly impractical. In this chapter we propose and investigate a simple and practical class of hashing functions. The results of the experiments with this class of hashing functions indicate, that the relative frequency of perfect hashing functions within the class is statistically the same as predicted by the theoretical analysis for the set of all functions. This indicates that the predicted performance can be achieved in practice. The problem of converting alphanumeric keys into integers is also investigated.

6.1. Introduction

Our analysis of the proposed external perfect hashing scheme was based on the assumption that hashing functions were chosen at random from the set of all functions. Given n keys to be mapped into m addresses, there are m^n possible mappings: each key may be mapped to any one of the addresses independently. Choosing a function at random from this set of all functions is expensive: $\lceil n \log m \rceil$ random bits need to be generated. Also, $\lceil n \log m \rceil$ bits are required to represent a function from this set. An implicit assumption in all work on hashing is that computing the hash address should be fast, and require O(1) operations. It appears that this can not be achieved by using the functions chosen at random from the set of all functions. (Under RAM model of computation, arithmetic operations involving n bit numbers requires $\Omega(n)$ operations [AH74]).

6.2. Universal classes of hashing functions

Consider two sets of integers $A = \{0, 1, 2, \ldots, a-1\}$ and $B = \{0, 1, 2, \ldots, m-1\}$ where $a \gg m$. We refer to the integers in A as keys and those in B as addresses. Let H be a class of hashing functions, $H = \{h_1, h_2, h_3, \ldots\}$, where each h_i is a function $A \rightarrow B$. Let x and y be two distinct integers, $x, y \in A$, and $h_i \in H$, a hashing function. If $h_i(x) = h_i(y)$, x and y are said to collide under the hashing function h_i. Now define

$$\delta_{h_i}(x, y) = \begin{cases} 1 & \text{if } x \neq y \text{ and } h_i(x) = h_i(y) \\ 0 & \text{otherwise} \end{cases}$$

H is said to be a *universal*$_2$ class of hashing functions if for any $x, y \in A$, $\sum_{h_i \in H} \delta_{h_i}(x, y) \leq |H|/m$. In other words, H is said to be a *universal*$_2$ class of hashing functions if for every pair $x, y \in A$, they collide under no more than $|H|/m$ of the functions. If a function is chosen at random from this class of hashing functions, the probability of a pair of keys colliding is less than or equal to $1/m$.

If two keys from the set A are distributed at random over m addresses, the probability that they collide is $1/m$. This implies that the collection of all functions from A to B (whose cardinality is m^a) is *universal*$_2$ [SD80].

Consider the *universal*$_2$ class H_1, defined as follows by Carter and Wegman [CW79]:

Z_p denotes the finite field $\{0, 1, 2, \ldots, p-1\}$ where p is a prime number, $p \geq a$. Without loss of generality, assume $a = p$.

$f_{c,d} : A \rightarrow Z_p$

$$f_{c,d}(x) = (cx + d) \bmod p \text{ where } c > 0,\ d \geq 0 \text{ and } c, d < p.$$

$g : Z_p \rightarrow B$

$$g(x) = x \bmod m.$$

H_1 is the set of functions $\{h_{c,d} \mid c, d \in Z_p \text{ and } c \neq 0\}$, where $h_{c,d} : A \rightarrow B$ is given by,

$$h_{c,d}(x) = g(f_{c,d}(x)).$$

The following properties of H_1 can be proven [CW79]:

(1) For any given $x \in A$, there are $\lfloor |H_1|/m \rfloor$ (or $\lfloor |H_1|/m \rfloor + 1$ depending on the address) functions in H_1 which map x to a given address.

(2) Any two keys belonging to the set A collide at most under $|H_1|/m$ functions.

The class H_1 and the set of all functions are equivalent as far as the above two properties are concerned. In addition the functions in the class H_1 have the following property:

(3) If all the keys in the set A are hashed using an arbitrary but fixed function in H_1, each of the addresses 0, 1, ..., $m-1$ will receive exactly $\lfloor a/m \rfloor$ or $\lfloor a/m \rfloor + 1$ keys.

This means that every function $h \in H_1$ is perfect for a large number of subsets of A, and that there are no "uniformly bad" functions in H_1. This is not true for the set of all functions. For example, a constant function mapping all the keys to a single address is perfect only for subsets of size b or less. These three observations and the fact that a function is uniquely specified by only two parameters led us to consider the class H_1 as a candidate for a practical class of hashing functions.

6.3. Description and results of the experiments

In view of the above observations regarding the class H_1 and the set of all functions, we performed a series of experiments using several test files to determine if the class H_1 could be used for selecting functions for trials. Our hypothesis is:

> *The relative frequency of perfect hashing functions mapping n keys into m pages of size b in the class H_1 is the same as that predicted by the analysis for the set of all functions.*

The experiments were aimed at testing the above hypothesis. Choosing a function at random from H_1 can be done simply by choosing the parameters c and d using a pseudo random number generator.

The key sets for the experiments were obtained from the following ASCII files:

(A) Dictionary of words on UNIX (24,000 keys) (used for spelling checking, etc.).

(B) Userids from the IBM/CMS System at the University of Waterloo (12,000 keys).

(C) Keywords from a library database at the EMS Library, University of Waterloo (3,000 keys).

(D) Part of a file containing titles from the EMS Library, University of Waterloo (10,000 keys).

(E) Call numbers from the Physics Department Library, Washington University (6,000 keys).

All the data was in alphanumeric form and the length of the individual keys varied from 2 to 25 characters. The keys were converted into integers (4 bytes long). Since the analysis assumed that the keys were distinct (this is the standard assumption in all work on hashing), if two distinct keys mapped to the same integer then one of the keys was eliminated. The problem of converting ASCII strings into integers so as to avoid collisions of this type will be discussed in section 6.4.

The keys in each test file were divided into groups using hashing functions of the form $H(x) = (cx + d) \bmod p \bmod s$, where s is the number of groups desired, group $i = \{x \mid H(x) = i\}$. This was done to partially simulate the organization of a file using external perfect hashing. For each file, s was chosen so as to obtain groups containing about 1000 keys. Eight groups of keys as indicated by table 6.3.1 were selected for detailed experiments.

Key set number	File from which obtained	Hashing functions used to separate groups	Size of the group
1	group 3 of file A	$(314159x + 27182) \bmod 2100000011 \bmod 19$	1475
2	group 9 of file A	$(314159x + 27182) \bmod 2100000011 \bmod 19$	1507
3	group 3 of file B	$(314159x + 27182) \bmod 2100000011 \bmod 11$	965
4	group 6 of file B	$(314159x + 27182) \bmod 2100000011 \bmod 11$	981
5	group 2 of file C	$(27182x + 314159) \bmod 2100000011 \bmod 3$	1120
6	group 2 of file D	$(27182x + 314159) \bmod 2099999999 \bmod 11$	901
7	group 0 of file E	$(314159x + 27182) \bmod 2100000011 \bmod 7$	855
8	group 1 of file E	$(314159x + 27182) \bmod 2100000011 \bmod 7$	880

Table 6.3.1 Key sets selected for experiments

Each experiment was performed as follows. The page size b, the number of pages in the group m, and the load factor α were specified. A set of 500 hashing functions was created by randomly generating 500 (c, d) pairs. The standard random number generator RANDOM( ) supplied with UNIX was used (seed = 314159). The prime number p was chosen as 2100000011. Since c and d should be less than p, all random numbers above the value of p were ignored. The number of records was then computed as $n = \alpha mb$ and the first n keys of a group were read in. The n keys were hashed by each of the 500 hashing functions and the number of perfect hashing functions was recorded. The same procedure was repeated for each of the groups indicated in table 6.3.1. The same set of hashing functions was used for the first four groups. A different set of hashing functions was generated and used for the last four groups (seed = 951413, p = 2100000017). The experiment was then repeated for various values of α. The key sets used from a group at different values of α are not independent of each other. To reduce the dependence of the results at different load factors, a different set of hashing functions was generated for each value of α. Ideally, at each different load factor, a different key set should be used for the experiment. This experiment was repeated for various values of m and b, in that order.

Figure 6.3.1a plots the results of one set of experiments with b = 10 and m = 15. The solid line is a plot of $P(n, 15, 10)$ computed using the recurrence relation given in chapter 4. For each value of the load factor varying from 50%

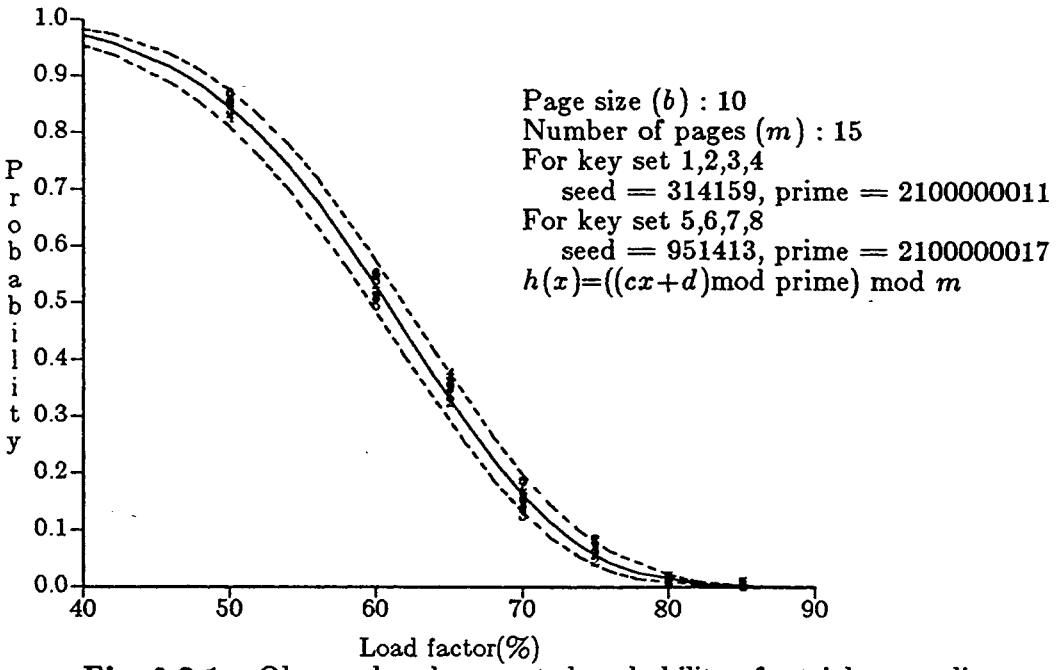


Fig. 6.3.1a Observed and computed probability of a trial succeeding

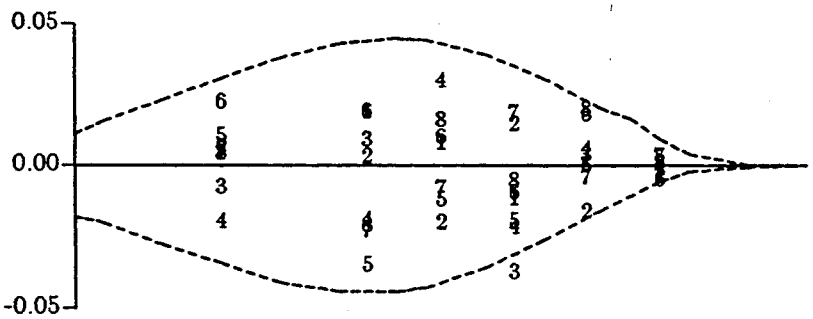


Fig. 6.3.1b Difference between observed and computed frequencies

to 85%, eight points are plotted using symbols 1, 2 , . . . , 8, corresponding to the eight data sets. Each point represents the relative frequency of perfect hashing functions (among the 500 generated for that load factor) for the corresponding key set.

The dashed lines enclose a critical region [FR71]. This critical region represents the central 95% of the sampling distribution, when the sample size is 500, for the relative frequency of perfect hashing functions within the set of all functions. For a particular value of the load factor α, the point on the solid line represents the probability $P(\alpha mb, m, b)$, which we shall denote by θ for convenience. Let $\overline{\theta}$ and $\underline{\theta}$ denote the corresponding points on the dotted line above and the dotted line below the solid line. This means that, if 500 functions are chosen at random from the set of all functions, the relative frequency of perfect hashing functions will fall within the range $\underline{\theta}$ to $\overline{\theta}$ with 95% probability. For a given value of θ, the approximate values of $\overline{\theta}$ and $\underline{\theta}$ may be obtained using the

relations $\overline{\theta} \approx \theta + 1.96\sigma$ and $\underline{\theta} \approx \theta - 1.96\sigma$ where $\sigma = \sqrt{\theta(1-\theta)/500}$. The exact values of $\overline{\theta}$ and $\underline{\theta}$ were obtained using the statistical tables [BR71].

Figure 6.3.1b shows the same results on a magnified scale. It shows the difference between the observed and the expected frequencies, that is, a point $< \alpha, y >$ in figure 6.3.1a is mapped onto $< \alpha, y - P(\alpha mb, m, b) >$ in figure 6.3.1b. The x axis in figure 6.3.1b corresponds to the solid line in figure 6.3.1a. We observe that all the experimental values fall within the critical region, except for one point at 70% load factor.

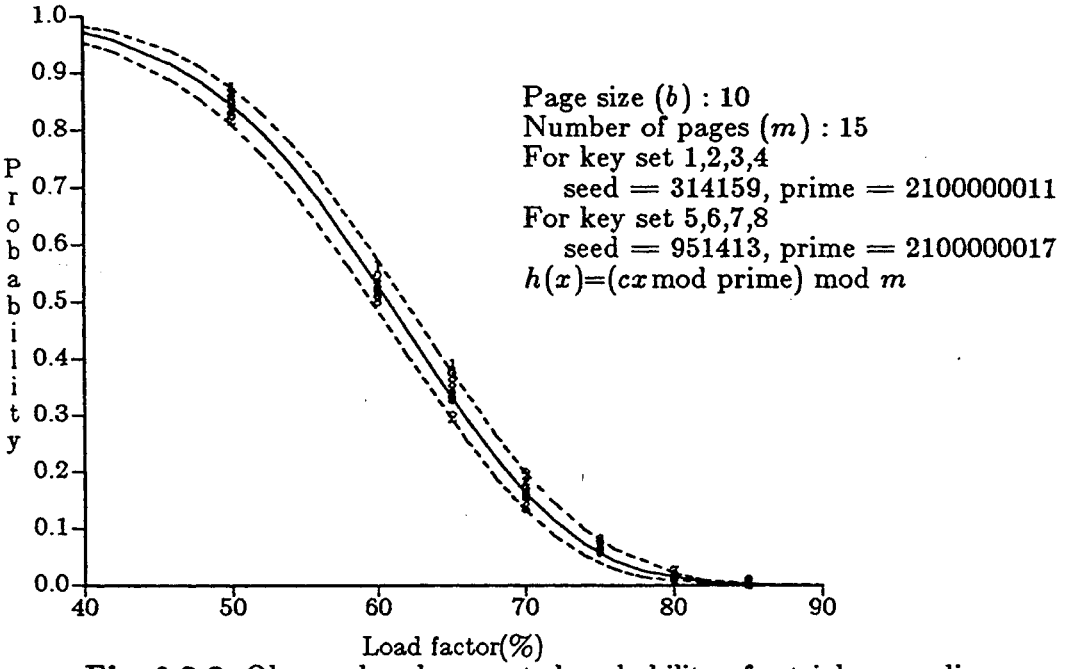


Fig. 6.3.2 Observed and computed probability of a trial succeeding

Intuitively, it appears that the addition of d to the product cx in the hashing function plays an unimportant role. Carter and Wegman [CW79] showed that the class of functions resulting from setting d to zero comes within a factor of 2 of being $universal_2$ (i.e. any x, y collide under no more than $2|H|/m$ of the functions). However, this does not seem to influence the relative frequency of perfect hashing functions as indicated by figure 6.3.2. The data for figure 6.3.2 was obtained by experiments exactly similar to those for figure 6.3.1a. The same sets of keys and hashing functions were used except that d was set to zero for all the hashing functions. We observe that all relative frequencies of perfect hashing functions fall within the critical region, except for key set 1 at 65% load factor.

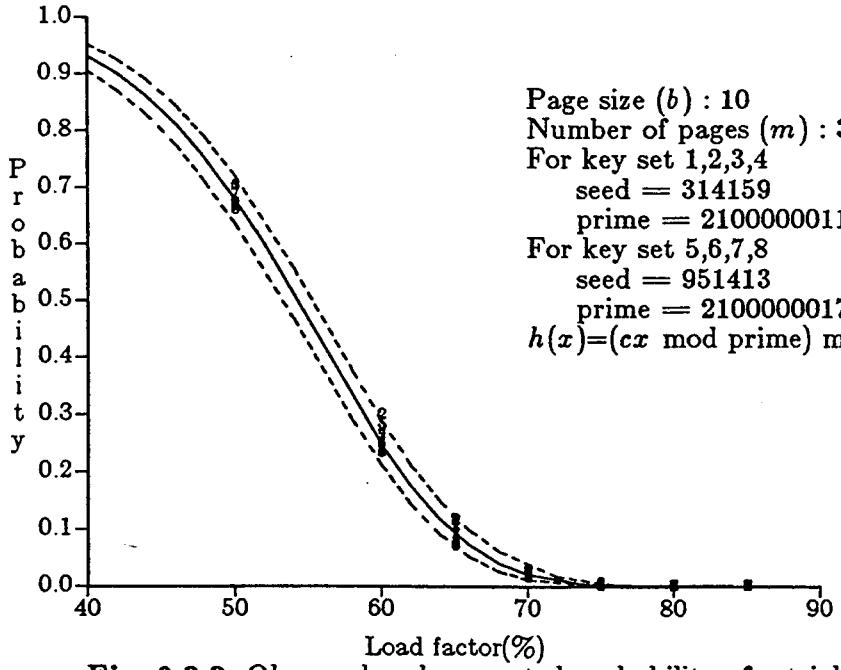


Fig. 6.3.3 Observed and computed probability of a trial succeeding

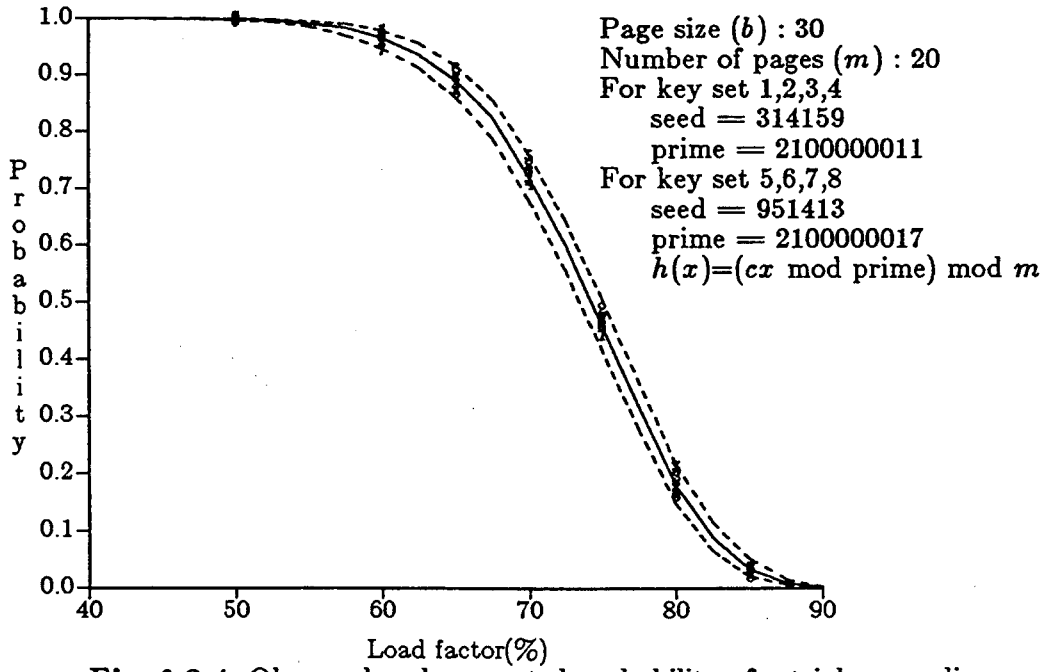


Fig. 6.3.4 Observed and computed probability of a trial succeeding

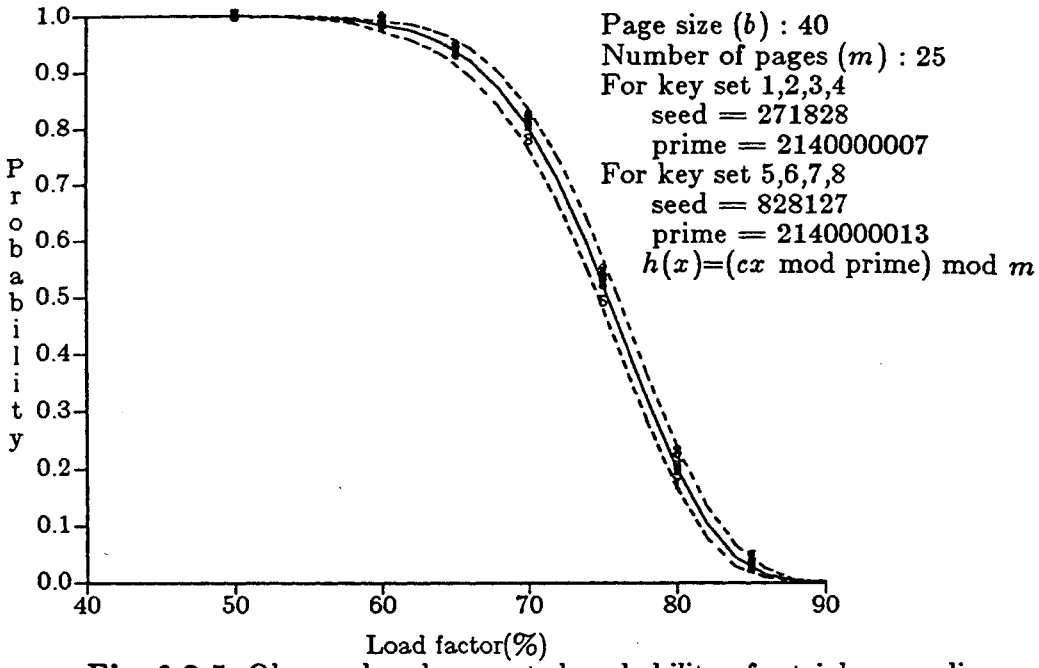


Fig. 6.3.5 Observed and computed probability of a trial succeeding

Figures 6.3.3 to 6.3.5 plot the results of experiments with different values of b and m. The same set of eight key sets were used for all experiments. Note that the required number of keys used for a given experiment depends on the value of α, m and b. The value of the seed used to initialize RANDOM( ) for generating hashing functions and the value p are also shown on the graphs.

We observe that only a few experimental results fall outside the critical region. In addition to the results shown, we have experimented with a number of other sets of keys and hashing functions for various values of b and m, and the results were similar to those shown here. Table 6.3.2 summarizes some of the statistics gathered. Column 1 and 2 show the page size and the number of pages. Columns 4 and 5 show the number of observations falling outside the critical region and column 3 the total number of observations. Out of a total of 344 observations shown in column 3, 11 observations (3.2%) fall outside the critical region. Columns 6 and 7 show the corresponding results when d is set to zero in the hashing functions. For this case 9 out of 344 (2.6%) observations fall outside the critical region. The expected number of observations which should fall outside the critical region is 17.2 (5%). The results of the experiments provide no evidence to contradict our hypothesis. Subject to the limits of the sampling variability, the relative frequency of perfect hashing functions within the class H_1 is the same as that predicted by the analysis for the set of all functions. This holds for both the case when $d \neq 0$ and for the case when $d = 0$.

b	m	Number of observations	$d \neq 0$		$d = 0$	
			No. above	No. below	No. above	No. below
10	15	48	0	1	1	0
10	20	40	1	1	0	1
20	10	48	0	2	2	1
20	15	40	1	0	1	0
30	15	40	1	0	1	0
30	20	48	1	0	0	1
40	20	40	1	1	1	0
40	25	40	1	0	0	0

Table 6.3.2 Summary of the experimental results.

To ensure statistical independence in the above experiments, a different set of hashing functions were used for each different load factor. It is also interesting to see how a single set of functions behave for different load factors. When the load factor is around 50%, almost all the functions (500 of them) are perfect. What happens if the number of records is increased gradually? Does the number of perfect hashing functions for the key set decrease as predicted by the theoretical model?

Figures 6.3.6, 6.3.7, 6.3.8 plot the results of experiments in which the same set of 500 hashing functions was used for all load factors. We observe that the behavior is again close to that predicted by the theoretical model. These results also show that when using the class H_1, the probability of an insertion causing a rehash given by (5.1) and (5.2) is very close to that predicted by the analysis. Rigorous statistical testing is difficult in this case because of the high correlation between observations for the same key set at different load factors.

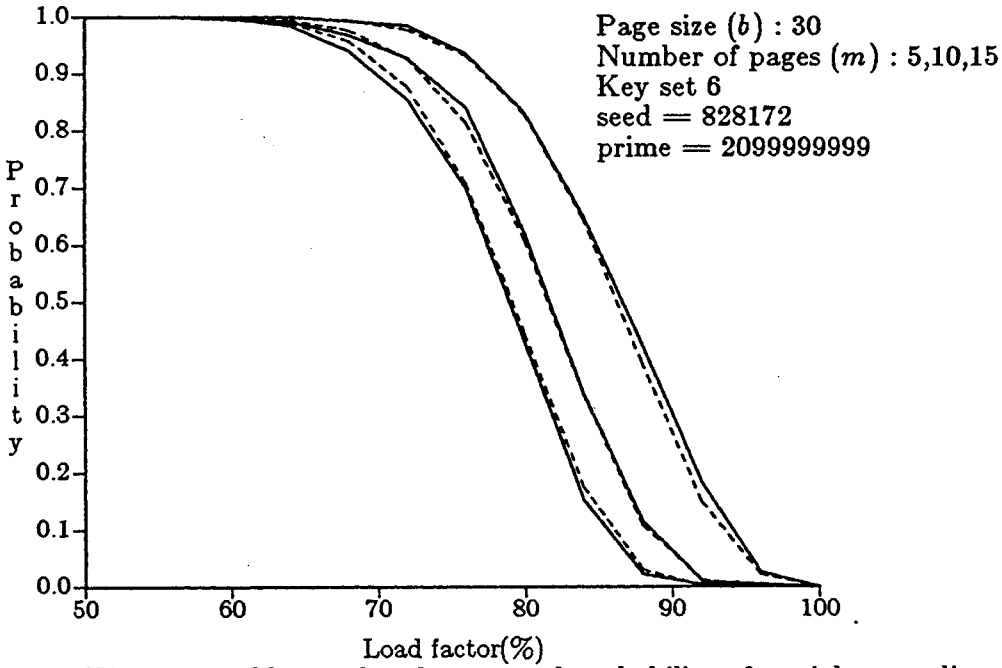


Fig. 6.3.8 Observed and computed probability of a trial succeeding
Same set of hashing functions for all α

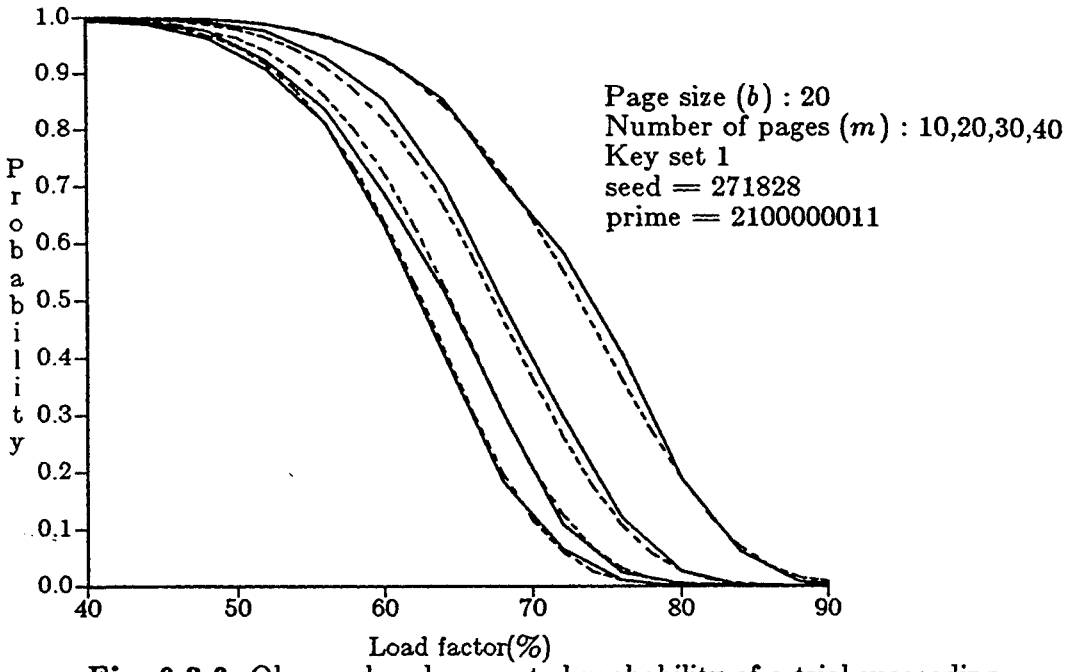


Fig. 6.3.6 Observed and computed probability of a trial succeeding
Same set of hashing functions used at all α

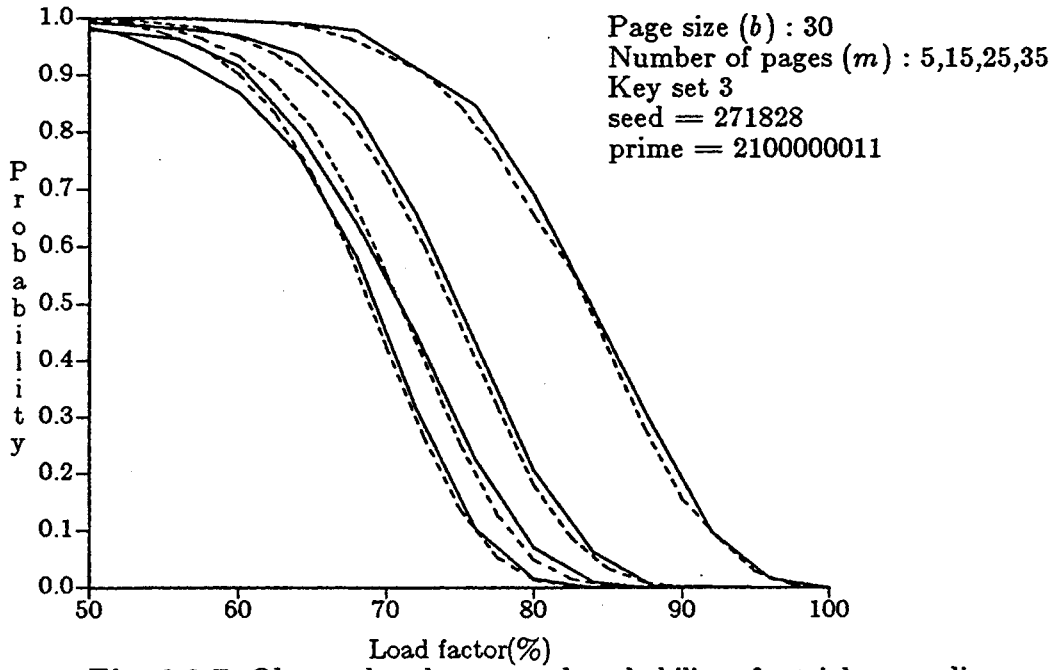


Fig. 6.3.7 Observed and computed probability of a trial succeeding
Same set of hashing functions for all α

In conclusion, we have exhibited a class of simple functions from which hashing functions can easily be chosen for trials. The experiments reported above indicate that the relative frequency of perfect hashing functions in this class is the same as that predicted by the theoretical analysis for set of all functions. These probabilities were the basis for all the performance results in chapters 3 to 5. Hence the results of the theoretical analysis can be used to predict the performance of the proposed scheme when the hashing functions are drawn from the class H_1.

6.4. Conversion of alphanumeric keys into integers

In the literature on hashing, the keys are usually assumed to be integers. However, in practice keys are often in alphanumeric form: a string of ASCII characters. Since information is loosely packed in the ASCII representation, treating the binary representation of ASCII strings as unsigned integers results in very large numbers for the key values.

One method used for converting an ASCII string into an integer key is given below [LK84]. The least significant 5 bits of each character are extracted and concatenated. Six such characters yield 30 bit integers which can fit into a single word. If the string is longer, the successive 30 bits obtained are XORed together. The algorithm is given in a Pascal-like notation below. In the following procedure, "ord" is a function which returns the integer corresponding to an ASCII character.

Algorithm to convert an ASCII string into an integer:

```
function XOR_Convert (keystring, len);
    keystring: array [1..len] of char;
    len: integer;
begin
    keyint, word, count, ch, i : integer;
    keyint := word := count := 0;
    for i := 1 to len do
        ch := AND ( ord(keystring[i]) , 37₈);
        word := shiftleft (word, 5) + ch;
        count := count + 1 ;
        if (count  = 6) then
            keyint := XOR (keyint, word);
            word := count := 0;
        endif;
    endloop;
    return (XOR (keyint, word));
end;
```

This method has an obvious problem. Two or more different keys may yield the same integer. This may be called a collision by regarding the conversion function as a hashing function from the set of alphanumeric strings into integers. The actual address translation function is then the composition of the conversion function operating on the alphanumeric string, and the hashing function mapping the resulting integer onto the address range. Obviously, all keys which collide

under the conversion function will collide under the composite function.

We shall define a K-**cluster** as a set of K distinct keys which map to the same integer under the string to integer conversion function. Clusters do not cause any serious problems for "normal" hashing schemes, because they are automatically handled by the regular overflow handling mechanism of the hashing scheme. Clusters pose a much more serious problem to any perfect hashing scheme because there is no overflow handling mechanism. The presence of clusters in a group increases the cost of finding a perfect hashing function for the group. If there is a K-cluster in a group and the page size is less than K, it is impossible to find a perfect hashing function for the group. In the next section we will analyze the effect of a cluster on the cost of finding a perfect hashing function. We next consider a few conversion methods which reduce the number and size of clusters.

When all the keys in file A were converted to 30 bit integers using the simple XORing scheme described above, seventeen 2-clusters resulted. A simple modification to the scheme resulted in the reduction of the number of 2-clusters from 17 to 7. An extra character, whose value is the length of the key string, was appended whenever the length of the key string was not a multiple of 6. (Note that in XOR-convert, bit patterns from each successive 6 characters were XORed together.) Similar improvements were observed for other files.

Cluster Size	Integer size in number of bits			
	30	25	20	15
2	7 (10^{-5})	55 (10^{-2})	497 (6)	4078 (3200)
3	0	1 (10^{-6})	12 (.05)	1208 (800)
4	0	0	0	333 (150)

Table 6.4.1 Number of clusters for file A using the modified XOR conversion procedure

Table 6.4.1 shows the number of clusters of different size in file A when using the modified XOR conversion procedure. The figures in parenthesis are the expected number of clusters[†] from an ideal conversion function, which randomly assigns an integer to each of the key strings. We observe that the actual number of clusters is much higher, indicating that it should be possible to further improve the conversion procedure.

[†] Let N be the total number of keys in the file and let the conversion procedure return B bit integers. The load factor $\lambda = N/2^B$. The expected number of K-clusters is given by $Ne^{-\lambda}\frac{\lambda^k}{k!}$, which can be approximated by $\frac{N\lambda^k}{k!}$ when $\lambda \ll 1$.

To this end, we experimented with the following conversion method based on a radix-36 representation. Each character of the alphabet is assigned an integer. If there are R characters in the alphabet, the key can be represented by a radix-R number. For example, for alphanumeric keys we can define a function "ord" as follows: ord('a') = 1, ord('b') = 2, ... ord('z') = 26, ord('0') = 27, ... ord('9') = 36. Any alphanumeric string can then be expressed as a radix-36 number. (Note that upper and lower case are not distinguished.) Such a radix based representation is also called packed form. If the key value in this representation exceeds the required word size, the modulus of the number to a suitable prime is taken. The conversion algorithm is given below. The largest prime number less than 2^B is used while converting key strings into B bit integers. (Thus $prime$ = 1,073,741,789 when B = 30, $prime$ = 33,554,393 when B = 25, and $prime$ = 1,048,573 when B = 20.)

Algorithm to convert an ASCII string into an integer:

```
function RADIX_Convert (keystring, len, prime);
    keystring: array 1..len of char;
    len, prime: integer;
begin
    keyint, i : integer;
    keyint := 0;
    for i := 1 to len do
        keyint := ((keyint * radix) mod prime) + ord(keystring[i]);
    endloop;
    return (keyint mod prime);
end;
```

The number of clusters decreased dramatically with this conversion algorithm as indicated by table 6.4.2. The figures in parenthesis are the number of clusters resulting from the modified XORing method. The relative magnitude of the number of clusters among the different files is not significant, because the number of keys in the files are not equal. The clusters are distributed among the various groups of the file (grouping functions given in table 6.3.1). The results for file C are not included because the file contained only 3000 keys.

Cluster size	Data	File size	Integer size in number of bits		
			30	25	20
2	file A	24000	0(7)	10(55)	286(497)
	file B	12000	0(12)	11(184)	51(309)
	file D	10000	0(3)	0(9)	51(53)
	file E	6000	0(2)	1(12)	19(73)
3	file A	24000	0(0)	0(1)	2(12)
	file B	12000	0(0)	0(5)	0(9)
	file D	10000	0(0)	0(0)	0(1)
	file E	6000	0(0)	0(0)	0(3)

Table 6.4.2 Comparison of the number of clusters resulting from RADIX_Convert and modified XOR_Convert.

6.5. Effect of a cluster on $P(n, m, b)$

It is obvious that the presence of a cluster in a group reduces the probability of a randomly chosen function being perfect. If the cluster size exceeds the page size then no perfect hashing function for the group exists. Let $P^k(n, m, b)$ denote the probability of a randomly chosen function being perfect for a group having $(n - k)$ distinct keys and one k-cluster (for a total of n keys) to be distributed into m pages of size b.

Consider the situation when the $(n - k)$ distinct keys are distributed randomly among m pages. The cluster will hash into a randomly chosen page. The conditional probability, p_0, that the cluster does not cause the page to overflow is given by

$$p_0 = \frac{P^k(n, m, b)}{P(n - k, m, b)}. \tag{6.1}$$

The page will overflow if there were already $(b - k + 1)$ or more keys in the page. The probability of this event, $1 - p_0$, is given by (this is analogous to 3.4)

$$1 - p_0 = \frac{\sum_{i=0}^{k-1} \binom{n-k}{b-i} P(n-k-b+i, m-1, b)(m-1)^{n-k-b+i}}{P(n-k, m, b)\, m^{n-k}}. \tag{6.2}$$

Each term of the sum in the numerator corresponds to the number of ways in which $(b-i)$ keys may be chosen from the $(n-k)$ distinct keys, and the other $n-k-(b-i)$ keys distributed among $(m-1)$ pages. The denominator represents the total number of ways in which $(n-k)$ keys may be distributed among m pages. Eliminating p_0 from (6.1) and (6.2) we obtain

$$P^k(n, m, b) = P(n-k, m, b) - \sum_{i=0}^{k-1} \binom{n-k}{b-i} P(n-k-b+i, m-1, b)\, \frac{(m-1)^{n-k-b+i}}{m^{n-k}}. \tag{6.3}$$

In particular, consider the case where k is 2. From the above equation we obtain:

$$P^2(n, m, b) = P(n - 2, m, b) - \binom{n-2}{b} P(n - b - 2, m - 1, b) \frac{(m-1)^{n-b-2}}{m^{n-2}} - \binom{n-2}{b-1} P(n - b - 1, m - 1, b) \frac{(m-1)^{n-b-1}}{m^{n-1}}. \tag{6.4}$$

From equations (4.5) and (6.4) we obtain the following:

$$P^2(n, m, b) = P(n, m, b) - P(n - b - 1, m - 1, b)\, \frac{(m-1)^{n-b-1}}{m^{n-2}} \left\{ \binom{n-2}{b-1} - \binom{n-1}{b} \frac{1}{m} \right\}$$

We can use the procedure described in chapter 4, with minor modifications, to compute $P^2(n, m, b)$, by rewriting $P^2(n, m, b)$ in the form

$$P^2(n, m, b) = P(n-1, m, b) - \binom{n-2}{b-1} P(n-b-1, m-1, b) \frac{(m-1)^{n-b-1}}{m^{n-2}}$$

$$= P(n-1, m, b) - \binom{n-1}{b} P(n-b-1, m-1, b) \frac{(m-1)^{n-b-1}}{m^n} \left\{\frac{mb}{n-1}\right\}.$$

We can now easily modify the procedure given in section (3.2) for computing $P(n, m, b)$, to compute $P^2(n, m, b)$: replace *term* with $term * \left\{\frac{mb}{n-1}\right\}$ during the last iteration. Figures 6.5.1 and 6.5.2 are plots of $P^2(n, m, b)$, showing the effect of a single 2-cluster on $P(n, m, b)$.

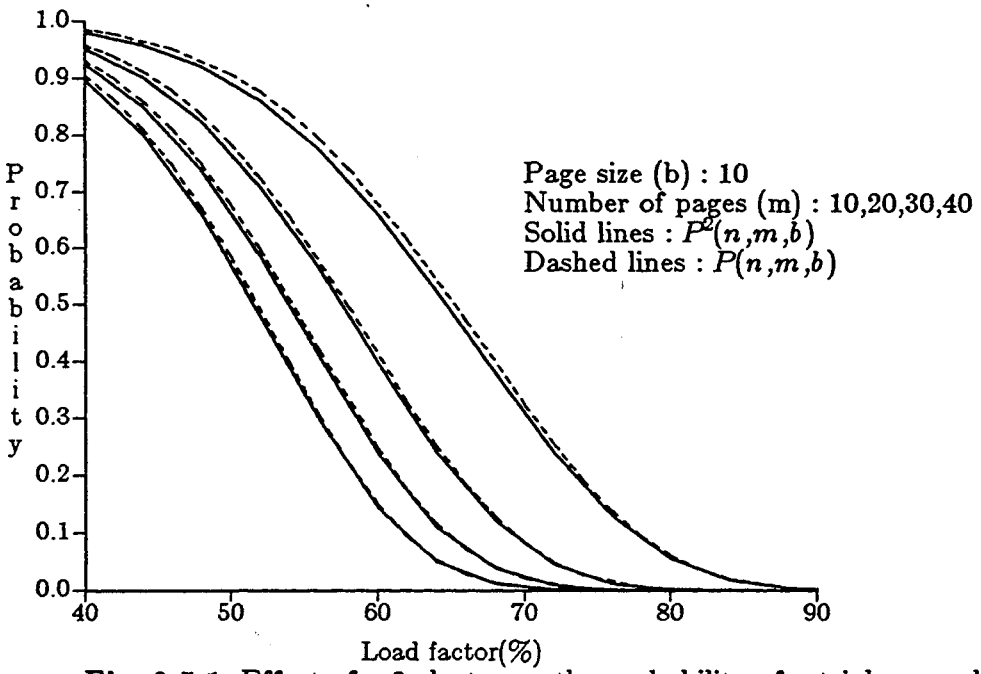


Fig. 6.5.1 Effect of a 2-cluster on the probability of a trial succeeding

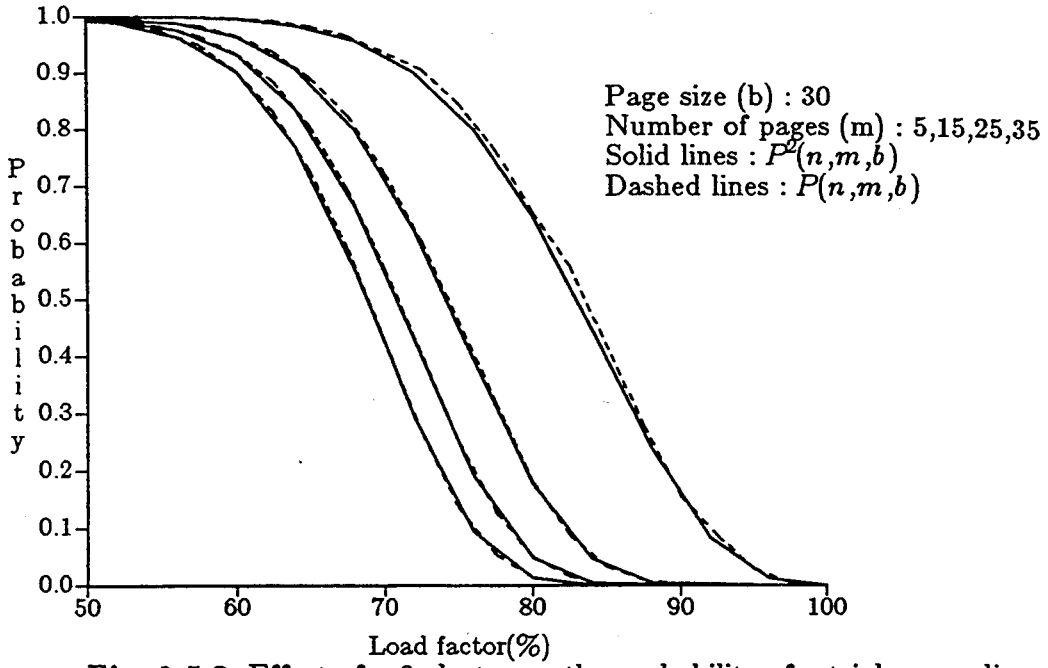


Fig. 6.5.2 Effect of a 2-cluster on the probability of a trial succeeding

It should be pointed out that the analysis and experiments described in this section aimed to gain a first indication of the effect of clusters. As may be expected, the effect of clusters is more noticeable for small pages than for large pages. A single cluster results in an observable (in the plots) reduction in the probability of a trial succeeding. We have conducted a single set of experiments to see the effect of a cluster in practice. Key set 7 was modified by inserting a key whose value was exactly the same as that of the first key in the file. In other words, the first two keys of the modified key set 7 were identical. Experiments similar to the ones performed to obtain data for figures 6.3.6 were conducted with the modified key set. Figures 6.5.3 and 6.5.4 are plots of the results obtained. The plots show that the effect of a 2-cluster on $P(n,m,b)$ is approximately the same as that predicted by the analysis. The results of the previous section show that very few 2-clusters are encountered in practice when RADIX_Convert is used for conversion of alphanumeric keys into integers. Clusters of size 3 or higher are highly improbable even with the simple modified XOR_convert. Consider an example with page size 10. When the group size is 10 pages and the load factor 80% , the effect of a 2-cluster is to increase the expected number of trials required to find a perfect hashing function from 14.7 to 15.2; with a group of 30 pages and load factor of 68%, the effect of a 2-cluster is to increase the number of trials from 21.7 to 25. In both cases above, $P^2(n,m,b)$ is within the critical region [section 6.3], i.e., the effect of a cluster is less than that of statistical variation. Thus the effect of clusters on the performance of the external perfect hashing scheme is rather insignificant. Hence, we conclude that the problem of clusters should not be a serious one in practice.

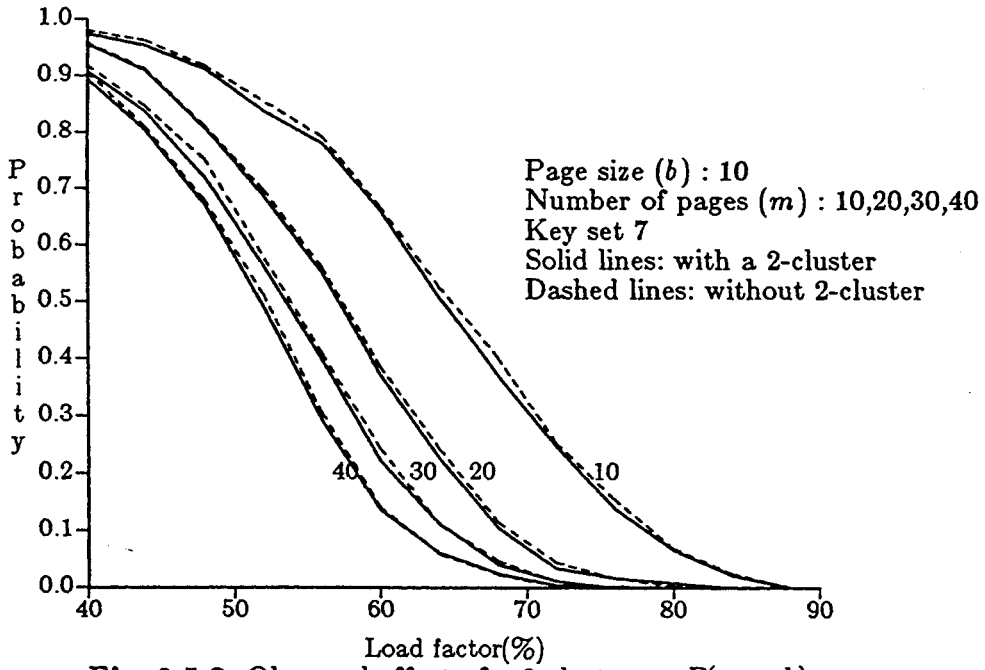


Fig. 6.5.3 Observed effect of a 2-cluster on $P(n,m,b)$

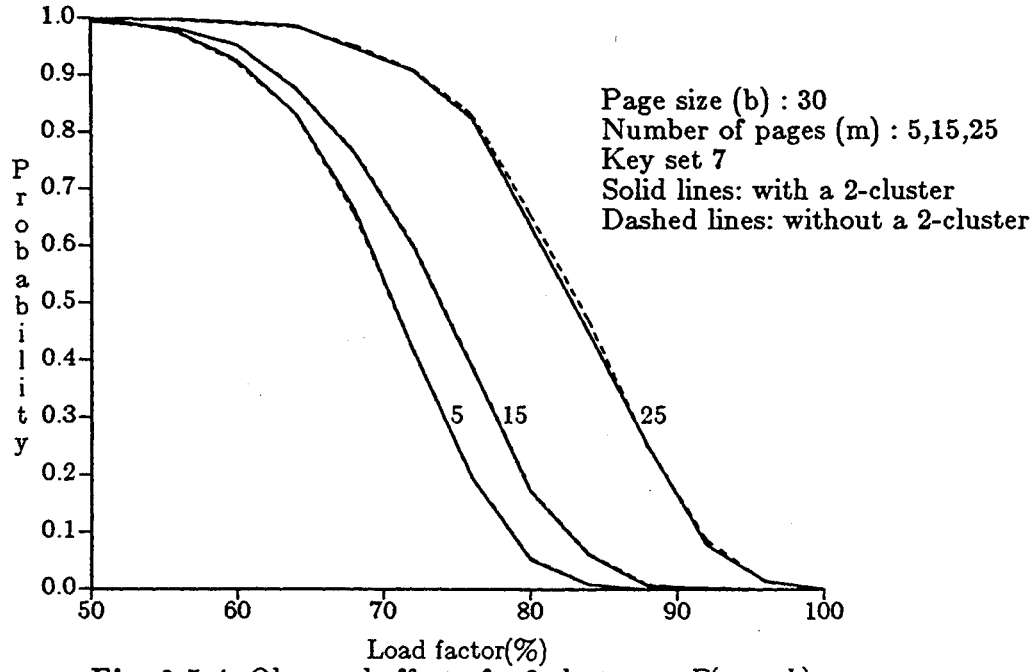


Fig. 6.5.4 Observed effect of a 2-cluster on $P(n,m,b)$

6.6. Conclusions

In this chapter we explained why functions for trials cannot be chosen from the set of all functions. A $universal_2$ class of hashing functions was chosen for experiments and the reasons for this choice were explained. Experimental results for several real life files were presented. The results support the hypothesis that the relative frequency of perfect hashing functions in the chosen class H_1 is statistically the same as predicted by the analysis for the set of all functions presented in chapter 3.

This chapter completed the answer to one of the main questions posed in chapter 2: How to find perfect hashing functions? In chapter 3 we proposed a trial and error method of finding perfect hashing functions and the analysis was based on choosing functions at random from the set of all functions. The results of this analysis formed the basis for the performance model in chapters 4 and 5. The results presented in this chapter show that the model can be used to predict the performance of the proposed scheme when hashing functions are drawn from the class H_1.

Chapter 7

Implementation considerations

7.0. Chapter overview

It is impractical to solve a nonlinear integer optimization problem, as described in chapter 4, to find the optimal rehashing policy every time a new perfect hashing function is needed. We give a simple heuristic for determining a policy, whose performance is close to that of the optimal policy. The practical aspects of rehashing and relocating the groups, and organizing the header table are discussed. Finally, extending the external perfect hashing scheme to make it fully dynamic is discussed briefly.

7.1. A simple heuristic for finding rehashing policies

In chapter 4 we described a policy for distributing the total number of trials over a range of load factors. We also defined an optimal rehashing policy and formulated a nonlinear optimization problem whose solution gives the optimal policy for distributing the trials. It is a difficult problem to solve, and we described a heuristic solution procedure based on dynamic programming to solve the problem approximately. As such the solution procedure is an efficient method for solving the optimization problem. From the point of view of determining a policy of finding perfect hashing functions, the procedure is simply too complex. Hence, it cannot be used in practice. However, all the effort spent on the optimization problem was not wasted. By examining the optimal policies obtained, we were able to design a simple heuristic for distributing the trials over different load factors.

The complexity of the solution procedure is $O(t^2_{max} * m * depth)$, where t_{max} is the maximum number of trials allowed, m is the number of pages in the group, and *depth* is a parameter of the solution procedure. For example, consider a group with 1,000 keys to be distributed into pages of size 40. If t_{max} is 20, we can expect around 80% load factor, corresponding to about 30 pages. Assuming a modest value of 3 for *depth*, the solution of the optimization problem involves about 15,000 evaluations of functions of the form $f_i = m_i(1 - q_i^{t_i}) + q_i^{t_i} f_{i+1}$. However, once a policy has been determined, the cost of actually finding a perfect hashing function is much less: on the average, about 8,000 evaluations of functions of the form $h(x) = (cx \ mod \ p) mod \ m$ are sufficient. There is less than a 1% chance (assuming $P_e = 0.99$) of requiring more than 20,000 evaluations.

The optimization problem has some "hidden" costs as well. The $P(n, m, b)$ values must be computed each time, or a table of $P(n, m, b)$ values could be computed once and stored. This table would be too large to be maintained in main memory and keeping it on secondary storage results in increased I/O cost.

Clearly, determining the optimal policy by solving the optimization problem is very expensive. The cost of the solution procedure can be reduced to some extent. The full table of $P(n, m, b)$ is not necessary: any required value can be obtained by interpolation from a smaller table of $P(n, m, b)$ values. Observing the optimal policies (i.e., the solutions obtained by solving the optimization problem) it appears feasible to modify the solution procedure so as to eliminate large amounts of search. These techniques cannot, however, bring down the cost to practically acceptable levels. The computational power can be better utilized by increasing t_{max} rather than finding an optimal policy with a lower value of t_{max}.

7.1.1. Heuristic policy

Our attempt to reduce the cost of formulating and solving the optimization problem led to a heuristic for determining the policy. This heuristic does not require formulating and solving an optimization problem.

Let us first clarify the terminology. We described a heuristic in chapter 4 to reduce the search space in the solution procedure based on dynamic programming; it gives solutions which are close to the optimal policy. In this section, we describe a heuristic to arrive at a policy for distributing the trials without formulating any optimization problem at all. This heuristic was derived from the optimal policies obtained using the procedure described in chapter 4. It is based on the following three observations.

number of keys (n)	policy
71	0 2 5* 2 0 1
72	0 1 5* 3 0 1 0
73	0 0 6* 2 1 1 0
74	0 0 6* 2 1 0 1
75	0 0 5* 2 2 0 1
76	0 0 3* 4 2 0 1
77	0 0 2 5* 2 0 0 1
78	0 0 2 4* 3 0 0 1
79	0 0 0 6* 2 1 0 1
80	0 0 0 5* 3 1 0 1
81	0 0 4* 3 2 0 1
82	0 0 3* 3 3 0 1
83	0 0 3 3* 3 0 0 1
84	0 0 2 3* 4 0 0 1
85	0 0 0 5* 3 1 0 1
86	0 0 0 5* 2 2 0 1
87	0 0 0 4* 2 3 0 1
88	0 0 0 3* 3 3 0 0 1
89	0 0 0 1 5* 3 0 0 1
90	0 0 0 0 6* 2 1 0 1
91	0 0 0 5* 2 2 0 1
92	0 0 0 5* 1 3 0 1
93	0 0 0 4* 1 4 0 1
94	0 0 0 2* 4 3 0 0 1
95	0 0 0 1 4* 4 0 0 1
96	0 0 0 0 6* 1 2 0 1
97	0 0 0 0 4* 3 2 0 1
98	0 0 0 0 3* 3 3 0 1
99	0 0 0 0 4* 1 4 0 0 1
100	0 0 0 0 3* 1 5 0 0 1

Table 7.1.1 Optimal distribution of trials, $b = 10$, $t_{max} = 10$, $depth = 3$.

(1) For successive values of n, the parameters of the optimization problem change by small amounts. Hence, drastic changes in the resulting policy are unlikely, and it should be possible to identify a pattern. For example, table 7.1.1 lists the optimal policies for a page size of 10 and t_{max} of 10. All the results were obtained with $depth = 3$. The starred entries correspond to $\lfloor E(m) \rfloor$ pages, where $E(m)$ is the expected number of pages resulting

from the policy. The first entry of each policy corresponds to the number of trials with $\lceil n/b \rceil$ pages, the second entry to $\lceil n/b \rceil + 1$ pages and so on.

(2) Figure 7.1.1 is a plot of the expected number of pages in a group as a function of the number of keys when the group is built incrementally. The optimal rehashing policy is followed whenever a new perfect hashing function must be determined. We see that the graphs are smooth except at the beginning (due to fragmentation). By storing an Em-table with a few entries, such as table 7.1.2 (table for $b = 10$, $t = 10$), the expected number of pages required to store a given number of keys can be easily obtained by interpolation.

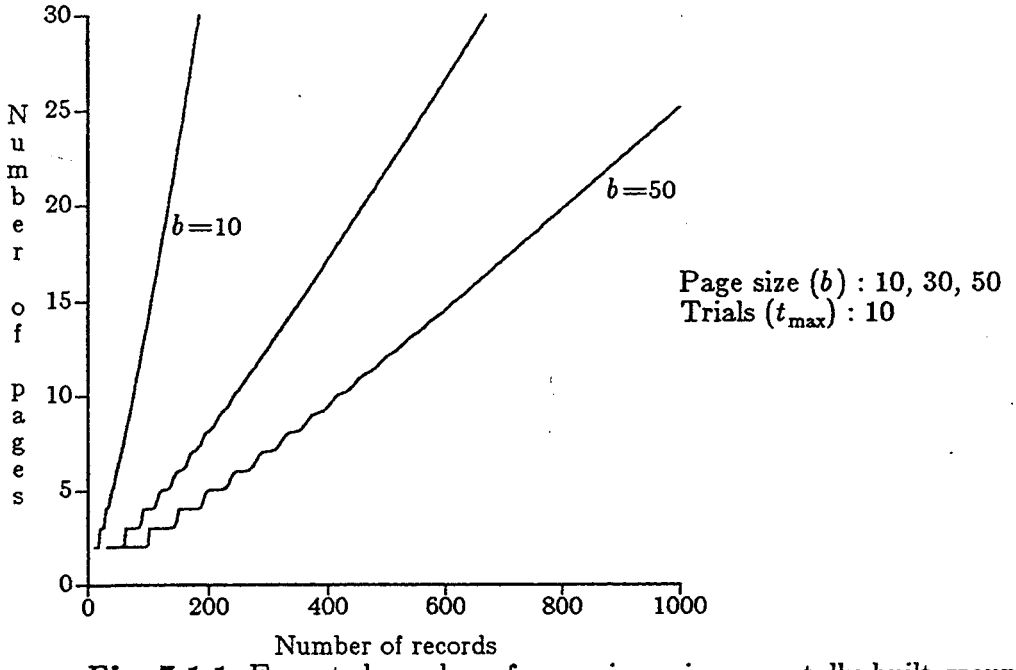


Fig. 7.1.1 Expected number of pages in an incrementally built group

No. of keys (n)	Expected no. of pages $E(m)$
28	3.15
31	4.03
35	4.13
40	5.05
44	5.46
50	6.36
75	10.41
100	14.40
150	23.33
200	33.00
250	41.87

Table 7.1.2 Em-table for $b = 10$, $t_{max} = 10$ (expected number of pages in a group).

(3) The distribution of the number of trials under the optimal policy appears to be centered around the expected number of pages; the majority of trials are made around the resulting expected number of pages, see table 7.1.1. For example, with a page size of 10 and t_{max} of 10, when the number of keys (n) is between 100 and 130, the optimal policy can be approximated as follows (assuming that the resulting load factor $E(m)$ is known a priori):

make 4 trials with m equal to $E(m)$† pages,
make 3 trials with m equal to $E(m) + 1$ pages,
make 1 trial with m equal to $E(m) + 2$ pages,
make 2 trials with m equal to $m_h - 1$ pages.

Recall that m_h corresponds to a load factor of about 50% ($P(n, m_h, b)$ is very close to 1.0). Similar approximations to the optimal policy can be found for different ranges of n and stored in a policy-table. Table 7.1.3 is a policy-table for $b = 10$. Each entry in the table is expressed as a fraction of t_{max}. Each row approximates the optimal policy for a range of n between the entries in the first column of that row and the next row. The example given for $100 \leq n < 130$, is represented by row 3 in the table.

Number of of keys (n)	$E(m)-1$	$E(m)$	$E(m)+1$	$E(m)+2$	$E(m)+3$	m_h-1	m_h
30	0.0	0.8	0.1	0.1	0.0	0.0	0.0
70	0.0	0.6	0.3	0.0	0.0	0.1	0.0
100	0.0	0.4	0.3	0.1	0.0	0.2	0.0
130	0.0	0.2	0.4	0.2	0.0	0.1	0.1
150	0.0	0.2	0.3	0.1	0.2	0.0	0.2
180	0.0	0.0	0.2	0.3	0.2	0.0	0.3
250	0.0	0.0	0.0	0.0	0.0	0.5	0.5

Table 7.1.3 Policy-table for $b = 10$.

From tables similar to 7.1.2 and 7.1.3 it is straightforward to obtain an approximation to the optimal policy for a given value of n and t_{max}.

Heuristic to approximate the optimal policy

Given n, use the Em-table to estimate the expected number of pages, $E(m)$, corresponding to the t_{max} specified. Distribute the t_{max} trials using the Policy-table over the range $E(m) - 1$ to m_h, where $E(m)$ is the interpolated value of the expected number of pages and m_h is the maximum number of pages.

Given an Em-table and a Policy-table it is straightforward to arrive at a distribution of trials using the above procedure. Our aim while constructing the tables was to keep them small and yet obtain a performance as close as possible to that of the optimal policy. Tables for various values of b and t_{max} are given in

† If $E(m)$ is not an integer, the trials are divided between the two group sizes $\lfloor E(m) \rfloor$ and $\lceil E(m) \rceil$ according to the ratio $(E(m)-\lfloor E(m) \rfloor)/(\lceil E(m) \rceil - E(m))$, and similarly for $E(m)+1$, and so on.

Appendix B.

7.1.2. Performance of the heuristic policy

Determining the heuristic policy requires computation and memory space to store the tables. The computational cost is very low: linear interpolation of the expected number of pages, and computing the number of trials using the approximate fractions from the policy-table. These computations are independent of t_{max} and the number of pages, and the cost is negligible. The memory space required to store the tables is also small. Around 100 words are required to store them in the indicated format. It is possible to encode the tables to reduce their size. For example, the entries of the policy-table could be stored as 8 bit numbers by storing (actual fraction) * 256.

The heuristic is based on the solutions of the optimization problem. The goal is to give a policy which performs close to that of the optimal policy. We compare the performance of the heuristic policy with that of the optimal policy in terms of the resulting load factor, probability of rehashing, etc.

Figure 7.1.2 compares the heuristic policy with that of the optimal policy with respect to the load factor of an incrementally built group and the expected number of trials required to find a perfect hashing function for a group. The page size is 30 and t_{max} is 10. The solid line corresponds to the results obtained using the heuristic policy. The tables for $b = 30$ are given in Appendix B. The load factor and the expected number of trials required to find a perfect hashing function using this policy were computed as described in chapter 5. The dotted line corresponds to the results of using the optimal policy (i.e., the optimization problem is formulated and solved to obtain the optimal policy of distributing the trials).

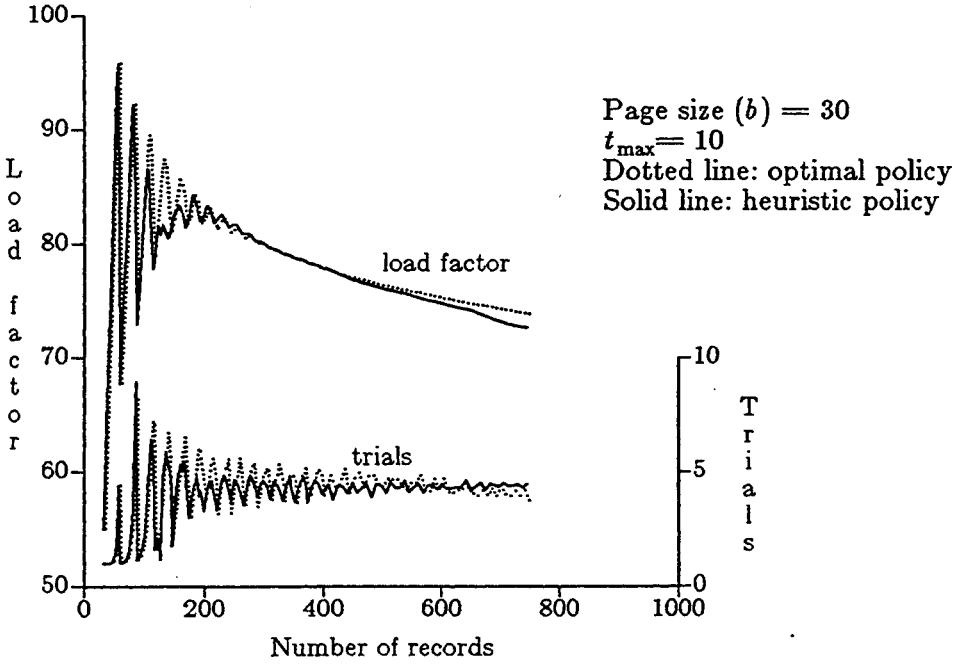


Fig. 7.1.2 Load factor and expected number of trials of an incrementally built group

We observe that the load factor resulting from the heuristic policy is very close to that of the optimal policy. In certain intervals the heuristic policy seems to outperform the optimal policy. This is only partly true, because often the higher load factor is associated with an increased value of the expected number of trials and/or with a decreased probability of succeeding within t_{max} trials. We have observed that, occasionally, the heuristic policy outperforms the optimal policy in all three aspects: load factor, expected number of trials and the probability of success in t_{max} trials. There are two possible reasons for this anomaly. First, the optimal policy we are determining is not necessarily the true optimum but rather a close approximation. The heuristic policy may be closer to the true optimum than the approximate solution. The second reason, which is more likely, is as follows. Optimal policy corresponds to optimality at a particular instant; the optimization problem was formulated to yield as high a load factor as possible locally (i.e., the load factor after rehashing has been carried out). Following the optimal policy at every stage, while inserting the nth key and the $(n+1)$st key may not be optimal for the combined operation. In other words, a non-optimal policy for the nth insertion and $(n+1)$st insertion may turn out to be optimal with respect to the end result (i.e., the load factor of the group with $(n+1)$ keys). The sharp peaks in the curve with the optimal policy correspond to a high expected number of trials. This extra work does not translate into an improved load factor (as compared with the heuristic policy). This is illustrated in table 7.1.4.

depth	1		3	
Number of keys, n	52	53	52	53
Load factor after rehashing, $RHE(\alpha)$	74.46	74.62	74.69	74.73
Overall load factor, $E(\alpha)$	77.90	77.47	77.91	77.42
Expected number of trials, $E(t)$	4.99	3.35	5.05	4.15
Probability of success in t_{max} trials	.9901	.9926	.9907	.9901
Probability of rehashing, $RH(\alpha)$	.2218	.2211	.2259	.2248
Rehashing policy	3 4 3 0 0	0 8 1 1 0	3 5 1 1 0	1 7 1 1 0

Table 7.1.4 An example with $b = 10$ and $t_{max} = 10$.

Table 7.1.4 was extracted from the results of computations for an incrementally built group with $b = 10$ and $t_{max} = 10$. There are two sets of results corresponding to the value of *depth* used in the solution of the optimization problem. The rehashing policies obtained with *depth* = 3 are exact optimal policies (verified by exhaustive searching). When n is 52, the optimal rehashing policy (3,5,1,1,0) yields a load factor of 77.91%; only slightly better than the 77.90% that can be obtained using the policy (3,4,3,0,0). Note that $RHE(\alpha)$ represents the load factor after rehashing using the policy, whereas $E(\alpha)$ represents the load factor taking into account the probability of rehashing. $RHE(\alpha)$ values

corresponding to *depth* = 3 are always higher than that for *depth* = 1. However, with *depth* = 3 and $n = 53$ the optimal policy yields an $E(\alpha)$ of 77.42% which is lower than that obtained with *depth* = 1 (77.47%).

The optimal policies corresponding to *depth* = 3 involve more trials with lower number of pages. This leads to "stiffer" policies: i.e., a higher value of $RHE(\alpha)$, but at the cost of higher $E(t)$ and $RH(n)$ (a lower value of $RH(n)$ implies a higher probability of the next insertion increasing the load factor). The overall load factor $E(\alpha)$ is not necessarily higher.

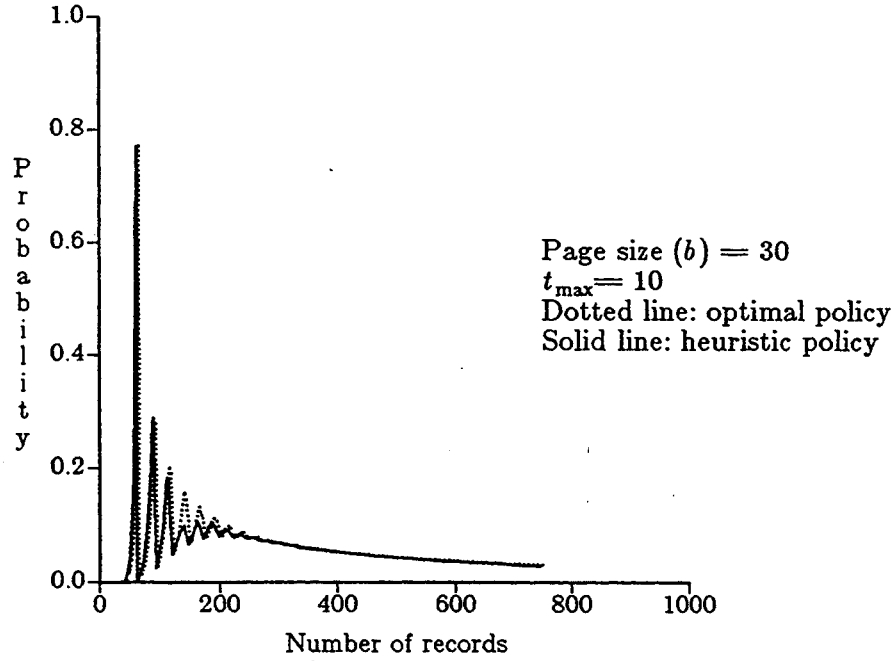


Fig. 7.1.3 Probability of an insertion causing rehashing of a group

Figure 7.1.3 plots the probability of an insertion causing rehashing as a function of the number of keys in the group. Figures 7.1.4 to 7.1.7 are similar plots for different values of b. In all cases, we observe that the performance of the heuristic policy is close to that of the optimal policy.

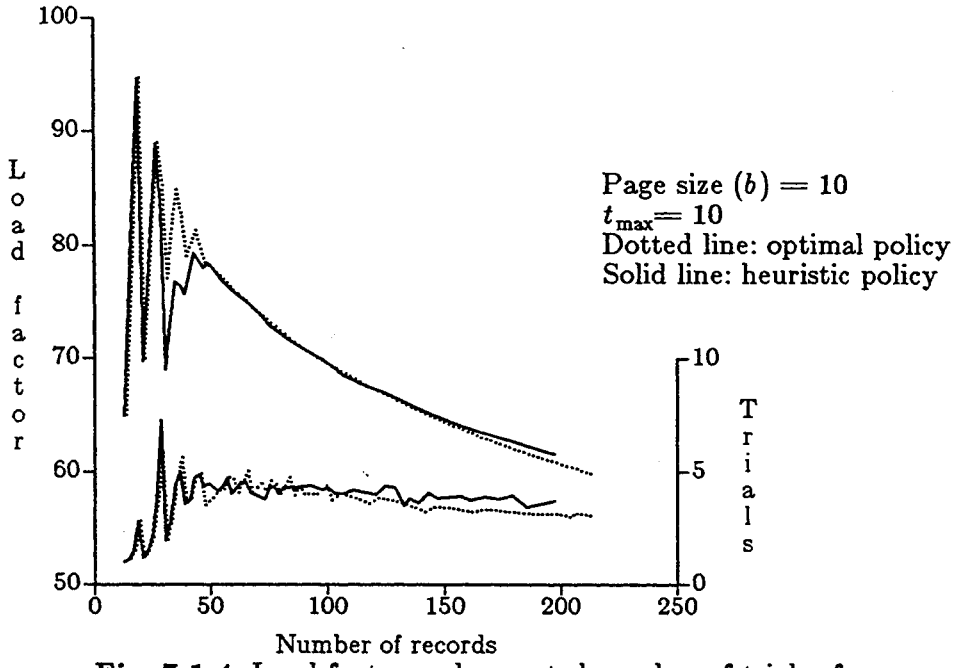


Fig. 7.1.4 Load factor and expected number of trials of an incrementally built group

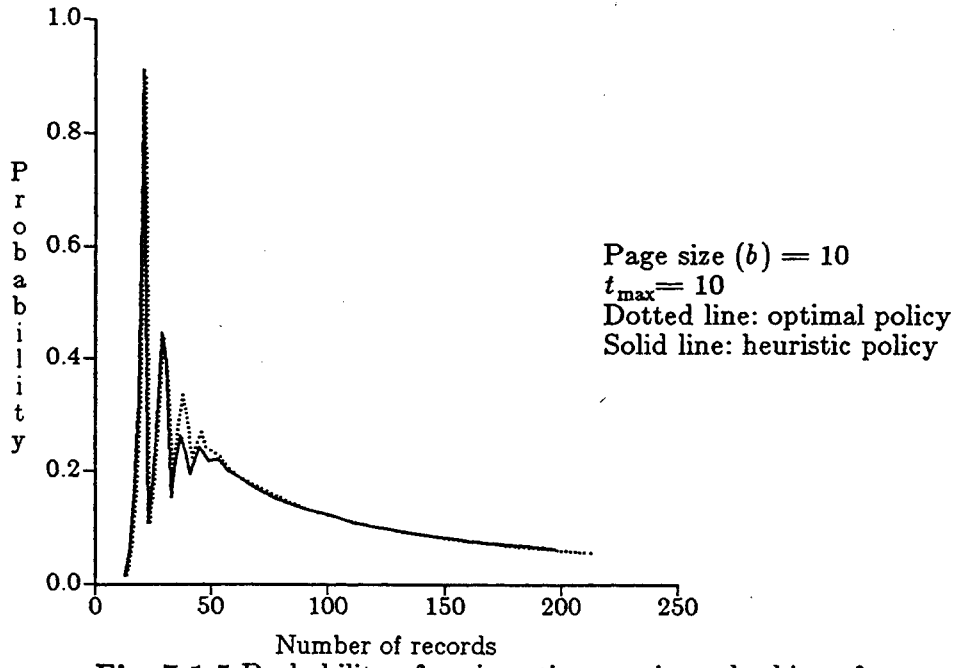


Fig. 7.1.5 Probability of an insertion causing rehashing of a group

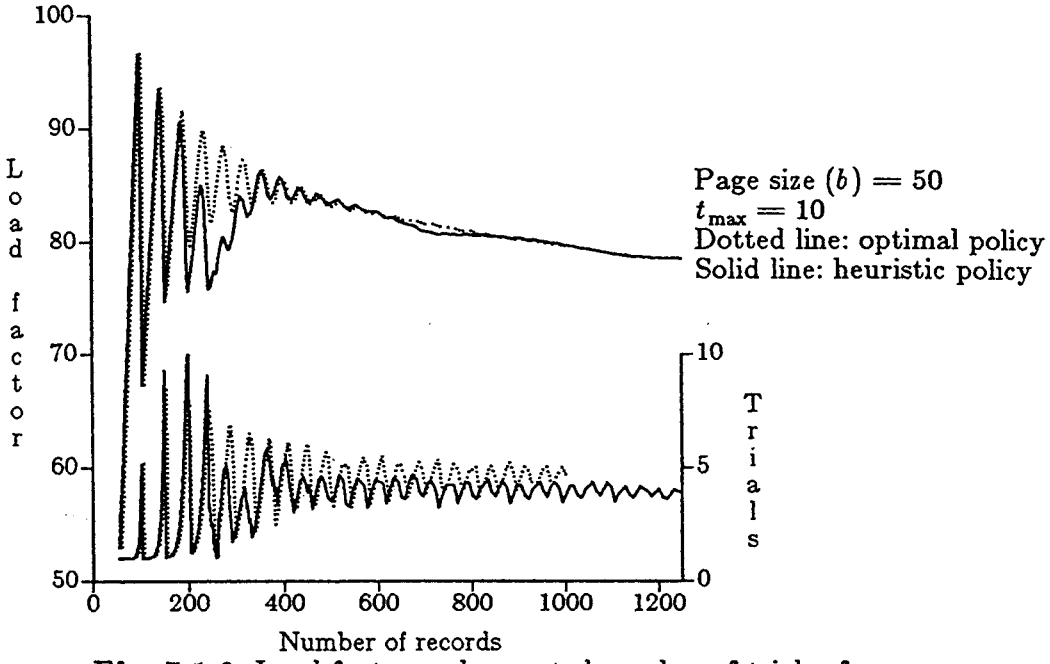


Fig. 7.1.6 Load factor and expected number of trials of an incrementally built group

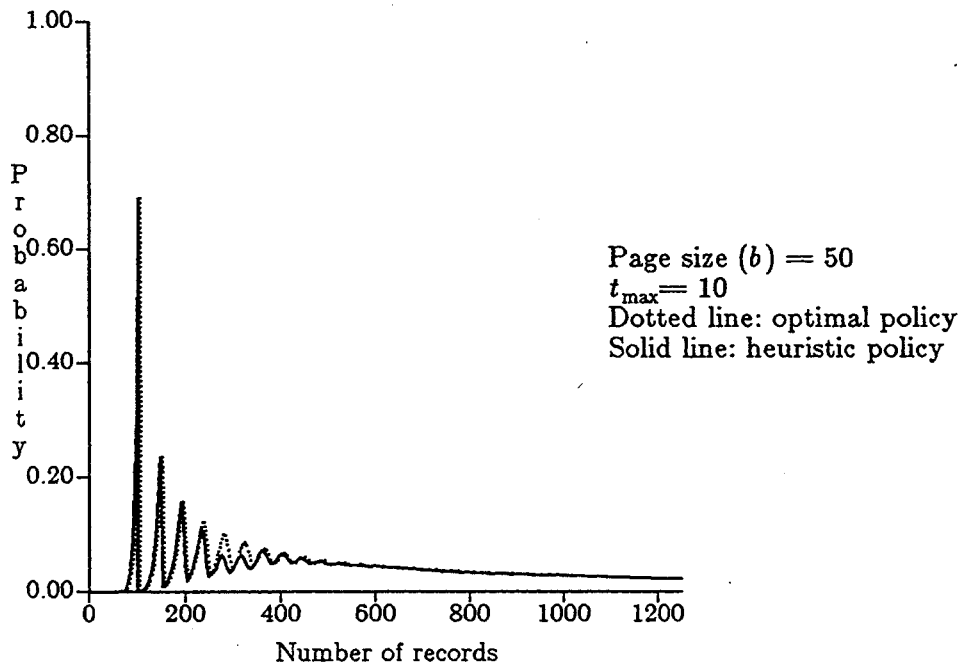


Fig. 7.1.7 Probability of an insertion causing rehashing of a group

Tuning the heuristic

As already explained, the *Em*-table and the Policy-table for the heuristic were constructed based on the results of the optimal policy. In practice, the parameters of the tables may be changed dynamically depending on the requirements and the performance of the particular class of hashing functions chosen, heuristic, etc. For example, when the load on the system is high, we may be

willing to settle for lower load factors. This can be achieved by reducing the fraction of trials with higher load factors (and thus decreasing the average number of trials required to find a perfect hashing function).

7.2. Performing a rehash

We have seen in chapter 5 that the probability of an insertion into a group causing a rehash is quite small (typically around 5%). When an insertion does not involve rehashing, the cost is minimal: insert the record at the right place within the page buffer and write back the buffer onto secondary storage. (The page would have been read into memory already.) However, the cost is much higher when an insertion triggers a rehash operation. This section deals with the tradeoff between the amount of buffer space and I/O cost of rehashing.

Minimizing the I/O cost

Consider a group consisting of n records distributed over m_1 pages of size b by a perfect hashing function h_1. Let the insertion of the $(n+1)$st key into the group cause it to overflow, thus triggering a rehash of the group. This involves reading in m_1 pages, finding a new perfect hashing function h_2 with m_2 pages for the $(n+1)$ records and writing back m_2 pages onto disk. If $\max(m_1, m_2)$ pages of buffer space is available, rehashing can be accomplished at the minimum cost of reading m_1 consecutive pages followed by in-buffer redistribution of records and then writing m_2 consecutive pages. Note that reading/writing a set of consecutive pages is not much more expensive than accessing a single page, and hence for all practical purposes reading/writing a whole group can be considered a single disk access (this is not entirely correct, see section 8.1).

Redistribution of records in place

Assume that m_1 pages (containing the records to be redistributed) have been read into m_1 buffers of a total of $\max(m_1, m_2)$ buffers available. Select the first key x in the first page(buffer) to be relocated. Compute the new hash address $m_x = h_2(x)$. If page m_x is not full the key is moved into an empty slot on that page. If the page m_x is already full, an arbitrary key is chosen from that page for relocation and the key x is moved into the empty slot created. This relocation chain ends whenever a nonfull page is encountered. Next, choose another key from page 1 for relocation and so on until the last key in page m_1 has been relocated. Note that during this process some keys may already be in their right place. This redistribution involves moving at most n records.

Reducing the buffer space

If we cannot afford $\max(m_1, m_2)$ buffers, redistribution can be accomplished at an increased I/O and processing cost. First of all, all the m_1 pages have to be read to extract the keys and a new perfect hashing function h_2 with m_2 pages for the $(n+1)$ keys needs to be found. We assume that at least all the extracted keys can be kept in core. Redistribution can then be accomplished using $k+1$ buffers ($k+1 < \max(m_1, m_2)$) by repeated partitioning as follows.

The idea is to initially partition the group into k subgroups, 0, 1, 2 , . . . , $k-1$, such that the records which would hash to the first $\lceil m_2/k \rceil$ pages of the group are in subgroup 0, those which would hash into the next $\lceil m_2/k \rceil$ pages of the group are in subgroup 1 and so on. More precisely, if x denotes a key of the group, then subgroup $i = \{x \mid i \lceil m_2/k \rceil \leq h_2(x) < (i+1)\lceil m_2/k \rceil\}$.

Designate one of the $(k+1)$ buffers as a read buffer, and the other k buffers as write buffers, 0, 1 , . . . , $k-1$. Read in the m_1 pages one at a time into the read-buffer. For each key in the read buffer, compute the subgroup to which it belongs, and move the record to the corresponding write buffer. Whenever the read buffer is empty, read in the next page until all the m_1 pages have been read. Whenever a write buffer is full, write it onto secondary storage. When all the m_1 pages have been processed, we have k subgroups each having $\lceil m_2/k \rceil$ pages. Subgroup i, $0 \leq i \leq k-1$, contains all and only the records belonging to pages $i\lceil m_2/k \rceil$, $i\lceil m_2/k \rceil + 1$, . . . , $(i+1)\lceil m_2/k \rceil - 1$ (with reference to the final desired distribution in m_2 pages). This first partitioning involves reading m_1 pages, moving n records and writing at most m_2 pages.

If $\lceil m_2/k \rceil \leq (k+1)$ then redistribution within each subgroup can be accomplished as described above. Otherwise, each of the subgroups may be treated as a separate problem with $\lceil m_2/k \rceil$ pages. If $m = m_1 = m_2$, each partitioning involves reading and writing m (at most $\lceil m/k \rceil * k$, to be precise) pages and moving n records. The problem size is reduced by a factor of k from m to $\lceil m/k \rceil$. Since problems of size $\leq (k+1)$ can be handled without further partitioning, we conclude that redistribution can be achieved in $\lceil \log_k m \rceil$ stages of partitioning, at a total cost of reading and writing $m\lceil \log_k m \rceil$ pages and moving $n\lceil \log_k m \rceil$ records among the buffers. (Note that, the initial accesses to extract the keys are not included.) In particular, if k is chosen to be $\lceil \sqrt{m} \rceil$ only two stages of partitioning are required. The first stage involves reading and writing m pages requiring $2m$ accesses to the disk. The second stage involves reading and writing $\lceil \sqrt{m} \rceil$ subgroups each having $\lceil \sqrt{m} \rceil$ consecutive pages requiring $2\lceil \sqrt{m} \rceil$ accesses. The total cost is $2(m+\lceil \sqrt{m} \rceil)$ accesses to the disk and moving at most $2n$ records. In practice m ranges from 10 to 50 and hence, reducing the value of k below $\sqrt{m}$ (4 to 7) does not seem necessary.

7.3. Header table format

In chapter 3, the header table entries were represented as $< p_t, m_t, R_t >$, where p_t is a pointer to the page group, m_t is the number of pages in the group, and R_t the set of parameters of the perfect hashing function. In chapter 6 we have seen that perfect hashing functions of the form $h(x) = (cx \bmod prime) \bmod m$ can be found by fixing the value of *prime* and choosing the value of c at random (using a pseudo random number generator). If the same *prime* is used for all the groups, R_t need only consist of the single parameter c.

For example, consider a file of one million records stored by external perfect hashing. Assuming a page size of 40 and a load factor of 80%, about 32,000 pages are required to store the file. If page addresses relative to the beginning of the file are used, 15 bits (or 2 bytes) are required to store p_t. Since the group

size would not exceed about 50 pages, 6 bits (or 1 byte) is adequate to store m. Assuming that keys(after conversion into integers) are 4 bytes long, 4 bytes are required to store c. Hence each entry of the header table requires about 7 bytes of storage. If the average group size is chosen to be 1,000 corresponding to 1,000 groups in the file, the header table needs 7 kilobytes of memory.

To find a perfect hashing function for a group we try no more than t_{max} (10 to 20) hashing functions. During the trials, instead of always generating "new" hashing functions, we may choose hashing functions which have already proven perfect for other groups. In other words, if a file consists of one thousand groups, the thousand parameters R_t need not be distinct and storing fewer may suffice. A set of hashing functions could be generated a priori and stored in a separate table, called the H-table. Then the entry R_t in the header table need only be an index to the H-table and hence fewer bits are needed to store the header table entries. In our above example, if the H-table has less than 256 entries, one byte is sufficient to store an index to the H-table. Each header table entry need only be 4 bytes long. This results in a saving of $7 * 1000 - (4 * 1000 + 256 * 4) = 2000$ bytes of memory space.

If the header table is organized as a B-tree, the space requirement to store $< p_t, m_t, R_t >$ is the same as described above. In addition, roughly speaking, space is required to store one key (or its prefix) per group in the file. Here it is assumed that the tree is very shallow; for example the tree may have a root node and $\sqrt{s}$ leaf nodes each having $\sqrt{s}$ entries for a total of s groups in the file.

7.4. Dynamic external perfect hashing

A hashing scheme is said to be dynamic if it can accommodate wide variations of the file size, without significant deterioration of the retrieval performance and storage utilization. Any file organization scheme can accommodate wide variations of file size by periodically reorganizing the whole file (maintaining the storage utilization and performance). Dynamic hashing schemes accommodate file size fluctuations without complete reorganizations of the file.

The load factor of a file organized using the external perfect hashing scheme varies slightly with the number of records per group. As the file grows or shrinks, the load factor decreases or increases continuously and gradually. A file may double in size causing only a small percentage drop in the storage utilization and it may shrink to a considerable extent without causing the storage utilization to decrease. However, if the file becomes too small, the load factor starts to decrease because of the fragmentation problem. When the file becomes too large, even if the reduced storage utilization is acceptable, rehashing of large groups poses practical problems.

If the header table is organized as a B-tree, the scheme is completely dynamic. In addition, the scheme enables efficient range searches and extra large groups (discussed in section 7.5) cannot occur. However, the B-tree based scheme requires more internal memory (approximately twice as much as that for the hash table based scheme, it depends on the length of the keys stored).

If the header table is organized as a traditional hash table, the file size cannot be allowed to grow or shrink by large factors. However, several of the dynamic hashing schemes proposed during the last few years can be adapted to

organize the header table so that the external perfect hashing scheme is completely dynamic. The external perfect hashing scheme can be considered as a hash table having elastic buckets (i.e., each group is a bucket of flexible size). With this view of the scheme, the linear hashing scheme proposed by Litwin[LT80], dynamic hashing proposed by Larson[LR78], or extendible hashing proposed by Fagin et. al. [FN79] can easily be adapted to organize the header table so that the overall hashing scheme is fully dynamic. An important point to remember is that a group need be split into two only when an insertion into the group causes rehashing (the elastic bucket "overflows" only when an insertion into the group causes rehashing).

7.5. Handling large groups

The worst case of hashing, when all the keys hash to the same address, is highly improbable. A less severe event that may arise in practice is when a few of the groups receive an exceptionally large number of keys as compared to the average group size. The adverse effect of a few large groups on the overall storage utilization is minimal. However, rehashing such large groups may pose practical problems. We propose a simple solution to handle such situations.

When rehashing a group, if it contains more than a certain predetermined number of keys, the group is split into two or more subgroups. The original entry in the header table for the group $< p_t, m_t, R_t >$ is replaced by $< p_1, m_1, R_1 >$ where p_1 is a pointer to a subordinate header table which is similar to the main header table and supports m_1 subgroups. R_1 are the parameters of the subgrouping function. Header table entries may be suitably encoded to distinguish those which have subordinate tables. For example, if the subgrouping functions are chosen from the class H_1, which require a single parameter, then a header table entry having $R_1 = c_1$ could correspond to a group having a subordinate header table if $c_1 <$ climit.

The problem of large groups does not arise if the header table is organized as a B-tree.

7.6. Conclusions

This chapter addressed several practical problems encountered in the implementation of the external perfect hashing scheme. The main practical problem addressed was the complexity of solving the optimization problem to obtain the optimal policy of distributing the trials. Although the solution procedure presented in chapter 4 is an efficient method of solving the nonlinear integer optimization problem, it is too costly in practice. In this chapter we presented a simple heuristic to obtain a policy for distributing the trials. The cost of the heuristic is small and its performance is close to that of the optimal policy.

In the previous chapters, we mentioned that rehashing is an expensive operation. In this chapter, we described a method of carrying out a rehash with minimum disk I/O. We also outlined a method of redistributing records using less buffer space but at an increased I/O cost.

In chapter 2 we gave the symbolic format of the header table entries. As a consequence of the results in subsequent chapters, we were now able to give exact details of the header table entries. We also suggested a minor modification

to the basic external perfect hashing scheme to handle the situation when a few of the groups are exceptionally large. The same idea can be extended to make the scheme completely dynamic: it is similar to "dynamic hashing" proposed by Larson [LR78]. In fact, most of the dynamic hashing schemes can be adapted to make the external perfect hashing scheme completely dynamic.

Chapter 8

Conclusions and further work

8.0. Chapter overview

In this chapter we summarize the performance of the proposed external perfect hashing scheme. The cost of insertions and internal memory requirements are compared with that of other hashing schemes. Several open problems and directions for future research are also outlined.

8.1. Comparison with other hashing schemes

In addition to optimal retrieval performance, the proposed external perfect hashing scheme has several other advantages over other traditional hashing schemes. In view of the differences between the various hashing schemes, it is difficult to make a strict comparison of the costs involved. The main costs associated with the external perfect hashing scheme are for insertions and internal memory for storing the header table. We compare the new scheme with a method called "External Hashing with Limited Internal Storage" (denoted by Eh-LIST) [GL82,LK84], which also guarantees single access retrieval. The insertion costs are also compared with those of double hashing under the uniform hashing model [LR83].

Load factor	Double hashing	External perfect hashing				External hashing with limited internal storage		
percent	accesses	hash ads	accesses	memory	t_{max}	accesses	memory	sepr
60	.099	80	.065	20 kb	20	.226	83 kb	4
		30	.055	24 kb	10	.193	125 kb	6
70	.243	95	.125	40 kb	20	.552	71 kb	4
		40	.129	55 kb	10	.467	107 kb	6
80	.913	125	.255	90 kb	20	1.344	62 kb	4
		40	.259	120 kb	10	1.079	94 kb	6

Table 8.1.1 Comparison of hashing schemes for $b = 10$.

Load factor	Double hashing	External perfect hashing				External hashing with limited internal storage		
percent	accesses	hash ads	accesses	memory	t_{max}	accesses	memory	sepr
76.5	.042	-	-	-	-	.128	13 kb	4
		35	.016	4.2 kb	5	.090	19.6 kb	6
80	.081	270	.030	4.2 kb	20	.242	12.5 kb	4
		100	.030	5.4 kb	10	.137	18.8 kb	6
85	.192	300	.061	8.4 kb	20	.582	11.8 kb	4
		107	.065	14.0 kb	10	.408	17.6 kb	6

Table 8.1.2 Comparison of hashing schemes for $b = 50$.

Table 8.1.1 summarizes the cost of the three hashing schemes for a file of one million records and page size 10. Table 8.1.2 corresponds to a page size of 50. The tables show the average cost per insertion and amount of internal memory required, if any. Under any file organization scheme a minimum of two disk accesses are required for insertions: one read access followed by one write access. The I/O cost of insertions given under the heading "accesses" represents the average number of extra read/write accesses required for insertion, that is, accesses above the minimum 2.0. Insertion in double hashing involves an unsuccessful search, followed by a write access, and hence the number of accesses were

computed using the formulae given in [LR83] for the expected length of an unsuccessful search.

For the external perfect hashing scheme, the column labeled "hash ads" represents the expected number of hash address evaluations per insertion. This depends on t_{max}. When an insertion does not cause rehashing, only two disk accesses are required. When an insertion does cause rehashing, three accesses are required: initial reading of a single page, then reading the whole group, followed by writing the whole group. This is under the assumption that enough internal buffer space is available to accommodate all the records of the group. We also assume that the cost of accessing a whole group of contiguous pages is roughly the same as that of accessing a single page†. If less buffer space is available, the I/O cost will be correspondingly higher. "memory" represents the internal storage space required for the header table when organized as a hash table. If the header table is organized as a B-tree the amount of memory required will be higher. The perfect hashing functions are assumed to be stored in a separate table and header table entries have indices to the perfect hashing functions. Each entry of the header table is assumed to be 5 bytes long when $b = 10$ (3+1+1 bytes for $< p_t, m_t, R_t >$) and 4 bytes long when $b = 50$. The size of the table which stores the randomly generated parameters of the perfect hashing functions is fixed at 1 kilobytes. The table lists two sets of values at each load factor corresponding to two different values of t_{max}.

For Eh-LIST "sepr" represents separator length in number of bits per page. The expected number of extra accesses per insertion was computed using the results in [GL82]. For each load factor, there are two sets of values corresponding to separator lengths of 4 and 6 bits.

It is surprising that the expected I/O cost for the external perfect hashing scheme is less than that for double hashing. At higher load factors, the number of extra accesses per insertion for the external perfect hashing scheme is about one third of that for double hashing.

Eh-LIST is a good candidate for comparison with the external perfect hashing scheme. Both guarantee single access retrieval and both have internal, memory requirements in addition to increased insertion costs. We observe from the tables that the external perfect hashing scheme outperforms Eh-LIST for the range of load factors and page sizes covered. Under Eh-LIST, for a given separator length, the total amount of internal memory required is almost independent of the load factor. For the external perfect hashing scheme, the amount of internal memory required is very sensitive to the load factor. We observe that the expected number of extra accesses per insertion for Eh-LIST is 4 to 8 times that for the external perfect hashing scheme, over the range of load factors and page sizes considered. At high load factors, the internal memory required is roughly the same for both schemes, but at lower load factors the external perfect hashing scheme needs much less memory (a factor of 4-5).

---

† This is not entirely true, of course. The relative cost of accessing a group and a single page, depends on the characteristics of the secondary storage medium, the group size, and the size of individual records. Later in this section, we discuss the relative costs with reference to disks.

There are a number of assumptions involved in this comparison. The main assumption is regarding the relative cost of accessing a group and a single page. When the external medium is a disk, the cost measure is the total delay involved in reading or writing (seek time + rotational latency + transfer time). For the sake of simplicity let us assume that the access time for a group is approximately twice the access time of a single page†. If the cost of accessing a single page is unity, then the I/O cost of rehashing is 5 units: 1 unit for initial reading of a single page, 2 units for reading the entire group, and 2 units for writing back the whole group after redistributing the records. That is, the extra accesses required is $5 - 2 = 3$. This implies that, the number of extra accesses for the external perfect hashing scheme given in column 4 of tables 8.1.1 and 8.1.2 should be multiplied by 3. For example, when the page size is 10 and the load factor is 70%, the number of extra accesses required for double hashing is 0.243 per insertion. External perfect hashing needs time corresponding to $0.125 * 3.0 = 0.375$ extra accesses per insertion, which is less than the 0.552 required for Eh-LIST. When the page size is 50, at 80% load factor double hashing needs 0.081 extra accesses per insertion. Eh-LIST needs 0.242 extra accesses per insertion, much higher than $0.030 * 3 = 0.090$ required for external perfect hashing. Thus, the main conclusions drawn from tables 8.1.1 and 8.1.2 regarding the relative I/O costs of Eh-LIST and external perfect hashing do not change. However, the I/O cost of double hashing is now slightly less than that of external perfect hashing.

Using the external perfect hashing scheme, a file of one million records can be stored at a load factor of 80% using about 6 kilobytes of internal memory, when the page size is 50. An average insertion involves about 100 hash address evaluations and 2.03 disk accesses. Only 3% of the insertions require temporary buffer space for about 900 records. The same file requires 19 kilobytes of internal memory if stored under Eh-LIST with 6 bit separators, and the expected cost of an insertion is 2.137 accesses.

When the header table is organized as a B-tree, the internal storage requirement of the external perfect hashing scheme will be higher (the entries in column 5 of tables 8.1.1 and 8.1.2 need to be multiplied by a factor of 2, roughly). The resulting organization is completely dynamic and permits efficient range searching.

---

† For the range of group sizes we are dealing with this is not unrealistic. If the average access time of the disk is 30 milliseconds and the transfer rate is 3 kilobytes per millisecond, one disk access corresponds to transferring 90 kilobytes. A group having 500 records, each record being 360 bytes long, can be accessed in $2 * 30$ milliseconds.

8.2. Contributions of the thesis

This thesis represents the first attempt at applying perfect hashing to large external files. The primary goal was to determine if and how large external files could be organized using perfect hashing, and whether this would be practical. The main contribution has been to show that perfect hashing is indeed practical for organizing large external files. Also, the proposed external perfect hashing scheme is competitive with other hashing schemes.

The recurrence relation for computing $P(n,m,b)$ was of crucial importance for this thesis. Davis and Barton [DB62, BD59] clearly state that there is no closed form expression for $P(n, m, b)$ simpler than the one given by (3.2). This expression is not useful for computational purposes and we have not found any reference to better algorithms. The algorithm given in chapter 3 to compute $P(n, m, b)$ is simple and requires only a few seconds of CPU time for the range of m and b we are interested in. For larger values of m and b, we derived an approximation for $P(n, m, b)$. The computation of $P(n, m, b)$ is required in other areas of computational probability theory. An example is the *Table Sufficiency Index* computations in [NY85], where Norton and Yeager were able to obtain only approximate values of $P(n, m, b)$ using a complicated procedure.

We could not prove the existence of a perfect hashing function within the class H_1, for a given set of keys. In view of this, the idea of limiting the cost of finding perfect hashing functions in chapter 4 is especially important. We have analyzed one particular optimality criterion, but the approach is general and can be applied to other criteria as well. This technique of limiting the cost may also be applied to other probabilistic procedures†.

Chapter 6 completes the answer to one of the main questions posed in chapter 2: How to find perfect hashing functions? The results of the experiments presented indicate that the relative frequency of perfect hashing functions within the class H_1 is the same as that predicted by the analysis in chapter 3. This implies that the performance characteristics of the external perfect hashing scheme as presented in chapter 5 can be attained in practice. Although there are good reasons for choosing the class H_1, many questions regarding perfect hashing functions remain open.

The results of the comparison of various hashing schemes, presented in the previous section of this chapter, are surprising. The proposed external perfect hashing scheme is not only practical but it can also compete with other hashing schemes. In short, the main contribution of the thesis is to show that perfect hashing is a practical and competitive technique for organizing large external files.

---

† A real life example: How should a student with limited funds, distribute applications among graduate schools of various quality? The aim is of course to get into the best possible school.

8.3. Directions for future work and open problems

In this final section we discuss various open problems and plans for future research.

1. Implementation

We have not made a complete implementation of the external perfect hashing scheme. All the results of the performance analysis in chapter 5 depend solely on the values of $P(n, m, b)$. The results of the experiments discussed in chapter 6 show that the performance of the suggested class of hashing functions is the same as predicted by the analysis of $P(n, m, b)$ in chapter 3. In addition, we have proposed solutions to several other implementation problems. However, a full implementation of the proposed external perfect hashing scheme has not been done and is left as future work. Both Eh-LIST and the external perfect hashing scheme should be implemented in order to compare their relative performance and difficulty of implementation.

2. Rehashing policies

In chapter 4 we discussed the main implications of a rehashing policy. We chose to define an optimal rehashing policy which minimizes the expected number of pages while limiting the cost of rehashing. It is possible to define other optimality criteria and study the performance under the policies so obtained. However, it seems that unless the optimality criterion is extremely simple, some form of heuristic to compute the optimal policy will be necessary in practice.

Intuitively, it appears that the size of the policy-table (discussed in chapter 7) could be reduced further. It would be interesting to study the result of a drastic reduction in the size of this table. For example, consider the following policy of distributing the trials: make 50% of the trials with $\lfloor E(m) \rfloor$ pages, 40% of the trials with $\lfloor E(m) + 1 \rfloor$ pages and the rest with $\lfloor E(m) + (5 - b/10) \rfloor$ pages, where $E(m)$ is the expected number of pages under the optimal rehashing policy obtained from the Em-table.

3. Classes of hashing functions

We have experimentally shown that the relative frequency of perfect hashing functions in the class H_1 is statistically the same as that predicted by the theoretical analysis for the set of all functions. However, there are many open questions regarding perfect hashing functions:

a) Are there other classes of simple hashing functions which behave similarly? What are the important characteristics of such a class of functions?

b) Is $universal_2$ a necessary and/or sufficient condition?

c) Given a set of keys, does a perfect hashing function always exist in the class H_1? Under what conditions? What effect has the load factor?

d) What are the characteristics of the sets, if any exist, for which no perfect hashing function exist in a given class?

The results of Mairson may be applicable in this regard[MR83, MR84].

4. Other methods of finding perfect hashing functions

We proposed finding perfect hashing functions by a trial-and-error method. Other methods could be investigated. Previously known methods of finding direct perfect hashing functions for small static sets have a complexity of at least $O(n^3)$. For group sizes in the range of a few hundred records, 10 to 20 trials for finding a perfect hashing function corresponds to a complexity of $O(n \log n)$. Hence, any new method will have to be very efficient to compete with the trial-and-error method. (All known methods for small static sets involve sorting the elements first.)

5. Deletions

We have not analyzed deletions under the external perfect hashing scheme. The only adverse effect of deletions is a reduction in the storage utilization for the group involved. As a positive effect, deletions reduce the probability of an insertion causing a rehash. Whenever a group is rehashed, the effects of any previous deletions vanish. If there is a large number of deletions without a corresponding number of insertions into a group, to maintain the storage utilization the group should be rehashed. In this rehashing process, trials need be made only with fewer pages than the current number of pages in the group. The main decision to be made is when to rehash a group. Optimal rehashing policies for deletions could possibly be defined and analyzed in the same way as was done for insertions in chapter 4.

6. Memory management

As blocks of pages are allocated and deallocated for various groups, fragmentation of secondary storage will occur. Memory management in this case is much simpler than the traditional problem. This is because groups can easily be relocated to reclaim fragmented memory space. Relocation of a group requires a single read access to the disk followed by a single write access, provided that enough buffer space is available to store all the records of the group. Further details of memory management need to be worked out and analyzed.

7. Conversion of key strings to integers

In practice, keys are often strings of alphanumeric characters. Before applying a hashing function to such a key, it is necessary to convert it into an integer. The goal is to always convert distinct keys into distinct integers. In chapter 6 we described an improved method of conversion, RADIX-Convert. However, in comparison with an ideal conversion function, there is still much room for improvement.

8. Improving the solution to the optimization problem

It is clear that the procedure presented in chapter 4 for solving the optimization problem is satisfactory for our purpose. However, we have some suggestions for obtaining better solutions (closer to the true optimal solution).

The proposed heuristic retains the best *depth* states $< \tau, h_l >$ for each τ. This may be modified by giving some consideration to the values of h_l in the retained states. The solution vector obtained using the heuristic solution procedure is not far from the true optimal solution. Exhaustive search may be carried out locally, near the heuristic solution obtained. The local search may be guided by treating the objective function as a continuous function of t_i's and moving in the direction of steepest descent. The search space may be further reduced by the following observation. Suppose $(t_1, t_2, \ldots, t_m, \ldots, t_r)$ is an approximate solution where t_m corresponds to trials with $E(m)$ pages ($E(m)$ is the resulting expected number of pages in the group). In the local exhaustive search process, if we increased the value of one t_i, $1 \leq i < m$, by 1 then reducing the value of t_j, $r \geq j \geq m$, by 1 will not likely help because the solution may become infeasible. So j has to be chosen such that $1 \leq j < m$.

9. Size of the H-table

In section 7.3 we suggested the use of the H-table, which stores the parameters of the hashing functions in use. Each header table entry stores a pointer to the H-table, and not the complete parameter set of the perfect hashing function for the group. This organization saves space because each function stored may be used by several groups. Suppose t_{max} is in the range of 10 to 20. Then at most 10 to 20 hashing functions are tried each time a new perfect hashing function is required. How many different hashing functions need be stored in the H-table? t_{max} is a lower bound and the header table size is the upper bound. What is the effect of reducing the size of the H-table below the upper limit?

It's not difficult being perfect...
Because, somebody has done it.

Appendix A

In chapter 4 we presented a typical optimization problem encountered when determining a rehashing policy. In this appendix we outline a few steps of the heuristic solution procedure for that problem. The problem is to determine $(t_1, t_2, \ldots, t_9)$ so as to minimize f_1, where

$$f_1 = 9(1 - 1.0^{t_1}) + 10 * 1.0^{t_1} * (1 - 0.995^{t_2}) + \cdots + 17 * 1.0^{t_1} * 0.995^{t_2} * \cdots * 0.070^{t_8} \tag{4.17}$$

subject to the constraints

$$t_1 + t_2 + \cdots + t_9 = 10 \tag{4.18}$$

$$0t_1 + 52\, t_2 + \cdots + 32797t_9 \geq 46052\ . \tag{4.19}$$

All the constants of the problem are given in table 4.4.1 on page 52.

Solution

The solution procedure uses *depth* = 3. The computation starts at stage 9 (the last stage) and proceeds backwards to stage 1. A table of f_9^* is constructed as follows (trivial):

$$f_\tau^*(\tau, h_l) = m_h$$

$$f_9^*(t_9, 32797 * t_9) = 17$$

$<\tau, h_l>$	f_9^*	t_9^*
<0, 0>	17.0	0
<1, 32797>	17.0	1
<2, 65594>	17.0	2
...	...	...
<10, 327970>	17.0	10

Table A.1

The computation proceeds with stages 8,7, $\cdots$,4. At stage 4 the final table retained for use in stage 3 is as follows:

$<\tau, h_l>$	f_4^*	t_4^*
<1, 8629>	14.688	0
...	...	...
<6, 37956>	12.389	3
<6, 33350>	12.342	4
<6, 28744>	12.359	5
<7, 37373>	12.229	5
<7, 32767>	12.240	6
<7, 38323>	12.244	6
...	...	...

Table A.2

The computations at the next stage, stage 3, are discussed in greater detail below. The general equations are

$$f_i(\tau, h_l + a_i t_i, t_i) = (m_l + i - 1)(1 - q_i^{t_i}) + q_i^{t_i} f^*_{i+1}(\tau - t_i, h_l), \quad 0 \le \tau \le t_{\max}$$

$$f_i^*(\tau, h_l + a_i t_i) = \min_{0 \le t_i \le \tau} f_i(\tau, h_l + a_i t_i, t_i) .$$

Substituting the appropriate values for the constants, we obtain

$$f_3(\tau, h_l + 1046 t_3, t_3) = 11(1 - 0.9^{t_3}) + 0.9^{t_3} * f_4^*(\tau - t_3, h_l), \quad 0 \le \tau \le 10,$$

$$f_3^*(\tau, h_l + 1046 t_3) = \min_{0 \le t_3 \le \tau} f_3(\tau, h_l + 1046 t_3, t_3) .$$

In particular, consider the computations for the states of the form $<9, h_l>$, i.e., the states with $\tau = 9$. For each value of t_3, $0 \le t_3 \le 9$, it is only necessary to evaluate f_3 for *depth* number of different states. One value of $f_3(9, h_l + 1046 t_3, t_3)$ is evaluated for each of the *depth* best values of $f_4^*(9 - t_3, h_l)$ in table A.2. For example, with $t_3 = 3$, f_3 is evaluated as follows:

$$f_3(9, h_l + 3138, 3) = 11 * .2695 + .7305 * f_4^*(6, h_l)$$

Using $f_4^*(6, 33350) = 12.342$ from table A.2 we obtain

$$f_3(9, 36488, 3) = 11.9803 .$$

The values of $f_3(9, h_l + 1046 t_3, t_3)$ are tabulated in table A.3. There are 3 entries for each value of t_3, $0 \le t_3 \le 9$.

t_3	$\tau, h_l + 1046t_3$	f_3	$<\tau - t_3, h_l> : f_4^*(\tau - t_3, h_l)$
0	<9, 77661>	13.002	<9, 77661> : 13.002
...	...	...	...
3	<9, 36488>	11.9803	<6, 33350> : 12.342
3	<9, 31882>	11.9927	<6, 28744> : 12.359
3	<9, 41094>	12.0146	<6, 37956> : 12.389
4	<9, ... >	...	...
4	<9, ... >	...	...
4	<9, ... >	...	...
...	...	...	...
9	<9, 9414>	13.339	<0, 0> : 17.0

Table A.3

The best 3 (*depth*) values of f_3 from table A.3 are retained in table A.4, corresponding to states of the form <9, ... >. Tables similar to A.3 are computed, one for each value of τ from 1 to 10. The final table of f_3^* for stage 3 is as follows:

$<\tau, h_l>$	f_3^*	t_3^*
...	...	...
...	...	...
<9, 33511>	11.995	4
<9, 39565>	11.997	2
<9, 36488>	11.981	3
<10, 49128>	11.915	4
<10, 52105>	11.922	3
<10, 53734>	11.934	3

Table A.4

Finally, the table of f_1^* computed for the first stage is as follows:

τ, h_l	f_1^*	t_1^*
...	...	...
...	...	...
<10, 49128>	11.915	0
<10, 52105>	11.922	0
<10, 53734>	11.934	0

Table A.5

The minimum value of f_1 is 11.915. The optimal rehashing policy is obtained by tracing back the t_i^* values corresponding to the final optimum value chosen. Alternatively, partial policies of the form $(t_i^*, t_{i+1}^*, \ldots, t_9^*)$ could be stored for each state in the table of f_i^* at stage i. The optimal rehashing policy for the above problem is (0, 0, 4, 4, 1, 0, 1, 0, 0).

Appendix B

In this appendix we give the policy-tables and Em-tables for a few different page sizes and values of t_{max}. Section 7.1.1 described a heuristic procedure to determine rehashing policies using the Em-table and the policy-table for the given page size.

Number of keys, n	Expected number of pages, $E(m)$	
	$t_{max} = 10$	$t_{max} = 20$
28	3.15	3.02
31	4.03	4.02
35	4.13	4.03
40	5.05	4.97
44	5.46	5.15
50	6.36	6.11
75	10.41	9.74
100	14.40	13.63
150	23.33	21.97
200	33.00	30.96
250	41.87	40.14

Table B.1 Em-table for $b = 10$, $t_{max} = 10, 20$ (expected number of pages in a group).

Number of of keys (n)	$E(m)-1$	$E(m)$	$E(m)+1$	$E(m)+2$	$E(m)+3$	m_h-1	m_h
30	0.0	0.8	0.1	0.1	0.0	0.0	0.0
70	0.0	0.6	0.3	0.0	0.0	0.1	0.0
100	0.0	0.4	0.3	0.1	0.0	0.2	0.0
130	0.0	0.2	0.4	0.2	0.0	0.1	0.1
150	0.0	0.2	0.3	0.1	0.2	0.0	0.2
180	0.0	0.0	0.2	0.3	0.2	0.0	0.3
250	0.0	0.0	0.0	0.0	0.0	0.5	0.5

Table B.2 Policy-table for $b = 10$.

Number of keys, n	Expected number of pages, $E(m)$	
	$t_{\max} = 10$	$t_{\max} = 20$
110	4.10	4.02
135	5.10	5.04
160	6.23	6.07
200	8.12	8.02
250	10.26	10.05
300	12.51	12.16
400	17.15	16.61
500	21.84	21.21
600	26.60	25.89
700	31.43	30.64
750	33.89	33.04
1000	46.19	45.04

Table B.3 Em-table for $b = 30$, $t_{\max} = 10, 20$ (expected number of pages in a group).

Number of of keys (n)	$E(m)-1$	$E(m)$	$E(m)+1$	$E(m)+2$	$E(m)+3$	m_h-1	m_h
110	0.0	0.8	0.2	0.0	0.0	0.0	0.0
300	0.0	0.8	0.1	0.1	0.0	0.0	0.0
450	0.0	0.7	0.2	0.1	0.0	0.0	0.0
550	0.0	0.6	0.3	0.1	0.0	0.0	0.0
650	0.0	0.6	0.1	0.1	0.1	0.0	0.0
1000	0.0	0.5	0.1	0.2	0.2	0.0	0.0

Table B.4 Policy-table for $b = 30$.

Number of keys, n	Expected number of pages, $E(m)$	
	$t_{max} = 10$	$t_{max} = 20$
240	5.79	5.57
270	6.11	6.03
290	7.01	6.93
330	7.93	7.73
354	8.2	8.06
400	9.38	9.12
450	10.75	10.39
500	12.05	11.76
650	15.87	15.40
800	19.80	19.27
950	23.80	23.17
1100	27.83	27.09
1250	31.91	31.06
2000	60.0	58.0

Table B.5 Em-table for $b = 50$, $t_{max} = 10, 20$ (expected number of pages in a group).

Number of of keys (n)	$E(m)-1$	$E(m)$	$E(m)+1$	$E(m)+2$	$E(m)+3$	m_h-1	m_h
240	0.0	0.8	0.2	0.0	0.0	0.0	0.0
500	0.0	0.8	0.1	0.1	0.0	0.0	0.0
900	0.0	0.8	0.1	0.0	0.1	0.0	0.0
1100	0.0	0.7	0.1	0.1	0.1	0.0	0.0
2000	0.0	0.6	0.1	0.2	0.1	0.0	0.0

Table B.6 Policy-table for $b = 50$.

Bibliography

[AH74] Aho,A.V., Hopcroft,J.E. and Ullman,J.D. *The Design and Analysis of Computer Algorithms.* Reading, Massachusetts: Addison-Wesley, 1974.

[BD59] Barton,D.E. and David,F.N. *Combinatorial extreme value distributions.* Mathematika, 6(1959), 63 - 76.

[BR71] Burstein,H. *Attribute Sampling: Tables and Explanations.* New York: McGraw-Hill, 1971.

[BU77] Bayer,R. and Unterauer,K. *Prefix B-trees.* ACM Trans. on Database Systems, 9, 1(1977), 1 - 21.

[CB85] Cercone,N., Boates,J. and Krause,M. *An interactive system for finding perfect hash functions.* IEEE Software, 2, 6(1985), 38 - 53.

[CC80] Cichelli,R.J. *Minimal perfect hash functions made simple.* Comm. of the ACM, 23, 1(1980), 17 - 19.

[CG83] Char,B.W., Geddes,K.O., Gonnet,G.H. and Watt,S.M. *Maple User's Manual, 3rd edition.* University of Waterloo Computer Science Department Research Report CS-83-41, 1983.

[CH84] Chang,C.C. *The study of an ordered minimal perfect hashing scheme.* Comm. of the ACM, 27, 4(1984), 384 - 387.

[CH85] Cormack,G.V., Horspool,R.N.S. and Kaiserswerth,M. *Practical perfect hashing.* The Computer Journal, 28, 1(1985), 54 - 58.

[CK83] Cercone,N., Krause,M. and Boates,J. *Minimal and almost minimal perfect hash function search with application to natural language lexicon design.* Comp. & Maths. with Appl., 9, 1(1983), 215 - 231.

[CP86] Celis,P. *Robin hood hashing.* Ph.D. thesis, Department of Computer Science, University of Waterloo, 1986.

[CW79] Carter,L.J. and Wegman,M.L. *Universal classes of hash functions.* Journal of Computer and System Sciences, 18, 2(1979), 143 - 154.

[DB62] David,F.N. and Barton,D.E. *Combinatorial Chance.* London: Griffin, 1962.

[DH83] Du,M.W., Hsieh,T.M., Jea,K.F. and Shieh,D.W. *The study of a new perfect hash scheme.* IEEE Trans. on Software Engineering, SE-9, 3(1983), 305 - 313.

[FK82] Fredman,M.L., Komlos,J. and Szemeredi,E. *Storing a sparse table with 0(1) worst case access time.* Proc. 23rd Symposium on Foundations of Computer Science, IEEE Computer Society, 1982, 165-168.

[FL68] Feller,W. *An Introduction to Probability Theory and its Applications.* Vol. 1. New York: John Wiley, 1968.

[FN79] Fagin,R., Nievergelt,J., Pippenger,N. and Strong,H.R. *Extendible hashing - a fast access method for dynamic files.* ACM Trans. on Database Systems, 4, 3(1979), 315 - 344.

[FR71] Freund,J.E. *Mathematical Statistics.* Englewood Cliffs, New Jersey: Prentice-Hall, 1971.

[GL82] Gonnet,G.H. and Larson,P.-Å. *External hashing with limited internal storage.* Proc. ACM Symp. on Principles of Database Syst., ACM, New York, 1982, 256 - 261.

[GS84] Gifford,D. and Spector,A. *The TWA reservation system.* Comm. of the ACM, 27, 7(1984), 650 - 665.

[GT63] Greniewski,M. and Turski,W. *The external language KLIPA for the URAL-2 digital computer.* Comm. of the ACM, 6, 6(1963), 322-324.

[HL80] Hillier,F.S. and Lieberman,G.J. *Introduction to Operations Research.* San Francisco: Holden-Day, Inc., 1980.

[JS81] Jaeschke,G. *Reciprocal hashing: A method for generating minimal perfect hashing functions.* Comm. of the ACM, 24, 12(1981), 829 - 833.

[KN74] Knuth,D.E. *The Art of Computer Programming, Vol 3.* Reading, Massachusetts: Addison-Wesley, 1974.

[LK84] Larson,P.-Å. and Kajla,A. *File organization - implementation of a method guaranteeing retrieval in one access.* Comm. of the ACM, 27,7 (1984), 670 - 677.

[LR78] Larson,P.-Å. *Dynamic hashing.* BIT 18(1978), 184 - 201.

[LR79] Larson,P.-Å. *Frequency loading and linear probing.* BIT 19(1979), 223 - 228.

[LR80] Larson,P.-Å. *Analysis of repeated hashing.* BIT 20(1980), 25 - 32.

[LR83] Larson,P.-Å. *Analysis of uniform hashing.* Journal of the ACM, 30, 4(1983), 805 - 819.

[LR84] Larson,P.-Å. *Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval.* Research Report CS-84-23, University of Waterloo, 1984.

[LR85] Larson,P.-Å. and Ramakrishna,M.V. *External perfect hashing,* Proc. ACM-SIGMOD Intern'l Conf. on Management of Data, (Austin,

1985), 190 - 200.

[LT80] Litwin,W. *Linear hashing: A new tool for files and tables addressing.* Proc. 6th Intern'l Conf. on Very Large Databases, (Montreal, 1980), 212 - 223.

[MR83] Mairson,H.G. *The program complexity of searching a table.* Proc. 24th Symposium on Foundations of Computer Science, IEEE Computer Society, 1983, 40 - 47.

[MR84] Mairson,H.G. *The program complexity of searching a table.* Ph.D. Thesis, Department of Computer Science, Stanford University, 1984.

[NY85] Norton,R.M. and Yeager,D.P. *A probability model for overflow sufficiency in small hash tables.* Comm. of the ACM, 28, 10(1985), 1068 - 1075.

[SD80] Sarwathe,D.V. *A note on universal classes of hash functions* Information Processing Letters, 10, 1(1980), 41 - 45.

[SG85] Sager,T.J. *A polynomial time generator for minimal perfect hash hunctions.* Comm. of the ACM 28, 5(1985), 523 - 532.

[SP77] Sprugnoli,R.J. *Perfect hashing functions: A single probe retrieving method for static sets.* Comm. of the ACM, 20, 11(1977), 841 - 850.

[YD84] Yang,W.P. and Du,M.W. *A dynamic perfect hash function defined by an extended hash indicator table.* Proc. 10th Intern'l Conf. on Very Large databases, (Singapore, 1984), 245 - 254.