To the Graduate Council:

I am submitting herewith a thesis written by Timothy Andrew Martin entitled "Recursive Hash Trees." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jens Gregor, Major Professor

We have read this thesis and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor and

Dean of The Graduate School

Recursive Hash Trees

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Timothy A. Martin

December, 1995

Abstract

As long as there have been computers, there has also been data sets to store in those computers. Data storage and retrieval is an important responsibility of computer scientists. Sometimes a specific framework must be constructed around the unique requirements of a particular application. There is seldom a single best solution for any given data storage and search problem yet often there are many suitable existing solutions that can be integrated with any given implementation.

Despite all attempts at generalizing data management systems (DMS), many such systems lend themselves well to only a limited class of applications. Hash tables are one such DMS. This paper presents a series of assumptions that bound the applicability of hash tables and similar DMS. The chief focus of this paper is a particular variation of the standard hash table, the Recursive Hash Tree. The Recursive Hash Tree and its associated extensions are fully detailed herein.

The use of the these hash tree extensions can reduce the two most significant problems facing hash tables: scalability and data ignorance. The extensions detailed herein can increase performance in existing hash table applications. Empirical results are given in order to compose a clear image of the relative performance that can be expected from the Recursive Hash Tree. These extensions do not, however, broaden the class applicability of hash tables.

Acknowledgments

It is necessary to recognize those individuals who aided in the completion of this thesis. Let their contributions hereby be made known:

I thank Richard and Patricia Martin, Raymond and Doyne Martin, Helen Kania, Ross and Cindy Tarlow and Jason Wessel for their perpetual love and support.

I thank the members of my thesis committee, Brad Vander Zanden and Jim Plank and my primary thesis advisor, Jens Gregor for sharing their wisdom and support.

I thank Michelle Thompson, Brian & Ginger LaRose and Pat Hale for knowing that I could, in fact, eventually complete this work. Their collective assurance was a profound source of my motivation. I thank Carianne Meystrik for both her support and her nutritional counseling. I thank Chris Jepeway, Chris Meystrik and Roy Good for their support and their wise advice. I thank again Chris Jepeway, Randy Bond and the members of UTK-CS Wizard Labstaff for their patience during the final weeks of this project.

I thank those members of the computing community that labor to provide excellent software for the public domain.

Finally, I thank God for everything. In particular, I thank Him for providing the remarkable people mentioned above.

This is for Burney.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many different ways to store data ranging in complexity from basic pools to lists to grand hierarchies. Typically, information is stored with the intension of later being retrieved. In order to avoid potentially costly exhaustive searches over the entire data space, there must be some deterministic system not only by which the data is initially stored but also through which the data will later be retrieved.

As the amount of data in the storage space grows, it becomes increasingly important to provide for retrieval rates that scale reasonably. Sophisticated data storage and retrieval systems go to great lengths to provide fast data retrieval rates.

While also reviewing some existing key search strategies, this paper presents some extensions of one particular information storage mechanism, the hash table. Hash tables rely

upon a hash function that attempts to quickly recognize differences between keys allowing constant retrieval time in optimal situations. These extensions provide methods to lessen the impact of suboptimal situations.

1.1 Assumptions

The boundaries of this work are delimited by a set of assumptions. These next several sections detail these assumptions.

1.1.1 No a priori knowledge of data

It is assumed that the precise data to be fitted to the hash table is unknown. That is, that there is no a priori knowledge of the type, quantity, or other special properties of the data. The data could be ASCII, binary, numerical or some other specially formatted type. Although specific examples will be given later, all structures presented herein will function, with any binary data, to which any other data type may be converted.

1.1.2 Internal search

It is assumed that all keys will reside within core memory. The data values themselves may be stored externally, however. All search scenarios presented herein can be trivially modified from their original form in order to yield pointers into an external file rather than housing

the data values themselves within core memory.

Although this assumption creates a restricted environment in that there is a finite amount of core memory with which to house all data keys, it is in this fashion that we choose to present the extensions to hash tables. There are countless applications where this restriction is not so burdensome. If the restriction precludes the use of the extensions proposed herein, then these extensions are clearly not suited to that particular application.

1.1.3 Data is volatile

It is assumed that the data stored within the search structures presented herein is volatile. That is, that there is not a requirement to store the data, once loaded into the search structure, externally. External data storage is left for future work.

1.1.4 Data search is actual

It is assumed that the searches for data within the search mechanisms presented herein are for actual data. By actual search, we mean the counterpart of attributive search. That is, searches are requested to perform exact match for existing data or else to fail.

Some search mechanisms, in particular digital and binary tree searches allow for a limited class of best–fit searches; however, "best–fit" has several potential meanings such as lexical, phonetical, synonymous (same value), etc. It is not the intention of this work to present any

such best–fit mechanisms.

1.1.5 Unique and primary keys

Finally, it is assumed that the data storage mechanisms provide for unique keys. No duplicate keys are permitted. Furthermore, all keys are primary. That is, no secondary keys are implemented.

1.2 On Searching

People have long since been using sophisticated methods of information retrieval. Historically, however, these methods have been implemented not in electronic computer equipment, but rather in human minds. The human mind seems to have a remarkable capacity to interrelate pieces of information to form an amazingly complex web of knowledge. The exact associations that are drawn in the human mind in order to facilitate subsequent information retrievals is dependent upon the nature of the information that is being stored.

There are well–established mechanisms for expediting information retrieval external to the human mind. For example, dictionaries list word definitions lexically. Many large dictionaries go a step further by also providing thumb–tabs which indicate section or letter boundaries. Thumb–tabs are useful in large volumes because the size of each section is generally larger. Bookmarks provide a similar function as thumb–tabs and are "user–configurable."

Analogous mechanisms exist in many computer-based key search strategies. All such search mechanisms must establish a means of ordering the keys. All DMS enforce an ordering upon the keys at some level. Not all DMS exhibit a complete ordering, however. Furthermore, not all DMS use the same ordering paradigm for any given set of keys.

1.3 Anatomy of a Data Retrieval System

There are four fundamental components of all data management systems. Albeit some DMS's add some additional features such as external storage capabilities for data, the principle components are as follows:

- a data structure to support the internal framework of the data space
- an heuristic for iteratively selecting a key for comparison
- an operation for comparing two keys with the goal of eventually locating a particular key within the data space, or to determine that a particular key is not present in the data space
- a set of operations to allow for key insertions, deletions, searches over the data space, etc.

1.3.1 Data structure and heuristic

The data structure contains all of the concrete properties of the individual data storage and search mechanisms. Similarly, the heuristic lends all abstract properties to the same. A sophisticated data structure coupled with a sophisticated heuristic can inherently weed–out huge groups of keys during the search for a particular record.

Side–effects of certain data structures

Depending upon the storage structure, it may be necessary or perhaps merely advantageous to periodically reorganize data. This later reorganization is usually some form of garbage collection, compression, or optimization as the DMS begins to strain the limits of the memory resources.

Necessary reorganization, however, is usually the result of an addition or deletion of a key from the data space. The magnitude of the reorganization is dependent not only upon the context of the particular data item that was added or removed, but also upon the definition of the storage mechanism structure. Pools require little or no reorganization. Other DMS's could possibly require a substantial amount of data reorganization in order to maintain the integrity of the data structure.

Data storage systems that are designed to support dynamic data spaces (i.e. those requiring on–line insertions or deletions) must strive to minimize the impact of data reorganizations,

particularly those reorganizations which are required in order to maintain data integrity. Highly efficient data retrieval systems that are subject to massive required reorganization are not good tools for dynamic data spaces. Such systems are limited to scenarios that require no dynamic activity, possibly permitting off–line data reorganization strategies.

1.3.2 Key Comparisons

Every search strategy can be reduced to a process of elimination. The goal of highly sophisticated search systems is to eliminate the largest number of keys as quickly as possible. The data structure and the heuristic combine to reduce the number of required key comparisons.

Knowledge is gained by iteratively comparing a search key against other keys in the dataspace. The result of each key comparison is then used and the heuristic applied in order to both eliminate other keys from future comparisons and to determine where to look for subsequent possible matches. This process continues until eventually either the appropriate key is found or it is determined that the key is not present in the data space. It is by means of both the heuristic and the data structure that it can be determined that a key is not present within the data space without performing an exhaustive search.

The precise number of required key comparisons in the search for an arbitrary key is rarely predictable; however, some DMS can guarantee an average or a maximum number of required key comparisons given an arbitrary data space size. The number of key comparisons,

however, is often a good metric for DMS efficiency. This will be later discussed in detail.

1.3.3 Dataspace manipulation operations

The details of the operations that are used to manipulate data within the data space are very nearly the same across all DMS. A good DMS interface will provide some form of these features. The Recursive Hash

Chapter 2

Related Work

2.1 Binary Search

Binary search is a key search algorithm that exploits a preordered group of keys to facilitate speedy key retrieval or else to determine key nonexistence. Binary search is very important because it lays the foundation for many other key search strategies.

The binary search tree is an implementation that has been crystallized from the notion of the binary search; however, binary searches are not limited to binary search trees. One of the most common such implementations operates within linear array structures. Within array structures, a process not altogether unlike Newton's Method for the approximation of roots of equations is used to locate the key, if present. In this section, however, we focus on

the binary search tree.

2.1.1 The simple binary search tree

A binary search tree is a means by which a binary search can be efficiently executed. Binary search trees are comprised of interrelated nodes. Each node contains two pointers: "left" and "right," each pointing to other nodes in the tree. Generally, whatever is pointed to by the "left" pointer of each node contains a key that is, in some ordering scheme, less than the key of the node itself. Conversely, whatever is pointed to by the "right" pointer of each node contains a key that is greater than the key of the node itself under the same ordering scheme.

Since there is exactly one node per key/value, the number of nodes is independent of the order of insertion of the keys; however, the structure of the tree itself is intimately tied to the order of key insertion. The overall tree structure has a direct impact on the performance of the tree.

2.1.2 Height–balanced binary trees

The structures of binary trees depend upon the interrelationships of the data and the order of insertion into the tree. Thus two binary trees may contain identical key sets and yet display vastly different tree structures. Algorithms have been devised to prevent or correct unbalanced trees in the interest of decreasing the average number of comparisons to retrieve

an arbitrary record.

The interrelationships that are implied from both the structure and the contents of a binary tree must be maintained in whatever tree is produced from the rebalancing process. The most common mechanism for rebalancing trees involves node rotation, or pivoting.

AVL trees

Height–balanced binary trees are trees in which the depth of left and the right subtrees of every internal node of the binary tree are within a certain tolerance of one another. AVL trees are a particular variety of height–balanced binary trees in which this tolerance is restricted to 1. That is, an AVL tree is a tree in which every internal node exhibits left and right subtrees whose respective maximum depths differ by at most one level. AVL trees implement node rotations upon key insertion in order to maintain these properties. AVL trees yield asymptotic insertion and deletion times of $\Theta(lgn)$.

Red–Black trees

Red–Black Trees (Red–Black Trees) provide a more efficient method for the implementation of balanced binary trees than AVL trees. Although Red–Black Trees do not guarantee perfectly balanced binary search trees, they do guarantee near perfect tree balancing. Red–Black Trees guarantee the total tree height to be no greater than $2log_2(n)+1$, thus insuring $O(lgn)$

retrieval time. Moreover, both insertions and deletions are also asymptotically performed in $O(lg(n))$ time [CLR95].

Red–Black Trees are used as a basis for comparison with hash trees in the empirical work presented herein.

All key search strategies that implement a tree structure provide more efficient key retrieval when those trees are wide and bushy. As with Red–Black Trees, enhancements have been made to many tree search algorithms in order to attempt to maintain a nearly optimal tree structure throughout the life of the tree. Regardless of tree structure, however, binary trees are somewhat limited in that they provide a maximum branching factor of 2.

The subsequent sections of this chapter will introduce several other data storage mechanisms which provide for increased branching factors.

2.2 Hashing

Hashing is a data storage mechanism wherein a function is used to map an arbitrary key, K to a position within a statically–sized array.

The following are the primary components of typical hash tables:

- a one–dimensional often fixed–size array;
- a function, h, defined for all possible keys that maps any possible key to a valid array

index;

- a key comparison operation;
- a method for handling collisions which arises when multiple keys are mapped to the same hash index.

2.2.1 Partially unsorted records in hash tables

Keys in the hash table are partitioned into equivalence classes according to their respective hash values. Keys with identical hash values under the hash function are bundled into unsorted groups. In practice, keys with identical hash values are often arranged according to the insertion sequence.

The unsorted classes help to make hash tables a useful DMS. The relaxation of the ordering requirements within the equivalence classes permits quick insertion and deletion operations with little or no required data restructuring.

Different hashing schemes attempt to do different things with those unordered keys which are mapped to identical hash values. The following subsections detail some such "overflow" and "collision resolution" workarounds and solutions.

2.2.2 Perfect hashing

Perfect hashing is a variation on hash tables that can be lent to implementations with known data sets. Such hashing schemes map every key to a distinct hash location and provide no mechanisms for overflow or collision resolution because situations requiring these mechanisms do not, by definition, arise. Perfect hashing requires two properties of the standard hashing implementation that have been specifically tailored to suit the requirements of the specific data set: a hash table sized to accommodate every key without overflow, and a hash function that provides the requisite unique mapping of all the keys of the data set.

While providing a hash table that has the minimum number of hash values is not a problem; finding a hash function that yields a specific unique mapping can be a daunting task. Typically such functions are discovered for specific data sets empirically by varying a set of parameters to several possible equations.

Perfect hashing schemes are important because they provide for quick search times. In particular, search time is fixed to the computation time of a single function evaluation and possibly (but not necessarily) a single key comparison. No other data search scenario provides such timely access. As mentioned earlier, however, perfect hashing schemes are only practical for static data sets in which the key values are known in advance. Thus, perfect hashing fills a very specific niche in the world of DMS. See [LC91, CBK91] for more details on perfect

hashing.

2.2.3 Minimal hashing

A minimal hashing scheme is one in which there are no unused hash slots. Thus the cardinality of the hash table is fixed to the cardinality of the data set. Some hash implementations strive to produce minimal–perfect hashing, one in which each key maps to a different hash slot, yet leaves no slots unused. Such functions are quite rare. Consequently, in practice perfect hashing schemes are used more often than minimal–perfect hashing schemes. Minimal hashing schemes are typically used when memory is severely restricted.

2.2.4 Segmentation in perfect hashing

Segmentation attempts to lessen the chore of locating a single perfect hash function for large data sets by relaxing the rules of perfect hashing. Keys are hashed once into a hash table that provides for multi–record buckets. Each bucket is then analyzed offline in order to locate a perfect hashing function specific to that bucket. Thus segmentation provides a very realistic divide–and–conquer approach to locating perfect hashing functions. Furthermore, segmentation, though not with respect to perfect hashing, serves as a basis for much of the work presented herein.

2.2.5 Overflow and collision resolution

In hash tables that provide for dynamic data or specifically do not support perfect hashing, it is highly likely that two or more keys will hash to the same position within the hash table. Provisions must be taken in order to handle such data collisions. There are a number of different well–known schemes that handle collisions gracefully.

Chaining

Chaining is a very simple form of overflow handling that implements a linked–list structure off of each hash location. The records associated with any collisions that resulted in overflow are inserted into the linked–list. Some chaining implementations generalize this scenarios and simply insert all records directly into the appropriate linked–list.

Figure 2.1 illustrates chaining in hash tables. The hash function, F used in these examples is defined to be the number of set bits in the binary representation of x.

Since there it is not necessary to maintain a sorted linked list off of each hash bucket, records may be shuffled around in order to optimize search time. In particular, it would be advantageous to relocate the most frequently accessed records toward the head of the linked list. There are two common mechanisms that work toward this end: Move–To–Front (MTF) and Parent Swapping.

The Move–To–Front process moves every record to the head of its respective list upon

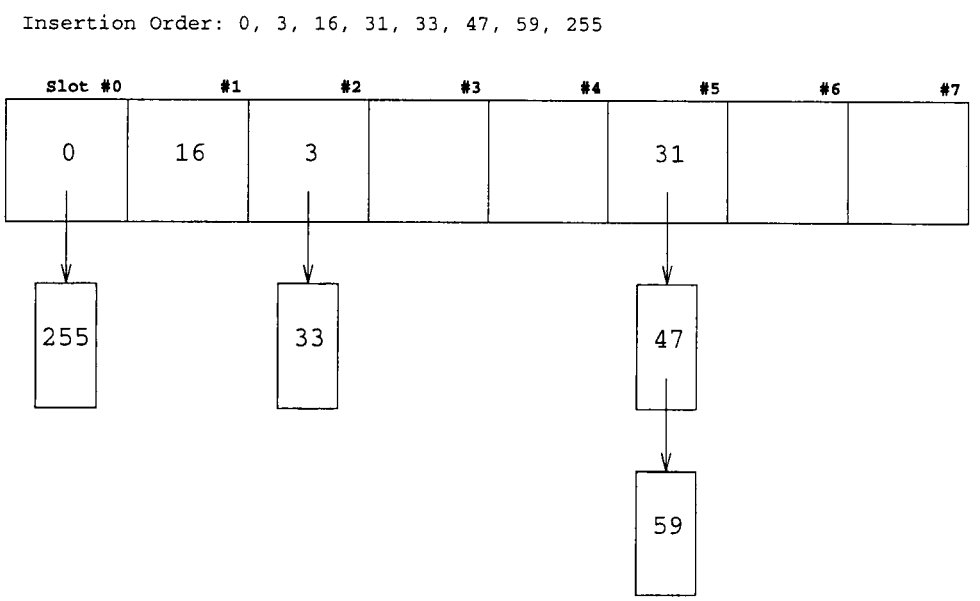

Figure 2.1: Hash table overflow via chaining

retrieval. Since the linked list is a fairly simple data structure, there is not a lot of overhead involved in this record movement; however, it is generally only successful at reducing future record retrievals when certain records are accessed much more frequently than other records.

Parent Swapping adds slightly more overhead not only to each record retrieval, but also to the data structure itself. Each record retrieval triggers a subsequent comparison of an additional field in each record, an access count. If the access count of the retrieved record exceeds that of its parent, then the records are swapped. Parent Swapping is generally only successful at reducing retrieval times in those non uniform retrieval frequencies described above.

Both of these record promotion scenarios are evaluated in the empirical work presented herein. These scenarios serve as a basis for comparison to the record promotion algorithm presented for the Recursive Hash Tree.

Open–addressing and double hashing

Open–addressing is a scheme that makes use of the Law of Large Numbers with an appropriately–sized hash table to exploit unused hash slots. Collisions and overflows result in a sequential scan (linear probe) to locate either the appropriate record or an empty hash slot. An empty hash slot indicates that the record is not present. Linear probing is used in order to locate an available hash slot (in the case of insertion) or to search for chained keys (in the case of retrieval).

Figure 2.2 illustrates open–addressing with linear probing using the same data and hash function in Figure 2.1.

Double hashing uses a similar concept except that instead of linear probing, a different hash function is applied to the key. This second hash function may in turn result in another collision or overflow which is generally resolved via linear probing as above. The benefit that double hashing provides, however, is that it tends to spread keys apart better than linear probing, thus reducing "clusters" which tend to attract increasing numbers of collisions.

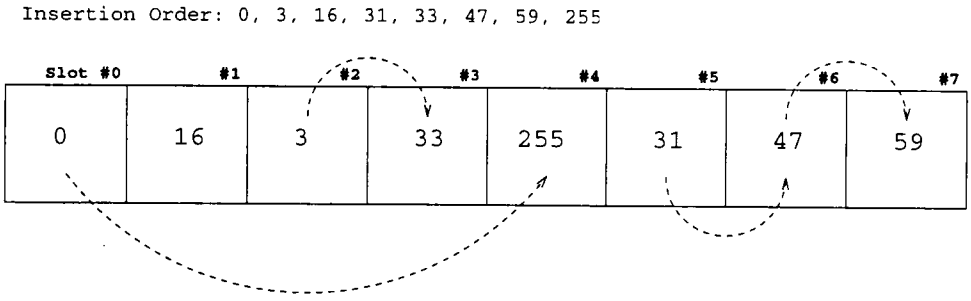

Figure 2.2: Hash table overflow via open–addressing

There are numerous problems to which both of these methods are vulnerable. The most obvious such problem is in dealing with a dataspace that has outgrown the hash table. Other complexities arise with record deletions. These two issues alone prevent both open–addressing and double hashing from being used in all but the smallest of data applications.

Buckets

Buckets are important because they introduce a whole family of hashing schemes that are based upon the same concept. A bucket allows multiple keys to be mapped to the same logical hash location. The hash table serves to direct the search to looking in the ballpark by identifying the appropriate bucket much like a thumbtab in a large dictionary. Eventually, some process must search through the bucket for the appropriate key.

Collision resolution is the process of determining what to do when the record that is

found in the relevant hash slot is not the search key itself. This event presupposes that there has been some provision for mapping multiple keys to the same logical hash position, a process known as overflow resolution. Overflow resolution defined simply as whatever process must transpire in order to facilitate record insertion once a bucket exceeds its predetermined maximum size.

The principle problem with buckets is their poor scalability. Large buckets serve to increase the average number of comparisons required to find an arbitrary record. Key retrieval rates generally increase with corresponding increases in the number of key comparisons.

A standard hash table with overflow handling via doubly–linked lists (chaining) is implemented in the empirical work presented herein as a basis for comparison.

2.3 Ordered Hashing and Trie Hashing

While most hashing implementations are designed to spread data apart into a pseudo–random ordering, some hashing schemes are constructed so as to produce an ordered data structure. Methods that yield ordered data have a distinct advantage over unordered hashing schemes in that data may be requested in sorted order upon retrieval. If the data is stored presorted, then there is little post-processing required once the data has been retrieved from the DMS in order to deliver the sorted list.

Trie hashing [Lit81] is a type of ordered–hashing that borrows concepts from both binary search trees and another search strategy known as digital search tries. Trie hashing provides the following important advantages over most other forms of hashing:

- Keys are ordered;
- Only the trie is stored in core memory. The trie is very small since nodes do not contain actual keys;
- Values may be retrieved in a single non–core memory access.

The trie hashing scheme consists of a binary trie where nodes contain information for multiple records. Each node also contains search instructions for comparing a given key against keys housed within that node. Leaves contain addresses into buckets that are stored in an external file. Although dubbed a hash algorithm by its author, the concept of trie hashing is probably better categorized as a digital search trie with a dynamic heuristic.

Chapter 3

Methodology

The idea behind the Recursive Hash Tree is based upon the following assumptions: first, the hash table is an efficient method of data retrieval up to the point of bucket overflow; second, although undesirable, hash collisions are nevertheless inevitable in practice within an implementation supporting dynamic data.

The first assumption is reasonable because up to the point of bucket overflow, the maximum number of key comparisons required in order to locate a record or to prove that the record is not present is equal to the bucket size for that hash table. The second assumption is reasonable because it is so unlikely to stumble upon a perfect hash function when dealing with dynamic data.

Although these two concepts are conflicting in nature, in the light of the Recursive Hash

Tree, they actually serve to complement each other. The hash tree attempts to decrease the hazard of bucket overflow by providing an overflow structure that is nearly as efficient as the original hash table was, before bucket overflow. The hash tree accomplishes this by supplying another hash table as the base data structure for each bucket overflow, thus forming trees of hash tables.

3.1 Hash tree design

The top–level structure of a hash tree is a simple hash table: an array of buckets to which data values are mapped in some reproducible fashion. Each bucket of the hash table contains zero, one or possibly multiple nodes. In hash trees, however, the overflow mechanism of each bucket is a "child" hash table. These child hash tables serve as mechanisms to handle bucket overflow while maintaining most of the same properties as the root (parent) hash table, including the potential to have further child hash tables. Thus hash tables are nested to form a single hash tree. Figure 3.1 highlights the high–level structures of a hypothetical hash tree.

Note that in Figure 3.1, not every full hash bucket contains a pointer to another hash table. This is because new tables are only allocated when a particular bucket overflows, not merely when a bucket has reached its capacity. Furthermore, the root hash table is larger

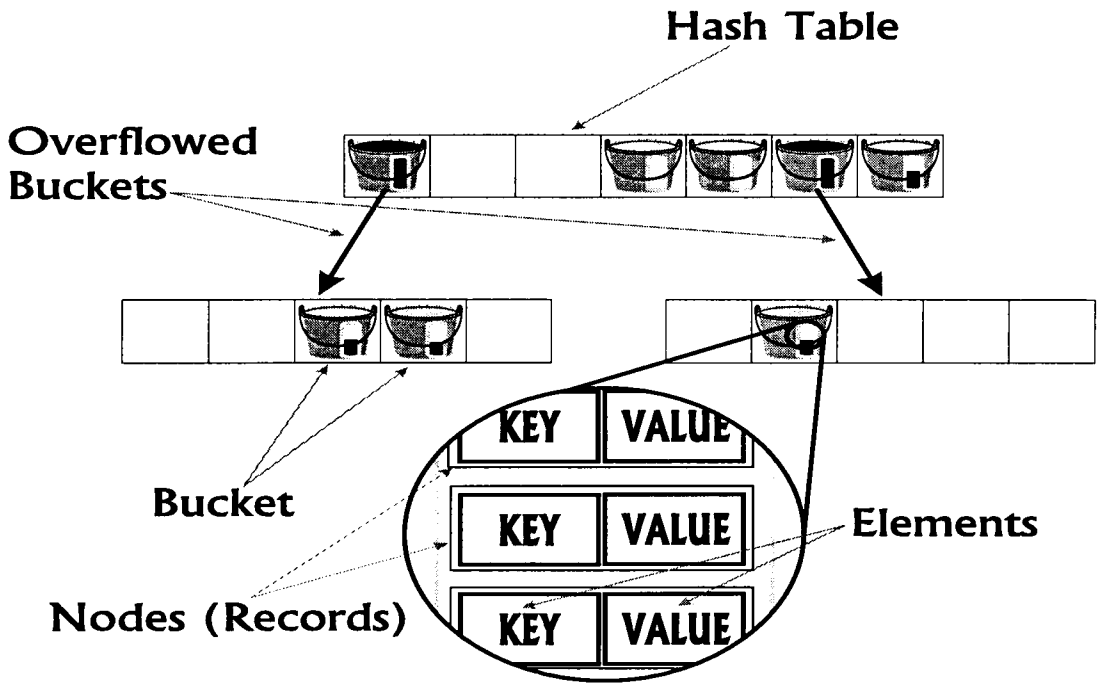

Figure 3.1: The high–level structure of a hash tree

than all of its children.

This is accomplished via a hash table reduction factor that limits the growth of the hash tree as new tables are created. This reduction factor can be tuned to balance efficiency and memory consumption.

3.1.1 Data structures of the hash tree

The hash tree is composed of several hierarchical data structures. The structures, in increasing order of complexity, are: elements, nodes, buckets, hash tables, and the hash tree itself. The following paragraphs detail the different data structures that are used in hash trees.

Elements

The element is the smallest compound data type that the Recursive Hash Tree uses. Each record uses exactly two elements, one for the key and one for the value.

Elements are comprised of a data (pointer) field and a length field. The length field is provided both to permit binary data where sentinel characters are not feasible, and to assist key comparisons. Library functions are provided to build element data types from standard data types such as string and numerical data.

Nodes

The hash node creates a logical binding between the key element and the value element that are associated with each record. A full hash node in the tree corresponds to an actual record in the dataspace.

The hash node consists of two elements (one each for the key and for the value), a counter for the access–count of this record, and a state that indicates whether or not the hash node is currently occupied.

There is exactly one hash node for each record in the hash tree. Each hash node is owned by exactly one bucket.

Buckets

In hash trees, the bucket serves two functions. The more obvious of these functions is to store multiple hash nodes. The less obvious function is to provide its own means for handling overflow. As previously mentioned, this overflow mechanism is another hash table.

The contents of a single bucket includes a linear array of hash nodes, a number indicating the hash value to which the bucket corresponds which is used for backward referencing in record promotion, and a pointer to the child hash table if the bucket has been inundated with records to the point of overflow.

Each bucket is owned by exactly one hash table. Each bucket owns at most one hash

table.

Hash tables

The hash table is a collection of buckets together with information indicating the size and level of the hash table.

Additionally, to allow for variable–sized buckets within the same hash tree, a number indicating the bucket size associated with the hash table is provided. All buckets of any given hash table are equally sized.

Each hash table is owned by at most one bucket. Each hash table will own multiple buckets, the precise number of which corresponds to number of full hash slots in each hash table up to a maximum value corresponding to the size of the hash table.

The hash tree

The hash tree contains: a pointer to the root hash table of the tree, several control registers that indicates which extended hash tree features are enabled, and a set of read–only registers indicating performance statistics upon the return from certain library functions.

The remaining information provides configuration values that are used by all hash tables in the tree. This information includes: the initial bucket size, the bucket size decay rate, the hash table size decay rate, among others.

3.2 Implementation Issues

There are a number of issues that must be considered in designing a hash tree. In particular, in order to accommodate a wide variety of dataspace sizes, there are certain tuning parameters to be considered. Also, certain adverse conditions can be difficult to prevent due to the lack of prior knowledge about the dataspace. These issues are confronted in this section.

3.2.1 Increasing Tree Splaying

In this subsection we address the undesirable effect of sparse branching. In particular, we present one particular method of achieving greater tree splay, or tree "bushiness," by modifying the hash function in each hash table.

Although their causes are entirely different, the results of single branching in hash trees and pre–ordered simple binary tree insertion are very similar. Unfortunately, however, single branching, unlike pre–ordered simple binary tree insertion, generally has no feasible dynamic resolution. When keys are inserted into a binary tree in a pre–sorted order, node rotation can quickly resolve the problem. Also, subsequent out–of–order key insertions to the binary tree will start to correct the linear tree. With single branching in hash tables, there can be no hope of changing the non–branching effect short of implementing a better hash function and rehashing all records contained in the affected hash tables.

The probability of tree splaying can often be increased, however, by a combination of several implementation issues. The most obvious way to attempt the prevention of single branching is to use a different hash function in each new hash table. This would be quite feasible if the implementation were coded in an interpreted language such as *Lisp*. In a compiled–language like *C*, however, it is not as easy to dynamically generate new hash functions.

The implementation presented herein has been accomplished with a much simpler method of hash function variation. We allow the hash function to be "seeded" by information which varies based upon certain characteristics of not only each hash table but also the tree as a whole. This way, a single hashing function may be written which is used by all hash tables.

The function seed is computed once, at hash table creation time, from several transient characteristics of the hash tree. In particular, we use the current table level and the current number of records in the tree in order to compute a pseudorandom seed.

Whatever implementation or seed generation techniques are used, ultimately it is left to the discretion of the hash function to use the seed information as an effective external impetus. The insightful user may supply her own hash function to the hash tree through a library interface at the time of hash tree creation.

3.2.2 Buckets

Buckets in any hash table implementation are a means of gathering multiple nodes whose keys hash to the same value into a contiguous segment of memory. Buckets have been used in hash tables for decades.

Organization of keys within the bucket

Since buckets contain a small number of records, the bucket implementation that was chosen for this work is currently an unsorted one–dimensional array. The routine that searches through the buckets for a specific key is optimized for the trivial cases of single and double record buckets since these are generally the most frequently occuring bucket sizes.

The most straightforward way to organize the keys within the bucket is to provide a small pool of unsorted keys, perhaps an unsorted linear linked list. Another possibility is to provide an ordering for the keys within a linear linked list. However, since a bucket is just a collection of equally-hashed keys, there are many ways to organize that collection of keys. The keys could, for example, be organized as small binary or B trees; or in a fractal nightmare: even as small hash tables. More intelligent storage algorithms would certainly provide speedier bucket search times.

It is not, readily clear whether or not the expected faster bucket search times would yield enough of a performance boost to offset the expense of maintaining such structures. Future

work may indeed show that the overhead involved in maintaining sorted buckets is in certain cases a wise investment in order to provide for timely bucket searches.

Buckets and the law of diminishing returns

Buckets are subject to the law of diminishing returns. Near the root of the hash tree buckets are expected to be very dense and can effectively conserve core memory. Near the leaves of the hash tree, however, it is more likely that the buckets will be sparse. Thus, buckets that supply space for multiple nodes may actually increase memory consumption in those sparse locations.

The extensions presented herein provide for a configurable bucket size reduction factor. This parameter is an integer that is subtracted from or added to the bucket size associated with the current hash table in order to establish the bucket size of a new hash table. Thus the bucket reduction factor can decrease the bucket size in each new hash table as the tree depth increases. It is expected that a proper tuning of this parameter could have a significant positive impact upon memory conservation. Further discussion and an empirical study of the diminishing returns of bucket sizes are presented later in this work.

When to use buckets

It is expected that buckets will provide the most satisfactory results only in two circumstances. Firstly, buckets should dramatically decrease memory consumption in implementations where the dataspace is very large with respect to the amount of core memory. Secondly, buckets should reduce the impact of streamlining caused by a poor hash function.

3.2.3 Hash table sizes

As the level of the hash tree increases, it becomes increasingly more likely that the hash tables at those lower levels will be rather sparse. In anticipation of this effect, we provide a table reduction factor for the hash table size that is proportionate with the hash table level. This reduction factor would be expected to minimize core memory consumption should the hash tree grow deeply.

It is expected that an appropriate table reduction factor would reduce core memory consumption and substantially reduce the need for multi–record buckets. The hash table size reduction factor can be configured on a per–tree basis via a control register in the Recursive Hash Tree. An empirical study of the effects of hash table size decay rate will be discussed later in this work.

3.2.4 Record insertions

Figure 3.2 presents the overview of the hash tree record insertion algorithm. The actual library routine has a user–level helper function that extracts the root hash table from the hash tree and calls INSERTRECORD with access_count = 0.

The insertion algorithm is straightforward. The insertion process begins by searching through the appropriate bucket for the given key. If the key is already present in the bucket, then the value information is overwritten since duplicate keys are not permitted. If the key is not already present in the bucket, then either there is room for this new record in this bucket or there is no such room. In the first case, where the bucket is not yet full, the routine BUCKETSEARCH will initialize a new node within the bucket. In the second case, where the bucket is full, overflow occurs and insertion must continue in the child hash table of this bucket. If there is no child hash table, then a new child hash table is created, and the record will be inserted into the appropriate bucket within the new hash table.

BUCKETSEARCH performs all of the decision making within a given bucket. It decides where, if at all, the record should be inserted within the bucket. In the case of record promotion which we discuss later in this chapter, BUCKETSEARCH also locates the record with the minimum access count within the bucket. This record is then communicated back to the record promotion algorithm for further evaluation.

INSERTRECORD (*key*, *value*, *access_count*, *hash_table*)

```
1   p       = F (key, hash_table)
2   bucket  = hash_table -> buckets[p]
3   if (!  bucket)
4   then    bucket = CREATEBUCKET (hash_table -> size)
5   node    = BUCKETSEARCH (key, CREATE, bucket)

6   if (node)
7   then    node -> value = value
8           node -> key   = key
9           node -> access_count = access_count
10          node -> state = FULL
11          return 0

12  else    new_table = bucket -> hash_table
13          if (!  new_table)
14          then bucket -> hash_table = CREATEHASHTABLE (hash_table)
15          return INSERTRECORD (key,
                                 value,
                                 access_count,
                                 bucket -> hash_table)
```

Figure 3.2: Hash tree record insertion pseudocode

3.2.5 Record retrievals

Figure 3.3 presents the overview of the hash tree record retrieval algorithm, FETCHRECORD. The algorithm shown here does not implement record promotion. The algorithm for record retrieval that incorporates hash tree record promotion will be presented later in this chapter.

Similar to INSERTRECORD, the actual record retrieval library function has a user–level helper function in order to extract the root hash table from the hash tree.

Again, the algorithm is very simple. The appropriate bucket is searched for the given key. If the key is found in the current bucket, then the corresponding value is returned. If the key is not found in the current bucket, then the search begins anew in the hash table of the bucket. If there is no such hash table, then the search ends in failure.

3.3 Handling Record Deletions

In some DMS, deletions are nontrivial. Many mechanisms that are bound to retain proper tree balance such as AVL and Red–Black Trees must in certain cases begin rebalancing following a record deletion. In some cases, such rebalancing might cause a chain reaction which could trigger further deletions and subsequent requisite rebalances.

FETCHRECORD (*key*, *hash_table*)

```
1   p       = F (key, hash_table)
2   bucket  = hash_table -> buckets[p]
3   node    = BUCKETSEARCH (key, NOCREATE, bucket)

4   if (node)
5   then    node -> access_count ++
6           return node

7   else new_table = bucket -> hash_table
8   if (!  new_table)
9   then    return NULL
10  else    return FETCHRECORD (key, new_table)
```

Figure 3.3: Hash tree record retrieval pseudocode

3.3.1 Deletions in open–addressing

Open–addressing in hash tables can also require significant processing following deletion. Since open–addressing places overflowing records into unused hash slots using empty slots to signal the end of an overflow chain, the deletion of a record could leave an empty slot in an inappropriate location.

For example, notice that in Figure 2.2, the retrievals of record **33** and of record **255** depend upon the presence of record **3** since the encounter of an empty hash slot signals the end of all currently running links in open–addressing. If record **3** were to be deleted outright, with no table reorganization, then the integrity of the resulting table would be lost.

In order to correct this problem with open–addressing when record **3** is deleted, records

33 and **255** must be shifted so that no unfilled hash slot separates their slots from those of their respective parents. Figure 3.4 shows the correct hash table structure following deletion of record **3**. The potential restructuring can be catastrophic in light of frequent record deletions.

3.3.2 Deletions in chaining

Chaining can handle record deletions much more efficiently than open-addressing. Since overflow results in a linked–list chained off of a particular hash slot, record deletion amounts to one of two cases. Either the record to be deleted resides in the hash slot itself, or the record resides somewhere in the linked list (chain).

Deletion from the linked–list is trivial. If the record resides in the hash slot itself, then one record is selected from the linked–list to overwrite the record in the hash slot. The now–promoted node is then deleted from the linked–list. To simplify this process, either the head or the tail record is often selected for promotion.

In order to avoid any special cases, some chaining implementation do not actually contain a hash slot. Instead, all records are stored in the linked–list.

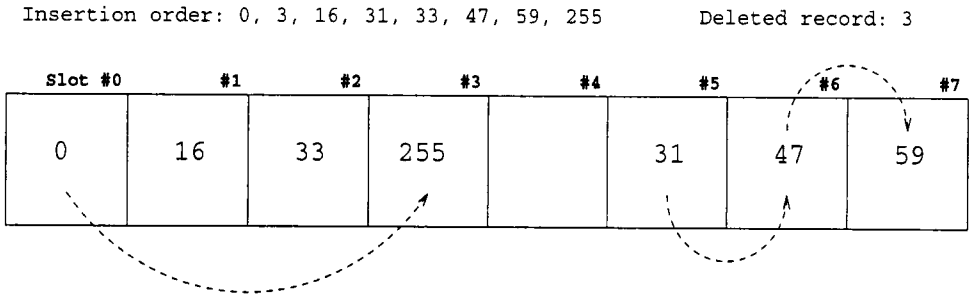

Figure 3.4: Open–addressing after proper deletion of record **3**

3.3.3 Deletions in hash trees

In hash trees, there is no such overhead incurred from deletions. Since records that overflow are placed into a new hash table, there is no possibility of leaving records stranded. Furthermore, hash tables are children of buckets, not of individual records. Consequently, if a bucket search fails for a particular record, then the search always continues in the hash table of that bucket, if present. Record promotion, if used, will eventually fill any empty non–leaf bucket slots.

Deletion is accomplished by setting a flag in a hash node that indicates that that particular node no longer contains a record.

The dynamic memory consumed by the key and values associated with that record are reclaimed, unless the record is being relocated due to displacement because of another record being promoted. In such cases, only the flag is set in the node because the memory will still

be used by the node to which the displaced key hashes.

Similar to INSERTRECORD, the actual record deletion library function has a user–level helper function in order to extract the root hash table from the hash tree.

Figure 3.5 presents the overview of the record deletion algorithm for Recursive Hash Trees. Notice that record deletion does not implement any means of garbage collection. Some method of garbage collection should be implemented and is left for future work.

3.3.4 Record promotion

As mentioned earlier, the fact that the equivalence classes induced by the hash function are unsorted allows us to move frequently accessed records closer to the root table.

The underlying assumption of record promotion is that certain records will be requested more frequently than other records. This assumption, however, is only reasonable in certain applications. In fact, certain applications may have quite the opposite behavior. Other applications may exhibit a more even distribution of record requests. Thus, this extension is user–configurable.

Each hash node contains a counter that indicates the number of times that the associated record has been non–incidentally accessed. Since record retrieval is recursive, each hash table "knows" what keys are being retrieved from its children and can easily look at the access count prior to passing the key up along the recursion chain. If the access count exceeds that

DELETERECORD (*key*, *hash_table*)

```
1   p       = F (key, hash_table)
2   bucket  = hash_table -> buckets[p]
3   node    = BUCKETSEARCH (key, NOCREATE, bucket)

4   if (node)
5   then    node -> flag = EMPTY
6           return 0

7   else    new_table = bucket -> hash_table
8           if (!  new_table)
9           then return -1
10          else  return DELETERECORD (key, new_table)
```

Figure 3.5: Hash tree record deletion pseudocode

of other records in the respective bucket by some predetermined value, then the retrieved record can be inserted into that higher position, either replacing a lesser-accessed record, or filling an empty position. The displaced record, if any, can then be reinserted beneath the hash table associated with the respective bucket.

It is important to note, however, that record promotion is not a swapping process, since downward motion in hash trees is determined by the respective hash function of each hash table. Consequently, it would be a poor assumption to presuppose that the displaced node will hash to the position from which the promoted node originated.

When a displaced record is reinserted beneath its original hash table, it can also trigger subsequent promotions and displacements. That is, displaced records are not necessarily inserted into the nearest available position. Instead, displaced records are inserted into the nearest position suitable for a record with the respective access count.

When to promote records

There are certain issues that must be considered when performing record promotion. The most important decision is that of when to promote a record. A threshold must be identified at which it is beneficial to incur the added expense of node promotion in favor of the probable speed increase for subsequent retrievals. This issue is left for empirical study and will be addressed in the results of this work.

How far to promote records

Once record promotion is enabled, it is possible that the tree will become very unstable if there is no set limit on the distance that records can be promoted. Since the root hash table is small with respect to the overall size of the dataspace, thrashing could ensue if too many records were too frequently promoted to the root table. We supply a configuration parameter that limits the number of levels that a single record may climb following a single record access. Again, empirical study will address this issue.

When to use record promotion

Because record promotion adds overhead to the data retrieval, it may be beneficial to use two modes of operation for the hash tree: training and production (post–training). Training mode would allow those frequently accessed records to "bubble up" to the top of the tree during a series of retrievals. If the overhead associated with record promotion were to be too great, then this extension could be turned off once the tree has been properly trained. Subsequent retrievals would take advantage of the fact that popular records are close to the top of the tree, while no longer incurring the overhead of actually performing more record promotions.

It is expected that record promotion most significantly reduces the expected number of comparisons in successful retrieval in configurations that satisfy the following criteria:

- retrieval rates mimic a Poisson or other similar peaked distribution pattern so as to greatly distinguish "popular" records
- the original hash tree is relatively deep so as to provide a nontrivial promotion path
- the insertion order is not ordered according to expected record retrieval frequency since such cases would preoptimize the tree in this respect

Figure 3.6 introduces the augmented record retrieval algorithm that implements record promotion. This algorithm is much like the original FETCHRECORD illustrated in Figure 3.3 with the exception that prior to iteratively returning the value, if found, the access count is updated and tested against that of all other records in the current bucket. If the record promotion configuration parameters are such that the located record is to be promoted, then the algorithm proceeds with the record promotion; otherwise, the value is returned with no other special treatment.

As with the previous algorithms, the actual routine for record retrieval with record promotion is accompanied by a user–level help function that extracts the root hash table from the hash tree.

There is one important change in the hash tree record promotion algorithm shown in Figure 3.6. BUCKETSEARCH returns not only the appropriate node, if any, but also the node with the minimum access count of all nodes in the current bucket. This minimum node is

FETCHRECORD_WITHPROMOTION (*key, hash_table, hash_tree*)

```
1   p           = F (key, hash_table)
2   bucket      = hash_table -> buckets[p]
3   (node, min) = BUCKETSEARCH (key, NOCREATE, bucket)

4   if (node)
5   then    node -> access_count ++
6           if (hash_tree -> control == RECORD_PROMOTION)
7           then    node -> promotion_d = hash_tree -> promotion_d
8           else    node -> promotion_d = 0
9           return node

10  else new_table = bucket -> hash_table
11  if (!  new_table)
12  then    return NULL
13  else    record = FETCHRECORD_WITHPROMOTION (key, new_table, hash_tree)
14          if (record)
15          then    if (hash_tree -> control == RECORD_PROMOTION)
16                  then    if (record -> distance)
17                          and (record -> access_count
                                > min -> access_count + hash_tree -> record_delta)
18                          then    INSERTRECORD (min -> key,
                                                  min -> value,
                                                  min -> access_count,
                                                  bucket -> hash_table)
                                                  hash_tree
19                                  min -> value = value
20                                  min -> key = key
21                                  min -> access_count = record -> access_count
22                                  min -> promotion_d = record -> promotion_d - 1
23                                  record -> state = EMPTY     /* delete record */
24                                  record = min
25  return record
```

Figure 3.6: Hash tree record retrieval pseudocode

used as the basis for comparison in order to determine if record promotion will occur at this level. Record promotion occurs in the unwinding stage of the recursion process.

Chapter 4

Results and Findings

In this chapter, a set of metrics is presented along with a series of experiments in order to determine the relative performances of the extensions that have been presented in the preceding chapter. Performance comparisons are drawn between hash trees with and without these extensions, to a standard hash table implementation with chaining, and to Red–Black Trees.

4.1 Testing methodology

4.1.1 Metrics

In this subsection, we present a set of metrics by which overall DMS performance will be measured. The two principle metrics are timing information and memory consumption statistics.

In general, asymptotic performance in any DMS can often be estimated by determining the number of comparisons required to complete key lookups. However, in hash trees, there are two principle parameters that must be taken into account in order to estimate overall performance. Both the number of function evaluations as well as the number of key comparisons must be estimated.

In DMS where tree structure is tightly regulated such as Red–Black Trees, it is possible to place accurate bounds on these results. Due to the unpredictable nature of hash tables and hash trees, however, empirical results are required in order to estimate performance. This is eminently true in situations that utilize highly dynamic extensions such as record promotion.

In some applications, it is important to be able to determine key existence or nonexistence. In these situations, it is equally important to discover rapidly that a given record is not present as it is to discover the existence of another record. Thus, not only should successful key comparisons be measured, but also unsuccessful key comparisons.

Memory utilization

Another useful metric in determining hash table performance is memory utilization. In particular, since the concept of buckets was presented with the notion of memory conservation in mind, overall memory consumption should be measured.

Time requirements

Finally, since the goal of all DMS is to retrieve certain information as quickly as possible, ultimately, we are most interested in the amount of time that is required in order to complete a sequence of operations. Thus, total insertion, retrieval and deletion times are measured in the experiments.

4.2 Empirical Environment

The following subsections detail the environment within which all experiments were executed.

4.2.1 Hardware

All experiments were executed on an Intel Pentium 90MHz machine with 32MB 70ns non–parity RAM. I/O was via E/IDE hard disk interface.

4.2.2 Software

Compiler and operating system

The operating system was Linux SlackWare version 1.2.3. The C compiler was GNU's gcc version 2.6.3. All libraries and source code were compiled with optimization turned on (i.e. -O2). No compiler debugging switches were used during the experimental trials.

The run–level of the machine was single–user mode with no networking operations enabled. I/O was identical both in number and in type for all trials within each experiment.

Red–Black Trees

The Red–Black Tree used herein was a slightly modified version of a public domain generic Red–Black Tree implementation [Pla]. The code was minimally modified in order to collect the same memory and key comparison statistics as the other DMS's.

4.2.3 The data

The data used for the tests is collected from an active USENET "history" database from the INN server at the University of Tennessee, Knoxville Department of Computer Science. This file contains ASCII data which uniquely identifies particular news postings (articles) with their appropriate date timestamp and file locations. For these experiments, the file location information is removed and only the unique article identification (Message–ID) and

date information is used for the record key and value, respectively.

The average length of a record key is 33 characters while the average record value is 12 characters in length. Though these values may not represent the typical ratio of key length to value length, the record values in these experiments are effectively discarded and only serve to consume core memory. We are concerned only with locating the record rather than the actual record value.

Records are read from an input file as key/value pairs, one record per line, key and value separated by whitespace. Figure 4.1 is an excerpt [1] from the actual files that were used in these experiments:

4.2.4 Experiment Descriptions

Experiment #1: Insertion and Uniform Retrieval Rates

There are two principle characteristics of the generic hash tree that must be considered in order to obtain baseline information prior to dealing with further hash tree extensions: hash table size and bucket size.

The size of the initial or root hash table is very important in that it determines to a large degree the branching factor of the tree. Table size reduction factor also plays an important role in determining the branching factor. In particular, if the table size reduction factor is

[1] This data is non-proprietary and is considered to be in the public domain.

```
<32baqiINNqa3@CS.UTK.EDU> 776547994~-~776547986
<32e47iINN5l2@CS.UTK.EDU> 776639545~-~776639538
<3il143INNk9v@CS.UTK.EDU> 793663435~-~793663427
<3j2773INNjsr@CS.UTK.EDU> 794075175~-~794075171
<3bl119INN13n@vixen.cs.utk.edu> 786302831~-~786302825
    .
    .
    .
```

Figure 4.1: Experiment data file excerpt

set to 1.0, then all hash tables in the tree maintain the initial table size and branching factor.

If the hash function is far from perfect, then a slight reduction factor can cause exhaustion of core memory by wasting many hash positions in each hash table. Thus together, the initial size and decay rate are very important in tuning the hash tree to a particular application's data requirements. Experiment #1 attempts to emphasize and isolate the behavior of these configuration parameters in order to derive a good baseline against which the remaining extensions may be compared.

As mentioned in earlier chapters, buckets can permit multiple records to reside in the same logical location in the hash table. This can be very advantageous in that a node within a bucket requires far less physical memory than a new hash table. Thus, memory can be conserved by implementing multi–record buckets.

Since multi–record buckets introduce multiple key comparisons at each hash table level, bucket size affects not only memory consumption, but also overall tree performance. As with hash table reduction factor, the bucket size reduction factor similarly affects bucket decay or growth.

In order to provide a sensible basis for comparison of hash trees with standard hash table performance, the same hash function is used in both standard hash tables and hash trees. Hash table size is varied in two different ranges, first with small initial root tables and next with relatively large initial root tables.

Experiment #2: Record Promotion

Record promotion is potentially the most significant extension to hash tables presented herein. The ability to move frequently–accessed nodes up closer to the root hash table serves to smooth problems caused by inappropriate bucket sizes and hash table sizes in applications conforming to this model.

Experiment #2 collects statistics for different scenarios implementing record promotion. In particular, this experiment examines the impact of the two record promotion configuration parameters: maximum promotion distance and the triggering promotion delta.

Initially a data set is inserted into the tree. A collection of frequencies generated according to a Poisson distribution is used to request certain records multiple times. This weighted

group of keys is read in twice. The first pass over the group trains the tree to the key retrieval frequencies. The second pass measures the performance increase from the newly–trained tree by first turning off the record promotion extension.

The results of this experiment model data applications that exhibit non–uniform record retrieval rates. This model tests both the overhead and the increased performance of a hash tree trained by record promotion.

The standard hash tables make use of Move–To–Front and record swapping as a basis for comparison for the hash tree record promotion technology.

Experiment #3: Record Deletion

Experiment #3 investigates the respective efforts of record deletions in each DMS. Furthermore, the resulting data structure efficiency is evaluated by measuring the time for series of both successful and unsuccessful record retrievals.

4.3 Experiments

4.3.1 Experiment #1: Table Sizes and Bucket Sizes

Experiment #1 inserts 100,000 unique records into each DMS then performs a sequence of 1,000,000 retrievals. The keys for retrieval are randomly selected from the original data set

with a uniform distribution. Since all keys are selected from the original data file, all retrieval keys are known to be present in the DMS and consequently result in successful retrieval.

Experiment #1 provides three important series of statistics: total insertion time, total retrieval time, and total memory consumption. Experiment #1 is designed to test the standard configuration parameters of each DMS. For Red–Black Trees, there are no such configuration parameters. In the case of Standard Hash Tables, the only such configuration parameter is the initial hash table size. There are several such configuration parameters for Recursive Hash Trees: initial hash table size, table reduction factor, initial bucket size and bucket size reduction factor.

The exact specifications for each DMS that are used in Experiment #1 are shown in Table 4.1

All of the DMS's shown in Table 4.1 are continuous, varying only one parameter at a time, with the exception of the Recursive Hash Tree. In particular, between table sizes of 1,000 and 5,000, the table size reduction factor is also changed. This will produce a jump that will be seen in all of the experiments. It is necessary to modify the table size reduction factor between root table sizes of 1,000 and 5,000 in order to allow the entire tree to fit within core memory.

Table 4.1: Experiment #1: Configuration parameters

DMS	CONFIGURATION PARAMETERS
Red–Black Tree	No configuration parameters
Standard Hash Tables	Two series of initial hash table sizes: • Small root table size: 250, 500, 750 , 1000 • Large root table size: 5,000, 10,000, 15,000, 20,000, 25,000
Recursive Hash Trees	Two series of root hash table sizes: • Small root table sizes: – Initial table sizes: 250, 500, 750, 1000 – Table size reduction factors: 0.25, 0.20, 0.15, 0.10 • Large root table sizes: – Initial table sizes: 5000, 10,000, 15,000, 20,000 25,000 – Table size reduction factors: .00060, .00080, .0010, .0012 Varying bucket sizes: • Initial bucket sizes: 1, 2, 3, 4 • Bucket reduction factors: 0, 1, 2, 3

Results

The statistics shown in the following figures for each DMS reflect the respective configuration parameters as detailed in Table 4.1.

Figure 4.2 and Figure 4.3 show the insertion and retrieval times, respectively, for each of the trial runs of the DMS's. In the case of the Recursive Hash Tree, the statistics shown reflect the trials involving the smallest hash table reduction factors for each initial table size (i.e. 0.25 and 0.0012 for small and large initial table sizes, respectively). No statistics are shown in this figure for the Recursive Hash Tree with buckets.

Figure 4.4 and Figure 4.5 show the retrieval times for the Recursive Hash Trees with small and large root table sizes, respectively. The results are shown as a function of both the initial hash table size and the hash table size reduction factor.

Figure 4.6 shows the total memory consumption for each of the trial runs of the DMS's.

Table 4.2 and Table 4.3 show the total memory consumption and total retrieval times, respectively, for Recursive Hash Trees with multi–record buckets. The results are shown as a function of both the initial bucket size and the bucket size reduction factor. Redundant configurations are eliminated from the table.

Table 4.3 shows the total retrieval time for Recursive Hash Trees as a function of both the initial bucket size and also the bucket size reduction factor. Redundant configurations

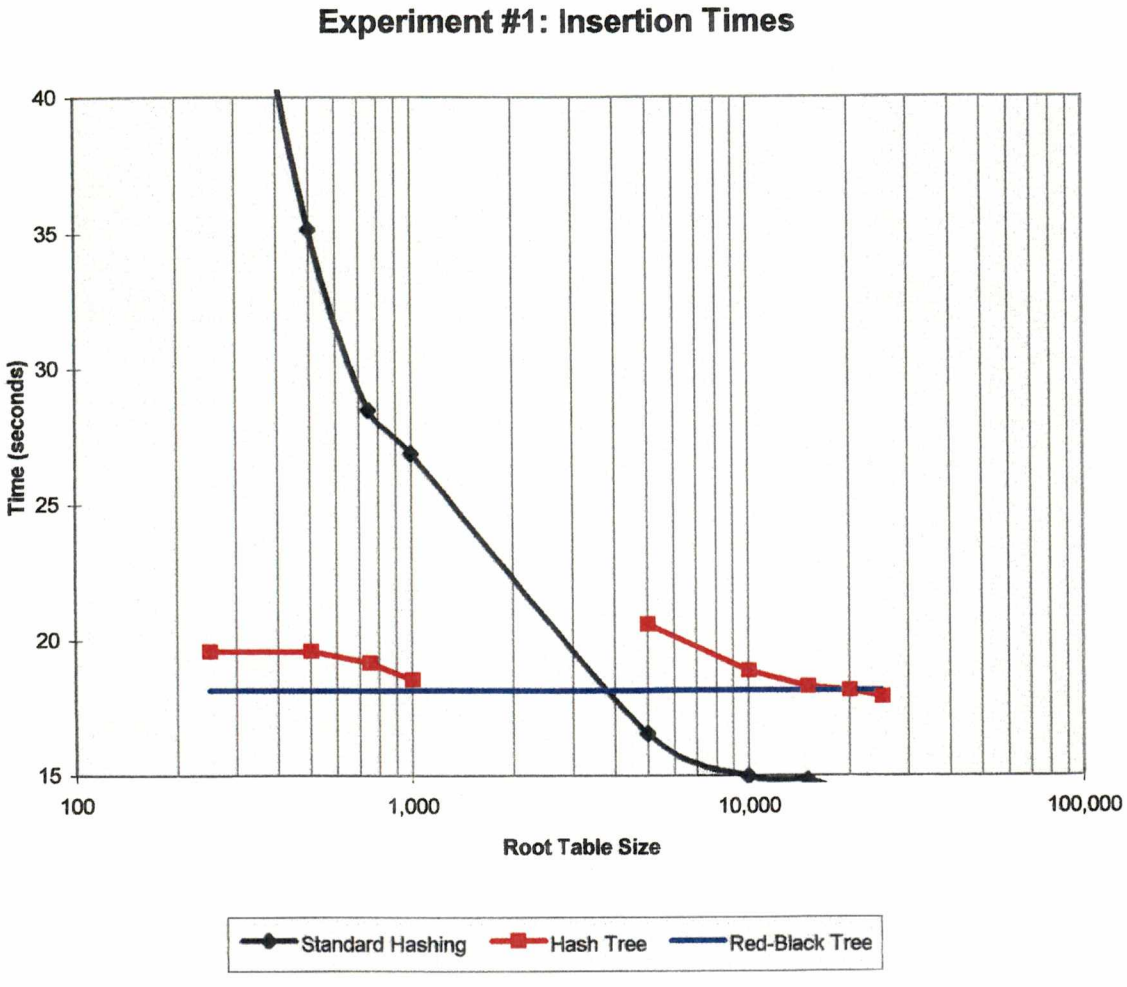

Figure 4.2: Experiment #1: Insertion times

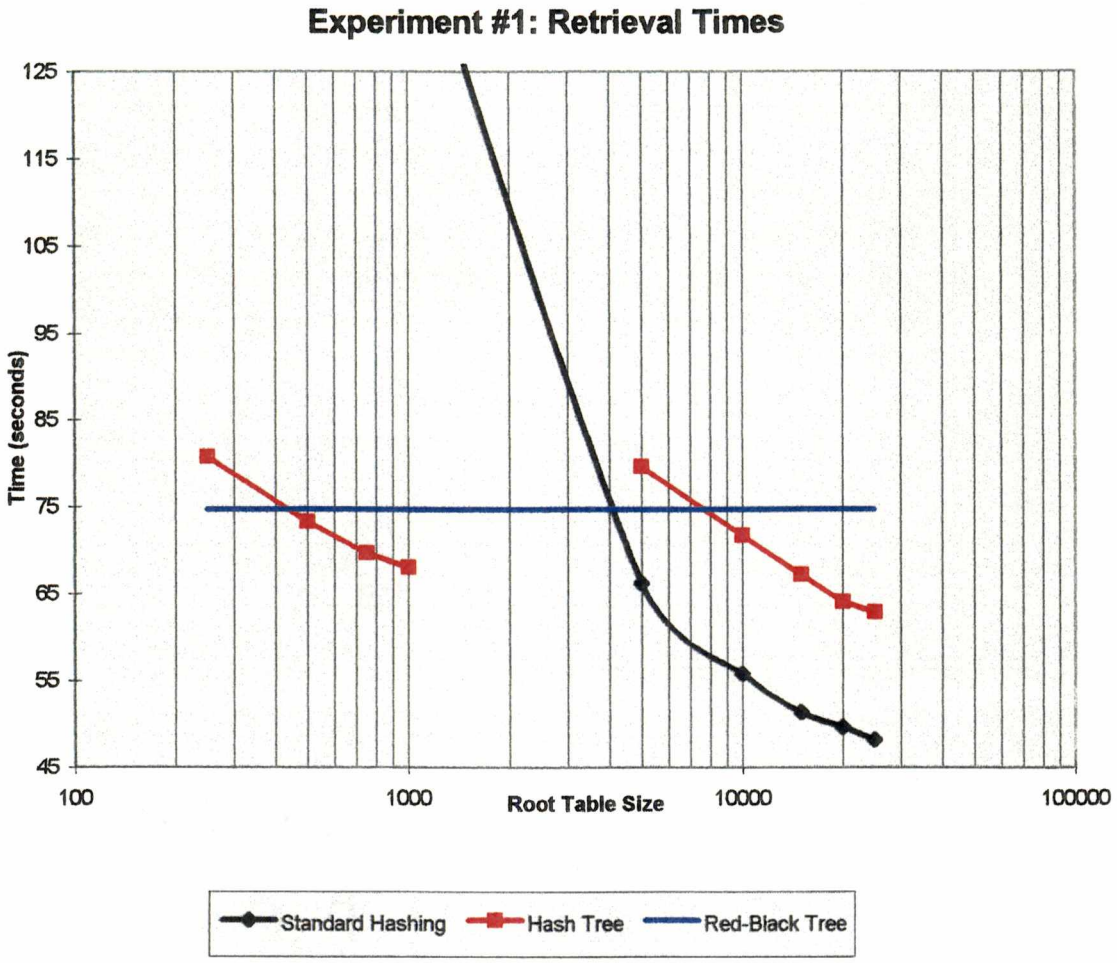

Figure 4.3: Experiment #1: Retrieval times

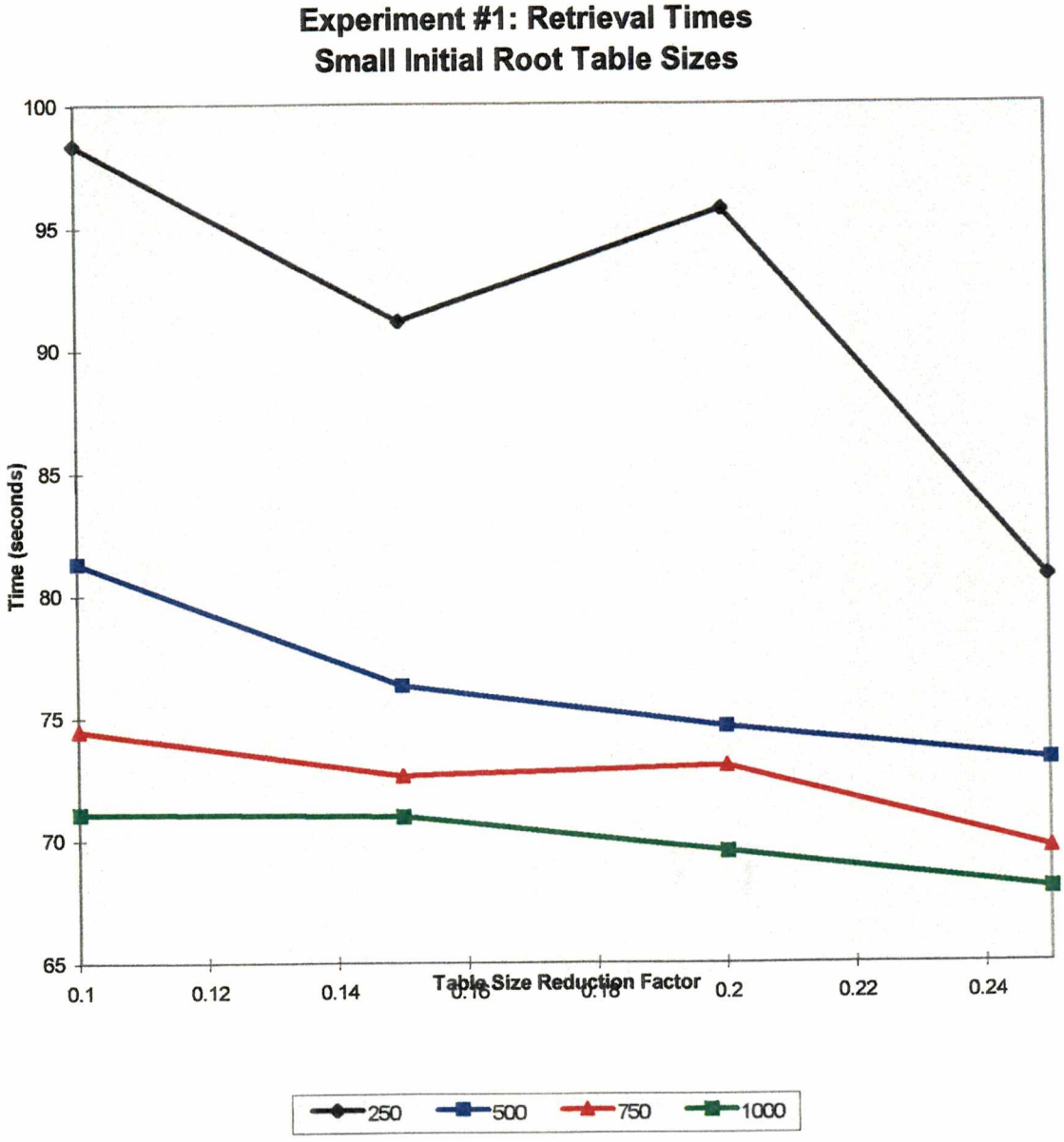

Figure 4.4: Experiment #1: Retrieval times for Recursive Hash Trees with small root table sizes

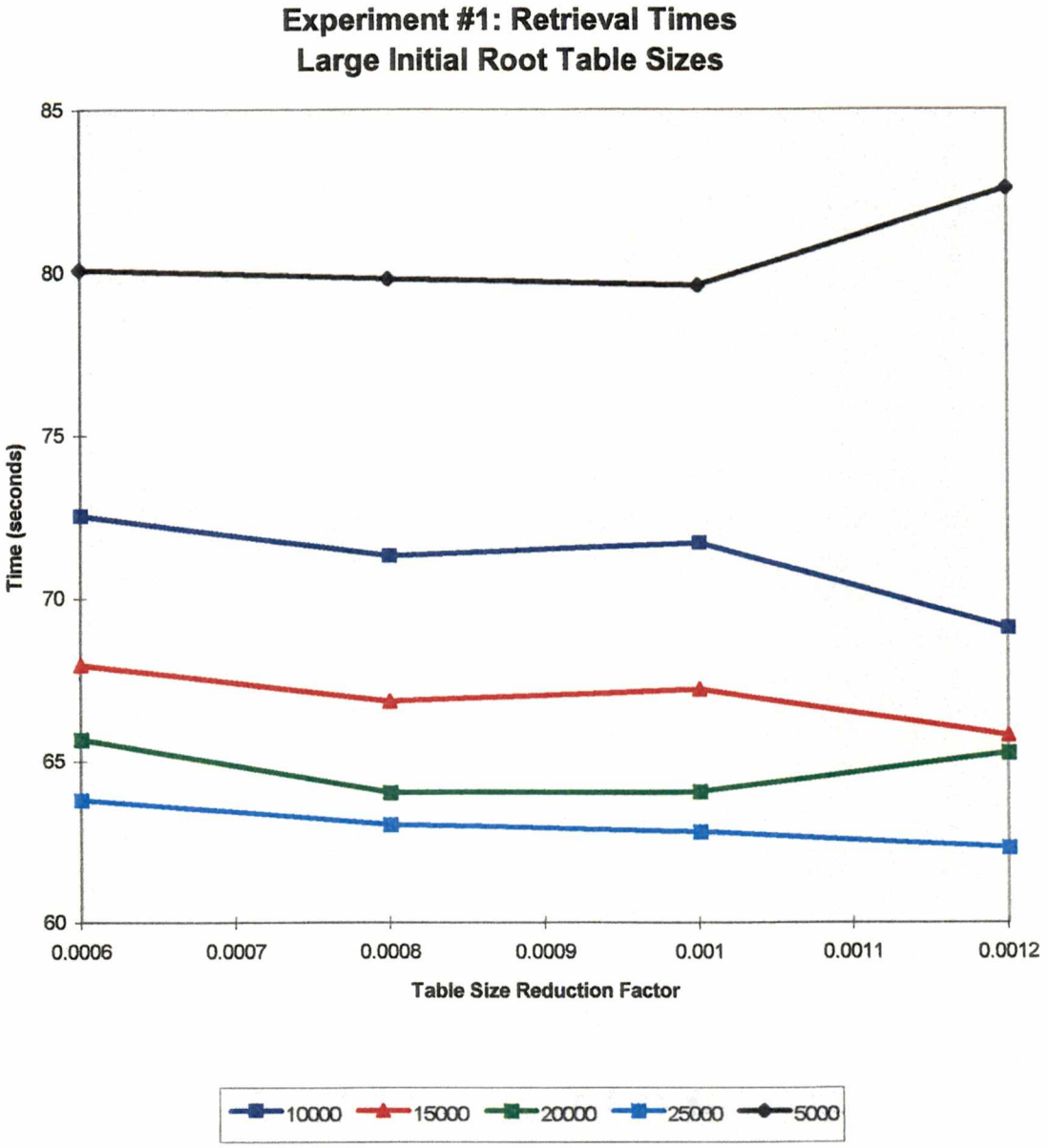

Figure 4.5: Experiment #1: Retrieval times for Recursive Hash Trees with large root table sizes

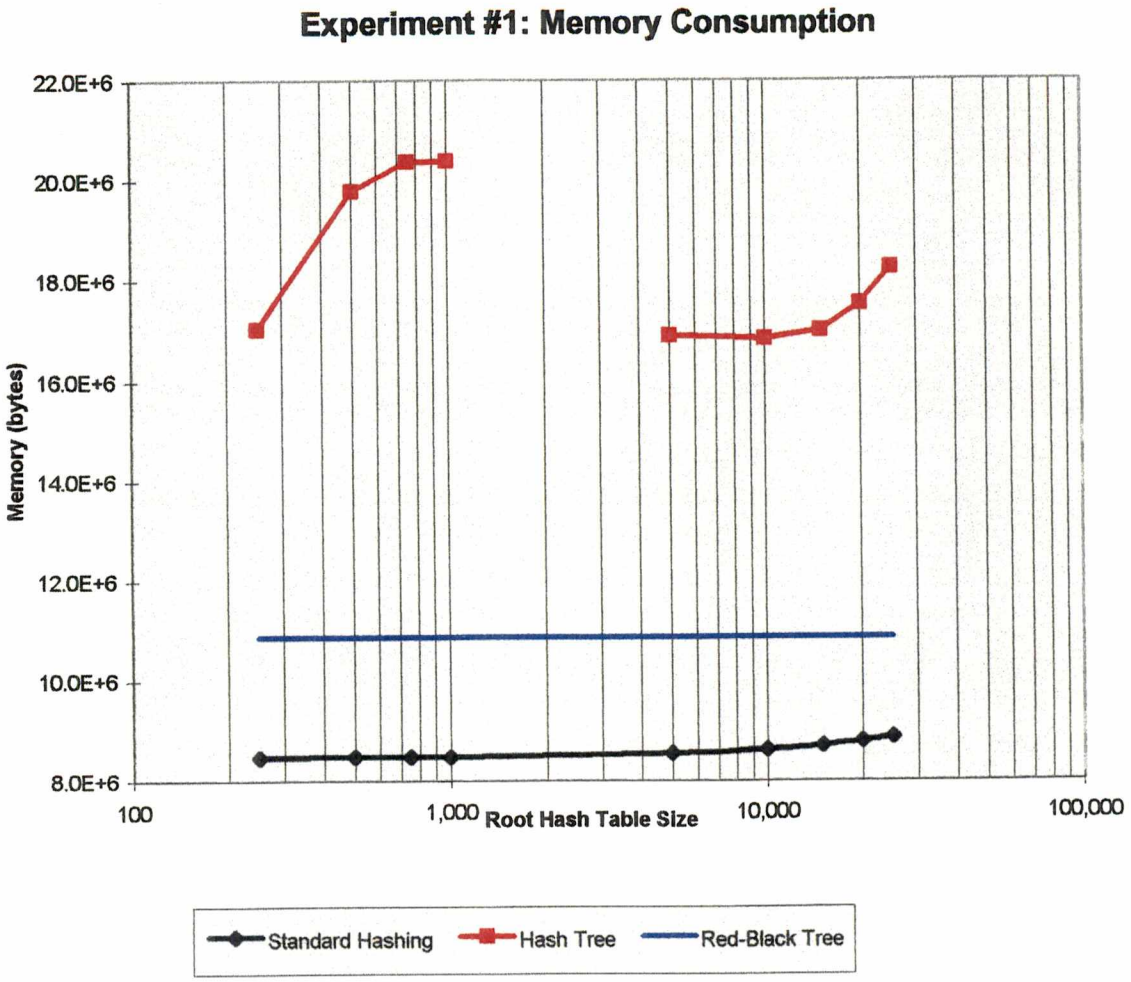

Figure 4.6: Experiment #1: Total memory consumption

Table 4.2: Experiment #1: Memory consumption for Recursive Hash Trees with buckets

Initial Bucket Size	Reduction Factor			
	0	1	2	3
1	20.9			
2	17.4	20.9		
3	17.3	20.6	20.9	
4	18.0	17.2	20.7	20.9

are eliminated from the table.

Table 4.3: Experiment #1: Retrieval times for Recursive Hash Trees with buckets

Initial Bucket Size	Reduction Factor			
	0	1	2	3
1	85.2			
2	83.3	88.0		
3	87.4	90.3	88.2	
4	89.2	87.9	91.5	88.6

Conclusions

The small initial hash tables model respective DMS implementations with poorly estimated dataspace sizes. In the cases of both insertion time and retrieval time, the standard hash tables with chaining performed very poorly. Initially, Red–Black Trees outperformed the other DMS's in both insertion and retrieval rates; however, as the initial root table size grew in the Recursive Hash Tree, performance approached and eventually surpassed that of the Red–Black Tree as shown in Figure 4.3.

Memory utilization was clearly at its worst in the case of the Recursive Hash Tree. The standard hash table with chaining lent the best memory utilization due to the extremely low overhead of the linked list structures.

The introduction of multi–record buckets did help to conserve memory in the Recursive Hash Trees. A sustained bucket size of 2 within the hash tree reduced memory consumption by 16.5% over the hash tree with single–record buckets. A sustained bucket size of 3 continued to conserve memory; however, the marginal increase from 2 to 3 was much smaller than from 1 to 2.

With a sustained bucket size of 2, insertion performance was boosted by approximately 7% while retrieval performance increased only marginally. The increase in insertion performance is do to the lower overhead involved in the creation of one extra node in each bucket than

that of the creation and initialization of a new hash table. This performance boost, however, is not likely to be extremely advantageous in most applications, however, as the speedup will only be encountered during initial bucket creation, a one–time effort.

As the bucket size grew higher than 2, performance decreased noticeably. This effect is due to the fact that the bucket searching algorithm was tuned for bucket sizes of 1 and 2. Consequently, a bucket size of 3 incurs increased overhead in bucket searches.

The anomalies in Tables 4.2 and 4.3 for the bucket size of 4 is due to the diminishing returns associated with inappropriate bucket sizes as described in Section 3.2.2.

Overall, both the Red–Black Tree and the Recursive Hash Tree would be an acceptable choice when given no information about the dataspace. If memory is limited, then the Red–Black tree would be the best choice of these three DMS's since it requires far less memory than Recursive Hash Trees with multi–record buckets.

If the size of the dataspace is known, then the standard hash table with chaining DMS may also provide a good implementation. It must be stressed, however, that standard hash tables are much less forgiving to both inappropriately sized root table sizes and poor hash functions than the Recursive Hash Tree.

4.3.2 Experiment #2: Nonuniform Retrieval Schedule

Experiment #2 uses the same insertion data set as Experiment #1; however, 1,000,000 keys are selected at random for retrieval via a Poisson function with mean of 50,000. The Poisson distribution is first applied to a shuffled version of the original data file so as to avoid weighing the retrievals based upon insertion order. The resulting file is then read for input to generate the subsequent record retrievals.

Experiment #2 introduces an important distinction between the hash table implementations that can reorganize the data interrelationships based upon differing retrieval frequencies, and other other DMS's similar to Red–Black Trees that cannot perform such reorganizations. In Experiment #2, we measure only retrieval times for each DMS while testing different configuration parameters that support this data reorganization. In particular, Table 4.4 identifies the configuration parameters that are used for each DMS.

As in Experiment #1, this experiment will produce noncontiguous results for the Recursive Hash Tree between the root table sizes of 1,000 and 5,000 in order to allow the entire tree to fit in core memory.

Results

The statistics shown in the following figures reflect the configuration parameters as detailed in Table 4.4.

Table 4.4: Experiment #2: Configuration parameters

DMS	CONFIGURATION PARAMETERS
Red–Black Trees	No configuration parameters
Standard Hash Tables	Two types of record promotion: • Move–To–Front • Predecessor access count comparative swapping
Recursive Hash Trees	Record promotion configuration parameters (bucket size = 1): • Maximum promotion distance: 1, 2, 3, 4 • Triggering promotion delta: 1, 10, 100, 1000

There was measurable difference in the retrieval times between the training and post training retrievals only in the smallest initial hash table size and greatest hash table size reduction factor. In this one case, the training retrieval required approximately 3% more time than the post training retrieval.

Figure 4.7 shows the retrieval times during the post–training retrieval runs. In the case of the Recursive Hash Tree, the statistics shown reflect trials with delta = 1; distance = 4.

Table 4.5 shows the retrieval times during the post–training retrieval run of Recursive Hash Trees only. The results shown reflect those record promotion configuration parameters detailed in Table 4.4.

Conclusions

The most notable performance boost from the DMS's was found in the Move–To–Front version of the standard hash tables. The performance of the Red–Black Tree was as expected

Table 4.5: Experiment #2: Post–training retrieval times for Recursive Hash Trees

Distance	Triggering Delta		
	1	10	100
1	71.2	68.3	68.3
2	70.8	68.8	67.9
3	70.4	68.2	68.7
4	70.4	68.2	67.5
5	70.0	68.2	67.8

in that total retrieval time was nearly identical as it was in Experiment #1.

Although the best overall performance was yielded from MTF with the largest root table, even the smallest (250–bucket) initial hash table outperformed all of the other DMS's. This is significant because this same root hash table size exhibited the worst overall performance in Experiment #1. It should be noted, however, that the insertion time for this was still approximately 3 times greater than that of the other DMS's.

With respect to Recursive Hash Trees, there was negligible performance difference as the triggering record promotion delta and the maximum record promotion distance were varied. In light of these findings, it would be best to remove these configuration parameters from the record promotion algorithm. The modification of the record promotion algorithm would result in a lower memory consumption and slightly faster promotion times. Overall 8 bytes per hash node would be conserved. This would result in 800,000 less bytes of core memory that is consumed for this particular data set.

Record promotion in Recursive Hash Trees does seem to be viable. Total retrieval time decreased between 24% and 29%. It is possible, however, that a variation of Move–To–Front record promotion in Recursive Hash Trees might yield better performance in similar conditions as it did in the standard hash table implementation. We leave this modification, however, to future work.

Since the training retrieval added negligible overhead to the retrieval of the records, it

seems reasonable that record promotion could remain enabled without significant performance loss whenever the retrieval schedule is expected to fit a similar model to this Poisson distribution.

4.3.3 Experiment #3: Record Promotion

Experiment #3 uses the same insertion data set as the previous experiments. Following initial insertion, 80,000 keys, selected at random with uniform distribution are deleted from each DMS. Deletion time is measured.

In order to test the efficiency of the resulting data structure, two sets of retrievals are then performed as follows:

- 1,000,000 keys resulting in successful retrieval. These keys are taken from the set of 20,000 keys remaining after deletion.
- 1,000,000 keys resulting in unsuccessful retrieval. These keys are sampled from the set of deleted records.

Total retrieval time is measured for each trial in each DMS.

The same root table sizes are used for Experiment #3 as were used in Experiments #2, shown in table 4.4. As in Experiment #1, Experiment #3 will produce noncontiguous results for the Recursive Hash Tree between the root table sizes of 1,000 and 5,000 in order to allow

the entire tree to fit in core memory.

Results

The statistics shown in the following figures for each DMS reflect the respective configuration parameters as detailed in Table 4.4 with the method for Experiment #3 described above.

Figure 4.8 shows the deletion time of the 80,000 records for each DMS.

Figure 4.9 and Figure 4.10 show the total times for the search for the 1,000,000 records resulting in successful and unsuccessful retrieval, respectively, for each DMS.

Conclusions

Deletion times were most acceptable with Red–Black Trees and the Recursive Hash Trees for all initial root hash table sizes. As in Experiment #2, the standard hash table yielded acceptable results only when the initial root table size was appropriate for the amount of data.

Standard hash tables with chaining were expected to perform poorly in the unsuccessful retrievals because each search, known to fail, would involve a full linear traversal of the appropriate linked list. Actual standard hash table performance fit this prediction as expected.

Performance was generally always better in the searches resulting in successful retrievals. The exception to this was found in the standard hash tables with large initial root table sizes.

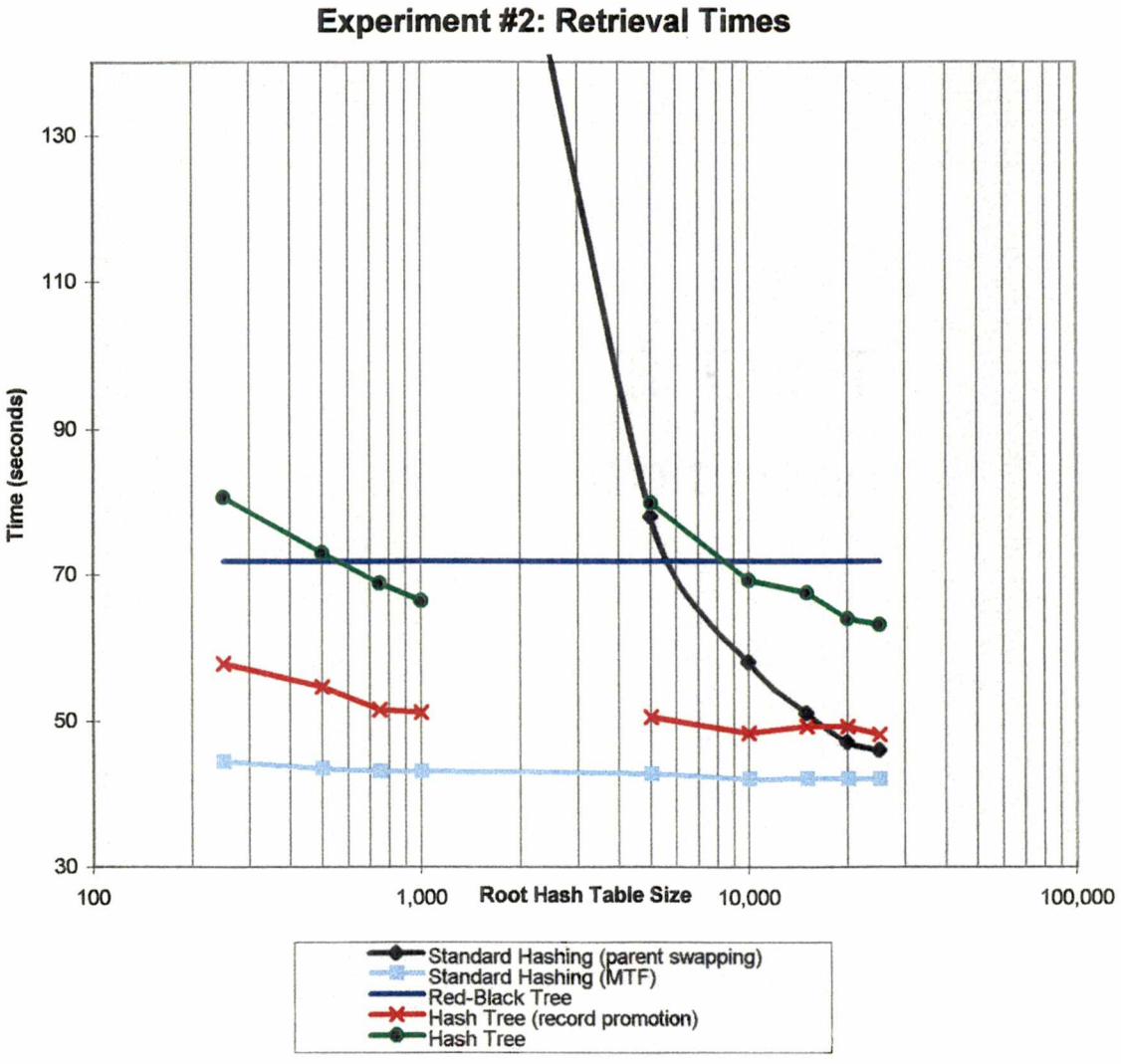

Figure 4.7: Experiment #2: Post–training retrieval times, all DMS's

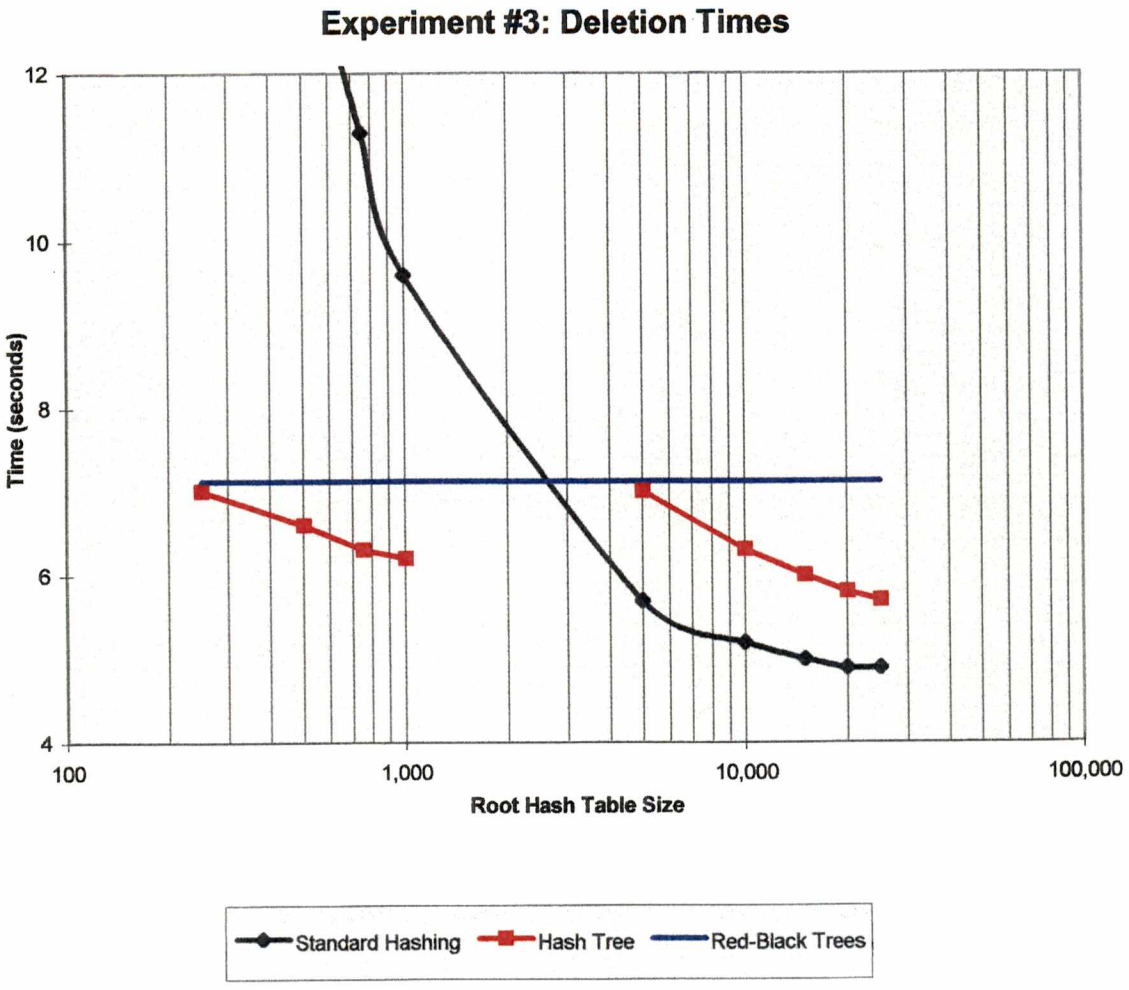

Figure 4.8: Experiment #3: Deletion times

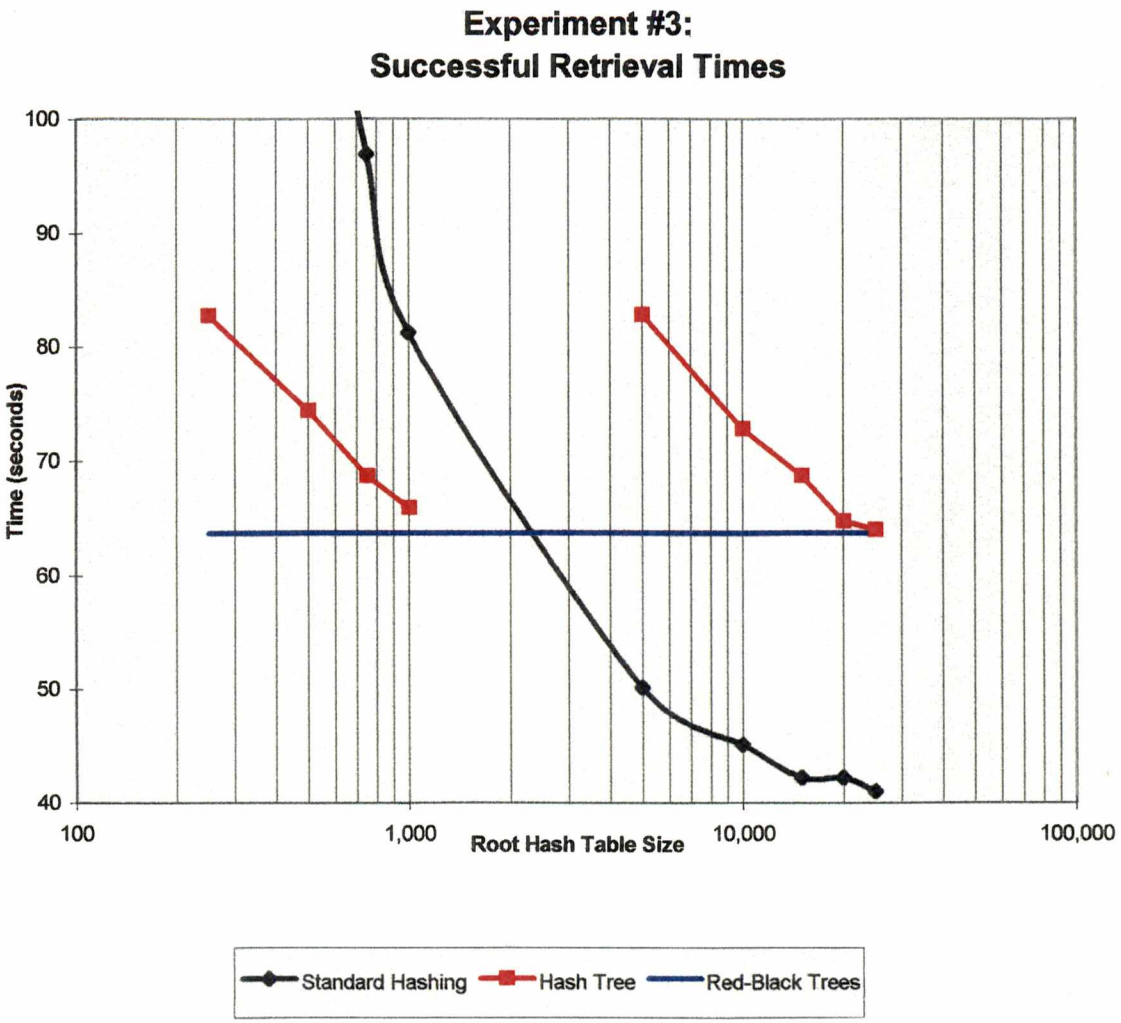

Figure 4.9: Experiment #3: Retrieval (successful) times

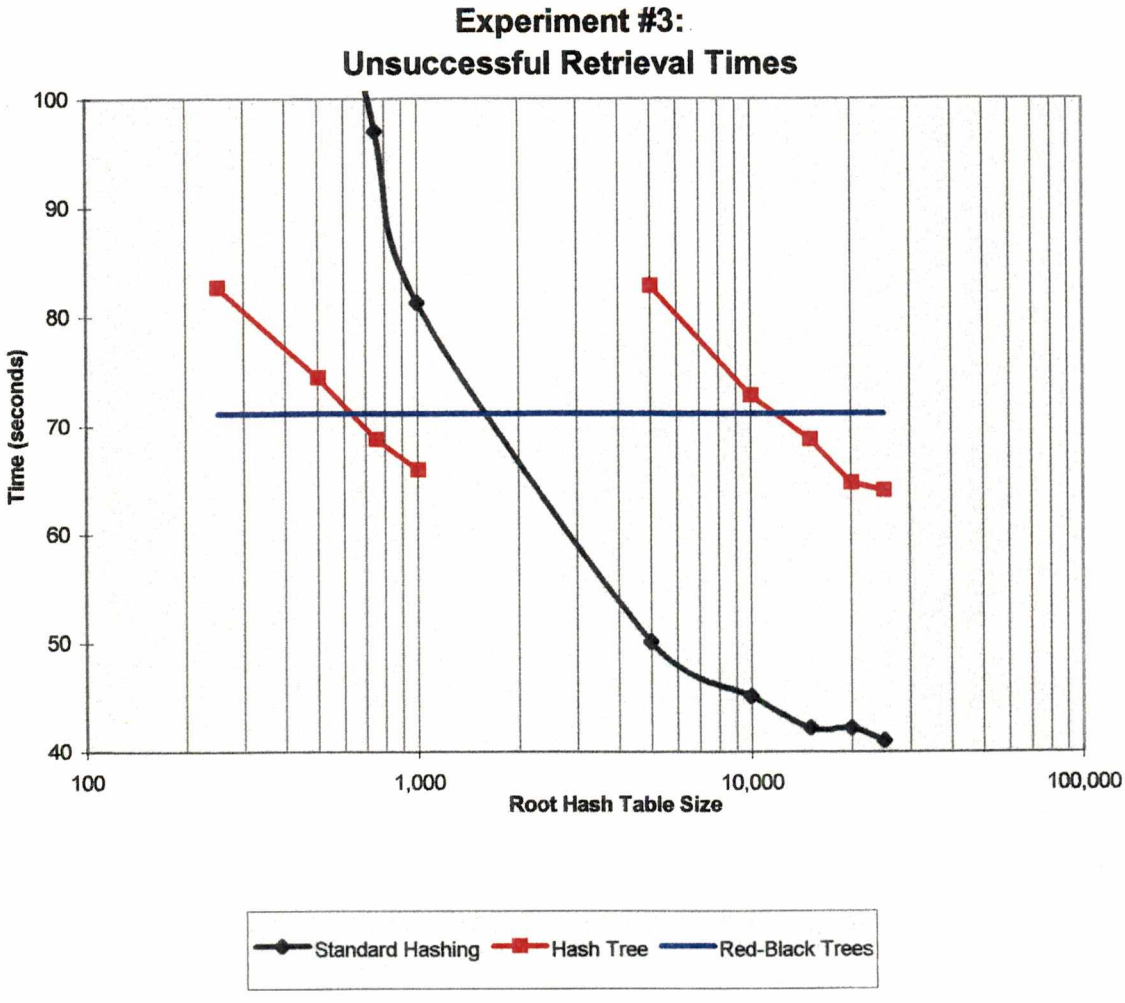

Figure 4.10: Experiment #3: Retrieval (unsuccessful) times

This effect is due to an increased number of empty lists in these larger table sizes. Since there were only 20,000 records remaining in the table following the deletions, the large table sizes allowed an increased number of hash slots to contain zero–length lists. These lists yielded negligible retrieval times since no key comparisons were required.

Similar conclusions can be drawn concerning overall performance from the results of Experiment #3 as were drawn from the results of Experiment #2. In particular, standard hash tables with chaining yielded superior performance in deletion time, time for successful retrieval and time for unsuccessful retrieval only when the root table is sized appropriately. When the root table is not sized appropriately, standard hash tables yield less than adequate performance. The Recursive Hash Trees provide a good overall performance while permitting a greater error margin in the sizing of the initial root table. Red–Black Trees yielded good all-around performance without requiring any information about the size of the dataspace.

Chapter 5

Conclusion

5.1 Concluding Remarks

There are six chief configuration parameters for the Recursive Hash Tree that were evaluated in the experiments as follows:

i) Initial hash table size

ii) Table size reduction factor

iii) Bucket size

iv) Bucket size reduction factor

v) Maximum record promotion distance

vi) Triggering record promotion delta

Initial hash table size most dramatically affected overall DMS performance in both the standard hash table with chaining and also the Recursive Hash Tree. Consequently, this parameter should be given the most consideration when initializing the dataspace. However, the Recursive Hash Tree was much more forgiving to inappropriately sized root table sizes.

Multi–record buckets should only be implemented in the Recursive Hash Tree when core memory is in danger of being exhausted. Since multi–record buckets increase the required number of key comparisons, they increase the overhead to bucket searches and consequently decrease record retrieval performance.

The record promotion configuration parameters presented herein had no significant impact upon retrieval rates. Consequently, these configuration parameters should be removed from the Recursive Hash Tree implementation in order to not only conserve memory but also decrease the overhead that is associated with their implementation.

In applications where it is possible to accurately estimate the dataspace size, the best all–around performance would most likely be obtained from the standard hash tables with chaining. This estimation, however, is based upon the admissibility of the hash function. For while the Recursive Hash Tree responds reasonably well to a poor hash function by modifying the hash function in each new hash table, standard hash tables respond poorly to such a hash function.

Red–Black Tree performance was the most reliable when given no information about the dataspace. This, coupled with the added feature of an inherently sorted dataspace makes Red–Black Trees a "safe" albeit sometimes suboptimal DMS in implementations fitting the model presented herein.

5.2 Future work

There are a number of features that could be added to the Recursive Hash Tree that are beyond the scope of this paper. The following sections present interesting possible areas of further study.

5.2.1 Delayed hashing

Streamlining cannot be guaranteed to be prevented, however, unless there is some dynamic property of the hash function. That is, unless the hash function has somehow gained knowledge a priori of the key space, there is no sure way to prevent streamlining.

One of the assumptions of this work, however, is that there is no knowledge a priori of the dataspace. Consequently, this knowledge must be simulated. One possibility to accomplish this is to delay the actual hashing of the record keys until the hash table has accumulated a number of records. Once a set number or percentage of records has been inserted into the hash table, an appropriate function would be selected. The keys would be hashed once a

good function had been discovered.

Though not implemented herein, delayed insertion seems to be a reasonable method of preventing streamlining. Perhaps results on locating good hash functions could be borrowed from [CBK91] for such a system.

Another interesting method might involve a simplified genetic algorithm to dynamically select a hash function, or parameters to a configurable hash function based upon standard genetic selection criteria.

5.2.2 External access

The system should be extended to support external file access in order to provide non–volatile a dataspace. Minimally, the system should provide a dump/load feature. This extension would allow hash trees to be used in applications that currently utilize **gdbm** or similar library.

5.2.3 Process division

It would be advantageous to create a separate process for the database "engine." The engine would communicate with other processes requesting information via shared memory, local or network sockets, **RPC**, or some other form of interprocess communication. This extension coupled with the external access extension mentioned above would provide all of the functions

necessary to use hash trees in the NIS in Unix.

Albeit this could be implemented at the user–level, it would be beneficial to provide for this feature in the hash tree library because it would allow for other extensions.

5.2.4 Delayed promotion

The effects of immediate promotion during data retrieval should be studied. It is possible that it is more efficient to delay the promotion of records until periods of low activity. Such an extension, however, would probably presuppose the process division extension mentioned above.

5.2.5 Garbage collection

It would be useful to provide for a means of compressing the tree, removing empty holes from deleted or since–promoted records. Such garbage collection is too costly to enact during periods of intense tree activity. Thus, this extension presupposes the process division extension mentioned above.

5.2.6 Data types

If data types were added to the element or node structures, then hardware–aided key comparisons could be achieved in certain cases. For example, it would be more efficient to compare

two integers with a native architecture operations than to generalize into binary comparisons. Likewise for floating point data.

5.2.7 Record promotion

The effect of Move–To–Front style record promotion should be investigated. Since remarkable performance improvements were yielded from MTF within the standard hash table with chaining, similar results might be expected from the Recursive Hash Tree given when presented with similar conditions.

5.2.8 Further testing

Since the results presented herein are drawn from empirical evidence yielded from only one data set with a specific nature, further evaluation should be performed on these DMS with other varieties of data. Possible variations include non text–based data (i.e. binary), larger (longer) record keys and mostly similar record keys. New and different hash functions should be evaluated in combination with other dataspaces.

Cited References

Cited References

[Aoe91] Jun-ichi Aoe, editor. *Computer Algorithms: Key Search Strategies.* IEEE Computer Society Press, Los Alamitos, Calif., 1991.

[CBK91] Nick Cercone, John Boates, and Max Krause. An interactive system for finding perfect hash functions. In Aoe [Aoe91], pages 16–31.

[CLR95] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 15 edition, 1995.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[LC91] Ted G. Lewis and Curtis R. Cook. Hashing for dynamic and static internal tables. In Aoe [Aoe91], pages 2–13.

[Lit81] Witold Litwin. Trie hashing. In Y. Edmund Lien, editor, *ACM - SIGMOD 1981 International Conference on Management of Data, April 29 - May 1*, pages 19–28, New York, New York, 1981. The University of Michigan, Ann Arbor, Michigan, ACM.

[Nel93] Philip A. Nelson. The gnu dbm utility. Free Software Foundation, Inc., 1993. Cambridge, MA.

[Pla] James Plank. Generic red–black trees. WWW free software distribution. Princeton, NJ.

[SM87] Inc. Sun Microsystems. yp - nis protocol. Sun Microsystems 4.1.4 RPC Services Library, October 1987.

[WS92] Larry Wall and Randal Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

Appendices

Appendix A

Data structures

The following data structures for the Recursive Hash Tree are presented in the **C** programming language.

A.1 The Element

```
typedef struct Element element_t;
typedef struct Element_info element_info_t;

struct Element
{
    data_t data;
    u16    length;
};
```

A.2 The Hash Node

```
typedef struct Hash_Node HASH_NODE;
struct Hash_Node
{
    element_t    * key;
    element_t    * value;

    u32            access_count;
    u8             state;
    u8             promote_count;
};
```

A.3 The Bucket

```
typedef struct Hash_Bucket HASH_BUCKET;
struct Hash_Bucket
{
    u8           position;

    HASH_NODE   * nodes;

    HASH_TABLE * hash_table;
    HASH_TABLE * table_point_back;
};
```

A.4 The Hash Table

```
typedef struct Hash_Table HASH_TABLE;
struct Hash_Table
{
    u8  bucket_size;

    u16 size;
    u16 level;

    u32 seed;
    HASH_BUCKET ** buckets;
};
```

A.5 The Hash Tree

```
typedef struct Hash_Tree HASH_TREE;
struct Hash_Tree
{
    u16      (* function)      (data_t, u16, u16, u32);

    struct
          { u32 control; } registers;

    HASH_TABLE * table;
    u8    initial_bucket_size;
    u8    min_bucket_size;
    s8    bucket_reduction;
    u16   min_size;
    u16   active_size;
    u16   initial_table_size;
    u8    max_promote_distance;
    u32   promote_delta;
    float table_decay;
};
```

Appendix B

The Hash Function

The following **C** code presents the hash function that was used for both the Recursive Hash Tree and the standard hash table implementation with chaining.

```
u16 F_ht (data_t data, u16 length, u16 size, u32 seed)
{
    u32 current;
    u16 result;

    current = seed;
    length --;
    while (length--)
    {
        current += seed * (* (data_t) (data + length));
        current = (current << 2) ^ (current >> 2);
    }
    result = current % size;
    return result;
}
```

Vita

Timothy Andrew saw his first employment in McHenry, Illinois as a supporting actor in a not–so major motion picture just moments after first filling his tiny lungs on November 5, 1969. The $20 that he received from this first acting role was invested and later aided the attendance of his Freshman year in collegiate academia. Long before attending college, however, he spent the majority of the intervening years learning what it meant to be fourth in the Martin soup line. The Martins lived in Crystal Lake, Illinois, a small suburb northwest of Chicago.

He did not see lucrative work again until the years of adolescence. Several non–career paths were incited during this time including newspaper delivery, and clerks at both a grocery store and a lumber yard. One miserable summer, he happened into a position as a sole–employee of a fly–by–night food and chemical processing plant where he quickly learned how to manufacture garlic salt, granulated calcium, fertilizer, detergent, red–hot peppers condiments and Tidy–Bowl.

At some point, probably while laminated in layers of the blue antiseptic, he came to the realization that something was not quite right in his life. This moment of truth led him to pursue a degree in Computer Science from Knox College which he went on to achieve in his 21^{st} year.

He was actively involved in athletics, participating in not only high school track and wrestling, but also collegiate wrestling, soccer and lacrosse. At the time of this writing, his current passions include photography, weightlifting and bicycling. He also enjoys playing piano, listening to good music, good food and good drink.