FAST SEARCH IN MAIN MEMORY DATABASES

Anastasia Analyti & Sakti Pramanik *

Computer Science Department
Michigan State University
East Lansing, Michigan 48824

ABSTRACT

The objective of this paper is to develop and analyze high performance hash based search methods for main memory databases. We define optimal search in main memory databases as the search that requires at most one key comparison to locate a record. Existing hashing techniques become impractical when they are adapted to yield optimal search in main memory databases because of their large directory size. Multi-directory hashing techniques can provide significantly improved directory utilization over single-directory hashing techniques. A multi-directory hashing scheme, called fast search multi-directory hashing, and its generalization, called controlled search multi-directory hashing, are presented. Both methods achieve linearly increasing expected directory size with the number of records. Their performance is compared to existing alternatives.

Index Terms : controlled search, extendible hashing, linear hashing, main memory databases, multi-directory hashing, optimal search, performance analysis.

1 INTRODUCTION

Hashing is a well known technique in database systems that permits fast access to both disk-based and main memory-based databases. Significant amount of research has been done on hash based search methods for disk-based files. Much of this research has focused on hashing schemes designed to accommodate files with dynamically changing size. Those include dynamic hashing [5], extendible hashing [2], linear hashing [13] and improved versions of it [6,7,9] and perfect hashing [10,15].

*This work was supported by National Science Foundation Grant No. CCR-8706069

Hashing schemes for disk-based databases have been designed with the assumption that data reside on the disk during transaction processing. However, substantial performance gains can be achieved when data reside in main memory. The rapidly decreasing cost of RAM makes main memory databases a cost-effective solution to high performance data management. An experimental transaction processing system for memory resident databases is described in [16]. Clearly, access methods designed for disk-based databases may not be suitable for main memory databases. This is due to the fast and random access capability of main memory databases. The retrieval time in main memory databases consists of the time to index the directory plus the time to compare the key values. In contrast, the retrieval time in disk-based databases is computed based mainly on the number of disk accesses. We define *optimal search* in main memory databases as the search that requires at most one key comparison to locate a record.

The objective of this paper is to develop and analyze high performance hash based search methods for main memory databases. Access methods in main memory databases have been investigated in [1,17,11,12,18]. DeWitt at al. [1] compare the performance of conventional data structures when large main memory is available. Shapiro [17] studies algorithms for computing the equijoin of two relations in a system with large amount of main memory. Lehman and Carey [11,12] propose a new index structure, called T tree, and present algorithms for selection and join processing in memory resident databases. Whang and Krishnamurthy [18] present techniques for optimizing queries in memory resident database systems.

The major contribution of this paper is the development of two hash based access methods, namely, *fast search multi-directory hashing* (FSMH) and *controlled search multi-directory hashing* (CSMH). These two methods are of most interest to applications requiring fast retrieval time. FSMH requires at most one key comparison to locate a record when hash addresses are unique. When hash addresses are uniformly distributed in a bounded interval, the method requires a near optimal number of key comparisons to locate a record. CSMH is a generalization of FSMH. In CSMH, the expected number of key comparisons for a successful search is bounded by a control parameter.

FSMH and CSMH use a tree-structured hash directory to access the data file. The hash directory consists of a set of single subdirectories that adapt their size dynamically to a changing number of records. This dynamic adaptation does not require any major reorganization of the directory structure because insertion and deletion of a record affect only one node of the tree. The authors have developed both analytical and experimental models for these techniques to compare their performance with existing methods.

Both FSMH and CSMH achieve linearly increasing directory size with the number of records. The expected number of index accesses to locate a record is in $O(1)$. FSMH obtains smaller directory size than extendible hashing [2], dynamic hashing [5] and M-tries [4] when those are modified to yield optimal search in main memory. CSMH yields smaller directory size than linear hashing [13,8] and any single-directory hashing model in the high performance range of the control parameter.

The rest of the paper is organized as follows. Section 2 presents a theoretical simple multi-directory hashing scheme. Section 3 presents FSMH and compares its performance with those of other existing hashing models. Section 4 presents CSMH and compares its performance with single-directory hashing models. Section 5 contains concluding remarks.

2 A SIMPLE MULTI-DIRECTORY HASHING SCHEME

In this section, we present a dynamic hashing scheme, called *simple multi-directory hashing* (SMH). The purpose is to provide justifications for multi-directory hashing schemes whose subdirectories adapt their size dynamically to a changing number of records. We assume a hash function with characteristics similar to that in extendible hashing [2].

Optimal search can be obtained by extendible hashing with bucket size 1 because in this case, at most one key comparison is needed to locate a record. A record is located by using the d least significant bits of its hash address to index the hash directory of size 2^d. When a collision occurs, the directory doubles in size and its depth d increases by one. Figure 1 presents an example directory of depth 3. In the figure, $R(x)$ represents a record where x is a sequence of its least significant hash address bits. Flajolet [3] has shown that the expected directory size of extendible hashing with bucket size $b > 1$ is in $O(m^{1+\frac{1}{b}})$, where m is the number of inserted records. We will show that when the single directory of extendible hashing is replaced by multiple subdirectories, the expected directory size decreases significantly.

In SMH, a tree-structured multi-directory of height one is used to access the data file. Each leaf subdirectory is located through the root subdirectory of the directory tree. The size of the root subdirectory equals the number of leaf subdirectories which is fixed. Here, the hash address of a record is partitioned into two parts. The rightmost part is used to index the root subdirectory and locate the leaf subdirectory and the leftmost part is used to index the leaf subdirectory. Thus, two index accesses are needed to locate a record. Each leaf subdirectory grows in the same way as the directory of extendible hashing to allow for more records. When the number of leaf subdirectories is 1, SMH coincides with extendible hashing. Figure 2 illustrates the directory structure of SMH. We present the insertion algorithm of SMH with N leaf subdirectories.

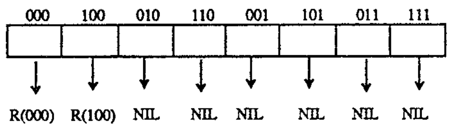

Figure 1: An example directory of extendible hashing with bucket size 1

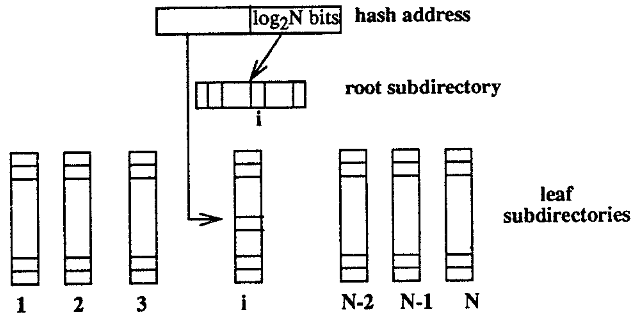

Figure 2: Directory structure of SMH

Insertion algorithm of SMH:

1. Obtain the hash address h of the record. Use the $\lceil log_2 N \rceil$ least significant bits of the hash address h to locate the leaf subdirectory i. Use the remaining of the bits of the hash address to index the leaf subdirectory i.

2. If there is space in the bucket corresponding to the hash address h, insert the record into the bucket and stop.

3. If the bucket corresponding to the hash address h is fully occupied and it is pointed to by more than

one directory entries, split the bucket into two as in extendible hashing and stop.

4. If neither of the conditions in step 1 and step 2 is true then double the leaf subdirectory i as in extendible hashing. Split the bucket corresponding to the hash address h into two and go to step 2.

The directory size of SMH consists of the size of all leaf subdirectories plus the root subdirectory size. The expected directory size of SMH with N leaf subdirectories, m records and bucket size b is denoted as $D_{SMH}(m, N, b)$. The following proposition characterizes the expected directory size of SMH when $b > 1$.

Proposition 1.
The expected directory size of SMH with bucket size $b > 1$, decreases as the number of leaf subdirectories N increases up to $N_{min}(m) = (\frac{3.92}{b^2})^{\frac{b}{b+1}}m$, where m is the number of inserted records. When the number of leaf subdirectories is $N_{min}(m)$, the directory size reaches a minimum value, approximately equal to $3.92^{\frac{b}{b+1}}(b^{\frac{1-b}{b+1}} + b^{\frac{-2b}{b+1}})m$.
Proof:
Let $D_{EH}(m, b)$ denote the expected directory size of extendible hashing. Flajolet has given a coarse approximation of the expected directory size of extendible hashing [3],

$$D_{EH}(m, b) \approx \frac{3.92}{b} m^{1+\frac{1}{b}}$$

We assume that the average number of records per subdirectory, $\frac{m}{N}$ is large. Then, the expected size of each subdirectory can be approximated by $D_{EH}(\frac{m}{N}, b)$. This approximation is justified because the number of records hashed into a subdirectory is a binomial random variable with mean $\frac{m}{N}$. Thus, the expected directory size of SMH is approximated by

$$D_{SMH}(m, N, b) \approx N\frac{3.92}{b}(\frac{m}{N})^{1+\frac{1}{b}} + N \qquad (1)$$

$D_{SMH}(m, N, b)$ is minimized when $\frac{\partial D_{SMH}(m,N,b)}{\partial N} = 0$. That is, when

$$N = N_{min}(m, b) = (\frac{3.92}{b^2})^{\frac{b}{b+1}} m$$

It follows that

$$\min_N D_{SMH}(m, N, b) \approx 3.92^{\frac{b}{b+1}}(b^{\frac{1-b}{b+1}} + b^{\frac{-2b}{b+1}})m.\Box$$

SMH with bucket size 1 yields optimal search in main memory databases because directory entries point to at most one data record. In Proposition 1, the bucket size of SMH is greater than 1. We will show that the expected directory size of SMH with bucket size 1 still decreases as the number of leaf subdirectories increases. The directory size of extendible hashing with bucket size 1 is 2^k if the smallest number of least significant

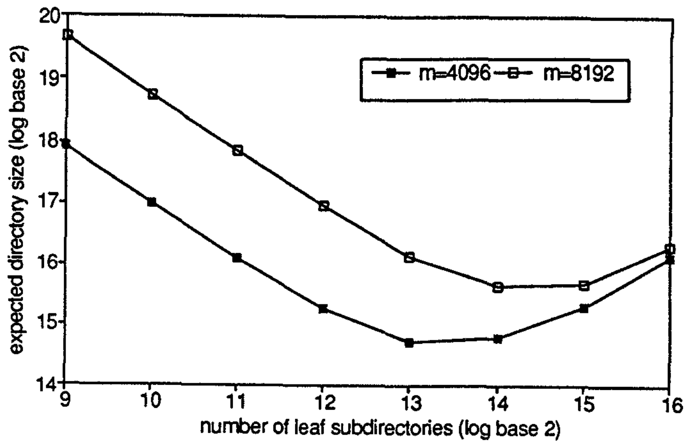

Figure 3: Expected directory size of SMH with bucket size 1

bits needed to differentiate the hash addresses of the inserted records is k. Let P_k^m be the probability that m hash addresses, uniformly distributed over the infinite space of non-negative integers, will be uniquely identified by their k least significant bits. This probability is given by the expression:

$$P_k^m = \frac{2^k(2^k-1)...(2^k-m+1)}{(2^k)^m}, \; k \geq \log_2 m$$

We first give a lemma, which states an important property of the probabilities P_k^m.

Proposition 2.

$$P_k^m = P_{k+\ell}^{(\sqrt{2})^\ell m} + O(\frac{m}{2^k}), \text{ for any positive integer } \ell \text{ and } k > \log_2 m.$$

Proof: It is given in the Appendix.□

Proposition 2 implies that doubling the number of hashed records increases the number of hash address bits needed to differentiate the hash addresses by 2. This implies that the expected directory size of extendible hashing with bucket size 1 is in $O(m^2)$. Following the steps of the proof of Proposition 1, we can show that the expected directory size of SMH with bucket size 1 almost halves when the number of leaf subdirectories doubles, assuming that $\frac{m}{N}$ is large enough.

The above analysis is verified by our experimental results. The expected directory sizes were computed by averaging the results over 50 different runs. The keys used in each run were generated by a random number generator that produces uniformly distributed integers. Figure 3 gives the expected directory size of SMH with bucket size 1 for different number of leaf subdirectories and records m. Note that the expected directory size of SMH decreases as the number of leaf subdirectories increases up to a certain value $N_{min}(m)$.

SMH is a theoretical model. The expected directory size of SMH decreases with the number of leaf subdirectories N and its minimum value, obtained for $N = N_{min}(m)$, is in $O(m)$. However, the expected directory size of SMH, given in expression (1), is in $O(m^{1+\frac{1}{b}})$ when the number of leaf subdirectories is fixed. In SMH, the number of leaf subdirectories and size of the root subdirectory are fixed. Sections 3 and 4 describe two multi-directory hashing models whose subdirectories adapt their size dynamically to a changing number of records. Those models achieve linearly increasing expected directory size with the number of records.

3 FAST SEARCH MULTI-DIRECTORY HASHING

This section describes and analyzes the performance of *fast search multi-directory hashing* (FSMH). We assume that data records reside in the main memory and that they could be scattered across all of the memory. FSMH yields optimal search in main memory databases. In Section 2, we showed that SMH obtains smaller directory size than extendible hashing. However, to obtain minimal directory size the number of the leaf subdirectories and consequently the root subdirectory size should be adapted to the number of hashed records. This gives rise to the idea of a multi-directory hashing model with adaptable subdirectory size. In FSMH, all subdirectories may be created, destroyed, expanded or shrunk as records are inserted or deleted. The objective is to achieve small directory size with reasonable reorganization cost. FSMH obtains linearly increasing directory size with the number of records. The expected number of index access to locate a record is small. The next two subsections describe FSMH and analyze its performance.

3.1 DESCRIPTION OF FAST SEARCH MULTI-DIRECTORY HASHING

In FSMH, a tree-structured hash directory is used to access the data records. All nodes of the tree are single subdirectories. Subdirectory entries point either to a lower lever subdirectory or to a data record. The size and the height of the hash directory are variable. To access a record, the hash address bit sequence of the key is partitioned into distinct parts. Each part is used to index a subdirectory at a particular level. An example FSMH directory is given in Figure 4. To access the record $R(1000)$, its hash address bit sequence '1000' is broken into three parts, '10', '0' and '00'. The bit sequence '00' indexes the root subdirectory. The bit sequence '0' indexes the corresponding child subdirectory and the bit sequence '10' indexes the corresponding leaf subdirectory. FSMH yields optimal search because directory entries point to at most one data record.

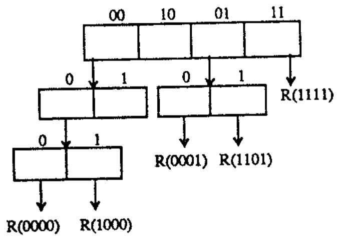

Figure 4: An example directory of FSMH

When a collision occurs, the directory should expand so that it can differentiate between the conflicting records. That is, either the conflicting subdirectory should double or a child subdirectory should be created and linked to the conflicting entry. In FSMH, this choice is made dynamically according to case. Upon collision, the number of children subdirectories of the conflicting node is examined. If this number is less than half the conflicting node size, a binary tree of height $(h-1)$ is created, where h is the number of additional hash address bits needed to differentiate between the conflicting records. This tree is attached to the conflicting entry. Each node of the attached binary subtree has two entries, one pointing to a child node and the other being empty. The conflict is resolved by having the two entries of the leaf node of the attached subtree point to the conflicting records. For example, in the directory of Figure 4, a binary subtree of height 1 was created to differentiate between the records $R(0000)$ and $R(1000)$.

If the number of children subdirectories of the conflicting node is equal to half of its size, we decide to double the conflicting node. This is because if the conflicting node doubles, children subdirectories of size two will be merged with the node, thus reducing the directory size and the expected index access time. This gain will compensate for the cost of memory space that results from doubling the conflicting subdirectory. The conflicting node keeps doubling until the conflict is resolved or the size of the node becomes more than twice the number of children nodes. If the conflict persists, a binary tree of height $(h-s-1)$ is created as before, where s is the number of times the conflicting node has doubled. When a node doubles, children subdirectories of size larger than two split into two buddy subdirectories. Figure 5 shows the result of inserting the record $R(0111)$ into the directory of Figure 4. Note that a record conflict will only require the conflicting node to be restructured. The height of the directory may grow or shrink as records are inserted. This results in a small and bounded expected number of index accesses to locate a record.

In the above description we have assumed that there will always exist a number of hash address bits to differentiate between two conflicting records. At the end of this subsection we will prove that for practical hash

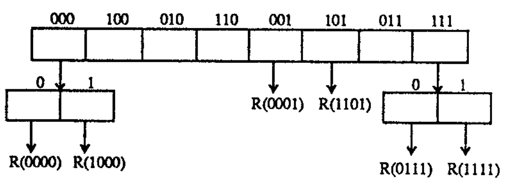

Figure 5: The directory of FSMH after inserting $R(0111)$ into the directory of Figure 4

functions the expected number of collisions that fail the above assumption is negligible.

When deletions form a considerable portion of the operations of an application, space can be saved by coalescing the buddy entries of subdirectories that underflow. We say that a subdirectory underflows if for any two subdirectory buddy entries, one is empty.

The steps of two algorithms for searching and inserting a record, called SEARCH and INSERT respectively, are given in Figure 6. Record keys are assumed to be unique. The routine SEARCH is recursive and is called to search for a record with key K, using the bit sequence H to index the subtree with root node S. The first time that SEARCH is called, S is the root of the tree-structured multi-directory and H is the hash address bit sequence of the record to be searched for. The SEARCH routine returns the address of the requested record or terminates unsuccessfully when it cannot locate the record. The INSERT routine is also recursive and it needs one additional parameter, the location of the record R to be inserted. If no record with the same key is found, a pointer to record R is inserted into the appropriate place of the multi-directory. Otherwise, the routine terminates unsuccessfully. Unless otherwise indicated, all variables in the routines SEARCH and INSERT are local variables. The global variable t is initialized to zero before the first invocation of the INSERT routine.

We have assumed that unique hash addresses are guaranteed. Since in practice hash functions produce uniformly distributed hash addresses in a bounded interval $[0, 2^b - 1]$, we will see how the above algorithms have to be modified. We will first calculate the expected number of collisions that are not resolved in b bits. We will show that this number, denoted as $O(m, b)$ is very small for a large b. In this case the problem of no unique hash addresses in the interval $[0, 2^b - 1]$ can be solved using a simple chaining scheme. In this scheme, pointers to records with the same hash address are linked together forming a chain.

The number of records N that are hashed into a particular hash address can be considered as a Poisson distributed random variable with mean $\lambda = \frac{m}{2^b}$. According

```
SEARCH(location of subdirectory $S$, bit sequence $H$, key $K$)
  1. Read the depth $d$ of the subdirectory $S$.
  2. Take the $d$ least significant bits of $H$ and find the corresponding entry $E$ of $S$.
  3. If the entry $E$ points to a child subdirectory $S'$ do the following:
     (a) Set $H'$ to the value of $H$ after it has shifted to the right $d$ bits.
     (b) Call SEARCH$(S', H', K)$.
  4. If the entry $E$ points to a record with key equal to $K$, return the address of the record.
  5. If the entry $E$ is empty or point to a record with key other than $K$, terminate unsuccessfully.

INSERT(location of subdirectory $S$, bit sequence $H$, key $K$, location of record $R$)
  1. Read the depth $d$ of the subdirectory $S$.
  2. Take the $d$ least significant bits of $H$ and find the corresponding entry $E$ of $S$.
  3. If the entry $E$ points to a child subdirectory $S'$ then do the following:
     (a) Set $H'$ to the value of $H$ after it has shifted to the right $d$ bits.
     (b) Increase the global variable $t$ by $d$. (The variable $t$ contains the total number of bits that the initial hash address has been shifted to the right.)
     (c) Call INSERT$(S', H', K, R)$.
  4. If the entry $E$ is empty, have the entry point to $R$ and terminate successfully.
  5. If the entry $E$ points to a record $R''$ with key $K''$ other than $K$ then do the following (Collision has occurred) :
     (a) If the number of children subdirectories of $S$ is less than half the size of $S$ then do the following:
         i. Create a subdirectory $S'$ of size 2 and link it to the entry $E$.
         ii. Set $H'$ to be the value of $H$ after it has shifted to the right $d$ bits.
         iii. Increase the global variable $t$ by $d$.
         iv. Set $H''$ to be the value of the hash address of $R''$ after it has shifted to the right $t$ bits.
         v. Call INSERT$(S', H', K, R)$.
         vi. Call INSERT$(S', H'', K'', R'')$.
     (b) If the number of children subdirectories of $S$ is half the size of $S$ then do the following:
         i. Double the subdirectory $S$. (Child subdirectories of size 2 are incorporated into the parent and those of larger size split into two buddy subdirectories.)
         ii. Set $H''$ to be the value of the hash address of $R''$ after it has shifted to the right $t$ bits.
         iii. Call INSERT$(S, H, K, R)$.
         iv. Call INSERT$(S, H'', K'', R'')$.
     (c) If the entry $E$ points to a record with key equal to $K$, terminate unsuccessfully.
```

Figure 6: SEARCH and INSERT algorithms of FSMH

to the Poisson distribution

$P(N=k)=e^{-\lambda}\frac{\lambda^k}{k!}$, where k is a non-negative integer

Hence,

$$\begin{aligned} O(m,b) &= 2^b\sum_{k=2}^{\infty}((k-1)e^{-\lambda}\frac{\lambda^k}{k!} = \\ &= 2^b(\frac{m}{2^b}+e^{-\frac{m}{2^b}}-1) \end{aligned}$$

Since $O(m,b) \simeq \frac{m^2}{2^{b+1}}$ for $m \ll 2^b$, $O(m,b)$ can be considered very small for $m \ll 2^b$. For example $O(m,b)=0.429$ for $b=32$ and $m=65,536$. To handle collisions not resolved in b bits, we use separate chaining. That is, pointers to records with the same hash address are linked together forming a separate chain. More than one key comparisons are needed to locate the records in the tail of the chain. However, the expected number of key comparisons for successful search, denoted as $S(m,b)$, may be considered equal to 1.0 for $m \ll 2^b$. More specifically,

$$\begin{aligned} S(m,b) &= \frac{2^b}{m}\sum_{k=1}^{\infty}\frac{k*(k+1)}{2}e^{-\lambda}\frac{\lambda^k}{k!} = \\ &= 1+\frac{m}{2^{b+1}} \end{aligned}$$

3.2 PERFORMANCE OF FAST SEARCH MULTI-DIRECTORY HASHING

We first prove that the expected directory size of FSMH increases in linear order with respect to the number of hashed records.

Let 2^r be the size of the root node and $\alpha(m)=\frac{m}{2^r}$, the expected number of records hashed into a particular root node entry. A child subdirectory is created when more than one records are hashed into the same entry. Note that $\sum_{k=2}^{\infty}e^{-\alpha(m)}\frac{\alpha(m)^k}{k!}=1-e^{-\alpha(m)}-\alpha(m)e^{-\alpha(m)}$ is the probability that more than one records are hashed into a particular directory entry. For simplicity, we assume that the number of child subdirectories of the root node always equals the expected number of child subdirectories of the root. Because the number of child subdirectories is not greater than half the parent node size

$$\begin{aligned} 2^r(\sum_{k=2}^{\infty}e^{-\alpha(m)}\frac{\alpha(m)^k}{k!}) &\leq \frac{2^r}{2} \Leftrightarrow \\ 1-e^{-\alpha(m)}-\alpha(m)e^{-\alpha(m)} &\leq \frac{1}{2} \Leftrightarrow \\ \frac{m}{2^r} &\leq \alpha^* \end{aligned}$$

where $\alpha^* \simeq 1.68$ satisfies the equation $1-e^{\alpha^*}-\alpha^*e^{-\alpha^*}=\frac{1}{2}$.
When the number of child subdirectories of the root subdirectory becomes equal to half the root node size, the root node doubles and $\alpha(m)$ halfs. After this, $\alpha(m)$ starts to increase again. So,

$$\begin{aligned} \frac{\alpha^*}{2} &\leq \frac{m}{2^r} \leq \alpha^* \Leftrightarrow \\ r &= \lceil\log_2(\frac{m}{\alpha^*})\rceil \end{aligned}$$

where $\lceil x\rceil$ denotes the smallest integer larger than or equal to x.
Replacing r in the expression $\alpha(m)=\frac{m}{2^r}$, we get

$$\alpha(m)=\frac{m}{2^{\lceil\log_2(\frac{m}{\alpha^*})\rceil}}$$

Note that $\alpha(2^\ell m)=\alpha(m)$, where ℓ is any non-negative integer. Let the expected directory size for m records be denoted by $D(m)$ and the expected size of a root node child subtree be denoted by $D'(m)$. Then

$$D(m)=\frac{m}{\alpha(m)}+\frac{m}{\alpha(m)}(1-e^{-\alpha(m)}-\alpha(m)e^{-\alpha(m)})D'(m) \qquad (2)$$

since $\frac{m}{\alpha(m)}$ is the root node size and $\frac{m}{\alpha(m)}(1-e^{-\alpha(m)}-\alpha(m)e^{-\alpha(m)})$ is the expected number of children subtrees of the root node.
The expected number of records hashed into a child subtree is

$$\begin{aligned} \alpha'(m) &= \frac{\sum_{k=2}^{\infty}ke^{-\alpha(m)}\frac{\alpha(m)^k}{k!}}{\sum_{k=2}^{\infty}e^{-\alpha(m)}\frac{\alpha(m)^k}{k!}} \\ &= \frac{\alpha(m)(1-e^{-\alpha(m)})}{1-e^{-\alpha(m)}-\alpha(m)e^{-\alpha(m)}} \end{aligned}$$

where m is the number of hashed records.
Note that $\alpha'(2^\ell m)=\alpha'(m)$, where ℓ is any non-negative integer. Since

$$D'(m)=\sum_{k=2}^{\infty}D(k)e^{-\alpha'(m)}\frac{\alpha'(m)^k}{k!}$$

we derive that

$$D'(2^\ell m)=D'(m) \qquad (3)$$

From relations (2) and (3) and the fact, $\alpha(2^\ell m)=\alpha(m)$, it follows

$D(2^\ell m)=2^\ell D(m)$, where ℓ is a non-negative integer.

The above relation implies that as the number of hashed records doubles, the expected directory size doubles. That is, the expected directory size of FSMH increases linearly with the number of records.

Simulation results show that the general performance of the method also does not change as the number of records doubles. Performance is shown to be a periodic function of $\log_2 m$, with period 1, where m is the number of hashed records. However, its cyclic oscillations are considered very small. Table 1 gives the performance of FSMH. The expected values in the table were obtained by averaging the results over 150 different data sets. Each set was generated by changing the seed of the random number generator. The random number generator produces integers uniformly distributed over the interval $[0, 2^{31} - 1]$. Values for only one period are given because the performance is periodic. The standard deviation of the directory size is less than 2.6% of the expected directory size.

	$\log_2 m - \lfloor \log_2 m \rfloor$			
	0.00	0.25	0.50	0.75
records per directory entry	0.431	0.433	0.433	0.428
key comparisons for successful search	1.0	1.0	1.0	1.0
key comparisons for unsuccessful search	0.547	0.587	0.626	0.590
index accesses for successful search	2.445	2.627	2.627	2.648
index accesses for unsuccessful search	1.382	1.489	1.612	1.536

Table 1: Performance of FSMH

We will compare the performance of FSMH with other existing hashing techniques. For the comparison, we will assume that hash addresses are unique. FSMH performs better than extendible hashing. The expected directory size of FSMH grows linearly with the number of records while that of extendible hashing grows in order higher than linear.

Optimal search can be achieved by M-ary trie when the bucket size is 1. The expected directory size of M-ary trie with bucket size 1 is approximately $\frac{m*M}{\ln M}$, where m is the number of hashed records. The maximum number of records per directory entry is 0.34 and this is achieved for $M = 2$. The expected number of index accesses to locate a record is in the order of $\log m$. Knuth has analyzed M-ary tries [4, Section 6.3]. From Table 1 we see that FSMH performs significantly better than M-ary trie.

Dynamic hashing with bucket size 1 achieves optimal search. The expected directory size of dynamic hashing with bucket size 1 is $\frac{m*2}{\ln 2} - N$, where N is the size of level 0 index [5]. The expected number of index accesses to locate a record is in the order of $\log m$. Larson has analyzed dynamic hashing [5]. Since N is fixed, the performance of dynamic hashing is similar to that of M-ary trie with $M = 2$. FSMH outperforms both of them.

4 CONTROLLED SEARCH MULTI-DIRECTORY HASHING

In *controlled search multi-directory hashing* (CSMH) the average number of key comparisons for a successful search is bounded by a *control parameter*. This control parameter is determined by the user who is offered a trade-off between access time and memory space. The directory structure of CSMH is the same with that of FSMH. The only difference is that a directory entry in CSMH may point to a chain of records with different hash addresses. Thus, one key comparison is not guaranteed in this model. After a record has been inserted and linked into the appropriate chain, the average number of key comparisons for a successful search is examined. If this is larger than the control parameter, the subdirectory where the record was inserted expands by splitting each entry into two buddy entries in the same way the subdirectories of FSMH expands. Correspondingly, each record chain splits into two buddy chains, pointed to by two buddy entries of the expanded subdirectory. When record searches are not always successful, the average number of comparisons for successful search is given by the expression

$$\frac{1}{m}\sum_{A}\frac{\ell(A)(\ell(A)+1)}{2}$$

where m is the number of hashed records and $\ell(A)$ is the length of the chain A.

We say that searches have the *inclusion property* when all record searches are successful. In this case, the average number of key comparisons for a search is given by

$$\frac{1}{m}\sum_{A}\frac{(\ell(A)-1)(\ell(A)+2)}{2}$$

This is because, when a chain of length $\ell(A)$ must be traversed, the search requires at most $\ell(A) - 1$ comparisons.

4.1 PERFORMANCE OF CONTROLLED SEARCH MULTI-DIRECTORY HASHING

To evaluate the performance of CSMH we will assume that successful searches form the main portion of the operations of the application. The performance is obtained by simulation. Since we want to achieve near optimal number of key comparisons for a successful search, we are interested in small values of the control parameter. When searches have the inclusion property the control parameter is set close to 0. This is because for a chain with only one record no key comparison is needed to locate the record, given that the search is successful. When the searches do not have the inclusion property, the control parameter is set close to 1.

Table 2 presents the performance of CSMH hashing when searches have the inclusion property. The performance of the method when searches do not have the inclusion property is given in Table 3. Only small values of the control parameter have been used because our goal is to achieve fast search with reasonable storage penalty. The expected values in the table were computed by averaging the results over 150 different runs. Each set was generated by changing the seed of the random number generator. The random number generator produces integers uniformly distributed over the interval $[0, 2^{31}-1]$. Since performance is shown to be a periodic function of $\log_2 m$ with period 1, expected values of only one period are given. The standard deviation of the directory size is less than 3.5% of the expected directory size.

	control parameter				
	0.05	0.25	0.45	0.65	0.85
records per directory entry	0.450	0.540	0.658	0.797	0.962
key comparisons for a search	0.049	0.248	0.448	0.649	0.847
index accesses for a search	2.607	2.286	2.113	1.983	1.763

Table 2: Performance of CSMH when searches have the inclusion property

	control parameter				
	1.02	1.12	1.22	1.32	1.42
records per directory entry	0.446	0.534	0.651	0.788	0.934
key comparisons successful search	1.019	1.119	1.219	1.319	1.419
key comparisons unsuccessful search	0.622	0.747	0.928	1.114	1.258
index accesses successful search	2.617	2.295	2.136	2.003	1.807
index accesses unsuccessful search	1.522	1.436	1.436	1.428	1.351

Table 3: Performance of CSMH when searches do not have the inclusion property

CSMH will be compared with a variation of linear hashing [13,8], called controlled search linear hashing. In controlled search linear hashing, the bucket size is 1 and the chains of buckets of linear hashing have been replaced by chains of records because a main memory database is considered here. Directory entries split as in linear hashing when the average number of key comparisons for a successful search exceeds the value of the control parameter. We will use the following proposition to compare the hashing schemes.

Proposition 3.

In controlled search linear hashing, the expected number of records per directory entry is $\frac{8}{3}(\ln 2)(d-1)$. The expected number of key comparisons for an unsuccessful search is $2(d-1)$, where d is the control parameter.
Proof: It is given in the Appendix.

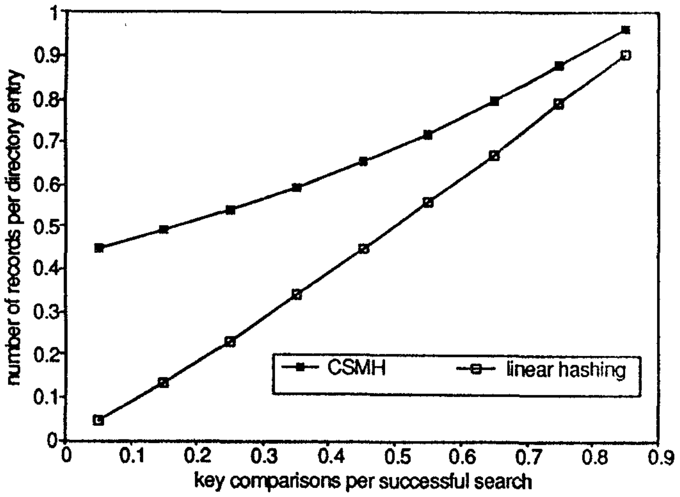

Figure 7: Performance of CSMH and controlled search linear hashing when searches have the inclusion property

Proposition 3 gives the performance of controlled search linear hashing when searches do not have the inclusion property. The performance of controlled search linear hashing when searches have the inclusion property is obtained by simulation. The experiments performed are similar to that of CSMH. Figure 7 compares the the performance CSMH with that of controlled search linear hashing when searches have the inclusion property. From the figure we see that CSMH obtains smaller expected directory size that of controlled search linear hashing for small values of the control parameter.

The following proposition proves that CSMH obtains smaller expected directory size than any single-directory hashing model.

Proposition 4.
The expected directory size of any single-directory hashing model is larger than $\frac{m}{2(d-1)}$, where d is the expected number of key comparisons for a successful search when searches do not have the inclusion property and m is the number of inserted records.
Proof:
Assume a single-directory hashing technique where collisions are handled by separate chaining. If the expected number of key comparisons for a successful search is fixed, the directory size of the method is minimized

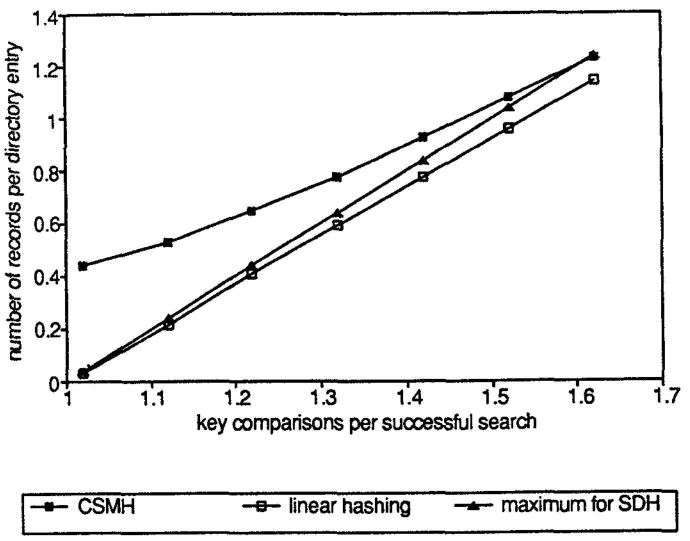

Figure 8: Performance of CSMH and single-directory hashing models when searches do not have the inclusion property

when the records are uniformly distributed over the directory entries. Then, according to Knuth [4, Section 6, Exer. 34], the expected number of key comparisons for a successful search equals $1 + \frac{m}{2D}$, where m is the number of inserted records and D is the directory size. This proves Proposition 4. □

Figure 8 shows the expected number of records per directory entry of CSMH and controlled search linear hashing when searches do not have the inclusion property. It also shows the maximum expected number of records per directory entry for any single-directory hashing model (SDH) as computed from Proposition 4. It is seen from the figure that CSMH outperforms controlled search linear hashing and any single-directory hashing model in the high performance range of the control parameter.

5 CONCLUSIONS

In this paper, we investigated high performance hash based access strategies for main memory database systems. First, we provided justifications for multi-directory hashing schemes with adaptable subdirectory size. Then, we presented two multi-directory hashing schemes that yield fast search in main memory databases. FSMH and CSMH use a tree-structured hash directory whose nodes adapt their size dynamically to a changing number of records. FSMH requires at most one key comparison to locate a record when hash addresses are unique. When hash addresses are uniformly distributed in a bounded interval, the method requires a near optimal number of key comparisons to locate a record. In CSMH, the expected number of key comparisons for a successful search is bounded by a control parameter. This control parameter is determined by the user who is offered a trade-off between access time and memory space.

When extendible hashing, dynamic hashing and M-tries are modified to yield optimal search in main memory, FSMH outperforms them. CSMH yields better directory utilization over controlled search linear hashing and over any single-directory hashing model in the high performance range of the control parameter. Both FSMH and CSMH achieve linearly increasing directory size with the number of records. The expected number of index accesses to locate a record is in $O(1)$. The authors have proved several performance results theoretically. Those can serve as the basis for proving other properties derived experimentally in the paper.

APPENDIX

Here, we give the proofs of Propositions 2 and 3.

Proof of Proposition 2

Since

$$\begin{aligned} P_k^m &= (1-\frac{1}{2^k})\cdots(1-\frac{m-1}{2^k}) \Rightarrow \\ \ln P_k^m &= \ln(1-\frac{1}{2^k})+\cdots+\ln(1-\frac{m-1}{2^k}) \end{aligned}$$

using the asymptotic relation: $\ln(1-x) = -\sum_{i=1}^{n}\frac{x^i}{i} + O(x^{n+1})$, for $|x| < 1$, we get

$$\begin{aligned} \ln P_k^m &= -\frac{m^2}{2*2^k} + O(\frac{m}{2^k}) \text{ or} \\ P_k^m &= e^{-\frac{m^2}{2*2^k}+O(\frac{m}{2^k})}, \text{ for } k > \log_2 m \end{aligned}$$

Now, using $e^{O(x)} = 1+O(x)$ and $e^x = O(1)$, for $|x| \le r$, where r is fixed, we get

$$P_k^m = e^{-\frac{m^2}{2*2^k}} + O(\frac{m}{2^k}), \text{ for } k > \log_2 m$$

Now, we derive the following from the above equation

$$\begin{aligned} P_k^m &= e^{-\frac{m^2}{2*2^k}} + O(\frac{m}{2^k}) \\ &= e^{-\frac{((\sqrt{2})^\ell m)^2}{2*2^{k+\ell}}} + O(\frac{m}{2^k}) \\ &= P_{k+\ell}^{(\sqrt{2})^\ell m} + O(\frac{(\sqrt{2})^\ell m}{2^{k+\ell}}) + O(\frac{m}{2^k}) \\ &= P_{k+\ell}^{(\sqrt{2})^\ell m} + O(\frac{m}{2^k}), \text{ for } k > \log_2 m.\Box \end{aligned}$$

Proof of Proposition 3

The proof will follow the steps of a similar analysis for linear hashing given in [8]. However, the control parameter is different here. Consider a static hash table where collisions are resolved by separate chaining [4, Section 6.4]. Assume that α is the ratio of the number of hashed records to the table size. Knuth [4, Section 6, Exer. 34] has shown that the average number of key comparisons for a successful search, denoted as $s(\alpha)$, is approximated by $1 + \frac{\alpha}{2}$. The average number of key comparisons for an unsuccessful search, denoted as $u(\alpha)$, equals the average chain length of the table, that is, $u(\alpha) = \alpha$.

We define as basic entries of a linear hashing directory, the directory entries before the last expansion. Consider a linear hashing directory where a proportion x of the basic directory entries have split. In this case, the expected number of key comparisons for a successful search, denoted as $S(\alpha, x)$, equals [8]

$$S(\alpha, x) = x * s(\frac{\alpha * (1 + x)}{2}) + (1 - x) * s(\alpha * (1 + x))$$

where α is the ratio of the number of records to the directory size. This is because, the probability that the search will hit an expanded entry is x. The average chain length of an expanded entry is $\frac{\alpha(1+x)}{2}$ and that of a non-expanded entry is $\alpha(1 + x)$.

Because in controlled search linear hashing the average number of key comparisons for a successful search is controlled by a control parameter d, the value of x should be such as $S(\alpha, x) = d$. This gives the following relation between x and α,

$$\alpha \doteq \alpha(x) = \frac{4(1 - d)}{x^2 - x - 2} \qquad (1)$$

The expected number of records per directory entry, $\overline{\alpha}$, is computed as

$$\overline{\alpha} = \int_0^1 \alpha(x)dx = \frac{8}{3}(\ln 2)(d - 1) \qquad (2)$$

Similarly, we have

$$U(x) = x * u(\frac{\alpha(x) * (1 + x)}{2}) + (1 - x) * u(\alpha(x) * (1 + x))$$

where $U(x)$ is the average number of key comparisons for an unsuccessful search of controlled search linear hashing.

From the above equation and equation (1), we get $U(x) = 2(d - 1)$. So, the average number of key comparisons for an unsuccessful search , denoted at $\overline{U}$, equals

$$\overline{U} = \int_0^1 U(x)dx = 2(d - 1) \qquad (3)$$

Equations (2) and (3) give Proposition 3. □

REFERENCES

1. D.J. DeWitt, et al., "Implementation techniques for main memory database systems", Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 1-8, 1984.
2. R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", ACM Transactions on Database Systems, Vol. 4(3), pp. 315-344, 1979.
3. Ph. Flajolet, "On the performance evaluation of the extendible hashing and trie searching", Acta Informatica, 20, pp. 345-367, 1983.
4. D.E. Knuth, "The Art of Computer Programming", Vol. 3, Addison Wesley, 1973.
5. P.A. Larson, "Dynamic Hashing", BIT, Vol. 18(2), pp. 184-201, 1978.
6. P.A. Larson, "Linear Hashing with Partial Expansions", Proc. of 6th VLDB Conference, pp. 224-232, 1980.
7. P. A. Larson, "Linear Hashing with Overflow-Handling by Linear Probing", ACM Transactions on Database Systems, Vol. 10(1), pp. 75-89, March 1985.
8. P. Larson, "Dynamic Hash Tables", Communications of the ACM, Vol. 31(4), pp. 446-457, April 1988.
9. P.A. Larson, "Linear Hashing with Separators- A Dynamic Hashing Scheme Achieving One-Access Retrieval", ACM Transactions on Database Systems, Vol. 13(3), pp. 366-388, Sept. 1988.
10. P. A. Larson, M.V. Ramakrishna, "External Perfect Hashing", Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 190-200, 1985.
11. T.J. Lehman, M. Carey, "Query Processing in Main Memory DBMS", Proc. of the ACM International Conference on Management of Data, pp. 239-250, May 1986.
12. T.J. Lehman, M. Carey, "A study of index structures for main memory database management systems", Proc. of the 12th International Conference on Very Large Data Bases, pp. 294-303, Sept. 1986.
13. W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", Proc. 6th VLDB Conference, pp. 212-223, 1980.
14. H. Mendelson, "Analysis of extendible hashing", IEEE Transactions of Software Engineering, Vol. 8(6), pp. 611-624, November 1982.
15. M. V. Ramakrishna, P.A. Larson, "File Organization Using Composite Perfect Hashing", ACM Transactions on Database Systems, Vol. 14(2), pp. 231-263, June 1989.
16. K. Salem, H. Garcia-Molina, "System M: A Transaction Processing Testbed for Memory Resident Data", IEEE Transactions on Knowledge and Data Engineering, Vol 2(1), pp. 161-172, March 1990.
17. L.D. Shapiro, "Join processing in database systems with large main memories", ACM Transactions on Database Systems, Vol. 11(3), pp. 239-264, Sept. 1986.
18. K. Whang, R. Krishnamurthy, "Query Optimization in a Memory-Resident Domain Relational Calculus Database System", ACM Transactions on Database Systems, Vol. 15(1), pp. 67-95, March 1990.