FLASH : A Main Memory Storage System

김병철, 정병관, 김문자

Pyung-Chul Kim, Byung Gwan Jung, Moon Ja Kim*

Abstract

In this paper, we introduce a new main memory storage system called FLASH that is designed for real-time applications. The FLASH system is characterized by the memory residency of data and a new fast and dynamic hashing scheme called *extendible chained bucket hashing*.

We compared the performance of the new hashing algorithm with other well-known ones. Also, we carried out an experiment to compare the overall performance of the FLASH system with a commercial one. Both comparison results show that the new hashing scheme and the FLASH system outperforms other competitives.

1. Introduction

Growing attention has been paid to main memory database systems due to its fast response time that is necessary for real-time applications, and the rapidly decreasing cost of RAMs. Although conventional database systems have been widely used due to its scalability to support very large databases and user friendly interface, their dependency on secondary storage to store and access data limits its usage into systems in which the speed of disk I/O is acceptable. The transaction response time in conventional database systems is limited by latency delay in disks, which is usually the order of 10~20 ms. This might be enough for traditional applications in which a response time of a few second is tolerable, but is unacceptable in real-time applications that require response within hundreds of micro seconds.

The challenge to overcome the hurdle of the I/O bound make the sight of researchers move from disk-based database systems to memory-based database systems [5]. In memory-based database systems, the entire database resides permanently in main memo-

*Computer Technology Division, ETRI

ry, thereby eliminating disk I/Os. In order to support the memory residency of the entire database, the size of a database must be smaller than the amount of available memory. Fortunately, as semiconductor memory becomes cheaper and the density increases, it becomes feasible to store relatively large databases in memory.

For last several years, *network information control/management system (NICS)* has been developed in our institute to perform both service control functions and service management functions under the intelligent network environment. Among the services supported by NICS, the free phone services and the credit card calling services are the most practical. To make these services applicable in the real world, it is essential to build a real-time database system that can store and access a large volume of customer information with a time constraint. At least 150 accesses and 40 updates per second should be affordable in this system. Since the requirement is far beyond the performance of disk-based database systems, we have built our own main memory resident storage system *called FLASH.*

The access method is a core part in the design of a database system. Numerous access methods in main memory databases have been investigated in [1, 6, 7, 8]. Hashing scheme is a well-known technique that permits very fast random access to a large file through key-to-address calculation. Several hashing schemes have been proposed in both main memory and disk-based databases [1, 3, 6, 7, 9]. They differ in how to expand the address space and how to deal with overflows incurred by collisions in which more than one key has the same hash value.

In main memory environment, *chained bucket hashing (CBH)* is known to provide the fastest access time to a static file [6]. CBH scheme, however, may not be directly applicable for a dynamic file. This is because it requires global reorganization of the hash table to expand or shrink its address range. Without such reorganization, CBH suffers from low storage utilization or long chain length when number of inserted keys is too small or large, respectively.

In disk-based environment, much of research has focused on hashing scheme that has ability to adapt its address space for files with dynamically changing size. These include extendible hashing (EH) [3], linear hashing (LH) [9] and its extensions. Clearly, these schemes may not be suitable for main memory resident database systems. This is because the access time for main memory is the order of magnitude less than disk devices and there is no block transfer concept in main memory. Lehman has investigated the performance and storage utilization of these schemes [8]. In particular, EH scheme suffers from the large size of directory when leaf node size is relatively small, and LH scheme shows too slow

response time for insertion and search.

Recently, two new hashing schemes were proposed for main memory resident database systems: a linear hashing modified for hash table stored in main memory [7], and controlled search multidirectory hashing (CSMH) [1]. Unlike original LH for disk files, a leaf node in the modified LH contains a single pointer, which is the head of a linked list of records hashing to that node. CSMH uses a tree-structured hash directory whose nodes adapt their size dynamically to a changing number of inserted keys. Both methods achieve linearly increasing expected directory size with the number of inserted keys. The expected number of key comparisons for a successful search is controlled by a parameter supplied by users. One shortcoming of modified LH is that the time to transform a key to address is not constant, which is due to the linear expansion of address space. The characteristics of CSMH has been analyzed in only very high performance margin where average number of key comparisons for a successful search is less than 2. Also, loading time of CSMH has not been analyzed. Both methods assume that hash values of records are unique, which may not be attainable in a practical application.

For the FLASH system, we invented a new hashing scheme called *extendible chained bucket hashing (ECBH)* which is seamless integration of EH and CBH. ECBH replaces each leaf node in EH by a chained bucket hash table and record identifier chains. In ECBH scheme, a hash table with sufficiently large size (e.g., 1024 entries) is expected to cause the same effect of a leaf node with large size in the original EH; the directory size is negligible even in a large file. Unlike a leaf node in original EH, however, the search time within a hash table can be controlled in $O(1)$. That is, ECBH scheme inherits its high performance from CBH and gradual extensibility from EH, respectively.

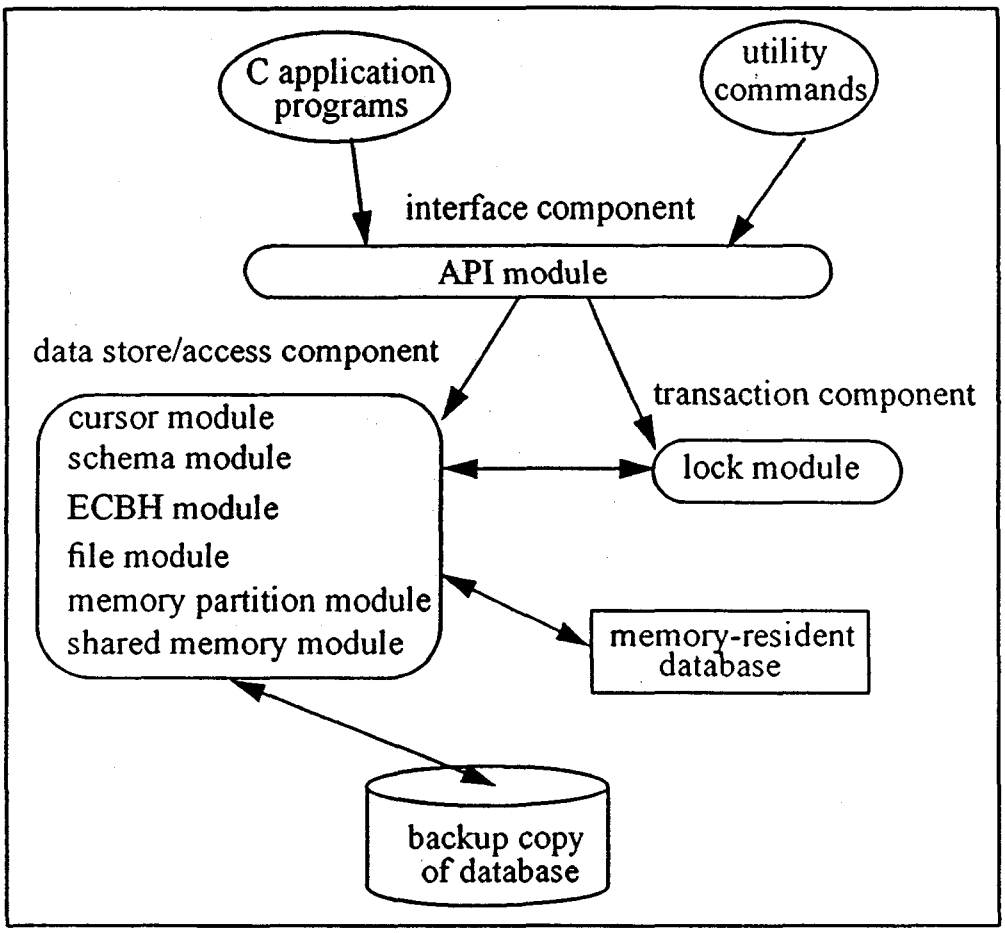
Performance and storage utilization of ECBH is controlled by a parameter given by users. The experiment results show that ECBH outperforms modified LH and CSMH in both loading time and search cost. ECBH and modified LH have similar storage requirement. In particular, the storage overhead of ECBH is smaller than that of CSMH in reasonable performance margin. We, also, show that the FLASH system with the ECBH scheme outperforms a competitive commercial main memory storage system with other hashing scheme.

This paper is organized as follows. The overall structure of the FLASH system and the modularity and functionality of each module are described in Section 2. Section 3 describes the ECBH algorithm in depth. The original EH and CBH schemes are reviewed and the way to integrate them are explained. The performance of ECBH and the FLASH system is presented in Section 4. Fi-

nally, we make a concluding remark including the future work in Section 5.

2. The FLASH System

The FLASH system consists of three components: interface component, data store and access component, and transaction component (see Figure 2.1). The interface component exports a library of functions with which application programs are built. The data store/access component creates and removes user files and indices, and stores records in a file in main memory. The transaction component manages a set of user operations as a unit of concurrency control and recovery. The FLASH system supports only simple lock/unlock primitives at this moment.



Figur. 2.1 The FLASH system components.

2.1 Application Programmer's Interface(API) Module

Controlling and accessing a FLASH database are done using the C−callable API library functions provided by this module. API library routines perform their functions by calling the internal routines of the FLASH system. The utility commands provide an interactive facility for various statistic on database schema, mounting and dismounting databases, etc.

The API module also performs the following functions :

• Configuration file management : The module keeps two UNIX files to store the disk database and the shared memory configuration.

• Session management: A session is a vehicle through which a user's process can manipulate a database. Authorization and access synchronization are based on the session concept. The module is responsible for allocation/free of sessions.

• Authorization management: The FLASH system discriminates the authorization of a user through passwords for the protection of the database. Three levels of authorization are supported; reader, updater, and DBA. The reader has read−only permission while the updater can read and update records. The DBA is allowed to change database schema as well as read/update records. The authorization level of a session is determined when the session is created.

• Mutual exclusion management : To keep the internal information of the FLASH system (i.e., system catalog, free partition list, etc.) consistent, the FLASH allows only one function to be active at any moment. This mutual exclusion is achieved using a shared variable. The API module first locks a shared variable before entering a FLASH internal function, and unlock it when leaving the function.

2.2 Cursor Module

A cursor in the FLASH system can be defined as a navigation vehicle through which only a set of qualified records is to be located and accessed. A user can read, update, and delete the record under the cursor located as desired.

There are two kinds of cursors supported: sequential cursors and index cursors. Sequential cursors allow a user to navigate records in a file sequentially. Index cursors support navigating the records having a particular key value with the help of the associated index. Users can associate a filter with a cursor to restrict the set of records to navigate. A filter is represented as a conjunction of comparison expressions each of which has the form of ⟨*field specification, comparison operator, constant*⟩.

2.3 Schema Module

This module manages two system catalog files: one is FILE_CATALOG_FILE, the other is ECBH_CATALOG_FILE. FILE_CATALOG_FILE stores the information on user-defined files such as file name, length of a record, storage map, lock status, and a pointer to its index information in ECBH_CATALOG_FILE, etc. FILE_CATALOG_FILE is associated with an index on the file name to provide fast lookups through a given file name.

ECBH_CATALOG_FILE keeps information on indices built such as key field description and storage map, etc.

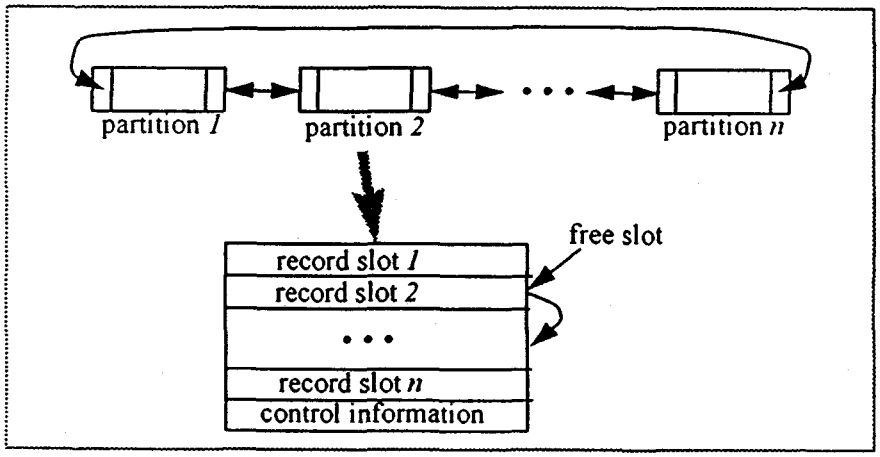
2.4 ECBH Module

The index scheme employed in the FLASH system is ECBH that is an integration of EH and CBH. The ECBH module supports creation/deletion of ECBH indices, and insertion/deletion of keys through the indices. The details of ECBH algorithm are presented in Section 3.

2.5 File Module

The file module manages a data file which is a collection of identical type of records. All records of a file have the same format. More precisely, we allow only fixed size fields such as integer, float, and fixed strings.

A file is implemented as a doubly linked list of fixed size of memory partitions (see Figure 2.2). A partition may not be shared by more than one file. A partition consists of array of record slots and control information. The control information contains the previous and the next partition numbers for the implementation of doubly linked list, and the first free slot number in the partition.



Figuere 2.2 Storage structure of a file.

And a record slot has the format of ⟨*status, data*⟩. Status represents the status of the slot: being used or free. Data is the concatenation of data fields.

The status part of a free slot contains the next free slot number in the same partition. In this way, free slots in a partition forms a stack. Also, the partitions of a file that have free slots form a stack by chaining together. This scheme for free space makes it easy and efficient to find room for a new record at insertion time. That is, allocation of a free slot can be done by just popping the top from the stack of free slots and partitions, and freeing of a slot can be processed simply by pushing it to the stack.

The location of a record is identified through its RID which consists of ⟨*partition number, slot number*⟩. The partition number identifies the partition that the record slot belongs to. The slot number is used to locate the record slot in the partition. In the FLASH system, a RID is represented in 32 bits to take advantage of fast transfer between functions using CPU registers, while allowing reasonable size of a database. A database, a file, or an index can grow up to 4 giga bytes. The number of files and indices is limited by the number of available partitions (2 millions to the maximum).

2.6 Memory Partition Module

The primary role of the memory partition module is to manage the shared memory partitions and provide the upper modules with the interfaces for allocation/free of partitions. The module also implements the functions to manage the backup copy of a database such as formatting disk volumes, mounting and dismounting the database between shared memory partitions and disks.

A disk database in the FLASH system consists of several disk volumes, each of which consists of a sequence of partitions of equal size (16K bytes). Each volume may be either a UNIX regular file or a raw device file. The volume concept is meaningful only when a backup copy is taken. Once the database is mounted into memory, no volume boundary exists, but only partitions are scattered in shared memory. The mapping between the disk volumes and the shared memory partitions is based on the database configuration file.

Figure 2.3 shows the structure of the shared memory partitions. A partition has a four-byte control information at a fixed location to identify its modified status and the next free partition number. The first partition is reserved for the generic information on a database such as owner, create/mount timestamp, the FLASH software version number used for the database creation, and the catalog file locations. Free partitions form a stack which is implemented as a linked list.

For the direct identification of the virtual

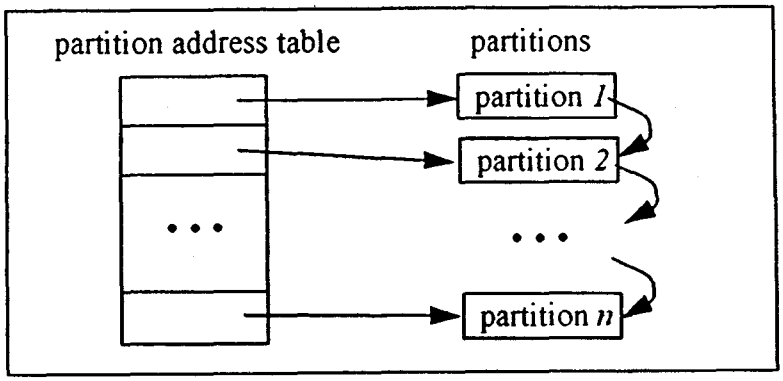


Figure 2.3 Structure of the shared memory partitions.

address of each partition, an array of address table is maintained as described in the figure. This array must be contiguous. Thus, the operating system must be configured so that the maximum size of a shared memory segment is large enough to hold the entire partition address table.

The entire contents of a database must be loaded into shared memory before accessed. Therefore, the shared memory must be configured large enough to store the entire database.

In order to mount a database, the memory partition module first constructs a collection of partitions of fixed size in shared memory with the help of the shared memory module, then reads each disk partition from the disk volumes to the corresponding memory partition.

2.7 Shared Memory Module

The databases mounted into memory must be accessible by multiple application processes. In order to achieve this, the shared memory module make use of the shared memory facility of the UNIX System V IPC package. Each application process first attaches the FLASH shared memory partitions at its address space and then accesses them. Note that we force the operating system not to swap out any partition of the shared memory region to disk.

In the FLASH system a database occupies only one shared memory segment in terms of the operating system. Therefore, the operating system must be configured to support space for a shared memory segment large enough to load an entire database.

The address at which a shared memory segment is attached depends on the application process, which implies that the address at which a database is mounted may be different at each mount time.Therefore, no upper modules including user programs in the FLASH system can store a virtual mem-

ory address in the database assuming that the address is consistent across mounts.

2.8 Lock Module

Concurrent accesses to the same data item by multiple users must be controlled in the way that they are given the illusion that their processes are executed serially. With out such control, the updates by one user may be interfered or even lost by other users.

Numerous algorithms for concurrency control have been proposed. They are categorized into locking, timestamp, and optimistic scheme [2]. The locking method is known simple to implement. It has been observed that a finer locking granule such as an individual record to reduce data contention is not much effective in main memory databases because transactions complete very shortly. The lock granule of a main memory database system is suggested to be as large as possible, in an extreme case, upto the entire database [5].

From the above reasons, the FLASH system implements locking scheme with a very coarse granule, i.e., a file. Only lock/unlock a file with exclusive mode is available at this moment.

3. Extendible Chained Bucket Hashing

3.1 Review of Extendible Hashing and Chained Bucket Hashing

We briefly review original CBH and EH methods. CBH consists of a hashing function, a hash table and linked lists of RIDs (see Figure 3.1) [6]. The hash function transforms a key value into an address value whose range is the cardinality of the hash table. The address is used to locate an

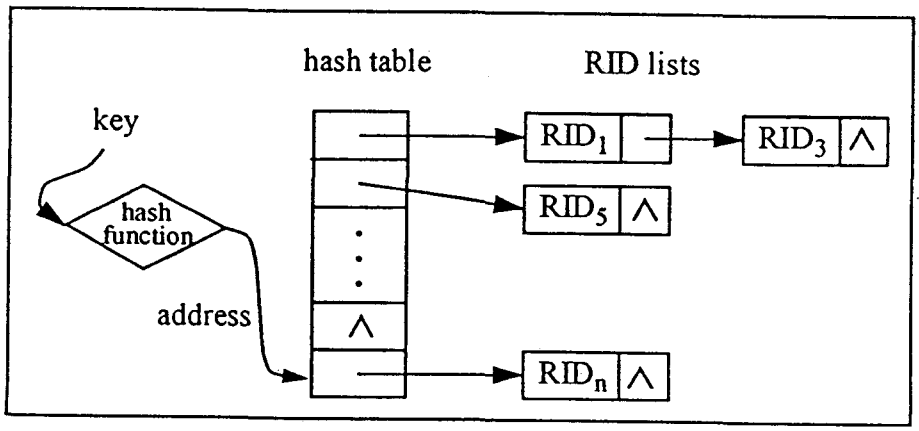


Figure 3.1 Chained bucket hashing scheme.

entry of the hash table. The entry points to a list of RIDs having the same hash address. The time to compute the hash value of a key and to locate an entry of the hash table is constant. When keys are uniformly distributed, the expected chain length of CBH equals , where m is the number of inserted keys and D is the hash table size. When D is fixed, therefore, the search cost linearly increases as m grows. This performance characteristics is not desirable because the primary goal of a hash-based access method is to achieve search time in O(1). It is possible to expand the hash table (e.g., doubling the hash table) whenever the expected chain length exceeds a threshold value. However, the expansion requires a total reorganization of the hash table and re-linking of the RIDs, which may cause an intolerable performance drop in a typical main memory resident database application.

EH employs a hash function, a dynamic directory, and leaf nodes pointed to by entries of the directory (see Figure 3.2) [3]. A key is transformed into a hash value. The global depth least significant bits of the hash value is used to locate a directory entry. The entry points to a leaf node which is fixed size (generally, a disk page). A leaf node with local depth d contains RIDs hash values of which have the same d least significant bits. Therefore, a leaf node may be shared by more than one entry of the directory. In the figure, R(x) represents the RID of a record where x is a sequence of global depth least significant bits of its hash value. When

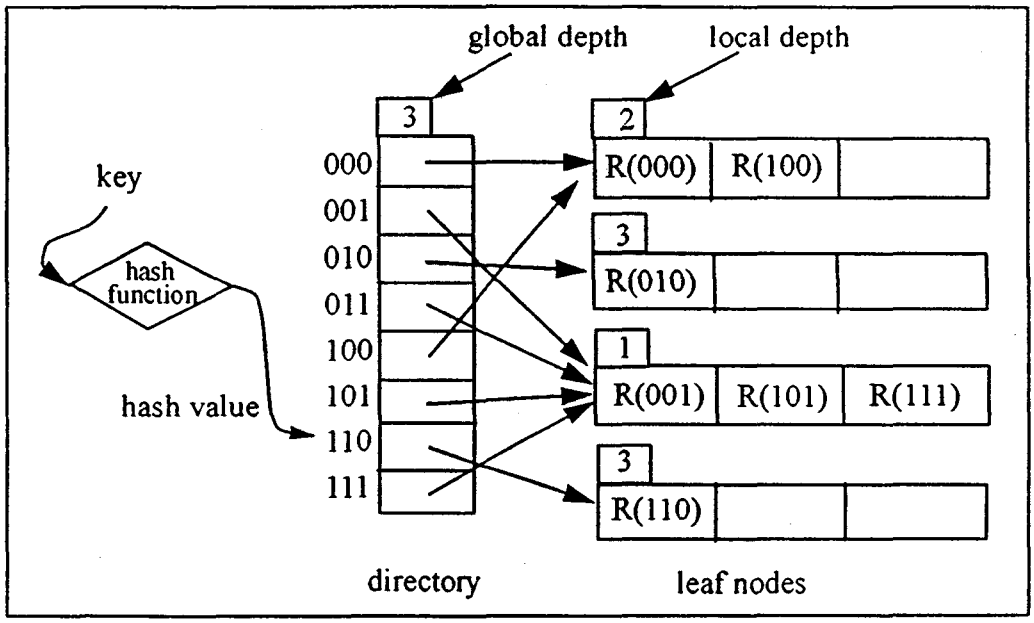


Figure 3.2 Extendible hashing scheme.

a leaf node with local depth d overflows, it splits into two nodes based on (d + 1)-th least significant hash address bit of its RIDs, and increases d. These two nodes correspond to siblings when the directory is represented as a binary radix search tree. In Figure 3.2, the nodes with local depth 3 (i.e., those containing R(010) and R(110), respectively) are siblings. If local depth of a leaf node is greater than the global depth after a split, the directory doubles to make room to point the split leaf nodes.

Let b be the maximum number of RIDs in a leaf node. When b > 1, Flajolet has investigated the expected number D of directory entries of EH with a coarse approximation [4],

$$D=\frac{3.92}{b}\, m^{1+\frac{1}{b}}$$

where m is the inserted records. Therefore, the directory size of EH with sufficient large node size increases almost linearly as m grows. In fact, the expected directory size is neg-ligible compared to the storage for leaf nodes. For example, if b = 1024, then after a mil-lion inserts, the number of directory entries is expected to be 4068.8. For such a large node size, however, the search cost would not be acceptable in many high performance main memory resident databases. As b becomes closer to 1, the number of directory entries grows in more than linear. For b = 1, for example, it has been shown that doubling the number of inserted records increases the number of hash address bits needed to differenti-ate the hash addresses by two: i.e., the expected directory size of EH with leaf node size 1 is in $O(m^2)$ [1].

3.2 Integrating Extendible Hashing and Chained Bucket Hashing

From the above observation, we develop a new hashing scheme called extendible chained bucket hashing (ECBH) by combining EH and CBH seamlessly. ECBH replaces each leaf node in EH by a chained bucket hash table and RID chains. In ECBH method, a hash table with sufficiently large size (e.g., 1024 entries) is expected to cause the same effect of a leaf node with large size in the original EH; that is, the directory size is negligible even in a large file. Unlike a leaf node in the original EH, however, the search time within a hash table can be controlled in O(1). That is, ECBH scheme inherits the high performance of CBH and the gradual extensibility of EH at the same time.

Figure 3.3 shows an example of ECBH structure. ECBH consists of a hash function, a directory, several hash tables and linked lists of RIDs. The directory in ECBH grows and shrinks in the same way of the original EH. Hash tables have the same fixed cardinality. Unlike in the original CBH, each hash table is associated with a local depth.

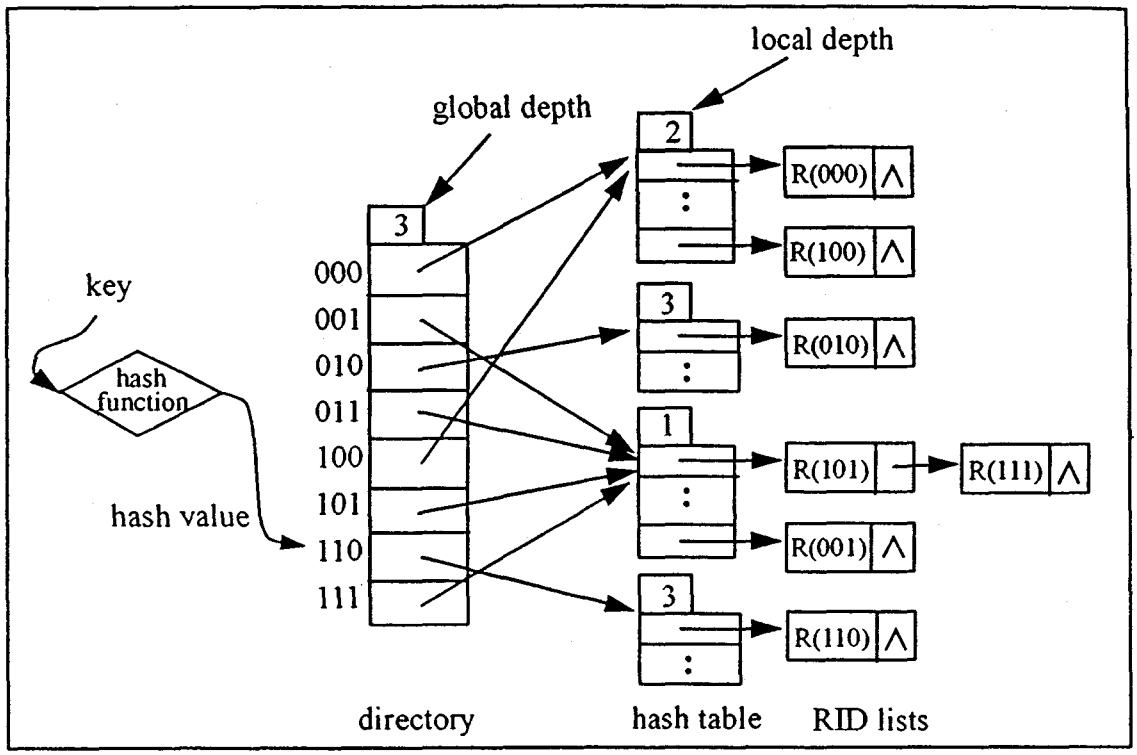


Figure 3.3 Extendible chained bucket hashing structure.

The local depth has the same meaning that of a leaf node in the original EH. A key K is transformed into a hash value H. We view a hash value H as a concatenation of two sequences of bits: ⟨H1:H2⟩. We use H2 to locate an entry of directory (i.e., to locate a hash table) and H1 to locate an entry of the hash table. If the cardinality of a hash table is C, the number of bits reserved for H1 is calculated as . In practice, the cardinality C should be power of two in order to take advantage of bit operator rather than modulo operator. The remaining least significant bits of H form H2 which determines the maximum global depth. For example, if H is represented 32 bits and the cardinality C is 1024, 10 most significant bits of H form H1 and remaining 22 least significant bits represent H2. In this case, the maximum global depth will be 22.

The performance of ECBH is controlled by a parameter l supplied by users. The parameter l is used to control the average chain length of ECBH. Average chain length Lavg is computed as

$$L_{avg}=\frac{1}{m}\sum_{i=1}^{m}L_{RID_i}$$

where represents the number of chains preceding RIDi, and m is the number of insert-

ed RIDs. In Figure 3.3, for example, LR (111) is 2 and Lavg is . After a record is inserted to a hash table with local depth d, Lavg is computed and checked if it exceeds l. If it exceeds, the hash table splits into two tables based on (d + 1)−th least significant hash address bit of its RIDs, and increases d. If the new local depth is greater than the global depth, the directory is doubled to make room to point the split hash tables. If the global depth reaches the maximum value, no more split is processed even when the average chain length exceeds the control parameter.

Figure 3.4 shows the algorithm to split a hash table in more detail. The maximum glo

Algorithm SPLIT
input : P–the hash table to be split

// The followings are constants :
// C : cardinality of a hash table
// l : control parameter for average chain length
// g_{mas} : maximum global depth

1. If local depth of P has reached to g_{max}, then stop.
2. Allocate a new hash table Q.
3. For $1 \leq i \leq C$, do the following.
   3.1 For each RID R in the list pointed by i–th entry of P, do the following.
      3.1.1 Form key value K from R.
      3.1.2 Compute hash value H from K.
      3.1.3 If $(d+1)$–th least significant bit of H is 1, where d is the local depth of P, then move R to i–th entry of Q.
4. Increase local depth of P by one, and set local depth of Q to that of P.
5. If local depth of P is greater than the gloval depth, double the directory(the new area is simply filled with the original directory contents), and increase global depth by one.
6. Set every sibling entry of the directory that points P to Q(in order to make the directory comply with Step 3.1.3).
7. Calculate average chain length L_{avg}, and if it is greater than l, then SPLIT using one of P or Q that has larger average chain length.

Figure 3.4 Algorithm SPLIT for ECBH.

bal depth gmax is calculated as (h - logC), where h is the number of bits of a hash value and C is the cardinality of a hash table. Insertion, search, and deletion are straightforward. In particular, deletion of a record may leave a hash table empty. When one of siblings becomes empty, we simply drop it, and change the directory entries containing it to point to its sibling. The local depth of its sibling is decreased by one. If every pointer in the directory equals to its sibling pointer, we halve the size of the directory and decrease the global depth by one. It is possible to use another parameter to control minimum storage utilization. Two siblings can be merged if their storage usage is less than the control parameter even when both are not empty.

3.3 Refinements for Practical Use

As in the case of other hash-based access methods, ECBH may also be skewed if hash val-ues are not uniformly distributed. Even though the hash function itself provides very good randomization, the degeneration can happen if many duplicated keys are inserted. In the-ory, none of modified LH, CSMH, and ECBH can store two RIDs keys of which are the same, while keeping the average chain length at 1.0. The directory will grow infinitely. However, there are many database applications that require very fast key-associative access to a large file that contains many duplicate keys. In order to broaden the applicability of ECBH, we present the following refinements to ECBH scheme.

First, we can limit the infinite growth of the directory by using a reasonable maximum global depth. Recall that the maximum global depth gmax is computed as $(h-\log C)$, where h is the number of bits of a hash value and C is the cardinality of a hash table. This number may be too large depending on application. For this case, users may introduce a different maximum global depth $k \leq g_{max}$ for a particular ECBH. Then, the number of directory entries of the ECBH is limited by 2^k.

Second, we can introduce another parameter u to control storage utilization of a hash table. Storage utilization of a hash table is defined as the ratio of the number of non-empty entries to total number of entries. A hash table is not split if its storage utilization is less than u even though the average chain length has exceeded the control parameter l. When the storage utilization of a hash table becomes less than u after a deletion, we test if merging it with its sibling still results in lower storage utilization than u. If the test is true, we merge them and decrease the local depth by one. The merging does not require any computation of hash values. Instead, an RID list linked from i-th entry of one hash table is simply appended to i-th entry of its sibling.

Third, splitting a hash table may introduce an empty hash table; that is, all RIDs may be moved to one hash table. We can implement an empty hash table without actually occupying the amount of storage for a non-empty hash table. It is sufficient to reserve one variable for local depth to represent an empty hash table.

4. Performance Experiment

4.1 ECBH Experiment

4.1.1 Experimental Environment

We implemented three hashing schemes, LH, CSMH, and ECBH for the purpose of per-formance comparison. Modified LH requires contiguous memory for the directory while it expands. We implemented a dynamic array which is suggested by [7]. The size of a seg-ment for the dynamic array is set to 1024 in order to take advantage of bit operators when locating a directory entry. Directory size shown in results includes the storage overhead for the dynamic array. A directory entry takes 8 bytes: a four-byte pointer to its RID chain and a four-byte integer to remember the number of RIDs in the chain.

A directory entry in CSMH may become a head of either an RID list or a subdirectory. We implement a directory entry in 8 bytes with the help of union facility of C programming language. The first byte of each entry is used to identify the type of the entry. If the type is the head of an RID list, then the remaining area of the entry contains the number of RIDs and start address of the list. If the type is the head of a subdirectory, the remaining area describes its depth, the number of its subdirectories, and start address of the array of its entries.

For ECBH test, we set the cardinality of a hash table to 1024. The size of a directory entry is 4 byte: address of its hash table. The size of a hash table entry is 8 byte: a four-byte pointer to the RID chain and a four-byte integer to store the number of RIDs in the chain. Each hash table is associated with 12 byte control information: local depth, chain length, and number of stored RIDs.

All test programs are implemented in C programming language. We use 32 bits to represent a hash value. Hash function is implemented based on permuted table [10]. The length of a key is 8 bytes. Keys are populated using library function random(3) which employs a non-linear additive feedback random number generator. Each key set contains 50,000 items which are unique. In fact, their hash values are also unique. Library function malloc(3) is used to allocate or expand memory area. Library function memcmp(3) is used to compare two key values. A record identifier takes 4 bytes. The

machine platform is Sun4c with 24 megabytes physical memory on which SunOS 4.1.1 is running. We locked test program code and data segment in core to keep them from swapping out to disk. All test programs were run in single-user mode. All measurement is the average of 10 runs each of which uses a set of keys generated with a different seed.

4.1.2 Experimental Results

In all strategies, storage overhead grows linearly as the number of inserted keys increases (see Figure 4.1). The storage space requirement for EBCH and modified LH is almost the same. In a very high performance margin (i.e., when the average chain length is close to 1.0), the amount of storage space for modified LH and ECBH grows up to about 4 megabytes. This implies that both schemes basically follow the storage utilization characteristics of chained bucket hashing. As pointed out in [1], CSMH reveals the best storage utilization when average chain length is less than 2.0. However, as the length grows, CSMH takes more

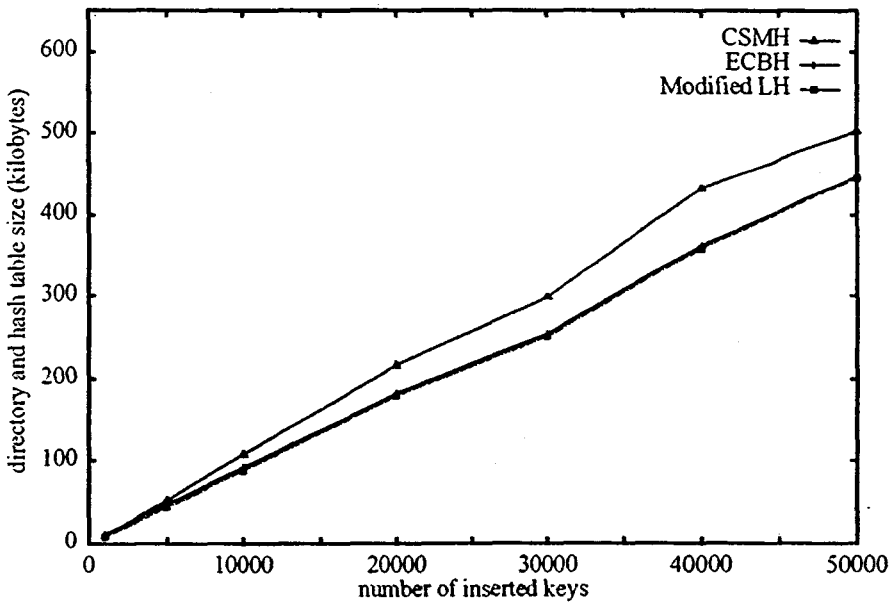


Figure 4.1 Storage cost vs. number of inserted keys.

storage than others (see Figure 4.2 and Figure 4.3). Note that storage cost in the figures does not include memory space to store RID chains because this space is the same regardless of hashing strategies. The results in Figure 4.3 verifies that the expected number of records per hash table entry obtained by analysis complies with the experiment result.

Figure 4.4 shows the time to load 50,000

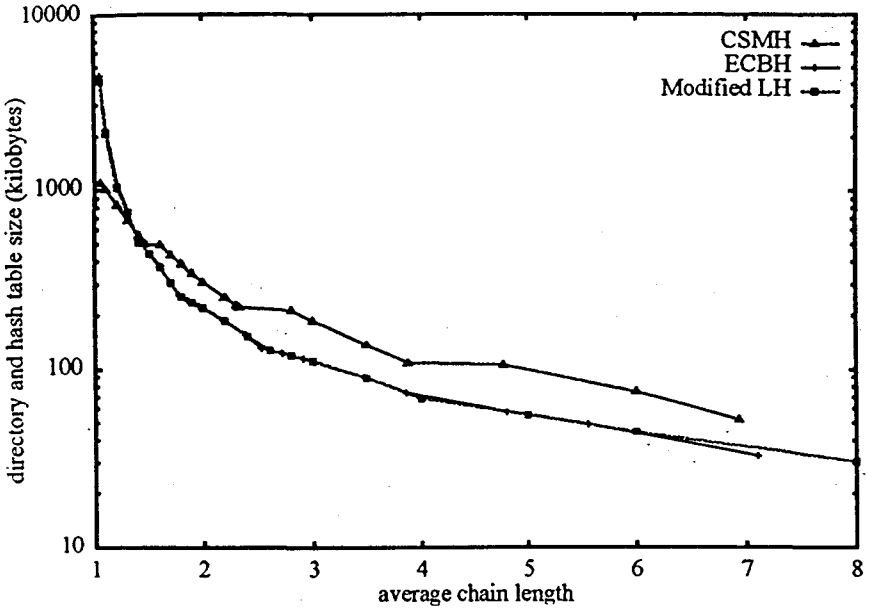


Figure 4.2 Storage cost vs. average chain length.

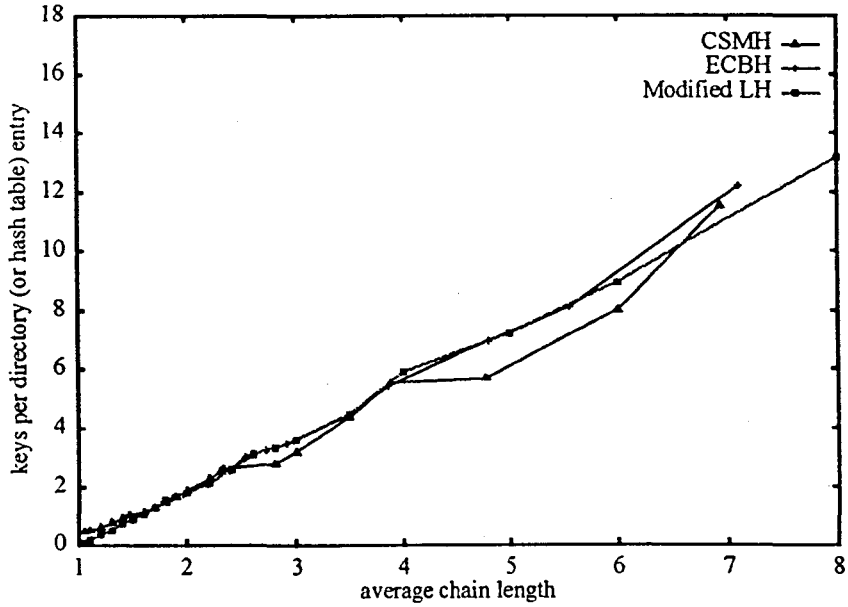


Figure 4.3 Keys per directory entry vs. average chain length.

keys with different average chain length. Modified LH and ECBH have almost the same shape of graphs. We believe that performance difference comes from the cost to locate a directory entry and the number of function calls to split. Modified LH relies on

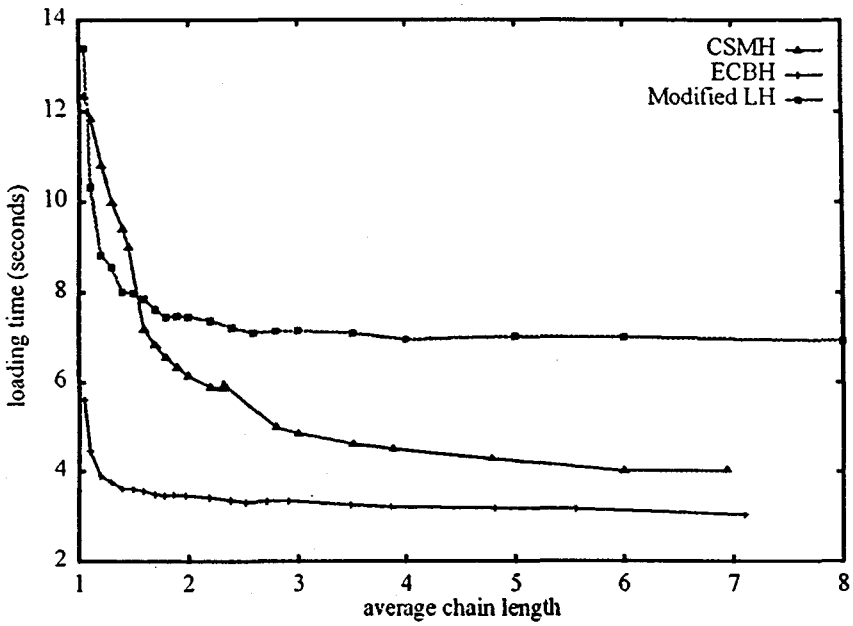


Figure 4.4 Loading time vs. average chain length.

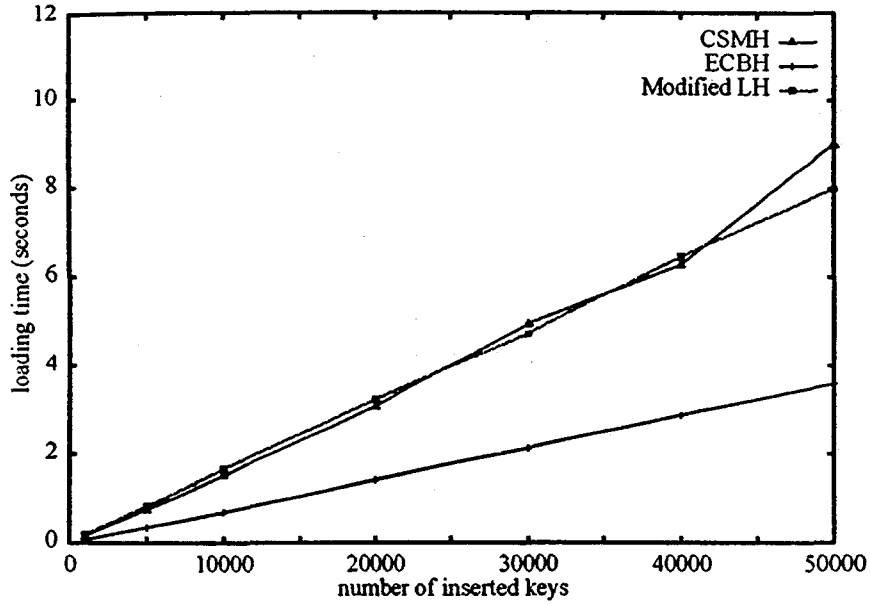


Figure 4.5 Loading time vs. number of inserted keys when the average chain length is fixed to 1.5

modulo operators to locate a directory entry from a hash value while ECBH makes use of bit operators. Modified LH splits one RID chain at a time and calls the split function at every split, while ECBH splits one hash table at a time. CSMH suffers from too fre quent reorganization of subdirectories. Still less, it is not easy to reuse or preallocate

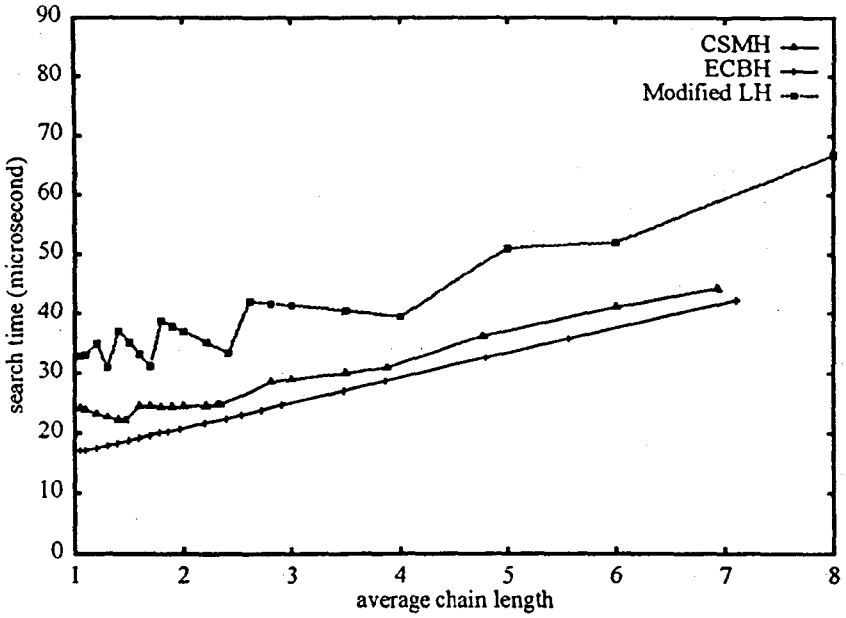


Figure 4.6 Search time vs. average chain length.

memory areas to reduce memory management cost because the size varies over subdirectories. As average chain length grows, the loading time becomes closer to that of ECBH. Figure 4.5 presents loading time when average chain length is fixed to 1.5. All strategies show linearly increasing response time as number of inserted keys grows.

Figure 4.6 represents average time to search a record in each hashing scheme. The average time is measured as the total time to search all records divided by number of records. ECBH outperforms both modified LH and CSMH, and its performance is linear. In modified LH, locating a directory entry requires extra computation if a hash value falls into expanded portion of directory entries. The portion of expanded entries varies over average chain length. As the expanded portion increases, the probability that a hash value falls into that portion increases. That is, the time to locate a directory entry increases. This explains that the search time of modified LH has a periodic function form. In the case of CSMH, the search time decreases slightly even when average chain length increases from 1.05 to 1.5. This is because increasing average chain length makes the directories more balanced, thereby reducing the number of directory accesses. In overall, modified LH is turned out slower than both ECBH and CSMH because of the cost of modulo operation to locate a directory entry. CSMH is slower than ECBH because of the overhead to traverse its tree-structured subdirectories.

4.2 FLASH Experiment

The test is performed in HP 9000 Series 800 (PA-RISC CPU, HP-UX version 8.0). The test database contains only one file consisting of 50,000 records. The length of a record is 24 bytes. An ECBH is built on a key field with 8 byte characters. We collected the response times of various data manipulation operations in a single user mode. Each response time is measured in seconds as the average of at least 10 runs of the same operation.

Table 4.1 summarizes the measurement items and the results of the performance comparison of the FLASH system and a well-known commercial product (we call this System-A from now on).

Table 4.1 Performance of the FLASH system and System-A.

No.	Description	System-A	FLASH 1.0.
P01	Load the entire database to memory	5473.948	6596.404
P02	Flush all updates of the file to disk	3507.337	2656.866
P03	Search a record without index	1167.863	973.086
P04	Search a record with index	0.077	0.068
P05	Insert a record with no-index/no-sync	0.042	0.025
P06	Delete a record with no-index/no-sync	0.076	0.041
P07	Update a record with no-index/no-sync	0.101	0.048
P08-F	First attempt to P08	256.148	N/A
P08	Insert a record with no-index/sync	0.280	60.047
P09-F	First attempt to P09	107.827	N/A
P09	Delete a record with no-index/sync	0.308	45.348
P10-F	First attempt to P10	0.757	N/A
P10	Update a record with no-index/sync	0.330	29.949
P11	Insert a record with index/no-sync	0.074	0.130
P12	Delete a record with index/no-sync	0.102	0.089
P13	Update a record with index/no-sync	0.195	0.194
P14-F	First attempt to P14	1703.827	N/A
P14	Insert a record with index/sync	0.916	120.653
P15-F	First attempt to P15	182.626	N/A
P15	Delete a record with index/sync	0.329	132.213
P16-F	First attempt to P16	657.570	N/A
P16	Update a record with index/sync	1.427	263.843
S1	Calculated size of the data file	1.200Mb	1.200Mb
S2	Size of the stored data file	N/A	1.409Mb
S3	Size of the index	N/A	0.885Mb
S4	Size of the entire database	2.483Mb	2.392Mb
S5	Size of the catalogs	6.376Kb	N/A

Before analyzing the experimental results, it is worthy to note some features specific to System-A. Firstly, the performance of the operations involving disk I/Os depends a lot on the buffer management of operating system since System-A is designed not to utilize the Unix raw device. For instance, the successive execution of the P01 operation may load a database not from the disk but from the operating system buffer once the previous operation leaves data in the buffer.

Secondly, the first execution of each update operation with disk synchronization (P08, P09 P10, P14, P15, P16) takes too much time compared to succeeding executions for some reason that we can not identify from the manual. We, therefore, exclude these cases in the performance comparison by separating them into P08-F, P09-F, P10-F, P14-F, P15-F, P16-F.

Lastly, the update operations with disk synchronization (P08, P09 P10, P14, P15, P16) are supposed to flush the updates out to the disk immediately. In a simple test, we recognized that one disk page write operation consumes more than 7 ms. System-A's results for the operations, however, show about 1 ms. We believe that the updates are synchronized only in the operating system buffer, not in disk, which is incorrect implementation. We ignore these operations in the performance comparison as well.

It is shown that the FLASH system outperforms System-A in all cases except for the insertion of a record. The search operations (P03, P04) of FLASH are not behind System-A in spite of the overhead of its cursor management for user customization. The updates without either an index or disk synchronization (P05, P06, P07) of the FLASH system are almost twice faster than System-A. However, the updates with an index (P11, P12, P13) of the FLASH system becomes about the same as or even slower in insertion case (P11) than System-A. This phenomenon is predictable from the hashing scheme; FLASH employs dynamic hash table while System-A does static one. Regarding delete and update operations with an index (P12, P13), it is hardly understandable until the System-A's implementation of the hash table is investigated in depth that FLASH's dynamic hashing scheme is faster than System-A's static hashing scheme. Unlike in System-A, P11 takes longer than P12 in FLASH. The reason is that an insertion may cause a bucket of an ECBH index to split, resulting in rehashing all keys in the bucket. In contrast, merging buckets caused by deletions does not need the rehashing process.

5. Conclusion and Future Work

The free phone service and credit card

calling service in NICS require real-time processing of a large volume of customers information. The database system for the services should support significantly fast response time that is far beyond the capability of the traditional disk-based database systems. To meet the requirements, we have designed and implemented a main memory storage system on top of UNIX.

We proposed a new access method for main memory database systems named extendible chained bucket hashing (ECBH) which is a complementary integration of original chained bucket hashing and extendible hashing. ECBH inherits the high performance and the gradual extensibility from chained bucket hashing and extendible hashing, respectively. The performance and storage utilization of ECBH is controlled by a parameter given by users. We also described how to refine ECBH structure to deal with duplicate keys and skewed key insertions.

We carried out an experiment to compare ECBH with other strategies proposed for main memory databases: linear hashing modified for main memory and controlled search multidirectory hashing. The experimental results show that ECBH outperforms the proposals in both loading time and search cost. ECBH and modified linear hashing require similar storage space. In particular, the directory and hash table size of ECBH is smaller than that of controlled search multidirectory hashing in a reasonable performance margin. We also showed the outstanding performance of the FLASH system in both speed and space costs by comparing it with a commercial product.

The current version of FLASH (FLASH 1.0) is a multi-user main memory storage system employing ECBH as the primary access method. FLASH 1.0 supports dynamic file management, key-associative exact match retrieval, primitive locking, taking anapshot, primitive form of authorization, etc. However, the functions such as recovery based on transaction concept, key-associated range match retrieval, and deadlock detection are not implemented yet. We leave the features for our future extension.

References

A. Analyti and S. Pramanik, "Fast Search in Main Memory Databases," Proc. of *ACM SIGMOD Conf. on Management of Data*, 1992, pp. 215-224.

P. Bernstein, P. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," *ACM Trans. on Database Systems*, Vol. 4, No. 3, 1979, pp. 315-344.

P. Flajolet, "On the Performance Evaluation of Extendible Hashing and Tree Searching," *Acta Informatica*, Vol. 20, 1983, pp. 345-369.

H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 4, No. 6, 1992, pp. 509-516.

D. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, 1973.

P. Larson, "Dynamic Hash Tables," *Comm. of ACM*, Vol. 31, No. 4, 1988, pp. 446-457.

T. Lehman and M. Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc. 12th Int. Conf. on Very Large Databases*, 1986, pp. 294-303.

W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," *Proc. 6th Int. Conf. on Very Large Databases*, 1980, pp. 212-223.

P. Pearson, "Fast Hashing of Variable Length Text Strings," *Comm. of ACM*, Vol. 33, No. 6, 1990, pp. 677-680.