Yugoslav Journal of Operations Research
3 (1993), Number 2, 171–188

Large File Operations Support Using Order Preserving Perfect Hashing Functions

Dušan STARČEVIĆ

Faculty of Organizational Sciences, University of Belgrade
Jove Ilića 154, 11000 Belgrade, Yugoslavia

Emil JOVANOV

Mihajlo Pupin Institute
P.O. Box 15, 11000 Belgrade, Yugoslavia

Abstract:[1] Most computer applications require efficient management of data and fast execution of basic file operations over large data volumes. Specifically, in real–time environment applications are faced with severe constraints for total execution time of basic operations. This paper introduces method for physical organization of large database files, based on order preserving hashing scheme. Hashing scheme combines two functions: an order preserving and an ordinary hashing function. An original set of algorithms take advantage of implemented physical organization to achieve efficient basic file operations. Proposed method guarantees retrieval of any record in a single disk access, and minimum number of disk accesses for range search and key sequential operations for large dynamic files.

Keywords: Physical file organization, hashing, B–tree algorithms, sorting, data management systems.

1. INTRODUCTION

Extending possibilities of present computer systems made possible new application areas like CAD systems, multimedia, real time process control and artificial intelligence. Basic characteristics of these applications is work with large data files and extensive data processing. For real time applications, process control and multimedia,

---

[1]This work was suppoorted in part by the National Science Foundation of Serbia under Grant No. 1002B.

every object from the very large set of objects must be retrieved within a certain time interval [11]. To fulfill this requirement two approaches could be followed: first, record position is calculated through supporting access data structure which is preferably in main memory, and second, record position on a disk is a function of primary key value only.

It is well known that B–trees are the most accepted access method [1, 17]. Moreover, they are most often used in contemporary multiuser database systems as concurrent search structures [12]. These algorithms provide fast access to a record with a particular key value, and also efficient retrieval of a set of records within a given range of key values. It is possible because of order preserving access data structure that is embedded in B–tree algorithms. Unfortunately, associated data structure is too large to be placed in main memory even in a case of moderate size data files. Therefore, every retrieval can require more than one disk access which is unacceptable for most multimedia and real–time applications [23, 24].

One of the most frequently used data access techniques suitable for guaranteed time retrieval is hashing. Although hashing resolves problem of single key value retrieval, or reports that such record does not exist, it is inappropriate for set retrieval operations. In order to guarantee O(1) access time specific hashing techniques should be applied [16, 17]. There is a number of algorithms that support large static file organization [6]. On the other hand, applications that use very large files with time dependent number of records require dynamic hashing schemes, as a combination of hashing techniques with trie structure [3]. Dynamic hashing functions could be divided according to the use of overflow area. If a hashing function does not cause overflow records for a given set of keys, it is called perfect. According to the use of main memory, hashing functions could be further divided to directory based and non–directory based.

One of the earliest perfect hashing schemes that uses index structure stored in main memory was proposed by Larson [18]. Also, extendible hashing scheme based on collapsed trie stored in main memory guarantees O(1) access time [5]. Fagin proposed use of first k binary digits of the hashed key to find hash table entry. Each index entry contains the address of a data bucket. In a case of data bucket overflow, the index could be doubled or a new bucket is allocated. Deficiency of this algorithm is poor space utilization in the index. In order to limit directory size Lomet introduced multipage nodes [20, 21]. Data node consist of variable–size bucket, and the size of the bucket is written in index entry. Therefore, the first k bits of a hashed key point to the index table entry, and the following w bits point to the selected page in a bucket.

Litwin and Lomet introduced concept of bounded disorder file organization [19]. Bounded disorder files are compromise between B–tree and hash table techniques. Internal memory based structure is derivative of B^+–tree, but parent–of–leaf level contains addresses of the contiguous set of the fixed number of buckets. Each bucket in a data node is the same size (one or more pages). When single data bucket overflows, all buckets in a node must increase number of pages. Gonnet and Larson studied a problem of external hashing with limited internal memory, based on hash signatures [8]. In addition to this concept Ramakrishna and Larson proposed and

analysed a composite perfect hashing scheme to locate record within the bucket [26]. They make use of trial–and–error method to find perfect hashing function from set universal class of hashing functions. Cesarini and Soda use the signature technique and the hash function based on generalized spiral storage to achieve O(1) access while maintaining a high load factor [2].

Correct choice of hashing function may significantly influence the performance of the system. It is shown that for open addressing schemes uniform hashing is optimal [28]. Simple uniform hashing can be achieved by circulary shift and XOR operations on key fragments. This approach leads to the very efficient hardwired solution [10]. Pearson proposed an elegant implementation of uniform hashing using small random look–up table and XOR operations on key fragments [25].

In addition to already mentioned original contributions, there is a number of surveys about physical database design. We suggest Salzberg' s practical guide for implementation of large file systems based on analytical approach [27], and Graefe' s survey [9].

In this paper we propose an original order preserving dynamic hashing algorithm for large data files. In Section 2 problem statement is given. Section 3 outlines proposed method for physical organization of large files. Algorithms of basic file operations are presented in Section 4, and in Section 5 we give some implementation consideration for the proposed algorithm.

2. PROBLEM STATEMENT

Let us assume that file is collection of N arbitrary fixed size records. Every record has a unique key k, and $\mathbf{K}$ is set of all keys

$$\mathbf{K} = \{k_i \mid 1 \le i \le N\}$$

Key k_i is a string of up to L characters:

$$k_i = c_{i1}\, c_{i2} \ldots c_{il}, \qquad 1 \le l \le L$$

Let file be accessed in three manners: random access, range search, and key sequential access. Random access must be accomplished in close to one disk access for most applications. To satisfy requirements of real–time multimedia applications we will consider here only algorithms that make possible random access in exactly one disk access. Execution time of memory operations compared to disk access time could be neglected. Generally, range search and key sequential access may require many disk accesses, i.e., in a case of hashed files up to N accesses. However, in order to satisfy limitations of real–time systems, we allow only a limited number of disk accesses in implemented algorithms.

We will consider here that single user application has limited amount of available main memory. Having in mind efficiency of B^+–tree algorithms and support for all three types of accesses, it seems that the best approach would be to organize B^+–tree structure in main memory. However for large files it cannot be possible to place whole index structure in main memory, and consequently random access will be performed in

more than one disk access due to access to index structure in secondary storage. Let M_{max} be maximum number of data bucket pointers that can be placed in available main memory by B^+–tree structure on parent–of–leaf node level. The question is: "How can we modify B^+–tree structure to preserve efficiency of the original B^+–tree algorithm when number of data buckets exceeds M_{max} ?"

Our objectives here were to achieve:

- an algorithm that will guarantee O(1) single record retrieval for files of different size,
- efficient range search and key sequential access, as well as support for sorting and sort–based operations,
- fast insert and delete operations.

Algorithms under consideration must take into account upper limit of available internal memory. Table 1 contains notation used in this paper to provide easy referencing.

Table 1. Notation used in this paper

Notation	Definition
N	number of records in a file
MS	available main memory size
k	key value of the currently selected record
l	key length (number of characters)
k_1	number of characters used in hashing function $h1$
w	width of hashing key fragment
c_{ij}	j–th character of key k_i
kd	key delimiter
b	bucket factor (number of records in a bucket)
C	bucket size
a	load factor
a'	initial partition load factor
a''	critical load factor for rehashing
p	starting bucket address
m	number of buckets in a partition
m_{max}	maximum number of buckets in a partition
n	number of records in a partition
M	number of table entries
M_{max}	the largest possible table size

3. THE PROPOSED METHOD

We propose here the method that makes use of slightly modified B–tree index structure based on order preserving hashing function to generate ordered partitions of records, and perfect hashing function to place records within the partition. The proposed method avoids bucket overflow, and rather performs rehashing on a different scale. In this chapter we will describe applied hashing and rehashing algorithms.

3.1. HASHING ALGORITHM

To achieve an efficient file organization we introduce two hashing functions: $h1$ as an order preserving hashing function, and $h2$ as an ordinary perfect hashing function. Using $h1$ over given key set **K** we generate a number of partially ordered sets of records, i.e., *hash* all records of the file into *partitions*. Value of $h1$ for a particular key k, determines the partition, and perfect hashing function $h2$ generates the address of *data bucket* containing record with the given key within that partition. Let each data bucket contain up to b records, and let it be up to M_{max} partitions each with capacity of m_{max} data buckets. Whole range of keys from set **K** is then divided into M partitions using $(M-1)$ comparison values or key delimiters **KD**. Function $h1$ must not generate more then M_{max} partitions due to available memory limitations. All records with key value between adjacent delimiters belong to the same partition, and all partitions should have nearly equal cardinality. Moreover, partitions are in presorted order according to their key values. It was shown that proposed "divide and conquer" technique makes possible an efficient range search and sort based operations [15]. We will define an order preserving hashing function $h1$ over key k, that generates M different values in the following way:

Algorithm 3.1: Hashing function $h1(k)$

```
i = 0;
while ( (k > KD[i]) && (i < (M-1) )          /* Search through key delimiter table */
        i++;
h1(k) = i;                                   /* Hashing function is the partition number */
```

There is a number of possible implementations of hashing function $h2$. The hashing function $h2$ must be a perfect hashing function for the selected key set, so that no one of m data buckets holds more then b records. For the given number of records n, and fixed bucket factor b, we are looking for number of data buckets m, where $m \le m_{max}$, such that hashing function $h2(k,m)$ is perfect for each key k. Search for perfect hashing function is completely different for static and dynamic files. Also, problem of finding minimal perfect hashing function that is very important for static files, is of no relevance for dynamic files. In a case of dynamic files, each partition must have some free space to accommodate following records.

To provide fast access to very large databases in real time environment, $h2(k,m)$ have to fulfill following conditions: fast calculation, machine independent algorithm, and good randomization for different key distributions. Simple algorithm of choice is the folding method. It divides keys into several short fragments and folds them using the eXclusive–OR (XOR) operations. Unfortunately, that algorithm does not resolve anagrams and could produce poor randomization. Therefore, intermediate XOR results are usually modified by bit shifting [10] or using table look–up randomization [25]. Algorithms based on XOR and shift operations are very fast, and could be efficiently hardwarized. Hashing function $h2$ based on shift and XOR operations is described in Algorithm 3.2.

Algorithm 3.2: Hashing function $h2(k,m)$

```
h2(k,m) = k(1)                          /* Take the first key fragment of width w */
for ( i=2; i <= l; i++ ) {
        h2(k,m) <<= 1;                  /* Rotate left one position */
        h2(k,m) = h2(k,m) ^ k(i);       /* XOR with the next key fragment */
        }
h2(k,m) = h2(k,m) mod m;
```

Graphical illustration of the proposed large file organization is given in Figure 1. Hashing function $h1(k)$ makes use of $(M-1)$ auxiliary key delimiters to determine appropriate **HT** table entry, so $h1$ can be considered as directory based hashing function. Each table entry contains a pair of parameters (p, m), where p addresses starting bucket position of the corresponding disk partition, and m is used by ordinary perfect hashing function $h2(k,m)$ to determine the data bucket within the partition which holds the requested record. If hashing table **HT** is present in main memory, single record retrieval with disk access O(1) is guaranteed.

Determining key delimiters set **KD**, is a very sensitive and data dependent problem. If key distribution is uniform, then whole range of key values can be simply divided into M subranges. For instance, if $M = 2^k$, then instead of whole key we can take only the first k bits to create 2^k partitions, as suggested by Lomet [20]. However, in most practical cases key distribution is rarely uniform, so this approach is useless for original keys. Nevertheless, regular partitioning can be used for hashed keys, but sacrificed key order.

If we fix key delimiter values for arbitrary chosen file, the requested nearly equal partition cardinality will not be fulfilled. Of course, it is possible to obtain set of optimal key delimiters for a given file in advance using statistical analysis, but in that case it is a rather static file organization. In practice, due to variable number of records in a file it will be necessary to change number of partitions as well as partition key delimiters from time to time. So, we have to provide an efficient way to dynamically adjust length and content of memory based key delimiters set **KD**. Having in mind that this table can be very large, captured comparison values must be given in sorted order to provide fast search within hashing function $h1(k)$. Delete and insert

operations over key delimiters table must be very fast as well, so some kind of B–tree structure is preferable, as it is noted earlier.

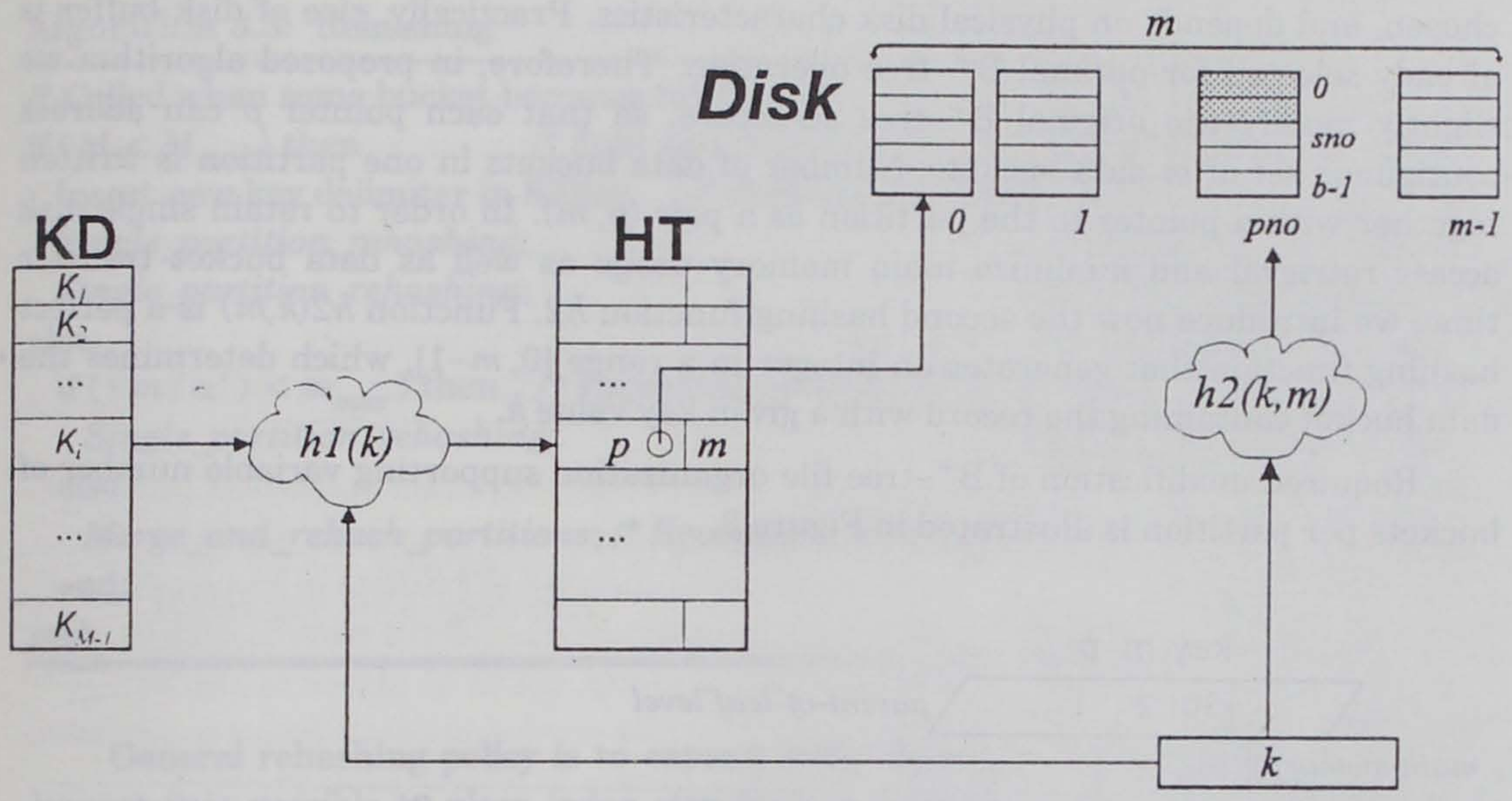

Figure 1. Large file organization using hashing functions $h1(k)$ and $h2(k,m)$

Tables **KD** and **HT** could be combined to generate a table with entries comprised of three parameters (kd, p, m). These parameters can be considered as an aggregate primary key of disk partition. It should be noted that the normal way of operation of B^+–tree is supported as long as the number of data buckets is less then M_{max}. We can start from this point of view to find out the requested solution. For the sake of simplicity, B^+–tree of order 2 used as a primary index is illustrated in Figure 2. Beware that only leaf level contains the data buckets. For a single disk access retrieval, the whole index structure excluding leaf level must be in main memory. It is possible as long as the number of data bucket allows that complete B^+–tree index structure could be placed in allocated main memory, so that address pointer points to the bucket of up to b records on secondary storage.

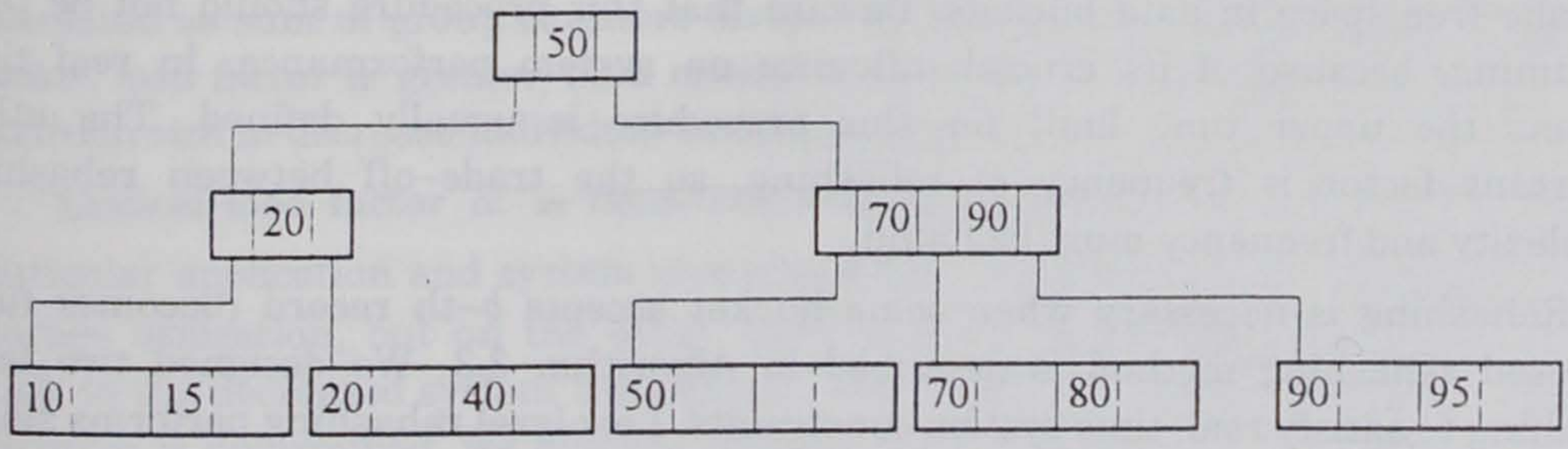

Figure 2. Primary B^+–tree index structure

Problem under consideration is physical organization of large files, where the number of data buckets exceeds M_{max}. We can resolve this problem by "brutal force", for instance by increasing the bucket size b. However, bucket size is not arbitrary chosen, and depends on physical disk characteristics. Practically, size of disk buffer is already selected for optimal B^+–tree operation. Therefore, in proposed algorithm we slightly modify the original B^+–tree structure, so that each pointer p can address contiguous set of m data buckets. Number of data buckets in one partition is written together with a pointer to the partition as a pair (p, m). In order to retain single disk access retrieval and minimize main memory usage as well as data bucket transfer time, we introduce now the second hashing function $h2$. Function $h2(k,m)$ is a perfect hashing function that generates an integer in a range $[0, m-1]$, which determines the data bucket containing the record with a given key value k.

Required modification of B^+–tree file organization supporting variable number of buckets per partition is illustrated in Figure 3.

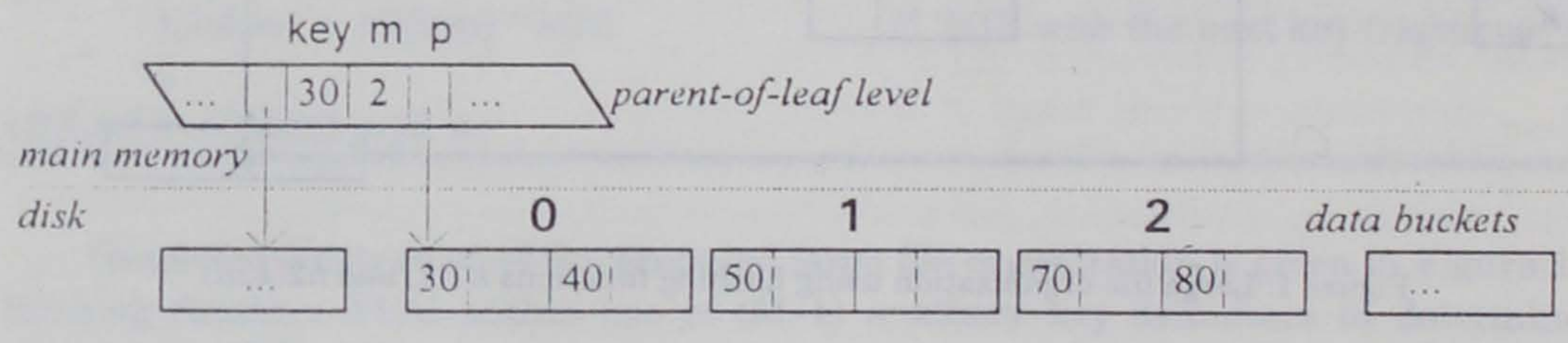

Figure 3. Fragment of a parent–of–leaf and leaf level of the modified B^+–tree

3.2. REHASHING ALGORITHM

We already mentioned that hashing function $h2(k,m)$ is a perfect hashing function. It is conditionally true, because the proposed method uses overflow avoidance paradigm. The algorithm guarantees that every new record does not make the bucket overflow, but the data bucket must be checked for the number of records after writing. If the data bucket became full, rehashing of corresponding partition is necessary.

Rehashing has to reconstruct the partitions and their corresponding data buckets, to make free space in data buckets. Beware that this procedure should not be time consuming, because of its crucial influence on system performance. In real time systems the upper time limit for this procedure is usually defined. The other important factor is frequency of rehashing, so the trade–off between rehashing complexity and frequency must be found.

Rehashing is necessary when some bucket accepts b–th record (becomes full). Proposed rehashing method is described in Algorithm 3.3. We designed two level rehashing to satisfy real–time system constraints. Low level rehashing performs single partition rehashing while high–level rehashing reorganize multiple partitions and index structure (tables **HT** and **KD**). High–level rehashing is required when partitions

become extremely skewed or too large. This operation can be very time consuming, and it is desirable to perform it off–line.

Algorithm 3.3: Rehashing

```
/* Called when some bucket becomes full */
if ( M < M_max ) then          /* Split partition as normal mode of B+–tree operation */
  Insert new key delimiter in KD[i];   /* Existing partition is split in two key ranges */
  Single_partition_rehashing;          /* Rehash i–th partition */
  Single_partition_rehashing;          /* Rehash (i+1)–th partition */
else                                   /* Index structure full */
  if ( ( m / α' ) < m_max ) then   /* Expanded partition could be placed in main memory */
    Single_partition_rehashing;        /* Rehash existing partition */
  else
    Merge_and_rehash_partitions; /* Reorganize some partitions and index structure */
  end;
end;
```

General rehashing policy is to expand index structure (number of partitions) as long as it is possible to place index structure in available memory. When number of partitions reach M_{max}, partitions are enlarged by increasing number of data buckets. Finally, when some partition get too many data buckets to be placed in main memory, we have to reorganize it together with adjacent partitions to equalize partition load factor. It will result in a change of key delimiters **KD** as well.

Single partition rehashing, presented in Algorithm 3.4, is an on–line procedure, which enables hashing function $h2(k,m)$ to operate without overflow buckets. Low level rehashing starts with reading all data buckets from the requested partition into main memory. We have to determine new number of data buckets (m) in the rehashed partition, so that load factor α of the partition is close to initial ($\alpha \leq \alpha'$). Initial load factor α', is determined to allow a number of insertions without rehashing. In order to determine distribution of hashed keys, we use rehash table (**RT**) to collect histogram of hashed values. Number of **RT** table entries must be large enough (compared with m_{max}) to provide uniform distribution. Having in mind that value of $h2$ must be in a range $[0, m–1]$ table **RT** is split into m groups. Every group is associated with the corresponding data bucket. Consequently, any bucket loading factor could be calculated as sum of group counters divided by bucket size b. If maximum calculated bucket load factor is greater than critical load factor α'', we have to further expand partition size to decrease individual bucket load factors.

Critical load factor α'' is determined as space/time trade–off, and depends on particular application and system characteristics. Higher value of α'' (≈ 1) maximize storage utilization, but on the other hand increase probability of rehashing, which leads to the decreased system throughput. When number of data buckets in rehashed partition is determined, we allocate appropriate contiguous disk area. After that, we

collect records from the partition with the same hash value $h2(k,m)$ and write them, bucket by bucket, to the disk.

Algorithm 3.4: Single_partition_rehashing (Low–level rehashing)

```
mt = size(RT);          /* Calculate temporary number of buckets as RT table length */
n = 0;                                   /* Initialize partition record counter */
do {                                     /* Generate histogram of hash values */
        get_next_record;                 /* Get next record within the partition */
        n++;                             /* Update partition record counter */
        Calculate h2(k,mt);
        RT[ h2(k,mt) ] ++;               /* Increment rehash table counter */
} while ! end_of_partition;
m := n / (b · α');                       /* The lowest expected number of buckets */
do {                                     /* Searching for optimal partition size */
        m ++;                            /* Increment number of buckets in the partition */
        Split rehash table RT entries into m groups;   /* All records with keys
                                         that belong to a group, are to be placed in one bucket */
} while ( max(α) > α'' );                /* Partition grows while any bucket load factor is critical */
/* Bucket load factor is sum of corresponding counters in rehash table group */
Allocate m free disk buckets for rehashed partition;
Insert all records from old partition into the new disk partition, according to h2(k,m).
```

In a presence of data skew, nonuniform key value distribution, number of buckets for some partitions can be prohibitively large. As a consequence, the performance of the key sequential and key range access becomes very poor. Partitions with large number of records will require less frequent but more complex rehashing, and the time for rehashing will increase. This problem is resolved using high level rehashing.

High level rehashing is requested in three cases. First, it is necessary when initial number of data buckets is larger then allowed number of data buckets m_{max}, as given in previous analysis. Second, high–level rehashing could be required during execution of single partition rehashing, if number of data buckets exceeds m_{max}. Finally, it is advisable to perform high level rehashing off–line from time to time to optimize bucket distribution within partitions. As a consequence, better balancing of data buckets will decrease cost of on–line high–level rehashing in real–time systems, and increase system performance.

The first step of high–level rehashing determines q adjacent partitions, so that average number of data buckets per partition is less then m_{max}. It is very fast operation, executed simply by reading parameter m from q adjacent entries of **HT** table. The next step merges records from adjacent partitions, and modify corresponding key delimiters from **KD** table. Records that should be moved to another partition are obtained applying fast partial sorting based on modified radix sort [13, 22]. The final step applies already presented single partition rehashing to generate q new partitions. Initial file loading can be considered as a high level rehashing problem.

4. BASIC OPERATIONS

Retrievals are the basic file operations. Presented here are: Retrieval of a single record, Set retrieval, Set retrieval in a sorted order, Insertion and Deletion. Afore mentioned Algorithm 3.4 guarantees single disk access retrieval for any requested record, provided that **HT** and **KD** data structures reside in main memory. Elementary sort and select operations used in a basic algorithms are described in [13, 15].

Algorithm 4.1: Retrieval of a single record with key k

Calculate $h1(k)$, $h2(k,m)$; /* Calculate hash functions $h1$ and $h2$ */
p = **HT**[$h1(k)$] . p; /* Pointer to the disk partition */
m = **HT**[$h1(k)$] . m; /* Number of buckets in the disk partition */
Compute the bucket address $A := p + h2(k,m)$; /* Bucket logical address */
Read in the bucket with address A;
Search bucket for a record with a given key k ; /* if not found there is no record with key k */

Algorithm 4.2: Retrieval of a set of records with key range [$k1$, $k2$]

Calculate starting and ending partition given by $h1(k1)$ and $h1(k2)$;
From starting partition $i = h1(k1)$ select all records with corresponding key k satisfying condition $k \geq k1$.
While partition number $i < h2(k,m)$ /* partitions are already in sorted order! */
    Select all records from partition i
From ending partition $i = h1(k2)$ select all records with corresponding key k satisfying condition $k \leq k2$.

Algorithm 4.3: Retrieval of a set of records with key range [$k1$, $k2$] in sorted order

Calculate starting and ending partition given by $h1(k1)$ and $h1(k2)$;
From starting partition $i = h1(k1)$ sort all records with corresponding key k satisfying condition $k \geq k1$
/* Suggested sort algorithm implements select & sort operation */
While partition number $i < h2(k,m)$ /* partitions are already in sorted order! */
    Sort_partition; /* Sort records from partition i */
From ending partition $i = h1(k2)$ sort all records with corresponding key k satisfying condition $k \leq k2$

Algorithm 4.4: Insertion of a record with key k

```
Calculate h1(k), h2(k,m);                    /* Calculate hash functions h1 and h2 */
/* Find a bucket within the partition */
p := HT[ h1(k) ] . p;
m := HT[ h1(k) ] . m;
Compute the bucket address A := p + h2(k,m);          /* Bucket logical address */
Insert record in the bucket;
if bucket_full then                  /* No free space in a bucket, rehash required */
    Rehash_partition;
end;
```

Algorithm 4.5: Deletion of a record with key k

```
Calculate h1(k), h2(k,m);                    /* Calculate hash functions h1 and h2 */
/* Find a bucket within the partition */
p := HT[ h1(k) ] . p;
m := HT[ h1(k) ] . m;
Compute the bucket address A := p + h2(k,m);          /* Address of the bucket
                                                         with the given key value */
Read in the bucket with address A;
Search for record with key k in the bucket;
if found then    {                           /* Record with key k found in the bucket */
  Delete record in the bucket;
  if bucket_empty then
    Rehash_partition;          /* Optional rehashing, possible off-line processing */
  end;
  }
else
  return not_found;                  /* Record with key k is not found in the bucket */
end;
```

5. IMPLEMENTATION CONSIDERATIONS

Practical implementation of the proposed algorithm must take into consideration physical system constraints, such as amount of available memory, disk access time and I/O transfer rate. We will present here some hints based on our experience in database machine realization [14].

In a case that file has more then M_{max} buckets, tables **HT** and **KD** are fixed size and the binary search is the fastest way to retrieve data from tables. However, we are dealing with dynamic files, and therefore the length and content of memory based tables are variable. Accordingly, modified B^+-tree structure outlined in previous sections is better solution for different number of records.

Our idea was to *extend* existing information in B$^+$–tree structure algorithm, to support basic operations on both moderate and large size files at the same time. Yu has shown that average key length in current databases is 9.5 bytes, and majority has less then 20 bytes [29]. Therefore, typical additional byte of information about number of data buckets m in a partition requires approximately 5% more of key space in a table.

5.1. CHOOSING SYSTEM PARAMETERS

Bucket size (C) is dependent of system characteristics, for instance disk sector size, I/O device buffer size, etc. Anyway, larger bucket size means slower random reading by record, but faster random reading by bucket. Usually, bucket size is between 512B and several tenths of kilobytes.

Maximum number of buckets in a partition (m_{max}) depends on number of buckets that could be placed in main memory of size MS and processed at the same time. It is particularly important for sort operation. Consequently, it is desirable that the number of buckets in a partition of the file satisfy:

$$m \le m_{\max} = \left\lfloor \frac{MS}{C} - 1 \right\rfloor$$

The second important constraint originates from initial load or multiple partition rehashing requirements. The best approach would be to execute these operations in two phases:

I Hash all records into M buffers using hashing function $h1(k)$

II Hash records from each buffer into m data buckets using hashing function $h2(k,m)$

The maximum number of partitions (M_{max}) could be calculated according to the following relation:

$$M \le M_{\max} = \left\lfloor \frac{MS}{C} - 1 \right\rfloor$$

It should be noted in previous formulas that one bucket (of size C) is reserved for input or output data. Finally, having in mind above constraints, proposed algorithm guarantees that performance would not be sacrificed as long as the number of records in a file satisfies the following conditions:

$$N \le b \cdot m_{\max} \cdot M_{\max} = b \cdot \left(\left\lfloor \frac{MS}{C} - 1 \right\rfloor \right)^2$$

The former formula for records of fixed size *record_length* can be rewritten as:

$$N \le \left\lfloor \frac{C}{record\_length} \right\rfloor \cdot \left(\left\lfloor \frac{MS}{C} - 1 \right\rfloor \right)^2$$

EXAMPLE: Let us consider that system has available main memory of 4MB, and let the bucket size be 4KB. If record length is 200B, then applying derived formulas we have that the number of file records N could be as large as 20×10^6 records, where total

file size is up to 4GB. At the same time, required hash table size is approximately 1K entries, or probably less then 30KB for key length of 20B! For the purpose of illustration only, classical B^+–tree allowing O(1) single record disk access would require more then 25MB of main memory.

Above calculation is not optimal from the point of view of total I/O cost. It is shown that larger bucket is more convenient for sorting and set retrieval operations. Consequently, optimal bucket size depends on mix of basic file operations in given applications [9].

For fixed bucket size C and bucket factor b, expected upper number of records per bucket b' is function of number of records n and buckets m in the partition. To minimize number of requests for rehashing, we introduce critical bucket load factor α'' as:

$$\frac{b'}{b} \le \alpha'' \le 1$$

During rehashing load factor of all buckets must be lower then the critical bucket load factor α''. Let us assume that $h2$ is an ideal hashing function, generating m different values. Number of collisions of function $h2$ can be approximated with Poisson distribution [4]. Probability that in a partition of n records, there is k collisions, is given by:

$$p(k) \approx \frac{\lambda^k}{k!} e^{-\lambda}$$

where λ is:

$$\lambda = \frac{n}{m}$$

Gonnet derived a rather complex formula for the length of the longest probe sequence in a case of hashing with separate chaining [7]. It was shown that for fixed n increasing number of buckets will decrease the expected longest probe sequence (expected upper number of records in a bucket). On the other hand, for the fixed load factor α , increasing m will cause very slow growth of upper number of records in a bucket with order $O(\ln m / \ln \ln m)$. This dependency should be taken into consideration to determine critical bucket load factor α''. Critical table load factor must be chosen to satisfy the condition that the longest probe sequence could be placed in a bucket.

5.2. Acceleration of Basic Operations

Proposed large file organization is designed to satisfy requirements for random access, range search and key sequential access with minimal number of disk accesses. Range search, key sequential access as well as rehashing relies on efficient partial sorting of data records. Although quicksort and heap sort are usually used, we implemented modified radix sort algorithm as described in [14, 15]. The main reasons for applying this algorithm to accelerate basic operations are:

- extremely fast generation of partial ordering,
- possible hardwarization,
- fast sorting for large number of records, due to O(N) algorithm complexity.

Large file organization with applied extended radix sort mechanism is illustrated in Figure 4. Let us have n records to be sorted in main memory. The proposed sorting algorithm makes use of first $k1$ characters (key prefix) of each key to generate set of *classes*. All members of a class has equal key prefix on first $k1$ characters, and classes are in sorted order in relation to corresponding prefixes. Size of table **RT** depends on key prefix length ($k1$). Table **RT** contains headers of all classes, while other class members are chained using pointers placed in chain table **CT**.

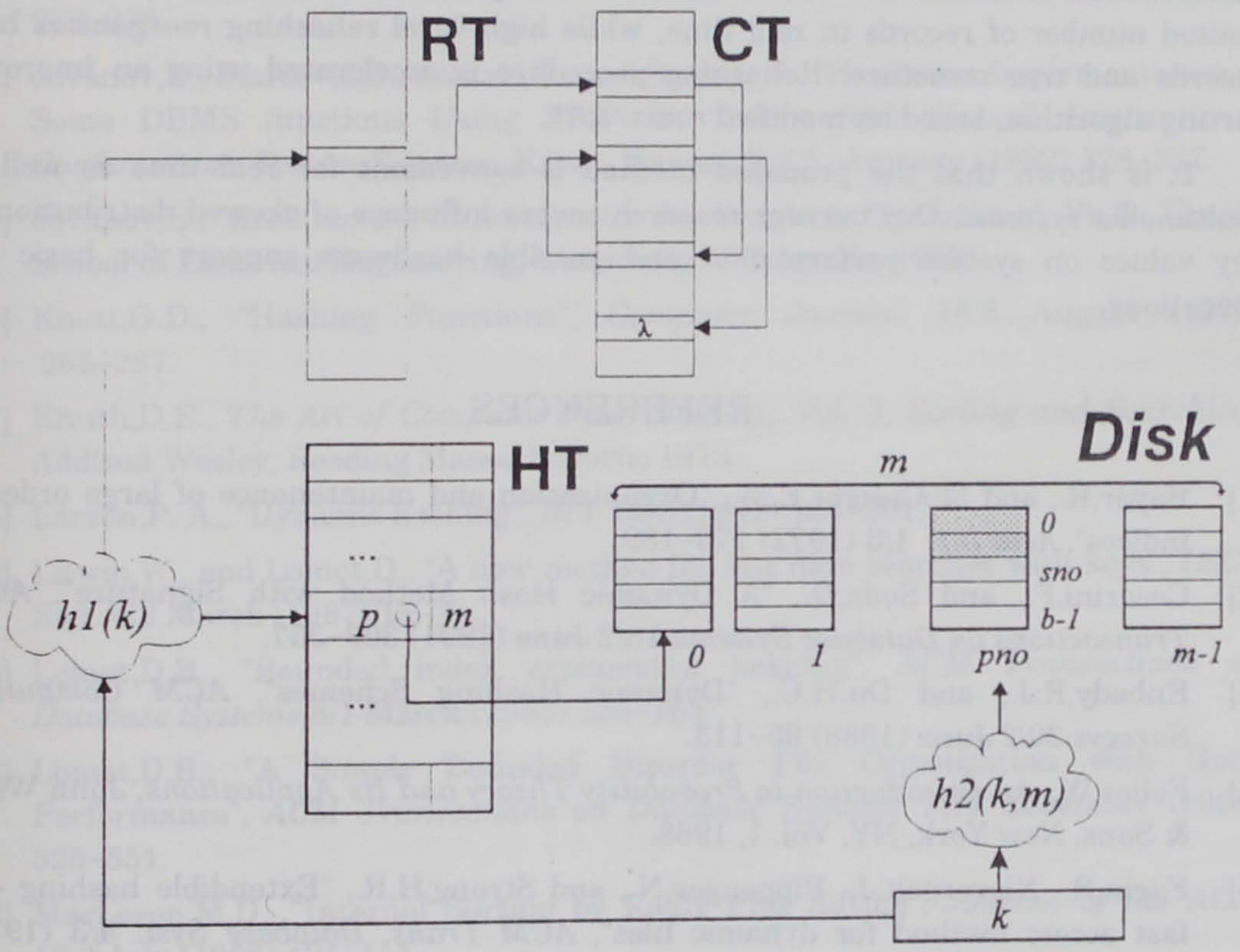

Figure 4. Large file organization with applied extended radix sort mechanism

6. CONCLUSION

In this paper we proposed a novel method for physical file organization, based on order preserving hashing scheme. Having in mind that present information systems make use of ever increasing data file sizes, with limited amount of main memory, we evaluated a unique and adaptable method that can be applied for files ranging from moderate to very large files. Moreover, for real–time and multimedia systems fast retrieval of single record must be guaranteed independently from file size.

The method combines an order preserving and an ordinary perfect hashing function, to achieve single disk access retrieval and support other file operations using modified B^+–tree structure. In addition to already proposed hashing schemes, our method provides an efficient support for sorting and sort–based file operations, as range search and key sequential operations. The performance is nearly the same as B^+–tree algorithms, and penalty is only 5–10% of key space in tree structure. In comparison to standard B^+–tree technique, where single disk access retrieval can be achieved only for limited number of records determined by the amount of available main memory, our approach guarantees one disk access retrieval almost for any number of records using the same main memory.

We also introduced a new rehashing policy to increase the overall system performance. It makes use of two level rehashing. Low–level rehashing reorganizes limited number of records in real time, while high–level rehashing reorganizes both records and tree structure. Rehashing procedure is accelerated using an improved sorting algorithm, based on modified radix–sort.

It is shown that the proposed method is convenient for real–time as well as multimedia systems. Our current research covers influence of skewed distribution of key values on system performance, and possible hardware support for basic file operations.

REFERENCES

[1] Bayer,R., and McCreight,E.M., "Organization and maintenance of large ordered indices", *Acta Inf.* 1/3 (1972) 173–189.

[2] Cesarini.F., and Soda,G., "A Dynamic Hash Method with Signature", *ACM Transactions on Database Systems* 16/2 June (1991) 309–337.

[3] Enbody,R.J., and Du,H.C., "Dynamic Hashing Schemes", *ACM Computing Surveys* 20/2 June (1988) 85–113.

[4] Feller,W., *An Introduction to Probability Theory and Its Applications*, John Wiley & Sons, New York, NY, Vol. 1, 1968.

[5] Fagin,R., Nievergelt,J., Pippenger,N., and Strong,H.R., "Extendible hashing – a fast access method for dynamic files", *ACM Trans. Database Syst.* 4/3 (1979) 315–344.

[6] Fox,E.A., Heath,L.S., Chen,Q.F., and Daoud,A.M., "Practical Minimal Perfect Hash Functions for Large Databases", *Communications of the ACM* 35/1 January (1992) 95–121.

[7] Gonnet,G.H., "Expected Length of the Longest Probe Sequence in Hashe Code Searching", *Journal of the ACM* 28/2 April (1981) 289–304.

[8] Gonnet,G.H., and Larson,P., "External Hashing with Limited Internal Storage", *Journal of the ACM* 35/1 January (1988) 161–184.

[9] Graefe,G., "Query Evaluation Techniques for Large Databases", *ACM Comp. Surveys* 25/2 June (1993) 73–159.

[10] Inoue,U., Satoh,T., Hayami,H., Takeda,H., Nakamura,T., and Fukuoka,H., "RINDA: A Relational Database Processor with Hardware Specialized for Searching and Sorting", *IEEE Micro* December (1991) 61–70.

[11] Ishikawa,H., Suzuki,F., Kozakura,F., Makinouchi,A., Miyagichima,M., Izumida,Y., Aoshima,M., and Yamane,Y., "The Model, Language, and Implementation of an Object–Oriented Multimedia Knowledge Base Management System", *ACM Transactions on Database Systems* 18/1 March (1993) 1–50.

[12] Johnson,T., and Shasha,D., "The Performance of Concurrent B–tree Algorithms", *ACM Transactions on Database Systems* 18/1 March (1993) 51–101.

[13] Jovanov,E., Aleksić,T., Stojkov,Z., and Starčević,D., "A Sorting Processor for Microcomputers", *Microprocessing and Microprogramming* 23/1–5 (1988), 273–278.

[14] Jovanov,E., Starčević,D., Aleksić,T., and Stojkov,Z., "Hardware Implementation of Some DBMS functions Using SPR", in *Twenty–fifth Hawaii International Conference on System Sciences*, Kauai, Hawaii, Vol.1, January (1992) 328–337.

[15] Jovanov,E., "Architecture of Accelerator for Database Operations", Ph.D. Thesis, School of Electrical Engineering, University of Belgrade, 1993.

[16] Knott,G.D., "Hashing Functions", *Computer Journal* 18/3 August (1975) 265–287.

[17] Knuth,D.E., *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison Wesley, Reading Massachusetts, 1973.

[18] Larson,P. A., "Dynamic hashing", *BIT* 18/2 (1978) 184–201.

[19] Litwin,W., and Lomet,D., "A new method for fast data searches with keys", *IEEE Soft*. 4/2 March (1987) 16–24.

[20] Lomet,D.B., "Bounded index exponential hashing", *ACM Transactions on Database Systems* 8/1 March (1983) 136–165.

[21] Lomet,D.B., "A Simple Bounded Disorder File Organization with Good Performance", *ACM Transactions on Database Systems* 13/4 December (1988) 525–551.

[22] MacLaren,M.D., "Internal Sorting by Radix Plus Sifting", *Journal of the ACM* 13/3 July (1966) 404–411.

[23] Milenković,Č., Starčević,D., "Computer System Architecture for Multimedia Information Systems", in *Proc. of the ISMM Intl. Symposium on Microcomputers and their Applications*, Cairo, Egypt, March 3–5 (1987).

[24] Milenković,Č., Starčević,D., Mučibabić,B., "PC–based Multimedia Messaging Systems", in *Proc. of the Thirteenth Symposium on Microprocessing and Microprogramming Euromicro 87*, Portsmouth, Great Britain, September 14–17 (1987).

[25] Pearson,P.K., "Fast Hashing of Variable–Length Text Strings", *Communications of the ACM* 33/6 June (1990) 677–680.

[26] Ramakrishna,M.V., and Larson,P., "File Organization Using Composite Perfect Hashing", *ACM Transactions on Database Systems* 14/2 June (1989) 231–263.

[27] Salzberg,B., *File structures: An analytical Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[28] Yao,A.C., "Uniform hashing is optimal", *Journal of the ACM* 32/3 July (1985) 687–693.

[29] Yu,P.S., Chen,M.S., Heiss,H.U., and Lee,S., "On Workload Characterization of Relational Database Environments", *IEEE Trans. on Software Engineering* 18/4 April (1992) 347–355.