A Dynamic Hash File for Random and Sequential Accessing

Jack A. Orenstein

Department of Computer and Information Science
University of Massachusetts, Amherst

Abstract
Linear hashing is a dynamic file organization proposed by Litwin. A version of it classifies records using a suffix of the binary representation of the key. Random accessing is possible but sequential accessing is not. By using a prefix instead, sequential accessing is possible but not always efficient. Other modifications are included to avoid poor performance for sequential accessing. In particular, only a limited number of sparsely filled buckets are allowed to exist. The performance of the new data structure is, in practice, expected to be better than that of the Btree.

1.0 INTRODUCTION

The basic operation performed on a file of records is the retrieval of a record given its key. This problem has generated a huge amount of research. (See [Knut73] or [Stan80] for a survey.) A hash file provides fast access to a record (in most cases). Btrees [Baye72] are somewhat slower but permit sequential accessing. That is, once a record has been retrieved, its successor can be found easily. Thus the Btree is an "indexed-sequential" data structure (ISDS).

Litwin has proposed a data structure, "linear hashing", which supports random but not sequential accessing [Litw80]. Linear hashing is dynamic and provides access to the primary page of any bucket in exactly one disk access. But linear hashing is not an ISDS because sequential accessing cannot be performed quickly: the address of the successor of r, a record in the file, is not related to the address of r. Other hashing methods can be made order-preserving in a trivial way: by using the hash function $h(r) = \lceil r/s \rceil$ where r is an integer and s is a fixed scaling factor. Linear hashing requires the use of hash functions incompatible with this strategy. We propose two variations of linear hashing which are order-preserving. Both random and sequential accessing will then be possible.

The performance of these new data structures is at least as good as that of the Btree: for uniformly distributed data a random access will usually cost one disk access. If the data is highly clustered then the cost of a random access is the same as for a Btree (or marginally more). In some rare cases insertion will be more expensive.

In the context of a relational database system, the ability to process "range queries" on multiple attribute data is important. It was shown in [Oren82] that any ISDS can be adapted for this purpose. So our new data structures yield new data structures for range searching.

We will use the following notation: $\langle s_1{:}n_1 \mid s_2{:}n_2 \mid \ldots \mid s_r{:}n_r \rangle$ denotes the string

$$\underbrace{s_1\ s_1\ \cdots\ s_1}_{n_1}\ \underbrace{s_2\ s_2\ \cdots\ s_2}_{n_2}\ \cdots\ \underbrace{s_r\ s_r\ \cdots\ s_r}_{n_r}$$

where each s_i is a string of one or more bits and each $n_i \geq 0$, i = 1, ... , r, is a repetition factor. If $n_i = 1$ then "$s_i{:}n_i$" may be abbreviated to "s_i". (E.g. <011:2 | 0> = 0110110.)

2.0 A VARIATION OF LINEAR HASHING

2.1 Linear hashing

A description of a special case of linear hashing is a prerequisite. The data to be stored consists of records of d bits each. Record r can be regarded as the integer $\langle r_0 \mid \ldots \mid r_{d-1} \rangle$ where r_i is the ith bit of the record. The records are to be stored in buckets 0, 1, ... The number of a bucket will also be the address of the bucket's primary page. Overflow pages are allocated from a separate address space.

The file is accessed using hash functions of the form

$$h_i(r) = r \bmod 2^i$$

(This function extracts the i bit suffix of r.) The value of i is one of two consecutive integers, m and m+1, where m is the *level* of the file. A pointer to the file, n, indicates whether h_m or h_{m+1} should be used, (see figure 1).

Buckets 0 through n-1 and 2^m through $2^m+(n-1)$ are at level m+1; buckets n through 2^m-1 are at level m. n is a pointer to the next bucket to be *split*. It travels from left to right so that every bucket is split in turn. When bucket n is split, its records are distributed between buckets n and $n+2^m$, both of which will then be at level m+1, (since n was incremented). When n reaches 2^m all buckets are at level m+1. m is then incremented, n is reset to 0 and starts travelling right again. (See the Split algorithm in section 2.3.)

A linear hash file can be grown from a single empty bucket at level 0. The number and address of this bucket are 0, m = 0 and n = 0. m = 0 means that 0 bit suffixes are used for classification: all records go to bucket 0.

A bucket is split (and n is incremented) whenever a record hashes to a full primary page, (i.e. there is a collision). The bucket that is split is not, in general, the one involved in the collision. But eventually, every bucket will be split and (ideally) all the overflow pages will be emptied and reclaimed. If splitting creates sparsely filled buckets and the load factor threatens to become "too low", the split is suppressed.

Litwin claims that a linear hash file can also shrink [Litw80] although he does not give the deletion algorithm. It is not difficult to imagine how deletions might be handled. For example, when the overflow pages of any bucket become empty, buckets n and $n+2^m$ could be combined and n decremented. (See the Join algorithm in section 2.3.)

To locate a record, r, Randac (r) (for "random access") is called to locate the bucket. We are not concerned with searching within the bucket. Randac returns the number of the bucket containing r.

```
Randac(r)
B := h_m(r)
if B < n
then
   (* bucket B has been split to *)
   (* give two buckets on level m+1 *)
   B := h_{m+1}(r)
end
return(B)
end Randac
```

One attractive feature of linear hashing is that the file grows smoothly; by one bucket at a time. The growth is "linear". The directory of extendible hashing, on the other hand, grows exponentially: it doubles in size occasionally (but these expansions are rare). In addition, buckets of extendible hashing split when they become full, requiring the directory to be updated [Fagi79]. Linear hashing does not have a directory.

2.2 Order-preserving linear hashing, (OPLH)

Consider the partitioning imposed by the hash function $h_m(r) = r \bmod 2^m$. All of the records in a given bucket (at level m) agree in the m least significant bits, $<r_{d-m} \mid \ldots \mid r_{d-1}>$. (Note that m varies as records are added and deleted. Thus, the number of bits used for classification is not fixed.)

If, instead, the records agreed in the most significant bits, each bucket would store all of the records of the file that fall in a certain range. The hash table would then be order-preserving. Let left(s,k) and right(s,k) denote, respectively, the k leftmost and rightmost bits of string s. Mir($<s_1 \mid s_2 \mid \ldots \mid s_r>$) is the "mirror image", $<s_r \mid \ldots \mid s_2 \mid s_1>$ where each s_i is a single bit.

The simplest way to partition the file on the basis of the most significant bits is to store record r in bucket $h_m(\text{mir}(r))$. That is, the bits are reversed before hashing. Clearly

$$h_m(\text{mir}(r)) = \text{right}(\text{mir}(r),m) = \text{mir}(\text{left}(r,m))$$

The bucket number is obtained by reversing the bits of the m bit prefix of the record. Searching and splitting work exactly as for linear hashing. This has to be true since, in effect, we are dealing with another file in which each record, r, has been replaced by mir(r). If the bits of the prefix were not reversed, i.e. $h_m(r)$ = left(r,m), then splitting works differently. This alternative is discussed in [Oren82].

Figure 2 shows an example of an order-preserving linear hash file. Notice that the mirror image of the m (or m+1) bit prefix of a record at level m (or m+1) matches the m (or m+1) bit representation of the bucket number. So bucket $3 = 011_2$ is at level 3 and stores records with prefix 110_2.

Burkhard [Burk83] and Ouksel and Scheuermann [Ouks83] have independently discovered OPLH. They apply techniques similar to those of [Oren82] to yield a multidimensional data structure for range searching. They do not discuss the use of $h_m(r)$ = left(r,m), nor do they address certain serious problems with the performance of OPLH which are discussed in detail in sections 4 and 5 of this paper. Also, their multidimensional transformations are limited to the context of OPLH. For a more complete and general discussion of the transformation see [Oren82].

2.3 Algorithms

Order-preserving linear hashing (OPLH) supports random and sequential accessing. Randac, given in section 2.1, can be used with OPLH if the argument to h_m and h_{m+1} is changed from r to mir(r).

Locating the successor of any record in a bucket, other than the last one, is trivial. Let last(b) be the last record in bucket(b). (b is a bucket number. Bucket(b) stores the records whose prefixes are mir(b).) Then the successor of last(b) can be found as follows:

The level of the bucket, m_b, is known: $m_b = m$ if $n \le b < 2^m$; $m_b = m+1$ otherwise. Bucket(b) stores all records in the range [<mir(b) | 0:d-m_b>, <mir(b) | 1:d-m_b>], (recall that b is an m_b bit number). The smallest record above this range is

$$S = \langle mir(b) \mid 1{:}d\text{-}m_b \rangle + 1$$
$$= \langle mir(b)+1 \mid 0{:}d\text{-}m_b \rangle$$

A search for S (using Randac) will locate b', the bucket containing the successor of last(b). There is one problem: b' may be empty. Repeating the above procedure until a non-empty bucket is found yields the bucket containing the successor of last(b). The algorithm is given in [Oren82].

The bucket splitting and joining algorithms are Split and Join.

```
Split()
Bit(<r_0 | ... | r_{d-1}>,i) is r_i.

for each record r in bucket n
   if bit(r,m) = 0
   then
      (* the record does not move *)
   else
      move r to bucket n + 2^m
   end
end

n := n + 1
if n = 2^m
then
   (* all buckets are at level m+1, *)
   (* start a new level *)
   n := 0
   m := m + 1
end

return
end Split
```

```
Join()
n := n - 1
if n < 0
then    (* go down one level *)
   m := m - 1
   n := 2^m - 1
end

move all records from bucket n+2^m to bucket n

return
end Join
```

3.0 OVERFLOW

The hash function used for OPLH is $h_m(r)$ = mir(left(r,m)). It is possible that the hash values generated will be clustered. That is, left(r,m) may not scatter the records very well. So overflow will be more common than with more traditional hashing methods.

So far, almost nothing has been said about how overflow is dealt with. Litwin suggests the use of overflow chains [Litw80]. A Btree (or variant) is a much more appropriate data structure in the present context. Since we want OPLH to be indexed-sequential, the data structure representing an overflowing bucket must be also. A chain of records is not an ISDS because it does not support random accessing. The Btree is necessary if we are to avoid the very bad worst case behaviour characteristic of overflow chains.

Clearly, this organization has very good performance for random accessing: reading a bucket that has not overflowed costs one disk access. In the worst case a Btree containing all of the records has to be searched. Sequential accessing is usually as fast as for the Btree.

The use of a Btree complicates operations on buckets: Split and Join. It is essential that these operations preserve the properties of Btrees, (the load factor in particular). The implementation of the Btree operations has been discussed in [Oren82].

4.0 MULTI-LEVEL OPLH, (MLOPLH)

4.1 Problems with OPLH

An OPLH file may contain an arbitrary number of sparsely filled buckets. This can result in poor performance for sequential accessing. Consider the situation shown in figure 3, (suppose that primary page capacity is four records). To retrieve all the records whose prefix is 00_2, buckets 0, 4 and 8 must be accessed, (this is clear from figure 2). These three disk accesses yield two records. If the entire file were at

level 2 then bucket 0 would contain the records which would be retrieved in one access, (since primary page capacity is 4). But if the file were at level 2, other searches, (e.g. for prefix 010_2), would be more expensive. Furthermore, it would take six joins to reach level 2. So the problem is not solved by joining more frequently.

The situation demonstrated in figure 3 is characterized by the appearance of several sparsely filled buckets. It can occur following a sequence of splits which distribute the records unevenly or following repeated deletions concentrated in a few buckets. Since it can occur as a result of deletions, suppressing splits does not solve the problem either.

4.2 Adding more levels to OPLH

Linear hashing, as described in section 2.1, is based on the binary trie: it classifies records according to a sequence of bits whose length varies with the number of records being stored. Other data structures based on this idea are extendible hashing [Fagi79], EXCELL [Tamm81a], HCELL [Tamm81b], trie hashing [Litw81] and, of course, the trie [Fred60, Knut73]. What all of these data structures have in common is the notion of "level". The *level* of a record is the number of bits used in its classification. Records are usually grouped into buckets (as we are doing). The level of a bucket is the level common to all records in the bucket.

The trie stores a record at the lowest level (i.e. nearest the root) providing a classification which avoids bucket overflow. The same is true of extendible hashing and EXCELL but each of these has a directory with all entries at the same level. Linear hashing (and OPLH) use no more than two consecutive levels but do not require directories.

The problem with OPLH, described above, would be alleviated if parts of the file could be stored at lower levels than normal, (i.e. below level m or m+1). E.g. if, in figure 3, the contents of buckets 0, 4 and 8 could be stored in a level 2 bucket, (corresponding to prefix 00_2), leaving the rest of the file at levels 3 and 4, the problem would be solved. Next, we discuss a "multi-level" version of OPLH, MLOPLH.

4.3 Sub-normal buckets

A bucket is *sparse* if it contains no more than a given number of records (which is a fraction of the capacity of the primary page). A sparse bucket will not be permitted to exist. It will be combined with its *brother* to form a *sub-normal* bucket: a bucket whose level is lower than normal. If level(b) < NormalLevel(b) then b is sub-normal. (Level(b) is the level of the bucket and NormalLevel(b) is m or m+1.) b and b′ are brothers iff level(b) = level(b′) and $|b-b'| = 2^{level(b)-1}$

Note that brothers are combined if at least one of them is sparse.

Not all buckets have brothers. In figure 3, buckets 0 and 8 are brothers but bucket 4 has no brother. (Before bucket 0 was split to give buckets 0 and 8, buckets 0 and 4 were brothers.)

Notice that buckets 0, 4 and 8 are sparse. To eliminate the problem, 0 and 8 are joined (this is possible because they are brothers), yielding a level 3 bucket at address 0. The resulting bucket is the brother of bucket 4 so buckets 0 and 4 can now be joined. The result of the two joins described is shown in figure 4.

Degenerate splits.

A split may yield one or two sparse buckets. It is not feasible to refrain from splitting until the situation changes: all further splits are also delayed. Instead, the bucket that should have been split remains at its current level and n, (the pointer to the next bucket to be split), is advanced. This is a *degenerate* split. For example, a split of bucket 2 in figure 4 yields a sparse bucket. The degenerate split leaves the bucket at level 3, (it is then sub-normal; see figure 5).

Forcing joins.

A bucket can also become sparse following a deletion. When this occurs, the bucket is joined with its brother, even if the brother is not sparse. For example, if a record is deleted from bucket 2 of figure 4, it becomes sparse. It is then joined with its brother, bucket 6, (see figure 6). (The brother can be created if it does not exist; see **Collection of brother buckets.**)

A sparse bucket may become non-sparse.

A sub-normal bucket can, due to insertions, yield higher level buckets that are both non-sparse. For example if the record deleted from bucket 2 were put back, it would be correct to distribute the records of bucket 2 in figure 6 returning to the situation of figure 4.

4.4 Algorithms

The modifications of OPLH described in section 4.3 are extensive. We now give the algorithms needed for the implementation of MLOPLH. Space limitations prevent us from stating all algorithms explicitly. A more complete exposition can be found in [Oren82].

Random accessing.

Since records are not always in the buckets they "should" be in, (e.g. due to a forced join), some mechanism is required for locating a record that has been moved. Reading an empty bucket (e.g. bucket 4 in

figure 4) is an indication that the records of the bucket have moved down at least one level and that the lower level bucket should be searched, (i.e. do the search again using a shorter prefix). The Randac algorithm, given below, searches for the first non-empty bucket by using successively shorter prefixes.

```
Randac(r)
L is the level of the bucket whose
range contains r.

if h_m(mir(r)) < n
then L := m+1
else L := m
end

while bucket h_L(mir(r)) is empty
   L := L - 1
end

return(h_L(mir(r))
end Randac
```

Note that in the worst case m+1 disk reads are necessary to locate the bucket. Let N be the number of buckets in the file. (N is proportional to the number of records in the file - see section 5.) Then, since $m=\log_2 N$, we have a problem if we want to compete with the Btree, (note that the base of the logarithm is 2). Fortunately, this problem can be avoided: the version of Randac given above may generate up to m+1 disk accesses during a "sequential search" for the length of the prefix used to locate the bucket. This search can be replaced by a binary search, reducing the cost to $o(\log_2 m) = o(\log_2 \log_2 N)$ disk accesses. The details of this modification are given in [Oren83].

Sequential accessing.

As before, Seqac must construct a successor record and perform a random access. But since there are no sparse buckets, it is not necessary to check for and skip over empty buckets. Thus the Seqac algorithm has been simplified: construct the successor record, S, and search for it using Randac.

Splitting and joining buckets.

The Split algorithm works as before except that a sub-normal bucket may be created. Similarly, Join is easily modified to deal with the joining of a sub-normal bucket and an empty bucket.

Insertion and deletion of records.

These algorithms work as before with the following exceptions:

An insertion to a sub-normal bucket may create a situation in which the bucket could be split to yield two non-sparse buckets. This split is performed by the Distribute algorithm (discussed below).

Similarly, if a deletion yields a sparse bucket, the Collect algorithm will be invoked to join the sparse bucket with its brother. (A brother can be created if it does not exist. The Collect algorithm is discussed below.)

Distribution of records in a bucket.

Distribute is called by the insertion algorithm when the records of a bucket b at level level(b) are to be distributed to brother buckets at level level(b)+1. The records are classified according to the value of the level(b)+1st bit. SplitBucket(b) performs this classification. (SplitBucket is discussed in [Oren82].)

Note that Distribute is recursive. Consider the situation of figure 7. Bucket 0 stores all the records whose prefix is 0_2 but the distribution of the records within the bucket is biased. As soon as two records with prefix 01_2 are inserted, bucket 0 can be split to yield buckets 0 and 2 (corresponding to prefixes 00_2 and 01_2 respectively). Now, assuming a uniform distribution of records within bucket 0 (at level 2), the distributions shown in figure 8 can occur, (the prefixes are shown). These distributions are performed by the recursive calls.

```
Distribute(b)
DistribOK(b) returns true iff bucket b can be
split to yield two non-sparse buckets.

if level(b) < NormalLevel(b)
then     (* distribution is possible *)
   oldlevel := level(b)
   b' := brother(b)
   SplitBucket(b)
   level(b) := oldlevel + 1
   level(b') := oldlevel + 1
   if DistribOK(b) then Distribute(b)    end
   if DistribOK(b') then Distribute(b')    end
end
return
end Distribute
```

Collection of brother buckets.

Collect is called by the deletion algorithm to combine a sparse bucket with its brother. JoinBuckets manipulates the buckets' data structures (see [Oren82]).

Collect, like Distribute, is recursive. Consider a Collect of bucket b at level L and b' = brother(b) at level L' > L. (Since level(b') > level(b), b does not really have a brother but if it did, its address would be brother(b).) Before the Collect can occur, b' must be at

level L. To ensure this, b´ is collected even though neither b´ or its brother (≠b) is sparse. (An example is given in section 5.1.)

```
Collect(b)

b´ := brother(b)

(* Bring b and b´ to the same level by *)
(* collecting the higher level bucket *)
if level(b´) < level(b)
then     (* interchange b and b´ *)
   b :=: b´
end
while level(b) < level(b´)
   Collect(b´)
end

(* Place result in the left brother, b *)
if b´ < b then b :=: b´     end
JoinBuckets(b,b´)
level(b) := level(b) - 1
return
end Collect
```

5.0 PROBLEMS WITH MLOPLH

By eliminating sparse buckets, the problem of potentially poor performance for sequential accessing has been solved. For random accessing, MLOPLH is at least as good as the Btree. It can be expected to have better performance unless the data is highly clustered.

Updates occasionally generate calls to Distribute or Collect. For some distributions of records (characterized by clusters of records) these calls can generate a lot of work; every bucket is accessed in the worst cases. In this section, the problem is explained and a solution is proposed.

5.1 The problem

Consider the situation of figure 9. A rash of deletions from bucket 0 have caused it to become sparse. It must be collected but its brother does not exist. If it did exist, it would be in bucket 1 which is currently at level 4. Bucket 1 must therefore be collected, putting it at level 3. By the time that bucket 1 reaches level 1, the contents of buckets 3, 5, 7 and 9 will have been moved to bucket 1. Half of the buckets have been affected.

A similar problem plagues Distribute. Generally, Collects and Distributes involving buckets at very low levels: 0, 1, 2, etc. involve very large fractions of the file.

The cost of moving records from one bucket to another is a minor concern: the records in brother buckets b and b´ can be merged in time O(log(n) + log(n´)) where n and n´ are the number of records in b and b´ respectively, (the algorithm is in [Oren82]). This is possible because all the records in b are smaller than those in b´ (or vice versa) and only the "edges" of the Btrees have to be modified. Splitting a Btree (required by the Distribute algorithm) costs $O(\log^2 n)$ [Oren82].

The major concern is the number of buckets involved. For example, collecting a bucket on level L may cause as many as about $N/2^L$ buckets to be accessed where N is the number of buckets, $2^m \leq N < 2^{m+1}$

5.2 The solution

Clearly, the solution involves placing a lower bound on the level of a bucket. For example, if the lowest level permitted is 5, then a bucket being collected will be at least on level 6 and no more than $1/2^6$ = 1/64th of the buckets could be involved in any Collect.

A consequence of this strategy is that some sparse buckets may exist. In general, if the lowest level permitted is L then there may be as many as 2^L sparse buckets.

A few modifications to MLOPLH are required to make this work:

- The file is initialized with 2^L empty buckets, (instead of one empty bucket).
- Sparse buckets must be kept track of. The address and level of each must be known. When a new sparse bucket is created, causing the number of sparse buckets to exceed 2^L, the sparse bucket at the highest level is collected, (this is probably the cheapest one to collect).

It is interesting that there is a tradeoff between the worst case costs of sequential accessing and updating: an update may require accessing $N/2^L$ buckets and there are up to 2^L sparse buckets which can slow down sequential processing.

6.0 PERFORMANCE

MLOPLH is a complicated but potentially faster alternative to the Btree. We can make the following qualitative statements about the performance of MLOPLH relative to that of the Btree.

- Random accessing costs one disk access if the accessed bucket has not overflowed and is not sparse. In the case of overflow, a (probably small) Btree has to be searched. In case of sparseness, the required bucket can be located in $o(\log_2 \log_2 N)$ disk accesses.

- Sequential accessing usually costs the same as for a Btree. The cost may be somewhat higher when a bucket boundary is crossed. (Links between consecutive buckets can be used to avoid the increased cost.)
- Updating will usually be comparable to the cost for the Btree but could be worse: i.e. when a Collect or Distribute accesses a fixed fraction of the buckets. This cannot happen very often.
- The worst case load factor of a Btree is 50% [Baye72]. The load factor of linear hashing is controllable but the use of Btrees will result in a lower load factor than the use of overflow chains. The important point is that the load factor of MLOPLH cannot get arbitrarily close to zero as is possible with some other trie-based methods, (e.g. the trie).

Clearly, a lot of work is necessary before MLOPLH can be recommended as the successor of the Btree. This work falls into four areas:
1) Fine tuning: selecting values for parameters (e.g. threshhold for sparseness, number of sparse buckets permitted).
2) Studying various strategies for administrative details such as when to split and when to join.
3) Performing experiments to compare MLOPLH and the Btree.
4) Finding a clever and melodious name to replace "MLOPLH". This is essential before widespread acceptance of the data structure is possible.

Acknowledgement

I am grateful to Prof. Tim Merrett for his comments on earlier drafts of this paper.

References

Baye72 R. Bayer, E. McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica 1, 3 (1972), 173-189.

Burk83 W.A. Burkhard.
Interpolation-based index maintenance.
Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, (1983), 76-85.

Fagi79 R. Fagin et al.
Extendible hashing - a fast access method for dynamic files.
ACM TODS 4, 3 (1979), 315-344.

Fred60 E.H. Fredkin.
Trie memory.
CACM 3, 9 (1960), 490-499.

Knut73 D.E. Knuth.
The Art of Computer Programming, vol. 3: Sorting and Searching.
Addison-Wesley, Mass., (1973).

Litw80 W. Litwin.
Linear hashing: A new tool for file and table addressing.
Proc. VLDB6, (1980), 212-223.

Litw81 W. Litwin.
Trie hashing.
Proc. ACM SIGMOD, (1981), 19-29.

Oren82 J.A. Orenstein.
Algorithms and data structures for the implementation of a relational database system.
Ph.D. thesis, McGill University, (1982).
Also available as Technical Report SOCS-82-17.

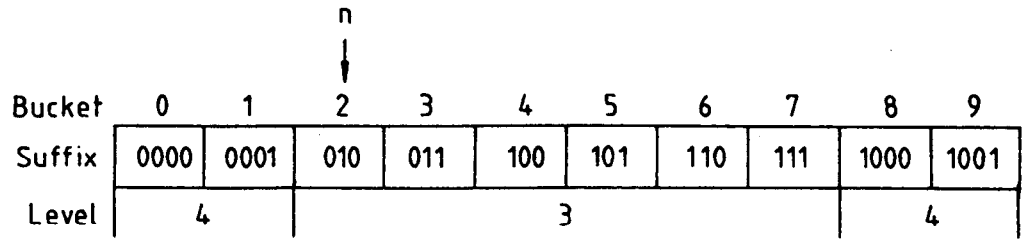
Oren83 J.A. Orenstein.
On the efficient searching of MLOPLH.
Manuscript, (1983), COINS Department, University of Massachusetts, Amherst.

Ouks83 M. Ouksel, P. Scheuermann.
Storage mappings for multidimensional linear dynamic hashing.
Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, (1983), 90-105.

Stan80 T.A. Standish.
Data Structure Techniques.
Addison-Wesley, Reading, Mass., (1980).

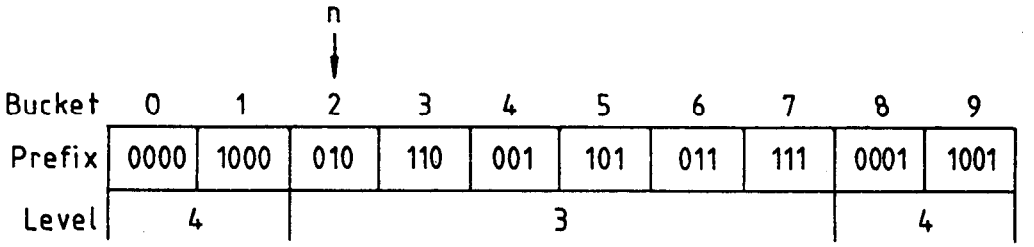
Tamm81a M. Tamminen.
Order preserving extendible hashing and bucket tries.
BIT 21, 4 (1981) 419-435.

Tamm81b M. Tamminen.
Expected performance of some cell based organization schemes.
Report HTKK-TKO-B28, (1981), Helsinkii University of Technology.



m = 3
n = 2

Figure 1. Linear hashing



m = 3
n = 2

Figure 2. Order-preserving linear hashing

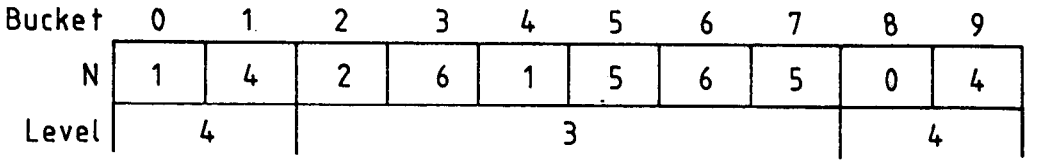


Figure 3. OPLH with sparse buckets. (A bucket is sparse if it contains 0 or 1 records.)

Bucket	0	1	2	3	4	5	6	7	8	9
N	2	4	2	6	-	5	6	5	-	4
Level	2	4	3	3	-	3	3	3	-	4
Normal level	4		3						4	

Figure 4. Multi-level OPLH: sparse buckets have been eliminated.

Bucket	0	1	2	3	4	5	6	7	8	9	10
N	2	4	2	6	-	5	6	5	-	4	-
Level	2	4	3	3	-	3	3	3	-	4	-
Normal level	4			3					4		

Figure 5. Bucket 2 has undergone a degenerate split.

Bucket	0	1	2	3	4	5	6	7	8	9
N	2	4	7	6	-	5	-	5	-	4
Level	2	4	2	3	-	3	-	3	-	4
Normal level	4		3						4	

Figure 6. Buckets 2 and 6 have been joined.

Bucket	0	1	2	3	4	5	6	7	8	9
N	24*	3	-	4	-	3	-	2	-	3
Level	1	4	-	3	-	3	-	3	-	4
Normal level	4			3					4	

* 00: 24
01: 0

Figure 7. A large cluster of records in bucket 0.

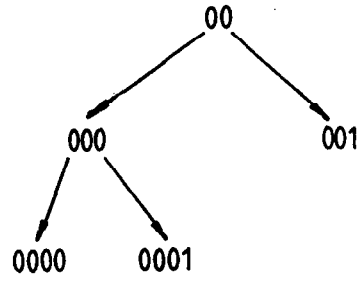


Figure 8. Distribution of the bucket 0 records.

Bucket	0	1	2	3	4	5	6	7	8	9
N	1	7	-	8	-	7	-	8	-	8
Level	1	4	-	3	-	3	-	3	-	4
Normal level	4			3					4	

Figure 9. Half of all the buckets are involved when bucket 0 is collected.