Filtered Linear Hashing

C.H. Ang, T.C. Tan
School of Computing
National University of Singapore
Republic of Singapore, 119260
E-mail: {angch, tantc}@comp.nus.edu.sg

Abstract

Filtered linear hashing is a combination of linear hashing and filtered hashing. It expands the file gracefully as linear hashing, and organizes the overflow buckets with the use of a filter buffer to ensure that any bucket can be retrieved in one disk access. At a load factor of 0.7, the cost of file creation is reduced by 15% as opposed to linear hashing, with higher cost saving for larger load factors.

1 Introduction

The volume of data to be handled by many information systems is ever increasing. In order to process the data efficiently, data collected are organized into huge disk files that can be accessed quickly with the use of appropriate index structures. The B^+-tree [4] is commonly used when the data are points from one dimensional space, while octree [16], multi-priority tree [18] and R-tree [6] may be used for higher dimensional space.

When one of these file structures is used, the number of disk accesses is proportional to the volume of data. Even if the fan out of a B^+-tree node is as large as 100, it requires at least 3 disk accesses to read in a leaf node for a file with a million items. More disk accesses are required for larger files. This motivates the search for methods that take only one disk access to retrieve an item most of the time. This means that the address of the disk block used to store the item should be determined based on the given item called the *key*. The relationship between the keys and their corresponding disk addresses defines a *hash function*. For the ease of discussion, we shall assume that the keys are numeric.

The easiest way to devise a hash function associated with a file, called the *primary file*, is to make use of modulo arithmetic. We regard a primary file as a table with T entries. The required hash function h is defined as $h(k) = k \ mod \ T$ where k is a given key, which is inserted into *bucket* (or *node*) $h(k)$. When a node is full, subsequent insertions will cause the node to overflow.

The overflow problem can be resolved by either keeping all overflowed items in an area for temporary holding, or by storing them in separate chains, one for each node that overflows. The block in the primary file is called the *home block* of the chain, and overflow blocks are grouped together in a file called the *overflow file*, or the *secondary file*. Both solutions have increasing access cost as the overflow area or the overflow chains are ever growing. The performance will deteriorate as the volume of data increases. In order to keep the response time within a certain reasonable bound, the data stored in the primary and secondary files are to be unloaded and restored into a new, bigger primary file. This is, however, a rather expensive housekeeping routine that needs to be carried out periodically and it severely affects its popularity in commercial applications.

To handle the file expansion gracefully, more sophisticated methods are needed. The general 2-prong approach is to increase the file size a little at a time and to maintain the disk block addresses separately in a directory. When overflow occurs, a new block will be appended and items affected will be redistributed whereas unrelated items in other blocks will stay put. The *grid file* [13] and the *extendible hashing* [5] are two such schemes. Both require that the directories with entries containing pointers to the relevant disk blocks be maintained.

When the directory is small enough to reside in the memory, both methods guarantee the retrieval of any item in 1 disk access. If the directory is large, 2 disk accesses are required: one for reading the directory block that contains the pointer, and one to read in the data block by following the pointer. It is clear that when the volume of data increases, the size of directory will grow in tandem. When data are clustered, it will cause excessive node splitting and the directory will be doubled quickly with many duplicated entries. This is a very serious weakness.

Linear hashing [10] and *spiral hashing* [12] can be used to get rid of the directory so that the data file will grow gracefully, but these methods have their weaknesses. Linear hashing can be improved in several ways. For example, it can be modified to expand partially so as to increase the storage utilization [9] [14], to make it order preserving [7], to optimize its performance for key-sequential access [7], by employing interpolation-based index [3], and by generalizing it to multidimensional space [8]. These variants of linear hashing use separate chaining to resolve the overflow problem. Only the *recursive linear hashing* [15] uses a predetermined number of linear hashing files to organize the overflow blocks.

On the other hand, Ilsoo Ahn [1] proposed the use of *filter buffer* to describe all the overflow blocks. Torn [17] proposed *Overflow Indexing (OVI)* to describe every overflowed key. Litwin [11] proposed *trie hashing* in which only one trie was used. All these methods rely on the additional memory structure to ensure that given a key, the required record can be retrieved in one read.

In this paper, we propose *filtered linear hashing*, a combination of linear hashing and filtered hashing. We discuss linear hashing and filtered hashing in sections 2 and 3. In section 4, the filtered linear hashing method is described. We present some empirical results for comparison in section 5 and conclude our presentation in section 6 with an outline of our future work.

2 Linear Hashing

Linear hashing is a hashing method applied to data files that grow or shrink dynamically as data are being inserted or deleted, with high space utilization and no significant deterioration in access time when the files become very big. It requires a family of hash functions $\{h_d\}$ such that most of the time two consecutive functions h_d and h_{d+1} are used. Each h_d is defined as: $h_d : k \rightarrow k \ mod \ N * 2^d$ where N is the number of buckets in the file at the beginning and d ($d >= 0$) the depth of the file.

Initially, $d = 0$, and $h_0(k) = k \ mod \ N$ is used. All incoming keys are inserted into the buckets according to the address computed. When the file is so densely populated that it exceeds a certain limit L, the *load factor*, the file is expanded by appending a block for redistribution of data at the end of the file. This will lower the storage utilization u, reduce the number of overflow blocks, and hence maintain the cost of accessing the overflow chains at a reasonably low level. For the first overflow encountered, all keys in bucket 0 will be rehashed into either bucket 0 or bucket N according to the hashing function h_1 instead of h_0. The *split pointer* sp is used to point to the block to be split and it is initialized to 0. After the block pointed to by sp is split, sp will be incremented by 1. Thus all blocks will take turn to split. When block $N-1$ is split, the size of the file will have been doubled to $2N$, sp will be reset to 0, and the depth d will be incremented by 1.

Below is the algorithm used to insert a new key into a primary file of depth d using linear hashing. Note that $u = n/t$ where n is the number of items (keys, records) inserted so far and t is the total number of slots provided. If the bucket capacity is b and the files (primary and secondary) have s buckets, then $t = b * s$.

Linear Hashing Insertion (Key k)

BucketAddress p, *p1*

StorageUtilization u

integer d

$p \leftarrow h_d(k)$

If $p \geq sp$

  then $p = h_{d+1}(k)$

Insert k into bucket p, or an overflow bucket when it is full

Compute u

If $u > L$

  then

    $p_1 \leftarrow 2^d * N + sp$
    Redistribute all keys in bucket p, its overflow buckets if any,
        and the new key k into buckets p and p_1 according to h_{d+1}
    $sp \leftarrow sp + 1$
    If $sp \geq 2^d * N$
      then $sp \leftarrow 0$; $d \leftarrow d + 1$

Figure 1(a) shows the result after the insertion of some keys, before the first node splitting occurs, assuming a bucket capacity of 3, maximum load factor of 80%, and initial hash

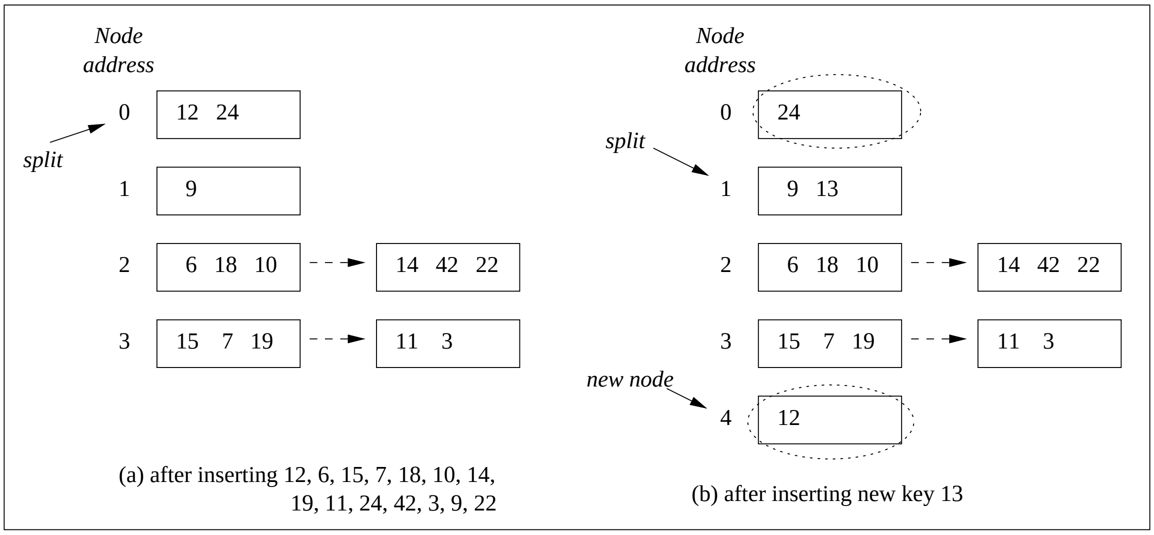

Figure 1: Linear Hashing Insertion.

function $h_0(k) = k \ mod \ 4$. Figure 1(b) illustrates node splitting and the redistribution of keys, with level-1 hash function $h_1(k) = k \ mod \ 8$.

As the roving split pointer sp gives blindly each block a chance to split, very often a block will overflow before it is split (such as node addresses 2 and 3 in Figure 1). When this occurs, a block in the overflow file will be allocated to store the overflowed keys and linked to the home block. When a chain of overflow buckets becomes very long, the time to retrieve an item in it also gets longer.

3 Filtered Hashing

Ilsoo Ahn [1] proposed a new method of hashing that can maintain the benefits of hashing even when there are many overflow records. When an overflow occurs, a bucket is appended at the end of the primary file, and all keys in the overflowed bucket will be redistributed into the overflowed bucket and the new bucket. In addition, the address of the overflow bucket is kept in a buffer that is small enough to remain in the memory. This buffer is called the *filter buffer*. With the use of a filter buffer, the cost of retrieval is guaranteed to be 1, and the cost of insertion or deletion is bounded by between 2 and 4 disk accesses.

The scheme employs only one hash function h and a family of split functions $\{s_d\}$ which are defined as follows:

$h(k) = k \ mod \ N$

$s_d(k) = (k/2^{d+1}) \ mod \ 2$

where N is the initial file size and d is the depth of an overflow chain. s_d's value is either 0 or 1 as we are only interested to know whether a key is to be placed in the original bucket (which may be an overflow bucket itself) or the new bucket just allocated. Different overflow chains may have different depths and the links leading from one to the other on the chain have to be computed through the use of the split functions as well as the filter buffer. Logically, the home blocks are in one file, the primary file, and all the overflow blocks are in a separate, overflow file. In implementation, the overflow blocks can be appended to the end of the primary file.

Initially all incoming keys are inserted into the buckets as dictated by the hash function. When the first overflow occurs for bucket p, the first block in the overflow file is used. All keys in bucket p will now be redistributed according to the split function s_1. When $s_1(k) = 0$, k is placed into bucket p. Otherwise it is put into the new overflow bucket. After the redistribution, p is stored in the filter buffer. Below is the algorithm used to insert a new key k.

Filtered Hashing Insertion (Key k)

FilterBuffer fb

BucketAddress p, p_1

StorageUtilization u

integer d'

$p \leftarrow h(k)$

$d' \leftarrow 0$

While $(p = fb[i])$

  Check off $fb[i]$
  $d' \leftarrow d' + 1$
  If $s_{d'}(k) = 1$
    then $p = N + i$

Insert k into bucket p

If bucket p overflows

then

Repeat

$d' \leftarrow d' + 1$;

Allocate an overflow bucket j

Redistribute all keys k' in bucket p and the new key k

into buckets p and $N + j$ according to $s_{d'}(k') = 0$ or 1

Add p to fb

Until both buckets do not overflow

Check on all those $fb[i]$'s that have been checked off

Figure 2(a) shows the result after the insertion of some keys, assuming a bucket capacity of 3, and hash function $h(k) = k \ mod \ 4$. A new key 29 is to be inserted into bucket 1. As bucket 1 is full, an overflow occurs. A new bucket 4 is allocated, and keys 5, 25, 17 and 29 are rehashed through $s_1(k) = (k/4) \ mod \ 2$. Since $s_1(25) = s_1(17) = 0$, and $s_1(5) = s_1(29) = 1$, keys 25 and 17 are to remain in bucket 1, while keys 5 and 29 are stored in bucket 4. The filter buffer now contains 1, the identifier of the overflowed bucket. Figure 2(b) shows the result.

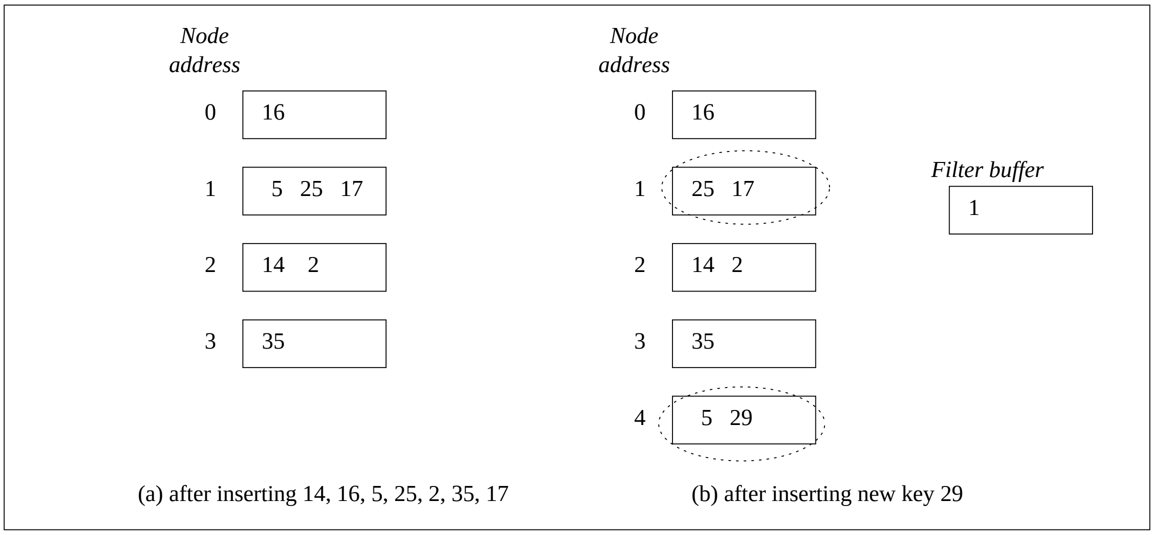

Figure 2: Filtered Hashing Insertion.

Note that a split function may not be able to divide the keys evenly. In the worst case, all keys are either being thrown into the original bucket or the overflow bucket, causing overflow again. Hence the process of allocating a new overflow bucket and redistribution of keys according to the new split function has to be repeated. This will create some empty buckets and at the same time, may cause some bucket addresses to appear in the filter buffer more than once.

When the primary file is too small, the filter buffer may become very large and more time will be needed to go through the filter. When this happens, a reorganization is needed in which all data are unloaded from the old file and reloaded into a bigger primary file.

4 Filtered Linear Hashing

Although linear hashing allows the data file to grow gracefully, the existence of some long overflow chains slows down the retrieval of items in these chains. There is no performance guarantee which states categorically that an item can be retrieved in a fixed number of disk accesses.

The weakness of linear hashing is due to the use of linked list structure to organize the overflow blocks into which all overflowed keys are dumped. As a result, the list has to be traversed sequentially and each bucket has to be inspected for a given key.

On the other hand, filtered hashing is able to organize the overflow blocks so that with the use of a filter buffer, we are allowed to skip all intermediate blocks on a chain leading to the one that contains the search key. Unfortunately, as the size of the primary file is fixed, the filter buffer grows in tandem with the increase in size of the collection of overflow buckets.

The combination of these two methods results in a hashing algorithm where the primary file expands gracefully while the filter buffer remains small in size. Whenever the primary file expands, the overflow blocks described in the filter buffer may be lifted from the overflow file back into the primary file, permitting the slots in the filter buffer to be freed for reuse.

Below is the insertion algorithm of the filtered linear hashing method. Note that integer d is the depth of the primary file and integer d' the depth of a particular chain of overflow buckets.

Filtered Linear Hashing Insertion (Key k**)**

BucketAddress p, p_1

StorageUtilization u

integer d,d'

$p \leftarrow h_d(k)$

If $p \geq sp$

  then $p = h_{d+1}(k)$

$d' \leftarrow 0$

While $(p = fb[i])$

  Check off $fb[i]$
  $d' \leftarrow d' + 1$
  If $s_{d'}(k) = 1$
    then $p = N + i$

Insert k into bucket p

If bucket p overflows

  then
    Repeat
      $d' \leftarrow d' + 1$
      Allocate an overflow bucket j
      Redistribute all keys k' in bucket p and the new key k
        into buckets p and j according to $s_{d'}(k') = 0$ or 1
      Add p to fb
    Until both buckets do not overflow

Check on all those $fb[i]$'s that have be checked off

Compute u

If $u > L$

  then
    $p_1 \leftarrow 2^d * N + sp$
    Redistribute all keys in bucket p, all of its overflow buckets,
      and the new key k into buckets p and p_1 according to h_{d+1}
    $sp \leftarrow sp + 1$
    If $sp \geq 2^d * N$
      then $sp \leftarrow 0$; $d \leftarrow d + 1$

Number of Items	Linear (a)	Filtered Linear (b)	b/a (%)
1000	2237	2176	97
2000	4556	4362	96
3000	7071	6629	94
4000	9264	8787	95
5000	11579	10922	94
6000	14095	13272	94
7000	16256	15423	95
8000	18454	17549	95
9000	20739	19644	95
10000	23181	21848	94

Table 1: Hash file creation cost. Load factor = 0.6

5 Empirical Results

Some experiments were conducted to determine how much saving in disk accesses can be achieved with filtered linear hashing as compared to the original linear hashing. In the experiments, the node capacity was set to 30, the initial hash file size was 5 buckets and the load factor limit L was set at 0.6, 0.7, 0.8, and 0.9. We created files of various sizes and measured the *total number of node accesses* required in the file creation. The statistics are listed in the following tables.

From the tables, it is evident that the number of disk accesses has been reduced by 5% for $L = 0.6$ to 63% for $L = 0.9$. This is quite a substantial improvement over the original linear hashing for file creation. When the load factor is increased, more overflow blocks are allocated and accessed. Since the accesses of the overflow blocks are now diverted to accessing the filter buffer, the reduction of the number of disk accesses to the overflow blocks is expected to be greater as the load factor goes up. This was confirmed by the outcomes of the experiments.

The use of the split functions $\{s_d\}$ effectively turns the overflow file together with the primary file into a structure created by a general bucket method of degree 2 [2], that is, each full bucket is split into 2 when it overflows. The storage utilization (or load factor) of this kind of structure is about 0.69. As a result, the load factor of filtered linear hashing is about 0.7 even when the load factor limit is set at 0.9.

Number of Items	Linear (a)	Filtered Linear (b)	b/a (%)
1000	2516	2185	87
2000	5188	4393	85
3000	7748	6649	83
4000	10516	8837	84
5000	12880	10899	85
6000	15574	12948	83
7000	18753	15560	83
8000	20920	17636	84
9000	23226	19696	85
10000	25674	21751	85

Table 2: Hash file creation cost. Load factor = 0.7

Number of Items	Linear (a)	Filtered Linear (b)	b/a (%)
1000	3142	2049	65
2000	6741	4098	61
3000	10078	6154	61
4000	13861	8179	59
5000	17021	10247	60
6000	20593	12299	60
7000	24494	14326	58
8000	28385	16370	58
9000	31551	18429	58
10000	34865	20487	59

Table 3: Hash file creation cost. Load factor = 0.8

Number of Items	Linear (a)	Filtered Linear (b)	b/a (%)
1000	4099	2047	50
2000	9746	4096	42
3000	14964	6152	41
4000	20292	10245	40
5000	25759	10245	40
6000	31365	12297	39
7000	37231	14324	38
8000	43067	16368	38
9000	49331	18427	37
10000	55109	20485	37

Table 4: Hash file creation cost. Load factor = 0.9

To have a fairer comparison on the retrieval cost, we examine the case of $L = 0.7$. The saving in file creation cost is 15% with our algorithm. As for the retrieval cost, it is always one with filtered linear hashing. Contrast this with the linear hashing method. Based on the file with 10000 keys, the average retrieval cost for linear hashing is 1.09 and the maximum is 2. If L is 0.8 (0.9), the corresponding figures are 1.51(2.62) and 3(5).

6 Conclusion

In this paper, we study the strengths and weaknesses of linear hashing and filtered hashing. Although linear hashing allows the data file to expand gradually on-line without the need to do an off-line file expansion, it fails to organize its overflow buckets for quick retrieval. On the other hand, filtered hashing imposes a structure onto the overflow buckets through a filter buffer for efficient handling, but it fails to expand the primary file on-line.

We propose to combine the two algorithms into one and call it filtered linear hashing. The new method gets the best of both worlds. It expands the file gradually as in linear hashing, and it organizes the overflow file through a filter buffer as in filtered hashing.

Some experiments have been carried out to measure the cost involved in terms of number of disk accesses to insert various number of keys using linear hashing versus filtered linear hashing. It is noted that when the file is moderately loaded at 70%, the cost in file creation is reduced by about 15% and the retrieval cost is guaranteed to be 1 whereas linear hashing requires 1.09 disk accesses on the average. The cost saving is greater when the load factor is higher.

Although filtered linear hashing performs better with smaller number of disk accesses, its loading factor is restricted to about 0.7. Our future work will focus on ways to improve its space requirement while maintaining its performance and to resolve the weaknesses of the filtered hashing as mentioned in [1].

References

[1] Ilsoo Ahn, Filtered Hashing, Lecture Notes in Computer Science, 730, 85-96.

[2] C.H. Ang and H. Samet, Approximate average storage utilization of bucket methods with arbitrary fanout, Nordic Journal of Computing 3(1996), 280-291.

[3] Walter A. Burkhard, Interpolation-based index maintenance, BIT 23 (1983), 274-294.

[4] D. Comer, The ubiquitous B-tree, Computing Surveys, 11: 121-137, 1979.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, Extendible hashing – a fast access method for dynamic files, ACM Transactions on Database Systems 4, 3(September 1979), 315-344.

[6] Antonin Guttman, R-tree: A dynamic index structure for spatial searching, Proceedings of the SIGMOD Conference, Boston, June 1984, 47-57.

[7] N. I. Hachem and P. B. Berra, Key-sequential access methods for very large files derived from linear hashing, Proceedings of the Fifth IEEE International Conference on Data Engineering, Los Angeles, February 1989, 305-312.

[8] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer, Globally order preserving multidimensional linear hashing, Proceedings of the Fourth IEEE International Conference on Data Engineering, Los Angeles, February 1988, 572-579.

[9] P.A. Larson, Linear hashing with partial expansions, Proceedings of the Sixth International Conference on Very Large Data Bases, Montreal, October 1980, 224-232.

[10] W. Litwin, Linear hashing: a new tool for file and table addressing, Proceedings of the Sixth International Conference on Very Large Data Bases, Montreal, October 1980, 212-223.

[11] W. Litwin, Trie hashing: further properties and performance, Proceedings of the International Conference on Foundation of Data Organization, May 21-24, 1985, Kyoto, Japan, 51-60.

[12] G.N.N.Martin, Spiral storage: incrementally augmentable hash addressed storage, Theory of Computation, Report No. 27, Department of Computer Science, University of Warwick, Coverntry, Great Britain, March 1979.

[13] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, ACM Transactions on Database Systems 9, 1(March 1984), 38-71.

[14] K. Ramamohanarao and J. W. Lloyd, Dynamic hashing schemes, Computer Journal 25, 4(November 1982), 478-485.

[15] K. Ramamohanarao and R. Sacks-Davis, Recursive linear hashing, ACM Transactions on Database Systems 9, 3(September 1984), 369-391.

[16] Hanan Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, 1989.

[17] Aimo A. Torn, Hashing with overflow indexing, BIT 24(1984), 317-332.

[18] Ching-Der Tung, Wen-Chi Hou, and Jiang-Hsing Chu, Multi-Priority Tree: An index structure for spatial data, Proceedings of 1994 International Computer Symposium, 1994, 1285-1290.