Tried Linear Hashing

C.H. Ang, S.T. Tan, and T.C. Tan
Department of Information Systems & Computer Science
National University of Singapore
Republic of Singapore, 119260
E-mail: {angch, tanst, tantc}@iscs.nus.edu.sg

Abstract

Tried linear hashing is a combination of linear hashing and trie hashing. It expands the file gracefully as linear hashing, and organizes each chain of overflow buckets with the use of a trie to ensure that any bucket can be retrieved in one disk access.

Keywords: Indexing, hashing.

1 Introduction

When a large volume of data is stored in a disk file, to retrieve the relevant data quickly for a given key, a hashing method which is able to compute the disk address of the record from the given key is needed. There are many hashing methods proposed in the past 2 decades.

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, and k 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 items in excess in an area for temporary holding, or they can be stored in separate chains, one for each node that overflows. The block in the primary file is called the *home block* of the chain, and those 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 limit, the data stored in the primary and secondary files are to be unloaded and restored into a new primary file of bigger size. This is a rather expensive housekeeping routine that has to be carried out periodically and thus it discourages the use of hashing method 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 not be moved. The *grid file* [12] and the *extendible hashing* [4] are two such schemes. Both schemes 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 will be required: one for reading the directory block that contains the pointer, and another to read in the data block by following the pointer. It is clear that when the volume of data is large, the size of directory will be large too. When data are clustered, it will cause excessive node splitting and the directory will be doubled very often with many entries being duplicated. This is a very serious weakness.

Linear hashing [9] and *spiral hashing* [11] can be used to get rid of the directory so that the data file will grow gracefully, but these methods have their weaknesses. Since our interest is on linear hashing and its variants, we will exclude spiral hashing from our discussion. Linear hashing can be improved in several ways. For example, it can be modified to expand partially so as to increase the storage utilization [8] [13], to make it order preserving [5], to optimize its performance for key-sequential access [5], and to generalize it to multidimensional space [6]. These variants of linear hashing use separate chaining to resolve the key collision problem. Only the *recursive linear hashing* [14] proposed by Ramamohanarao and Sacks 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 [15] proposed *Overflow Indexing (OVI)* to describe every overflowed key. Litwin [10] 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 always be retrieved in one read.

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

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 2 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 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 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
$p \leftarrow h_d(k)$
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** $sp \leftarrow 0$; $d \leftarrow d + 1$

As the roving split pointer sp gives blindly each block a chance to split, very often a block will overflow before it is split. 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 on it also gets longer.

3 Trie Hashing

Trie hashing is one of the fastest access methods for dynamic and ordered files. Its efficiency lies in the use of a trie. It starts out with a bucket in which all keys will be stored. When an overflow occurs, another bucket will be appended at the end of the primary file. All keys will then be redistributed into the overflow bucket and the new bucket just allocated by comparing the value of the first character of each key with a discriminator, which is a suitable value that will usually divide the keys evenly. A key having the first character smaller than or equal to the discriminator will go into the original bucket, otherwise it will go into the new bucket. No secondary file is needed.

The result of splitting the buckets is described in a trie with the discriminator and its associated position within the key stored in each internal node, and the bucket addresses stored in the leaf nodes.

When the keys are numbers, a bit is used for comparison instead of using the whole character. As a result, the discriminators are not required to be stored in the internal nodes. During the search, each bit of the given key will be examined. If it is zero, proceed to the left subtree, otherwise go to the right subtree. This is the *digital searching* [7].

We may describe the bit checking by a family of functions $\{s_d\}$ where $s_d(k) = (k/2^d)\ mod\ 2$, d is the depth of the node in which s_d is being used. Below is the algorithm used to insert a new key k.

Trie Hashing Insertion (Key k)

```
TrieNode p
integer d
p ← the root of the trie
d ← 0
While (p is an internal node)
        If s_d(k) = 0
        then p ← p.left
        else p ← p.right
        d ← d + 1
Read in p.bucket
If (p.bucket is not full) insert k
else   Allocate one more bucket
       Perform bucket splitting and update the trie
```

It is possible that after redistribution, all keys go into the same bucket and overflow again. This may result in multiple empty buckets being allocated and the depth of the trie will be increased by more than one. If the keys are uniformly distributed, these empty buckets will be filled subsequently.

4 Tried Linear Hashing

Although linear hashing allows the data file to grow gracefully, the existence of some long overflow chains will make the retrieval of items in these chains slow. On the other hand, although trie hashing allows us to access the overflow blocks through a trie, the number of nodes to visit is large when many keys are stored in the file.

Intuitively, the combination of these two methods should give us a hashing algorithm in which the primary file can expand gracefully while the tries, used to describe the chains of overflow buckets, one trie for each home bucket with overflow buckets, are usually very small and require the visit of one or two nodes most of the time. Whenever the primary file expands, the overflow blocks described in the trie affected may be lifted from the overflow file back into the primary file, and hence the height of the tries will not grow without bound.

What is needed in maintaining many tries is an array $trie[\,]$ of pointers to the roots of the tries. This array should be large enough for the primary file used. Below is the insertion algorithm of the tried linear hashing method.

Tried Linear Hashing Insertion (Key k)

BucketAddress p, p_1
StorageUtilization u
TriePointer $trie[\,]$
integer d
$p \leftarrow h_d(k)$
If $p \geq sp$
then $p = h_{d+1}(k)$
If $(!trie[p])$
then
    Insert k into bucket p
    Handle bucket overflow and the creation of trie accordingly.
else
    Follow the trie structure $trie[p]$
    Insert k into the bucket found
    Handle bucket overflow and the upating of trie accordingly.

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 $sp \leftarrow 0$; $d \leftarrow d + 1$

5 Empirical Results

Several experiments have been conducted to find out how much saving in disk accesses can be achieved when tried linear hashing is used as compared to the original linear hashing. In the experiments, the node capacity is set to 30, the initial hash file size is 5 buckets and the load factor limit L is set at 0.6, 0.7, 0.8, and 0.9. We created several files containing different number of items and measure the total number of node accesses during file creation. The statistics are listed in the tables on the next two pages.

From the tables, it is noticed that the number of disk accesses has been reduced by 5% for $L = 0.6$ to 62% for $L = 0.9$. This is quite a substantial improvement over the original linear hashing during file creation. When the load factor is increased, more overflow blocks are allocated and accessed. Since the accesses of the overflow blocks have been diverted to the associated tries in memory, the reduction in the number of accesses to the overflow blocks is expected to be greater as the load factor increases. This is confirmed by the outcomes of the experiments.

The use of the split functions $\{s_d\}$ effectively turns the files into a structure created by a general bucket method of degree 2 [3], in which each full bucket is split into 2 when it overflows. The storage utilization of this kind of structure is about 0.69. As a result, the load factor of tried linear hashing is about 0.7 even when the load factor limit is set at 0.9.

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

Note that when $L = 0.7$, the cost in retrieving a key from a file created through linear hashing or tried linear hashing are comparable when the keys are uniformly distributed. When the distribution of the keys are skewed or clustered, there will be many long chains of overflow buckets. The performance of tried linear hashing remains unchanged whereas that of linear hashing is greatly affected.

Number of Items	Linear (a)	Tried 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

Number of Items	Linear (a)	Trie 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)	Trie 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)	Tried 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

6 Conclusion

In this paper, we study linear hashing and trie hashing. Although linear hashing allows the data file to expand gradually online without the need to do an offline file expansion, it fails to organize its overflow buckets for quick retrieval. On the other hand, trie hashing imposes a trie structure onto the overflow buckets for efficient handling, but it requires many nodes of the trie to be visited.

We propose to combine the two algorithms into one and call it tried linear hashing. The new hashing method has the best of both worlds. It expands the file gradually as in linear hashing, and it organizes the overflow file through many small tries, one for each chain of overflow buckets as in trie hashing. In fact, the original trie hashing is a special case of our tried linear hashing in which only one bucket is used in the primary file and only one trie is used for the overflow buckets.

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

In another paper [2], we describe a new hashing method *Filtered linear hashing* in which linear hashing is being combined with filtered hashing. Both tried linear hashing and filtered linear hashing are equally efficient in terms of the file creation and key retrieval cost. In view of the complexity of merging the empty buckets when substantial deletions are performed in filtered linear hashing, the use of tried linear hashing is preferred.

References

[1] Ilsoo Ahn: Filtered Hashing. LNCS 730, 85-96.

[2] C. Ang, S. Tan, and T. Tan, Filtered linear hashing, in preparation.

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

[4] 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.

[5] 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.

[6] 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.

[7] D. E. Knuth, The art of computer programming, Vol. 3, Addison-Wesley, 1973.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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