Ideal Hash Trees

Phil Bagwell

Hash Trees with nearly ideal characteristics are described. These Hash Trees require no initial root hash table yet are faster and use significantly less space than chained or double hash trees. Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches. Array Mapped Tries(AMT), first described in Fast and Space Efficient Trie Searches, Bagwell [2000], form the underlying data structure. The concept is then applied to external disk or distributed storage to obtain an algorithm that achieves single access searches, close to single access inserts and greater than 80 percent disk block load factors. Comparisons are made with Linear Hashing, Litwin, Neimat, and Schneider [1993] and B-Trees, R.Bayer and E.M.McCreight [1972]. In addition two further applications of AMTs are briefly described, namely, Class/Selector dispatch tables and IP Routing tables. Each of the algorithms has a performance and space usage that is comparable to contemporary implementations but simpler.

Categories and Subject Descriptors: H.4.m [**Information Systems**]: Miscellaneous

General Terms: Hashing,Hash Tables,Row Displacement,Searching,Database,Routing,Routers

1. INTRODUCTION

The Hash Array Mapped Trie (HAMT) is based on the simple notion of hashing a key and storing the key in a trie based on this hash value. The AMT is used to implement the required structure efficiently. The Array Mapped Trie (AMT) is a versatile data structure and yields attractive alternative to contemporary algorithms in many applications. Here I describe how it is used to develop Hash Trees with near ideal characteristics that avoid the traditional problem, setting the size of the initial root hash table or incurring the high cost of dynamic resizing to achieve an acceptable performance. Tries were first developed by Fredkin [1960] recently implemented elegantly by Bentley and Sedgewick [1997] as the Ternary Search Trees(TST), and by Nilsson and Tikkanen [1998] as Level Path Compressed(LPC) tries. AMT performs 3-4 times faster than TST using 60 percent less space and are faster than LPC tries.

During a search bits are progressively used from the hash to traverse the trie until a key/value pair is found. During insert the AMT levels are extended using more hash bits until a new hash is differentiated from previously stored ones. It will be shown that the methods for Insert, Search and Delete are fast and independent of key set size. All these may be realized yet still guarantee that worst case operation times are no more than a few times the average case.

Linear Hash (LH), Litwin, Neimat, and Schneider [1993] derived from Dynamic Hash Tables, Larson [1988] offer an effective solution to collision management and

Address: Es Grands Champs, 1195-Dully, Switzerland

external storage growth by maintaining blocks with an acceptable load factor and splitting on collisions. The underlying principle of HAMT, partitioning using a hash, has been developed and combined with the conceptual foundations of LH to give an algorithm that has better performance. The new Partition Hashing(PH) algorithm splits buckets evenly by the number of records and uses adjacent bucket sharing to improve load factor. A hash partition value is maintained for each bucket that guides inserts and searches to the correct bucket. This results in a single access search, inserts costing less than 1.1 accesses for a load factor exceeding 80 percent. If load factor is more critical than insert times for a given application the algorithm can be optimized to give a load factor up to 100 percent with increased accesses per insert.

Adapting the algorithm to distributed processing is discussed briefly. The conceptual framework being adapted from LH*.

Further examples of the utility of AMT are covered in the section giving brief overview of their application to Class/Selector dispatch tables, and IP Routing tables.

First I start with a brief summary of the AMT data structure before covering the HAMT and PH in more detail. Performance comparisons are included by section.

A more detailed description of the benchmark method can be found at the end of the paper.

2. ESSENTIALS OF THE ARRAY MAPPED TRIE

It should be noted that all the algorithms that follow have been optimized for a 32 bit architecture and hence the AMT implementation has a natural cardinality of 32. However it is a trivial matter to adapt the basic AMT to a 64 bit architecture. AMT's for other alphabet cardinalities are covered in the paper, Fast and Space Efficient Trie Searches Bagwell [2000].

A trie is represented by a node and number of arcs leading to sub-tries and each arc represents a member of an alphabet of possible alternatives. Here, to match the natural structure of a 32 bit system architecture, the alphabet is restricted to a cardinality 32 limiting the arcs to the range 0 to 31. The central dilemma in representing such tries is to strike an acceptable balance between the desire for fast traversal speed and minimizing the space lost for empty arcs.

The AMT data structure uses just two 32 bit words per node for a good compromise, achieving fast traversal at a cost of only one bit per empty arc. An integer bit map is used to represent the existence of each of the 32 possible arcs and an associated table contains pointers to the appropriate sub-tries or terminal nodes. A one bit in the bit map represents a valid arc, while a zero an empty arc. The pointers in the table are kept in sorted order and correspond to the order of each one bit in the bit map. The tree is depicted in Fig 1. Note that map entries correspond to the non-empty table entries of the next level down sub-trie.

Finding the arc for a symbol s, requires finding its corresponding bit in the bit map and then counting the one bits below it in the map to compute an index into the ordered sub-trie. Today a CTPOP (Count Population) instruction is available on most modern computer architectures including the Intel Itanium, Compaq Alpha, Motorola Power PC, Sun UltraSparc and Cray, or can be emulated with non-memory referencing shift/add instructions, to count selected bits in a bit-map.

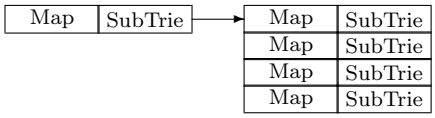

Fig. 1. Array Mapped Trie

```
const unsigned int SK5=0x55555555,SK3=0x33333333;
const unsigned int SKF0=0xF0F0F0F,SKFF=0xFF00FF;

         int CTPop(int Map)
           {
           Map-=((Map>>1)&SK5);
           Map=(Map&SK3)+((Map>>2)&SK3);
           Map=(Map&SKF0)+((Map>>4)&SKF0);
           Map+=Map>>8;
           return (Map+(Map>>16))&0x3F;
           }
```

Fig. 2. Emulation of CTPOP

3. IDEAL HASHING

Hashing is well known and the basic characteristics are widely documented. Knuth [1998] and Sedgewick [1998]. The central difficulty remains the initial sizing of the root hash table and subsequent resizing as the key set grows to obtain optimum performance. The larger the root hash table the less time is spent in collision resolution. HAMT assumes that the hash table is infinite in size and uses an AMT to represent the sparse array. At key insertion time a hash function is used to generate an array index prefix. The most significant bits of the hash are considered to represent the most significant bits of the array index. Using the AMT structure the hash is consumed each time a sub-trie is chosen until a leaf node is found with a key. If different, a new key is added by removing the existing one, calculating its hash and adding sub-tries until the two keys no longer collide. Additional hash bits are generated as needed.

The realization of this approach is illustrated by the basic HAMT data structure that is depicted in Fig 3. It comprises a root hash table of size 2^t, where t typically starts as 5, with each entry being the root node for a tree of sub-hash tables with size 32. Each entry in the hash table is either a key/value pair or an AMT pointer and bit map.

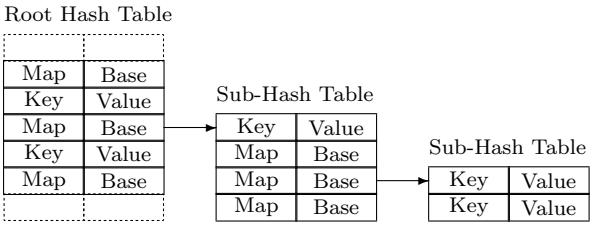

Fig. 3. Hash Array Mapped Trie (HAMT)

3.1 Search for a key

Compute a full 32 bit hash for the key, take the most significant t bits and use them as an integer to index into the root hash table. One of three cases may be encountered. First, the entry is empty indicating that the key is not in the hash tree. Second the entry is a Key/Value pair and the key either matches the desired key indicating success or not, indicating failure. Third, the entry has a 32 bit map sub-hash table and a sub-trie pointer, Base, that points to an ordered list of the non-empty sub-hash table entries.

Take the next 5 bits of the hash and use them as an integer to index into the bit Map. If this bit is a zero the hash table entry is empty indicating failure, otherwise, it's a one, so count the one bits below it using CTPOP and use the result as the index into the non-empty entry list at Base. This process is repeated taking five more bits of the hash each time until a terminating key/value pair is found or the search fails. Typically, only a few iterations are required and it is important to note that the key is only compared once and that is with the terminating node key. This contributes significantly to the speed of the search since many memory accesses are avoided. Notice too that misses are detected early and rarely require a key comparison.

Unlike Dynamic Perfect Hashing Dietzfelbinger, Karlin, Mehlhorn, auf der Heide, Rohnert, and Tarjan [1994]no effort is taken to minimize the size of the sub-hash tables. Instead the AMT data structure has been used to minimize the cost of empty table entries. As the tree grows there is a small but increasing probability that the number of iterations will cause the 32 bits in the hash to be exhausted and a new one must be computed. This is covered further in the following description of the insertion algorithm.

Assuming that the hash function generates a random distribution of keys then on average the key hash will uniquely define a terminal node after lgN bits. With an AMT 5 bits of the hash are taken at each iteration giving a search cost of $\frac{1}{5}lgN$ or $O(lgN)$. As will be shown later this can be reduced to an $O(1)$ cost. Table 1 shows search times for HAMTC with $O(1)$ cost, HAMTL with $O(lgN)$ and then a chained hash tree with a root hash table of 2000, 64000 and 512000 entries. (The benchmark system memory capacity limited the Hash tests to 4096000 keys.)

A small, but important implementation consideration is the identification of sub-tries or key/value nodes by a bit flag in the data structure. Using C++ the bit uses the least significant bit of the pointer and encapsulates this within the pointer

access methods. In Java the sign of an array index could be used to achieve the same result. Since the node is kept to 8 bytes on many 32 bit architectures both words are loaded into the cache on reference and the cost of a second memory access avoided. Note that the performance of the algorithm is seriously impacted by the poor execution speed of the CTPOP emulation in Java, a problem the Java designers may wish to address.

Table 1. Comparative Search performance

SetSize	HAMTC	HAMTL	Hash2K	Hash64K	Hash512K
8K	0.82	0.75	1.00	1.00	1.00
16K	0.81	0.88	1.44	0.94	1.00
32K	0.93	1.00	2.53	1.03	0.94
64K	1.06	1.13	4.66	1.19	0.97
128K	1.29	1.42	9.07	1.63	1.16
256K	1.06	1.29	19.27	1.70	1.18
512K	1.16	1.34	37.90	2.13	1.49
1024K	1.26	1.41	77.32	3.27	1.46
2048K	1.37	1.52	165.00	5.83	2.04
4096K	1.45	1.63	347.00	11.28	2.49
8192K	1.35	1.73	–	–	–

3.2 Insertion

The initial steps required to add a new key to the hash tree are identical to the search. The search algorithm is followed until one of two failure modes is encountered.

Either an empty position is discovered in the hash table or a sub-hash table is found. In this case, if this is in the root hash table, the new key/value pair is simply substituted for the empty position. However, if in a sub-hash table then a new bit must be added to the bit map and the sub-hash table increased by one in size. A new sub-hash table must be allocated, the existing sub-table copied to it, the new key/value entry added in sub-hash sorted order and the old hash table made free.

Or the key will collide with an existing one. In which case the existing key must be replaced with a sub-hash table and the next 5 bit hash of the existing key computed. If there is still a collision then this process is repeated until no collision occurs. The existing key is then inserted in the new sub-hash table and the new key added. Each time 5 more bits of the hash are used the probability of a collision reduces by a factor of $\frac{1}{32}$. Occasionally an entire 32 bit hash may be consumed and a new one must be computed to differentiate the two keys. Table 2 shows Insert times corresponding to search times in Table 1.

3.3 Memory Allocation

As Nilsson and Tikkanen [1998] remark, performance critically depends on the memory allocation system and careful organization is rewarded. Since sub-hash tables range from 1 to 32 entries demand for new hash tables becomes progressively distributed across the full 32 range of sizes. This distribution is exploited by keeping an array of 32 linked lists of free sub-hash tables. The first entry in the array

Table 2. Comparative Insert performance

SetSize	HAMTC	HAMTL	Hash2	Hash64	Hash512
8K	3.55	3.56	3.00	3.38	2.75
16K	3.58	3.54	3.44	2.75	2.81
32K	3.83	3.78	4.56	3.16	2.97
64K	4.06	4.01	6.80	3.27	2.86
128K	3.92	3.90	11.42	3.38	2.95
256K	4.10	3.93	21.08	3.99	3.31
512K	4.31	4.18	40.96	4.44	3.62
1024K	4.72	4.63	80.55	5.46	3.44
2048K	4.94	4.95	175.00	7.83	3.82
4096K	4.87	5.00	343.00	13.31	4.51
8192K	5.22	5.37	–	–	–

contains the list of all free sub-hash tables of length one, the second the list of all sub-hash tables of length two and so on. Thus allocating a new sub-hash table requires the minimum of work, either remove one from the appropriate free list or allocate a new one from free memory pool. The memory pool is allocated from system memory in blocks. A freed sub-hash table is simply attached to the appropriate free list.

Clearly the distribution is not quite perfectly balanced, as the key set grows more small sub-hash trees are freed than are required later. Unchecked this leads to a set of free lists that are typically 3 times the used space, however with a little ingenuity memory pool blocks can be de-fragmented to recover the lost free space. A de-fragmentation threshold is set as a percentage of waste free memory allowed. If this is exceeded a memory pool block will be de-fragmented during the next key insertion. With this modification, space wasted is restricted to a few percent with minor impact on insertion times. Table 3 shows total node usage, including free and the amount free, but excluding key string storage, with a threshold of 15 percent. The variance in node consumption is also small when compared across a large number of key sets.

The pool block size partially determines the time for insert or removal. In general the larger the pool blocks the better the amortized insertion time however blocks beyond 2000 nodes give marginal improvement. When a pool block is de-fragmented all the allocated sub-hash tables must be moved. Although not a costly activity it could represent many insert times in the worst case. However, the de-fragmentation can be carried out progressively, moving only one sub-hash table at each insert until the complete block is de-fragmented. With this modification the worst case becomes a few average insert times.

A pessimistic average de-fragmentation cost can be shown to be $O(1)$. Suppose the free space ratio is set to a fraction f of the total space used S and a further n keys are added. Note f=0.15 implies that 15% space is reserved in the linked free lists.

When a key is added a sub-table is freed with e entries, placed on the appropriate free list and a new one is required with $e + 1$ entries. Hence after n inserts n sub-tables are freed and n new ones required. In order to keep the free space a constant the de-fragmentation must recover worst case $n(1 - f)$ trie nodes if they are not available on free lists in the required size.

On average each memory block contains a fraction $1 - f$ sub-tables and f free ones.

In order to recover space $n(1-f)$ then $n(1-f)/f$ space must be de-fragmented. Each de-fragmentation requires $1 - f$ sub-tables to be reallocated. Hence de-fragmentation cost of n inserts become $\frac{n(1-f)}{f}(1-f)$ or $\frac{n(1-f)^2}{f}$. Therefore average pessimistic cost per insert is $\frac{(1-f)^2}{f}$ or $O(1)$. The average cost is significantly better than this, free lists often contain the required memory, becoming progressively more likely as the free pool grows with N.

Table 3. HAMTC Space Utilization

SetSize	**Total Nodes**	Free Nodes
8K	12K	2K
16K	24K	4K
32K	49K	7K
64K	101K	15K
128K	190K	27K
256K	365K	49K
512K	729K	105K
1024K	1569K	231K
2048K	3259K	484K
4096K	6261K	933K
8192K	11799K	1610K

3.4 Lazy Root Hash Table Re-Sizing

As can be seen from the performance characterization, search and, to a lesser extent, insert times improve with larger root hash tables. At some point as the hash tree grows it would be an advantage to resize the root hash table. This proves to be a low cost activity that can be performed lazily to guarantee worst case insert and search times.

Each Sub-hash tree represents 32 entries, which may be filled or empty. If the root tree is resized to $2^{(t+5)}$ each of the sub-hash table entries has a corresponding place in the new root hash table. The sub-hash table non-empty entries need to be copied into the new root hash table and the sub-hash table made free. Unlike other schemes, notice that rehashing is not required. The probability of the root hash table containing a key/value pair, which would require re-hashing to locate it correctly, is close to zero, however a practical algorithm must still consider the case.

The entire root hash table can be processed at once or each sub-hash table migrated to the new root table only when needed. First a new root table is created with all the entries set to empty, a process that can be amortized over several insertions. A resizing index is maintained that is moved progressively through the original root hash table. A modified search algorithm compares the first t bits of the hash with the resize index. If less then the search uses the old hash table otherwise the search starts with the new root hash table taking $t + 5$ bits from the hash to form the index and the search then continues normally. The insert algorithm is similarly modified.

The hash table should be resized when the new hash table represents a reasonable fraction, say, $\frac{1}{f}$ of the total allocated space. This means that the first $\frac{1}{5}lg\frac{N}{f}$ accesses will be replaced by a single access in the hash table. Hence the average search and insert costs become $\frac{1}{5}lgN - \frac{1}{5}lg(\frac{N}{f})$ or $\frac{1}{5}(lgf)$, i.e. $O(1)$. Note, table resizing will only occur when the new table is 32 times the size of the old one or when the original table is $\frac{1}{32f}$ in size. It is sufficient therefore to move entries from the old to the new root table once every, say, $\frac{32f}{2}$ inserts to complete this process before the next resize could be required. Since one root table entry move is equivalent to a few average insert times this becomes the worse case insert time. All the HAMTC times in Table 1 and 2 result from using this lazy resizing algorithm.

If the final size of the HAMT can be estimated in advance then the root hash table can be given an initial size to match this. Performance improves accordingly. See Table 4 for a comparison of HAMT with initial root tables of 16000, 128000, and 512000 entries and then lazy resizing on additional growth.

Table 4. HAMTC with Initial Hash Table - Search

SetSize	16K Table	128K Table	512K Table
8K	0.25	0.50	0.63
16K	0.81	1.06	0.63
32K	0.72	0.66	0.69
64K	0.81	0.73	0.91
128K	1.21	0.88	0.80
256K	0.97	0.99	0.97
512K	1.16	1.09	1.27
1024K	1.32	1.19	1.28
2048K	1.40	1.34	1.30
4096K	1.44	1.32	1.25
8192K	1.51	1.36	1.27

3.5 Space Used

With root hash table resizing the algorithm requires space that is a constant multiple of N. Suppose the table has just been resized and the resize factor has been set to 4. At this point the root hash table is close to $\frac{N}{4}$ and each location has an average of 4 keys which hash to that index. Then each of the cases of 0 through to n are considered assuming a Poisson distribution. Accordingly it is found that on average 1.8% of the locations will be empty 7.3% of the locations will have a key/value pair and the rest will have sub-trie entries with two or more keys. Then taking each of the cases 2 to n the average number of sub-tries which contain further sub-tries can be calculated in a similar way. When $n > 5$ the contribution is negligible. As a result it is found that on average 18.8% of the locations will have sub-tries with a further sub-trie. Hence the space required becomes the total number of key/value pairs N, plus the number of sub-trie links in the tree, less the number of key/value pairs in the root hash table or $N + 0.25N(1 + 0.188 - 0.073)$ or approximately $1.28N$. Changing the resize factor will cause this constant to change. Table 5 demonstrates this relation showing both the theoretical value and a corresponding measured empirical value. Note that the search time constant decreases

as the space constant increases. In practice this allows the algorithm to be tuned dynamically to give best performance for small N. Then as N becomes large the resize factor and free space can be adjusted to give efficient space use for a slightly lower performance.

Table 5. Factor and Space Use

Factor	**Theory**	**Empirical**
1	1.65	1.63
2	1.40	1.39
4	1.28	1.28
8	1.14	1.11

3.6 Key Removal

Key removal presents few complications and two cases need be considered. If the sub-hash tree contains more than two entries then the key entry is removed and marked empty. This requires that a new smaller sub-hash table be allocated and the old one made free. If the sub-hash table contains two entries then the remaining entry is moved to the parent sub-hash table and the current sub-hash table made free.

A threshold may be set on free memory pool and if exceeded then the allocated sub-hash tables in a pool block are moved into free space in other pool blocks. The free entries are removed from the free lists and the pool block returned to the system.

3.7 Hash Function

Although not absolutely critical the selection of a good hash function does significantly impact the HAMT overall performance. The Universal Hash Sedgewick [1998], p593, proved to be the best when compared with Elf Hash and PJW Hash, Table 6. Both the HAMT and the chained Hash tables used the Universal hash.

The hash function was tailored to give a 32 bit hash. The algorithm requires that the hash can be extended to an arbitrary number of bits. This was accomplished by rehashing the key combined with an integer representing the trie level, zero being the root. Hence if two keys do give the same initial hash then the rehash has a probability of 1 in 2^{32} of a further collision. With a good hash function, the hash should be unique after, on average, lgN bits so it is rare in practice for the rehash to be needed.

3.8 Guaranteed Worst Case Insertion, Search and Removal Times

Some real time applications require that all operation times be bounded. With one exception all of the operations described so far are bounded.

During the insert operation the hash function is relied on to produce key differentiation after a few sub-hash tables. There is a high probability that this will be so but there remains a remote possibility that it will prove to be a fatally lengthy operation. Happily this potential Achilles heel can be avoided by allowing rehashing to continue to some limit and then the whole key is utilized in place of the

```
unsigned int HashKey(char *key)
  {
  unsigned int a=31415,b=27183;
  for(unsigned int vHash=0;*key;key++,a*=b)
    vHash=(a*vHash+*key);
  return vHash;
  }

unsigned int ReHashKey(char *key,int Level)
  {
  unsigned int a=31415,b=27183;
  for(unsigned int vHash=0;*key;key++,a*=b)
    vHash=(a*vHash*Level+*key);
  return vHash;
  }
```

Fig. 4. Universal Hash and Rehash with Level

Table 6. Hash Function Comparison - Search

SetSize	Univ	Elf	PJW
8K	0.75	2.00	0.88
16K	0.75	2.06	1.25
32K	0.75	2.03	1.03
64K	0.88	2.09	1.17
128K	1.05	2.87	1.25
256K	1.24	2.43	1.35
512K	1.34	2.64	1.53
1024K	1.06	2.39	1.87
2048K	1.16	2.49	2.10
4096K	1.20	2.55	2.14
8192K	1.29	2.65	2.28

hash. This guarantees that the keys will be differentiated and in a worst case time proportional to the key length.

Hence the algorithm is shown to be time bounded for the principal operations of insertion, search and delete.

3.9 Conclusion

It seems reasonable to conclude that HAMT is indeed a Hash Tree with near ideal characteristics achieving small O(1) costs for principle operations and having small bounded worst case times. They are considerably simpler to code than LPC trees and Dynamic Perfect Hashing yet are space efficient too. The fundamental data structure is small and, as illustrated in sections 5,6 and 7, can be easily adapted to suit other applications.

4. PARTITION HASHING EXTERNAL STORAGE

The essential problem in external storage retrieval systems is to be able to find records with the minimum of costly disk accesses while maintaining high disk space

utilization. The ideal would be 1 access for search (success/failure) or insertion and 100 percent disk space usage for data records. External Hashing, Fagin, Nievergelt, Pippenger, and Strong [1979] , B-Trees, R.Bayer and E.M.McCreight [1972] and more recently Linear Hashing, Litwin, Neimat, and Schneider [1993],[Litwin and Schwarz 2000], have striven to do this, Linear Hashing coming close to the ideal. Partition Hashing comes closest of all, achieving 1 access for search (success/failure), 1.1 access for insertion at 80 percent load factor or 2 accesses for load factor approaching 90 percent.

In Linear Hashing (LH) a $h(C) = C \ mod \ N*2^i$ hash function is used to distribute records between buckets and hence control the subsequent splitting required on data set growth. When a bucket needs to be split i is is set to $i + 1$ and the records distributed into the new and old bucket accordingly. The problematical issue of low load factor is offset by using chained overflow buckets to delay splitting at the cost of very few access times to achieve an impressive performance.

In Partition Hashing(PH) a bucket is split into two buckets by moving half of the records from the old bucket to the new one. To do this a list of the records is created in their key hash order. The mid-point of this list, called the Hash Partition Point, has the hash value that divides the number of records equally in two. The records in the top half are moved to the new bucket, those in the bottom stay in the existing one. The Hash Partition value then is kept in a Hash Partition table associating this value with the bucket address. If buckets are non-contiguous then one entry is needed per Bucket. Note that the precision of this partition value only needs to be about $5 + lgN$ bits, where N is the maximum key set size, to satisfactorily maintain split accuracy.

Partition Hash Table entries may be reduced to the bucket address plus a few bits by eliminating the redundancy in the leading bits of successive partition hash values, Knuth [1998]page 512 and using a single bit indicating free space.

4.1 Search

To find a given record, first compute the keys' hash. Then find the entry in the Hash Partition table where the key hash lies between its Hash Partition value and the one in the entry above. The desired record is in bucket with the corresponding address.

Finding the entry in the Hash Partition table is simpler than it first appears. Because the keys are well distributed, the initial entry e can be found by $e = \frac{h}{h_{max}}b$ and then making a linear search to find the correct record. Here h is the key hash, h_{max} is the maximum hash value and b the number of bucket entries in the table. Simulation shows the first entry is typically the correct one while in fewer than 10 percent of the cases the search continues to an adjacent entry. Note this property can be used to remove the requirement for the Hash Partition Table if the buckets are contiguous and more accesses can be tolerated on search or insert operations. Using a small table of correction offsets improves the hit rate.

4.2 Insert

First a search is performed to find the correct bucket as described in the previous section. If there is space the new record can be added. However, if the bucket is full additional space must be found.

Rather than just splitting the bucket on overflow, first a check is made to ascertain if the adjacent Hash Partition table entries have buckets with available space to share. Then the one with most space is shared with the full bucket by redistributing the records over the two buckets and updating the Hash Partition value in the table. The process is identical to the splitting described previously. First a list of the records for the two buckets is created in key hash order. The Hash Partition Point is defined and then records are moved from the full bucket to the less full one based on this hash partition value. This simple device delays the need to split and increases the overall load factor. Only if sharing fails to find space is a split finally performed. Simulation shows that about 10 percent of the inserts will need a share or a split and the load factor stabilizes at just over 80 percent independent of bucket size. Though using larger buckets extends the number of inserts to stabilization.

At the cost of additional insert assesses load factors approaching 100 percent are practical. To achieve this the sharing is extended beyond the immediate neighbors when a full bucket is encountered. The Partition Hash Table is search for up to t places above and below the full bucket to find space available in a bucket perhaps a distance s from it. Bucket s is then shared with bucket $s-1$, then $s-1$ with $s-2$, and so on, always moving at least one record, until a space for at least one record is finally created in the full bucket. The sharing process will tend to balance the load factor in the s buckets manipulated in this way.

Table 7 shows the load factors achieved while varying the value of r and the bucket size. The average number of insert accesses that correspond to these load factors are shown too. By setting the one parameter t the load factor of the file may be set to achieve a desired load factor and insert cost relationship. Alternatively a maximum size available for buckets could be set and t increased as the limit is approached. In this way while the file is small fastest insert times are achieved but as the file limit approached insert times progressively degrade until the file is finally full but with a load factor close to 100 percent. These additional insert accesses does not change the search cost, remaining constant at one access.

Note that duplicate keys represent a small complication. A single bit flag must be added to the Partition Hash Table entry to indicate if the partition Point is a duplicate key. Then, when adding a new record with a duplicate key it can be correctly positioned at the end of the existing duplicate records. While searching for a record the search can correctly be terminated at the first of the duplicate entries. During shares order is preserved so correct sequencing of records is maintained in the file.

4.3 Bucket Searches

It has been assumed that the records are stored in a bucket with no particular order. This assumption follows from the fact that a linear search after a bucket has been loaded from disk still takes a small fraction of the load time. However, buckets can become large enough that significant processor time is lost in this linear search or the buckets are RAM resident. Other methods are then appropriate. For example, records can be placed using linear probe hashing or, for a small space overhead, a list, sorted by hash value, of record pointers can be added to the beginning of a bucket. Note pointers in this structure need only be $lg\ r$ bits long where r is the number of records in the bucket, giving a small overhead of $int((lg\ r)+1)$ bits

Table 7. Average Insert Accesses and Load Factor %

Range(t)	$r = 2$	$r = 5$	$r = 10$	$r = 20$	$r = 50$	$r = 100$	$r = 500$
1	1.00/74	1.22/80	1.24/82	1.19/84	1.10/86	1.07/88	1.02/89
2	1.49/85	1.45/85	1.39/85	1.27/87	1.15/88	1.09/90	1.03/89
3	1.82/88	1.79/89	1.69/89	1.55/90	1.35/92	1.23/90	1.09/92
4	2.13/89	2.08/90	1.99/91	1.85/91	1.60/92	1.43/91	1.21/93
5	2.44/91	2.39/91	2.31/91	2.12/93	1.82/92	1.60/94	1.34/94
6	2.72/91	2.73/91	2.58/92	2.38/93	2.05/94	1.84/93	1.47/95
7	3.04/92	2.96/93	2.86/93	2.59/93	2.20/94	1.88/96	1.68/95
8	3.33/93	3.29/93	3.18/93	2.90/95	2.47/95	2.19/95	1.87/96
9	3.59/93	3.56/93	3.43/94	3.15/95	2.76/96	2.44/95	2.00/95
10	3.91/94	3.89/94	3.59/94	3.42/95	2.97/95	2.50/96	2.32/97
20	6.53/96	6.38/96	6.10/95	5.51/97	4.78/95	4.11/97	4.12/96
30	8.59/96	8.49/97	8.24/97	7.57/95	6.63/97	5.06/97	5.47/97
40	10.77/96	10.14/97	9.85/96	8.99/97	8.57/99	6.54/99	7.00/97
50	12.42/97	13.04/97	11.63/98	11.05/98	9.66/96	7.36/98	8.78/97

per record. A given record can then be found using the same method as with the Hash Partition Table, namely, the initial entry e is found by $e = \frac{h-h_l}{h_h-h_l}r$ and then making a linear search to find the correct record. Here r is the number of records in a bucket, h_l the lower hash partition value and h_h the higher partition boundary. Notice too that this is exactly the table needed to support shares and splits.

Where r is large and record length small the overhead of this table may become excessive. An obvious solution is to treat each bucket as a collection of sub-buckets and create a Hash Partition table for it. The overhead then reduces to one Hash Partition Table entry for each sub-bucket and a sub-bucket is found in the same way as a bucket.

4.4 Client and Server Distributed Files

In Linear Hashing for Distributed Files it is shown how to create large data bases distributed over any number of servers. Here an outline is given on how to achieve the same functional characteristics using Partition Hashing.

Each server may be considered a 'super bucket' and the distribution of records across n servers done in the same way as for buckets. A server maintains a Server Partition Table for all servers. Each entry has the server address, Hash Partition boundary and the size of its free space.

To use the database the client first requests the Server Partition Table from any server, the server addresses and corresponding Hash Partition Values are returned. Then to search for a record the client computes the hash and dispatches the request to the appropriate server via the Server Partition Table. The server finds the record using its own Bucket Partition Table and returns the requested data to the client.

Suppose that a server is running out of storage space. Reference is made to the Server Partition Table to find which Server Hash Partition Table adjacent partner has the most available space. A request to share space is made to that partner, the partner agrees and replies with a size to move. The requester now transfers the buckets, starting from the end of its Bucket Partition Table, to the partner one bucket at a time, until the agreed size is reached. After each bucket transfer has been completed successfully the corresponding Server Partition Entries are sent

to the Partner who adds them to its own Partition Table. Then the requester removes the Server Partition Table entry and frees up the bucket and notifies its other adjacent partner of the Hash Partition change. Any search, insert or delete requests for the moved bucket are forwarded to the partner who services them on completion of the move and returns them. These are then returned to the client with the Hash Partition update. A boundary change message is propagated to all the other servers at completion of the share. Notice that this redistribution negotiation is entirely local to the two servers involved and does not involve the client or the other servers, although it may cause a knock-on request from the partner to its adjacent server. In fact any number of servers could be undergoing redistribution simultaneously.

The client meanwhile, unaware of this update makes a further request to find a key using its now out of date Server Partition Table. Doing so it sends the request to the wrong server, the requested record has been moved to a partner during the previous space balancing transaction. The server redirects the request to its adjacent partner and returns the result of the request to the client together with a Server Partition Table boundary update. The client modifies its Server Partition Table so that future requests will be directed correctly.

As with bucket load balancing, server load balancing is an asynchronous process and can proceed independently as a background task. Server balancing activity can be expected to be infrequent and hence Server Partition Table updates rare. A new server coming on line would be added into the Server Partition table where convenient and load sharing triggered with adjacent servers. Taking a server off-line is managed by first moving occupied buckets to adjacent servers and then removing it from the Server Partition Table.

Notice that as with Linear Hashing Server Partition Tables do not need to be perfectly synchronized, differences can be resolved at service request time.

4.5 Conclusion

Partition Hashing builds on the conceptual framework provided by Linear Hashing to create algorithms for dynamic external storage more closely approaching the ideal. Searches both successful and unsuccessful require just one access while inserts cost 1.1 accesses for an 80 percent load factor or by varying one parameter load factors approaching 100 percent can be achieved at the cost of extra assesses on insert. When used in a multi-processor environment the distributed update mechanism appears to be better matched to the ideals outlined by Devine. Devine [1993]

5. SORTED ORDER AMT

In the HAMT the key is hashed before insertion into the tree. However, the entire key can be treated as a hash. In this case keys will be stored in the trie in sorted order and become the familiar character trie.

This is a special case of the more general AMT algorithm included in Bagwell [2000]. However, it is still a valuable algorithm as it has a speed and space occupancy comparable to the more general algorithm yet uses exactly the same efficient memory allocation as HAMT. The structure supports all the expected sorted order functions such as ranges, ordered iteration and so on. Table 8 gives times space

utilization using the same benchmark sets as the HAMT benchmarks in previous tables and can be compared with the TST. The times are expected to be comparable and faster than HAMT as with randomly distributed keys they are unique after about $log_s N$ characters on average, where s is the cardinality of the alphabet, and this is equivalent to the lgN bits of the hash. However, the key is not hashed first, saving the key hashing time and the costly memory accesses to the rest of the characters in the key. However, if the keys are not randomly distributed then the depth of the trie will grow, requiring more space, take more trie levels and consequentially more access time.

The root table may be resized automatically too and gains in speed made. However, more care must be taken as typical key sets, symbol tables for example, use a small part of the available alphabet. As an alternative the first n characters could be hashed and used to index the root hash table. Subsequent characters are taken without hashing and the sort order is retained. An additional two pointers are required in the root hash table entries, a pointer to the next ordered entry and a pointer to a key containing at least the n character prefix. Then sorted order functions can be undertaken and dynamic shrinking feasible. Since the root table is infrequently restructured the cost of maintaining the additional order information is negligible and performance is improved. Further the full key does not need to be examined during resizing. n can be set dynamically allowing the root table to be resized automatically as with HAMT. This arrangement has not yet been characterized.

5.1 Ordered Distributed External Storage

Unsurprisingly the Partition Hashing structure can be adapted for sorted order record storage. If in the above discussion of Partition Hashing and Distributed Files the lower key is substituted for lower Hash Partition then the same Partition Tables structure and splitting may be used to give a distributed ordered data base. The result provides an interesting alternative to B-Trees but with smoother performance, higher load factor and no stored dictionary. Performance will be the same as Partition Hashing but more space could be required in the Partition Table to store a full key and rapid searches can no longer use direct table indexing. However, replacing the table with a Ordered AMT resolves this problem nicely. Similarly the list in each bucket is replaced by an AMT too. In both cases no keys then need to be stored in the trie hence making it only a little more costly than Partition Hashing.

Further research is needed to complete the characterization. However, the structure looks promising.

6. IP ROUTING

IP routers must select an appropriate outgoing route based on the IP address of an incoming packet. The IP address is 32 bits long and for public routing, for class A, B and C sub-nets only the first 24 bits need be considered in order to route a message to the appropriate sub-net. The problem therefore becomes to associate target route for each of the incoming packets from the first 24 bits of the address. Nilsson and Karlsson [1998] describe a high performance algorithm to do just this while here I demonstrate that a comparable performance algorithm can be created

Table 8. Ordered Insert, Search and Size

	AMT	AMT	AMT	AMT	TST	TST
SetSize	**Insert**	**Search**	**Total Pool**	**Free Pool**	**Insert**	**Search**
8K	3.40	0.59	14K	2K	10.23	2.78
16K	3.74	0.69	27K	4K	11.08	2.99
32K	3.67	0.78	50K	8K	12.67	3.27
64K	3.62	0.86	88K	13K	14.72	3.57
128K	3.95	1.02	171K	25K	14.59	3.87
256K	4.76	1.05	382K	58K	13.76	4.16
512K	5.16	1.20	860K	117K	13.07	4.51
1024K	4.59	1.27	1670K	238K	12.26	4.86
2048K	4.68	1.31	2927K	425K	–	–
4096K	5.19	1.30	5646K	839K	–	–
8192K	6.33	1.34	12006K	1755K	–	–

simply with the AMT data structure.

Fig 6 illustrates the data structure to be used and is clearly a special case of the hash tree previously described. The root table is created with a length of $\frac{2^{24}}{32}$ that is to have 2^{19} entries requiring 4 Mb. Each entry is either empty, contains a route or is a 32 entry sub-trie and pointer. To identify a route for an IP address the first 19 bits are used as an index into the root table and then the remaining 5 bits used to select the entry in the sub-trie. Unknown addresses can be identified and dropped with just the index and a bit test on the bit map.

The IP routing table can be updated using the same basic insertion algorithms described in the hash tree above while the non-critical memory allocation is left to the system.

For the new address format with IPV6 using 128 bit addresses a HAMT would provide an efficient solution with O(1) access.

```
// IP is the IP address
  if((E=RootTbl[IP>>13])&&(E->Map&(1<<((IP>>8)&0x1F)))
  Route=E->Base+CTPOP(E->Map&(~((~0)<<((IP>>8)&0x1F));
else // Failed
```

Fig. 5. Code fragment for IP dispatch

7. CLASS-SELECTOR DISPATCH

In modern object oriented language design it is desirable to be able to create Class-Selector tables dynamically. As classes inherit from super classes the compiler must maintain a table valid method selectors. At run time it must be a fast operation to map a class-selector pair to an actual method and verify that it is also valid or perhaps dynamically load new classes with their associated methods. A good solution to the compilers static task is described in Driesen [1993] and Driesen and Holzle [1995]. A comparable solution to the problem is simply constructed with an AMT data structure yet has the added property of being dynamically updateable.

The class and selector are assumed to be packed in one integer, the first 18 bits representing the class while the last 14 bits the selector.

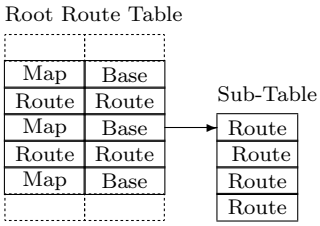

Fig. 6. IP Routing Table

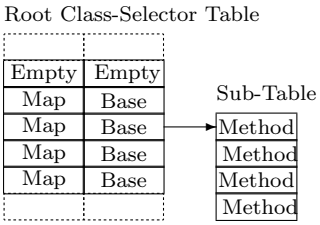

Fig. 7. Class-Selector Table

It is initially assumed that there will be 2^c classes with any of 2^s selectors. An empty root table is then created of length $2^{(c+s-5)}$. As classes and methods are compiled they are added to the table. The last c bits of the class is combined with the last s bits of the selector and the method pointer entry made in the appropriate node of the sub-trie. For method pointer look up the same process is followed. c class bits are combined with s selector bits. The first $c+s-5$ bits are used to index into the root table while the last 5 bits index into the sub-trie. Either the method pointer is retrieved or the class/selector is invalid.

```
// Class C, Selector S
Index= (C<<(s-5))|(S>>5);
if((E=RootTbl[Index])&&(E->Map&(1<<(S&0x1F)))
  MethodPtr=E->Base+CTPOP(E->Map&(~((~0)<<(S&0x1F));
else // Failed
```

Fig. 8. Code fragment for Method dispatch

The class-selector table can be updated using the same basic insertion algorithms described in the hash tree above while the non-critical memory allocation is left to the system.

However, the number of classes or selectors may overflow the allocate 2^c or 2^s entries initially reserved. When this occurs the table can be easily restructured to accommodate the additional entries. Suppose for the moment that the classes

exceed 2^c. The new class limit is set to c' where $c' = c + 1$ and thereby double the number of allowed classes. A new root table is created with a length of $2^{(c'+s-5)}$, the old root table entries are copied to the new one and the old table deleted. Future searches will use c' bits to identify the class. Similarly if the selectors overflow 2^s the new selector limit is set to s' where $s' = s+1$. A new root table is created with length $c + s' - 5$ and the entries copied from the old table to the new one, taking s entries at a time from the old table and inserting these in the new one. After each move s entries are left empty, for future selector insertions, before copying the next block.

Although apparently longer this algorithm makes the same number of memory accesses as the row displacement algorithm referred to above, yet is dynamic, allows run time updating and using memory progressively. With an additional memory access, more memory may be conserved by allowing the trie one more level.

8. PERFORMANCE COMPARISONS

All the algorithms were tested with the same wide variety of unique key sets. The sets were in random, sorted order or semi-ordered sequence. The test sets were produced using a custom pseudo random number generator. These were created by first generating a random prime number which was repetitively added to an integer to produce a sequence of 32 bit integer keys. For high value prime numbers this sequence is pseudo random while for low value prime numbers the sequence is ordered. Values in between generate semi-ordered sequences.

An alphabet cardinality of 250 was used as a radix to convert the integer key to a string of characters. Additional characters were added to create an eight-character key. All the test results were for 8 character key sets created in this manner.

The technique ensures unique key production and allows search test key sets to be generated with different sequences to those used during insertion yet from the same key set.

Each algorithm's performance was measured on an Intel P2, 400 MHz with 512Mb of memory, NT4 and VC6. Times shown are in μS per insert or search for 8 character keys averaged across 50 test key sets. CTPOP was emulated using the algorithm in Fig 2.

The external disk and distributed storage algorithms were tested by simulation. No actual timings were made but the critical characteristic of disk accesses per search and insert together with load factor was monitored in the simulation model over a large number of runs.

ACKNOWLEDGMENTS

I would like to thank Prof. Martin Odersky and Christoph Zenger at the Labo. Des Methodes de Programmation (LAMP), EPFL, Switzerland for their review of the draft paper and valuable comments.

REFERENCES

Bagwell, P. 2000. Fast and space efficient trie searches. *Technical Report, EPFL Swtzerland*.

Bentley, J. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (1997), SIAM Press*

(1997).

Devine, R. 1993. Design and implementation of DDH: A distributed dynamic hashing algorithm. *Lecture Notes in Computer Science 730*, 101–114.

Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F. M., Rohnert, H., and Tarjan, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing 23*, 4, 738–761.

Driesen, K. 1993. Selector table indexing sparse arrays. In *Conference on Object-Oriented* (1993), pp. 259–270.

Driesen, K. and Holzle, U. 1995. Minimizing row displacement tables. In *OOPSLA 95* (1995).

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems 4*, 3, 315–344.

Fredkin, E. 1960. Trie memory. *Communications of the ACM 3*, 490–499.

Knuth, D. 1998. *The Art of Computer Programming, volume 3: Sorting and Searching, 2nd Ed.* Addison-Wesley, Reading, MA.

Larson, P. 1988. Dynamic hash tables. In *Comm. of ACM (CACM)*, Volume 31 (1988), pp. 446–457.

Litwin, W., Neimat, M., and Schneider, D. 1993. Linear hashing for distributed files. *ACM-SIGMOD Intl. Conf. on Management of Data, 1993.*

Litwin, W. and Schwarz, T. 2000. LH* RS : A high-availability scalable distributed data structure using reed solomon codes. In *SIGMOD Conference* (2000), pp. 237–248.

Nilsson, S. and Karlsson, G. 1998. Fast address look up for internet routers. In *Proceedings of IEEE Broadband Communications 98* (April 1998).

Nilsson, S. and Tikkanen, M. 1998. Implementing a dynamic compressed trie. In *2nd Workshop on Algorithm Engineering WAE 98 Saarbruecken Germany* (August 1998).

R.Bayer and E.M.McCreight. 1972. Organization and maintenance of large ordered indices. *Acta Informatica 1*, 3, 173–189.

Sedgewick, R. 1998. *Algorithms in C++, 3rd Ed.* Addison-Wesley, Reading, MA.