HOW TO ROLL YOUR OWN DBM/NDBM
Ozan S. Yigit

Introduction

External Hashing is a well-known technique for organizing direct-access files, and has been investigated in the last two decades. If the key-order related operations are not required, external-hashing is often the choice for a fast (one or at most two page faults) database interface.

Such a database system (dbm or ndbm) is available under some versions of the Un*x operating system, but general unavailability under *all* versions of Un*x, as well as the lack of source code implies that a re-write of an dbm/ndbm work-alike would be welcomed.

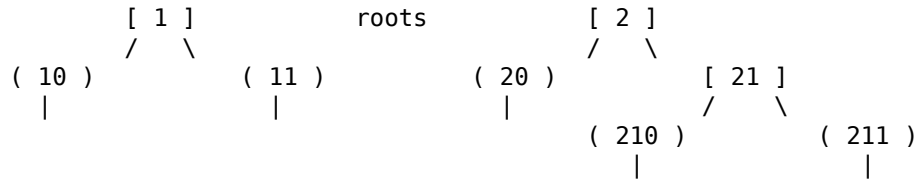
The rest of this document describes a particular implementation of an external hashing scheme, called "Dynamic Hashing", by Per-Ake Larson, BIT 18 (1978) pp 184-201. This implementation (partly based on section 5: refinements and variations) is believed to resemble dbm/ndbm implementation under Un*x, and achieves comparable performance, using the same index/data file scheme. It requires essentially the exact DBM data structure presented in ndbm.h. A detailed discussion of Dynamic Hashing, and its analysis may be found in Larson's paper.

A simplified overview

As with most other external hashing schemes, Dynamic Hashing requires a data file, and an index to the data file. In the case of Dynamic Hashing, the index is a forest of binary hash trees, which are very closely related to binary tries, or radix search tries. The data file is simply a number of buckets (pages) of fixed size. A record R to be stored is assumed to contain a unique key K.

The file is loaded by using a hash function H0 on K and the index to locate the appropriate buckets. Once a bucket is full, it is split, a new bucket is allocated, and all the records in the overflowing bucket is distributed between the two buckets. Index is updated accordingly.

The following index depicts a data file of five buckets as a result of three splits: (the leafs point to the data pages in the data file)



What happened above is as follows: bucket 1 filled up, split into two buckets pointed by nodes 10 and 11. Bucket 2 filled up, split into two buckets pointed by nodes 20 and 21 and so on.

In order to locate a bucket in which the key resides during the lookup, the decision to whether to put a record in the left or the right bucket when splitting must be uniquely determined by the key of the record. This

also determines a unique path through the tree for a given key.

In order to facilitate this, Larson uses a function B [suggested to be a pseudo-random number generator that is designed to return 0 or 1 with probability 0.5 when called], which maps the key into an infinite binary sequence, B(K) = (Di...Dn) where D is 0 or 1, for i = 0, 1, 2 ... n [random number generator is seeded by another hash function H1(K) and will return the same infinite sequence given the same seed] This sequence can be used to determine a unique path in a binary tree: if the next digit D in the binary sequence is 0, take the left branch, otherwise that the right branch of the tree etc.

To insert a record, R with key K, H0(K) is computed to locate the root of the tree. Starting from the root, a unique path determined by B(K) is taken until a leaf node is reached, and the record is inserted into the bucket pointed by the leaf node. If the bucket is full, the node is split into two leaf nodes (itself is no longer a leaf), a new bucket is allocated and every key in the overflowed bucket is split between the two buckets, using the next digit in the binary sequence for each key. [This is also known as the "digital splitting".]

In the early sections of his paper, Larson presents the index tree as a conventional data structure, with a type tag, left-son, right-son etc. fields. The tag field indicates whether the node is an internal node (1) or a leaf node (0). The following is a somewhat altered chunk of his algorithm for record insertion in a nawk-like pseudo-code.

```
store(key, dat) {
        node = findroot(H0(K))  # rootnode
        initB(H1(K))            # random n generator initialized
#
# traverse the tree
#
        for (seqc = 0; TTAG(node) == 1; seqc++)
                if (B() == 0)
                        node = LSON(node)
                else
                        node = RSON(node)
#
# found a leaf node [TTAG(node) == 0]
#
        while (putpage(node, key, dat) == 0) {  # cannot insert ??
                ... obtain new buckets          # split time
                foreach K in PAGE(node)
                        initB(H1(K))
                        ... call B seqc times   # skip the initial sequence
                        if (B() == 0)           # use the next one
                                movekey(bucket-a)
                        else
                                movekey(bucket-b)
                ... etc
        }
        ... etc
}
```

In the above [brief] algorithm, seqc simply counts the number of digits used to reach to a leaf node. This count is later used during the split

process, and the *next* digit in the sequence is used to split the keys into two pages. It should also be obvious that the crucial part of the algorithm is the tree-traversal, as it is used to traverse a unique path for each key that ends in a leaf node. This portion is needed for insertion, deletion and lookup operations.

Refinements and Variations

At this point, there is enough material to consider Larson's refinements and variations. [Interested reader should consult the original paper for a through and probably a more accurate description of the algorithms] He observes that a bit-vector is adequate to reflect the structure of the binary trie, as the tag field is either a 1 or a 0, depending on the node type, and the locations of the left and right sons can be obtained with simple arithmetic. If we ignore Larson's original structure as a "forest" of tries, and reduce it to a single trie (rooted at 0th node), the index depicted earlier (0 indicates a leaf node) would look something like this:

```
+---+---+---+---+---+---+---+ - - +---+---+ - -
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |     | 0 | 0 |
+---+---+---+---+---+---+---+ - - +---+---+ - -
  0   1   2   3   4   5   6   . .  13  14
```

In this 0-based bit array form, given a node number, the locations for its left-son and right-son may be calculated as 2*i + 1 and 2*i + 2. Thus: (turn it 90 deg. clockwise to see the that this does depict our original index)

```
0 -> 2 --> 6 ---> 14 (leaf)
           6 ---> 13 (leaf)
     2 --> 5 (leaf)
0 -> 1 --> 4 (leaf)
     1 --> 3 (leaf)
```

The next variation involves B, the binary sequence generator. Larson's reason for using B is that usual key transformations (e.g. hashing) may not be able to generate enough bit-randomization. Assuming that there exists such a hash function, the bits of the transformed key may be used directly to traverse the trie. When both of the above variations are implemented, the traversal portion of the algorithm may look something like the following: [Yes, that if-else construct within the while loop can be simplified, but this code is just to illustrate the logic flow.]

```
hbits = H0(K)   # hash
i = 0           # bitmap index (starting at root)
x = 0           # hash bit index

while (getbit(bitmap, i) == 1) {        # non-leaf
        if (getbit(hbits, x) == 0)
                i = 2 * i + 1           # lson
        else
                i = 2 * i + 2           # rson
        x++
}
# found a leaf node
```

One remaining issue is the bucket address associated with a leaf node.

Larson proposes that unused bits in the bitmap may be used to store the bucket address. A simpler scheme, appearently used by Un*x dbm/ndbm (based on the public accounts of the algorithm) is simply to use the indexed portion of the hash value as a page number. This may be accomplished by using a table of masks (or by building a mask on-the-fly) to mask-out the unused portion of the hash value. After incorporating this last variation into the above algoritm, we can write a function "getpage" that takes one argument, the hash value for the key, and returns a block number associated with that key.

```
getpage(hbits) {
        i = 0           # bitmap index (starting at root)
        x = 0           # hash bit index

        while (getbit(bitmap, i) == 1) {        # non-leaf
                if (getbit(hbits, x) == 0)
                        i = 2 * i + 1           # lson
                else
                        i = 2 * i + 2           # rson
                x++
        }
        curbit = i                              # found a leaf node
        return hbits | (curmask = masks[x])     # block number
}
```

This is essentially the heart of our "dynamic hashing" implementation, as it is required by all other routines. The other crucial portion, "bucket splitting" is easy to construct: recall that Larson used the next digit in the binary sequence (after skipping the indexed portion) for each key to split the keys between two buckets. Given that we have saved the mask (curmask above) that is used to extract the page number, we know that the next bit that was masked-out is the bit to use for a split:

```
        while (putpage(page, key, val) == 0) {  # cannot insert ??
                ... obtain a new page           # split time
                foreach K in page
                        if (hash(K) & (curmask + 1))
                                movekey(newpage)
                setbit(bitmap, curbit)
        }
```

Note that we are only moving the keys with the next bit set into the new page, and once that is done, the splitting node in the bitmap is set to 1, indicating the split. Next time around, (say, during a lookup) one more bit in the hash value will be examined, and if it is 0, the page address will be the previously split page, else, it will be the new page.

Once the core algorithms are exposed, it is easy to build the rest of the support functions and the appropriate wrappers into a complete hashed database library (we assume key/value pairs are stored together on the page):

```
#
# fetch the value of a given key. returns value
# or nullvalue
#
```

```
dbmfetch(key) {

        page = getpage(hash(key))

        if ((val = getpair(page, key)) != null)
                return val
        return nullval
}

#
# delete both the key and its value from the database
# returns 1 (success) or 0 (failure)
#
dbmdelete(key) {

        page = getpage(hash(key))

        if (delpair(page, key) != null)
                return 1
        return 0
}

#
# insert a key/value pair into the database
# returns 1 (success) or 0 (failure)
#
dbminsert(key, val) {

        page = getpage(hash(key))

        if (putpair(page, key, val) == 1)
                return 1
        else                                            # split time
                for (i = 0; i < maxsplit; i++) {
                        newp = getnewp();               # new page
                        foreach K in page
                                if (hash(K) & (curmask + 1)) {
                                        delpair(page, K);
                                        putpair(newp, K, val(K))
                                }

                        setbit(bitmap, curbit)

                        if (putpair(page, key, val) == 1)
                                return 1
                        # still did not fit. try some more
                }
        #
        # if we are here, key did not fit after N splits.
        #
        return 0
}
```