Reihe Informatik
19 / 2001

A Robust Scheme for Multilevel Extendible Hashing

Sven Helmer Thomas Neumann Guido Moerkotte

A Robust Scheme for Multilevel Extendible Hashing

Sven Helmer, Thomas Neumann, Guido Moerkotte

April 17, 2002

Abstract

Dynamic hashing, while surpassing other access methods for uniformly distributed data, usually performs badly for non-uniformly distributed data. We propose a robust scheme for multi-level extendible hashing allowing efficient processing of skewed data as well as uniformly distributed data. In order to test our access method we implemented it and compared it to several existing hashing schemes. The results of the experimental evaluation demonstrate the superiority of our approach in both index size and performance.

1 Introduction

Almost all dynamic hashing schemes proposed up to now do not consider the case of skewed hash keys. In practice, however, it is very difficult to guarantee uniformly distributed hash keys. This leads to severe problems for these hashing schemes as exponentially growing directories are the result.

We propose a robust extendible hashing scheme able to handle skewed hash keys much more efficiently than existing hashing schemes, without compromising the performance for uniformly distributed data. Surprisingly, the main idea behind our index is easy to grasp, while at the same time being very effective. We divide the directory of our hash table hierarchically into several subdirectories, such that the subdirectories on lower levels share pages in an elegant way. This allows us to save space without introducing a large overhead or compromising retrieval performance. We show the effectiveness of our approach by presenting results from our extensive experiments.

What are the reasons for skewed hash keys? The predominant opinion is that by using appropriate hash functions even starting out with skewed data will result in a reasonably uniform distribution of the hash keys. We do not share this point of view. Obviously good hash functions can be chosen when the data that is to be indexed is known beforehand. In practice, this is very rarely the case, here one has to expect bursts of data that are heavily skewed in not always anticipated ways. Futhermore multiple identical data values also induce skew in hash keys as do specialized hash functions, e.g. those allowing order preserving hashing. So it is not sufficient to look only at skewed data, but to consider skewed hash keys as well. One of the unresolved issues with respect to the distribution of the hash keys is the lack of hashing schemes whose performance does not deteriorate when faced with non-uniformly distributed hash keys. This has prevented the large-scale use of indexes based on hashing in practice. Vendors of commercial databases are reluctant to integrate these access methods into their database management systems.

The paper is organized as follows. In the next section we describe the state of the art of dynamic hashing. Section 3 covers our new approach. In Sections 4 and 5 we describe in detail the environment and the results of the experiments we conducted to test our approach and compare it to existing techniques. Section 6 contains a detailed comparison with the approach by Otoo [15] and Du and Tong [3]. Section 7 concludes our paper.

2 State of the Art

The predominant index structure used in relational database systems today is the B-tree [1]. Although tree structures are excellent for most applications found in relational database systems, there are problematic areas like the indexing of large, similar strings. Usually, strings are stored in so called prefix B-trees [2]. When splitting nodes we have to find a *separator string* that distinguishes the entries in the nodes. In the worst case, these separator strings can become quite long. This leads to a loss of performance, because large separator strings reduce the fanout significantly. If bounded-length keys are used, it may not even be possible to index certain data.

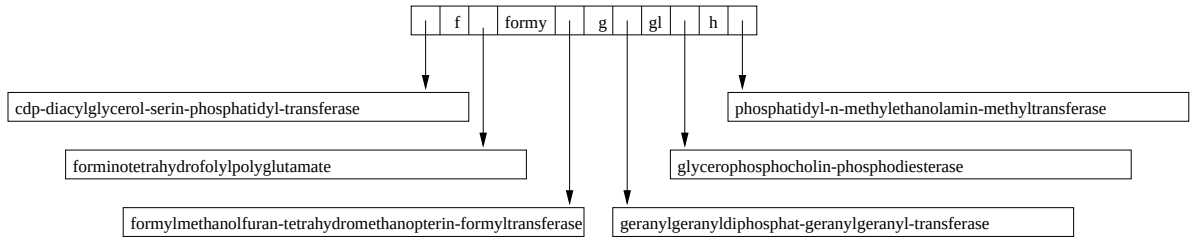


Figure 1: Indexing strings with dissimilar prefixes

Let us give a brief example of the problems that can arise in prefix B-trees. For simplicity let us assume that we have 64 byte pages in our B-trees and that we use 8 byte references. As long as the prefixes of the indexed strings are diverse enough, we have no trouble finding short separators, even if the strings are long (see Figure 1). If, however, we have long, identical prefixes, the separators will also be quite large (see Figure 2). This reduces the fanout of the B-tree considerably. A reduced fanout in turn leads to a worse performance. Compressing the separators attenuates this effect, but not enough to make prefix B-trees competitive for these applications.

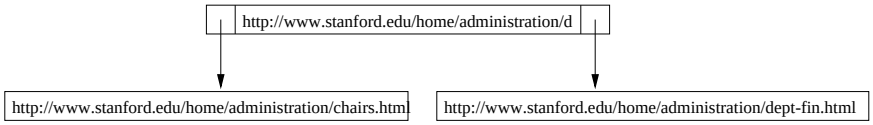


Figure 2: Indexing strings with similar prefixes

Hash tables are one of the fastest (if not the fastest) data structures for searching. In a dynamic environment, where it is not known beforehand how many data items will be inserted, one of the few disadvantages of hash tables becomes apparent, however. If

we exceed the capacity of a hash table, the performance deteriorates. So we need to reorganize the table, possibly rehashing all inserted data items. While in main memory a total rebuild may be tolerable, on secondary storage this is out of question. There have been several approaches to solve this problem (for secondary storage). These hashing schemes are known as dynamic hashing. One of the earliest dynamic hashing schemes was linear hashing by Litwin [8]. Linear hashing employed chained overflow buckets, which led to performance loss during lookups as linked lists had to be traversed in secondary storage. Fagin, Nievergelt, Pippenger, and Strong eliminated overflow buckets in their extendible hashing scheme [5], but at the price of an exponentially growing directory for skewed data. Let us have a more detailed look at this situation.

2.1 Problems with Extendible Hashing

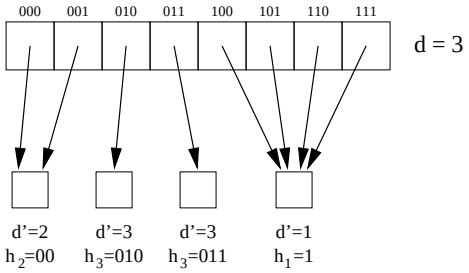


Figure 3: Extendible Hashing

An extendible hashing index is divided into two parts, a directory and buckets (for details see also [5]). In the buckets we store the full hash keys of and pointers to the indexed data items. We determine the bucket into which a data item is inserted by looking at a prefix h_d of d bits of the hash key h. For each possible bit combination of the prefix we find an entry in the directory pointing to the corresponding bucket. The directory has 2^d entries, where d is called *global depth* (see Figure 3). When a bucket overflows, this bucket is split and all its entries are divided among the two resulting buckets. In order to determine the new home of a data item the length of the inspected hash key prefix has to be increased until at least two data items have different hash key prefixes. The size of the current prefix d' of a bucket is called *local depth*. If we notice after a split that the local depth d' of a bucket is larger than the global depth d, we have to increase the size of the directory. This is done by doubling the directory as often as needed to have a new global depth d equal to the local depth d'. Pointers to buckets that have not been split are just copied. For the bucket that was split, the new pointers are put into the directory and the global depth is increased.

As long as we insert uniformly distributed data into an extendible hashing scheme, everything is fine. If we insert skewed data, however, we waste a lot of space by storing redundant pointers. Let us give an example. Assuming that due to heavily skewed data it is always the first bucket that overflows and splits. We get a structure as depicted in Figure 4.

Although we only have 5 buckets we need 16 pointers to reference them. In this worst case scenario we have to double the number of pointers for each overflow that occurs. As

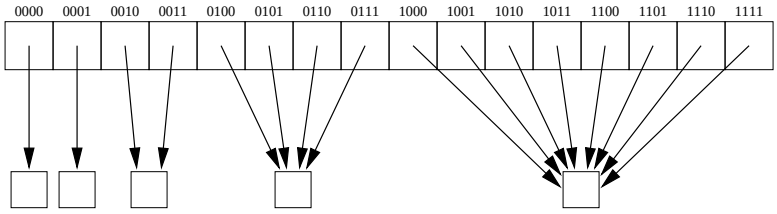


Figure 4: Inserting skewed data into an extendible hash table

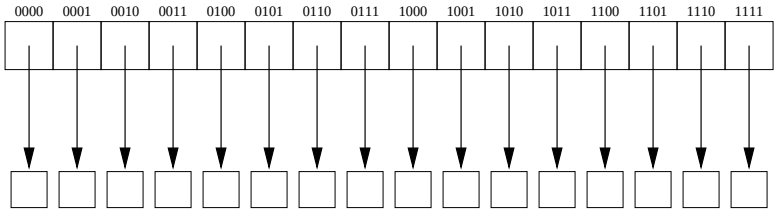


Figure 5: Inserting uniformly distributed data into an extendible hash tree

one can easily imagine, the physical limit of the directory growth is reached quite fast.

When reaching the limit we have two alternatives. We can introduce overflow buckets or use a hierarchical directory. Overflow buckets are contrary to the basic idea of extendible hashing, which tries to avoid overflow buckets [13]. The reason for this is that long chains of overflow buckets lead to severe performance losses. Simply organizing the extendible hash directory hierarchically as a hash tree (as suggested in [5] and [19]) does not get the job done, either. Although superior to an ordinary extendible hashing scheme for skewed data, extendible hash trees waste a lot of space for uniformly distributed data. Let us illustrate this behavior.

Figure 5 shows an extendible hash tree with a single root node (before allocating nodes on subsequent levels). When a bucket overflows we have to allocate another hash table one level lower and insert the elements of the split bucket into this hash table (Figure 6 shows the resulting structure). This newly created table has only two entries, the rest of the page is not utilized yet (shaded in gray). For uniformly distributed data we expect all buckets to split at roughly the same time. That means for each bucket in Figure 5 we

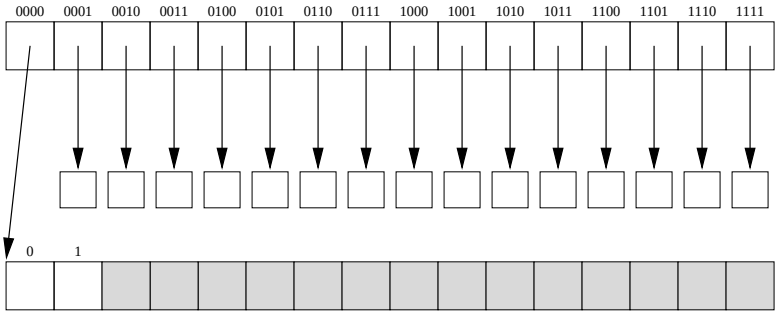


Figure 6: Inserting uniformly distributed data into an extendible hash tree

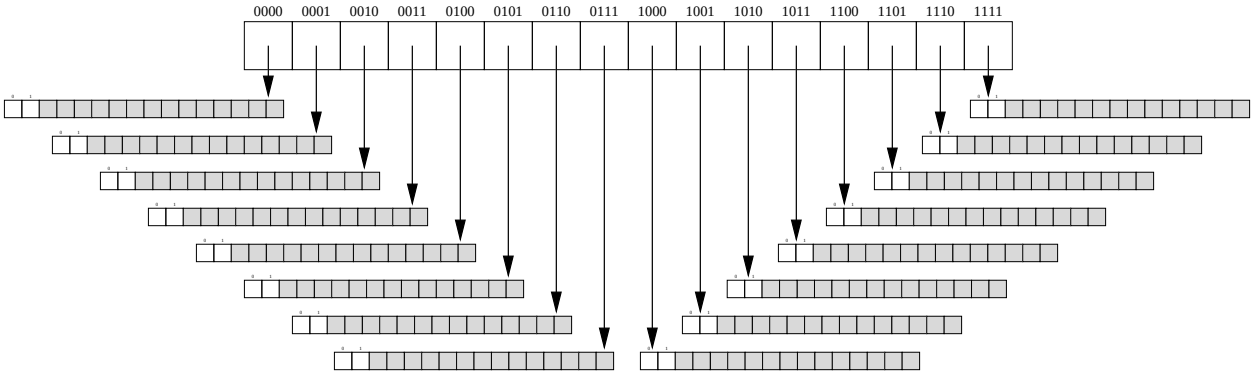


Figure 7: More expansions in the extendible hash tree

now have a hash table with two entries (see Figure 7). In this example we need 17 pages to distinguish the first 5 bits of the hash keys. In an ordinary extendible hashing scheme we would only need 2 pages.

2.2 Related Work

Since the initial work by Larson [7] there have been many proposals for improving dynamic hashing (for an overview see [4]). Most of these schemes, e.g. [12, 17, 18], assume that the hash values are distributed in a reasonable way, i.e., close to uniform distribution, so that any skew in the original keys is eliminated for the most part by an adequate hash function. Theoretical analyses [6, 14] also have not considered heavily skewed hash keys.

As already mentioned real data is very rarely distributed uniformly. In addition to that, in practice it will often not be possible to generate hash values from skewed data in such a way that these values will be reasonably distributed. This is due to several reasons. First of all, there is no universal hash function that will perform equally well for all data that is ever to be introduced into a database system. It is also unrealistic to rely on the average database user to supply an adequate hash function when creating an index. So the database system has to analyse the data that is to be inserted. This assumes that all data is known at the time of index creation, which is not realistic. So we have to make a decision based on insufficient knowledge. Once we have decided on a hash function, however, there is no turning back without rehashing the whole table, when the distribution of the inserted data changes over time. Even if it were possible to find a perfect hash function, as soon as we allow multiples in the inserted data values, we would not be able to avoid skewed hash values anymore.

Therefore it is of utmost importance for dynamic hashing structures to be able to handle pathetic cases of data without failing completely, i.e., we have to be able to get the upper hand of the directory growth without compromising the retrieval performance.

Remarkably little work has been published on non-uniform distributions of hash keys. Otoo proposes a balanced extendible hash tree in [15], which grows from the leaves to the root, similar to a B-tree. Although this solves the problem of wasted space, the lookup costs increase. In a balanced extendible hash tree all lookups will cost as much as the most expensive lookup. For example, if we can store tables up to a depth of 10 on a page and we need 30 bits to distinguish the most skewed data items, the tree will have (at

least) three levels. Due to the balancing, all queries have to traverse three levels.

One possible approach to conserve space in extendible hash trees is page sharing of tables on lower levels. Du and Tong suggest a multi-level extendible hash tree in [3], which is based on this technique. They try to reduce the space expenditure further by compressing the entries. This, however, backfires as their scheme is very complicated, due to the fact that compression destroys the regular structure of extendible hashing. Consequently, more organizational data has to be stored, which cancels out the effect of compression. We give a more detailed comparison with the approaches by Otoo and Du/Tong in Section 6.

A different approach, trie hashing, to index long alphanumeric keys has been taken by Litwin [9]. In principle this is a trie mapped to a binary tree. The original proposal assumed a main memory structure, but later this was adapted to secondary structure [11] and improved further [10]. Trie hashing degenerates when data is inserted in an unfavorable order (e.g., sorted or almost sorted). This case results in an unbalanced binary tree. Otoo and Effah try to solve this problem by introducing a structure similar to red-black trees [16]. The balancing eliminates degenerate binary trees, but because of the complex rotation operations, this approach would be to costly for secondary storage.

3 Our approach

As Du and Tong in [3], we propose a multi-level extendible hash tree scheme that also shares pages. We, however, do not compress the index pages and therefore have a very regular, well-ordered scheme. This allows us to store only pointers on the pages. No additional information is required. This way we achieve the maximal possible fanout for a given page size.

3.1 General description

Let us illustrate our index with an example. Assume that we only allow one page for the top level directory. We expand the top level directory to a maximum of 2^n entries. That means, we assume an initial configuration as in Figure 5. However, this time we proceed differently than for simple hash trees. If another overflow occurs (for example in the first bucket) we allocate a new hash table of depth 1 one level below the top level to distinguish the elements in the bucket according to the next bit. We do this not only for the overflowing bucket, but also for all buddies of this bucket (i.e. all buckets at positions starting with 0). The hash tables for the buddies are created in anticipation of further splits. All of these hash tables can be allocated on a single page, resulting in the structure shown in Figure 8.

If another overflow occurs in one of the hash tables on level 2 causing the growth of this hash table, we increase the global depth of all hash tables on this page by 1, doubling their directory sizes. We now need two pages to store these tables, so we split the original page and copy the content to two new pages. Adjusting the pointers in the parent directory is our next task. The left half of the pointers referencing the original page now point to one of the new pages, the right half to the other new page (see Figure 9).

The space utilization of our index can be improved by eliminating pages with unnecessary hash tables. The page on the right-hand side of the second level in Figure 9 is

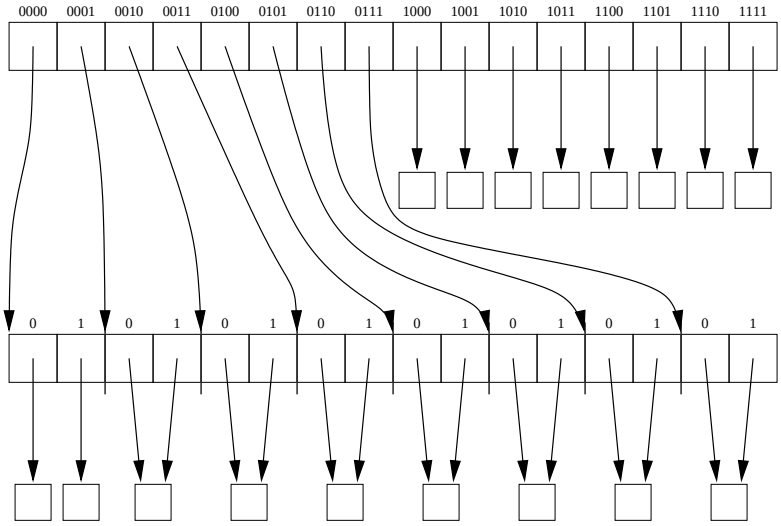


Figure 8: Overflow in our multi-level hash tree

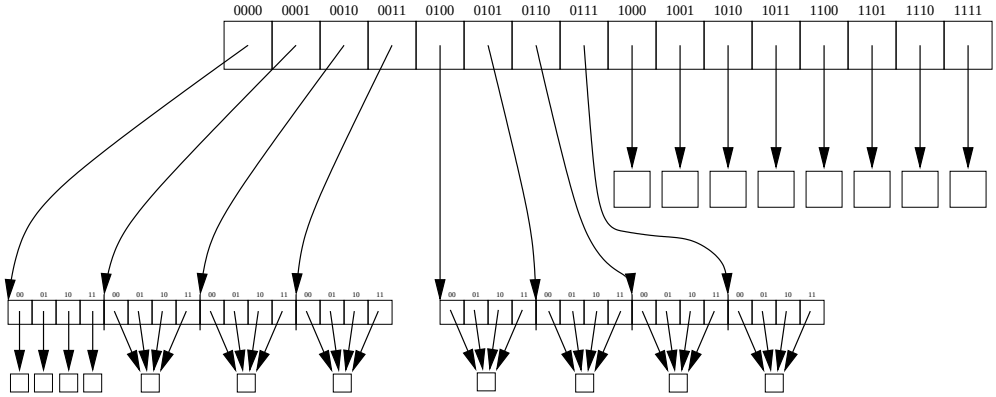


Figure 9: Overflow on the second level

superfluous as the entries in the directories of all hash tables point to a single bucket,i.e., all buckets have local depth 0. In this case the page is discarded and all buckets are connected directly to the hash table one level higher. (see Figure 10).

3.2 Details

In this section we give a more detailed description of our multi-level extendible hashing scheme. We illustrate the calculation of the global depths of the hash tables and the procedures for lookup and insertion of data items. We also describe how to integrate order preserving hashing and range queries into our scheme

3.2.1 Lookups

The global depth of a hash table is stored implicitly in the parent hash table in our scheme. If we have 2^{n-i} pointers in the parent hash table referencing the same page (i.e.,

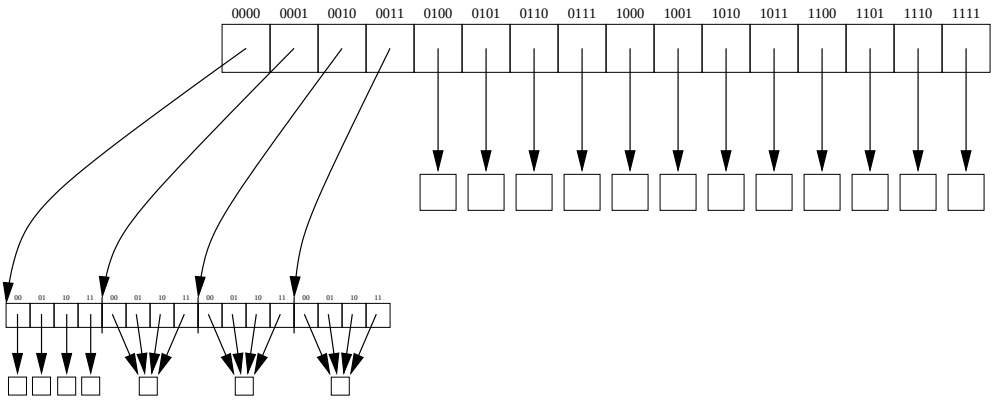


Figure 10: Eliminating unnecessary hash tables

we have 2^{n-i} buddies with the same depth at the moment), all those buddy hash tables have depth i. This means, we can compute the global depths of all hash tables in the directory (except the depth of the root table) and do not need to store this information anywhere, giving us more space to store the actual hash tables.

Lookups are easily implemented. We have to traverse inner nodes until we reach a bucket. On each level we determine the currently relevant part of the hash key. This gives us the correct slot in the current hash table. As more than one hash table can reside on a page, we may have to add an offset to access the right hash table. Due to the regular structure, this offset can be easily calculated. We just multiply the last $n - i$ bits of the relevant pointer in the parent table with the size of a hash table on the shared page (i times the size of a pointer). If $n - i = 0$, we do not need an offset, as only one hash table resides on this page. If we reach a bucket, we search for the data item in this bucket. If the bucket does not exist (no data item is present in the bucket at the moment), we hit a NULL-pointer and can abort the search. Principally, there are two different, efficient ways to distinguish directory pages from bucket pages. One is to mark bucket pages with a flag (e.g. `0xffffffff`) at a certain offset. The other is to use one bit (preferably the most significant bit) in pointers to tell apart pointers referencing buckets from those referencing directory pages. We used the latter approach to save an additional lookup while folding up superfluous hash tables.

Let us now give the pseudo code for lookups:

```
lookup(hashkey, value) {
  currentLevel = 0;

  while(true) {
    pos = relevant part of hashkey
          (for current level);
    determine offset;
    pos = pos + offset;

    nodeId = slot[pos];
    if(nodeId is a null-pointer) {
```

```
      return false;
    }
    if(node is bucket) {
      search in bucket;
      return answer;
    }
    currentLevel++;
  }
}
```

3.2.2 Insertions

Inserting new data items into the hashing scheme is not very difficult either. First of all, we have to find the correct place to insert the data item. Finding the right bucket works as in the lookup procedure. Then several different cases have to be distinguished. If we reach a NULL-pointer, we just allocate a new bucket and insert the data item. Obviously, placing a data item into a non-full bucket does not cause any problems, either. We have to take care if an overflow occurs. The simplest case, in which the global depth of the current hash table does not change, just involves splitting the bucket and adjusting the pointers in the current hash table. If the global depth increases, but the new, larger hash table is not larger than a page, we split the inner node. The global depths of all hash tables on this page are increased by one. The hash tables in the first half of the page remain on this page. The hash tables in the second half of the page are moved to a newly allocated page. Then the pointers in the parent hash table are modified to reflect the changes. We optimize the space utilization at this point, if we discover that the buckets of all hash tables in one of the former halves have a local depth of one (or are not yet present). In this case (compare to the node in the lower right corner of Figure 9) we do not need this node yet and connect the buckets directly to the parent hash table. We have to investigate one more case, an insertion into a hash table of maximal size. If this happens, we add a new level with 2^{n-1} hash tables of depth 1 to the existing index structure (compare to Figure 8). After modifying the index structure we reinsert the data item. The pseudo code for the insertion procedure looks as follows:

```
insert(hashkey, value) {
  currentLevel = 0;

  while(true) {
    pos = relevant part of hashkey
          (for current level);
    determine offset;
    pos = pos + offset;

    nodeId = slot[pos];
    if(nodeId is a null-pointer) {
      allocate new bucket;
      insert pointer to bucket into hash table;
      insert data item;
```

```
      return;
    }
    if(node is bucket) {
      if(node is not full) {
        insert data item;
        return;
      }
      if(local depth of bucket <
         global depth of table) {
        split bucket;
        adjust hash table;
        insert(hashkey, value);
        return;
      }
      if(global depth < bits per level) {
        split inner node;
        adjust buddies;
        insert(hashkey, value);
        return;
      }
      // we have a full grown hash table
      insert new level;
      insert(hashkey, value);
      return;
    }
    // keep on searching
    currentLevel++;
  }
}
```

3.2.3 Range Queries

Given an order preserving hash function it is also possible to support range queries with our scheme. One important prerequisite for dynamic hashing schemes supporting range queries is the ability to handle skewed hash keys, as the application of order preserving hash functions seldomly results in an uniform distribution of hash keys. Therefore our index structure is an ideal candidate for order preserving hashing.

Technically all we have to do (once an order preserving hash function has been applied) is to link the buckets of our index in a double linked list. When a new bucket is allocated due to a split operation, it is always a neighbor of the previous unsplit bucket (which knows its neighbors). So it is straightforward to link the new bucket into the list of buckets.

Inserting a new bucket into a previously empty slot of a hash table is slightly more difficult (see Figure 11). We just have to follow the pointer in the previous/next non-empty slot to find the preceeding/succeeding bucket (dotted lines in the figure). This may involve more than one level of the directory, but we only need to traverse downwards.

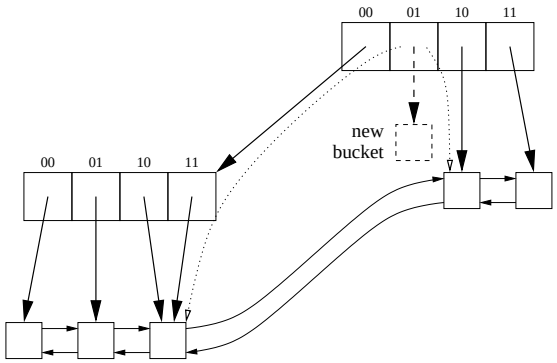


Figure 11: Double linked list of buckets

4 Evaluation

When evaluating index structures, there are several principal avenues of approach: analytical approach, mathematical modeling, simulations, and experiment [20]. We decided to do extensive experiments, because it is very difficult, if not impossible, to devise a formal model that yields reliable and precise results for non-uniform data distribution and average case behavior.

We implemented our multi-level hashing scheme (also abbreviated ML Hash Tree in the further text) along with extendible hashing (Ext. Hash) and the simple version of the extendible hashing tree (Ext. Hash Tree) and compared the performance of these access methods. We also compare our method qualitatively with Balanced Extendible Hash trees (BEH) by Otoo [15] and multi-level extendible hash trees by Du and Tong [3] later on in Section 6.

In the following sections we reveal the system parameters and the specification of our experiments. Following that we present the results from our experiments.

4.1 System Parameters

The experiments were conducted on a lightly loaded PC (1GHz Athlon processor) with 512MByte main memory running under Windows NT4.0 SP6. We implemented the data structures and algorithms of the index structures in C++ using the Borland C++ Compiler Version 5.5. The index is fully integrated into our database system SOD [1]. We used 16K plain pages to store the directories and buckets of the index structures. The buffer manager of SOD buffered up to 256MByte in main memory. The buffer size is rather small for today's machines, but we chose this size on purpose. It is unrealistic to assume that an index will get a large part of the total available memory as buffer. In DBMSs and Web-Servers the memory will have to be shared by many applications. We wanted to test the suitability of the indexes for secondary storage and not for main memory. If each index fits almost completely into main memory, we would just measure main memory performance.

The size of the page references was 32 bits, i.e. the maximal hash table depth on a page was equal to 12, which is equivalent to 2^12 entries on a directory page.

[1] http://sod.tneumann.de

4.2 Test Data

We used different sets of data to test our index structures. First, we generated synthetic hash keys with varying degrees of skew by creating bit vectors with a length of 64 bits. The state of each bit (0 or 1) was determined randomly with different distributions (see Table 1). Distribution 1 yields uniformly distributed hash keys, Distribution 4 the most skewed hash keys.

	Probability that bit is set to	
	1	0
Distribution 1 (unif)	0.5	0.5
Distribution 2 (40:60)	0.4	0.6
Distribution 3 (30:70)	0.3	0.7

Table 1: Different distributions for hash keys

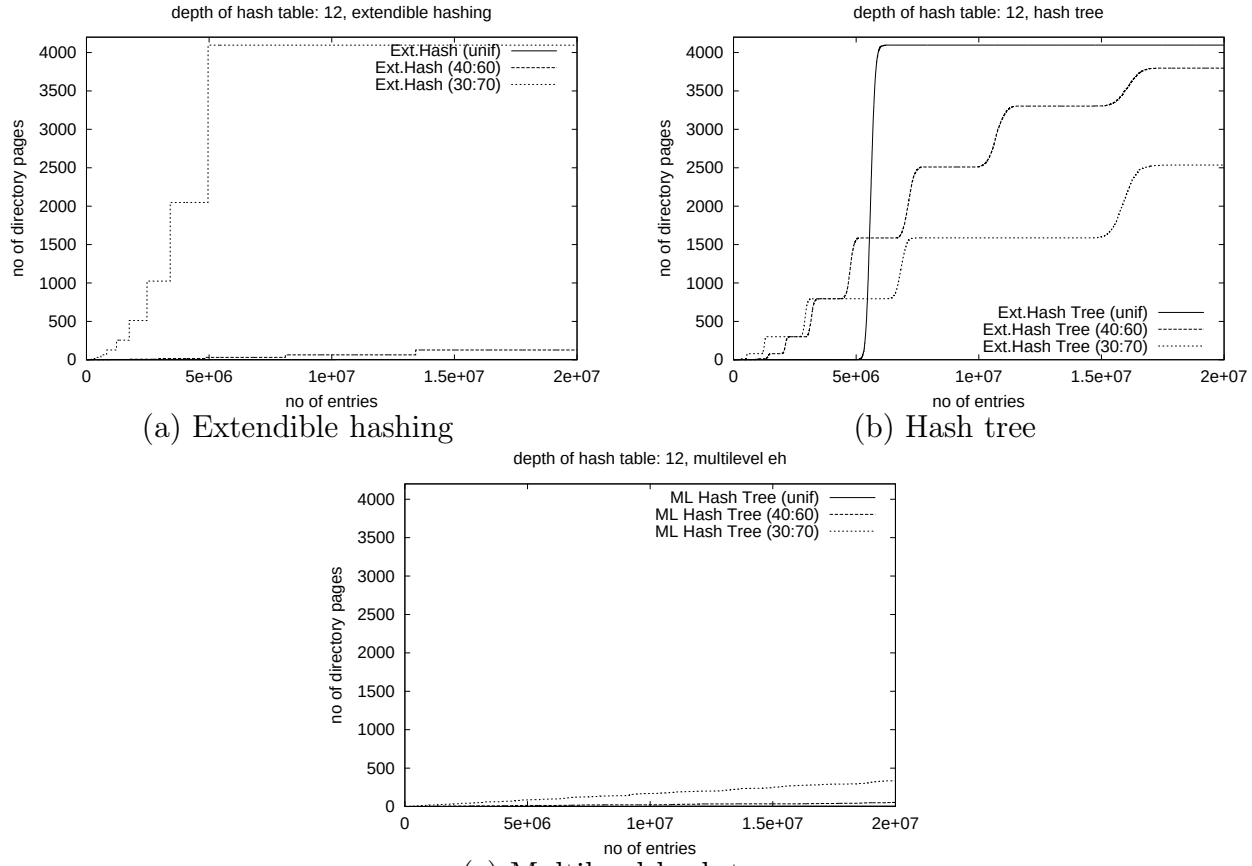
Second, we indexed a list of URLs generated by a Web crawler. An excerpt from this list can be seen here:
http://soe.stanford.edu/programs/research/civilenveng.html
http://soe.stanford.edu/programs/research/computerscience.html
http://www.berkeley.edu/services/index_text.html
http://learning.berkeley.edu/ugis.html
http://ls.berkeley.edu/ugis/masscomm/page1.html
http://ls.berkeley.edu/ugis/masscomm/page2.html
http://www.wisc.edu/wisconsinpress/
http://www.onwisconsin.com/
http://auctions.onwisconsin.com/
http://www.onwisconsin.com/scripts/staticpage.dll

The strings from the latter data set were hashed using the following hash function:

```
uint64 hash(const unsigned char* buffer)
{
   uint64 value=0;
   for (;*buffer;buffer++)
      value=((value<<6)|(value>>58))^(*buffer);
   return value;
}
```

5 Experimental Results

In our experiments we focussed on the two most important parameters: the size of the directory and the average lookup costs (elapsed time and page accesses). First we discuss the results for synthetically generated data, then those for real data.



(a) Extendible hashing

(b) Hash tree

(c) Multilevel hash tree

Figure 12: Size of directory in pages for varying data skew

	ext. hash		hash tree		ml hash tree	
skew	time (msec)	page acc.	time (msec)	page acc.	time (msec)	page acc.
uniform	5.91	2.00	6.74	3.00	6.01	3.00
40:60	5.91	2.00	8.04	3.00	5.58	3.00
30:70	9.01	2.10	6.51	3.01	5.86	3.24

Table 2: Average retrieval performance per hash key

5.1 Synthetic Data

In this section we present the results of the experiments for the synthetically generated hash keys (see also Section 4.2). We are particularly interested in the effect of skewed data on the index structures.

5.1.1 Size of Directory

One of the most important aspects is the size of the directory of the hashing schemes, as an exponentially growing directory leads to severe problems eventually. In Figure 12 we plotted the growing directory sizes depending on the number of inserted hash keys.

We observe in Figure 12(a) for the extendible hashing scheme that the more skewed the data, the fewer hash keys can be inserted before the directory has to be doubled again. Only the fact that we limit the growth of the directory to 4096 pages (we then use overflow buckets) keeps the directory size at a tolerable level. A skyrocketing directory size is the main reason extendible hashing is not used in practice. (The size for uniformly distributed hash keys is too small to be discernible in the figure.)

The simple hash tree (Figure 12(b)) exhibits a better behavior for heavily skewed hash keys, as the growth of the directory is not as steep as for extendible hashing. It is a different story for uniformly distributed hash keys, however. The flat parts of the curve for uniformly distributed hash keys show the periods of time in which the allocated hash tables are filled. When the hash tables become saturated eventually, the growth of the directory speeds up significantly, as a new level of hash tables is added (around 5,000,000 entries in Figure 12(b)). Extendible hashing is clearly superior to a simple hash tree for uniformly distributed hash keys, while for heavily skewed hash keys the opposite is true.

Nevertheless, the smallest directories are found in our multi-level hashing scheme (Figure 12(c), again the size for uniformly distributed hash keys is too small too be discernible). It has no trouble handling uniformly distributed hash keys, since in this case hash tables on lower levels utilize pages almost optimally. For skewed hash keys the directory size of our scheme also increases, but it does so gracefully. That means, the growth appears to be linear in the number of inserted hash keys rather than exponential.

5.1.2 Retrieval

Table 2 shows the retrieval performance of the different hashing schemes. The left column for each index structure contains the average running time in milliseconds for a lookup in a completely filled index (20,000,000 inserted hash keys). In each right column the average number of page accesses is displayed. We determined the numbers by averaging 20,000,000 random accesses to the hash tables.

Several observations are worth mentioning. The increase in page accesses for extendible hashing for heavily skewed data is due to the increased use of overflow buckets in this case. Sooner or later the performance of extendible hashing always deteriorates, as overflow buckets have to be used when the directory cannot be doubled anymore. As less and less pages of the index can be held in memory with increasing skewedness of the hash keys, we have more page faults during access, which results in a higher running time. Although our hashing scheme has the highest number of page accesses, its performance measured in run time is, on average, still better than that of the other schemes. Due to the smallest directory of all schemes, for our hashing scheme a smaller fraction of the page accesses are page faults, i.e., pages that are held in the main memory buffer need not be fetched from disk. This emphasizes the importance of a small directory, as it directly influences the retrieval performance.

5.2 Real Data

We were particularly interested in the performance of the index structures when faced with real data. The results of the experiments for URLs are described in this section. We looked at the same parameters as for synthetically generated data, namely size of directory and retrieval costs.

5.2.1 Size of Directory

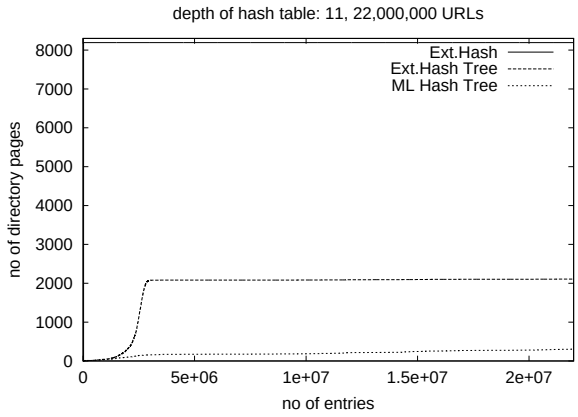


Figure 13: Size of directory in pages for URLs

When comparing the results in Figure 13 with those in Figure 12 for synthetic data, we clearly see that real data is far from uniformly distributed. (Note that the depth of the hash tables is 11, as we increased the pointer size from 32 to 64 bits.)

The directory of extendible hashing reaches the limit (set to 8192 pages this time) almost instantaneously. Indexing this kind of data with extendible hashing is out of the question. The extendible hash tree starts out strong, but deteriorates as more hash keys are inserted. It first allocates directory pages and then begins to fill them gradually. Ours seems to be the only index able to keep an almost linear growth.

5.2.2 Retrieval

	ext. hash		hash tree		ml hash tree	
	time (msec)	page acc.	time (msec)	page acc.	time (msec)	page acc.
URLs	12.37	2.38	6.95	3.41	6.08	3.41

Table 3: Average retrieval performance for URLs

The performance of the access methods for real data are summarized in Table 3. Overall, the retrieval costs for real data are very similar to those for skewed synthetic hash keys. Extendible hashing lags behind due to its use of overflow buckets and its large size. The gap is even wider than for synthetic data. On account of its small size, our scheme is faster than a hash tree in spite of a higher number of page accesses.

6 Further Comparisons with Other Approaches

Our approach ist not the first proposal to handle skewed hash keys. We are aware of two other developments, namely Balanced Extendible Hashing (BEH) by Otoo [15] and Multi-level Extendible Hashing bye Du and Tong [3]. In this section we discuss the shortcomings of these two methods and show why our hashing scheme is superior.

6.1 Balanced Extendbible Hashing (BEH)

BEH starts and expands like a regular extendible hashing scheme until the disk page containing the hash table is filled. Then if another bucket overflows that would make it necessary to double the directory once more, the directory is split into two directory pages. However, in order to permit order preserving hashing, the leading bit of the hash keys in the old directory is "shifted" onto a directory page one level higher (see Figure 14 for an illustration). A consequence of this technique is that the global depth of the right part of the hash table containing the hash keys with leading 1s (now on page P_2^1) decreases by one.

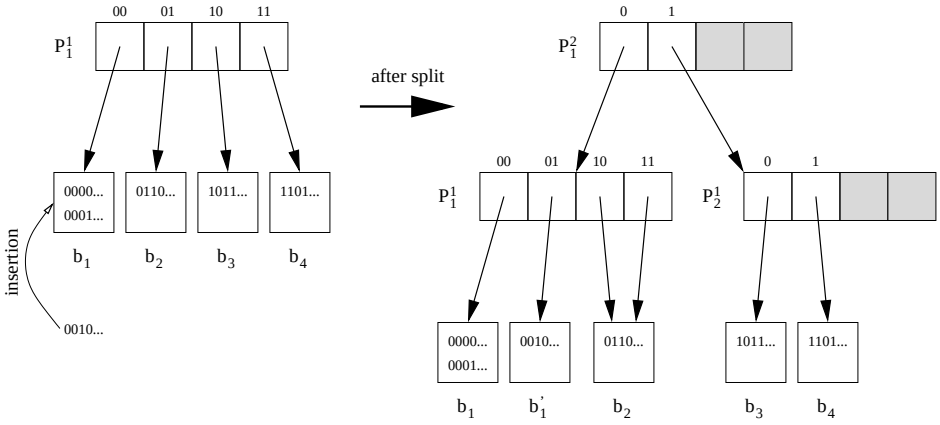


Figure 14: Splitting directory pages in BEH

This "shifting" of bits during a split can lead to very complicated situations. We will briefly sketch the associated problems, as an elaborate discussion is beyond the scope of this paper (a more detailed description of BEH can be found in [15]). Assume for a moment that bucket b_1 in Figure 14 overflows again. Consequently page P_1^1 on which the hash table pointing to b_1 resides has to be split again. After "shifting" the leading bit to the parent page P_1^2 we modify the global depths fo the split page and its parent page accordingly. But what do we do with other hash tables that are pointed to by the parent page? Also "shifting" bits from these hash tables could mean a total rebuild of the entire directory in the worst case. In order to avoid this rebuild Otoo also stores a local depth with each pointer that tells us at how many bits of the hash key we have to look at on the current level (see Figure 15).

So, for example, if we want to look up the hash key $1010\ldots$, we would access the third entry of the hash table on page P_1^2. The local depth of 1 means that we have to look up the key $010\ldots$ in the hash table on page P_2^1 leading us to bucket b_3.

After this short introduction to BEH, let us now evaluate this method. A BEH scheme has to store additional information (in the form of local depths) with each pointer. Assuming that pointers and page sizes are powers of 2, we have twice the fanout of BEH on our directory pages, as we do not have to store any additional information.

Another disadvantage of BEH compared to our approach is the fact that all buckets are on the same level. When accessing a bucket the full height of the tree has to be traversed, regardless of the accessed data. In our scheme only the most skewed hash keys can be found on the lowest level of the tree. All other buckets are placed at higher

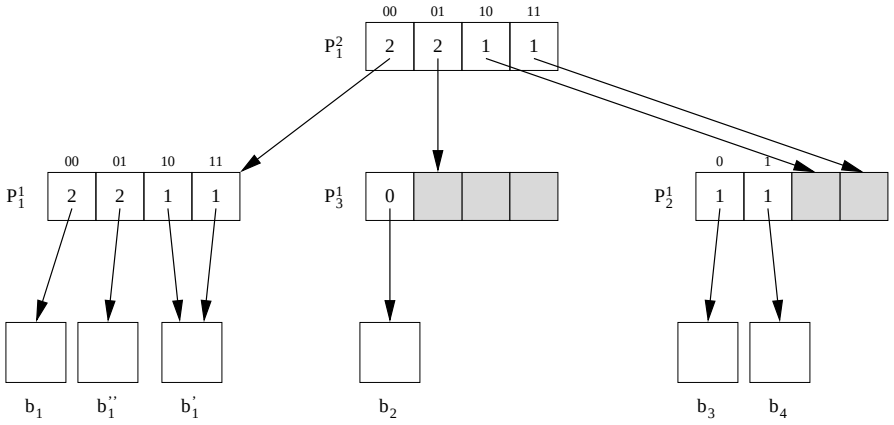


Figure 15: Further splitting of directory pages in BEH

levels. Inserting the same data from the example above into our scheme would result in the structure depicted in Figure 16.

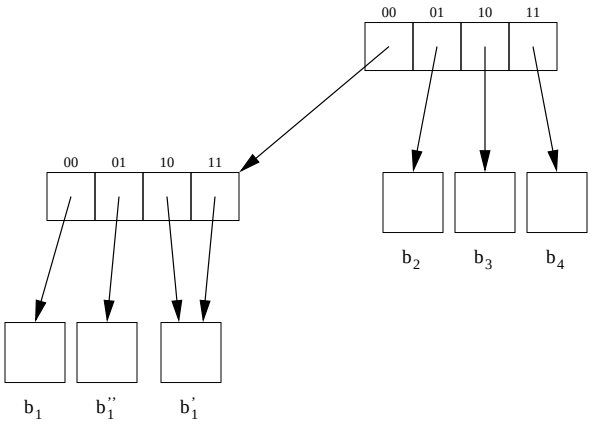


Figure 16: Our scheme

We can reach the buckets b_2, b_3, and b_4 with one additional directory access, only for the buckets b_1, b_1', and b_1'' we would need two additional directory accesses. (When the size of pointers and disk pages are a power of 2, our scheme would be even better due to the increased fanout.) Hence the maximal number of page accesses needed in our scheme is lower or equal to the warranted number of page accesses in BEH.

Given an order preserving hash function range searching is also easier to accomplish in our scheme, as we can link the buckets and scan through them without further directory accesses. Linking buckets in BEH would cost performance during insertion, as (opposed to our scheme) empty leaf nodes in the directory may exist. This makes it more difficult to find neighboring buckets when finally inserting data items into these leaf nodes (upward and downward traversal in the directory without prior knowledge of the needed number of steps is needed in this case). Therefore Otoo proposes to link the leaf nodes in a BEH scheme.

6.2 Du and Tong's Approach

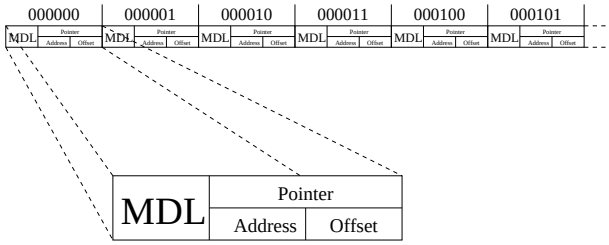


Figure 17: Linear (top-level) directory in Du and Tong's approach

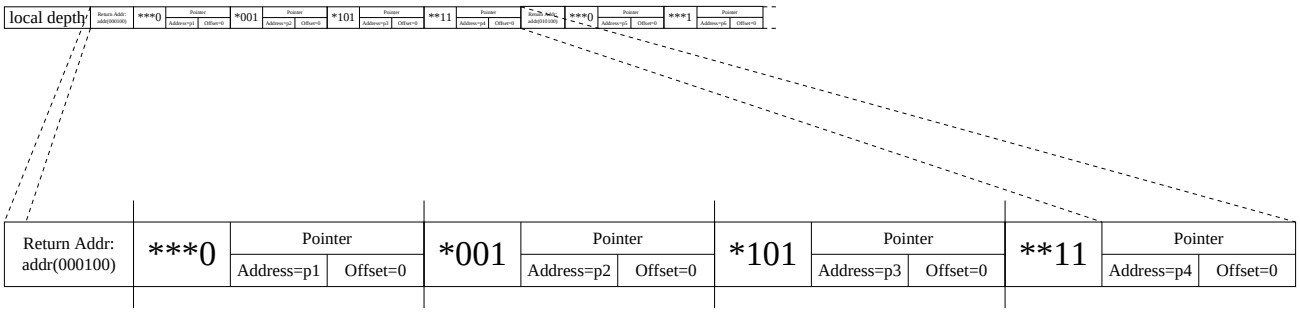


Figure 18: Lower directory pages in Du and Tong's approach

In Du and Tong's scheme the first directory page on a given level is shared by all pages that are one level higher. If this page has to be split, Du and Tong also use a buddy principle to determine how the entries of this page are assigned to the new pages. Du and Tong use different pages for the top-level directory than for other directory pages. Figure 17 illustrates the format of a top-level directory page (called *linear directory* by Du and Tong). An entry in a top-level page consists of a MDL (Maximum Directory Level) counter and a pointer. The pointer comprises a page address and an offset in that page. In contrast to that, our scheme only needs to store page addresses in a directory page, as we can calculate the offset due to the regular structure of our scheme. The MDL counter indicates the lowest level of directory pages reached by the pointer in this entry. These counters are used to check for pages that can be shared on a certain level to deal with special cases during insertion. We do not need such a counter, as page sharing follows rigorous rules in our scheme. As page addresses and page sizes are usually a power of 2, we have twice the fanout in the top-level directory as Du and Tong.

Figure 18 shows the structure of a lower-level directory page in Du and Tong's scheme. The *local depth* indicates the possible range of entries at the previous level sharing this page. For example, a local depth of 3 would mean that all hashed items whose hash key starts with 100 can share this page (note that Du and Tong start with the least significant bit). We, however, do not need store a local depth on a directory page, the depths of hash tables on one of our directory pages can be deduced by looking at the parent page. Furthermore, each entry in a directory page in Du and Tong's scheme contains a

Return Address field to identify the associated item pointing to the entry. The subentries belonging to a Return Address are composed of a label (identifying the bits of the hash key that are relevant) and a pointer made up of a page address and an offset. The Return Address is not only needed to identify the subentries, but to allow the restructuring of the hash scheme. The reason for this is that a split of a directory page may cause further splits on levels above (up to the top-level directory), so Du and Tong have to be able to traverse and modify all directory pages on the path to the top-level directory. By contrast, if a split occurs in a directory page in our scheme, only the pointers in the parent page of the split page are affected. All other directory pages above it are already filled to the maximum and cannot split further. So on our way down the directory during an insertion, we only need to buffer the parent page.

Moreover, the technique adopted by Du and Tong to compress subentries in directory pages will only pay off for skewed data. For uniformly distributed data, all entries will expand about evenly leading to a reduced fanout when compared to the other schemes, because of the overhead. Our scheme handles skewed data by expanding only the part of the hash table that is affected by the skew. The other parts are folded back up again. Uniformly distributed data will lead to highly utilized pages with a superior fanout for us.

7 Conclusion and Outlook

We developed a robust extendible hashing index able to efficiently handle skewed data. We implemented the hashing scheme and tested it thoroughly comparing it to several other existing access methods. Our approach proved to be superior in view of directory size as well as retrieval performance, especially for real data. The size of our directory does not grow exponentially. Consequently our index is also faster, because a larger fraction of the directory can be held in main memory.

The results obtained from our experiments encourage us to extend the scheme to allow processing of multi-attribute queries and partial-match retrieval. Another important subject is the creation time of the index. The current version of creating the index is not efficient enough. So a next step would be to devise efficient bulk-loading procedures to reduce the creation time significantly.

Acknowledgments

We thank Thorsten Fiebig for crawling millions of URLs for us.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[2] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[3] D.H.C. Du and S.-R. Tong. Multilevel extendible hashing: A file structure for very large databases. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):357–370, 1991.

[4] R.J. Enbody and H.C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113, June 1988.

[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):315–344, September 1979.

[6] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20(4):345–369, 1983.

[7] P.A. Larson. Dynamic hashing. *BIT*, 18:184–201, 1978.

[8] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of the 6th VLDB Conference*, pages 212–223, Montreal, 1980.

[9] W. Litwin. Trie hashing. In *Proc. of the 1981 ACM SIGMOD*, pages 19–29, Ann Arbor, Michigan, 1981.

[10] W. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. *Transactions on Software Engineering (TSE)*, 17(2):678–691, July 1991.

[11] W. Litwin, D. Zegour, and G. Levy. Multilevel trie hashing. In *EDBT '88, International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 309–335, Venice, Italy, March 1988. Springer-Verlag.

[12] D.B. Lomet. Bounded index exponential hashing. *ACM Transactions on Database Systems*, 8(1):136–165, March 1983.

[13] Y. Manolopoulos, Y.Theodoridis, and V.J. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publishers, Dordrecht, 1999.

[14] H. Mendelson. Analysis of extendible hashing. *IEEE Trans. Software Eng.*, 8(6):611–619, November 1982.

[15] E.J. Otoo. Linearizing the directory growth in order preserving extendible hashing. In *Int. Conf. on Data Engineering*, pages 580–588, 1988.

[16] E.J. Otoo and S. Effah. Red-black trie hashing. Technical Report TR-95-03, Carleton University, Ottawa, Canada, 1995.

[17] M.V. Ramakrishna and P.A. Larson. File organization using composite perfect hashing. *ACM Transactions on Database Systems*, 14(2):231–263, June 1989.

[18] K. Ramamohanarao and J.W. Lloyd. Dynamic hashing schemes. *The Computer Journal*, 25(4):478–485, 1982.

[19] M. Tamminen. Order preserving extendible hashing and bucket tries. *BIT*, 21:419–435, 1981.

[20] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *ACM SIGMOD Record*, 25(3):10–15, September 1996.