Indexing Set-Valued Attributes with a Multi-level Extendible Hashing Scheme

Sven Helmer[1], Robin Aly[2], Thomas Neumann[3], and Guido Moerkotte[4]

[1] University of London, United Kingdom
[2] University of Twente, The Netherlands
[3] Max-Planck-Institut für Informatik, Germany
[4] University of Mannheim, Germany

Abstract. We present an access method for set-valued attributes that is based on a multi-level extendible hashing scheme. This scheme avoids exponential directory growth for skewed data and thus generates a much smaller number of subqueries for query sets (so far fast-growing directories have prohibited hash-based index structures for set-valued retrieval). We demonstrate the advantages of our scheme over regular extendible hashing both analytically and experimentally. We also implemented a prototype and briefly summarize the results of our experimental evaluation.

1 Introduction

Efficiently retrieving data items with set-valued attributes is an important task in modern applications. These queries were irrelevant in the relational context since attribute values had to be atomic. However, newer data models like the object-oriented (or object-relational) models support set-valued attributes, and many interesting queries require a set comparison. An example would be to find persons who match a job offering. In this case the query set *required-skills* is a subset of the persons' set-valued attribute *skills*. Note that we assume to work on a large number of objects, but with limited set cardinality. We believe that this is the most common case found in practice. This belief is backed by our observations on real applications for object-oriented or object-relational databases (as found, for example, in product and production models [6] and molecular databases [21]).

One way to support the efficient evaluation of queries is by employing index structures. Hash-based data structures are among the most efficient access methods known, allowing retrieval in nearly constant time. Nevertheless, when applying hash-based techniques to set-valued retrieval on secondary storage we have to meet two main challenges. As it is too expensive to completely reorganize hash tables on secondary storage, dynamic hashing schemes, like linear hashing [15] and extensible hashing [4], are used. However, dynamic hashing schemes exhibit exponentially growing directory sizes on skewed data. (Even if the employed hash function works reasonably well, it cannot offset the effect of multiple copies of certain sets.) Moreover, evaluating set-valued queries on hash tables is difficult: in order to access all subsets/supersets of a query set, we have to generate all possible subsets/supersets of the query set and probe the hash table with

R. Wagner, N. Revell, and G. Pernul (Eds.): DEXA 2007, LNCS 4653, pp. 98–108, 2007.
© Springer-Verlag Berlin Heidelberg 2007

them. Obviously, in the average case this will have an exponential running time. However, many of the generated sets are redundant, as the respective entries in the directory of the hash table point to the same (shared) buckets or are empty.

We propose a dynamic multi-leveled hashing scheme to remedy this situation. As we have shown in [10], this hashing scheme can handle skewed data much better than existing schemes. Here we focus on adapting this index structure to retrieving data items with set-valued attributes efficiently. We demonstrate that hash-based schemes are a viable approach to indexing set-valued attributes.

The remainder of this work is organized as follows. The following section describes related work and the context of our work. We give a brief introduction to superimposed coding and show how to apply signatures to set-valued retrieval in Section 3. Section 4 contains a short description of our (regular) multi-level hashing scheme, while Section 5 describes how this scheme is adapted to set-valued retrieval. In Section 6 we summarize the results of our experimental evaluation. Section 7 concludes the paper.

2 Related Work

Work on the evaluation of queries with set-valued predicates is few and far between. Several indexes dealing with special problems in the object-oriented [2] and the object-relational data models [18] have been invented, e.g. nested indexes [1], path indexes [1], multi indexes [16], access support relations [13], and join index hierarchies [22]. These index structures focus on evaluating path expressions efficiently.

One of the dominant techniques for indexing set-valued attributes is superimposed coding, where sets are represented by bit vector signatures. Existing techniques for organizing signatures include: sequential files [12], hierarchical organization (signature trees [3], Russian Doll Trees [7]), and partitioning (S-tree split [19], hierarchical bitmap index [17]).

At first glance, methods from text retrieval appear to be similar to set retrieval. However, text retrieval methods (like [23]) focus on partial-match retrieval, that is, retrieving supersets of the query set. Set retrieval also supports subset and exact queries, which are relevant and common for example in molecular databases (e.g. searching for characteristic parts of a large molecule).

3 Preliminaries

3.1 Querying Set-Valued Attributes

Let us assume that our database consists of a finite set O of data items o_i ($1 \leq i \leq n$) having a finite set-valued attribute A with a domain D. Let $o_i.A \subseteq D$ denote the value of the attribute A for some data item o_i. A *query predicate* P consists of a set-valued attribute A, a finite *query set* $Q \subseteq D$, and a *set comparison operator* $\theta \in \{=, \subseteq, \supseteq\}$. A query of the form $\{o_i \in O | Q = o_i.A\}$ is called an *equality query*, a query of the form $\{o_i \in O | Q \subseteq o_i.A\}$ is called a *subset*

query, and a query of the form $\{o_i \in O | Q \supseteq o_i.A\}$ is called a *superset query*. Note that *containment queries* of the form $\{o_i \in O | x \in o_i.A\}$ with $x \in D$ are equivalent to subset queries with $Q = \{x\}$.

3.2 Signature-Based Retrieval

Superimposed coding is a method for encoding sets as bit vectors. It uses a coding function to map each set element to a bit field of length b (b is the *signature length*) such that exactly $k < b$ bits are set. The code for a set (also known as the set's *signature*; abbreviated as *sig*) is the *bitwise or* of the codes for the set elements [5,14].

The following properties of signatures are essential (let s and t be two arbitrary sets):

$$s \; \theta \; t \Longrightarrow \mathrm{sig}(s) \; \theta \; \mathrm{sig}(t) \;\; \text{for } \theta \in \{=, \subseteq, \supseteq\} \tag{1}$$

where $\mathrm{sig}(s) \subseteq \mathrm{sig}(t) := \mathrm{sig}(s) \&\tilde{\ }\mathrm{sig}(t) = 0$ and $\mathrm{sig}(s) \supseteq \mathrm{sig}(t) := \mathrm{sig}(t) \&\tilde{\ }\mathrm{sig}(s) = 0$ (& denotes *bitwise and* and ˜ denotes *bitwise complement*).

As set comparisons are very expensive, using signatures as filters is helpful. Before comparing the query set Q with the set-valued attribute $o_i.A$ of a data item o_i, we compare their signatures $sig(Q)$ and $sig(o_i.A)$. If $sig(Q) \; \theta \; sig(o_i.A)$ holds, then we call o_i a drop. If additionally $Q \; \theta \; o_i.A$ holds, then o_i is a *right drop*; otherwise it is a *false drop*. We have to eliminate the false drops in a separate step. However, the number of sets we need to compare in this step is drastically reduced as only drops need to be checked.

There are three reasons for using signatures to encode sets. First, they are of fixed length and hence very convenient for index structures. Second, set comparison operators on signatures can be implemented by efficient bit operations. Third, signatures tend to be more space efficient than explicit set representation.

4 Multi-level Hashing

As in other dynamic hashing schemes (e.g. [4,15]), our multi-level hashing index (MLH index) is divided into two parts, a directory and buckets. 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_g of g bits of a hash key h. Let us take a look at a non-hierarchical hashing scheme first. It has a directory with 2^g entries, where g is called the *global depth* of the hash table. The prefix h_g identifies one of these entries and we follow the link in this entry to access the corresponding bucket.

On the other hand, in our MLH index things are done differently. We also check the prefix of a hash key to find the right bucket, but the length of the prefix that we check may vary depending on the level in the directory where we finally find the correct bucket (our hashing scheme is not necessarily balanced).

4.1 General Description

Due to space constraints, we can only give a brief description; for details see [9,10]. We employ a multi-level extendible hash tree in which hash tables share pages according to a buddy scheme. In this buddy scheme, z-*buddies* are hash tables that reside on the same page and whose stored hash keys share a prefix of z bits. Consequently, all buddy hash tables in our tree have the same global depth z.

Let us illustrate our index with an example. We assume that a page can hold 2^n entries of a hash table directory. Furthermore, we assume that the top level hash table directory (also called the root) is already filled, contains 2^n different entries at the moment, and that another overflow occurs (w.l.o.g. in the first bucket). In this case, we allocate a new hash table of global depth 1 (beneath the root) to distinguish the elements in the former bucket according to their $(n+1)$st bit. However, we do this not only for the overflowing bucket, but also for all 1-buddies of this bucket. 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 1.

In a naive hierarchical hash tree, we would have allocated just one hash table with depth n for the overflowed bucket. If other buckets overflow, we allocate new recursive hash tables for them as well. The main problem with naive hash trees is waste of memory: almost all entries in these newly allocated hash tables share the same buckets, i.e. we do not need a directory with depth n yet. At first glance our scheme does not seem that much different, as we also allocate a whole page. However, due to the data skew we expect splits near buckets that have already split. Even when the anticipated splits do not occur, we can eliminate unnecessary directory pages.

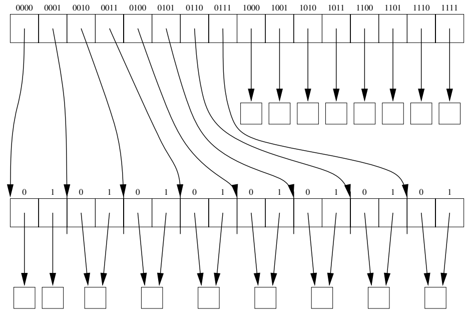

Fig. 1. Overflow in our multi-level hash tree

If another overflow occurs in one of the hash tables on level 2, causing it to grow, 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

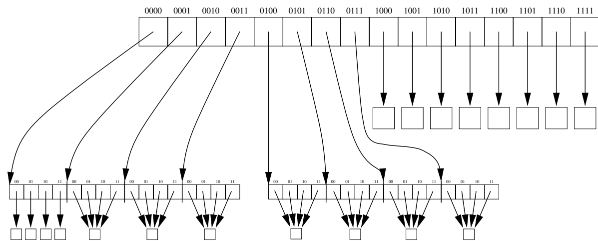

Fig. 2. Overflow on the second level

```
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++;
  }
}
```

(a) Pseudocode for lookups

```
lookup(hashkey, buddies, curdepth, localwidth) {
  for all nodes n in buddies {
    subhash = hashkey[b-curdepth...b-depth-width];
    for all sub-/supersets s of subhash {
      pos = buddyoffset(n) + s;
      nodeId = slot[pos];
      if(nodeId is not a null pointer) {
        if(node is an unmarked bucket) {
          scan(node);
          add content to answer;
          mark(node);
        }
        else {
          add nodeId and localwidth to children;
        }
      }
    }
  }
  for all c in children {
    lookup(hashkey, c.nodeId,
           curdepth+localwidth, c.localwidth);
  }
}
```

(c) Pseudocode for looking up sub-/supersets

```
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;
      }
      insert new level;
      insert(hashkey, value);
      return;
    }
    currentLevel++;
  }
}
```

(b) Pseudocode for insertions

Fig. 3. Pseudo-code for multi-level hashing

content that does not fit to a new page. Then we adjust the pointers in the parent directory. The left half of the pointers referencing the original page still point to this page, the right half to the new page (see Figure 2).

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 2 is 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 on the next higher level.

Due to our buddy scheme, we have a very regular structure that can be exploited. Indeed, we can compute the global depths of all hash tables (except the root) by looking at the pointers in the corresponding parent table. Finding 2^{n-i} identical pointers there means that the referenced page contains 2^{n-i} i-buddies of global depth i. Consequently, we can utilize the whole page for storing pointers, as no additional information has to be kept.

4.2 Lookups

Lookups are easily implemented (for the pseudocode see Figure 3(a)). 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 shift the last $n-i$ bits of the relevant pointer in the parent table by the size of a hash table on the shared page. 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. If the bucket does not exist (no data item is present there at the moment), we hit a NULL-pointer and can abort the search.

4.3 Insertions

After finding the bucket where the new data item has to be inserted (using the lookup procedure), we have to distinguish several cases for inserting the new item (for the pseudocode see 3(b)). We concentrate on the most difficult case, where an overflow of the bucket occurs and the global depth of the hash table on the current level increases. The other cases can be handled in a straightforward manner.

If the hash table has already reached its maximal global depth (i.e. it resides alone on a page), we add a new level with 2^{n-1} hash tables of global depth 1 to the existing index structure (comparable to Figure 1). If we have not reached the maximal global depth yet (i.e. the hash table shares a page with its buddies), the global depth of all hash tables on this page is increased by 1. The hash tables on the first half of the page remain there. The hash tables on 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 present yet). In this case (compare the node in the lower right corner of Figure 2) we do not need this node yet and connect the buckets directly to the parent hash table.

5 Adapting ML-Hashing to Set-Valued Queries

Using a (non-hierarchical) hashing scheme in a naive way to evaluate a set-valued query is quite straightforward. All the hashing keys employed in our

scheme are made up of signatures encoding sets. When processing a query we first determine the signature of the query set via superimposed coding. Depending on the type of the query (subset or superset query) we generate all supersets or all subsets of the query signature's prefix h_g and initiate subqueries with all of these generated sets. When we reach a bucket, we compare the full query signature to all signatures stored there to decide whether to access a data item or not. For our multi-level hashing scheme we generate the relevant supersets and subsets of the query signature on demand on each level of the data structure. If we encounter buckets on our way down we also compare the full query signature to the signatures stored in each bucket. Figure 3(c) shows this algorithm in pseudocode (parameters for the lookup function are the hashkey, a set consisting of the root node, current depth 0, and the local width of the root table). For insertions the same code as in Figure 3(b) is used.

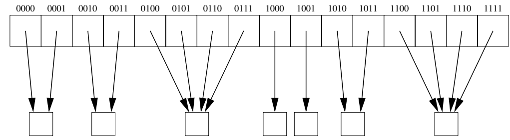

Fig. 4. Accessing a non-hierarchical hash table

5.1 Example

The following example demonstrates the difference between non-hierarchical hashing schemes and our multi-level approach. Let A be a non-hierarchical hash table with a global depth of four. We wish to obtain the supersets of our query set Q with signature $\text{sig}(Q) = 001011101110$. The relevant prefix of $\text{sig}(Q)$ is 0010, and for A we must now generate all eight superset prefixes, namely 0010, 0011, 0110, 1010, 0111, 1011, 1110, and 1111. Thus, for a non-hierarchical hash table, we must start eight subqueries to access three of the seven buckets (see also Figure 4).

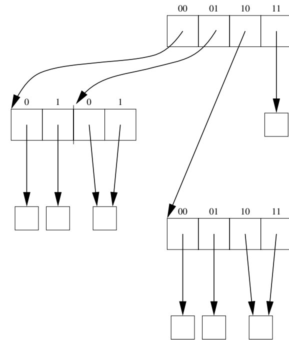

Fig. 5. Accessing a multi-level hash table

For our multi-level hashing approach, on the other hand, we begin by generating only the top level supersets 00, 01, 10, and 11 and then the superset 1 for the hash table on the left-hand side of the second level, followed by the supersets 10 and 11 for the hash table on the right-hand side of the second level (see also Figure 5). Thus, we need to generate only seven rather than eight supersets; at first glance, this may not seem like huge savings, but the next section will show that the savings grow when the tables are larger.

5.2 Comparison of ML-Hashing with Regular Extendible Hashing

If skewed data is inserted into a hash table, the directory of a non-hierarchical hashing scheme grows exponentially. This is bad news for the naive method of generating all subset or supersets, as on average we have to generate

$$\frac{2^{\lfloor \frac{g}{2} \rfloor} + 2^{\lceil \frac{g}{2} \rceil}}{2} \tag{2}$$

signatures (including the original prefix of sig(Q)).[1] For large values of g this is clearly infeasible. The worst thing is that most of these signatures are generated needlessly. Hash tables containing skewed data look a lot like the one depicted in Figure 4. In this example sixteen entries share seven buckets, which means that most of the subqueries will access the same buckets over and over again.

How do we cope with this situation? First of all, our MLH index can handle skewed data much better than other dynamic hashing schemes resulting in a much smaller directory. Summarizing the results from [9,10], in which we have substantiated our claim experimentally, we can say that the main idea is to unbalance the hierarchical directory of our hash table on purpose. We did this because obviously we are unable to change the fact that skewed data has been inserted into our hash table, meaning that we have many data items on our hands whose hash keys share long prefixes. In order to distinguish these data items we need a hash table with a large depth. However, we want to make sure that other data items are not "punished" for this. Second, when generating subsets and supersets of query signatures while evaluating set-valued queries, we do not generate them en bloc for the whole prefix. Instead, we generate the appropriate subsets and supersets for each level separately. On each level we have hash tables with a maximum depth of n, so we have to generate $\frac{2^{\lfloor \frac{n}{2} \rfloor} + 2^{\lceil \frac{n}{2} \rceil}}{2}$ signatures on average. We have to do this for each level we look at. Let us assume that the largest prefix we distinguish in our MLH index is g. Then we generate

$$\frac{2^{\lfloor \frac{n}{2} \rfloor} + 2^{\lceil \frac{n}{2} \rceil}}{2} \cdot \lfloor \frac{g}{n} \rfloor + \frac{2^{\lfloor \frac{g \bmod n}{2} \rfloor} + 2^{\lceil \frac{g \bmod n}{2} \rceil}}{2} \tag{3}$$

signatures in the average case.[2]

Formula (3) does not yet consider that we can have hash tables with different depths on the same level in our directory. If the left page on the second level in Figure 2 were to split again, this would result in two pages containing two hash tables with depth three each. The other page on the second level is unaffected by this, still keeping its four hash tables with depth two. So in the worst case we have to generate signatures for each depth up to n on each level (except the first; if $g < n$ use Formula (2)):

[1] Here we assume that on average half of the bits in a signature are set to 0 and half are set to 1. This is the case if the parameters b and k (the size of a signature and the number of set bits per hash value) have been optimized correctly.

[2] If we traverse all levels of the directory.

$$\frac{2^{\lfloor \frac{n}{2} \rfloor} + 2^{\lceil \frac{n}{2} \rceil}}{2} + \left(\sum_{i=1}^{n} \frac{2^{\lfloor \frac{i}{2} \rfloor} + 2^{\lceil \frac{i}{2} \rceil}}{2} \right) \cdot \lfloor \frac{g-n}{n} \rfloor + \sum_{i=1}^{g \bmod n} \frac{2^{\lfloor \frac{i}{2} \rfloor} + 2^{\lceil \frac{i}{2} \rceil}}{2} \quad (4)$$

For a closed-form formula of (4) see our technical report [8]. Figure 6 compares the number of generated signatures for our hierarchical directory versus a non-hierarchical directory. As can be clearly seen, the curves for the hierarchical directories break away at some point from the exponentially growing curve for non-hierarchical directories. This happens when the top-level directory page reaches n, the maximum depth of the hierarchical hash tables.

In summary we can say that our MLH index is suited better for set-valued retrieval than other hash-based indexes, because it does not need exponential running time for generating the subqueries and it is able to cope better with data skew.

6 Summary of Experimental Evaluation

Due to space constraints, we can only give a summary of the experimental evaluation here. For a detailed description see [8].

MLH clearly shows the best behavior among all the index structures we compared it to: a sequential signature scan [12], an extensible signature hashing scheme [11], and two S-tree approaches [20] (one with a linear splitting and one with a quadratic splitting algorithm). It is best both in terms of the number of page accesses and total running time when evaluating subset queries. While for uniformly or mildly skewed data, ESH achieves a performance comparable with that of MLH, the drawbacks of ESH become apparent when the data is heavily skewed: in that case, ESH suffers due to directory growth and the exponential cost of generating subqueries. The scanning methods (SIGSCAN and SETSCAN) which have mainly been added as a reference are not able to compete with MLH either. The big surprise is the hierarchical S-tree index structure. In contrast to the results presented in [20] we show that the tree-based access methods are not suitable for indexing set-valued attributes, because they do not scale - the prevalence of all-1

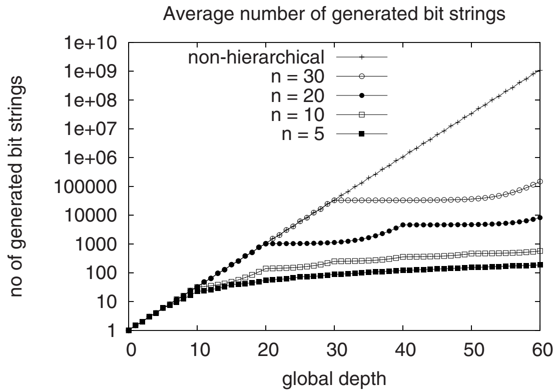

Fig. 6. Reducing the number of generated sets

nodes nullifies the inner nodes' filtering capacity. The superiority of hash-based schemes for equality queries does not come as a big surprise, since point queries are the strong point of hash table approaches. We have demonstrated that even in terms of index size, MLH copes extremely well with skewed data: unlike ESH the directory does not grow exponentially. Instead, the growth is linear, much as for lightly skewed or uniformly distributed data.

7 Conclusion

We presented the first secondary-storage, hash-based access method for indexing set-valued attributes that is able to outperform other index structures for set retrieval. Until now the fast directory growth of hash-based schemes has prevented their use for evaluating queries with subset and superset queries, as the number of subqueries that had to be submitted was exponential in the size of the directory. Our approach generates a number of subqueries linear in the global depth of the hash table. We demonstrated the competitiveness of our index structure analytically (and experimentally).

Although superimposed coding and dynamic hashing schemes have attracted some attention when they first appeared, they were not able to make their way into industrial strength database systems. One of the main reasons was their susceptibility to skewed data, which robust, data-driven index structures like B^+-trees were able to handle much better. Our multi-level hashing scheme represents an interesting compromise between data-driven and space-driven data structure and could renew the interest in hash-based, superimposed coding schemes.

References

1. Bertino, E., Kim, W.: Indexing techniques for queries on nested objects. IEEE Trans. on Knowledge and Data Engineering 1(2), 196–214 (1989)
2. Cattell, R. (ed.): The Object Database Standard: ODMG 2.0. Morgan Kaufmann, San Francisco (1997)
3. Deppisch, U.: S-tree: A dynamic balanced signature index for office retrieval. In: Proc. of the 1986 ACM Conf. on Research and Development in Information Retrieval, Pisa (1986)
4. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing – a fast access method for dynamic files. ACM Transactions on Database Systems 4(3), 315–344 (1979)
5. Faloutsos, C., Christodoulakis, S.: Signature files: An access method for documents and its analytical performance evaluation. ACM Transactions on Office Informations Systems 2(4), 267–288 (1984)
6. Grobel, T., Kilger, C., Rude, S.: Object-oriented modelling of production organization. In: Tagungsband der 22. GI-Jahrestagung, Karlsruhe, September 1992, Springer, Heidelberg (1992)
7. Hellerstein, J.M., Pfeffer, A.: The RD-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison (1994)

8. Helmer, S., Aly, R., Neumann, T., Moerkotte, G.: Indexing Set-Valued Attributes with a Multi-Level Extendible Hashing Scheme. Technical Report BBKCS-07-01, Birkbeck, University of London, http://www.dcs.bbk.ac.uk/research/techreps/2007/
9. Helmer, S., Neumann, T., Moerkotte, G.: A robust scheme for multilevel extendible hashing. Technical Report 19/01, Universität Mannheim (2001), http://pi3.informatik.uni-mannheim.de
10. Helmer, S., Neumann, T., Moerkotte, G.: A robust scheme for multilevel extendible hashing. In: Yazıcı, A., Şener, C. (eds.) ISCIS 2003. LNCS, vol. 2869, pp. 220–227. Springer, Heidelberg (2003)
11. Helmer, S., Moerkotte, G.: A performance study of four index structures for set-valued attributes of low cardinality. VLDB Journal 12(3), 244–261 (2003)
12. Ishikawa, Y., Kitagawa, H., Ohbo, N.: Evaluation of signature files as set access facilities in OODBs. In: Proc. of the 1993 ACM SIGMOD, Washington, pp. 247–256. ACM Press, New York (1993)
13. Kemper, A., Moerkotte, G.: Access support relations: An indexing method for object bases. Information Systems 17(2), 117–146 (1992)
14. Knuth, D.E.: The Art of Computer Programming. In: Sorting and Searching, Addison Wesley, Reading, Massachusetts (1973)
15. Larson, P.A.: Linear hashing with partial expansions. In: Proc. of the 6th VLDB Conference, Montreal, pp. 224–232 (1980)
16. Maier, D., Stein, J.: Indexing in an object-oriented database. In: Proc. of the IEEE Workshop on Object-Oriented DBMSs, Asilomar, California (September 1986)
17. Morzy, M., Morzy, T., Nanopoulos, A., Manolopoulos, Y.: Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS 2003. LNCS, vol. 2798, pp. 236–252. Springer, Heidelberg (2003)
18. Stonebraker, M., Moore, D.: Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann, San Francisco (1996)
19. Tousidou, E., Bozanis, P., Manolopoulos, Y.: Signature-based structures for objects with set-valued attributes. Information Systems 27(2), 93–121 (2002)
20. Tousidou, E., Nanopoulos, A., Manolopoulos, Y.: Improved methods for signature-tree construction. The Computer Journal 43(4), 301–314 (2000)
21. Will, M., Fachinger, W., Richert, J.R.: Fully automated structure elucidation - a spectroscopist's dream comes true. J. Chem. Inf. Comput. Sci. 36, 221–227 (1996)
22. Xie, Z., Han, J.: Join index hierarchies for supporting efficient navigation in object-oriented databases. In: Proc. Int. Conf. on Very Large Data Bases (VLDB), pp. 522–533 (1994)
23. Zobel, J., Moffat, A., Ramamohanarao, K.: Inverted files versus signature files for text indexing. Technical Report CITRI/TR-95-5, Collaborative Information Technology Research Institute (CITRI), Victoria, Australia (1995)