Reihe Informatik
2 / 1999

A Study of Four Index Structures for Set-Valued Attributes of Low Cardinality

Sven Helmer   Guido Moerkotte

A Study of Four Index Structures for Set-Valued Attributes of Low Cardinality

Sven Helmer Guido Moerkotte

Lehrstuhl für Praktische Informatik III
Universität Mannheim
68131 Mannheim
Germany

helmer|moerkotte@informatik.uni-mannheim.de
Phone: +49 (621) 292 8819 Fax: +49 (621) 292 8818

Abstract

We review and study the performance of four different index structures for indexing set-valued attributes designed to speed up set equality, subset and superset queries. All index structures are based on traditional techniques, namely signatures and inverted files. More specifically, we consider sequential signature files, signature trees, extendible signature hashing, and a B-tree based implementation of inverted lists. The latter is refined by a compression scheme in order to keep space requirements within acceptable bounds. The performance study is based on real implementations subjected to a benchmark accounting for different set sizes, domain sizes, and data distributions (uniform and skewed).

1 Introduction

Will such venerable tools as inverted files and signature files stand up to the job of retrieving sets efficiently? We think that the answer is yes and show how these traditional techniques, when modified to suit the needs, speed up set-retrieval tremendously. We illustrate our approach for small sets and back up our claims with results from our extensive benchmarks.

Imagine a database containing the recipes of various cocktails. Each recipe is stored in a data item r_i with a set-valued attribute called *ingredients*. If the resources of your home bar are depleted, a typical query would be: “Given the ingredients ’Scotch’, ’Vermouth’, and ’lemon juice’, which drinks can I mix just using these?” This translates to the more formal query of “Fetch all recipes r_i for which $\{\text{Scotch}, \text{Vermouth}, \text{lemon juice}\} \supseteq r_i.\text{ingredients}$ (a possible answer to this query would be ’Hole-In-One’ [3]). If one of your guests prefers ’Vodka’ and ’orange juice’, you might want to know different variations and ask for the following “Fetch all recipes r_i for which $\{\text{Vodka}, \text{orange juice}\} \subseteq r_i.\text{ingredients}$”. Besides helping you to be a perfect host there are more serious applications where it is necessary to answer queries with predicates involving set-valued attributes efficiently. Some examples are keyword searches and queries in annotation databases containing information on images [6, 17], genetic, or molecular data [2, 11]. Further universal quantifiers can be transformed into set comparisons [8], which can now be supported efficiently. Most if not all sets found in set-valued attributes are small, containing less than a dozen elements. Examples for applications where almost all sets are of low cardinality can be found in product and production models [12] and molecular databases [1, 26]. Therefore we focus on indexing small sets.

Work on evaluation of queries with set-valued predicates is few and far between. Several indexes dealing with special problems in the object-oriented [7] and the object-relational data models [23] have been invented, e.g. nested indexes [4], path indexes [4], multi indexes [21], access support relations [18], join index hierarchies [27]. The predominant problem attacked by these index structures is the efficient evaluation of path expressions. With the exception of signature files [16] and Russian Doll Trees [14] the problem of indexing data items with set-valued attributes has been neglected by the database community. The methods used in text retrieval are very similar to our subset queries, where we look for sets that are supersets of a query set. However, superset queries, where sets are fetched that are subsets of a query set, are not necessary in text retrieval and therefore are not efficiently supported. This query type is needed in other applications, like molecular databases [26], and must therefore be supported efficiently by our index structures.

We adapt several index structures to suit the needs of efficiently indexing set-valued attributes. We give a quick review of the index structures and describe the performance-enhancing modifications. Our extensive benchmarks show that one of the index structures, the inverted file, is superior to the others in terms of retrieval time and index size.

The paper is organized as follows. The next section covers preliminaries, i.e. a formal description of queries and a brief introduction to the technique of superimposed coding. In Section 3 we briefly describe the index structures. A detailed description of the benchmark environment is the content of Section 4. We present and analyze the results of the benchmarks in Section 5. Section 6 concludes our paper.

2 Preliminaries

Before proceeding to the actual index structures, we need to explain some basics. A summary of the symbols used throughout this paper and their definitions are shown in Table 1. In the remainder of this section we define set-valued queries and take a quick look at the technique of superimposed coding.

2.1 Set-valued Queries

Our database consists of a finite set O of data items or objects o_i $(1 \leq i \leq n)$ having a set-valued attribute A with a domain D. Let $o_i.A \subseteq D$ denote the finite value of the attribute A for some data item o_i. A *query predicate* P is defined in terms 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 $Q = \{x\}$.

Symbol	*Definition*
O	set of data items (our database)
n	total number of data items
o_i	i-th data item of O
$ref(o_i)$	reference to o_i (e.g. an OID)
A	set-valued attribute
D	domain from which elements of A are taken
P	query predicate
Q	query set
θ	set comparison operator, here $=$, $\subseteq$, and $\supseteq$
s, t	arbitrary sets
$sig(s)$	signature of set s
$sig_d(s)$	prefix (first d bits) of $sig(s)$
b	length of signature
k	number of bits set in signature for each mapped element

Table 1: Used symbols

2.2 Superimposed Coding and Signatures

There are three reasons for using signatures to encode sets. First, signatures are much more space efficient than explicit set representations. Second, they are of fixed length and hence very convenient for index structures. Third, set comparison operators on signatures boil down to efficient bit operations.

When applying the technique of *superimposed coding*, each element of a given set s is mapped via a coding function to a bit field of length b—called *signature length*—where exactly $k < b$ bits are set. These bit fields are superimposed by a *bitwise or* operation to yield the final *signature* of the set s (denoted by $sig(s)$) [10, 20].

In our case, the coding function maps an element of D onto an integer which is used as a seed for a random number generator. The latter is called several times to determine each of the k bits to be set. The following properties of signatures are essential (let s and t be two arbitrary sets):

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

where $\mathrm{sig}(s) \subseteq \mathrm{sig}(t)$ and $\mathrm{sig}(s) \supseteq \mathrm{sig}(t)$ are defined as

$$\begin{aligned} \mathrm{sig}(s) \subseteq \mathrm{sig}(t) &:= \mathrm{sig}(s)\&\tilde{}\,\mathrm{sig}(t) = 0 \\ \mathrm{sig}(s) \supseteq \mathrm{sig}(t) &:= \mathrm{sig}(t)\&\tilde{}\,\mathrm{sig}(s) = 0 \end{aligned}$$

(& denotes *bitwise and* and ˜ denotes *bitwise complement*.)

As set comparisons are very expensive, signatures are helpful as filters. 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.

3 The Competitors

Let us briefly introduce the four index structures that we adapted for set retrieval. We distinguish two main strategies: signatures and inverted files. We start by discussing access structures based on signatures, then move on to inverted files.

3.1 Signature-based retrieval

There are basically three ways of signature organization: sequential, hierarchical, and partitioned. We examine sequential signature files (SSF) for sequential organization, signature trees (ST) for hierarchical organization, and extendible signature hashing (ESH) for partitioned organization.

3.1.1 Sequential Signature File (SSF)

$[sig(o_1.A), ref(o_1)]$	$[sig(o_2.A), ref(o_2)]$	$\ldots$	$[sig(o_n.A), ref(o_n)]$

Figure 1: Sequential signature file index structure (SSF)

General description A sequential signature file (SSF) [16] is a rather simple index structure. It consists of a sequence of pairs of signatures and references to data items. $[sig(o_i.A), ref(o_i)]$ (see Figure 1). During retrieval the SSF is scanned and all data items o_i with matching signature $sig(o_i.A)$ are fetched and tested for false drops.

Implementation details An SSF index consists of a root object, which is copied into main memory when the index is opened, and pages containing the signatures and references. The entries of the root object are described in Table 2.

The pages containing the signatures and references are doubly linked. The first 8 bytes contain the link (page number, area number) to the next page, the next 8 bytes the link to the previous page (see Figure 2). For empty links the page and area numbers are set to 0.

next page number	next area number	prev. page number	prev. area number	

Figure 2: Signature and reference lists

3.1.2 Signature Tree (ST)

General description The leaf nodes of the Signature Tree (ST) [9, 14] contain pairs $[sig(o_i.A), ref(o_i)]$. In the leaf nodes of an ST we find the same information as in an

Offset	*Name*	*Description*
0	noOfEntries	the number of data items inserted into index
4	sizeOfSig	b, the size of the signatures (in bits)
8	pageSize	the size of the pages in the database (in bytes)
12	sigPageNo	the starting page number of signature list
16	sigAreaNo	the area number of signature list
20	lastSigPageNo	the ending page number of signature list
24	lastSigPos	the first free position on last page of signature list
28	refPageNo	the starting page number of reference list
32	refAreaNo	the area number of reference list
36	lastRefPageNo	the ending page number of reference list
40	lastRefPos	the first free position on last page of reference list
44	noOfSetBits	k, the number of bits set in signature for each element

Table 2: Root object of an SSF index

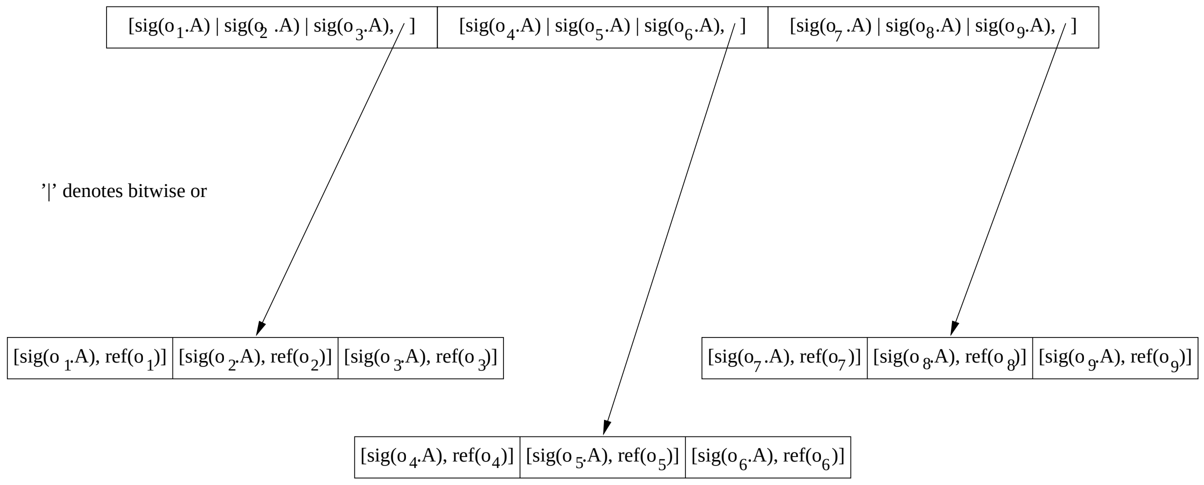


Figure 3: A Signature tree (ST)

SSF. We can construct a single signature representing a leaf node by superimposing all signatures found in the leaf node (with a bitwise or operation). An inner node contains signatures and references of each child node (see Figure 3). Signature trees are very similar to R-trees [13].

When we evaluate a query we begin by searching the root for matching signatures. We recursively access all child nodes whose signatures match and work our way down to the leaf nodes. There we fetch all eligible data items and check for false drops. Matching signatures in leaf nodes are determined by the appropriate bitwise operation (see Section 2.2). Matching signatures in inner nodes are determined as follows. For equality and subset queries we check if $sig(Q) \subseteq sig(\text{child node})$. For superset queries there has to be a non-empty intersection, i.e. $|sig(Q) \cap sig(\text{child node})| \geq k$. More details on ST can be found in our technical report [15].

Implementation details A Signature Tree consists of a root object, which is copied into main memory upon opening the index, and pages comprising the tree itself. Table 3 shows the contents of the root object.

Offset	*Name*	*Description*
0	depth	the height of the tree
4	sizeOfSig	the size of the signatures (in ints)
8	pageSize	the size of the pages in the database (in bytes)
12	rootPageNo	the page number of the root of the tree
16	rootAreaNo	the area number of the root of the tree
20	noOfSetBits	k, the number of bits set in signature for each element
24	sizeInBits	b, the size of the signatures (in bits)

Table 3: Root object of an ST index

We distinguish two different kinds of nodes in a signature tree, inner nodes and leaf nodes. The first 4 bytes of each page are identical. They are used to store the offset of the first free byte on a page. Page of inner nodes contain pairs of signatures and 8 byte references (page numbers and area numbers) to child nodes. In leaf pages we store EOS object identifiers of data items instead of page references. EOS oids also have a size of 8 bytes (see Figure 4).

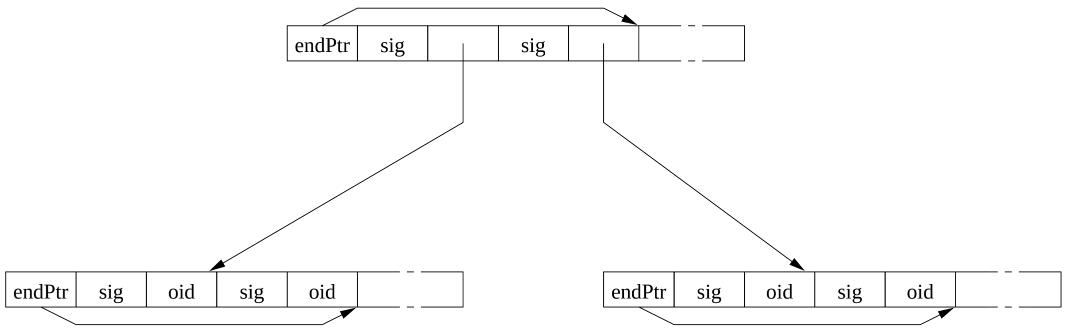


Figure 4: Pages of a signature tree

3.1.3 Extendible Signature Hashing (ESH)

General description An extendible signature hashing index (ESH) is divided into two parts, a directory and buckets. In the buckets we store the signature/reference pairs of all data items. We determine the bucket into which a signature/reference pair is inserted by looking at a prefix $sig_d(o_i.A)$ of d bits of a signature. For each possible bit combination of the prefix we find an entry in the directory pointing to the corresponding bucket.

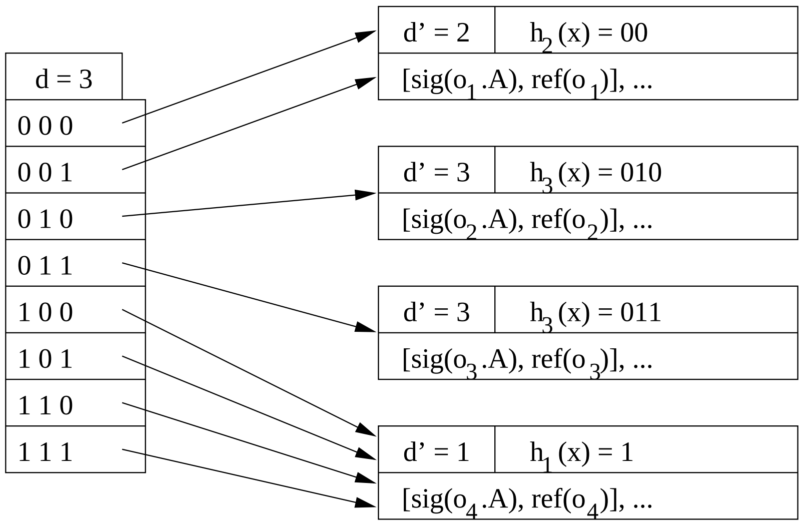


Figure 5: Extendible signature hashing (ESH)

Obviously the directory has 2^d entries, where d is called *global depth*. 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 signature the inspected prefix has to be increased until we are able to distinguish the signatures. 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. Pointers to buckets that have not been split are just copied. For the split bucket the new pointers are put into the directory and the global depth is increased (see Figure 5). We stop splitting the directory beyond a global depth of 20 and start using chained overflow buckets at this point, as further splitting leads to a huge directory, which is difficult to manage.

The evaluation of an equality query is straightforward. We look up the entry for $sig_d(Q)$ in the directory, fetch the content of the corresponding bucket, check the full signatures, and eliminate all false drops. In order to find all subsets (supersets) of a query set Q, we determine all buckets to be fetched. We do this by generating all subset (supersets) of $sig_d(Q)$ with the algorithm by Vance and Maier [24]. Then we access the corresponding buckets, all the while taking care not to access a bucket more than once. Afterwards we check the full signatures and eliminate the false drops.

ESH is similar to Quickfilter by Zezula, Rabitti, and Tiberio [28]. However, we use extendible hashing instead of linear hashing as our underlying hashing scheme and we also optimize the bucket accesses. For more details see our technical report [15].

Implementation details Like in the other index structures the general information of an ESH index is kept in a root object. The structure of an ESH index root object can be seen in Table 4.

In the root node of the directory the first 10 bits of the query signature are checked, i.e. they are used as an offset to find the corresponding entry. This entry points to a directory subpage, which is fetched in order to check the remaining 10 bits of the query

Offset	*Name*	*Description*
0	depth	the global depth of the hash table
4	sizeOfSigInInts	the size of the signatures (in ints)
8	sizeOfSigInBits	the size of the signatures (in bits)
12	noOfSetBits	k, the number of bits set in signature for each element
16	pageSize	the size of the pages in the database (in bytes)
20	rootPageNo	the page number of the root node of the directory
24	areaNo	the area number of the root node of the directory
28	maxBucketSize	not used
32	maxPossDepth	the maximal global depth of the hash table (before chaining)

Table 4: Root object of an ESH index

signature. The root node contains 4 byte references (page numbers) to the subpages. The subpages contain 4 byte references (page numbers) to the buckets. In a bucket the first 4 bytes store the link to the next page (in form of a page number), if chaining has been invoked. The next 4 bytes contain an offset pointing to the first free byte on the page. This is followed by 4 bytes holding the local depth of the bucket. For an illustration see Figure 6.

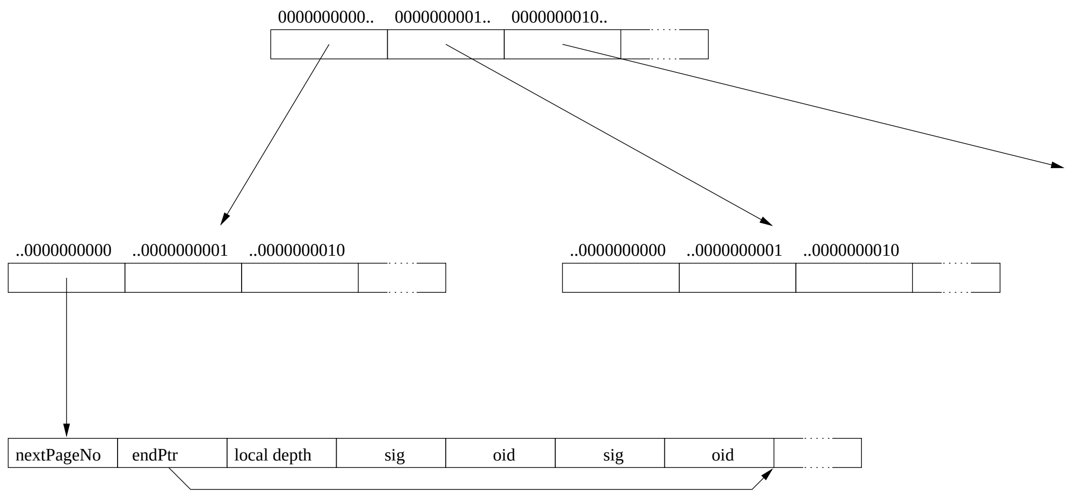


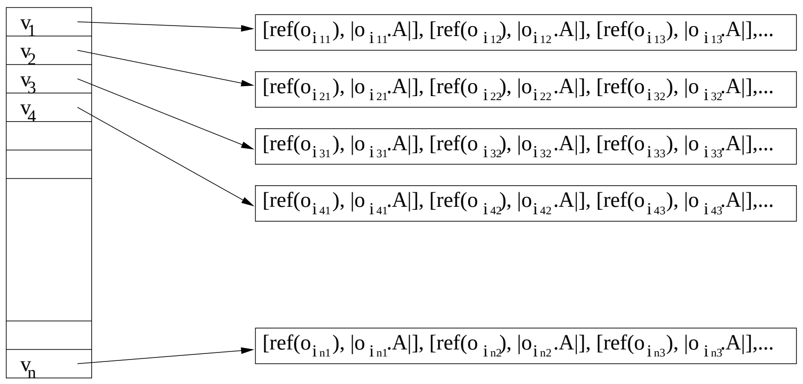
Figure 6: Pages of an ESH index

As already mentioned we need a way to rapidly step through all subsets and supersets of a given signature. An algorithm utilized by Vance and Maier in their blitzsplit join ordering algorithm quickly generates all subset representations of a given bit string [24]. It is repeated on the left-hand side of Figure 7. When executed, x passes through all possible subsets of $sig(s)$. Its counterpart on the right-hand side generates all supersets by inverting the signature $sig(s)$, stepping through the subsets of the inverted $sig(s)$ and inverting the generated sets (˜ stands for the *bitwise complement*, - *for the two-complement*). $f(x)$ denotes a function processing the current subset/superset.

```
x = sig(s) & -sig(s);                x = ~sig(s) & -~sig(s);
f(x);                                f(~x);
while(x)                             while(x)
{                                    {
   x = sig(s) & (x - sig(s));           x = ~sig(s) & (x - ~sig(s));
   f(x);                                f(~x);
}                                    }
```

Figure 7: Quick generation of subsets/supersets

3.2 Inverted Files



v_1
v_2
v_3
v_4
v_n
$[ref(o_{i_{11}}), |o_{i_{11}}.A|], [ref(o_{i_{12}}), |o_{i_{12}}.A|], [ref(o_{i_{13}}), |o_{i_{13}}.A|],...$
$[ref(o_{i_{21}}), |o_{i_{21}}.A|], [ref(o_{i_{22}}), |o_{i_{22}}.A|], [ref(o_{i_{32}}), |o_{i_{32}}.A|],...$
$[ref(o_{i_{31}}), |o_{i_{31}}.A|], [ref(o_{i_{32}}), |o_{i_{32}}.A|], [ref(o_{i_{33}}), |o_{i_{33}}.A|],...$
$[ref(o_{i_{41}}), |o_{i_{41}}.A|], [ref(o_{i_{42}}), |o_{i_{42}}.A|], [ref(o_{i_{43}}), |o_{i_{43}}.A|],...$
$[ref(o_{i_{n1}}), |o_{i_{n1}}.A|], [ref(o_{i_{n2}}), |o_{i_{n2}}.A|], [ref(o_{i_{n3}}), |o_{i_{n3}}.A|],...$

Figure 8: Inverted File

General description An inverted file consists of a directory containing all distinct values that can be searched for and a list for each value containing all references to data items in which the corresponding value appears (for an overview see [19, 22]). We hold the search values of the directory in a B-tree. We modify the lists by storing the cardinality of the set with each reference. This enables us to answer queries efficiently by using the cardinalities as a quick pretest (see Figure 8).

Using an inverted file for evaluating subset queries is straightforward. For each item in the query set the appropriate list is fetched and all those lists are intersected. This query type is comparable to partial match retrieval, which is the main application of inverted files in text retrieval. When evaluating equality queries we proceed the same way as with subset queries, but we also eliminate all references to data items whose set cardinality does not match the query set cardinality. When evaluating a superset query we search all lists associated with the values in the query set. We count the number of occurrences for each reference appearing in a retrieved list. When the counter for a reference does not match the cardinality of its set, we eliminate that reference. We can do this, because this reference also appears in lists associated with values that are not in the query set. So the referenced set cannot be a subset of the query set.

We use several well-known techniques to increase the performance of inverted lists. To reduce the size of the lists we compress them. We keep the lists sorted and encode the gaps using very light-weight techniques [25]. It is also sensible to fetch the lists in increasing size and to use thresholding, i.e. instead of fetching very large lists, we immediately access the data items. For a more detailed explanation of these techniques see [31, 32].

Implementation details The root object of an inverted file index and its description are depicted in Table 5.

Offset	*Name*	*Description*
0	noOfEntries	the number of data items inserted into index
4	depth	the height of the B-tree
8	pageSize	the size of the pages in the database (in bytes)
12	rootPageNo	the page number of the root of the tree
16	rootAreaNo	the area number of the root of the tree
20	minNoOfEntries	minimal number of entries in a node

Table 5: Root object of an inverted file index

The first 4 bytes of a node in the B-tree directory of an inverted file index are used to store the offset of the first free byte on a page. In leaf nodes we have search-keys along with the references to the corresponding list of data item references (which will be described later). In inner nodes we store references to child nodes which are separated by search keys. The size of search keys is 4 bytes, the size of references 8 bytes (see Figure 9).

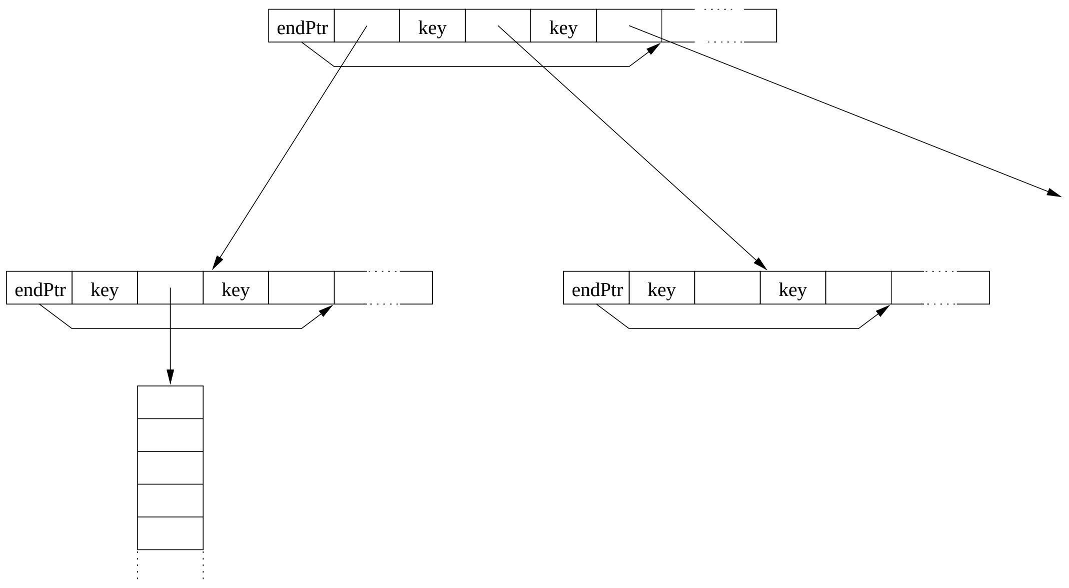


Figure 9: Pages of an inverted file index

We distinguish two different types of lists, compressed and uncompressed. List containing less than 8 oids are not compressed to avoid overhead. In Table 6 the internal structure of a compressed list is shown.

Offset	*Name*	*Description*
0	compressionFlag	for a compressed list contains char 'c'
1	noOfOids	number of oids in the list
5	firstOid	first oid in uncompressed form
9	gapBits	number of bits used to code the gaps
10	setCardBits	number of bits used to code set cardinalities
11	oids and set cardinalities	a list of compressed oids followed by a list of compressed set cardinalitites

Table 6: Compressed list

An uncompressed list merely consists of a compression flag, which is set to 'u' in this case, and a list of oids and set cardinalities (see Table 7). The number of oids in the list is derived from the size of the list, which can be determined by an EOS function.

Offset	*Name*	*Description*
0	compressionFlag	for an uncompressed list contains char 'u'
1	oids and set cardinalities	a list of (uncompressed) pairs of oids and set cardinalities

Table 7: Uncompressed list

4 The Benchmark Environment

When comparing index structures, there are several principal avenues of approach: analytical approach, mathematical modelling, simulations, and experiment [30]. 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. In the following sections we present our benchmark specification.

4.1 System Parameters

The benchmarks were conducted on a lightly loaded UltraSparc2 with 256 MByte main memory running under Solaris 2.6. The total disk space amounted to 10 GByte. All index structures were set atop the EOS storage manager, release 2.2, using the C++ interface of the manager [5]. We implemented the data structures and algorithms of the index structures in C++ using the GNU C++ Compiler Version 2.8.1. The data structures were stored on 4K plain pages. The algorithms were not parallelized in any way. We allowed no buffering/caching of any sort, i.e. each benchmark was run under cold start conditions. We kept the storage manager from buffering pages read from disk by running the queries locally in the single-user mode of EOS (no client/server mode) and terminating all EOS processes after the processing of a query was done. For the next query EOS was restarted from scratch. We prevented the operating system from buffering by using RawIO instead of the file system. We cleared the internal disk cache of relevant pages by transferring 2 MBytes of data between the queries. Within a single run, the buffer was large enough to prevent accessing pages more than once.

4.2 Generating Data

We generated databases containing data items with set-valued attributes, varying the cardinality of the database (in number of data items contained), the cardinality of the set-valued attributes (in number of elements contained), and the cardinality of the domain of the set elements. For a summary see table 8. Each data item with a set-valued attribute was stored on a separate page to eliminate any clustering effects.

parameter	*symbol*	*min value*	*max value*
database cardinality	$\|O\|$	50000	250000
set cardinality	$\|o_i.A\|$	5	15
domain cardinality	$\|D\|$	200	1000000

Table 8: Parameters for generation of databases

The data items in the databases were generated randomly. We investigated the performance of the index structures for uniformly distributed data and skewed data. For the skewed data we used a Zipf distribution with $z = 1$.

4.3 Generating Queries

In order to guarantee hits during query evaluation we generated the query sets out of data sets inserted into the database. For equality queries the sets were taken directly from the bulkload files. Query sets for subset queries were generated by creating subsets from data sets containing 15 elements. Generating query sets for superset queries was done by adding random values to data sets containing 5 elements. The data sets used for generating query sets came from the head, middle, and tail of the bulkload files.

5 Results

We present an excerpt of the results of our extensive benchmarks emphasizing query evaluation speed. We also look at scalability, and the influence of skewed data and domain size. As unit of measurement we use the number of page accesses (and to a lesser extent total elapsed time in msec). In Tables 9, 10, 15, and 16 only the number of page accesses within the index structures appear, since the costs for fetching qualifying data items are the same for all index structures. In Tables 11, 12, 17, and 18 the total elapsed time appears including the costs for fetching qualifying data items. Additionally for the signature-based techniques we need to consider the false drops, of which statistics are shown in Figure 11.

5.1 Retrieval Costs

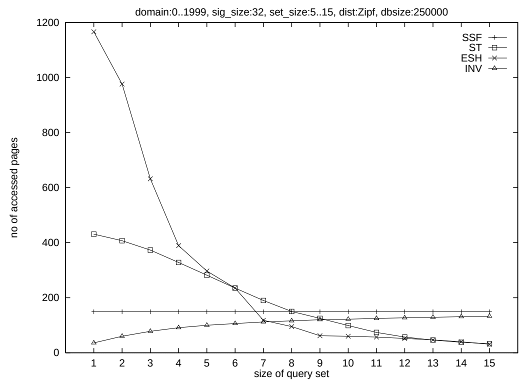
	uniform distribution											
	$=$				$\subseteq$				$\supseteq$			
DB size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
50000	149	32	2	21	149	59	69	16	149	219	48	24
100000	296	62	2	21	296	99	144	16	296	438	103	24
150000	443	63	2	21	443	165	157	16	443	659	139	24
200000	590	83	2	21	590	205	192	16	590	881	150	24
250000	737	131	2	18	737	255	408	16	737	1091	207	24

Table 9: Retrieval costs for uniformly distributed data (in number of pages accessed)

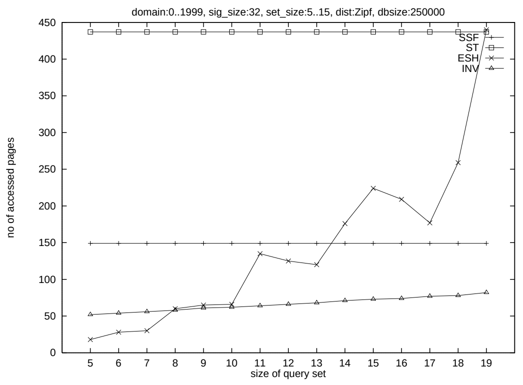
	Zipf distribution											
	$=$				$\subseteq$				$\supseteq$			
DB size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
50000	149	31	2	41	149	96	142	32	149	222	63	33
100000	296	87	2	58	296	191	287	59	296	437	141	36
150000	443	103	3	67	443	235	284	73	443	655	142	53
200000	590	153	3	86	590	403	397	106	590	887	190	66
250000	737	176	3	118	737	425	485	127	737	1108	193	83

Table 10: Retrieval costs for Zipf distributed data (in number of pages accessed)

Tables 9 (uniformly distributed data) and 10 (Zipf distributed data) show the query evaluation costs for different database sizes. For equality queries the hash table index is unbeaten, as it only takes 2 (respectively 3) page accesses to reach the searched data item. Subset and superset queries are the more difficult and more interesting cases. In these cases the inverted file index reigns supreme. For uniformly distributed data the retrieval costs depend mainly on the size of the query set as the (compressed) lists grow slowly. For skewed data the performance is not as impressive as for uniformly distributed data. Here the lists for the most frequently appearing value grow unproportionally faster. These lists are also queried more frequently, because the query sets are also skewed. Skewed data also influences the other index structures (except SSF). While performance losses of ESH are marginal, ST has severe problems. ESH still achieves low retrieval costs, because it compensates by rapidly doubling its directory. The number of overflow buckets is still limited by the doubling. ST has trouble with balancing the tree structure. Skewed data lead to very many splits during the insertion of the data, thereby increasing the depth of the tree causing a negative impact on the retrieval costs.



(a) Subset queries



(b) Superset queries

Figure 10: Retrieval cost depending on query set size

The signature-based index structures (except SSF) have another disadvantage. Usually subset queries are formulated with small query sets and superset queries with large query sets. In these cases ST and ESH show their worst performance (see Figure 10), as ST needs to traverse many branches in the tree and ESH needs to visit almost all buckets.

	uniform distribution											
	=				$\subseteq$				$\supseteq$			
DB size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
50000	1263	317	33	146	6827	5919	7029	495	1770	1976	1257	161
100000	2662	661	23	218	13946	12761	14421	870	3721	4274	2240	249
150000	4061	680	25	242	21977	21524	23239	1158	5340	6134	2905	278
200000	5396	1028	37	243	30025	29898	33767	1354	7137	8156	4282	287
250000	6866	1970	33	221	35767	36552	40767	1546	8863	10302	4945	301

Table 11: Retrieval costs for uniformly distributed data (total elapsed time in msec)

Table 11 (uniform distribution) and Table 12 (Zipf distribution) present the query evaluation costs measured in total elapsed time. These are not directly comparable to the query evaluation costs measuring page accesses as the time for fetching the qualifying data items is included. For the average total number of right and false drops of the signature-based index structures see Table 13. Let us first look at uniformly distributed data (Table 11). Not surprisingly ESH is superior to the other index structures for equality queries with inverted files close behind on second place. The influence of drops (both right and false drops) is minimal in this case (see Table 13). For subset queries inverted files outperform all other index structures. The poor performance of the signature-based index structures is due to the high number of false drops. ESH also needs to generate a large number of sets for the subqueries, which slows it down even further. Superset queries are generally handled better than subset queries, because the number of drops is much lower. The gap between inverted files and the signature-based index structures has become

	Zipf distribution											
	=				⊆				⊇			
DB size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
50000	1348	392	25	252	18279	21719	25512	25019	2906	3213	2549	564
100000	2771	1304	29	302	37693	49835	57124	68754	6325	6926	5342	1525
150000	4086	2665	35	331	53270	76151	82615	73983	9237	10184	7604	1894
200000	5348	3793	41	418	78803	115158	125955	85157	10939	12491	8901	2336
250000	6912	3146	46	531	92317	138423	149502	98140	14298	16327	11901	2962

Table 12: Retrieval costs for Zipf distributed data (total elapsed time in msec)

smaller, but inverted files still lead unchallenged. Zipf distributed data (Table 12) leads to some drastic performance losses. What are the reasons for this? For subset queries the main reason is the large increase in qualifying data items because of the skewed data (see Table 13). A slight increase in false drops can also be noticed for superset queries. Skewed data influences the split behavior of ST and ESH unfavorably, i.e. ST has a greater height while ESH has longer overflow chains. Because of the skewed data the lengths of the lists in an inverted file will also be skewed, i.e. we have a small number of large lists. Those lists will also be accessed most, because the query sets are skewed likewise.

	uniform distribution						Zipf distribution					
	=		⊆		⊇		=		⊆		⊇	
DB size	right	false	right	false	right	false	right	false	right	false	right	false
50000	1	0	17	983	1	79	1	0	3514	1121	1	119
100000	1	0	34	2020	1	174	1	0	7077	2130	1	160
150000	1	0	50	3238	1	211	1	0	9809	3249	1	430
200000	1	0	69	4419	1	280	1	0	14599	4894	1	501
250000	1	0	85	5128	1	658	1	0	17833	5016	1	655

Table 13: Number of right and false drops for signature-based index structures

5.2 Scaling

	uniform distribution				Zipf distribution			
DB size	SSF	ST	ESH	INV	SSF	ST	ESH	INV
50000	149	220	434	269	149	223	461	221
100000	296	439	868	530	296	438	916	428
150000	443	660	1272	788	443	656	1178	637
200000	590	882	1743	1046	590	888	1501	845
250000	737	1092	2165	1302	737	1109	1845	1060

Table 14: Index sizes (in number of 4K pages)

Table 14 shows the benchmarks results pertaining to the scalability of the index structures. The compression of the inverted lists makes them competitive. An earlier uncompressed version of the inverted files we used was about 8 to 9 times larger. Zipf distributed data lead to even smaller files as we have fewer, longer lists which can be compressed better. There is no effect of data skew on SSF, as the signatures are stored sequentially in a file. Therefore the size depends solely on the size of the signatures and not on the organization of the signatures. The size of ST increases slightly for skewed data, because the signatures are organized less favorably. The size of ESH decreases slightly, because the space is better utilized. The reason for this is the larger number of chained overflow buckets.

5.3 Influence of domain size

	uniform distribution											
	$=$				$\subseteq$				$\supseteq$			
Domain size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
200	296	54	2	22	296	122	113	16	296	443	75	23
2000	296	62	2	21	296	99	144	16	296	438	103	24
1000000	296	56	2	30	296	104	175	24	296	432	79	36

Table 15: Retrieval costs for uniformly distributed data (in number of pages accessed)

	Zipf distribution											
	$=$				$\subseteq$				$\supseteq$			
Domain size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
200	296	126	3	42	296	195	233	59	296	437	162	41
2000	296	87	2	58	296	191	287	59	296	437	141	36
1000000	296	81	2	31	296	139	202	39	296	427	122	41

Table 16: Retrieval costs for Zipf distributed data (in number of pages accessed)

Tables 15 and 16 show the influence of the domain size on the retrieval costs. Although the performance of the inverted files index decreases for increasing domain sizes, it still has a strong showing. The skip from 24 page accesses to 36 page accesses for uniformly distributed data is due to a deeper B-tree structure (going from depth 2 to depth 3). For Zipf distribution we did not notice this effect, because many values of the domain do not appear in the data sets.

In Tables 17 and 18 the total query processing time is presented. The main reason for the discrepancy between costs measured in page accesses and the total processing time is the cost for fetching data items. In Table 19 the average number of right and false drops for each query type can be found. Generally small domains intensify the skewing of data, because the variety of appearing values is lowered. ST and ESH are the index structures

	uniform distribution											
	=				⊆				⊇			
Domain size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
200	2705	691	26	231	16708	15943	17492	3364	3890	4371	2129	315
2000	2662	661	23	218	13946	12761	14421	870	3721	4274	2240	249
1000000	2716	561	25	215	13759	12546	14464	178	3694	4036	1870	234

Table 17: Retrieval costs for uniformly distributed data (total elapsed time in msec)

	Zipf distribution											
	=				⊆				⊇			
Domain size	SSF	ST	ESH	INV	SSF	ST	ESH	INV	SSF	ST	ESH	INV
200	2627	2577	40	240	40000	51165	60768	36198	9193	9677	8464	601
2000	2771	1304	29	302	37693	49835	57124	68754	5641	6233	4492	593
1000000	2695	1369	31	185	22335	25399	28846	29134	3728	4099	2414	714

Table 18: Retrieval costs for Zipf distributed data (total elapsed time in msec)

that suffer most from this effect, especially for subset queries. ST has to traverse a larger number of branches, while the explosive growth of the ESH directory makes it necessary to generate a large number of subqueries during query processing.

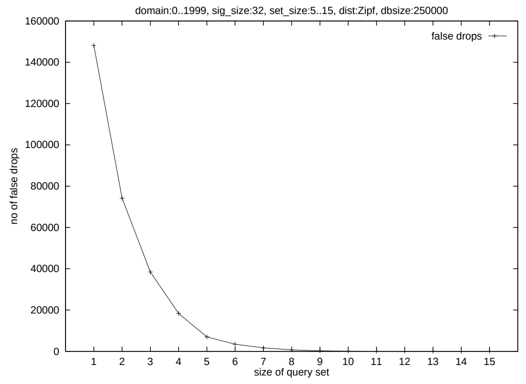
	uniform distribution						Zipf distribution					
	=		⊆		⊇		=		⊆		⊇	
Domain size	right	false	right	false	right	false	right	false	right	false	right	false
200	1	0	350	2258	1	195	1	0	8836	1327	1	1086
2000	1	0	34	2020	1	174	1	0	7077	2130	1	160
1000000	1	0	1	1995	1	158	1	0	3011	1453	1	486

Table 19: Number of right and false drops for signature-based index structures

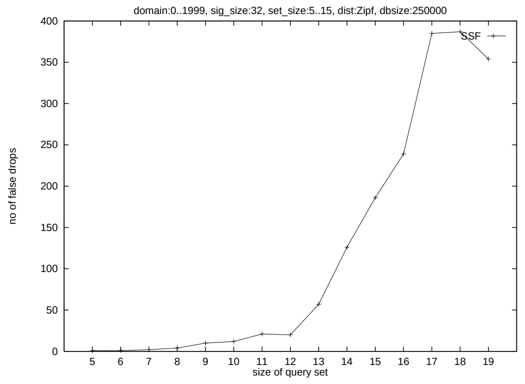
Table 20 shows the results of the influence of the domain size on the index size. The smaller the domain, the better the space demand of inverted files. The obvious reason for this is that for each value appearing in a set, a list has to be allocated. The number of lists could be reduced by merging lists of infrequently appearing values into one. This would lead to higher retrieval costs, however, as false drops would have to be eliminated. For skewed data the size of the inverted file index is smaller than for uniformly distributed data due to a better compressibility of the data (there are fewer, longer lists). SSF is not influenced at all by the domain size. ST is influenced slightly. ESH has a smaller index structure for small domains at the price of higher retrieval costs. Smaller domains have a similar effect as Zipf distributed data leading to chained overflow buckets with a better space utilization.

	uniform distribution				Zipf distribution			
Domain size	SSF	ST	ESH	INV	SSF	ST	ESH	INV
200	296	444	879	369	296	438	758	266
2000	296	439	868	530	296	425	787	341
1000000	296	433	886	1559	296	428	823	960

Table 20: Influence of domain on index size (in number of 4K pages)



(a) Subset queries



(b) Superset queries

Figure 11: Number of false drops (for database cardinality 250000)

5.4 False drops

Figure 11 summarizes the false drop rates of the signature-based index structures. The rates are identical for all three index structures as the same signatures are stored in each index. The index structures differ only in the organization of the signatures. We have registered no false drops for equality queries in our benchmarks. Small query sets for subset queries lead to sparse query bitvectors and large query sets for superset queries to dense query bitvectors. So for subset and superset queries the signature-based index structures show the worst performance for these cases.

6 Conclusion

We have studied the performance of several different index structures for set-valued attributes of low cardinality. All index structures were based on traditional methods, which we refitted for indexing set-valued attributes. We implemented sequential signature files, signature trees, extendible signature hashing and B-tree based inverted files designed to support the evaluation of queries containing set-valued predicates. Using these implementations we conducted extensive benchmarks. The inverted file index dominated the field clearly, though for equality queries the hash-based ESH was faster. Generally the signature-based index structures have difficulties with skewed data and frequently used query sets (small sets for subset queries, larger sets for superset queries). Zobel, Moffat, and Ramamohanarao made a similar observation for text retrieval while comparing inverted files to signature files [29].

In summary, we can say that for applications with set-valued attributes of low cardinality inverted lists showed the best overall performance of all studied index structures. It was least affected by the variation of the benchmark parameters and displayed the most predictable behavior, which makes it a good choice for practical use.

References

[1] J.E. Ash, P.A. Chubb, S.E. Ward, S.M. Welford, and P. Willet. *Communication, Storage and Retrieval of Chemical Information.* Ellis Horwood, Chichester, England, 1985.

[2] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its new supplement TrEMBL. *Nucleic Acids Research*, 24(1):21–25, 1996.

[3] S.A. Berk. *The New York Bartender's Guide.* Black Dog & Leventhal Publishers, Inc., New York, 1995.

[4] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[5] A. Biliris and E. Panagos. EOS user's guide. Technical report, AT&T Bell Laboratories, 1994.

[6] K. Böhm and T.C. Rakow. Metadata for multimedia documents. *SIGMOD Record*, 23(4):21–26, December 1994.

[7] R. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[8] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In *Proc. of the 23rd VLDB Conference*, pages 286–295, Athens, August 1997.

[9] U. Deppisch. 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.

[10] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Informations Systems*, 2(4):267–288, October 1984.

[11] K.H. Fasman, S.I. Letovsky, R.W. Cottingham, and D.T. Kingsbury. Improvements to the GDB human genome data base. *Nucleic Acids Research*, 24(1):57–63, 1996.

[12] T. Grobel, C. Kilger, and S. Rude. Object-oriented modelling of production organization. In *Tagungsband der 22. GI-Jahrestagung*, Karlsruhe, September 1992. Informatik Aktuell, Springer-Verlag. (in German).

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD*, Boston, Mass., 1984.

[14] J.M. Hellerstein and A. Pfeffer. The RD-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison, 1994.

[15] S. Helmer. Index structures for databases containing data items with set-valued attributes. Technical Report 2/97, Universität Mannheim, 1997. http://pi3.informatik.uni-mannheim.de.

[16] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In *Proc. of the 1993 ACM SIGMOD*, pages 247–256, Washington D.C., 1993.

[17] R. Jain and A. Hampapur. Metadata in video databases. *SIGMOD Record*, 23(4):27–33, December 1994.

[18] A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[19] H. Kitagawa and K. Fukushima. Composite bit-sliced signature file: An efficient access method for set-valued object retrieval. In *Proc. Int. Symposium on Cooperative Database Systems for Advanced Applications (CODAS)*, pages 388–395, Kyoto, Japan, December 1996.

[20] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison Wesley, Reading, Massachusetts, 1973.

[21] D. Maier and J. Stein. Indexing in an object-oriented database. In *Proc. of the IEEE Workshop on Object-Oriented DBMSs*, Asilomar, California, September 1986.

[22] R. Sacks-Davis and J. Zobel. Text databases. In *Indexing Techniques for Advanced Database Systems*, pages 151–184. Kluwer Academic Publishers, 1997.

[23] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1996.

[24] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 35–46, Montréal, Canada, June 1996.

[25] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. Technical Report 3/98, Universität Mannheim, 1998. http://pi3.informatik.uni-mannheim.de.

[26] M. Will, W. Fachinger, and J.R. Richert. Fully automated structure elucidation - a spectroscopist's dream comes true. *J. Chem. Inf. Comput. Sci.*, 36:221–227, 1996.

[27] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigation in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 522–533, 1994.

[28] P. Zezula, F. Rabitti, and P. Tiberio. Dynamic partitioning of signature files. *ACM Transactions on Information Systems*, 9(4):336–369, October 1991.

[29] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. Technical Report CITRI/TR-95-5, Collaborative Information Technology Research Institute (CITRI), Victoria, Australia, 1995.

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

[31] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for fulltext database systems. In *Proc. of the 18th VLDB Conference*, pages 352–362, Vancouver, Canada, 1992.

[32] J. Zobel, A. Moffat, and R. Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proc. of the 19th VLDB Conference*, pages 290–301, Dublin, Ireland, 1993.