The VLDB Journal (2003) 12: 244–261 / Digital Object Identifier (DOI) 10.1007/s00778-003-0106-0

A performance 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
(e-mail: helmer|moerkotte@informatik.uni-mannheim.de)

Edited by E. Bertino. Received: May 2000 / Accepted: October 18, 2000
Published online: September 17, 2003 –

Abstract. The efficient retrieval of data items on set-valued attributes is an important research topic that has attracted little attention so far. We studied and modified four index structures (sequential signature files, signature trees, extendible signature hashing, and inverted files) for a fast retrieval of sets with low cardinality. We compared the index structures by implementing them and subjecting them to extensive experiments, investigating the influence of query set size, database size, domain size, and data distribution (synthetic and real). The results of the experiments clearly indicate that inverted files exhibit the best overall behavior of all tested index structures.

Keywords: Database management systems – Physical design – Access methods – Index structures – Set-valued attributes

1 Introduction

One demand of modern day applications, especially in the object-oriented and object-relational world, is the efficient evaluation of queries involving set-valued attributes. The related research topic of indexing data items on set-valued attributes has attracted little attention so far. We modified and combined several existing techniques to speed up set retrieval and subjected our access methods to a series of experiments.

Examples of applications involving set-valued attributes include keyword searches and queries in annotation databases containing information on images [6,18] or genetic or molecular data [2,12]. Further, universal quantifiers can be transformed into set comparisons [8] that can now be supported efficiently. Many sets found in set-valued attributes are small, containing less than a dozen elements. Examples of applications where almost all sets are of low cardinality can be found in product and production models [13] and molecular databases [1,30]. We focus on applications where sets are small enough to be embedded into the data items.

Indexing set-valued attributes is a difficult task for traditional index structures because the data cannot be ordered and the queries are neither point queries nor range queries. We have combined many techniques from several different areas (partial-match retrieval, text retrieval, traditional database index structures, spatial index structures, join algorithms) to implement four index structures for set-valued attributes: sequential signature files, signature trees, extendible signature hashing, and inverted files. We are looking for a single index structure that is capable of supporting queries involving equal, subset, and superset predicates efficiently. Some of the existing index structures have been shown to excel at one or two of these query types but fail at the others. We investigate the index structures in view of overall performance for all mentioned query types.

An evaluation of these index structures is very complex since there is a huge number of parameters involved. The complexity of the task and the lack of theoretical tools for evaluating the performance of index structures motivated us to conduct extensive experiments. Having determined the evaluation method, we decided to reveal implementation details of our index structures so that the results of the experiments would become more transparent.

1.1 Related work

Studies on the evaluation of queries with set-valued predicates are few and far between. Several indexes dealing with special problems in the object-oriented [7] and the object-relational data models [27] have been invented, e.g., nested indexes [3], path indexes [3], multi indexes [23], access support relations [19], and join index hierarchies [32]. The predominant problem attacked by these index structures is the efficient evaluation of path expressions. With the exception of signature files [17] and Russian Doll Trees [15], the problem of indexing data items with set-valued attributes has been neglected by the database community. The methods used in text retrieval, though at first glance similar to our methods, have a different focus. Most searches involve ranking, and even if exact Boolean queries are used, this is largely partial-match retrieval (i.e., query set $\subseteq$ data set). When indexing set-valued attributes, however, we have to look beyond that. In molecular databases, for example, searching for characteristic parts of a large molecule is a common query. So we also need to support queries looking for subsets of a query set efficiently, i.e., query set $\supseteq$ data set. Retrieving sets equal to a query set, though a more simple case, should also not be forgotten. Text

Table 1. Used symbols

Symbol	*Definition*
O	finite 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 of A
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

retrieval techniques largely ignore the efficient evaluation of these two query types simply because they are not needed in that context.

1.2 Outline

The paper is organized as follows. The next section covers preliminaries, i.e., a formal description of queries. In Sect. 3, we describe the index structures. A detailed description of the environment in which the experiments were conducted is the content of Sect. 4. We discuss the fine-tuning of the signature-based index structures for the experimental environment in Sect. 5. We present and analyze the results of the experiments in Sect. 6 for uniformly distributed data, in Sect. 7 for skewed data, and in Sect. 8 for real data. Section 9 concludes the paper.

2 Preliminaries

Before proceeding to the actual index structures, we need to explain some basics. Table 1 contains a summary of the symbols used throughout this paper and their definitions. The remainder of this section presents the definition of 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 $Q = \{x\}$.

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 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, depending on the application, signatures may be much more space efficient than explicit set representation.

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)) [11,22]. The parameters k and b have to be chosen carefully to suit the needs of the application (more on tuning these parameters in Sect. 5).

In our case, the coding function maps an element of D onto an integer that is used as a seed for a random number generator. The latter is called k 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):

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

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

$$\text{sig}(s) \subseteq \text{sig}(t) := \text{sig}(s) \& \tilde{} \text{sig}(t) = 0$$
$$\text{sig}(s) \supseteq \text{sig}(t) := \text{sig}(t) \& \tilde{} \text{sig}(s) = 0$$

($\&$ denotes *bitwise and* and $\tilde{}$ 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.

Basically, there are three ways of signature organization: sequential, hierarchical, and partitioned. We examine sequential signature files (SSF) [17] for sequential organization, signature trees (ST) [9,15] for hierarchical organization, and extendible signature hashing (ESH) [16] for partitioned organization.

3.1.1 Sequential signature file (SSF)

General description. A sequential signature file (SSF) [17] is a rather simple index structure. It consists of a sequence of signature/reference pairs (the references pointing to the data items) $[sig(o_i.A), ref(o_i)]$ (see Fig. 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.

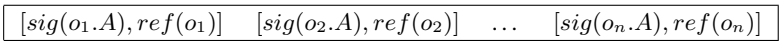


Fig. 1. Sequential signature file index structure (SSF)

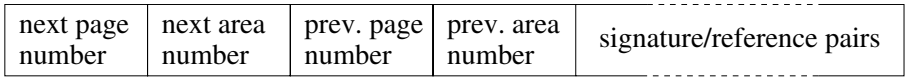


Fig. 2. Signature and reference list node

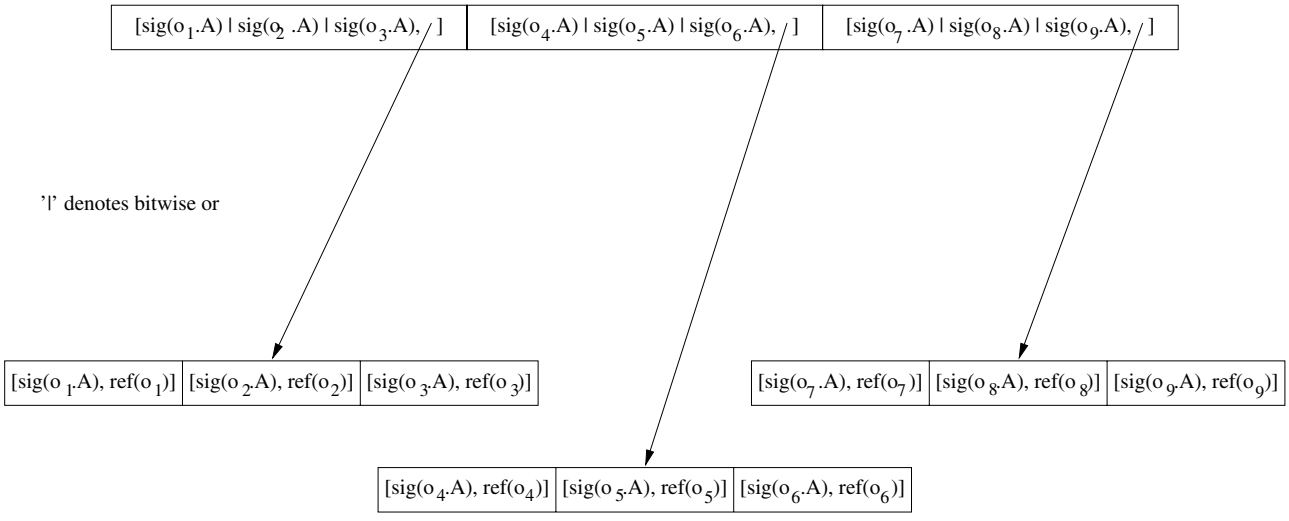


Fig. 3. A signature tree (ST)

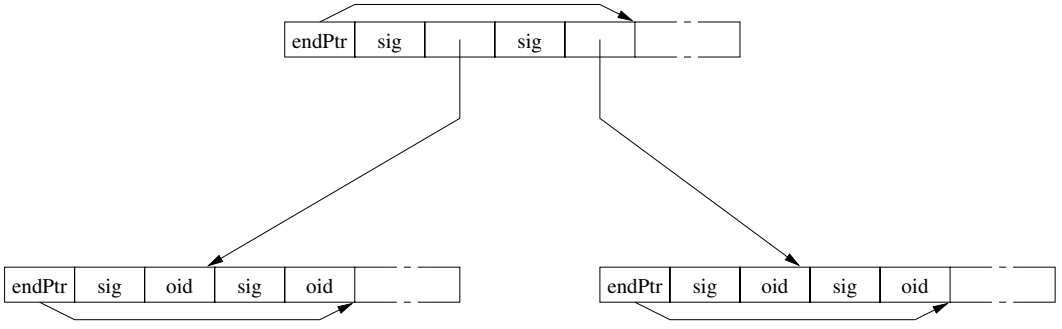


Fig. 4. Pages of a signature tree

Implementation details. An SSF index consists of a root object that is copied into main memory when the index is opened and pages containing the signatures and references. 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 Fig. 2). For empty links the page and area numbers are set to 0.

3.1.2 Signature tree (ST)

General description. The internal structure of a signature tree is very similar to that of R-trees [14]. The leaf nodes of the signature tree (ST) [9,15] contain pairs $[sig(o_i.A), ref(o_i)]$, so there we find the same information as in an 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 (Fig. 3).

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 the beginning of Sect. 3.1). 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 nonempty intersection, i.e., $|sig(Q) \cap sig(\text{child node})| \geq k$. More details about ST – especially on updates – can be found in our technical report [16].

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.

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. The pages 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 (see Fig. 4).

3.1.3 Extendible signature hashing (ESH)

General description. An extendible signature hashing (ESH) index [16] is based on extendible hashing [10]. It 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. 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 (Fig. 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 too large directory, which in turn would slow down subset and superset queries considerably.

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 subsets (supersets) of $sig_d(Q)$ with the algorithm by Vance and Maier [28]. Then we access the corresponding buckets sequentially, 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 [33]. 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 [16].

Implementation details. We have limited the size of the directory to 2^{20} entries. As this directory obviously does not fit on a 4K page, we implemented a hierarchical directory (Fig. 6). On the root page, which is always kept in main memory, we evaluate the first 10 bits of the query signature. Then we branch to the corresponding directory page to evaluate the other 10 bits of the query signature and fetch the corresponding bucket reference. A bucket page contains a reference to the next page of the bucket (in case of overflow records), a pointer to the end of the data on this page, the local depth of the bucket, and the signatures and references of all data items belonging to this bucket.

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 [28]. It is repeated on the left-hand side of Fig. 7. When executed, x passes through all possible subsets of $sig(s)$. Its counterpart on the right-hand side generates all supersets (˜ stands for the *bitwise complement*, - *for the two-complement*). Instead of printing the set, we could also process it in any other conceivable way. Before working with the sets generated by the right-hand side algorithm, we have to calculate the bitwise complement of bit vector x.

3.2 Inverted files

General description. An inverted file (Fig. 8) consists of a directory containing all distinct values in the domain D and a list for each value consisting of the references to data items whose set-valued attribute contains this value. For an overview on traditional inverted files, see [20,26]). As is done frequently, we hold the search values of the directory in a B-tree. However, we modify the lists by storing the cardinality of the set-valued attribute with each data item reference. This enables us to answer queries efficiently by using the cardinalities as a quick pretest.

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 is not equal to 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 is not equal to the cardinality of its set, we eliminate that reference. We can do this because this reference then 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. Data items with empty set-valued attributes need special treatment, as they cannot be assigned to values in the directory. We store references to these data items in a separate list.

We use several well-known techniques to increase the performance of inverted lists. In order to reduce the size of the lists, we compress them. We keep the lists sorted and encode the gaps using lightweight techniques [29]. 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 [24,31].

Implementation details. 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

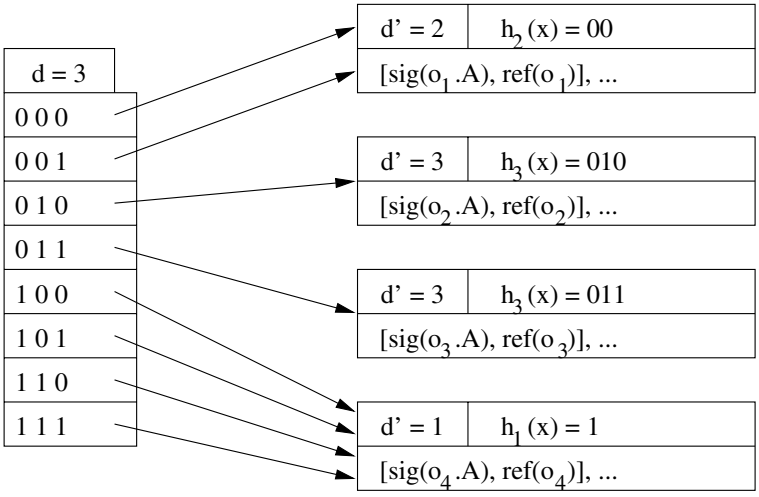


Fig. 5. Extendible signature hashing (ESH)

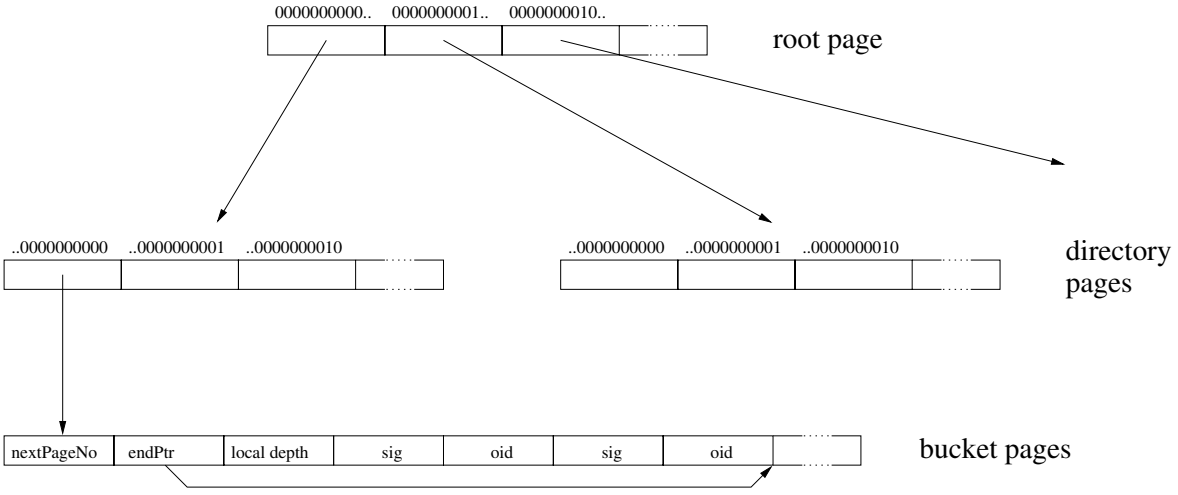


Fig. 6. Pages of an ESH index

```
x = sig(s) & -sig(s);

print(x);

while(x)
{
   x = sig(s) & (x - sig(s));
   print(x);

}
```

```
x = ~sig(s) & -~sig(s);

print(x);

while(x)
{
   x = ~sig(s) & (x - ~sig(s));
   print(x);

}
```

Fig. 7. Quick generation of subsets/supersets

by search keys. The size of search keys is 4 bytes, the size of references 8 bytes (Fig. 9).

We distinguish two different types of lists, compressed and uncompressed. Lists containing less than 8 OIDs (unique object identifiers) are not compressed in order to avoid overhead. An uncompressed list merely consists of OIDs and set cardinalities. Compressed lists contain additional data describing the compression parameters and compressed OIDs and set cardinalities.

4 Environment of the experiments

When comparing index structures, there are several principal avenues of approach: analytical approach, mathematical modeling, simulations, and experiment [34]. 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 nonuniform data distribution and average case behavior. In the following sections, we present the system parameters and the specification of our experiments.

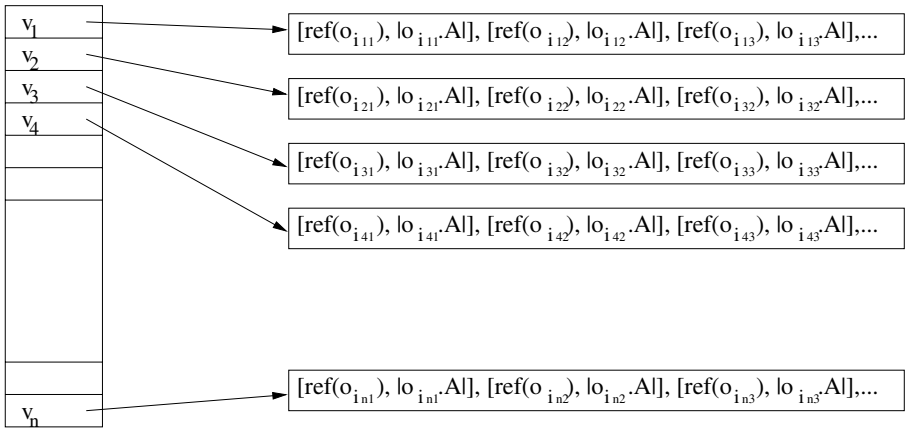


Fig. 8. Inverted file

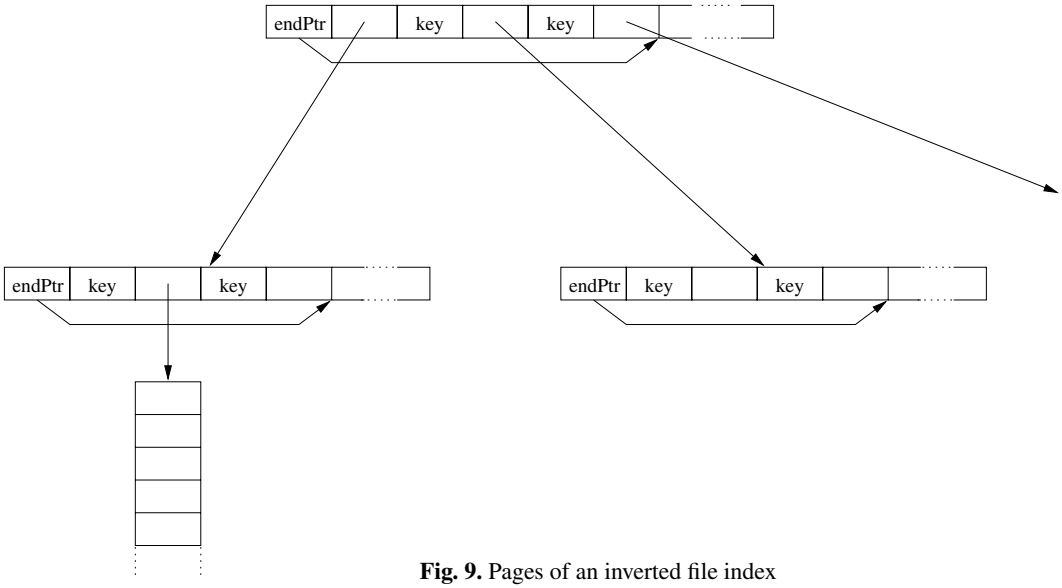


Fig. 9. Pages of an inverted file index

4.1 Storage manager

The index structures were implemented atop the storage manager EOS, release 2.2, using the C++ interface of the manager. EOS provides key facilities for the fast development of high-performance DBMSs [4,5]. We exploited only a small part of the offered features, namely, standard EOS objects and plain database pages. EOS objects are stored on slotted pages and are identified by a unique object identifier (OID) with a size of 8 bytes. We stored all data items with set-valued attributes in EOS objects. Plain database pages (with a size of 4096 bytes) belong to a particular storage area (a raw disk partition in our case) and do not contain any control information. We used these pages to implement our index structures.

4.2 System parameters

The experiments were conducted on a lightly loaded UltraSparc2 with 256-MB main memory running under Solaris 2.6. The total disk space amounted to 10 GB. 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 4-K 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 MB of data between the queries. Within a single run, the buffer was large enough to prevent accessing pages more than once.

4.3 Synthetic data

We used three different sets of data in our experiments. Two of them were generated synthetically, one set contains real data. In this section, we describe the creation process of the artificial data, and in Sect. 4.4 we specify the real data.

Table 2. Parameters for generation of databases

Parameter	*Symbol*	*Min value*	*Max value*
Database cardinality	$\|O\|$	50000	250000
Set cardinality	$\|o_i.A\|$	5	15
Domain cardinality	$\|D\|$	200	100000

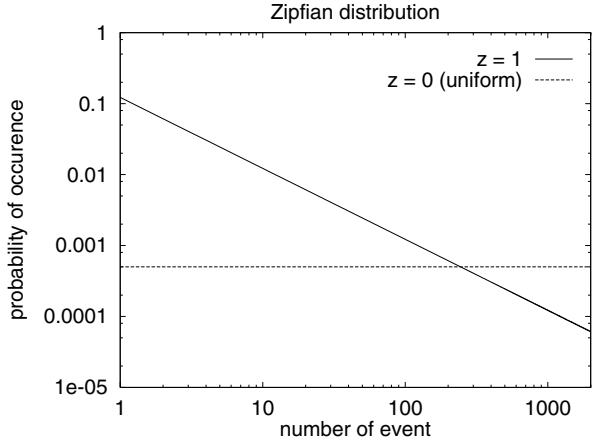


Fig. 10. Zipf distribution

4.3.1 Generating data

We generated many different databases 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 2. Each data item with a set-valued attribute was stored on a separate page to eliminate any clustering effects.

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 skewed data we decided to use a Zipf distribution (with $z = 1$) since various naturally occurring phenomena exhibit a certain regularity that can be described by it, e.g., word usage or population distribution [25]. A discrete Zipf distribution is defined by $P_z(x)$, which denotes the probability of event x occurring, with $x \in \{1, 2, \ldots, n\}$.

$$P_z(x) = \frac{1}{x^z} \cdot \frac{1}{H_n} \tag{2}$$

with

$$H_n = \sum_{i=1}^{n} \frac{1}{i^z} \tag{3}$$

Note that for $z = 0$ we obtain a discrete uniform distribution. Figure 10 shows the graphs of $P_1(x)$ and $P_0(x)$ for n=2000.

4.3.2 Generating queries

Usually, queries that are meaningful do not return an empty answer set. So in order to guarantee hits during the query evaluation, we generated the query sets in a special way. For equality queries, we took as query sets the values of set-valued attributes of data items that were actually inserted into the database. For subset queries, we used set-valued attributes of inserted data items with the highest set cardinality (i.e., 15 elements) and randomly removed elements from these query sets. For superset queries, we selected set-valued attributes of inserted data items with the lowest set cardinality (i.e., 5 elements) and randomly added elements from the domain D to these query sets.

Query selectivity is not a parameter we look at explicitly. It is determined implicitly by the cardinality of the query set, the cardinality of the data sets, the cardinality of the domain, and the distribution of the data. Furthermore, query selectivity directly determines the number of qualifying data items (i.e., the cardinality of the answer set).

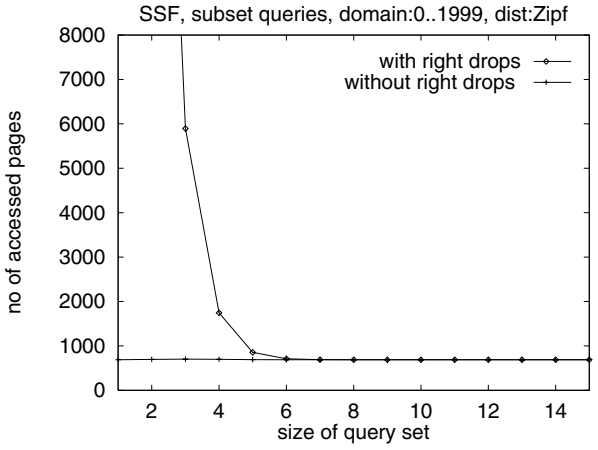


Fig. 11. Considering accesses to right drops in retrieval costs

4.4 Real data

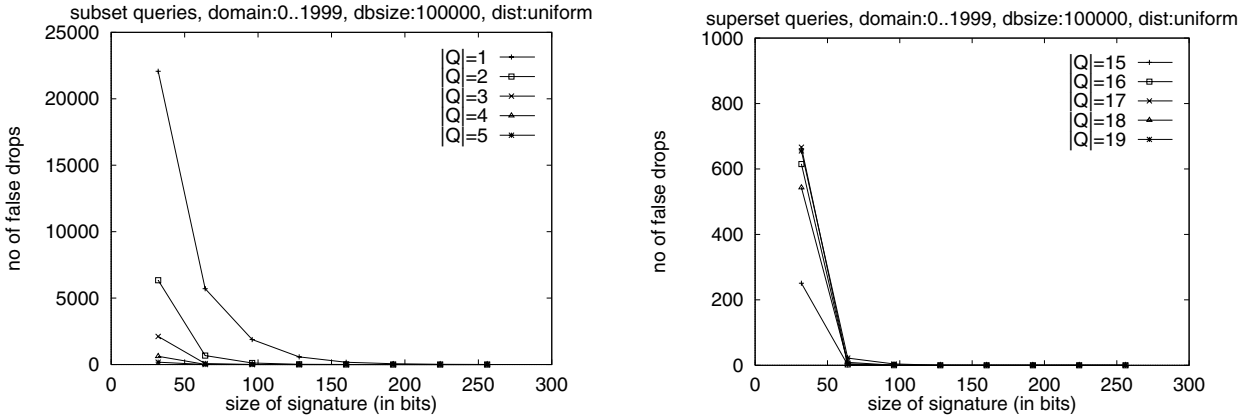
4.4.1 Building the database

We built a dictionary containing all different words from the Bible in English, Danish, and Swedish languages, "Don Quijote" in Spanish, "Le Tour du Monde en Quatre-Vingts Jours" in French, and "Faust" in German (to model a translation application). However, we did not index the words directly but generated a set of 3-grams for each word, e.g., "along" has the set { alo, lon, ong }. N-grams are usually used for queries with partially specified terms [31].

Altogether we inserted 100,916 sets with 14,384 different 3-grams into our index structures.

4.4.2 Generating queries

For queries with equality predicates, we took words from our dictionary randomly, generated their sets of 3-grams, and searched for those sets. For subset queries, we used the 3-gram sets of all prefixes of randomly selected words as query sets. The query sets for superset queries were unions of 3-gram sets of three words picked randomly from the dictionary.

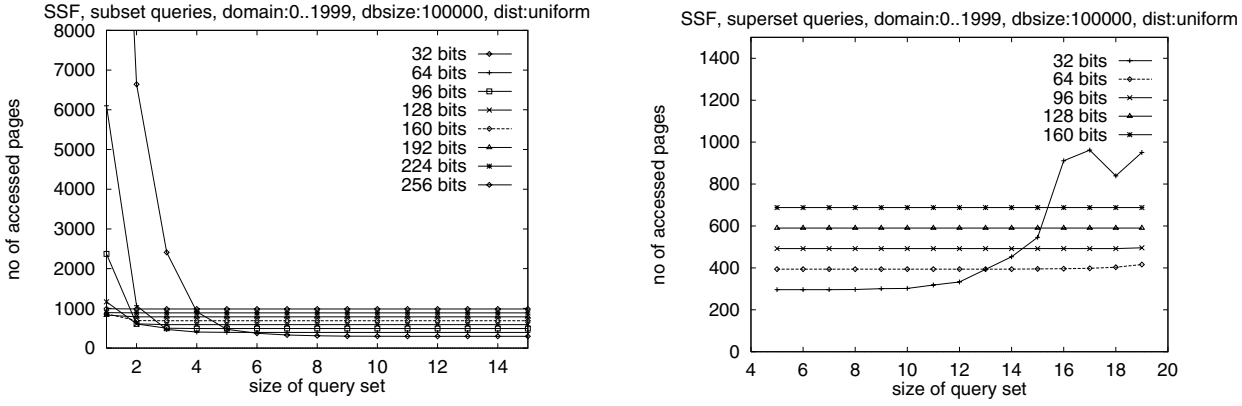


(a) Subset queries (b) Superset queries

Fig. 12a,b. Number of false drops for subset and superset queries

Table 3. Size of SSF index in 4K pages

Signature size (in bits)	32	64	96	128	160	192	224	256
Index size (in 4K pages)	296	394	492	590	688	786	887	985



(a) Subset queries (b) Superset queries

Fig. 13a,b. Number of page accesses for subset and superset queries (SSF)

4.5 Unit of measurement

In our experiments, we decided to measure the number of page accesses rather than elapsed time. We did so for the following reasons. First, we wanted to have a machine-independent unit of measurement. Second, the costs of accessing the qualifying data items are the same for all index structures. Therefore, when presenting the results of our experiments in Sects. 6 and 7, we focus on the additional work that each index structure has to invest to fetch those data items (Fig. 11 shows the effect of including accesses to right drops in the retrieval costs). When counting page accesses, we merely need to subtract the number of accesses to fetch right drops from the total number of page accesses. It would be much more difficult to do this with time measurements.

5 Tuning signature-based indexes

Signature-based indexes have to be tuned carefully to yield optimal performance. That is, the parameters k (the number of bits set per item) and b (the width of a signature) need to be chosen properly. Before comparing the indexes with each other, we determined the optimal parameters for each query type in our experimental environment. For each signature-based index structure, we increased the width of the signature until no more performance improvement could be perceived. The optimal value for k was determined by using the following formulas taken from [21]:

$$k_{opt=} = \frac{b \ln 2}{|Q|} \tag{4}$$

$$k_{opt\subseteq} = \frac{b \ln 2}{|o_i.A|} \tag{5}$$

$$k_{opt\supseteq} = \frac{b \ln 2}{|Q|} \tag{6}$$

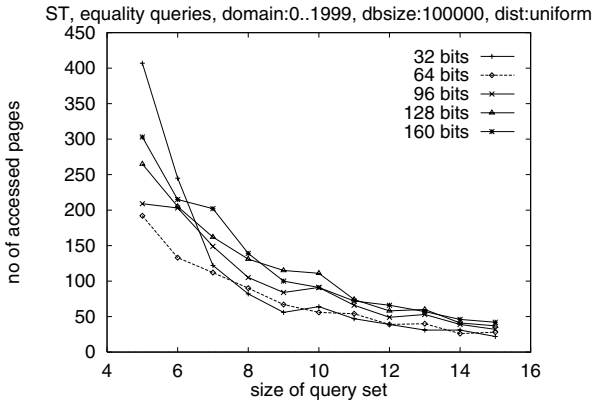
Our observations are presented in the following sections.

Table 4. Size of ST index in 4K pages

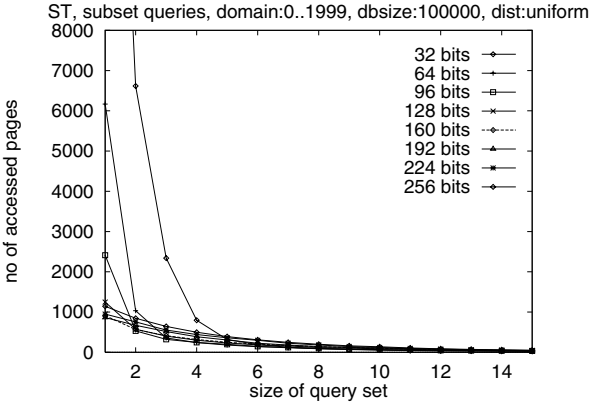
Signature size (in bits)	32	64	96	128	160	192	224	256
Index size (in 4K pages)	436	580	731	882	1010	1173	1296	1443

Table 5. Size of ESH index in 4K pages

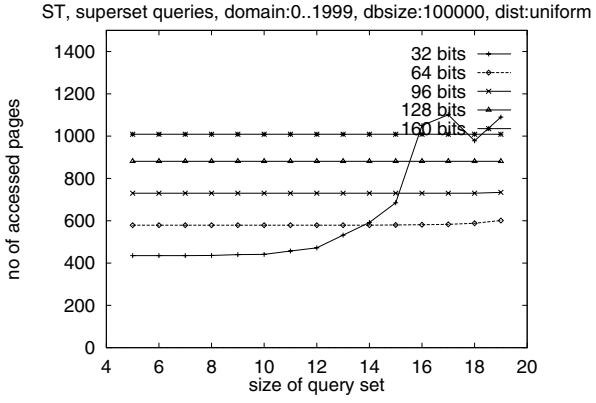
Signature size (in bits)	32	64	96	128	160	192	224	256
Index size (in 4K pages)	872	1160	1457	1749	1992	2096	2156	2369



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 14a–c. Number of page accesses for equality, subset, and superset queries (ST)

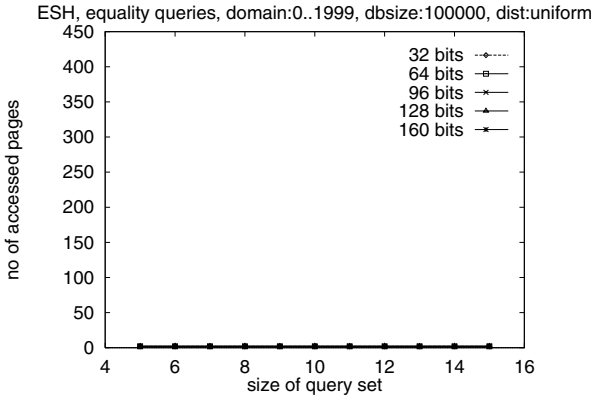
5.1 Sequential signature files

Increasing the width of the signatures has two effects on an SSF index. On the one hand, it reduces the number of false drops, thus improving the performance. On the other hand, it increases the size of the index, leading to greater costs for scanning the file.

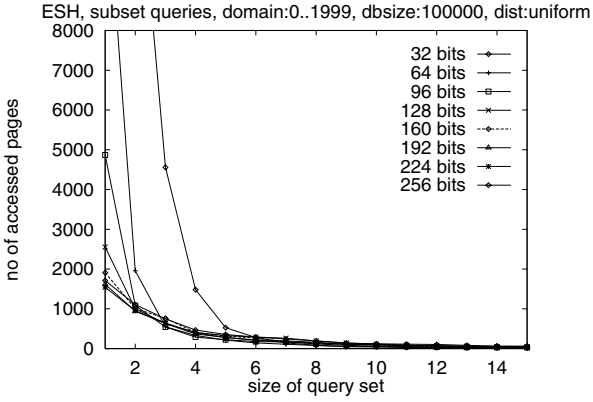
The size of an SSF index increases proportionally to the size of the used signatures. This can be seen in Table 3.

Let us now look at the query performance. For equality queries, we had no false drops, regardless of the used signature size (32, 64, 96, 128, 160, 192, 224, or 256 bits), as the false drop probability is very low. Note that we do not guarantee uniqueness of keys. For subset and superset queries, the number of false drops is depicted in Fig. 12. As can be clearly seen, an increase in the signature width lowers the number of false drops. The number of false drops also depends on the cardinality of the query set. Small query sets lead to "light" signatures. For subset queries, this increases the number of false drops (Fig. 12a). Large query sets show analogous behavior for superset queries (Fig. 12b).

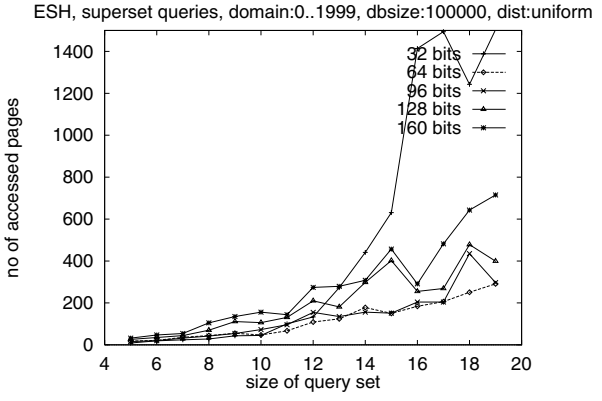
The consequences for equality queries are clear. An index with a signature width of 32 bits suffices. For subset and superset queries, we have to find a break-even point. Figure 13 shows the number of page accesses for subset and superset queries with different signature widths. Before drawing conclusions we should bring to mind that subset queries with small query sets and superset queries with large query sets are probably going to be the predominant queries. Unfortunately, these are the cases where signatures show weak performance. For superset queries (Fig. 13b) we found that a signature width of 64 bits gave us the best overall performance. When averaging the page accesses for all query set sizes, the index with a signature of 64 bits comes out on top. When taking into consideration



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 15a–c. Number of page accesses for equality, subset, and superset queries (ESH)

that larger query sets are more important, a signature size of 64 bits is by far superior to the other signature sizes. For subset queries (Fig. 13a), the choice is harder to make. Applying a signature size of 96 or 128 bits yields the best average performance. However, this does not consider well enough the important cases, i.e., small query sets. A query set cardinality of 1 is crucial, as it represents containment queries (see Sect. 2). Therefore, we chose a signature size of 160 bits.

5.2 Signature trees

We observe similar effects for an ST index as for an SSF index. A larger signature width means fewer false drops, but the nodes of an index tree hold fewer signatures. We omit figures for the number of false drops, as these are identical to those in Sect. 5.1. We cannot eliminate false drops with index structures because they are inherent to signatures.

Table 4 shows the size of an ST index depending on the signature size. Larger signatures mean that fewer signatures will fit on a page of the tree, which leads to a larger index.

When evaluating the query performance, we also have to consider equality queries this time (unlike for SSF) because searching the inner nodes of signature trees involves subset predicates (Sect. 3.1.2). The optimal size for the signatures is 64, 160, and 64 bits for equality, subset, and superset queries, respectively.

5.3 Extendible signature hashing

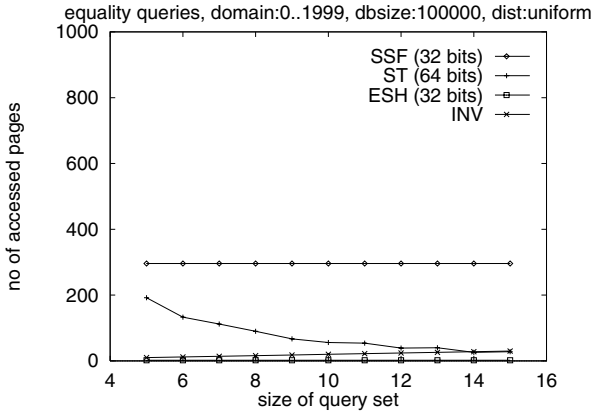
As ESH cannot filter out false drops either, the number of false drops is the same as that for SSF (Fig. 12).

In order to restrict an otherwise limitless growth of the directory, we do not allow the directory to grow beyond 2^{20} entries (Sect. 3.1.3). In this case, we use overflow buckets instead of splitting the directory any further. For large signatures, we have to allocate more overflow buckets. This means that the internal fragmentation in the buckets decreases, which leads to a flattening growth of the index size (Table 5).

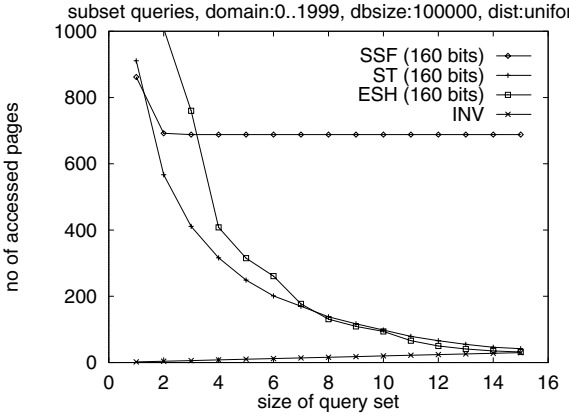
Equality queries have a very low number of false drops, and only one bucket needs to be accessed during the evaluation of a query. Therefore, a long chain of overflow buckets is detrimental to performance. The length of a chain depends on the size of the signature. Different signature sizes yield very similar results for equality queries (Fig. 15a). Therefore, we choose a signature size of 32 bits, as this results in the smallest index structure without giving up performance. The query performance of subset and superset queries is in accordance with the performance of the other index structures, i.e., signature sizes of 160 and 64 bits, respectively, yield the best results.

6 Results for uniformly distributed data

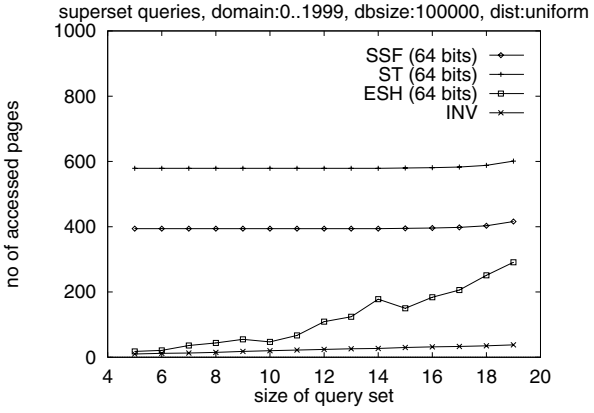
We present an excerpt of the results of our extensive experiments emphasizing query evaluation speed and index size. In



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 16a–c. Retrieval costs for uniformly distributed data, varying query set cardinality

this section, we deal with uniformly distributed data and in Sect. 7 with skewed data. As the unit of measurement we use the number of page accesses. Because we are interested in the overhead of each index, we count the page accesses needed to traverse an index structure during query evaluation. Moreover, for signature-based index structures, we take into account the accesses to false drops by subtracting the page accesses for qualifying items from the total page accesses. This means we do not consider page accesses for accessing qualifying data items (as these are the same for all index structures).

6.1 Retrieval costs

One of the most important aspects of an index is the cost of finding and retrieving the answer to a query. We investigated the influence of several parameters on these costs. In detail, these were cardinality of query sets, database size, and domain size.

6.1.1 Influence of query set cardinality

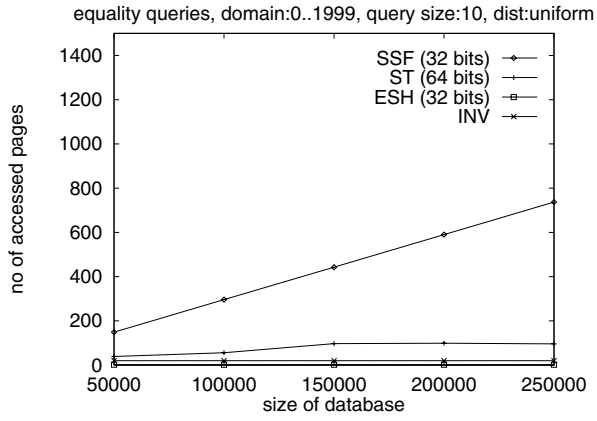
Results of the experiments where the query set cardinality is varied are found in Fig. 16. As can be seen, the influence of query set cardinality on the retrieval cost of equality queries (Fig. 16a) is marginal. The only major exception is ST, as a subset query has to be performed on the inner nodes of the tree. Subset queries with a small query set cardinality are a difficult case, as we will see. There is also a slight increase in retrieval costs for inverted files for equality queries, as more lists have to be searched. In summary, we can say that the hash table index is unbeaten for equality queries, as it only takes two page accesses to reach a data item.

The results for subset queries are depicted in Fig. 16b. Here the signature-based index structures have a severe disadvantage. Usually, subset queries are formulated with query sets having a small cardinality. In these cases, ST and ESH show their worst behavior as ST needs to traverse many branches and ESH needs to visit almost all buckets. So for subset queries, the inverted file index reigns supreme.

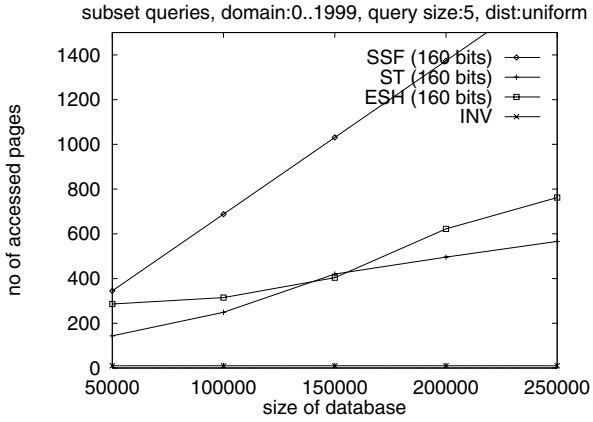
For superset queries, the important cases are query sets with a large cardinality. As can be seen in Fig. 16c, the cardinality of query sets does not have much influence on SSF and the inverted file index. ST is not suited at all for superset queries, and ESH has problems with large query set cardinalities. Again the inverted file index is superior.

6.1.2 Influence of database size

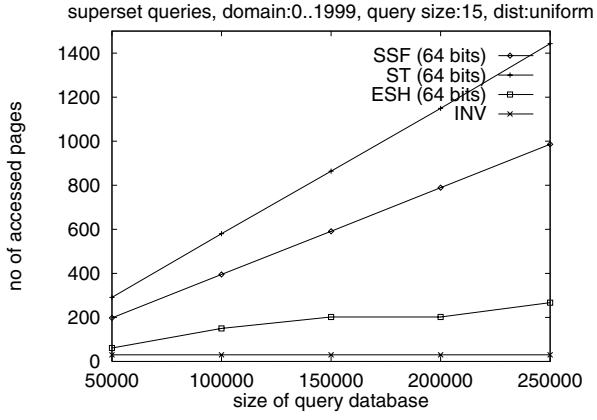
Another important parameter we investigated is the size of the database (in number of data items). We wanted to know if the index structures would scale gracefully. The results of our



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 17a–c. Retrieval costs for uniformly distributed data, varying database size

experiments can be seen in Fig. 17 ((a) equality queries, (b) subset queries, and (c) superset queries). The retrieval costs for SSF are proportional to the database size since the whole index has to be scanned. Given the different indexes for each query type (see Sect. 5), the costs for equality, subset, and superset queries differ.

For equality and subset queries, the retrieval costs of ST seem to grow sublinearly, while for superset queries they grow linearly due to the fact that in the this case almost all nodes are traversed.

ESH is unbeaten for equality queries as the retrieval cost is constant. For subset and superset queries, we noticed that the number of generated subqueries increases steadily on account of the increasing size of the directory.

The retrieval costs for an inverted file index grow very slowly for all query types. Thanks to the optimization techniques described in Sect. 3.2, we can keep the retrieval cost almost constant. Clearly, the inverted file index shows the best overall scaling behavior of the examined index structures.

6.1.3 Influence of domain size

In Fig. 18, we present the influence of domain size on the retrieval costs. Parts (a), (b), and (c) show the results for equality, subset, and superset queries, respectively. As can clearly be seen, this parameter affects the retrieval costs minimally.

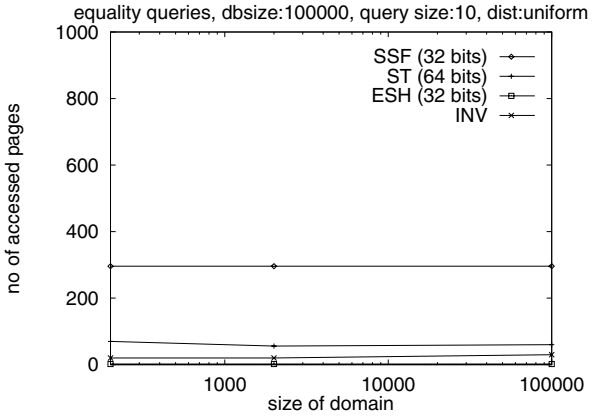
Table 6. Index size in 4K pages for uniformly distributed data, varying database size

Database size	50000	100000	150000	200000	250000
SSF (32 bit)	149	296	443	590	737
SSF (64 bit)	198	394	590	786	982
SSF (160 bit)	345	688	1031	1374	1717
ST (64 bit)	292	580	864	1147	1440
ST (160 bit)	502	1010	1503	2024	2554
ESH (32 bit)	435	872	1271	1747	2166
ESH (64 bit)	573	1160	1639	1956	2390
ESH (160 bit)	1045	1992	2663	3247	3962
Inverted file	268	530	788	1046	1303

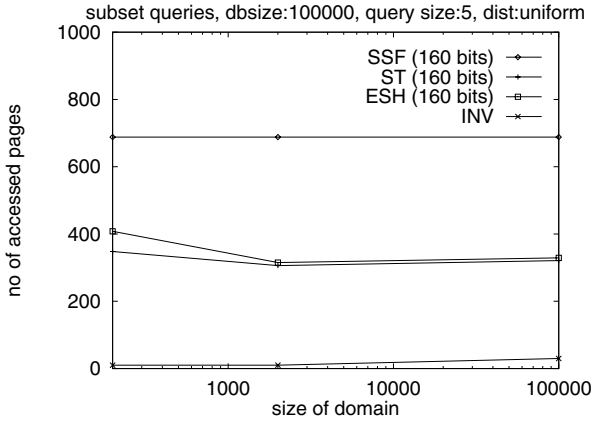
Even for an inverted file index, which has to manage larger numbers of lists for increasing domain size, we cannot detect a noticeable increase in retrieval costs.

6.2 Index size

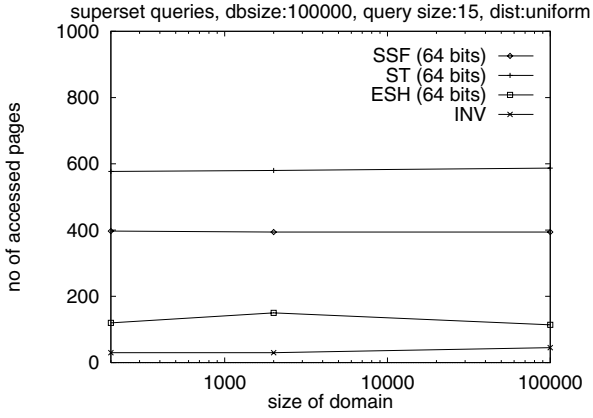
Another important aspect to consider when investigating index structures is the size of these structures. We will look in turn at the influence of database size and domain size on the index size.



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 18a–c. Retrieval costs for uniformly distributed data, varying domain size

6.2.1 Influence of database size

Table 6 shows the experimental results pertaining to the scalability of the index structures. Except for the ST index, which shows some irregularities in growth, all index structures grow linearly with the database size. As was already seen in Sect. 5, the signature size also has a direct influence on the index size. The compression of inverted files makes them competitive in comparison to the signature-based index structures. An earlier uncompressed version of inverted files we used was about eight to nine times larger, which would have been unacceptable.

6.2.2 Influence of domain size

Table 7 shows the influence of domain size on the index size. As expected for the signature-based index structures, this influence is marginal. For inverted files, we can say that the smaller the domain, the better the space requirement. The obvious reason for this is that for each value appearing in a set, a list has to be allocated. The total number of lists could be reduced by merging lists of values that appear infrequently into one list. This would lead to a better compression of the small lists. However, the retrieval costs would also rise as false drops would be introduced, which would have to be eliminated.

Table 7. Index size in 4K pages for uniformly distributed data, varying domain size

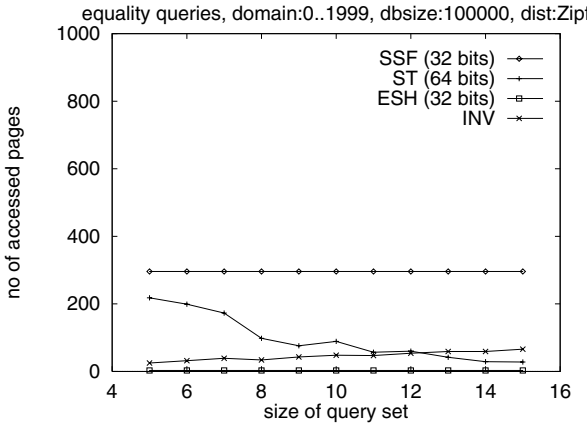
Domain size	200	2000	100000
SSF (32 bit)	296	296	296
SSF (64 bit)	394	394	394
SSF (160 bit)	688	688	688
ST (64 bit)	575	580	587
ST (160 bit)	1012	1010	1018
ESH (32 bit)	874	872	886
ESH (64 bit)	1093	1160	1199
ESH (160 bit)	1639	1992	1992
Inverted file	369	530	1559

7 Results for skewed data

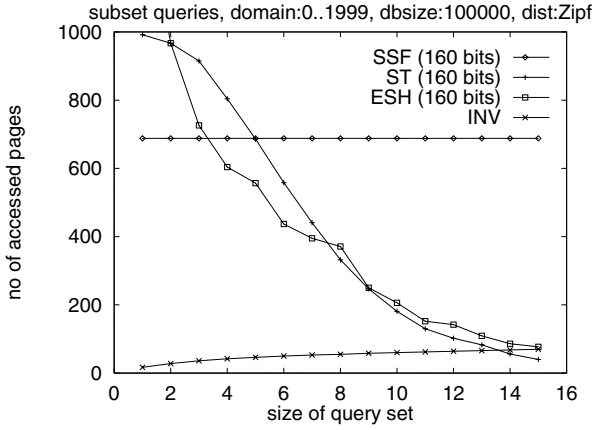
In this section, we present the results for skewed data. The first part of this section deals with retrieval costs, the second part with index size.

7.1 Retrieval costs

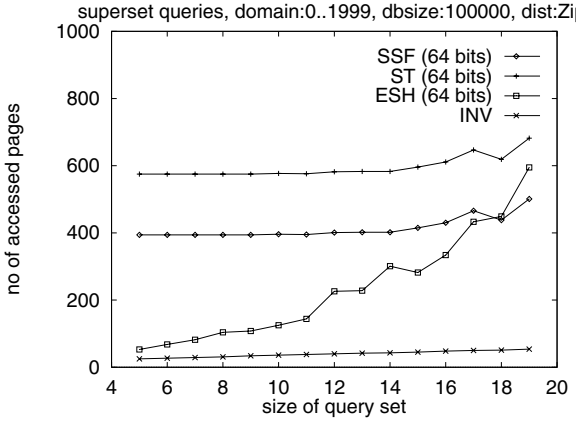
Analogously to uniformly distributed data we evaluated the influence of query set cardinality, database size, and domain size on retrieval costs.



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 19a–c. Retrieval costs for skewed data, varying query set cardinality

Table 8. Index size in 4K pages for real data

SSF			ST		ESH			INV
32 bits	64 bits	160 bits	64 bits	160 bits	32 bits	64 bits	160 bits	
297	396	693	567	984	702	884	1470	488

7.1.1 Influence of query set cardinality

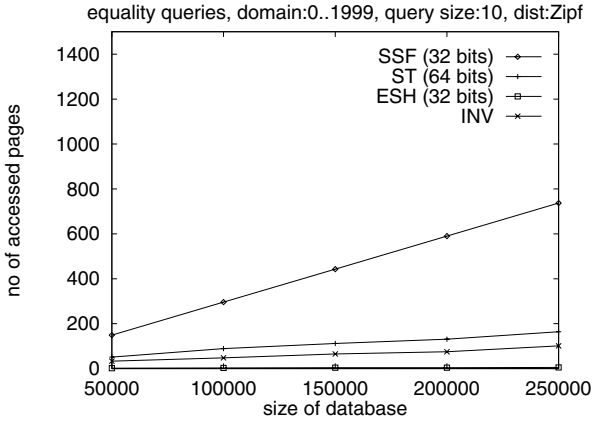
Varying the cardinality of the query sets yields the result depicted in Fig. 19. For equality queries, we obtained results very similar to those for uniformly distributed data. SSF and ESH are not influenced at all, while ST shows a slight increase in costs. We noticed a small decrease in performance for the inverted file index. This stems from the fact that the lists for the most common values are longer than the lists in the case of uniformly distributed data. Since these values also appear more often in queries, they are searched for more often during query evaluation. As for uniformly distributed data, the hash index is still the fastest index for equality queries.

For subset queries (Fig. 19b), the performance of all index structures except SSF deteriorates. In the ST index, more branches have to be traversed during query evaluation, while in the ESH index more buckets are accessed. The decrease in performance for inverted files has the same reasons already mentioned for equality queries. In spite of the deterioration, the inverted file index is still the best index for subset queries.

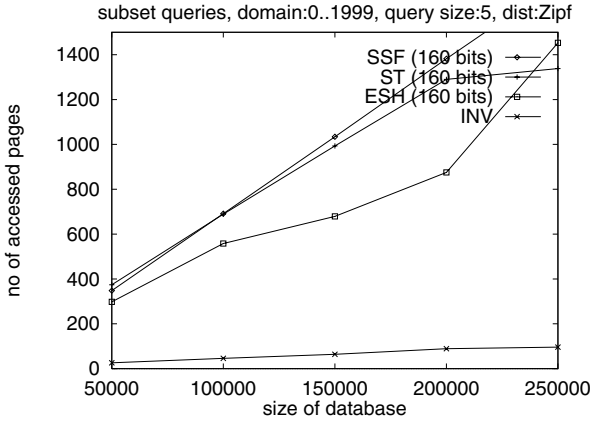
The insights for SSF, ESH, and inverted files for subset queries also apply to superset queries. ST shows no further deterioration for superset queries for skewed data because it already displays worst-case behavior for superset queries for uniformly distributed data. So for superset queries, the inverted files are also the index of choice.

7.1.2 Influence of database size

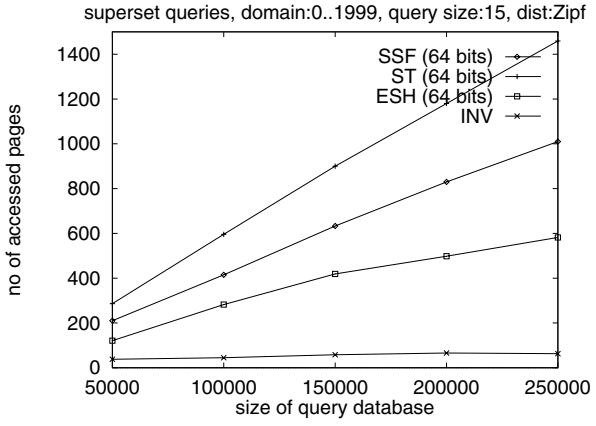
The influence of database size on the index structures is shown in Fig. 20. We observed a linear growth of retrieval costs for SSF for all query types, i.e., skewed data has no influence whatsoever on the internal structure of SSF. In both cases (uniformly distributed and skewed data), SSF consists of sequential files with the same length, which are scanned sequentially during query evaluation. The performance of the other index structures for skewed data decreases. Notable exceptions to this rule are ESH for equality queries (best case for hash-based indexes) and ST for superset queries (worst case



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 20a–c. Retrieval costs for skewed data, varying database size

for ST). However, inverted files still show the best overall behavior, as the performance of other index structures deteriorates dramatically in some cases (ST and ESH for subset queries and ESH for superset queries).

7.1.3 Influence of domain size

We summarized the results for the influence of the domain size in Fig. 21. Again we have a parameter that has no influence on SSF. For the other signature-based index structures and inverted files, the performance deteriorates because small domains amplify the skewing of data as the variety of occurring values is lowered and the distribution of these values is less balanced (the probability of appearance for the most common value for $|D| = 200$ is 0.17, for $|D| = 2000$ 0.12, and for $|D| = 100000$ 0.08). However, the most important point is that the performance of inverted files does not worsen significantly for large domains.

7.2 Index size

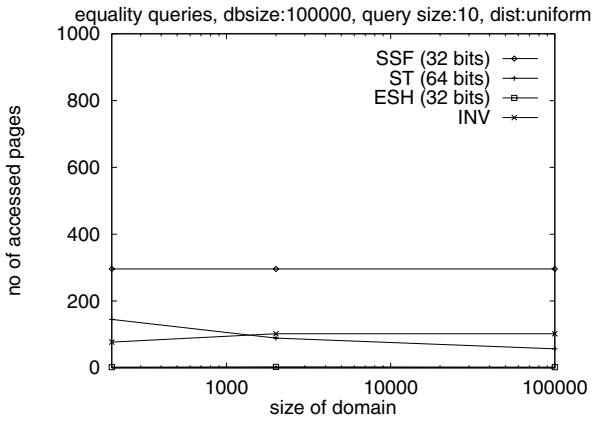
In the following two sections, we illustrate the effects of skewed data on the index size. We vary the size of the database and the size of the domain.

Table 9. Index size in 4K pages for skewed data, varying database size

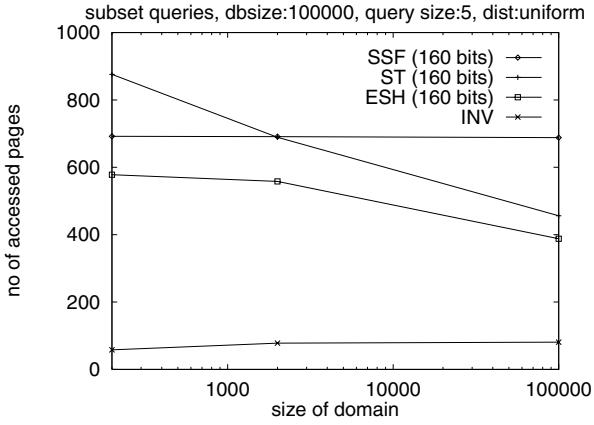
Database size	50000	100000	150000	200000	250000
SSF (32 bit)	149	296	443	590	737
SSF (64 bit)	198	394	590	786	982
SSF (160 bit)	345	688	1031	1374	1717
ST (64 bit)	274	576	858	1137	1432
ST (160 bit)	492	993	1473	1964	2457
ESH (32 bit)	489	790	1086	1377	1654
ESH (64 bit)	473	859	1263	1642	2041
ESH (160 bit)	809	1490	2195	2884	3564
Inverted file	178	340	505	668	830

7.2.1 Influence of database size

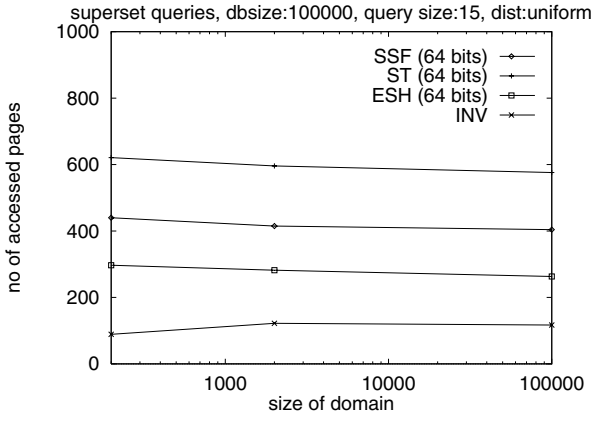
In Table 9, we depict the influence of skewed data on the size of the index structures when varying the database size. The values for SSF for skewed data correspond to those for uniformly distributed data (Fig. 6). When comparing the other index structures, we notice that the index size is smaller for skewed data. In the case of ST, skewed data lead to a higher fill degree of the nodes, which makes the index smaller. For ESH, skewed data cause an unbalanced directory and many splits. Since the size of the directory is limited, we have to introduce overflow pages. Alongside many disadvantages, however, overflow pages have one advantage: the pages are better



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 21a–c. Retrieval costs for skewed data, varying domain size

Table 10. Index size in 4K pages for skewed data, varying domain size

Domain size	200	2000	100000
SSF (32 bit)	296	296	296
SSF (64 bit)	394	394	394
SSF (160 bit)	688	688	688
ST (64 bit)	576	576	567
ST (160 bit)	982	993	988
ESH (32 bit)	762	790	807
ESH (64 bit)	848	859	891
ESH (160 bit)	1468	1490	1539
Inverted file	266	340	961

utilized. This is also the case for the inverted files, which need not fear comparison with the other index structures. The compression rate of the lists is better than for uniformly distributed data because we have fewer, larger lists.

7.2.2 Influence of domain size

Table 10 shows the influence of domain size on the index size. As usual, SSF is insensitive to changes in domain size. For the other index structures, we can see that skewed data are stored more compactly but are not necessarily better suited for efficient retrieval. So the reason for the deterioration of the retrieval costs is not the larger index size, but an inferior internal organization of the indexes. Obviously, the larger the domain, the larger the inverted file will be, but the inverted file index is still competitive when compared to the other index structures.

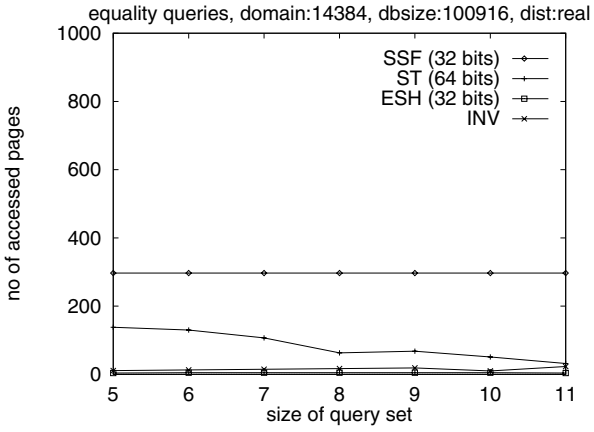
8 Results for real data

In this section, we present the results from our experiments with real data. As our dictionary has a fixed size of 100,916 words with a domain of 14,384 3-grams, we investigate the influence of the cardinality of the query set. Additionally, we look at the size of each index structure.

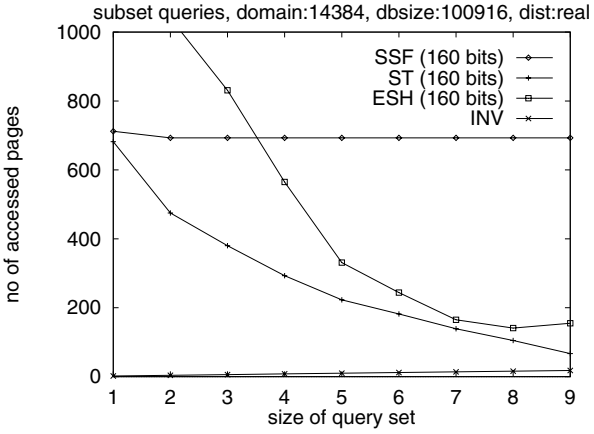
8.1 Retrieval costs

Figure 22 shows the retrieval costs for real data. Note that the size range of the query sets differs from the diagrams in Sects. 6 and 7. For values outside the ranges, we did not have enough query sets (more than ten) to guarantee representative results.

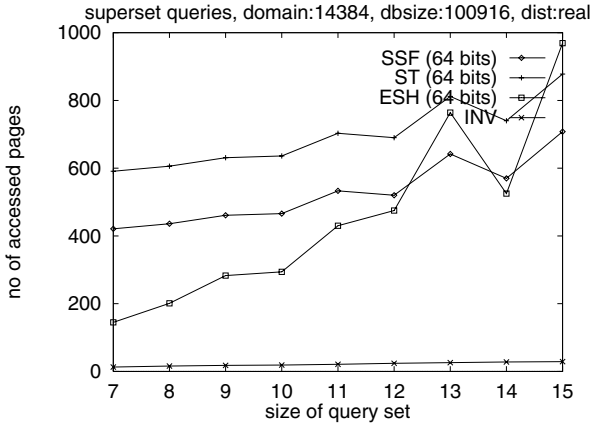
For equality queries (Fig. 22a), we attain results similar to those for synthetic data. ESH is fastest, followed by inverted files, ST, and (last) SSF. We do not have many new insights on subset queries (Fig. 22b) either. It is interesting to note, however, that ESH is worse than ST for all query set sizes in



(a) Equality queries



(b) Subset queries



(c) Superset queries

Fig. 22a–c. Retrieval costs for real data, varying query set cardinality

this case (in contrast to synthetic data, where ESH was able to best ST in some cases). We obtained the most deviating results for superset queries (Fig. 22c). Retrieving real data poses a greater problem for signature-based index structures than retrieving synthetic data. Especially the performance of ESH deteriorates, being the worst access method for query set size 15.

The inverted file index shows a strong performance for real data and clearly is the index of choice for this important test case.

8.2 Index size

Table 8 shows the size of the different index structures for real data. SSF shows no significant change when compared to synthetic data (it is a couple of pages larger, because we have 100,916 data items instead of 100,000). For real data, we have the smallest ST of all three data distributions. At first glance, it looks like real data are even more skewed than the Zipf distributed synthetic data. The sizes of ESH and inverted files, however, do not indicate this as clearly. Overall, we can say that the indexes (with the exception of SSF) are smaller for real data than for uniformly distributed synthetic data but show differences in the results for skewed synthetic data.

9 Conclusion

We studied the performance of several different index structures for set-valued attributes of low cardinality for three different query types. For that reason, we implemented sequential signature files, signature trees, extendible signature hashing, and B-tree-based inverted files in order to conduct extensive experiments. We refitted the index structures to support the evaluation of queries containing set-valued predicates (namely, equality, subset, and superset predicates). The inverted file index clearly dominated the field, though for equality queries the hash-based ESH was faster. Generally, the signature-based index structures have difficulties with skewed data and some important query cases (small query sets for subset queries, large query sets for superset queries). The inverted file index showed robustness for skewed data, and we were able to keep the space demands reasonable. Zobel, Moffat, and Ramamohanarao made a similar observation for the special case of text retrieval while comparing inverted files to signature files [35].

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

Acknowledgements. We thank the anonymous reviewers for their helpful hints on an earlier version of this paper. Thanks also goes to Simone Seeger for her help in preparing the manuscript.

References

1. Ash JE, Chubb PA, Ward SE, Welford SM, Willet P (1985) Communication, storage and retrieval of chemical information. Ellis Horwood, Chichester, UK
2. Bairoch A, Apweiler R (1996) The SWISS-PROT protein sequence data bank and its new supplement TrEMBL. Nucleic Acids Res 24(1):21–25
3. Bertino E, Kim W (1989) Indexing techniques for queries on nested objects. IEEE Trans Knowledg Data Eng 1(2):196–214
4. Biliris A (1992) An efficient database storage structure for large dynamic objects. In: Proceedings of the 8th international conference on data engineering, Tempe, AZ, February 1992, pp 301–308
5. Biliris A, Panagos E (1994) EOS user's guide. Technical report, AT&T Bell Laboratories, Florham Park, NJ
6. Böhm K, Rakow TC (1994) Metadata for multimedia documents. SIGMOD Rec 23(4):21–26
7. Cattell R (ed) (1997) The object database standard: ODMG 2.0. Morgan Kaufmann, San Francisco
8. Claussen J, Kemper A, Moerkotte G, Peithner K (1997) Optimizing queries with universal quantification in object-oriented and object-relational databases. In: Proceedings of the 23rd VLDB conference, Athens, Greece, August 1997, pp 286–295
9. Deppisch U (1986) S-tree: a dynamic balanced signature index for office retrieval. In: Proceedings of the 1986 ACM conference on research and development in information retrieval, Pisa, September 1986, pp 77–87
10. Fagin R, Nievergelt J, Pippenger N, Strong HR (1979) Extendible hashing – a fast access method for dynamic files. ACM Trans Database Sys 4(3):315–344
11. Faloutsos C, Christodoulakis S (1984) Signature files: an access method for documents and its analytical performance evaluation. ACM Trans Office Inform Sys 2(4):267–288
12. Fasman KH, Letovsky SI, Cottingham RW, Kingsbury DT (1996) Improvements to the GDB human genome data base. Nucleic Acids Res 24(1):57–63
13. Grobel T, Kilger C, Rude S (1992) Object-oriented modelling of production organization. In: Tagungsband der 22. GI-Jahrestagung, Karlsruhe, September 1992. Informatik Aktuell, Springer, Berlin Heidelberg New York, (in German)
14. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, June 1984, Boston, pp 47–57
15. Hellerstein JM, Pfeffer A (1994) The RD-tree: an index structure for sets. Technical Report 1252, University of Wisconsin at Madison
16. Helmer S (1997) Index structures for databases containing data items with set-valued attributes. Technical Report 2/97, Universität Mannheim
http://pi3.informatik.uni-mannheim.de
17. Ishikawa Y, Kitagawa H, Ohbo N (1993) Evaluation of signature files as set access facilities in OODBs. In: Proceedings of the 1993 ACM SIGMOD international conference on management of data, Washington, DC, May 1993, pp 247–256
18. Jain R, Hampapur A (1994) Metadata in video databases. SIGMOD Rec 23(4):27–33
19. Kemper A, Moerkotte G (1992) Access support relations: an indexing method for object bases. Inform Sys 17(2):117–146
20. Kitagawa H, Fukushima K (1996) Composite bit-sliced signature file: An efficient access method for set-valued object retrieval. In: Proceedings of the international symposium on cooperative database systems for advanced applications (CODAS), Kyoto, Japan, December 1996, pp 388–395
21. Kitagawa H, Fukushima Y, Ishikawa Y, Ohbo N (1993) Estimation of false drops in set-valued object retrieval with signature files. In: Proceedings of the 4th international conference on foundations of data organization and algorithms, Chicago, October 1993, pp 146–163
22. Knuth DE (1973) The art of computer programming, vol. 3: Sorting and searching. Addison-Wesley, Reading, MA
23. Maier D, Stein J (1986) Indexing in an object-oriented database. In: Proceedings of the IEEE workshop on object-oriented DBMSs, Asilomar, CA, September 1986
24. Moffat A, Zobel J (1996) Self-indexing inverted files for fast text retrieval. ACM Trans Inform Sys 14(4):349–379
25. Poosala V (1995) Zipf's law. Technical report, University of Wisconsin at Madison
26. Sacks-Davis R, Zobel J (1997) Text databases. In: Indexing techniques for advanced database systems. Kluwer, Amsterdam, pp 151–184
27. Stonebraker M, Moore D (1996) Object-relational DBMSs: the next great wave. Morgan Kaufmann, San Francisco
28. Vance B, Maier D (1996) Rapid bushy join-order optimization with cartesian products. In: Proceedings of the ACM SIGMOD international conference on management of data, Montréal, June 1996, pp 35–46
29. Westmann T, Kossmann D, Helmer S, Moerkotte G (2000) The implementation and performance of compressed databases. SIGMOD Rec 29(3):55–67
30. Will M, Fachinger W, Richert JR (1996) Fully automated structure elucidation – a spectroscopist's dream comes true. J Chem Inf Comput Sci 36:221–227
31. Witten IH, Moffat A, Bell TC (1999) Managing gigabytes. Morgan Kaufmann, San Francisco
32. Xie Z, Han J (1994) Join index hierarchies for supporting efficient navigation in object-oriented databases. In: Proceedings international conference on very large data bases (VLDB), Santiago, September 1994, pp 522–533
33. Zezula P, Rabitti F, Tiberio P (1991) Dynamic partitioning of signature files. ACM Trans Inform Sys 9(4):336–369
34. Zobel J, Moffat A, Ramamohanarao K (1996) Guidelines for presentation and comparison of indexing techniques. ACM SIGMOD Rec 25(3):10–15
35. Zobel J, Moffat A, Ramamohanarao K (1998) Inverted files versus signature files for text indexing. Trans Database Sys 23(4):453–490