Reihe Informatik
2 / 1997

Index Structures for Databases Containing Data Items with Set-valued Attributes

Sven Helmer

Index Structures for Databases Containing Data Items with Set-valued Attributes

Sven Helmer

September 29, 1999

Abstract

We introduce two new hash-based index structures to index set-valued attributes. Both are able to support subset and superset queries. Analytical cost models for the new index structures as well as for the two existing index structures, sequential signature file and Russian Doll Tree, are presented and experimentally validated. Using the validated cost model, we express the performance of all four index structures in terms of the performance of the sequential signature file. This allows a direct analytical comparison of their performance. Last, we report on our benchmark results comparing the real performance of all four index structures. We especially investigate their performance for skewed data.

1 Introduction

Since the invention of database management systems, tremendous effort has been undertaken in order to invent index structures. Historically this work was mostly carried out in the context of relational databases. The impressive results comprise very versatile index structures like B-trees [BM72, Com79]. These traditional index structures concentrate on the task of indexing single-valued attributes.

Modern database systems support data models which allow set-valued attributes. Examples of such data models are the object-oriented model [Cat97] and the object-relational data model [SM96]. Several indexes dealing with special problems in these data models have been invented, e.g. nested indexes [BK89], path indexes [BK89], multi indexes [MS86], access support relations [KM92], join index hierarchies [XH94]. The predominant problem attacked by these index structures is the efficient evaluation of path expressions.

However, the problem of indexing data items with set-valued attributes is of no less importance. Consider for example the following queries: retrieve all students attending at least a given set of lectures, retrieve all researchers attending only a given set of database conferences. Besides user queries involving a set comparison between a constant set and set-valued attributes, this kind of query is also generated by transforming universal quantifiers into set comparisons.

1.1 Related work

Signatures have traditionally been used in information retrieval. There, signature-based indexes accelerate the evaluation of partial match retrieval. Partial match queries are similar in spirit to our subset queries. Let us briefly review indexes for partial match queries in the context of document retrieval. The sequential organization of signatures resulted in signature files in many variants [Rob79], [PBC80], [SD85]. Two level signature files [SDR83] were introduced to enhance their performance. Generalizing this idea results in multi-level signature files [CS89] and one of its important variants, the S-tree [Dep86]. The S-tree is based on the R-tree [Gut84].

Besides tree-based index structures for partial match retrieval hash-based index structures exist. Otoo proposed a hash index for partial match retrieval in [Oto84]. However, signatures are not yet used. Later, the partitioned signature file combined hash indexes with superimposed coding for partial match retrieval of text documents [LL89]. Quickfilter [RZ90, ZRT91, CZ93] is a variant of the partitioned signature file using linear hashing [Lit80]. The algorithms applied to step through all possible candidate sets (semi-consecutive page retrieval in [RZ90], ordered page retrieval in [ZRT91], in [CZ93] no algorithm is given) are ad-hoc and suboptimal. All the index structures mentioned so far are designed for partial match retrieval and are not able to answer both subset and superset queries.

There are only two indexes designed to index set-valued attributes such that subset and/or superset queries can be answered. The first one is based on signature files and comes in the variants sequential and bit-sliced signature file [IKO93]. The second one is the RD-tree [HP94]. It exhibits the same structure as the S-tree [Dep86] and does not support superset queries. Since bit-sliced signature files are inappropriate for dynamic environments, we included only sequential signature files and RD-trees in our investigations.

1.2 Our contribution

We study two new hash-based index structures for set-valued attributes, thereby increasing the number of alternatives to four. The first is based on extendible hashing [FNPS79], the second on recursive linear hashing [RSD84]. Our new index structures have two advantages over the Russian doll tree: Contrary to the Russian doll tree both support the $\supseteq$ predicate. The new index structures are able to outperform the sequential signature file and the Russian doll tree. Further contributions of the paper are 1) validation of analytical cost models for all these index structures, 2) an analytical evaluation based on the cost models, 3) a comparison of the index structures based on benchmarks for uniform distribution and in the presence of skew using the Zipf-distribution. Further, to the knowledge of the authors the paper contains the first performance evaluation of extendible hashing versus recursive linear hashing. The important—and maybe surprising—result will be that our variant of extendible hashing will prove to be much more robust against skew than our variant of recursive linear hashing.

1.3 Outline

The remainder of the paper is organized as follows. In the next section we deal with the preliminaries (e.g. underlying database, query types) needed to explain the index structures. We present the different index structures in the following section. The already known index structures (sequential signature file and Russian doll tree) are described briefly, the newly proposed structures (extendible signature hashing and recursive linear signature hashing) in more detail. Section 4 covers the theoretical comparisons of the index structures using the cost models presented in the section before. In section 5 we specify the simulation environment in which the benchmarking takes place. The results of the benchmarks are presented in the following section. In section 7 we check the validity of the cost models. A summary and outlook conclude this paper.

2 Preliminaries

In this section we demonstrate the relevance of set-valued queries in databases. We give a general description of a database containing data items with set-valued attributes and illustrate it with an example database. We look at different possible query types and also give exemplary queries for each type.

We explain the technique of superimposed coding that is used for representing the sets in the index structures. A fast algorithm for generating all subsets (or supersets) of a given set, which is needed for the query evaluation, is also presented.

2.1 Databases

We assume a database consists of a large number of data items o_i, with $1 \leq i \leq m$. We are interested in those data items which have a set-valued attribute A. $o_i.A$ denotes the value of attribute A of data item o_i. Without loss of generality we assume that all n data items have a set-valued attribute A. The set-valued attribute A is associated with a domain, denoted by $D(A)$, from which the elements of a set are taken. For an example of a database using set-valued attributes one might imagine a database containing students, lectures, and lecturers. Students take courses on different subjects. The information on taken courses can be stored in a set-valued attribute for each student. The same holds for all lectures given by one lecturer. Generally speaking, all one-to-many (or many-to-many) relations between entities can be represented by a set (or two sets). Example 2.1 introduces the lecture database formally.

Example 2.1 *Let us assume a database containing information on lectures. We present a structural model using OMT and give a schema in ODL. Figure 1 shows the structural model.*

Figure 2 depicts the ODL schema of the database.

2.2 Queries

A query is defined by a query set Q and a predicate θ. Q contains arbitrary elements from the domain $D(A)$ of the attribute A. Formally speaking Q is selected among the power set $\mathcal{P}(D(A))$ of the $D(A)$, so $Q \in \mathcal{P}(D(A))$. Given two sets M and N a *predicate*

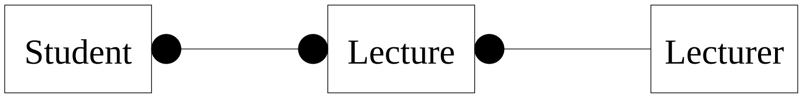

Figure 1: Database example modeled in OMT

```
interface     Student {
extent        allStudents;
attribute     String Name;
relationship  set < Lecture > attends inverse Lecture::attendees;
};

interface     Lecturer {
extent        allLecturers;
attribute     String Name;
relationship  set < Lecture > gives inverse Lecture::lec;
};

interface     Lecture {
extent        allLectures;
attribute     String Name;
relationship  set < Student > attendees inverse Student::attends;
relationship  Lecturer lec inverse Lecturer::gives; };
```

Figure 2: Schema in ODL

θ describing the relationship between M and n is defined by $M\theta N$. We focus on the following predicates:

θ = "$=$": $M = N$ is true, iff M contains exactly the same elements as N. This is called an *equality predicate.*

θ = "$\subseteq$": $M \subseteq N$ is true, iff all elements in M are also found in N. This is called a *subset predicate.*

θ = "$\supseteq$": $M \supseteq N$ is true, iff all elements in N are also found in M. This is called a *superset predicate.*

θ = "$\cap$": $M \cap N (\neq \emptyset)$ is true, iff there exists at least one element from $D(A)$ which is contained in both M and N. This is called an *intersection predicate.*

A complete query, consisting of the query set Q and the predicate $\theta \in \{=, \subseteq, \supseteq, \cap\}$, searches for all data items o_i ($1 \leq i \leq n$) in the database which satisfy the following predicates:

1. $Q = o_i.A$

2. $Q \subseteq o_i.A$

3. $Q \supseteq o_i.A$

4. $Q \cap o_i.A (\neq \emptyset)$

Queries asking for all data items o_i, whose attribute A contains a specific element e from $D(A)$, are a special case of 2. and can be rewritten as a query with $Q = \{e\}$ and $\theta = "\subseteq"$, i.e. $e \in o_i.A \equiv \{e\} \subseteq o_i.A$.

Example 2.2 *Let us formulate some example queries based on the lecture database in example 2.1. We give one example for each query type defined above.*

1. *We want to know, if there is a professor who gives the lecture "Database Systems I", the lecture "Operating Systems I", and no more lectures. ($Q =\{$"Database Systems I", "Operating Systems I"$\}$ and $\theta = "="$)*

   select l.Name
   from l in allLecturers
   where l.gives = set("Database Systems I", "Operating Systems I");

2. *We want to check, which students have met the requirements ("Math I", "Math II") to attend the lecture "Math III". ($Q =\{$"Math I", "Math II"$\}$ and $\theta = "\subseteq$)*

   select s.Name
   from s in allStudents
   where set("Math I", Math II") <= s.attends;

3. *Let us assume we are interested in all those students who have taken only mandatory lectures among "Computer Science I", "Math I", "Programming I". ($Q =\{$"Computer Science I", "Math I", "Programming I"$\}$ and $\theta = "\supseteq$)*

   select s.Name
   from s in allStudents
   where s.attends <= ("Computer Science I", "Math I", "Programming I");

4. *Get the names of all students who go to at least one of the following lectures: "Database Systems I", "Operating Systems I", "Math II". ($Q =\{$"Database Systems I", "Operating Systems I", "Math II"$\}$ and $\theta = "\cap$)*

   select s.Name
   from s in allStudents
   where s.attends intersect set("Database Systems I", "Operating Systems I", "Math II") $\neq$ set();

2.3 Superimposed coding

Superimposed coding is a technique based on the idea to hash attribute values into random k-bit codes in a b-bit field and to superimpose the codes for each attribute value in a record [Knu73]. A code word created by superimposing bit fields is called a *signature* [FC84]. We use signatures to represent sets in the index structures. There are two reasons why we employ signatures. One reason is the constant length of the signatures. Keys of constant length are easier to manage in index structures than keys of variable length. The other reason is the great speed with which signatures can be compared by using only bit operations. In this section we explain how to encode sets as signatures.

2.3.1 Basic principles

A signature is a bit field of a fixed length b called the *signature length*. Signatures are used to represent or approximate sets. The signature of a set is generated by hashing all the elements of the set into binary code words of length b. For each element a binary code word, in which exactly k bits are set, is generated. Afterwards all binary code words are superimposed by using a *bitwise or* operation creating the final signature (see figure 3 for the algorithm). For a set s, let us denote the signature of s by $sig(s)$.

```
generateSig(set s)
{
    sig = 0;
    for(all items s_i in s)
    {
        tmpSig = 0;
        i = 0;
        srandom(s_i); /* set seed in random number generator */
        while(i < k)
        {
            rnd = random() % b;
            while(rnd-th bit is set in tmpSig)
            {
                rnd = random() % b;
            }
            set rnd-th bit in tmpSig;
            i++;
        }
        sig |= tmpSig;
    }
    return sig;
}
```

Figure 3: Algorithm for generating signatures

We cannot assume that the signatures of distinct sets are also distinct, due to the hashing and the bitwise or-operation. But still, the following property holds:

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

where $\mathrm{sig}(s)\theta\mathrm{sig}(t)$ and $|\mathrm{sig}(s)|$ are defined as

$$\begin{aligned}
\mathrm{sig}(s) \subseteq \mathrm{sig}(t) &:= \mathrm{sig}(s) \& \neg\mathrm{sig}(t) = 0 \\
\mathrm{sig}(s) \supseteq \mathrm{sig}(t) &:= \mathrm{sig}(t) \& \neg\mathrm{sig}(s) = 0 \\
\mathrm{sig}(s) \cap \mathrm{sig}(t) &:= \mathrm{sig}(s) \& \mathrm{sig}(t) \neq 0 \\
|\mathrm{sig}(s)| &:= \text{number of bits set in } s \text{, also called the } \textit{weight} \text{ of } \mathrm{sig}(s)
\end{aligned}$$

with & denoting *bitwise and* and ¬ denoting *bitwise complement.*

Hence, a *pretest* based on signatures can be very fast since it involves only bit operations.

Example 2.3 *Let us illustrate the technique of superimposed coding with an example taken from our lecture database. We encode the query sets of the queries 2. and 3. of the previous example.*

$$\begin{aligned}
Q_2 &= \{\text{"Math I"}, \text{"Math II"}\} \\
Q_3 &= \{\text{"Computer Science I"}, \text{"Math I"}, \text{"Programming I"}\}
\end{aligned}$$

We use a signature length B of 8 bits and set exactly 2 bits in the binary code words of each element, so $k = 2$. We call our hash function to map the elements onto binary code words $H(x)$ with $x \in D(A)$. Without loss of generality let us assume that $H(x)$ maps the elements of Q_2 and Q_3 onto the following binary code words.

$$\begin{aligned}
H(\text{"Math I"}) &= 1001\ 0000 \\
H(\text{"Math II"}) &= 0100\ 0100 \\
H(\text{"Computer Science I"}) &= 0100\ 1000 \\
H(\text{"Programming I"}) &= 0000\ 1100
\end{aligned}$$

Superimposing the code words via an (inclusive) bitwise or-*operation, we get the following signatures:*

$$\begin{aligned}
sig(Q_2) &= 1101\ 0100 \\
sig(Q_3) &= 1101\ 1100
\end{aligned}$$

As can be clearly seen by this example, if signatures satisfy a certain predicate this does not imply that the sets themselves satisfy this predicate, because $sig(Q_2) \subseteq sig(Q_3)$, but $Q_2 \not\subseteq Q_3$. We go into the details on this matter in the next section about false drop probabilities. ◇

2.3.2 False drop probabilities of signatures

Consider a query set Q, the attribute value $o_i.A$ of a data item o_i, their signatures $sig(Q)$ and $sig(o_i.A)$, and a predicate $\theta \in \{=, \subseteq, \supseteq, \cap\}$. If $sig(Q)\theta sig(o_i.A)$ is true, then o_i is called a *drop*. If additionally $Q\theta o_i.A$ holds, we call o_i a *right drop*. If, however, $Q\theta o_i.A$ does not hold, o_i is a so called *false drop*.

The probability that a data item turns out to be a false drop—called *false drop probability* d_θ—has been studied intensively [FC84, IKO93, KFIO93, Rob79, SDR83] and can be approximated by formulas (3) to (6). Here, $|Q|$ denotes the size of the query set, i.e. the number of elements in Q. $|o_i.A|$ is the size of the attribute value A of data item o_i. The weight of a signature of a set M can be estimated, when the size of M is known:

$$|sig(M)| \approx b \cdot \left(1 - \left(1 - \frac{k}{b}\right)^{|M|}\right) \tag{2}$$

We now present the false drop probabilities for each predicate type as described by [KFIO93].

$$\begin{aligned}
d_=(b,k,|Q|,|o_i.A|) &= \frac{1}{\binom{b}{|\mathrm{sig}(Q)|}} \quad \text{for } |\mathrm{sig}(Q)| = |\mathrm{sig}(o_i.A)| && (3)\\
d_\subseteq(b,k,|Q|,|o_i.A|) &\approx (1 - e^{-\frac{k}{b}|o_i.A|})^{k\cdot|Q|} && (4)\\
d_\supseteq(b,k,|Q|,|o_i.A|) &\approx (1 - e^{-\frac{k}{b}|Q|})^{k\cdot|o_i.A|} && (5)\\
d_\cap(b,k,|Q|,|o_i.A|) &\approx 1 - \sum_{j=0}^{k-1} \binom{|\mathrm{sig}(Q)|}{j} \cdot (1-(1-\frac{k}{b})^{|o_i.A|})^j \cdot (1-\frac{k}{b})^{|o_i.A|\cdot(|\mathrm{sig}(Q)|-j)} && (6)
\end{aligned}$$

Each query produces a certain number of drops c_d. Among these drops are c_r right drops, which are the answer to the query, and c_f false drops with

$$c_f = (n - c_r) \cdot d_\theta \tag{7}$$

We cannot distinguish between right drops and false drops by only comparing signatures. So all c_d data items which are drops have to be fetched from secondary storage and checked for false drops. We strive for a low false drop probability to keep c_f and with it the number of accesses to secondary storage small. There is an optimal ratio between k and b for which d_θ becomes minimal.

For equality predicates $d_=$ becomes minimal, when

$$\frac{k}{b} = \left(1 - \left(\frac{1}{2}\right)^{|Q|}\right) \tag{8}$$

For subset predicates $d_\subseteq$ becomes minimal, when

$$\frac{k}{b} = \frac{\ln 2}{|o_i.A|} \tag{9}$$

For superset predicates $d_{\supseteq}$ becomes minimal, when

$$\frac{k}{b} = \frac{\ln 2}{|Q|} \tag{10}$$

For $d_{\cap}$ for intersection predicates no closed formula exists to calculate the optimal ratio between k and b. Fortunately (6) only contains discrete values and for a fixed b an optimal value for k can be calculated fast with a brute force algorithm, computing $d_{\cap}$ for all possible values for k.

2.4 Fast enumeration of subsets/supersets

For our new index structures presented in the next section, we need a way to rapidly step through all subsets and supersets of a given signature. We use the same algorithm utilized by Vance and Maier in their blitzsplit join ordering algorithm [VM96]. The algorithm to generate all subsets of a given bitstring a is given below (& denotes *bitwise and* and ˜ denotes *bitwise complement*). When executed, s passes through all possible subsets of a.

```
s = a & -a;
while(s)
{
    s = a & (s - a);
    process(s);
}
```

Generating all supersets is achieved by inverting the signature a, stepping through the subsets of the inverted a and inverting the generated sets.

```
s = ˜a & - ˜a;
while(s)
{
    s = ˜a & (s - ˜a);
    process(˜s);
}
```

3 Index structures

We present four different indexing schemes suitable for indexing data items with set-valued attributes. In all four index structures we store the signatures $sig(o_i)$ and references $ref(o_i)$ of the data items. The general concept behind these index structures is to provide fast access to all signatures satisfying a query. The corresponding data items are then checked and all false drops are eliminated.

We describe each index structure generally and also give details about storage costs and retrieval/update costs. We begin with the Sequential signature file and Russian Doll tree, which are already known [LL92], [IKO93], [HP94]. The two new index structures are based on extendible hashing [FNPS79] and recursive linear hashing [RSD84], which we describe briefly. We then explain how to adapt those index structures to support queries with set predicates.

When looking at the costs for the index structures, there are operations that are independent of the index structures. For the query evaluation costs these are the costs C_{fetch} for fetching the data items. The fetch costs, which are the same for all index structures, can be approximated by the formula by Yao [Yao77] (B is the number of pages needed for the data items). C_{fetch} is measured in number of page accesses.

$$C_{fetch} = n \cdot \left(1 - \prod_{i=1}^{c_r + c_f} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1}\right) \tag{11}$$

When deleting items, the deletion of the data item itself has nothing to do with the index structure. The deletion costs C_{delete} for a data item depend on the used database management system. We assumed one writing page access for a deletion for our theoretical calculations.

$$C_{delete} = 1 \tag{12}$$

For the calculation of the deletion costs we assume that we always have successful deletions, i.e. the data item to be deleted is present in the database.

3.1 Sequential signature files

3.1.1 General description

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

Figure 4: Sequential signature file index structure (SSF)

The sequential signature file (SSF) method is a simple way to index data items using signatures [LL92], [IKO93]. Signature files have entries in the form of a tuple $[sig(o_i), ref(o_i)]$ containing the signature $sig(o_i)$ of an indexed data item o_i and an unique identifier $ref(o_i)$ with which the data item can be accessed. The signatures and identifiers of all data items o_i, $1 \leq i \leq n$ are stored sequentially in a file. Instead of scanning all data items during query evaluation, the signature file is scanned and only data items, the signature of which satisfies the query condition, are fetched and tested for false drops.

A sequential signature file index supports all query types discussed in the previous section.

3.1.2 Cost formulas

As mentioned in the last section, a signature $sig(o_i)$ of a data item o_i and its unique identifier $ref(o_i)$ are stored pairwise in tuples of the form $[sig(o_i), ref(o_i)]$. The size of such a tuple (in bytes) is denoted with S_{tuple}. The storage costs S_{SSF} (in number of pages) of a sequential signature file are equal to

$$S_{SSF} = \left\lceil \frac{n \cdot S_{tuple}}{P} \right\rceil \text{pages} = \left\lceil \frac{n}{P} \cdot \left\lceil \frac{(b + id)}{y} \right\rceil \right\rceil \text{pages} \tag{13}$$

where P is the number of bytes per page, y the number of bits per byte (which is 8, of course), and id the size of an unique identifier in bits. The value of b, the number of bits in a signature, depends on the desired false signature drop probability (see 2.3.2)

Evaluating queries of the kind described in section 2.2 with the help of a SSF is straightforward. First we construct the signature S_Q for the query set Q. Then we traverse the signature file and compare S_Q with all signatures $sig(o_i)$, $1 \leq i \leq n$, in the signature file. If the signature $sig(o_i)$ matches the query set S_Q, then we have to fetch the corresponding data item via the reference $ref(o_i)$. After the quick pretest using signatures, we have to check whether the set-valued attribute of o_i satisfies the query or not, remembering the false drops mentioned in section 2.3.2.

The retrieval costs C_θ^{SSF} for an access (measured in page accesses) are described by

$$C_\theta^{SSF} = S_{SSF} + C_{fetch} \text{ for } \theta \in \{=, \subseteq, \supseteq, \cap\} \tag{14}$$

C_{fetch} is the number of page accesses needed to access the data items, which turn out to be drop, i.e. the signatures of which satisfy the query.

Insertions into a SSF index structure are very easy to realize. The signature $sig(o_{n+1})$ for the new data item o_{n+1} is generated and stored with the identifier $ref(o_{n+1})$ in a tuple which is appended to the signature file. The cost for an insertion I_{SSF} is normally one reading and one writing page access. An exception to this is, when the last page of the signature file is full, then a new page must be appended and the catalog information of the SSF must be updated.

$$I_{SSF} = 2 \tag{15}$$

When deleting a data item, the reference to the data item is marked as deleted or set to NULL, respectively. The deletion of the object and the adjustment of the index take place during times of low user activity. The costs D_{SSF} for marking an object as deleted are the searching costs and one (writing) access for the marking. C_{delete} are the costs for deleting the data item itself.

$$\overline{D}_{SSF} = \frac{1}{2} \cdot S_{SSF} + 1 + C_{delete} \tag{16}$$

As already mentioned we assume successful deletions, so on average we have to traverse about half of the signature file before finding the desired data item.

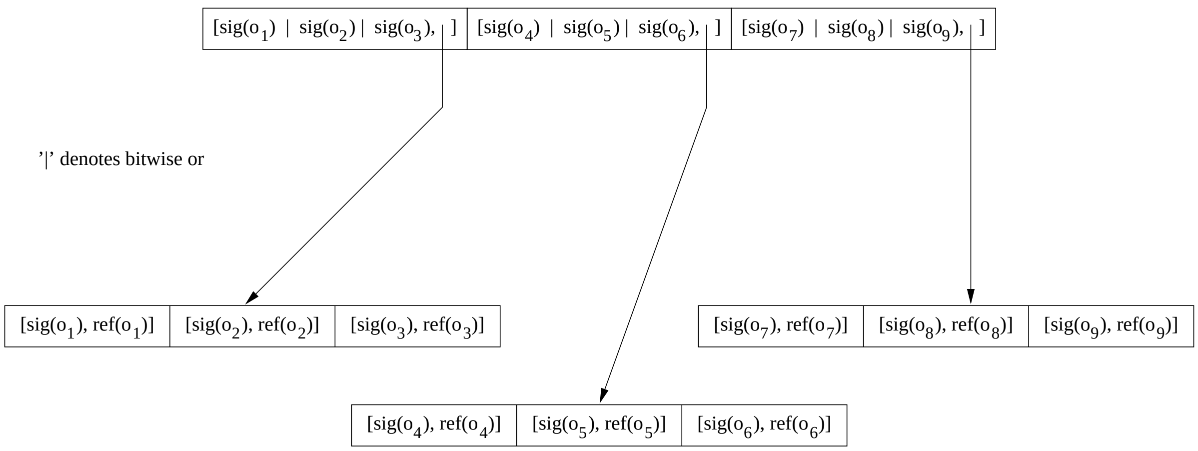

Figure 5: RD-tree using signatures as keys

3.2 Russian Doll trees

3.2.1 General description

The Russian Doll (RD) tree [HP94] is a variant of the R-tree [Gut84]. The leaf nodes of an RD-tree contain as entries the sets of the data-items and the identifiers of the data items. An entry in an inner node of an RD-tree consists of the union of all sets of its child nodes, called *bounding set*, and the references to the child nodes. In our implementation (see figure 5) we use the signatures of the sets as keys in the RD-tree, making it similar to the S-Tree proposed by Deppisch [Dep86] for office retrieval.

When evaluating a query, we begin by constructing the signature $sig(Q)$ of the query set Q. With the signature $sig(Q)$ we start to search at the root of the tree. In each inner node we compare $sig(Q)$ with the signatures $sig(B_{ji})$ of the bounding sets B_{ji} in the node m_j. For a query with a subset or equality predicate this means, that B_{ji} qualifies, if $sig(Q) \subseteq sig(B_{ji}$. For an intersection predicate $sig(Q) \cap sig(B_{ji}) \neq \emptyset$ has to be satisfied. All branches that do not qualify need not be searched, because none of the descendants can satisfy the query. In a leaf node all data items, whose signatures match the signature of the query set are fetched and checked.

An RD-tree supports queries with equality, subset, and intersection predicates. Although it might appear that a variant in which the signatures within the inter nal nodes are formed by intersection supports superset queries, this variant exhibits very poor performance [HP94]. This is due to the fact that after a few intersection operations no bit will be set anymore and consequently all branches of the variant have to be searched during retrieval.

3.2.2 Cost formulas

We assume that the size of a node in an RD-tree is equal to the size of a page. Under the assumption that each node in the RD-tree is utilized to degree α, i.e. $\alpha \cdot P$ bytes of each page are filled with data, the storage costs for an RD-tree can be estimated similar to the storage costs of an R-tree [Gut84] with a branching factor $bf = \alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor$. Hence, the total storage size S_{RDT} is equal to

$$\begin{aligned}
S_{RDT} &= \sum_{i=1}^{\lceil \log_{bf}(n) \rceil} \left\lceil \frac{n}{\left(\alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor\right)^i} \right\rceil && (17)\\
&= \left\lceil \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor} \right\rceil + \left\lceil \frac{n}{\left(\alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor\right)^2} \right\rceil + \ldots + 1 \text{ pages} && (18)\\
&\approx \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor - 1} \text{ pages} && (19)
\end{aligned}$$

An upper bound for the expected number of page accesses can be estimated by looking at each node separately. The probability that a node m_j is accessed (independently of the other nodes) is

$$\Pr(sig(Q) \text{ retrieves } m_j \text{ in a subset, equality query}) = \Pr(sig(Q) \subseteq sig(B_j)) \qquad (20)$$

for queries with subset and equality predicates. $sig(B_j)$ is the signature of the bounding set for m_j. For queries with intersection predicates, the probability is given by

$$\Pr(sig(Q) \text{ retrieves } m_j \text{ in an intersection query}) = \Pr(sig(Q) \cap sig(B_j) \neq \emptyset) \qquad (21)$$

Now the expected number of page accesses $C^{RD}_{\subseteq,=}$ for queries with subset and equality predicates can be estimated by

$$C^{RD}_{\subseteq,=} \leq 1 + \sum_{j=2}^{m} \Pr(sig(Q) \subseteq sig(B_j)) + C_{fetch} \qquad (22)$$

The root page always has to be accessed. The other nodes are accessed with different probabilities. When the bounding set of the parent node does not satisfy the query predicate, then the child node will not be accessed. Therefore (22) is an upper bound for the expected number of page accesses. For intersection predicates the following holds

$$C^{RD}_{\cap} \leq 1 + \sum_{j=2}^{m} \Pr(sig(Q) \cap sig(B_j) \neq \emptyset) + C_{fetch} \qquad (23)$$

When inserting a new data item o_{n+1}, we start at the root. For each inner node the signature $sig(o_{n+1})$ of the new data item is compared to the signature $sig(B_{ji})$ of every bounding set in node m_j. We follow the branch for the smallest value of $\text{diff}(sig(B_{ji}), sig(o_{n+1}))$ with

$$\text{diff}(sig(B_{ji}), sig(o_{n+1})) = (sig(B_{ji}) \vee sig(o_{n+1})) \oplus sig(B_{ji}) \qquad (24)$$

$\vee$ denotes the *bitwise or* operation, $\oplus$ the *bitwise xor* operation.

When we reach a leaf node m_l, the tuple $[sig(o_{n+1}), ref(o_{n+1})]$ is inserted. The signatures of the bounding sets in the parent nodes of m_l are modified (up to the root). The number of page accesses for an insertion I_{RD} can be estimated by

$$I_{RD} \leq 2 \cdot H(T) \tag{25}$$

where $H(T)$ is the height of the tree RD-tree T. When an overflow occurs during an insertion (i.e. the leaf node m_l is full), the leaf node m_l needs to be split. For the split of a node, we used the following (linear-cost) split algorithm.

```
split()
{
    find signature S_i with greatest weight;
    assign S_i to group 1;
    S_1 = signature of bounding set of group 1;

    find signature S_i with greatest diff(S_1, S_i);
    assign S_i to group 2;
    S_2 = signature of bounding set of group 2;

    for(remaining signatures S_i)
    {
        if(diff(S_1, S_i) ≥ diff(S_2, S_i))
        {
            assign S_i to group 2;
            S_2 = S_2 ∨ S_i
        }
        else
        {
            assign S_i to group 1;
            S_1 = S_1 ∨ S_i
        }
    }
}
```

After splitting the node, the signatures in the parent nodes of m_l need to be adjusted. During the modification of the signatures, it may be necessary to continue splitting nodes all the way up to the root.

The first step when deleting a data item is to search for it in the tree. When we find it in a leaf node, we delete the corresponding tuple and readjust the signatures in all parent nodes. For the costs D_{RD} for a deletion this means (note that we do not restructure the tree to guarantee a minimum number of entries in each node)

$$D_{RD} \leq C_{=}^{RD} + H(T) + C_{delete} \tag{26}$$

3.3 Extendible signature hashing

3.3.1 General description

Extendible hashing [FNPS79] is one of the several varieties of dynamic hashing, where the size of the hash table varies during its lifetime. In extendible hashing a hash function $h(x)$ is chosen, that maps each element x into a domain of hash keys that is larger than the range of the current hash table. The value of $h(x)$ is represented as a binary word. Let $h_d(x)$ be a prefix of $h(x)$ consisting of the first d bits of $h(x)$.

An extendible hashing index is divided into two parts, the directory and the buckets. A bucket contains tuples consisting of the hash key of the data item and an identifier with which to reach the data item. The directory begins with a header that holds the value for the (global) depth d. The directory has 2^d entries which are references to the buckets. When looking up a data item in the directory, $h_d(x)$ is determined to find the reference to the bucket where the item is to be found. The entries in the directory are not necessarily distinct, so there may be more than one reference in the directory pointing to the same bucket. The local depth d' of a bucket specifies the length of the prefix actually used in this bucket, i.e. only the first d' bits of the hashkeys of all entries in this bucket must be equal.

When an overflow occurs in a bucket, this bucket needs to be split. The elements of the overflown bucket are divided into two buckets. d' has to be enlarged by one bit in order to distinguish between these two buckets. If the local depth of a bucket is larger than the global depth after a split has occurred, then the directory of the hash table has to be doubled. The references in the directory to the buckets, that are not split, are copied.

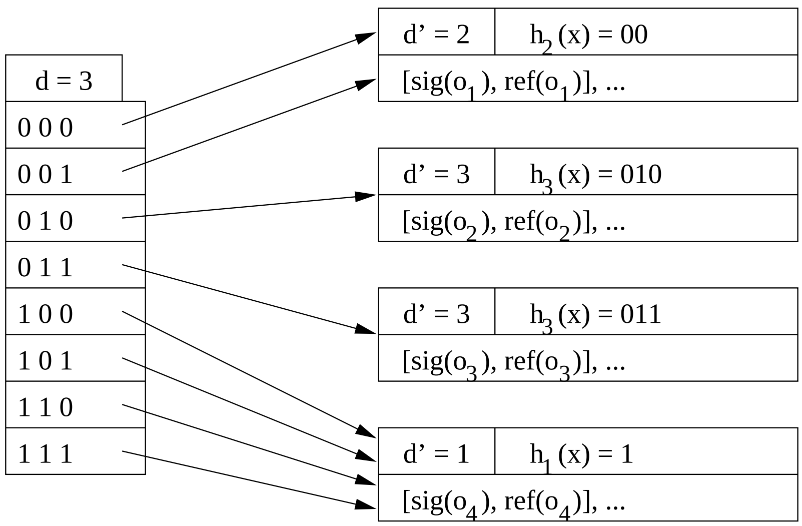

Figure 6: Extendible signature hashing

We modify extendible hashing by using the signatures of the data items as hashkeys. $sig_d(o_i)$ denotes the first d bits of the signature $sig(o_i)$.

$$
\begin{aligned}
h(o_i) &= sig(o_i) &(27)\\
h_d(o_i) &= sig_d(o_i) &(28)
\end{aligned}
$$

Further, since we do not assume that the attribute's sets are unique, we must provide overflow buckets in case a page cannot be split since all objects have the same attribute value or the same signature. An overflow bucket is only used if all the signatures in the bucket are equal an d, hence, further splitting is not possible. Overflow buckets are chained.

This modified version of extendible hashing will be called extendible signature hashing. Extendible signature hashing has two advantages. It is possible to map sets onto bitstrings that can be used as hashkeys. But more importantly, we are able to support queries with subset and superset predicates, whereas ordinary extendible hashing only supports queries with equality predicates. The next paragraph shows, how subset and superset predicates are handled.

Obviously,

$$
\begin{aligned}
s \subseteq t &\implies \mathrm{sig}(s) \subseteq \mathrm{sig}(t) \implies \mathrm{sig}_d(s) \subseteq \mathrm{sig}_d(t) &(29)\\
s \supseteq t &\implies \mathrm{sig}(s) \supseteq \mathrm{sig}(t) \implies \mathrm{sig}_d(s) \supseteq \mathrm{sig}_d(t) &(30)
\end{aligned}
$$

So in order to find all sets which are subset (supersets) of the query set, we have to find all data items whose signature has a prefix which is a subset (superset) of the prefix of the query signature. We are able to generate all subsets (supersets) of the prefix of the query signature very quickly by using the technique of Vance and Maier [VM96] (see section 2.4). All buckets which possibly contain qualifying signatures are searched. After a quick pretest of the signatures the corresponding data items are fetched and checked.

For (29) and (30) we are able to argue, that if a prefix of a signature s is no subset (superset) of a prefix of a signature t, then there is no possibility whatsoever that s is a subset (superset) of t. Unfortunately, for queries with intersection predicates

$$
|\mathrm{prefix}(\mathrm{sig}(s)) \cap \mathrm{prefix}(\mathrm{sig}(t))| \geq k \implies |\mathrm{sig}(s) \cap \mathrm{sig}(t)| \geq k \qquad (31)
$$

holds. The fact that a prefix of signature s does not intersect with a prefix of signature t does not implicate that s does not intersect t. That means, we will not be able to support queries containing intersection predicates with an extendible signature hashing index.

We take care not to access a multiply referenced bucket more than once. We applied a sort-based duplication elimination algorithm to the retrieved set of bucket identifiers.

3.3.2 Cost formulas

The size of an extendible signature hashing index S_{ESH} in pages depends on the size of the directory S_{DIR} and the sum of the size S_{BUCK} needed by the m buckets. We assume that a bucket has exactly the size P of a page. (id is the size of a reference to a bucket in the directory.)

$$
\begin{aligned}
S_{ESH} &= S_{DIR} + S_{BUCK} && (32)\\
S_{DIR} &= \left\lceil \frac{2^d \cdot id}{P} \right\rceil \text{ pages} && (33)\\
S_{BUCK} &= m \text{ pages} \leq 2^d \text{ pages} && (34)
\end{aligned}
$$

For the application of formula (34) the number of buckets has to be known, otherwise a rough estimate is given. For uniform distribution of the hashing keys, a more accurate formula for the average size of an extendible hashing index is given in [FNPS79]:

$$
\overline{S}_{ESH} \approx \left\lceil \frac{n \cdot S_{tuple}}{P} \right\rceil \cdot \log_2 e \text{ pages} \qquad (35)
$$

When evaluating a query with an equality predicate, there are very few page accesses needed to fetch the searched data item. We need one page access to get the reference to the right bucket from the directory and another page access to load the bucket into main memory. All data items, whose signature satisfies the query, have to be retrieved. There are c_r right drops and c_f false drops. C_{fetch} is the number of page accesses needed to fetch the data items, which are drops.

$$
C_{=}^{ESH} = 2 + C_{fetch} \qquad (36)
$$

When the parameters k and b for the signatures are optimized (see section 2.3.2), the expected weight of a signature is $\frac{b}{2}$. Assuming that the set bits in a signature are uniformly distributed among all b bits, the expected weight of a prefix of length d of this signature is $\frac{d}{2}$. Therefore the expected number of generated subsets or supersets for a query with subsets or superset predicates is equal to $2^{\frac{d}{2}}$. That means we have to look at $2^{\frac{d}{2}}$ entries in the directory. However, we do not always need to access $2^{\frac{d}{2}}$ buckets, because not all entries in the directory are distinct (remember the sharing of buckets). For the number of page accesses for a query evaluation this means

$$
\begin{aligned}
C_{\subseteq,\supseteq}^{ESH} &\approx 2^{\lceil \frac{d}{2} \rceil} \cdot 2 + C_{fetch} && (37)\\
&= 2^{\lceil \frac{d}{2} \rceil + 1} + C_{fetch} && (38)
\end{aligned}
$$

assuming C_{fetch} page accesses are necessary to fetch the data items which are drops.

In order to insert a new data item o_{n+1}, two page accesses to reach the correct bucket and one access to write the tuple $[sig(o_{n+1}), ref(o_{n+1})]$ are needed.

$$
I_{ESH} = 3 \qquad (39)
$$

If an overflow occurs, the bucket has to be split. This increases the local depth of the buckets. If the new local depth is larger then the global depth, the directory has to be doubled. The last step taken is the readjustment of the references in the directories.

Deleting an item is similar to inserting an item. First of all, the corresponding bucket has to be found, then one writing access deletes the data item in the index. C_{delete} is the cost for deleting the item itself.

$$D_{ESH} = 3 + C_{delete} \tag{40}$$

Two neighboring buckets, i.e. buckets which emerged from the same split operation, may be merged, if the sum of their entries falls below a certain threshold. If all local depths fall below the global depth, then the directory can be halved again. However, this overhead is only justified, if no new growth is expected to replace the deletions. We did not implement the merging of buckets and the halving of the directory during deletion.

3.4 Recursive linear signature hashing

3.4.1 General description

Recursive linear hashing is another variant of dynamic hashing. It is typically assumed that in extendible hashing the size of the hash table can grow quite large, because the directory must be doubled every time the global depth is increased. For that reason linear hashing, in which the directory grows linearly with the number of entries, was devised by Litwin [Lit80].

Like an extendible hashing index, a linear hashing index is divided into two parts, the directory and the buckets. At the start of the dth expansion, the directory references 2^d buckets. If there is no space left in a bucket during insertion, the entry is placed into an overflow bucket connected to the original bucket. Whenever L insertions have taken place, the directory is extended by adding an entry at the end of the directory at position $2^d + p$, which references a new bucket. p is the position of the split pointer, which marks the next bucket to be split. During a split operation all entries in the pth bucket are divided among the pth and $2^d + p$th bucket. After the split occurs, p is increased by 1. If p is equal to 2^d, i.e. the end of the current expansion has been reached, then d is increased by 1 and p now points to the first bucket in the hash table again.

Recursive linear hashing is a variant of linear hashing that avoids the use of overflow buckets [RSD84]. All entries which cannot be placed in the appropriate buckets are hashed into a second hash table (using the same hash function). As long as there is no space left in the appropriate bucket on level l, the process of rehashing continues at a lower level until a bucket with enough free space is found. Usually there are only two to three levels of recursive hash tables storing overflow entries. The directory of each recursive hash table has its own depth d_l and its own split pointer p_l. For an example of recursive linear hashing, see figure 7.

We modify recursive linear hashing by using signatures as hashkeys, thus calling the technique recursive linear signature hashing. We are able to support queries with subset and superset predicates, by quickly generating all subsets and supersets (see section 2.4). For the same reasons given in section 3.3 about extendible hashing (see (31)), we are not able to support queries with intersection predicates with a recursive linear signature hashing index.

To search for a data item, the following recursive algorithm is used. We begin to search on the top level ($l = 1$, call search(1, $sig(o_i)$)). If we do not find the data item on

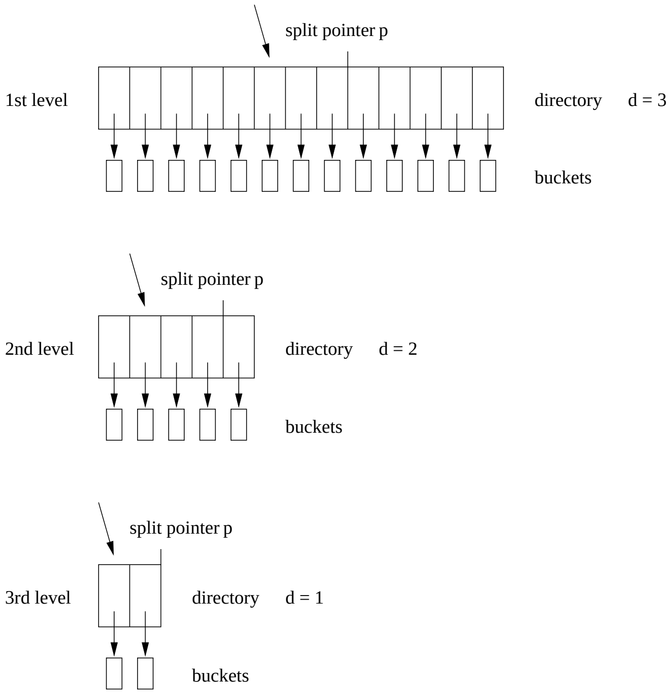

Figure 7: recursive linear hashing

the current level, we continue searching on the next lower level. This continues until the item has been found or we have reached the lowest level.

search(level, $sig(o_i)$)
{
    if($sig_{d_{level}}(o_i) \geq p_{level}$)
    {
        entry_no = $sig_{d_{level}}$;
    }
    else
    {
        entry_no = $sig_{d_{level}+1}$;
    }
    if($sig(o_i)$ found in bucket referenced by dir_{level}(entry_no))
    {
        retrieve data item o_i;
    }
    else
    {
        if(level $\geq$ maxlevel)
        {
            data item not found;
        }
        else
        {

```
            search(level + 1, sig(o_i))
        }
    }
}
```

3.4.2 Cost formulas

Let n_l be the number of references to data items inserted in the hash table on level l. The total size of the recursive linear hashing index can be determined by adding the size of the directories and buckets of all recursive hash tables together.

$$S_{RLSH} = \sum_{l=1}^{maxlevel} (S_{DIR_l} + S_{BUCK_l}) \tag{41}$$

Let u_l be the number of buckets on level l, d_l the depth and p_l the split pointer of the directory on level l. L is the load factor, i.e. after L insertions in a hash table it is expanded by enforcing the splitting of a bucket.

Then

$$u_l = \left\lceil \frac{n_l}{L} \right\rceil \tag{42}$$

$$d_l = \lfloor \log_2(u_l) \rfloor \tag{43}$$

$$p_l = u_l \bmod d_l \tag{44}$$

Of all data items inserted on level l, r_l data items are actually stored in the hash table on level l. The remaining $n_l - r_l$ data items are stored in tables on successive levels. The value of n_l can be calculated recursively by

$$n_{l+1} = n_l - r_l \tag{45}$$

Before inserting a data item into any hash table we try to insert it into the hash table on the first level, so $n_1 = n$. For uniform distribution of the hashkeys Ramamohanarao and Sacks-Davis present a formula in [RSD84] to approximate r_l

$$r_l \approx 2 \cdot p_l \cdot L_l + (2^{d_l} - p_l) \cdot R_l \tag{46}$$

where L_l is the estimated number of data items per bucket on the left side of the split pointer p_i on level l, and R_i is the estimated number of data items per bucket on the right side of the split pointer. For details on how to calculate L_l and R_l see the appendix.

Now we can calculate S_{DIR_l} and S_{BUCK_l}, assuming a bucket fits on a page of size P.

$$S_{DIR_l} = \left\lceil \frac{(2^{d_l} + p_l) \cdot id}{P} \right\rceil \text{ pages} \tag{47}$$

$$S_{BUCK_l} = u_l \text{ pages} \tag{48}$$

The retrieval costs in a recursive linear signature hashing index depend on the number of recursive hash tables that are searched. We look at the evaluation of a query with an equality predicate first.

Ramamohanarao and Sacks-Davis distinguish between two cases in their recursive linear hashing [RSD84], the costs for a successful search, i.e. the data item is found, and the costs for an unsuccessful search, i.e. the data item is not found. When the item is found, the search ends immediately. This can be done, because Ramamohanarao and Sacks-Davis assume data items with unique keys. We cannot abort the search prematurely, when we find a data item, because our keys are not unique. There may be more than one set containing exactly the same elements. This means, we have to search all recursive hash tables.

On each level we need one page access to the directory and one page access to fetch the corresponding bucket. The expected number of page accesses $C_{=}^{RLSH}$ for a query evaluation with an equality predicate is equal to

$$C_{=}^{RLSH} = 2 \cdot maxlevel + C_{fetch} \tag{49}$$

Let us now look at the evaluation costs of a query with subset or superset predicates. During the processing of the query all subsets or superset of the query set are generated and looked up. As we have already determined in section 3.3, the expected number of generated sets is equal to $2^{\frac{d}{2}}$. For the number of page accesses $C_{\subseteq,\supseteq}^{RLSH}$ this means

$$\begin{aligned} C_{\subseteq,\supseteq}^{RLSH} &\approx \left(\sum_{l=1}^{maxlevel} 2^{\lceil \frac{d_l}{2} \rceil} \cdot 2 \right) + C_{fetch} &(50)\\ &= \left(\sum_{l=1}^{maxlevel} 2^{\lceil \frac{d_l}{2} \rceil + 1} \right) + C_{fetch} &(51) \end{aligned}$$

When inserting an item, we have to find the correct bucket in which to insert. If this bucket would overflow, we have to find the correct bucket on the next lower level. This may continue recursively until a bucket, which is not full, is found. We always have to look at the hash table on the top level. Whether a recursive level is checked, depends on the probability that the correct bucket on the next higher level is full. Let $\Pr(bucket_{li} = bucket_{max})$ denote the probability that the number of items $bucket_{li}$ in the i-th bucket on level l is equal to the maximum number of items $bucket_{max}$ a bucket can contain. When a free and correct bucket has been found, one additional writing access for the insertion is needed.

$$I_{RLSH} \leq 2 + \left(\sum_{l=2}^{maxlevel} 2 \cdot (l+1) \cdot \Pr(bucket_{li} = bucket_{max}) \right) + 1 \tag{52}$$

If the $k \cdot L$-th insertion (with $k = 1, 2, 3, \ldots$) takes place in a hash table, the bucket referenced by the split pointer via the $p - th$ entry in the directory must be split. All items in the bucket are divided among the original bucket and the new appended bucket using the following algorithm.

```
for(all items o_i in bucket referenced by p)
{
    if(h_d(o_i) == h_{d+1}(o_i))
    {
        leave [sig(o_i), ref(o_i)] in p-th bucket;
    }
    else
    {
        move [sig(o_i), ref(o_i)] to 2^d + p-th bucket;
    }
}
p++;
if(p == 2^d)
{
    d++;
    p = 0;
}
```

After splitting a bucket all hash tables on lower levels than the current one have to be searched for items which actually belong to the p-th or $2^d + p$-th bucket on the current level. Any items found in this way are reinserted into the index at the current level. The number of levels may decrease after the restructuring of the index.

The first step taken when deleting an item is to find the item. When we find the right bucket, the item has to be removed from the index. When deleting we can abort the search when we have found the item, because the item can be uniquely identified by its reference $ref(o_i)$. The probability that o_i is found on level l can be calculated as follows.

$$\Pr(o_i \text{ is on level } l) = \frac{n_l}{n} \tag{53}$$

Now we are able to determine an upper bound for the expected number of page accesses needed to find o_i. When found, one writing access to update the bucket is needed.

$$D_{RLSH} \leq \left(\sum_{l=1}^{maxlevel} 2 \cdot l \cdot \frac{n_l}{n} \right) + 1 + C_{delete} \tag{54}$$

When expecting no growth, the index structures could be shrunk by reversing the page splitting during deletion (if the number of entries fall below a certain threshold). To avoid the additional complexity of this, we did not implement shrinking during deletion.

4 Comparison of index structures using cost models

In this section the different index structures are compared using the cost models presented in section 3. Mathematical modeling often simplifies the modeled system. Therefore we also compare the index structures by simulation (see section 6).

Each part of this section is dedicated to one criterion described in section 4.1. We begin by comparing the required diskspace (many of the cost formulas for the other criteria depend on the diskspace costs). We then compare the query evaluation costs, differentiating between the different query types. We continue by taking a look at the update costs and last but not least we inspect the creation costs.

4.1 Criteria for comparisons

In this section we outline how we compare the index structures. There are several important criteria with which the efficiency of an index structure may be specified. The most important one is the cost for the evaluation of a query. For the theoretical comparison we express this cost in the number of page accesses, assuming that the needed CPU-time can be neglected. Another important factor is the diskspace needed by an index structure. In dynamic environments the costs to update an index structure are at least as important as the query evaluation costs. An index does not come into existence spontaneously, so the time needed to construct the index also figures. Table 1 sums up the criteria used for the comparisons.

criterion	*measured for/in*
query evaluation time	no of page accesses
diskspace	no of pages
update	insertion deletion
creation	no of page accesses

Table 1: Criteria for comparison of index structures

4.2 Diskspace

First of all we compare the index structures in regard to the required diskspace. We start with the required diskspace, not because it is the most important criterion, but because many other criteria can be expressed in terms of the diskspace. We consider each index structure in turn, and end this section with a conclusion on the comparison.

In order to store tuples consisting of a signature and a reference to a data item, the sequential signature file index has the lowest demand for diskspace. This is not surprising, because it stores the tuples with no further information in one large file not organizing them in any way. For this reason we use the sequential signature file index as a reference to which all other index structures are compared to.

As already mentioned (in formula (13)) the storage costs for a sequential signature file index can be calculated as follows.

$$S_{SSF} = \left\lceil \frac{n \cdot S_{tuple}}{P} \right\rceil \text{ pages} \approx \frac{n \cdot S_{tuple}}{P} \text{ pages} \tag{55}$$

The diskspace required by a Russian doll tree (see (19)), can be approximated further assuming a large number of entries on a page. For the Russian doll tree we also assume a storage utilization of 64%, which is typical for an R-tree with uniform distribution an of keys and a linear split algorithm [BKSS90]. So α is equal to 0.64 in our case.

$$
\begin{aligned}
S_{RDT} &\approx \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{tuple}} \right\rfloor - 1} \text{ pages} && (56)\\
&\approx \frac{n}{\alpha \cdot \frac{P}{S_{tuple}}} \text{ pages} && (57)\\
&= \frac{1}{\alpha} \cdot S_{SSF} && (58)\\
&\approx 1.56 \cdot S_{SSF} && (59)
\end{aligned}
$$

Assuming the approximation (35) given [FNPS79], the storage cost for a extendible signature hashing index can be expresses as follows.

$$
\begin{aligned}
\overline{S}_{ESH} &\approx \left\lceil \frac{n \cdot S_{tuple}}{P} \right\rceil \cdot \log_2 e \text{ pages} && (60)\\
&\approx 1.44 \cdot S_{SSF} && (61)
\end{aligned}
$$

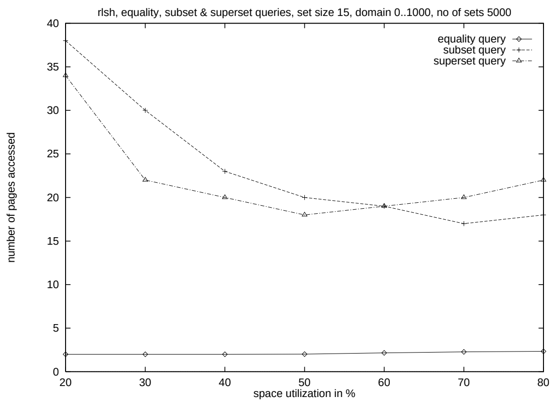

Figure 8: Query evaluation costs for different space utilizations

A high space utilization (i.e. a high load factor) leads to small index structures for the recursive linear signature hashing index, so usually a space utilization of 80% to 90% is chosen [RSD84]. A higher space utilization also means more items overflowing into recursive hash tables. This results in higher query evaluation costs. The effect of hitting overflow pages multiplies for queries with subset/superset predicates, because many different buckets are accessed during a query evaluation. The space utilization should not be too low, though, because there is another effect for subset/superset predicates. The lower the space utilization, the more splits occur, which leads to a larger depth of the hash table on the top level. A larger depth means that for queries containing subset/superset predicates, more sets have to be generated and therefore more pages have to be accessed. We achieved the best results for a space utilization of 60% (see figure 8).

$$L = \frac{0.6 \cdot P}{S_{tuple}} \tag{62}$$

We can now approximate the required diskspace for the recursive linear signature hashing index (see (41)) in terms of the sequential signature file index.

$$\begin{aligned}
S_{RLSH} &= \sum_{l=1}^{maxlevel} (S_{DIR_l} + S_{BUCK_l}) && (63)\\
&= \sum_{l=1}^{maxlevel} \left(\frac{u_l \cdot id}{P} + u_l \right) && (64)\\
&= \sum_{l=1}^{maxlevel} \left(\frac{n_l \cdot id}{L \cdot P} + \frac{n_l}{L} \right) && (65)\\
&= \sum_{l=1}^{maxlevel} \left(\frac{n_l \cdot id \cdot S_{tuple}}{0.6 \cdot P^2} + \frac{n_l \cdot S_{tuple}}{0.6 \cdot P} \right) && (66)
\end{aligned}$$

When assuming a page size of 4096 bytes and the size of a reference *id* of 8 bytes, we can reference up to 512 buckets with one page of the directory. So the number of buckets determines the size of the index structure, the number of directory pages plays only a minor role.

$$S_{RLSH} \approx \sum_{l=1}^{maxlevel} \frac{n_l \cdot S_{tuple}}{0.6 \cdot P} \tag{67}$$

Ramamohanarao and Sacks-Davis show that there are at most two to three recursive hash tables, which are very small compared to the hash table on the top level. So almost all data items can be inserted into the hash table they are intended to be inserted, i.e. $n_l \approx r_l$.

$$\begin{aligned}
S_{RLSH} &\approx \sum_{l=1}^{maxlevel} \frac{r_l \cdot S_{tuple}}{0.6 \cdot P} && (68)\\
&= \frac{n \cdot S_{tuple}}{0.6 \cdot P} && (69)\\
&\approx 1.67 \cdot S_{SSF} && (70)
\end{aligned}$$

The results of this section concerning the diskspace required by the different index structures are summed up in table

4.3 Query evaluation costs

We look at queries with equality, subset, superset, and intersection predicates (in this order). For the theoretical calculations we assume that the time used up for page accesses is larger by several orders of magnitude than the time needed on the CPU. Therefore, we neglect the CPU costs and concentrate on the number of page accesses.

index structure	*relative size*
sequential signature file	1
Russian doll tree	1.56
extendible signature hashing	1.44
recursive linear signature hashing	1.67

Table 2: Relative (theoretical) sizes of index structures

4.3.1 Equality predicates:

It is straightforward to compute the costs for a query evaluation employing the sequential signature file index structure. The signature file has to be traversed fully and each data item, whose signature qualifies, is fetched from disk. So the costs for a query evaluation are (see 14):

$$C_{=}^{SSF} = S_{SSF} + C_{fetch} \tag{71}$$

The query evaluation costs for the Russian doll tree were calculated as follows (see 22):

$$C_{=}^{RD} \leq 1 + \sum_{j=2}^{m} \Pr(sig(S_Q) \subseteq sig(S_{B_j})) \tag{72}$$

When evaluating a query in a typical R-tree, approximately 10% of all pages in the tree are touched [Gut84]. So the cost $C_{=}^{RD}$ can be estimated by

$$\begin{aligned} C_{=}^{RD} &\approx \frac{1}{10} \cdot S_{RDT} + C_{fetch} && (73)\\ &\approx \frac{1.56}{10} \cdot S_{SSF} + C_{fetch} && (74)\\ &\approx 0.156 \cdot S_{SSF} + C_{fetch} && (75) \end{aligned}$$

A Russian doll tree is expected to be about 6 to 7 times faster than a sequential signature file when evaluating a query with equality predicates.

The extendible signature hashing index needs only 2 page accesses to find the bucket in which qualifying signatures may be found (see 36)

$$C_{=}^{ESH} = 2 + C_{fetch} \tag{76}$$

For large databases the size S_{SSF} of a sequential signature file is equal to several hundred pages, so the query evaluations costs for the sequential signature file and the Russian doll tree are no match for the two page accesses of the extendible signature hashing index.

The costs for the recursive linear signature hashing index depend on the number of recursive hash tables that exist (see 49)

$$C_{=}^{RLSH} = 2 \cdot maxlevel + C_{fetch} \tag{77}$$

As long as the number of recursive hash tables can be held low, the recursive linear signature hashing index is still fast compared to a sequential signature file of a Russian doll tree. It cannot match the speed of extendible signature hashing, though.

4.3.2 Subset and superset predicates:

Query evaluation for queries with subset or superset predicates is very similar, so we present them in one section.

The query evaluation costs for the sequential signature file does not depend on the query type. For all query types it is

$$C_{\subseteq\supseteq}^{SSF} \;=\; S_{SSF} + C_{fetch} \tag{78}$$

For a Russian doll tree the query evaluation costs for queries with subset predicates are almost identical to the query evaluation costs for queries with equality predicates. In a leaf node the signatures are now checked for the subset property instead of equality (which does not reduce the number of touched pages). The number of false drops may also vary, because of the different false drop probabilities.

$$\begin{aligned} C_{\subseteq}^{RD} &\leq 1 + \sum_{j=2}^{m} \Pr(sig(S_Q) \subseteq sig(S_{B_j})) && (79)\\ &\approx 0.156 \cdot S_{SSF} + C_{fetch} && (80) \end{aligned}$$

Again we expect the Russian doll tree to 6 to 7 times faster than the sequential signature file index. A Russian doll tree should not be used to support queries with superset predicates (see section 3.2), so we do not have a cost formula for this query type.

The query evaluation costs for an extendible signature hashing index depend on the depth d of the hash table (see 38).

$$\begin{aligned} C_{\subseteq,\supseteq}^{ESH} &\approx 2^{\lceil \frac{d}{2} \rceil} \cdot 2 + C_{fetch} && (81)\\ &= 2^{\lceil \frac{d}{2} \rceil + 1} + C_{fetch} && (82) \end{aligned}$$

We can estimate d by using the formulas for the size of an extendible signature hash table (see (34) and (61)).

$$\begin{aligned} S_{ESH} = S_{DIR} + S_{BUCK} &= S_{SSF} \cdot \log_2 e && (83)\\ \frac{2^d \cdot id}{P} + m &= S_{SSF} \cdot \log_2 e && (84)\\ d &= \log_2\left(\frac{(S_{SSF} \cdot \log_2 e - m) \cdot P}{id}\right) && (85)\\ d &= \log_2(1.44 \cdot S_{SSF} - m) + \log_2(P) - \log_2(id) && (86) \end{aligned}$$

The size S_{SSF} of the corresponding sequential signature file, the size P of a page, and the size id of a reference are constant, so d is a function of m, the number of buckets in the hash table. The value of m has to be between the lowest possible number of buckets in which to fit the information and the highest possible number of buckets the directory can support, so

$$\frac{n \cdot S_{tuple}}{P} \leq m \leq 2^d \tag{87}$$

$m = 2^d$ is the best case, because in this case all keys are spread (uniformly) across all buckets and an overflow will be unlikely. $m = \frac{n \cdot S_{tuple}}{P}$ is the worst case, because all keys are cramped into the lowest possible number of buckets, which means that the data is heavily skewed. Let us now look at the worst and best case.

$$\begin{aligned} d_{worst} &= \log_2(1.44 \cdot S_{SSF} - \frac{n \cdot S_{tuple}}{P}) + \log_2(P) - \log_2(id) && (88) \\ &= \log_2(1.44 \cdot S_{SSF} - S_{SSF}) + \log_2(P) - \log_2(id) && (89) \\ &= \log_2(S_{SSF}) + \log_2(0.44) \log_2(P) - \log_2(id) && (90) \end{aligned}$$

The page size P is equal to 4096 bytes and the size id of a reference is equal to 8 bytes in our case. Therefore

$$d_{worst} \approx \log_2(S_{SSF}) + 8 \tag{91}$$

For the query evaluation costs we get

$$\begin{aligned} C^{ESH}_{\subseteq,\supseteq worst} &\approx 2^{\frac{d_{worst}}{2}+1} + C_{fetch} && (92) \\ &= 2^5 \cdot \sqrt{S_{SSF}} + C_{fetch} && (93) \\ &= 32 \cdot \sqrt{S_{SSF}} + C_{fetch} && (94) \end{aligned}$$

For the best case on the other hand we get (starting with (84))

$$\begin{aligned} \frac{2^{d_{best}} \cdot id}{P} + 2^{d_{best}} &= S_{SSF} \cdot \log_2 e && (95) \\ 2^{d_{best}} &= \frac{1.44 \cdot S_{SSF} \cdot P}{id + P} && (96) \\ d_{best} &= \log_2(1.44) + \log_2(S_{SSF}) + \log_2(P) - \log_2(id + P) && (97) \end{aligned}$$

For a page size P of 4096 bytes and a size of $id = 8$ bytes for references, we can estimate d_{best}.

$$d_{best} \approx \log_2(S_{SSF}) + 0.5 \tag{98}$$

We can now approximate the query evaluation costs for the best case.

$$\begin{aligned}
C^{ESH}_{\subseteq,\supseteq best} &\approx 2^{\frac{d_{best}}{2}+1} + C_{fetch} && (99)\\
&= 2^{1.25} \cdot \sqrt{S_{SSF}} + C_{fetch} && (100)\\
&\approx 2.38 \cdot \sqrt{S_{SSF}} + C_{fetch} && (101)
\end{aligned}$$

When the sequential signature file reaches a size of about 1000 pages the trade-even point for the worst case of the extendible signature hashing is reached. For the best case this point is already reached for a size of about 6 pages. For the Russian doll tree the trade even point is reached much later, at a size of the tree of about 65.000 pages and and a size of 360 pages respectively. Based on the observations made during the simulation (see section 6.2) we can assure, however, that the extendible signature hashing index performs very close to the best case

For the recursive linear signature hashing index we also analyze the worst and best case behavior. We start with the worst case, which happens when we always insert into exactly one bucket on each level, which then overflows into exactly one bucket on the next level. That means, we only have one bucket on each level containing signatures, all other buckets are empty. During query evaluation we have to look at one bucket on each level. The maximum number of levels in the worst case is

$$maxlevel_{worst} = \lceil \frac{n}{L} \rceil \qquad (102)$$

So for the query evaluation costs we get

$$\begin{aligned}
C^{RLSH}_{\subseteq,\supseteq worst} &\approx 2 \cdot maxlevel + C_{fetch} && (103)\\
&= 2 \cdot \lceil \frac{n}{L} \rceil + C_{fetch} && (104)\\
&\approx 2 \cdot 1.56 \cdot S_{SSF} + C_{fetch} && (105)\\
&= 3.12 \cdot S_{SSF} + C_{fetch} && (106)\\
& && (107)
\end{aligned}$$

In the best case all data items are inserted into the topmost level and stay there, i.e. the recursive hash tables are very small or nonexistent. So:

$$\begin{aligned}
C^{RLSH}_{\subseteq,\supseteq best} &\approx 2^{\frac{d_1}{2}} \cdot 2 + C_{fetch} && (108)\\
& && (109)
\end{aligned}$$

The depth d_1 of the hash table on the topmost level depends on the number of buckets u_1 on the first level.

$$\begin{aligned}
d_1 &= \log_2(u_1) && (110)\\
u_1 &= \log_2(\frac{n_1}{L}) && (111)
\end{aligned}$$

We assume that all items are inserted into the hash table on the first level, so $n_1 = n$

$$\begin{aligned} C^{RLSH}_{\subseteq,\supseteq best} &\approx 2^{\frac{\log_2\left(\frac{n}{L}\right)}{2}} \cdot 2 + C_{fetch} && (112) \\ &= \sqrt{\frac{n}{L}} \cdot 2 + C_{fetch} && (113) \end{aligned}$$

We remember that L was the load factor (see 62), i.e. how much signature and reference tuples fit on one page. We assumed a storage utilization of 60%, so

$$\begin{aligned} C^{RLSH}_{\subseteq,\supseteq best} &\approx \sqrt{\frac{n}{\frac{0.6 \cdot P}{S_{tuple}}}} \cdot 2 + C_{fetch} && (114) \\ &\approx \sqrt{1.56 \cdot S_{SSF}} \cdot 2 + C_{fetch} && (115) \\ &\approx 2.5 \cdot \sqrt{S_{SSF}} + C_{fetch} && (116) \end{aligned}$$

The worst case query evaluation costs for the recursive linear signature hashing are about 3 times higher than the query evaluation costs for a sequential signature file. The differences between the query evaluation costs for an extendible signature hashing index and a recursive linear signature hashing index are very small (about 5%) for the best case. Unlike the extendible signature hashing index the recursive linear signature hashing index does not always show near optimal behavior (see measurements with skewed data in section 6.2).

4.3.3 Intersection predicates

In this section we look at the evaluation costs of queries containing intersection predicates. Both of the hashing index structures are not suitable for supporting queries with intersection predicates (see section 3.3 and 3.4). Therefore we only discuss the sequential signature file and the Russian doll tree.

For the sequential signature file nothing changes compared to the equality, subset, and superset predicates. The query evaluation costs are still

$$C^{SSF}_{\cap} = S_{SSF} + C_{fetch} \quad (117)$$

Theoretically the query evaluation costs for the Russian doll tree should also be independent of the query type. This is not quite true, however, because the false drop probabilities for intersection predicates are much higher than the false drop probabilities for the other predicate types. When keeping the same signature size as for equality, subset, and superset predicates, more (false drop) signatures qualify. When increasing the size of the signatures to lower the number of false drops, the tree becomes larger. Either way results in a higher number of page accesses during a query evaluation. So we can only give a lower bound for the query evaluation costs for intersection predicates.

$$C^{RD}_{\cap} \geq 0.156 \cdot S_{SSF} + C_{fetch} \quad (118)$$

4.3.4 Summary

Table 3 gives an overview for the expected query evaluation costs for the different index structures. C_{fetch} are the costs to fetch the actual data items.

index structure	*query type*		
	equality pred.	*sub-/superset pred.*	*intersection pred.*
seq. sig. file	$S_{SSF} + C_{fetch}$	$S_{SSF} + C_{fetch}$	$S_{SSF} + C_{fetch}$
Russian doll tree	$0.156 \cdot S_{SSF} + C_{fetch}$	$0.156 \cdot S_{SSF} + C_{fetch}$ / -	$\geq 0.156 \cdot S_{SSF} + C_{fetch}$
ext. sig. hashing	$2 + C_{fetch}$	best: $2.38 \cdot \sqrt{S_{SSF}} + C_{fetch}$ worst: $32 \cdot \sqrt{S_{SSF}} + C_{fetch}$	-
rec. lin. sig. hashing	$2 \cdot maxlevel + C_{fetch}$	best: $2.5 \cdot \sqrt{S_{SSF}} + C_{fetch}$ worst: $3.12 \cdot S_{SSF} + C_{fetch}$	-

Table 3: Expected query evaluation costs

4.4 Update costs

In this section we examine the costs for updates. We distinguish between insertions and deletions of data items in the index. Updates that change a data item can be emulated by a deletion of the original data item and an insertion of the changed data item.

4.4.1 Insertion costs

The insertion costs for the sequential signature file are straightforward. We have to read the last page of the index, insert the new item there and write the page back. So

$$I_{SSF} = 2 \tag{119}$$

In a Russian doll tree, all references to the data items are stored in the leaf nodes. If we assume a storage utilization of 64%, then we can calculate the number $node_{leaf}$ of leaf nodes.

$$node_{leaf} \approx \frac{n \cdot S_{tuple}}{0.64 \cdot P} \tag{120}$$

The fanout of the inner nodes, i.e. the number of nodes that are referenced by the inner nodes $node_{fanout}$ is approximately equal to

$$node_{fanout} \approx \left\lfloor \frac{0.64 \cdot P}{S_{tuple}} \right\rfloor \tag{121}$$

A Russian doll tree has minimal height, when all items are distributed evenly in the tree. A lower bound for the height $H(T)$ of a Russian doll tree is given by

$$H(T) \geq \log_{node_{fanout}}(node_{leaf}) \tag{122}$$

When inserting an item, we have to find the correct leaf node checking $H(T)$ nodes on the way down. After inserting the item we have to go back up and adjust the signature in the parent nodes. The total insertion costs are therefore

$$I_{RDT} \approx 2 \cdot \log_{node_{fanout}}(node_{leaf}) \tag{123}$$

The extendible signature hashing index structure needs to reading accesses to find the right bucket and one writing access to insert the item into the bucket. So the total page accesses are

$$I_{ESH} = 3 \tag{124}$$

For the recursive linear signature hashing index we assume that almost all of the data items are inserted into the topmost hash table, i.e. the probability that an item is inserted into one of the lower recursive hash tables is near zero.

$$\begin{aligned} I_{RLSH} &\approx 2 + \left(\sum_{l=2}^{maxlevel} 2 \cdot (l+1) \cdot \Pr(bucket_{li} = bucket_{max}) \right) + 1 && (125) \\ &\approx 3 && (126) \end{aligned}$$

The sequential signature file, the extendible signature hashing index, and the recursive linear signature hashing index have constant time for insertion, while the insertion costs for the Russian doll tree depend on the height of the tree. The sequential signature file has the lowest insertion costs, followed closely by the hashing index structures. We expect the Russian doll tree to have the highest insertion costs. Table 4 sums up the insertion costs.

index structure	*page accesses*
sequential signature file	2
Russian doll tree	$2 \cdot H(T)$
extendible signature hashing	3
recursive linear signature hashing	3

Table 4: Insertion costs of index structures

4.4.2 Deletion costs

When deleting a data item, we have to search the index structure for a certain signature and identifier. When we find a signature equal to the query signature, we have to compare the identifiers to check, if we have found the data item that is to be deleted. On finding the data item we have to delete the item and readjust the index structure.

The average costs for deleting an item from a sequential signature file are

$$\overline{D}_{SSF} = \frac{1}{2} \cdot S_{SSF} + 1 + C_{delete} \tag{127}$$

When searching for an item in a Russian doll tree index about 10% of the pages have to be touched. When the data item has been found and deleted all pages on the path to the root have to be modified. The height $H(T)$ of the tree can be estimated by (122).

$$D_{RDT} \approx 0.156 \cdot S_{SSF} + H(T) + C_{delete} \tag{128}$$

The extendible signature hashing index needs to read two pages to reach the correct bucket and write one back to modify the index.

$$D_{ESH} = 3 + C_{delete} \tag{129}$$

For the recursive linear signature hashing index we assume that almost all deletions take place in the topmost hash table and that the modification of recursive hash tables is rare.

$$\begin{aligned} D_{RLSH} &\leq \left(\sum_{l=1}^{maxlevel} 2 \cdot l \cdot \frac{n_l}{n} \right) + 1 + C_{delete} && (130) \\ &\approx 3 + C_{delete} && (131) \end{aligned}$$

We expect the two hashing index structures to have the fastest deletion operations. Comparing the deletion costs for the sequential signature file and the Russian doll tree we arrive at the following conclusion.

$$\begin{aligned} \overline{D}_{SSF} - D_{RDT} &\approx \frac{1}{2} \cdot S_{SSF} + 1 - 0.156 \cdot S_{SSF} - H(T) && (132) \\ &\approx 0.35 \cdot S_{SSF} - \log_{node_{fanout}}(1.56 \cdot S_{SSF}) && (133) \end{aligned}$$

The deletion costs speak in favor of the Russian doll tree. The larger S_{SSF} becomes, the larger the difference will become in (133) between the deletion costs for the sequential signature file and the Russian doll tree. Table 5 sums up the results for the deletion costs.

4.5 Creation costs

The cost model for the creation costs is kept very simple. When inserting n data items, we assume that n insertion operations take place. So the creation costs for each index structure are the costs for one insertion times n.

index structure	*page accesses*
sequential signature file	$\frac{1}{2} \cdot S_{SSF} + 1 + C_{delete}$
Russian doll tree	$0.156 \cdot S_{SSF} + H(T) + C_{delete}$
extendible signature hashing	$3 + C_{delete}$
recursive linear signature hashing	$3 + C_{delete}$

Table 5: Deletion costs of index structures

$$B_{SSF} = I_{SSF} \cdot n = 2 \cdot n \tag{134}$$
$$B_{RDT} = I_{RDT} \cdot n \approx 2 \cdot H(T) \cdot n \tag{135}$$
$$B_{ESH} = I_{ESH} \cdot n = 3 \cdot n \tag{136}$$
$$B_{RLSH} = I_{RLSH} \cdot n \approx 3 \cdot n \tag{137}$$

Table 6 sums up the results of the costs for creating each index structure.

index structure	*page accesses*
sequential signature file	$2 \cdot n$
Russian doll tree	$2 \cdot H(T) \cdot n$
extendible signature hashing	$3 \cdot n$
recursive linear signature hashing	$3 \cdot n$

Table 6: Insertion costs of index structures

5 Description of the simulation environment

We mainly compare the different index structures by means of a simulation. In this section we describe the environment in which the simulation takes place. The environment includes the description of the used hardware, software, etc.. The parameters for the generation of the test databases are also discussed. Then we explain how the simulation was run, i.e. how the queries were generated and the measured data was obtained.

5.1 The environment of the simulation

5.1.1 Employed hard- and software

The benchmark runs were conducted on a lightly loaded Sparc20 with 128 MByte main memory running under Solaris 2.5.1. The algorithms were not parallelized in any way. 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 [BP94]. We took advantage of plain pages (with a size of 4K), thereby improving the performance of the index structures. We implemented the data structures and algorithms of the index structures in C++ using the GNU C++ Compiler Version 2.7.2 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 UNIX file system. Last, but not least, we cleared the internal disk cache of relevant pages by transferring 2 MBytes of data in between the queries.

5.1.2 Generating the databases

Unfortunately, we had no real data available for our simulation. That means, we had to generate databases containing data items with set-valued attributes. We varied the size of the database (in number of data items contained), the size of the individual sets (in number of items contained), and the size of the domain of the sets. For a summary see table 7. Each data item was stored on a separate page to eliminate any effects on the measured data caused by clustering.

parameter	*min value*	*max value*
size of database	1000	50000
size of sets	10	80
size of domain	100	1000000

Table 7: Parameters for generation of database

The data items in the databases were generated randomly. We used two different distributions. One was an uniform distribution, which models 1:n relationships between unique data items, like parent-child relationships or owning relationships. The other distribution we used was Zipf, which is useful when dealing with non-unique objects, like words or numbers in documents. Figure 9 demonstrates the differences in the probability that an element of the domain is chosen for a particular set between uniform and Zipfian distribution. One domain includes elements numbered from 0 to 100, the other elements numbered from 0 to 1000. (Uniform distribution is a special case of the Zipfian distribution with $z = 0$.) The probability for uniform distribution in the diagram on the right hand side has the value 0.0001 for all elements of the domain.

5.1.3 Generating the queries

The queries for the benchmarks runs were also generated randomly. We varied the size of the query sets, the size of the domains of the query sets, and the type of queries. For a summary see table 8.

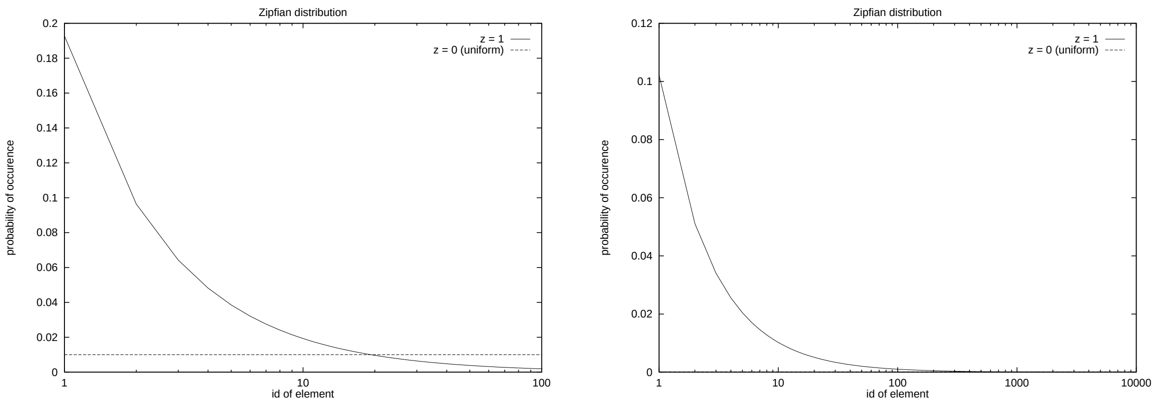

Figure 9: Uniform vs. Zipfian distribution

parameter	*min value*	*max value*
size of query set	10	80
size of domain of query set	100	1000000
query type	$=, \subseteq, \supseteq, \cap$	

Table 8: Parameters for generation of queries

5.2 Criteria for comparisons

In the last section we defined the environment in which the index structures are compared. In this section we briefly review the criteria for comparisons. We already mentioned several criteria in section 4.1 for the theoretical analysis. For the comparison by simulation we expand those criteria, i.e. we not only look at the number of page accesses, but also examine the total elapsed time for each criterion. Table 9 sums up the criteria used for the comparisons by simulation.

We have not considered the main memory requirements of each index structure, because we did not buffer any pages in main memory. That means for each page access we had to fetch the page from disk.

6 Comparison of index structures by simulation

In this section the different index structures are compared by benchmarks. Each part of this section is dedicated to one criterion listed in table 9. We use the same order as in section 4, so we begin by comparing the required diskspace. We then compare the query evaluation costs, differentiating between the different query types. We continue by taking a look at the update costs and last but not least we inspect the creation costs.

criterion	*measured for/in*
query evaluation time	total time no of page accesses
diskspace	no of pages
update	insertion deletion
creation	total time

Table 9: Criteria for comparison of index structures

6.1 Diskspace

In this section the results of our simulation are depicted. We vary the parameters database size, set size, and domain size as shown in table 7.

6.1.1 Size of database:

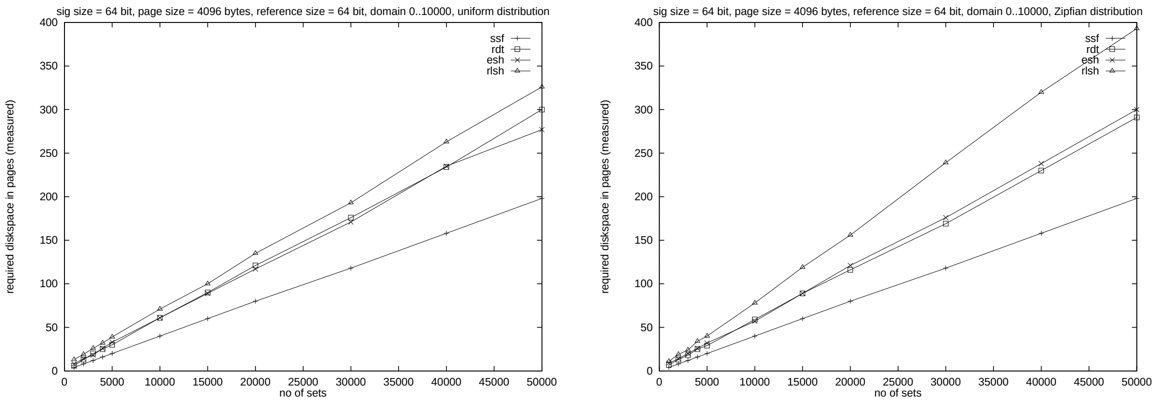

Figure 10: Uniform vs. Zipfian distribution (varying database size)

As can be clearly seen in figure 10 the required diskspace grows linearly with the size of the database for all index structures. The left hand side of figure 10 displays the diskspace that was allocated for uniformly distributed data. The data in many databases is not uniformly distributed, however. For that reason we introduce skew in the form of a Zipfian distribution (see table 9) to see what happens. The right hand side of figure 10 shows the required diskspace for skewed data.

The skew has no influence on the diskspace required for the sequential signature file index, which is not surprising. In the sequential signature file index all signatures and references to data items are stored "as is" and are not organized in any way, i.e. they are stored sequentially in one large file. As long as we do not change the parameters affecting the size of the signatures (or references) no effect will be seen.

The Russian doll tree also has no problems coping with skewed data. We noted a very small decrease (3% at most) in the allocated diskspace. We do not know the reason for this slight decrease yet.

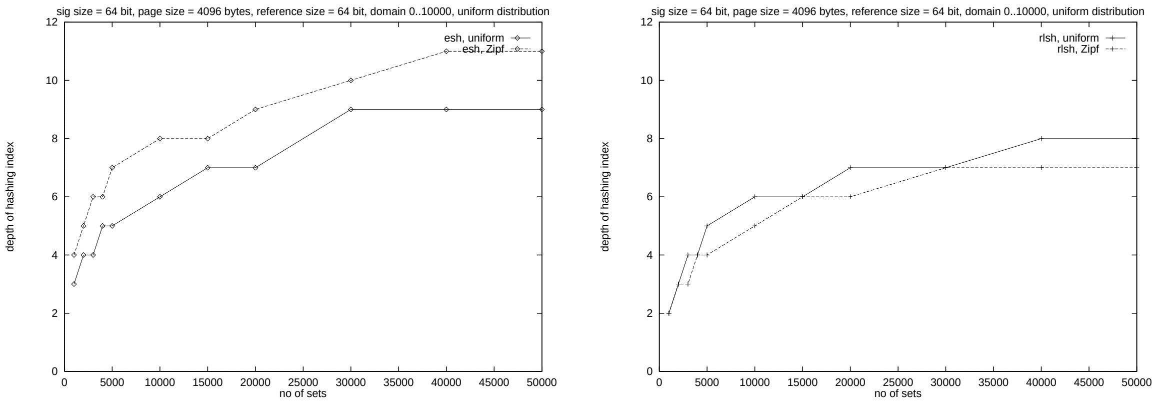

Figure 11: Depth of hashing index structures (varying database size)

For the same number of data items the directory of the extendible signature hashing index is two to four times larger when using skewed data (see left hand side of figure 11). On the other side, many entries in the directory are either empty, i.e. not referencing any bucket, or sharing the same bucket. So, although the size of the directory increases, the number of buckets in the index structure stays roughly the same.

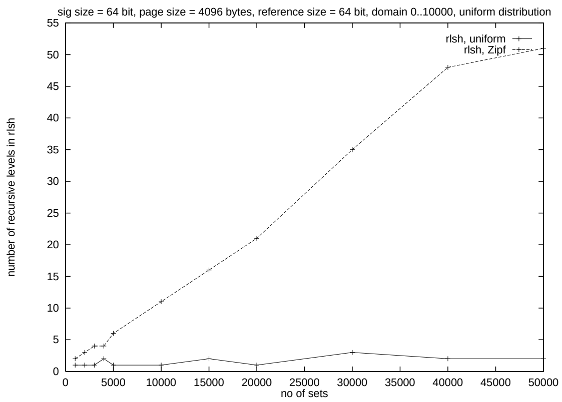

Figure 12: Recursive levels for rec. lin. sig. hashing (varying database size)

Skewed data has the greatest impact on the recursive linear signature hashing index. Unlike the extendible signature hashing index the higher rate of directory splitting is suppressed (see right hand side of figure 11). This, however, comes at a price. The number of buckets overflowing in not smaller than in extendible hashing. All data items from overflowing buckets are reinserted into a recursive hash table at a lower level. This leads to a large number of recursive hash tables (see figure 12), which is unacceptable for an index structure using hashing. As described in section 3.4 we have to traverse all recursive hash tables during a query evaluation. The unpalatable consequences on the query costs can be seen in section 6.2.

6.1.2 Size of data items:

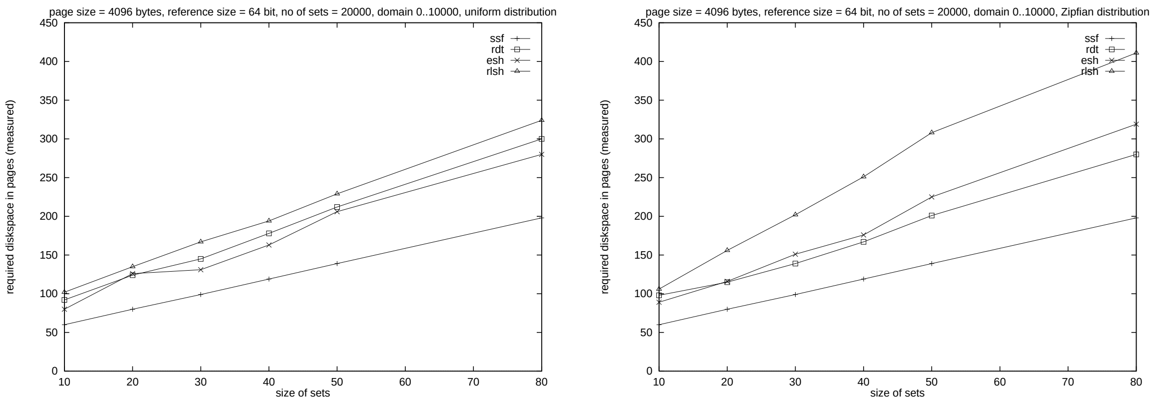

Figure 13: Uniform vs. Zipfian distribution (varying set size)

When we varied the size of the sets, we also modified the size of the signatures in order to achieve the same false drop probability for all cases. The left hand side of figure 13 shows the required diskspace for uniform distribution of the data. The diskspace grows linearly with the size of the data sets (and signatures) for all index structures. The results are similar to those for the variation of the database size. The sequential signature file requires the least diskspace of all index structures, while the recursive linear signature hashing has the highest overhead.

On the right hand side of figure 13 the results for skewed data are depicted. The sequential signature file is not influenced in any way by the skew. Strangely the Russian doll tree seems to be influenced positively by the skew, needing up to 7% less diskspace. The directory of the extendible signature hashing index is enlarged by a factor of 2 to 4 again, resulting in a lower storage utilization. The recursive linear hashing index has the most severe problems when dealing with skewed data. This becomes apparent, when looking at the number of recursive hash tables 14.

6.1.3 Size of domain:

The domain size does not have any influence on the size of the sequential signature file index structure (see left hand side of figure 15). The size of the domain also seems not to have an effect on the Russian doll tree for uniformly distributed data. However, it does

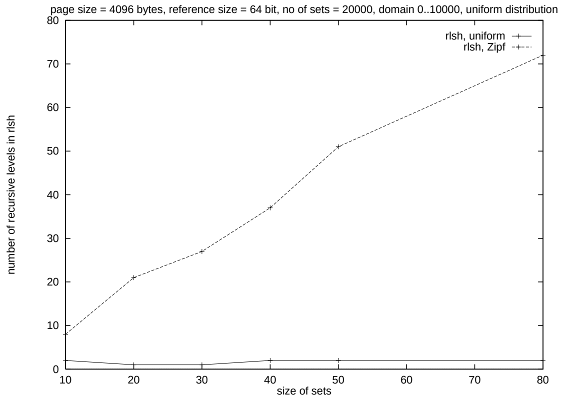

Figure 14: Recursive levels for rec. lin. sig. hashing (varying set size)

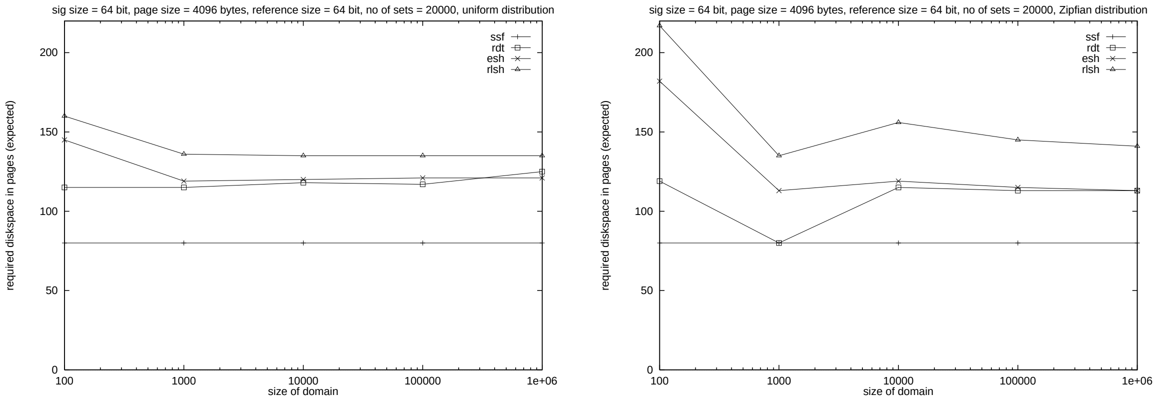

Figure 15: Uniform vs. Zipfian distribution (varying domain size)

have an influence on the hashing index structures. The smaller the domain, the smaller the range becomes from which the elements of the sets are picked. The probability to generate similar or even identical signatures for data items increases when the variety of values in those sets decreases. The buckets will not be filled as evenly as with a large domain, so the buckets will overflow faster. This leads to an expansion of the hash table in the case of extendible signature hashing (see left hand side of figure 16) and an increase in the number of recursive hash tables in the case of recursive linear signature hashing (see figure 17), respectively.

The right hand side of figure 15 shows the influence of skewed data on the index structures. As in all previous benchmarks skewed data has no influence on the sequential signature file index. The evaluation of the behavior of the Russian doll tree index is the most difficult of all index structures and we do not have an explanation yet for the

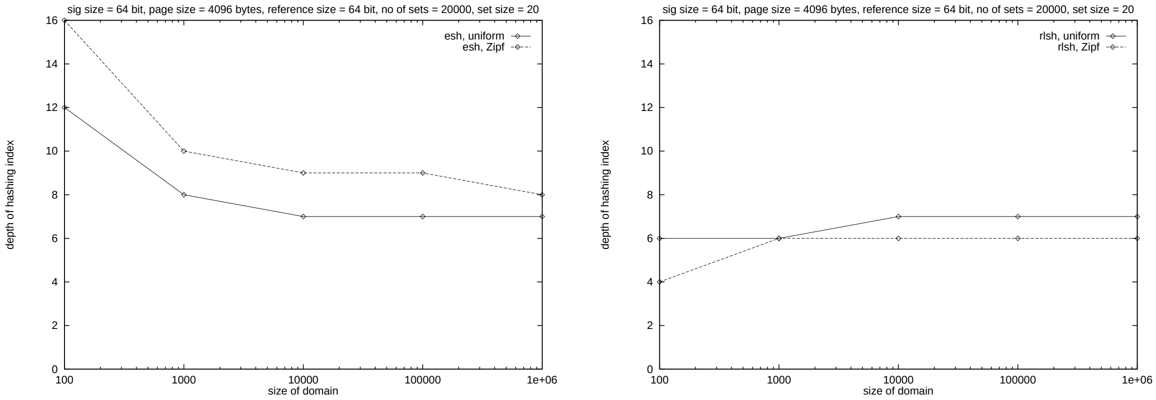

Figure 16: Depth of hashing index structures (varying domain size)

Russian doll tree with skewed data. The effect of the varying domain size that could be observed for uniformly distributed data is amplified for skewed data. When using skewed data, the choice on which elements to include in a set is even more restricted, because few elements of domains are picked very often. The extendible signature hashing index expands its directory (see figure 16), while the recursive linear signature hashing index structure increases the number of recursive hash tables dramatically (see figure 17).

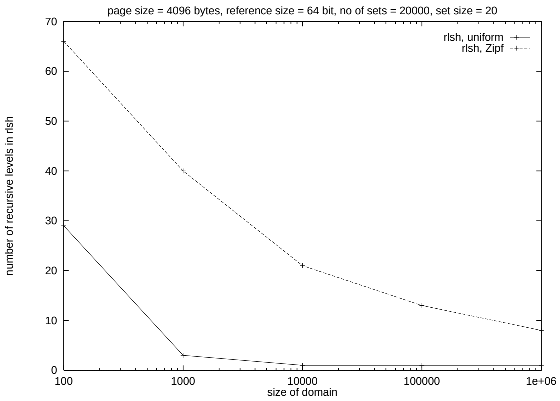

Figure 17: Recursive levels for rec. lin. sig. hashing (varying domain size)

6.1.4 Summary

The results of the previous sections concerning the diskspace required by the different index structures can be recapitulated in a few points. The sequential signature file index has the lowest diskspace requirements of all index structures and is not affected by skewed data. There is no clear winner, when comparing the Russian doll tree with extendible signature hashing, although the extendible signature hashing is at a disadvantage with skewed data. The recursive linear signature hashing index demands the most diskspace of all index structures and the size of the diskspace increases noticeably with skewed data. In table 10 the results of the previous sections are summarized.

index structure	*relative size*	
	unif. dist.	*Zipf dist.*
sequential signature file	1	1
Russian doll tree	1.54	1.51
extendible signature hashing	1.58	1.62
recursive linear signature hashing	1.99	2.11

Table 10: Relative (measured) sizes of index structures

6.2 Query evaluation costs

In this section we present the results of our benchmarks for the query evaluation costs. We examine the different query types, the parameters database size, set size, and domain size, and the influence of skewed data on the costs. At the end of the section the results for the query evaluation costs are briefly summarized.

6.2.1 Equality predicates

In this section we investigate the query evaluation costs for queries with equality predicates. We distinguish between the variation of the database size, the variation of the set size, and the variation of the domain size.

Size of database: Figure 18 shows the query evaluation costs for uniformly distributed data. The left hand side of figure 18 depicts the total running time of the query evaluation, while the right hand side presents the costs in terms of page accesses.

As expected the sequential signature file index has the highest evaluation costs, because it has to traversed the entire signature file to find the qualifying data items. The costs grow linearly with the size of the database (as the size of S_{SSF} of the sequential signature file grows linearly with the database size). The Russian doll tree is several times faster than the sequential signature file (1.5 times for 1000 data items, 10 times for 50,000 data items). The costs for the Russian doll tree seem to grow logarithmically. Extendible signature hashing and recursive linear signature hashing are by far the two fastest index structures. In order to evaluate a query with equality predicates, the extendible hashing index and the recursive linear signature hashing index merely need two page accesses per query.

We also examine the influence of skewed data on the index structures (see figure 19). The performance of the sequential signature file is independent of the distribution of the data. The Russian doll tree and the extendible signature hashing index also do not seem

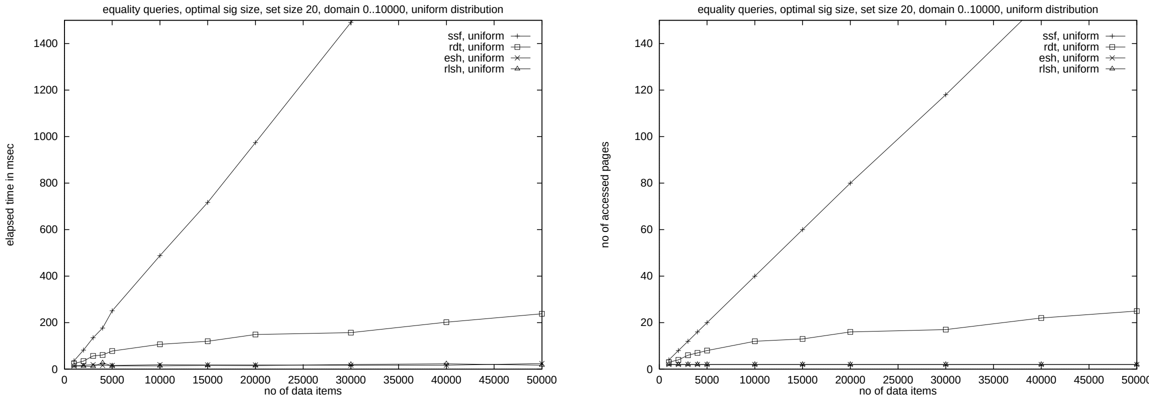

Figure 18: Query eval. costs (equality pred., uniform dist., varying database size)

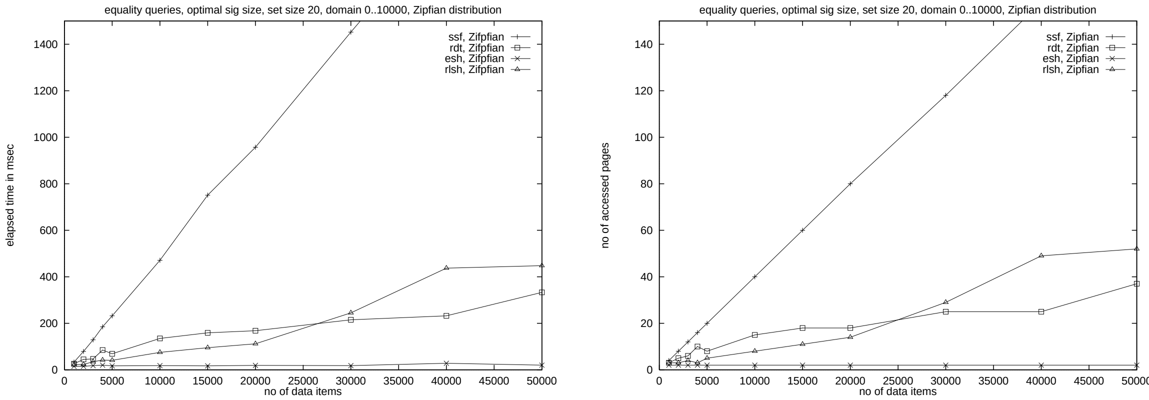

Figure 19: Query eval. costs (equality pred., Zipfian dist., varying database size)

to be influenced in terms of the query costs when confronted with skewed data. In the case of the recursive linear signature hashing index matters look bleak. It suffers heavy performance losses, due to the large number of recursive hash tables which have to be searched. For databases containing more than 30,000 data items it is inferior to the tree index, which is very disappointing for a hashing index.

Size of data items: Figure 20 shows the query evaluation costs for uniformly distributed data when the size of the data items is varied. On the left hand side we have the total elapsed time, on the right hand side the total number of page accesses.

The size of a sequential signature file index depends directly on the size of the signatures in it. When increasing the size of the sets of the data items (thereby increasing the size of the signatures to hold the false drop probability on the same level), the size of the

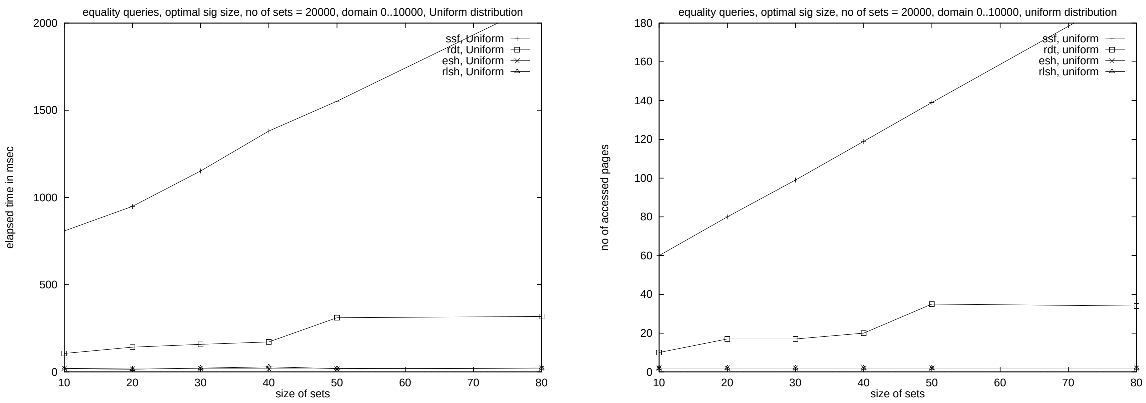

Figure 20: Query eval. costs (equality pred., uniform dist., varying set size)

index also increases leading to higher query costs. The remarkable bend for the query evaluation costs of the Russian doll tree can be explained by the height of the tree. Up to a size of 40 elements per set, the tree has a height of two. For 50 or more elements per set, however, the tree has a height of three. For the hashing index structures larger data items lead to larger index structures (see section 6.1.2 on diskspace), but have no influence on the query evaluation costs. Every data item can still be reached with two page accesses.

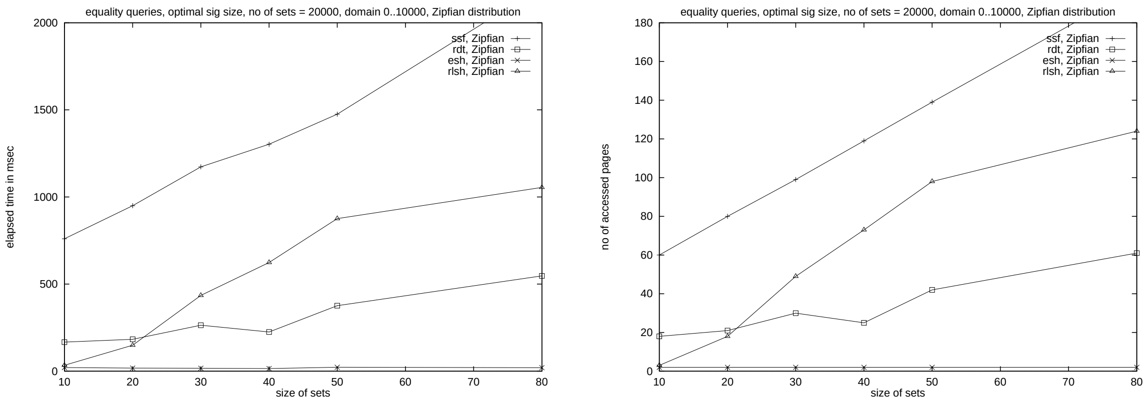

Figure 21: Query eval. costs (equality pred., Zipfian dist., varying set size)

Figure 21 exhibits the role of skewed data. As always the sequential signature file is not influenced by skewed data. The Russian doll tree loses some performance compared to uniformly distributed data. The size of the extendible signature hashing index increases even more for skewed data, but this has no influence on the query evaluation costs. The recursive linear signature hashing index performs poorly for skewed data, because of the large number of recursive hash tables.

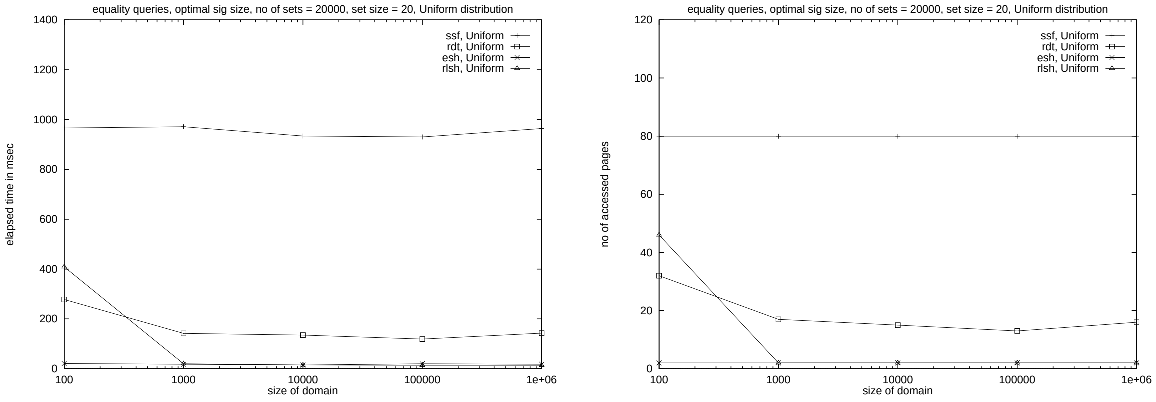

Figure 22: Query eval. costs (equality pred., uniform dist., varying domain size)

Size of domain: Figure 22 describes the query evaluation costs (total time and number of page accesses) for varying domain sizes. As can be clearly seen the sequential signature file index is not affected by the variation of the domain size. The Russian doll tree has slightly higher query evaluation costs for small domains, although we do not know the reason yet. The query evaluation costs for the extendible signature hashing index are not influenced by small domains, the required diskspace increases, however (see section 6.1.3). The growing diskspace for the recursive linear signature hashing index comes along with an increase of recursive hash tables which in turn leads to higher query evaluation costs.

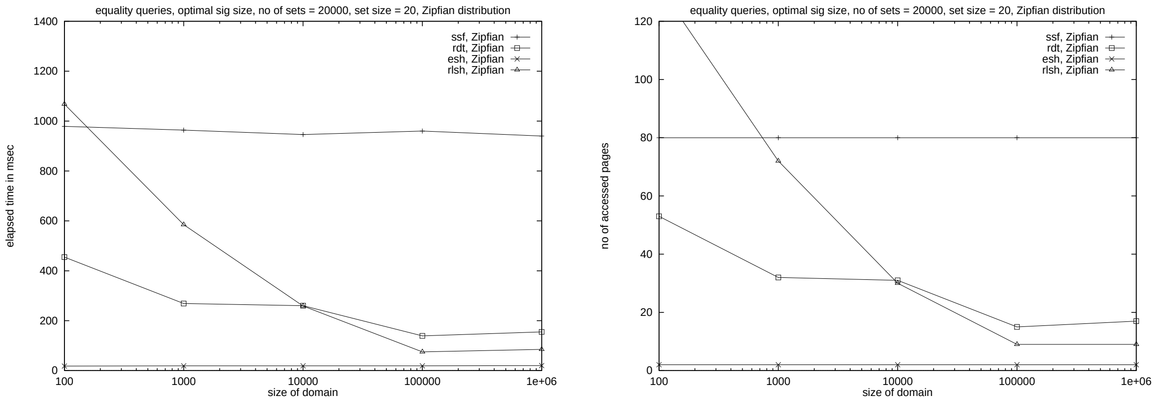

Figure 23: Query eval. costs (equality pred., Zipfian dist., varying domain size)

The skewed data apparently has no influence on the query evaluation costs for the sequential signature file and the extendible signature hashing index (see figure 23). The effect on the Russian doll tree and the recursive linear signature hashing index is amplified by the skewed data. Only few elements appear often in sets when generating sets containing skewed data, which has almost the same effect as decreasing the size of a domain for sets with uniformly distributed data.

6.2.2 Subset and superset predicates

This section covers the query evaluation costs for queries with subset and superset predicates. We deal with both predicates in one section because of the similarity. The results for subset and superset predicates do not differ noticeably. The only exception to this is the Russian doll tree, which does not support superset predicates. So there are no results for superset predicates for the tree index.

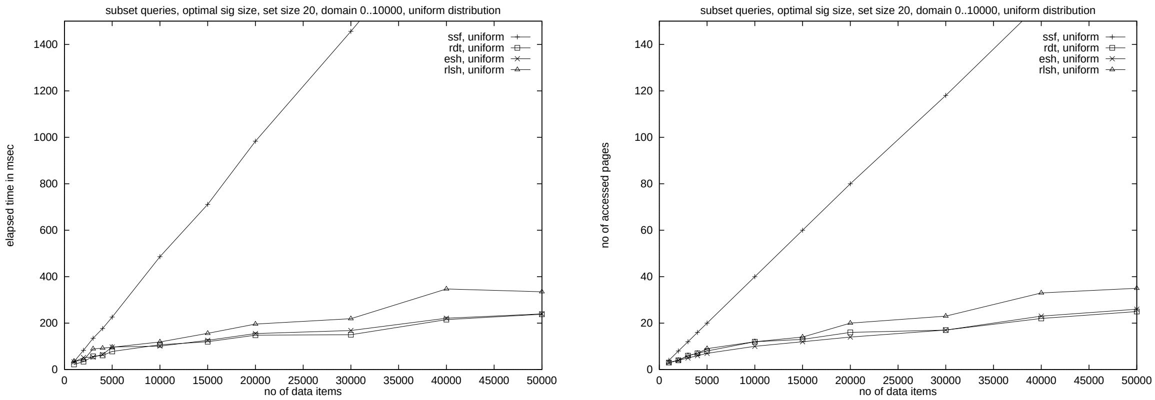

Figure 24: Query eval. costs (subset pred., uniform dist., varying database size)

Size of database: Figure 24 shows the query evaluation costs for uniformly distributed data. The left hand side presents the total running time, the right hand side the number of page accesses.

The query evaluation costs for the sequential signature file and the Russian doll tree for queries with subset/superset predicates do not differ from those for queries with equality predicates. This is not surprising, because very similar algorithms are used. The evaluation costs for the hashing index structures are not comparable to those for queries with equality predicates. Reaching a data item with just two page accesses cannot be guaranteed anymore, because several queries searching for each superset/subset of the query set are processed. The extendible signature hashing index can keep up with the Russian doll tree, whereas the recursive linear signature hashing index falls behind.

The influence of skewed data is depicted in figure 25 (total elapsed time on left hand side, number of page accesses on right hand side). The sequential signature file index has no problems coping with skewed data, there is no difference in the query evaluation costs compared to uniformly distributed data. The query evaluation costs of the extendible signature hashing index and the Russian doll tree increase by a small amount. The larger

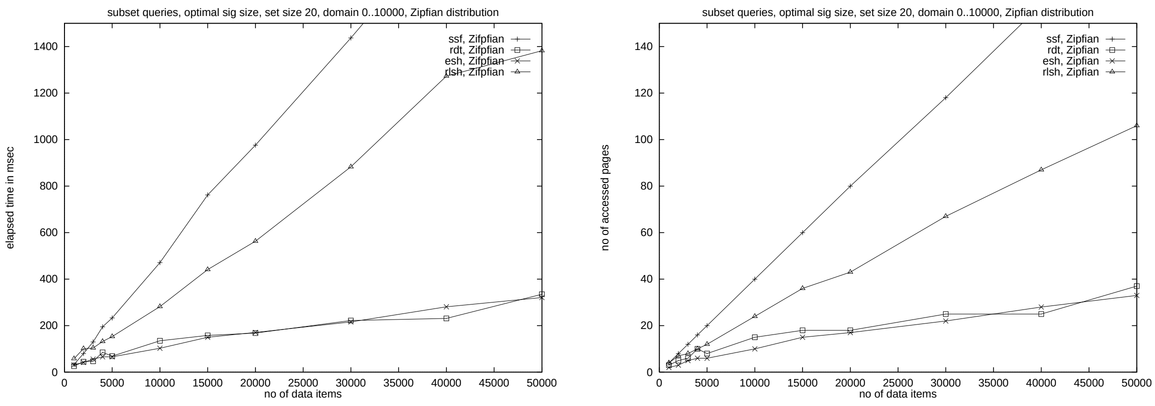

Figure 25: Query eval. costs (subset pred., Zipfian dist., varying database size)

directory in the extendible signature hashing index results in a larger depth and therefore in the generation of more subsets during the query evaluation in the case of skewed data. The Zipfian distributed data slows down the recursive linear signature hashing index even more. Each generated subquery has to go through all recursive hash tables, which leads to a terrible performance in this case.

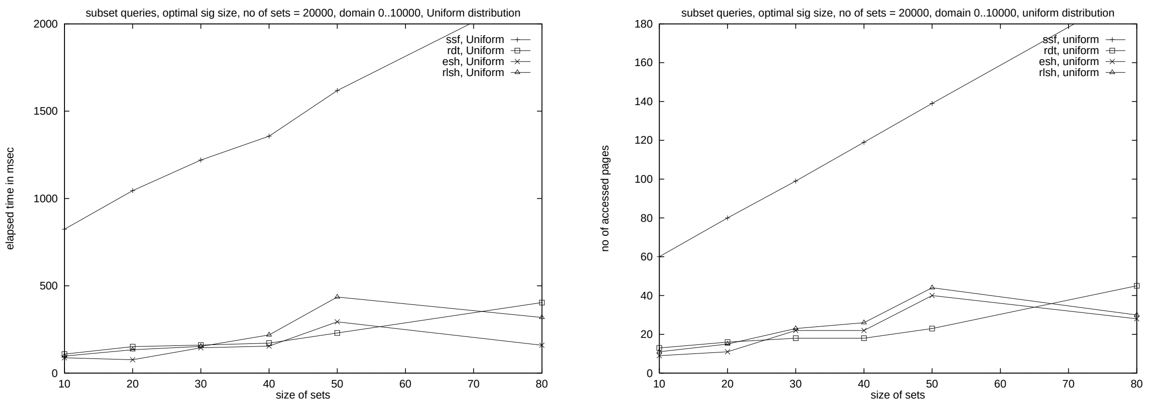

Figure 26: Query eval. costs (subset pred., uniform dist., varying set size)

Size of data items: Figure 26 shows the results for queries with subset predicates with uniformly distributed data when varying the size of the data items' sets. The query evaluation costs for a sequential signature file index grow proportionally to the index size, which depends on three parameters. One of these parameters is the size of a signature/reference tuple S_{tuple}. We increase the size of the signatures for larger sets to keep the false drop probability stable. The hashing index structures definitely do not have an advantage over the Russian doll tree for subset queries. The multiple subqueries that have to be started for each query become noticeable in the query evaluation costs. The distinctive rise in the costs when increasing the set size from 40 elements to 50 elements per set stems from significant changes taking place in the index structures. The height of the Russian doll tree increases by one layer, the depth (the number of significant bits) of the extendible signature increases from 7 to 8 bits and the recursive linear signature hashing index gains a recursive level.

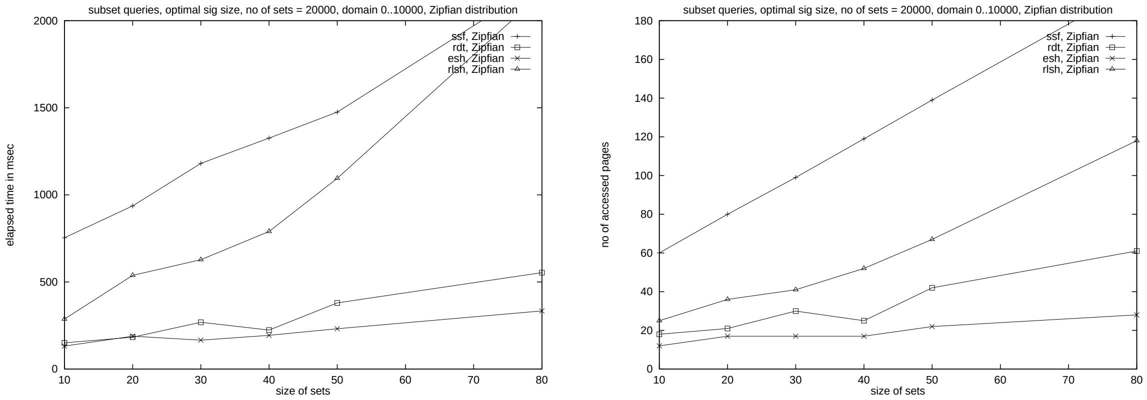

Figure 27: Query eval. costs (subset pred., Zipfian dist., varying set size)

The results for skewed data are presented in figure 27. The sequential signature file is not influenced by skewed data at all. The Russian doll tree loses some performance for larger sets, whereas the extendible signature hashing index is not influenced greatly. The recursive linear signature index has to contend with the problem of a rapidly growing number of recursive levels for skewed data. This manifests itself in heavy performance losses.

Size of domain: In figure 28 we have the results for the query evaluation costs for varying domain sizes. The effects on the index structures are similar to the effects observed for queries with equality predicates. We note no change in behavior for the sequential signature file and the extendible signature hashing index structure. The evaluation costs for the Russian doll tree and the recursive linear signature hashing index increase for small domains.

As already mentioned in section 6.2.1 on the costs for queries with equality predicates skewed data narrows the range from which elements are chosen during the generation of the data items' sets. The effects of small domains on the index structures are intensified by skewed data in this way.

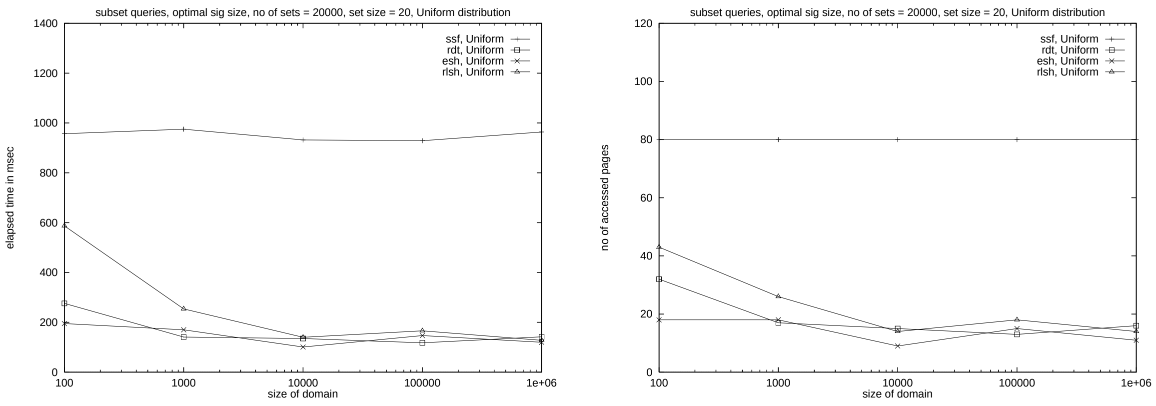

Figure 28: Query eval. costs (subset pred., uniform dist., varying domain size)

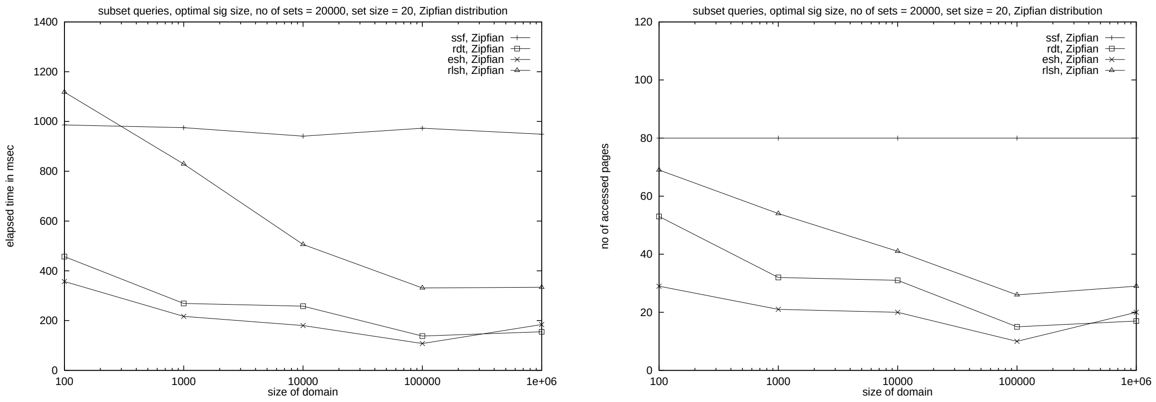

Figure 29: Query eval. costs (subset pred., Zipfian dist., varying domain size)

6.2.3 Intersection predicates

In this section we look at the query evaluation costs for queries with intersection predicates. Like the previous sections on equality and sub/superset predicates, we subdivide this section into paragraphs about the effects of varying the database size, the set size, and the domain size. We examine the sequential signature file and the Russian doll tree, as these are the only two index structures capable of supporting queries with intersection predicates.

Size of database: In figure 30 the query evaluation costs for queries with intersection predicates can be seen. The sequential signature file shows the usual performance, i.e. the query evaluation costs are exactly the same as for all other predicates. It is unusual, however, that the performance of the Russian doll tree is worse than that of the sequential

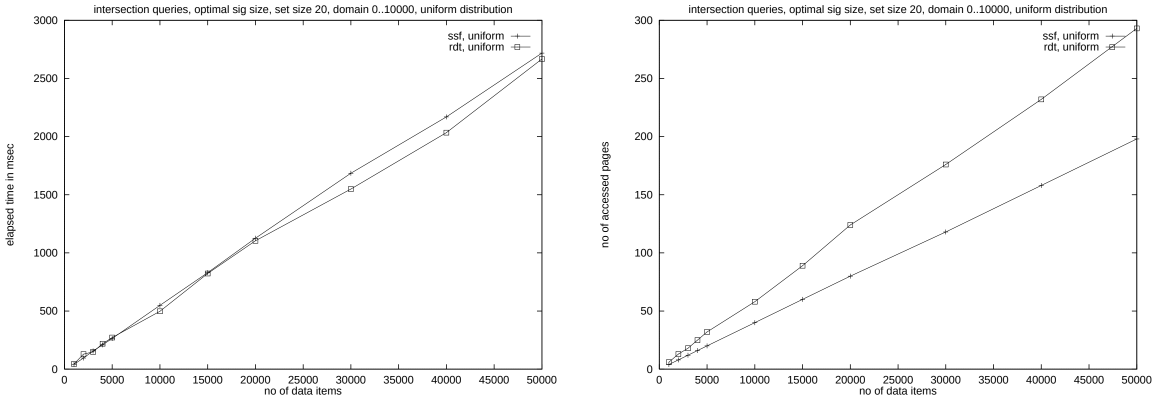

Figure 30: Query eval. costs (intersec. pred., uniform dist., varying database size)

signature file. The Russian doll tree has severe disadvantages when dealing with intersection predicates. For the processing of queries with equality and subset predicates the Russian doll tree accesses at most one sixth of all pages in the tree. When processing a query with an intersection predicate this changes drastically, as virtually the entire tree is traversed (compare to section 6.1 on diskspace). This is not entirely the fault of the Russian doll tree, though. When using signatures for evaluating intersection predicates, this leads to a much higher false drop probability than for equality of subset predicates. The query evaluation costs cannot be improved by increasing the size of the signatures to lower the false drop probability. The relative performance of the Russian doll tree becomes better for larger signatures when compared to the sequential signature file. The absolute performance, however, becomes worse, because of the increased size of the index structure (see figure 32).

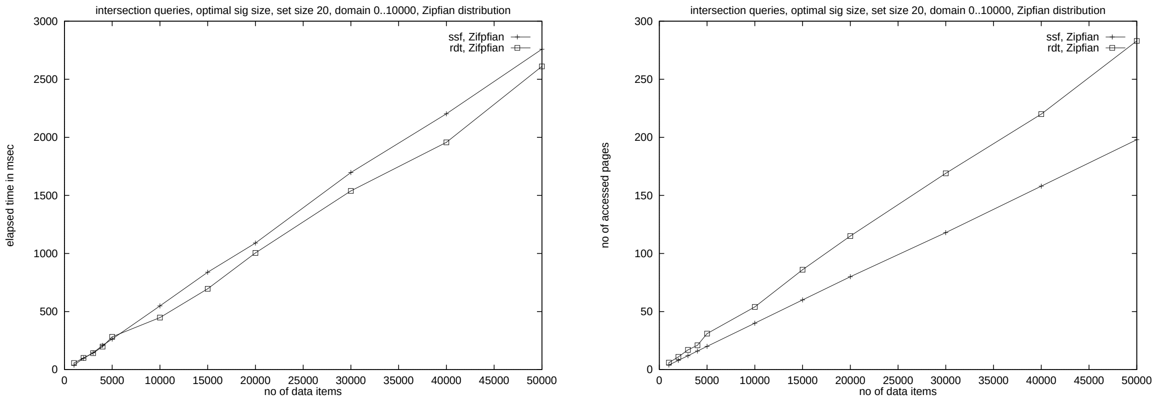

Figure 31: Query eval. costs (intersec. pred., Zipfian dist., varying database size)

Figure 31 depicts the measured results for skewed data. No changes can be reported for the sequential signature file. The performance of the Russian doll tree for skewed data is superior to that for uniformly distributed data, although it is not significantly better than the performance of the sequential signature file.

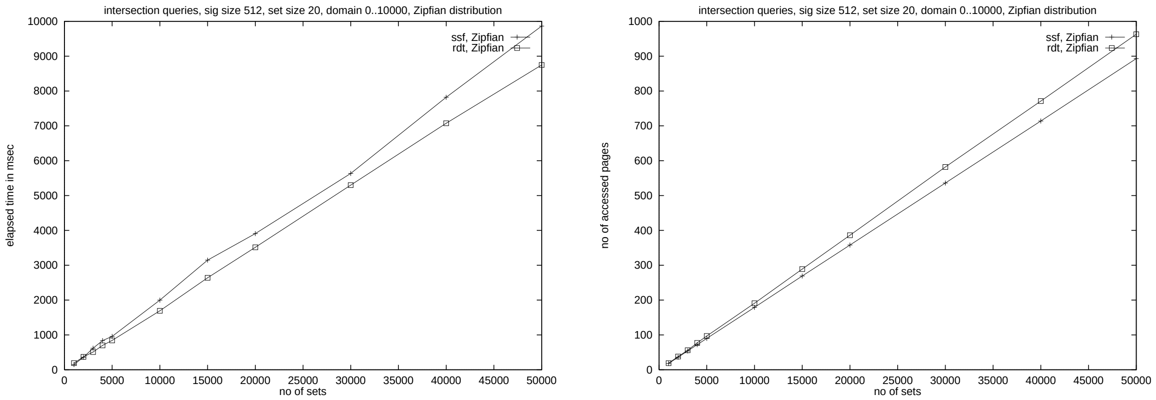

Figure 32: Query eval. costs (intersec. pred., Zipfian dist., varying database size, large signatures)

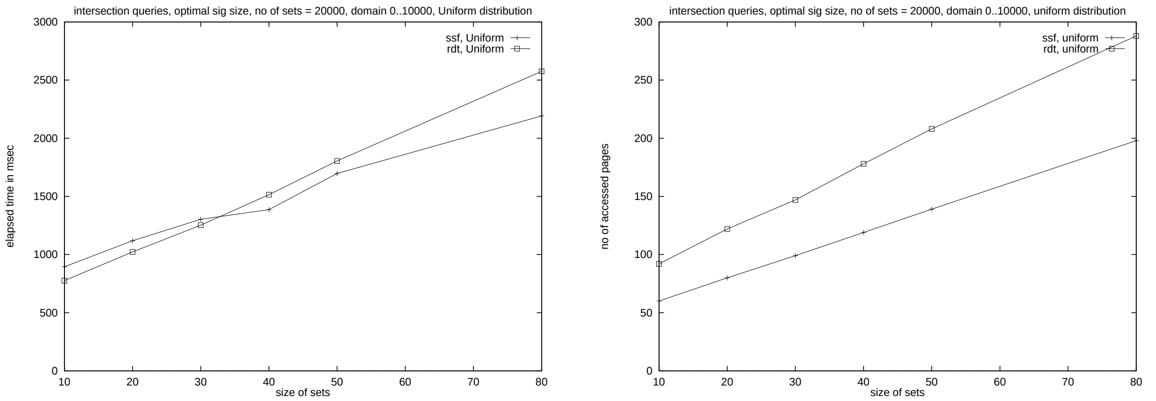

Figure 33: Query eval. costs (intersec. pred., uniform dist., varying set size)

Size of data items: Figure 33 shows the effects of varying set sizes (and varying signature sizes) on the index structures. As for equality and sub/superset predicates the query evaluation costs for the sequential signature file rise linearly with the increased set size. As already mentioned the Russian doll tree accesses almost all its pages when evaluating a query with an intersection predicate. Increasing the size of the sets and therefore the size of the signatures enlarges the Russian doll tree, which directly influences the query evaluation costs.

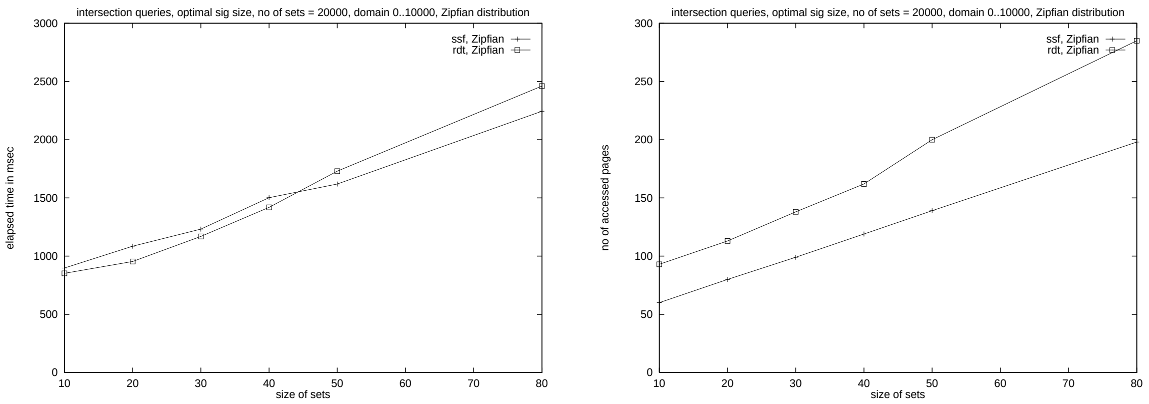

Figure 34: Query eval. costs (intersec. pred., Zipfian dist., varying set size)

We illustrated the results for skewed data in figurequeryintersectzipfset. This time both index structures, the sequential signature file and the Russian doll tree, are not influenced by the skewed data.

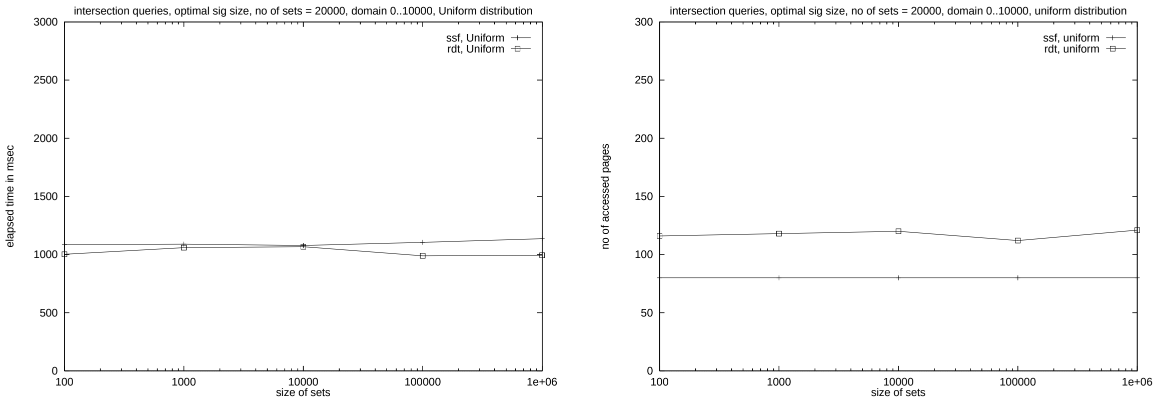

Figure 35: Query eval. costs (intersec. pred., uniform dist., varying domain size)

Size of domain: In figure 35 and 36 the remaining set of measurements for the query evaluation costs for queries with intersection predicates, varying the domain sizes, are shown. The interpretation of these curves is straightforward. Neither the variation of the domain size nor the skewing of the data has any effect on the performance of the index structures. All query evaluation costs are constant for this case.

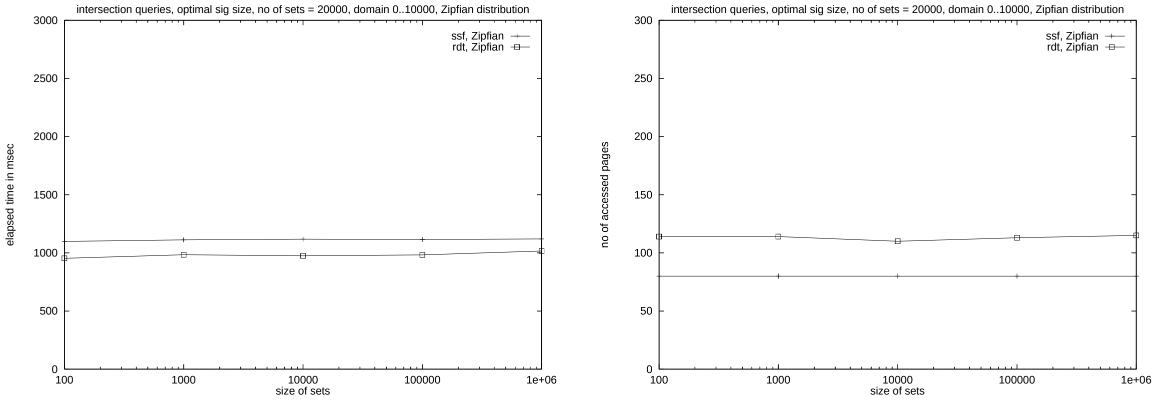

Figure 36: Query eval. costs (intersec. pred., Zipfian dist., varying domain size)

6.2.4 Summary

In this section we give a short summary on the performance of the different index structures. For the support of queries with equality predicates there is one index structure which is superior to all others, the extendible signature hashing index. Although the recursive linear signature hashing index performs as well as the extendible signature hashing index for uniformly distributed data, the recursive linear signature hashing index suffers significant performance losses for skewed data. The Russian doll tree and the sequential signature file, which was run as a reference, are no match against the guaranteed two page accesses of the extendible signature hashing index. For the support of queries with sub/superset predicates there is no such clear choice. The extendible signature hashing index slows down when processing queries with sub/superset predicates, because each query is broken down into several subqueries with equality predicates. The Russian doll tree evaluates queries with subset predicates very similar to queries with equality predicates and is on one level with the extendible signature hashing index. The recursive linear signature hashing index is out of the race (together with the sequential signature file), because of its poor performance for skewed data. Supporting queries with intersection predicates seems to be the hardest case. The hashing index structures are not even capable of supporting this kind of query. The Russian doll tree is not better than the sequential signature file, simple scanning is often faster than a more elaborate search.

We conclude that the overall performance of the extendible signature hashing index for query evaluation is clearly the best among all tested index structures. None of the index structures supports queries with intersection predicates satisfactorily, however.

6.3 Update costs

In this section we take a look at the dynamic behavior of the index structures, namely insertion and deletion operations and their costs. We present the measured results of all benchmarks for the different index structures comparing them to each other.

6.3.1 Insertion costs

First of all we inspect the insertion costs taking into consideration varying database sizes, varying set sizes, and varying domain sizes. We look at each of the different parameters in turn.

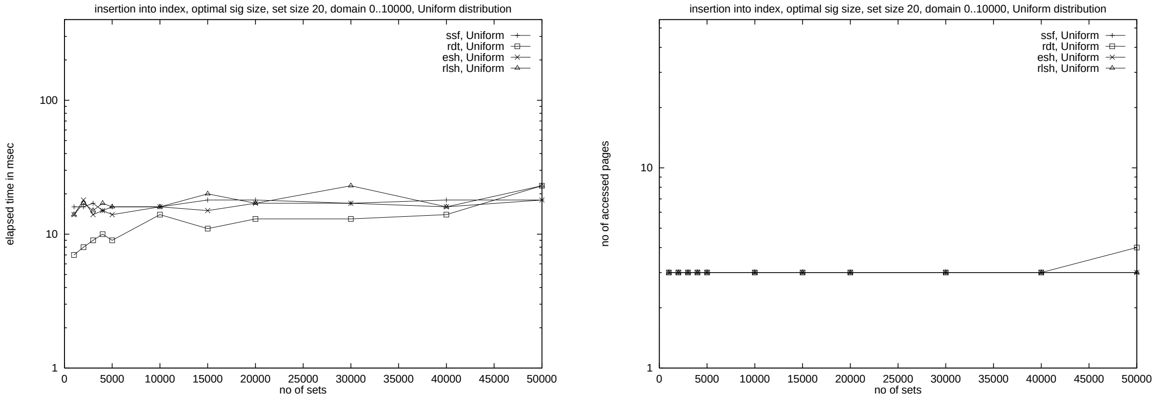

Figure 37: Insertion costs (Uniform distribution, varying database size)

Size of database We start with the variation of the database size. In figure 37 the results for uniformly distributed data are shown. All index structures have very similar insertion costs, except for the Russian doll tree, whose costs increase with the growing height of the tree.

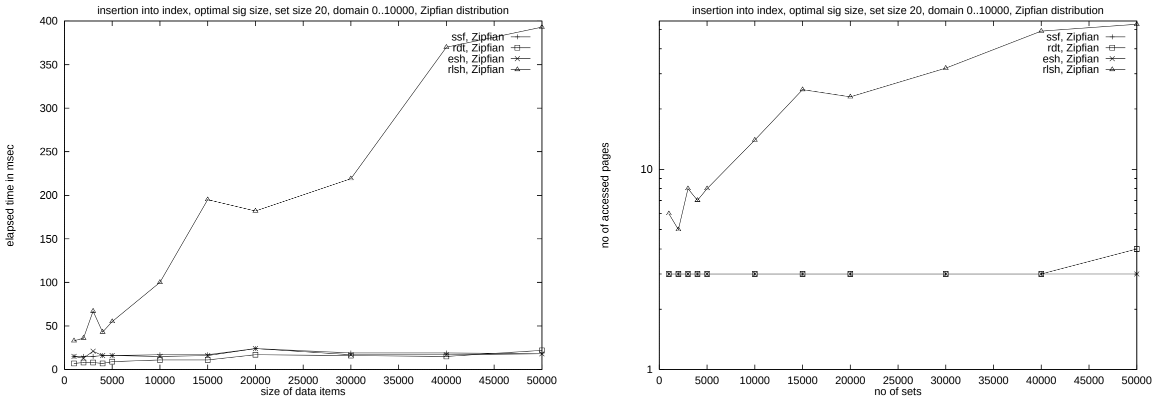

Figure 38: Insertion costs (Zipfian distribution, varying database size)

Figure 38 depicts the results for skewed data. The sequential signature file, the Russian doll tree, and the extendible signature hashing index structures' insertion costs are not influenced by skewed data. As in all previous benchmarks, however, the number of recursive hash tables increases rapidly for the recursive linear signature hashing index. This leads to very high insertion costs as many recursive hash tables have to be traversed.

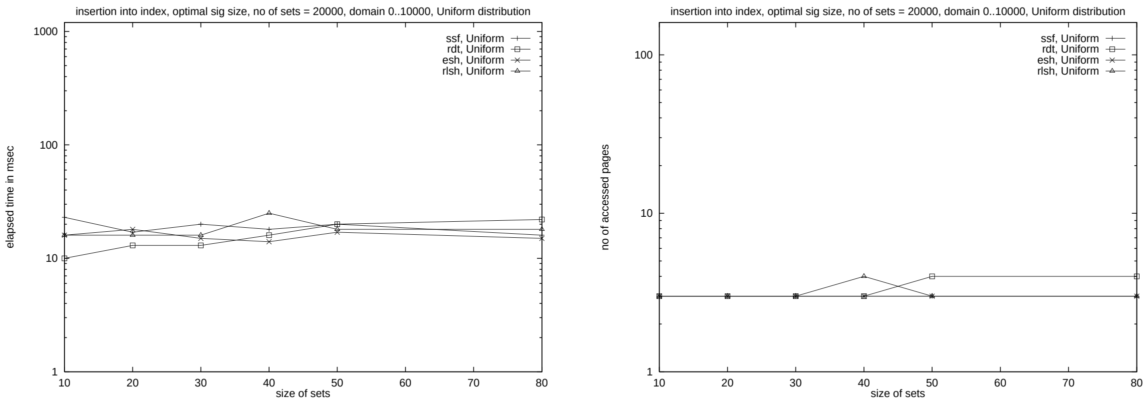

Figure 39: Insertion costs (Uniform distribution, varying set size)

Size of data items The insertion costs for varying set sizes and uniformly distributed data are shown in figure 39. Increasing the set size leads to larger signature (to hold the false drop probability stable), which in turn results in larger index structures. Except for the Russian doll tree, the insertion costs are independent of the size of the sets (or the size of the database, respectively). The height of the Russian doll tree increases with a growing index. This can be clearly seen in figure 39, when changing the size of the sets from 40 elements per set to 50 elements per set.

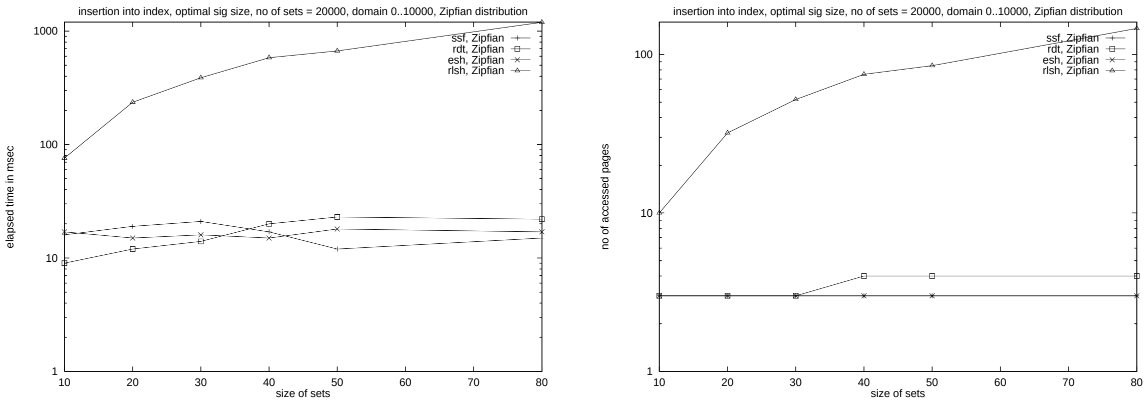

Figure 40: Insertion costs (Zipfian distribution, varying set size)

For skewed data (the result of the benchmarks are depicted in figure 40) the performance of the sequential signature file and the extendible signature hashing index do not change at all. The Russian doll tree's insertion costs start to rise earlier, when compared to uniformly distributed data, as the height of the tree already increases for a set size of 40 elements. The skewed data affects the performance of the recursive linear signature hashing index negatively, because of the growing number of recursive hash tables.

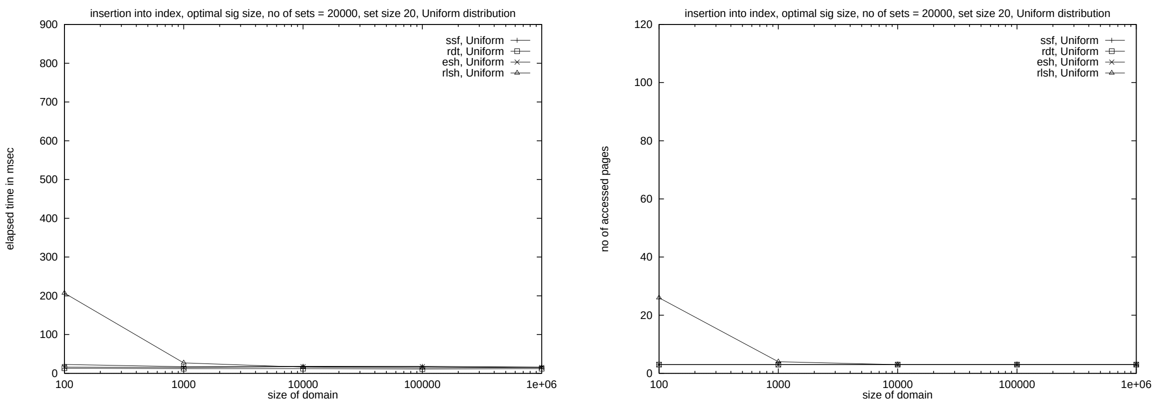

Figure 41: Insertion costs (Uniform distribution, varying domain size)

Size of domain The last parameter we look at is the variation of the domain size. Figure 41 shows the costs for uniformly distributed data. As already mentioned in section 6.1 on the comparison of the diskspace small domains lower the variety of the sets, because the elements are picked from a smaller pool. Hence, small domains have a similar effect on the costs as skewed data with larger domains. The recursive linear signature hashing index is very susceptible to this kind of effect and therefore loses some performance for small domains. The other index structures are unaffected by the size of the domain. Although the extendible signature hashing index' size increases for small domains (see section 6.1), each bucket can still be reached with two page accesses.

As already observed in previously taken measurements with varying domain sizes the skewed data amplifies the effects noticed for uniformly distributed data. This can be clearly seen in figure 42 showing the results for the skewed data. The performance losses of the recursive linear signature hashing index are more severe for skewed data.

Summary The insertion costs for the sequential signature file and the extendible signature hashing index are the lowest of all index structures. More importantly these costs are not influenced by any of the examined parameters. The costs for the Russian doll tree are neither affected by skewed data nor by the size of the domain. However, the larger the index and therefore the larger the height of the tree becomes, the costlier the insertions are. For uniformly distributed data the recursive linear signature hashing index is as efficient as the sequential signature file and the extendible signature hashing index,

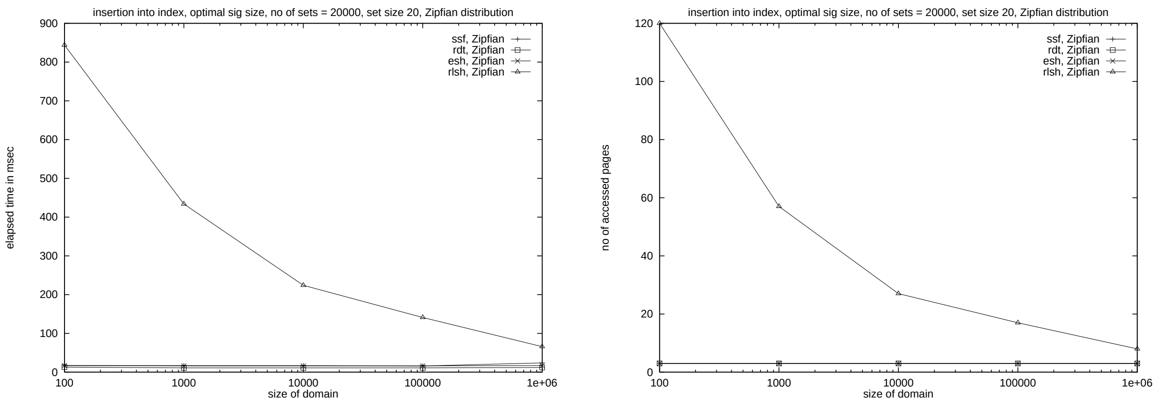

Figure 42: Insertion costs (Zipfian distribution, varying domain size)

for skewed data this changes drastically, due to the increasing number of recursive hash tables. In terms of the insertions costs the sequential signature file and the extendible signature hashing index are clearly the index structures of choice.

6.3.2 Deletion costs

The second part of the update costs are the costs for the deletion of a data item form an index. Again we look at the parameters database size, set size, and domain size. As we will see the deletion costs are similar to the query evaluation costs for queries with equality predicates. This is not surprising, because before a data item can be removed from an index it has to be found.

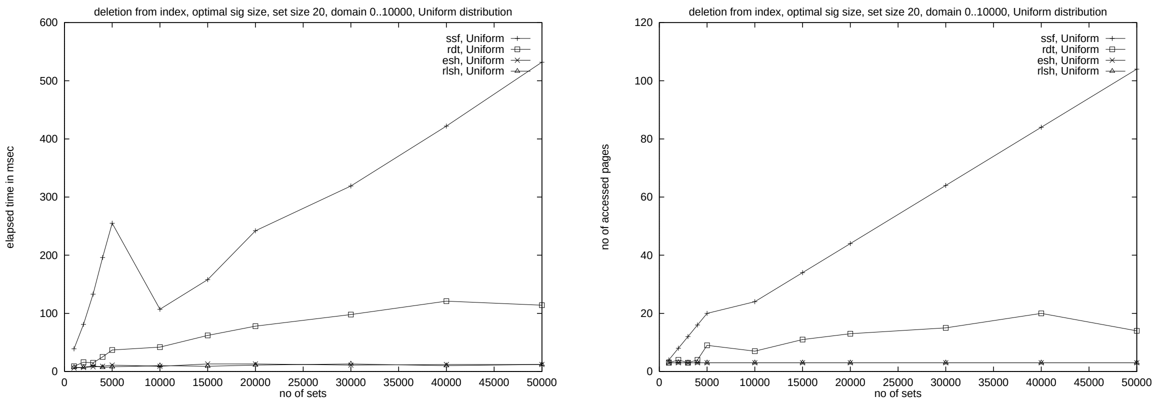

Figure 43: Deletion costs (Uniform distribution, varying database size)

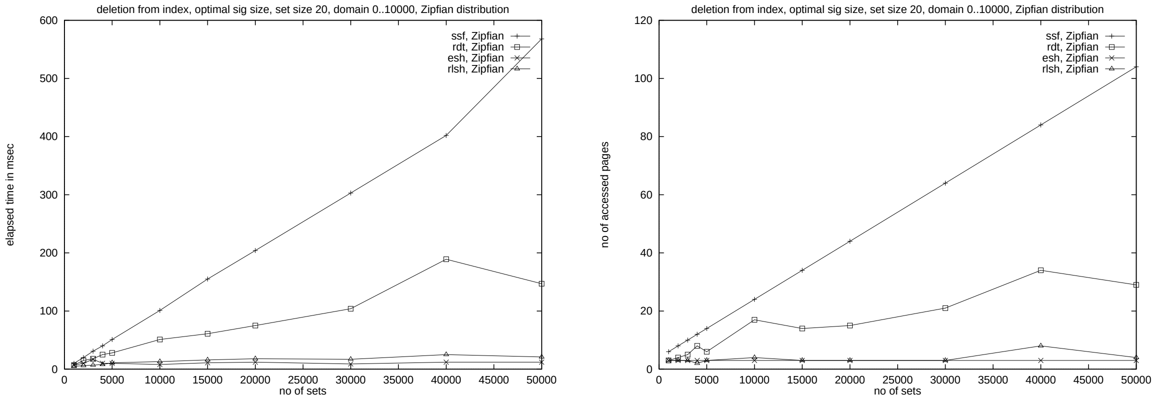

Figure 44: Deletion costs (Zipfian distribution, varying database size)

Size of database In this paragraph we investigate the impact of database size on a deletion operation. Figure 43 shows the results for uniformly distributed data. The costs for deleting a data item in a sequential signature file are about half as large as the costs for a search with an equality predicate during a query evaluation (for a comparison see figure 18). The reason for this is that we only investigate successful deletions, i.e. the data item designated for deletion always exists in the index, after finding and deleting the item we do not have to search for further items. For the sequential signature file this means, that on the average half of the signature file has to be traversed. For the Russian doll tree the difference between the deletion costs and the query evaluation costs are not quite as large. Although the number of visited pages is roughly halved (because of depth-first search instead of breadth-first search, where all inner nodes have to be traversed), the deletion costs are larger than half of the query evaluation costs. After deleting a data item in a leaf node we need to adjust the signature entries in all parent nodes of the leaf node. The hashing index structures do not access unnecessary pages during query evaluation when searching for a data item. Therefore searching cannot be made faster for deletion. The hashing index structures have deletion costs of two page accesses, which are the lowest of all index structures.

Figure 44 shows the results for skewed data. The sequential signature file has the same behavior as for uniformly distributed data, because it is not affected by skewed data. The performance of the Russian doll tree worsened, but deletion is still faster than searching during a query evaluation. Although the size of the extendible hashing increases for skewed data, this has no effect on the costs for searching. Surprisingly the recursive linear signature hashing index does not show the usual poor performance for skewed data. Despite consisting of many recursive hash tables, the largest part of the data items is stored in the top most tables. During deletion the probability to hit the data item to be deleted early is quite high. After finding the item the search process is terminated, avoiding the costly traversal of further recursive hash tables. During query evaluation, on the other hand, each and every hash table is searched.

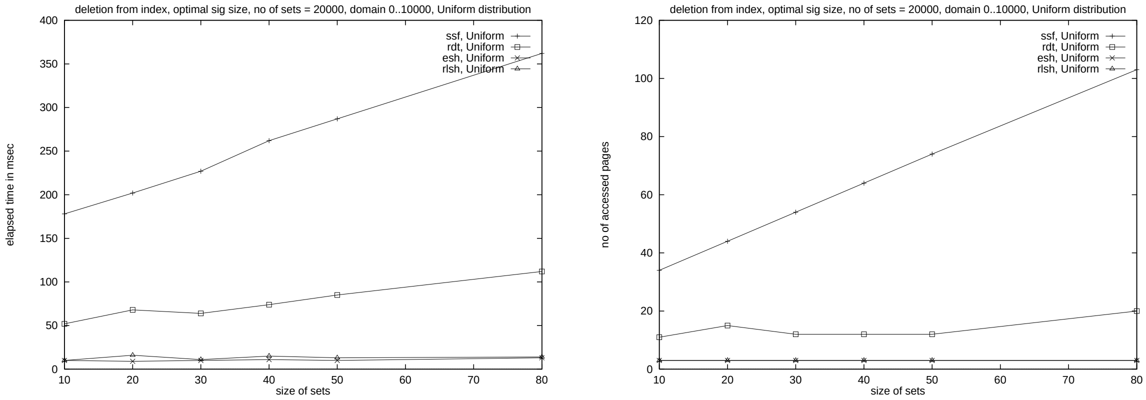

Figure 45: Deletion costs (Uniform distribution, varying set size)

Size of data items In figure 45 the measured deletion costs for uniformly distributed data varying the set (and signature) size are depicted. For the sequential signature file the deletion costs are approximately half of the query evaluation costs for queries with an equality predicate (see figure 20), because on the average we go through half of the signature file. The deletion costs for a Russian doll tree are considerably lower than the corresponding query evaluation costs in figure 20. We notice that the deletion costs do not increase suddenly when the height of the tree increases as it is the case for the query evaluation costs. Let us explain the reasons for this. The signatures in the inner nodes of the tree are generated by superimposing signatures of lower levels. Increasing the height of a Russian doll tree leads to denser signatures (i.e. signatures with a high percentage of set bits), which in turn deteriorates the false drop probability. When evaluating a query the lower fan-out of the nodes in the tree does not compensate for the higher false drop probability. When deleting a data item, we stop the search as soon as we have found the data item. In this case we profit fully from the lower fan-out, but do not suffer the disadvantage of the higher false drop probability fully, because we skip approximately half of the eligible nodes during a deletion. The search performance of the hashing index structures is not influenced by the size of the index , therefore the size of the sets (and signatures) has no impact on the deletion costs.

Figure 46 shows the result for skewed data. As in previous benchmarks the sequential signature file and the extendible hashing index are not influenced by skewed data. The Russian doll tree suffers some performance losses. The performance of the recursive linear signature hashing index, although lower than that for uniformly distributed data, is much better than its query evaluation performance for skewed data. This is due to the fact that we can abort the search as soon as we have found the item to be deleted.

Size of domain In figure 47 the results for varying domain sizes and uniformly distributed data are displayed. The deletion costs are dominated by the costs for searching for the data item to be deleted. Similar to the query evaluation costs (see figure 22) of the sequential signature file and the extendible signature hashing index, the deletion costs

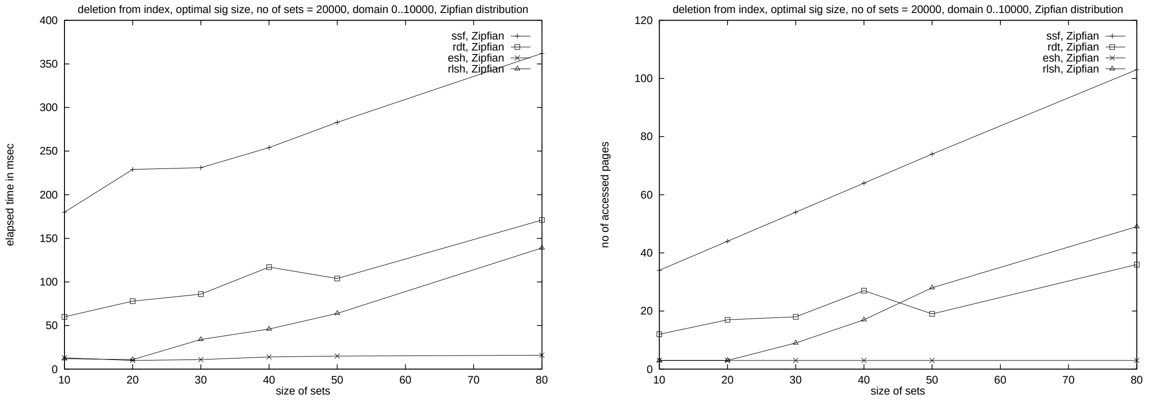

Figure 46: Deletion costs (Zipfian distribution, varying set size)

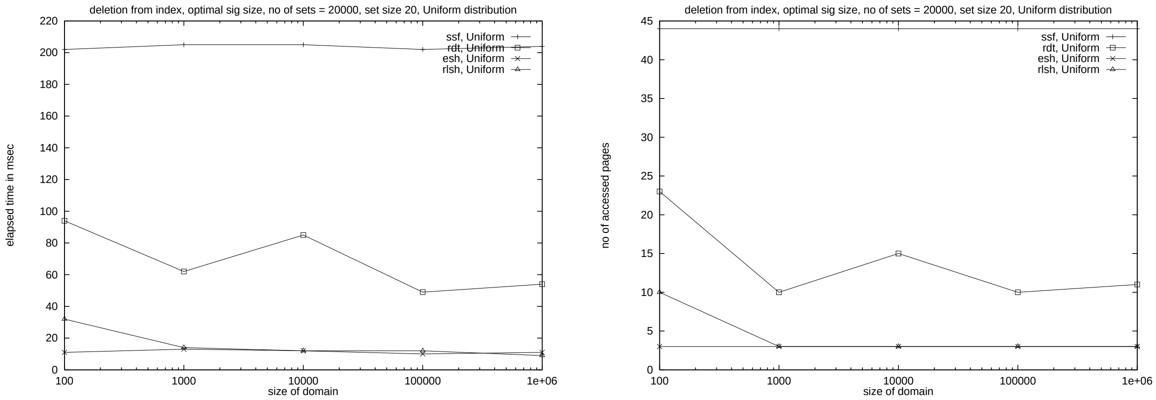

Figure 47: Deletion costs (Uniform distribution, varying domain size)

of these index structures are not influenced by the size of the domain. The Russian doll tree and the recursive linear signature hashing index structures lose some performance for small domains. As already mentioned in section 6.2.1 about the query evaluation costs the recursive linear signature hashing index is very susceptible to a large number of identical hash values. Small domains lead to many similar (or even identical) sets, several of which are hashed onto identical signatures. In order to handle the occurring overflow many recursive hash tables are allocated. Fortunately when deleting a data item not all recursive hash tables need to be searched, limiting the performance loss in this case.

The results for skewed data are shown in figure 48. The deletion costs for the sequential signature file and the extendible signature hashing index are not influenced by skewed data. The costs for the Russian doll tree are slightly higher for skewed data than for uniformly distributed data. Skewed data amplifies the effect of small domains, limiting

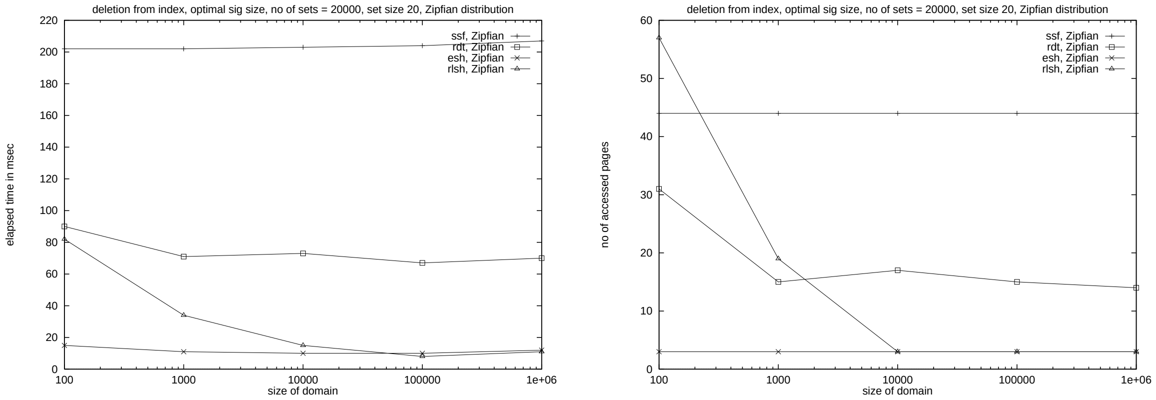

Figure 48: Deletion costs (Zipfian distribution, varying domain size)

the range from which elements are chosen even further. Therefore the performance of the recursive linear signature hashing index is worse for skewed data than for uniformly distributed data.

Summary Similar to the query evaluation costs for queries with equality predicates the extendible signature hashing index dominates the field. The other index structures cannot compete with the two page accesses to reach the data item and one page access to update the index. For uniformly distributed data the recursive linear signature hashing index is on equal grounds with the extendible signature hashing index, but for skewed data the extendible signature hashing index is clearly superior.

6.4 Creation costs

The costs for constructing an index should not be neglected, because an index does not come into existence spontaneously. For large databases the creation of several index structures can take a considerable amount of time. When measuring the speed of query evaluations, insertions and deletions, we cleared all buffers and caches between two queries (or two insertion or deletion operations, respectively). When measuring the creation costs we cleared the buffers and caches before starting with the creation, i.e. during the creation of the index we allowed the buffering and caching of pages that have already been accessed. Like in all previous benchmarks we varied the parameters database size, data item size, and domain size.

6.4.1 Size of database

Figure 49 shows the results for uniformly distributed data when varying the size of the database. The sequential signature file has the lowest costs for creation (in terms of page accesses and total elapsed time) as it uses the most simple data structure. The number of page accesses for the Russian doll tree is very similar to those for the sequential signature

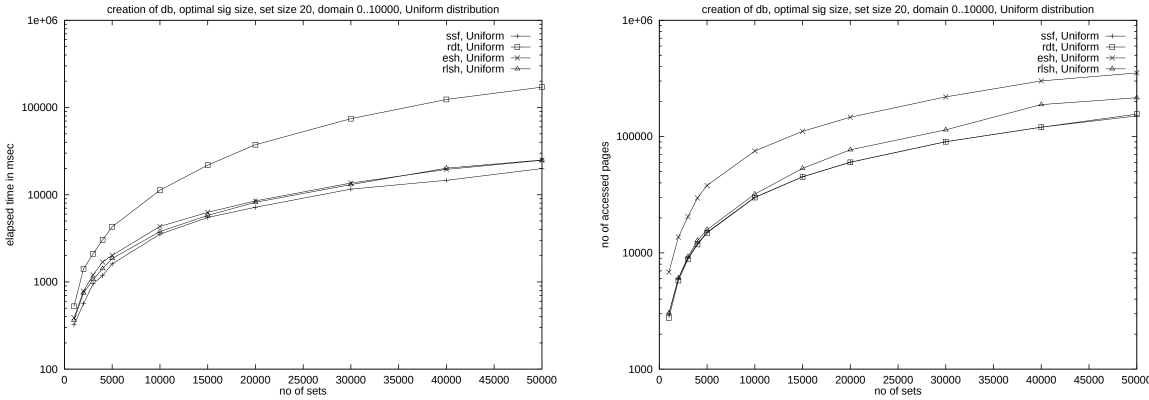

Figure 49: Creation costs (Uniform distribution, varying database size)

file, but in terms of total time the Russian doll tree takes the longest time to construct. This has to do with the relatively time-consuming node splitting algorithm which has to be executed when an overflow occurs. The total number of pages accessed by the hashing index structures during creation is relatively high. The most costly part in this case is handling the overflows, during which several new pages may have to be allocated and initialized (which is mainly composed of I/O costs and not CPU-costs).

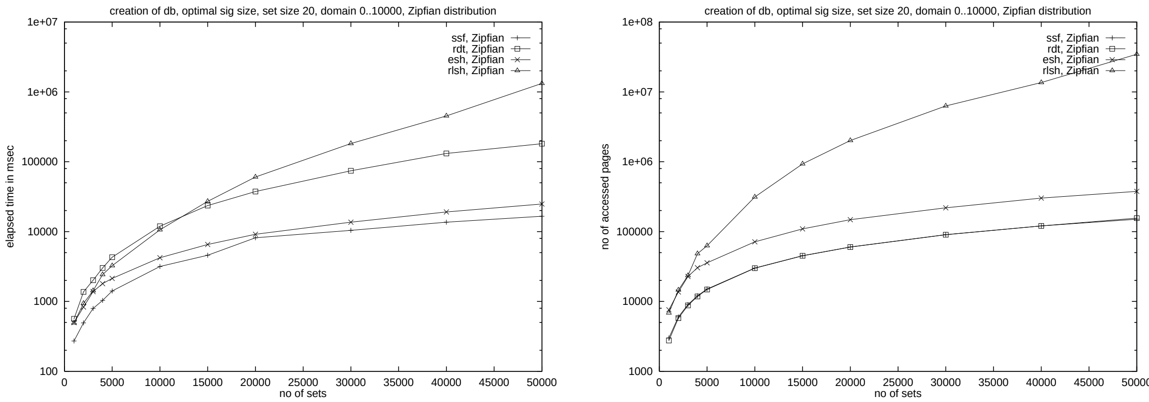

Figure 50: Creation costs (Zipfian distribution, varying database size)

Figure 50 depicts the creation costs for skewed data. There is no apparent change in the performance of the sequential signature file, Russian doll tree, and extendible signature hashing index when compared to the performance for uniformly distributed data. The performance of the recursive linear signature hashing index, however, goes down considerably. The reason for this is the large number of recursive hash tables that have to be allocated and searched through.

6.4.2 Size of data items

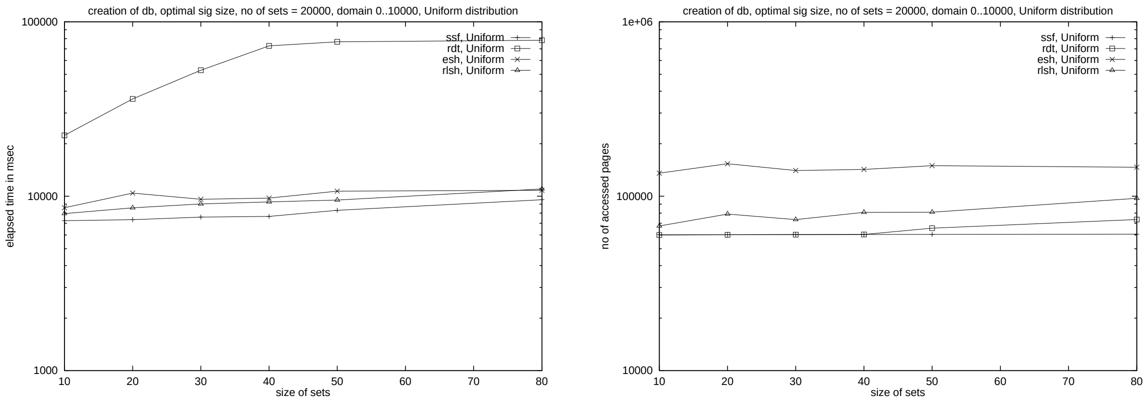

Figure 51: Creation costs (Uniform distribution, varying set size)

The next parameter we vary is the size of the data items (and consequently the size of the signatures to keep the false drop probability stable). Figure 51 displays the results of the benchmarks for uniformly distributed data. For the sequential signature file and the hashing index structures the costs do not increase significantly for larger data items. We create index structures containing a constant number of data items (20,000) and increased the size of the data items which leads to more (and earlier) overflows in the index structures. When looking at figure 51 this effect cannot be seen, i.e. that the insertion costs for each inserted data item clearly dominate the costs for expanding the index when an overflow occurs. The only exception to this is the Russian doll tree, which has to increase the use of its time-consuming split algorithm until the tree reaches a height of three levels.

Figure 52 for skewed data shows a similar picture as figure 51 for the sequential signature file, the Russian doll tree, and the extendible signature hashing index. Due to the large number of recursive hash tables that have to be allocated and searched through, the performance of the recursive linear signature hashing index deteriorates.

6.4.3 Size of domain

The last parameter we look at is the variation of the domain size. In figure 53 the results for uniformly distributed data can be seen. The size of the domain has no influence on the performance of the sequential signature file, the Russian doll tree, and the extendible signature hashing index. When comparing the number of page accesses with the elapsed time, it can be noticed that the Russian doll tree has the highest CPU-costs of all index structures, whereas the CPU-costs of the extendible signature hashing index are low. Like skewed data small domains lead to a deep recursive structure for the recursive linear signature hashing index, which has a negative impact on the creation costs.

Figure 54 displays the results for skewed data. The sequential signature file, the Russian doll tree, and the extendible signature hashing index are neither affected by the

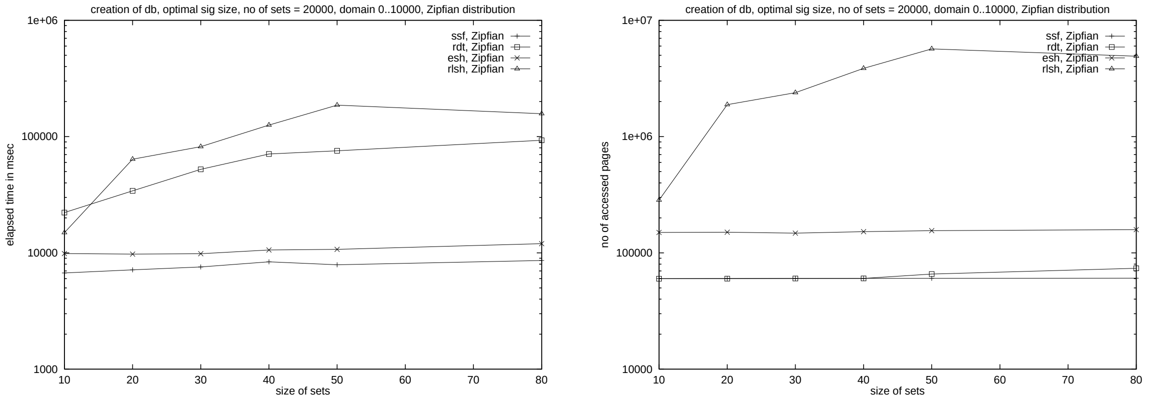

Figure 52: Creation costs (Zipfian distribution, varying set size)

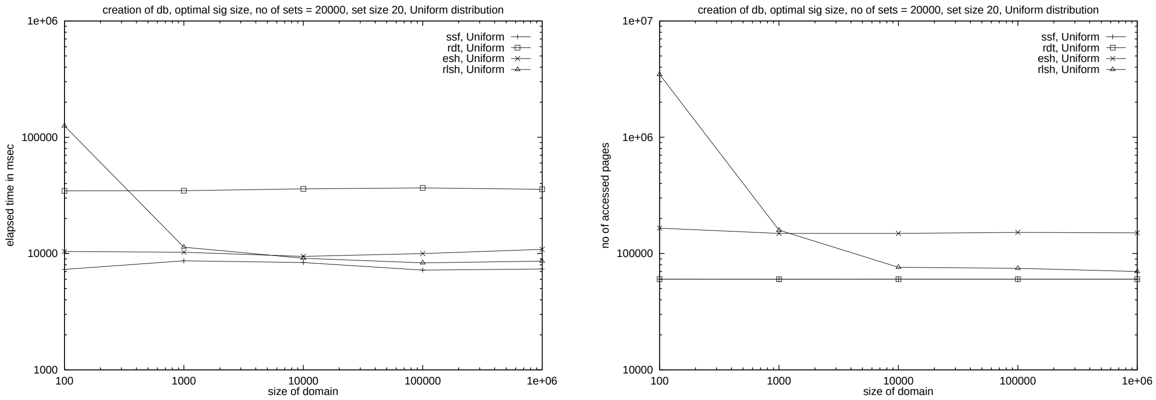

Figure 53: Creation costs (Uniform distribution, varying domain size)

size of the domain, nor are they affected by the skewed data. For the recursive linear signature hashing index the skewed data adds to the problems it already hash with small domains.

6.4.4 Summary

It is not surprising that the sequential signature file has the lowest creation costs, as it hash the most simple structure. Although the extendible signature hashing needs quite a few page accesses when it is created, this is compensated in part by low CPU-costs, resulting in the second lowest costs. The recursive linear signature hashing index would have been a contender, if it were not for its bad performance for skewed data. The Russian doll tree has low I/O costs for the creation of an index, the CPU-costs, however, are the highest of all index structures, even though we used a split algorithm with linear

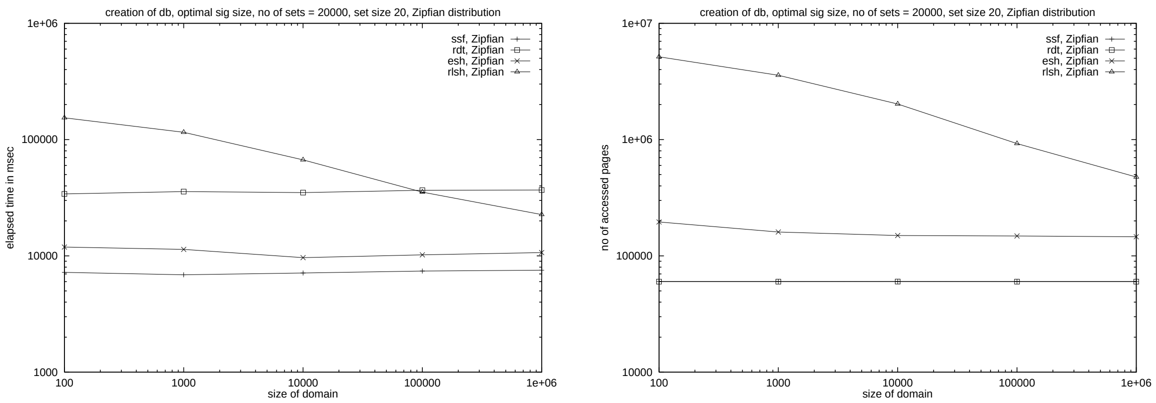

Figure 54: Creation costs (Zipfian distribution, varying domain size)

complexity instead of quadratic complexity.

6.5 Summary of the simulation results

Let us now sum up the results of the simulation as we look at each index structure in turn. The sequential signature file was mainly used as a reference for the comparison. Although the performance of the sequential signature file is worse than the performance of the other index structures in almost all tested areas, the insertion costs (and consequently the creation costs) are among the lowest of all index structures. This does not come as a surprise as the sequential signature file possesses a very simple structure, allowing even slightly faster insertions than the hashing index structures. Because of the simple structure, the sequential signature file also has the lowest overhead of all index structures. Until other index structures are available it is also appropriate when none of the other index structures can be used (as it is the case for queries with intersection predicates.

The Russian doll tree has a moderately high overhead when compared to the other index structures. It does not quite compensate for this overhead, however. For the evaluation of queries with equality predicates the hashing index structures are clearly better. For queries with subset predicates the tree is not significantly faster than the hashing index structures. The update operations are among the slowest of all index structures. An advantage of the Russian doll tree is that it is not influenced negatively by skewed data.

The extendible signature hashing index has shown excellent performance for uniformly distributed data as well as for skewed data. The fear that the directory has to be expanded almost endlessly for skewed data did not come true. Although the directory is two to four times larger for skewed data than for uniformly distributed data, the largest part of the allocated diskspace is used for the buckets, containing the signature and links to the data items, and not in the directory.

The performance of the recursive linear signature hashing index, although for uniformly distributed data on equal grounds with the extendible signature hashing index, suffers severe losses for skewed data. The skewed data leads to very unbalanced hash tables resulting in a large number of recursive hash tables. So the recursive linear signature hashing index can be recommended without reservations only for uniformly distributed data.

The extendible signature hashing index has been proven the most efficient index structure for set-valued attributes by our simulation. We do not want to withhold the one weakness of the extendible signature hashing index, though. It cannot be applied when evaluating queries with intersection predicates. We have not found an index yet, however, that is capable of supporting queries with intersection predicates efficiently. This is an area of future research.

7 Validating the cost models

In this section we compare the theoretical cost models with actually measured costs. We do this by dividing the actually measured costs by the expected theoretical costs and plotting the resulting factor. Please note that we use logarithmic scales on the y-axis in the following figures.

7.1 Diskspace

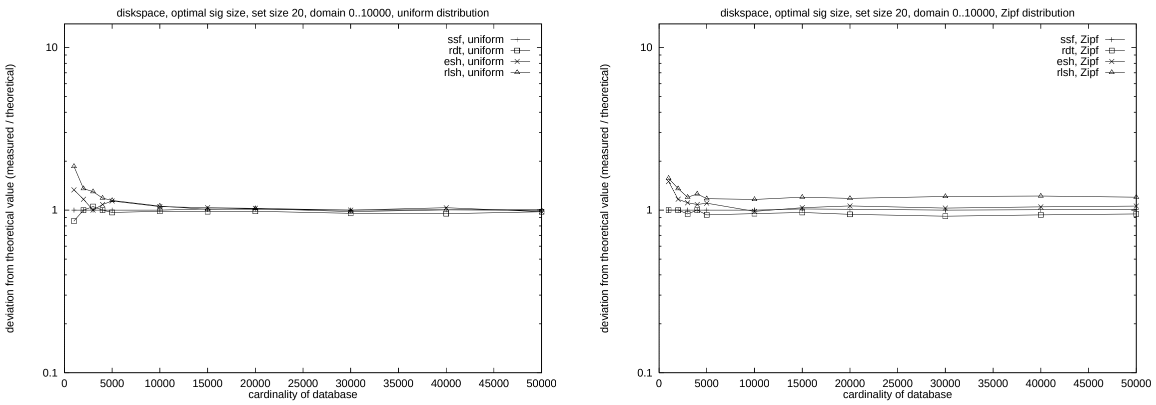

Figure 55: Theoretically required diskspace vs. actually allocated diskspace

In this section we compare the theoretical cost models for the required diskspace, used in section 4.2 to compare the size of the index structures, with actually measured values. The formulas used to calculate the expected allocated diskspace were (55), (59), (61), and (70). Figure 55 shows on the left hand side the comparison for uniformly distributed data and on the right hand side the comparison for skewed data. There is almost no deviation between the theoretically predicted diskspace requirements and the actually allocated diskspace. The formulas for the diskspace requirement do not take into account skewed data. As the skew does not affect the sequential signature file and the Russian doll tree, the cost formulas are fully valid for these two cases. For the extendible signature hashing index and recursive linear signature hashing index the actually required diskspace is only slightly larger. The recursive linear signature hashing index needs up to 36% more diskspace than expected, which is a tolerable deviation from the cost model. In table 11 the sample standard deviation for uniformly distributed and skewed data is given.

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	0.84 pages	0.84 pages
Russian doll tree	5.19 pages	8.99 pages
extendible signature hashing	3.94 pages	7.24 pages
recursive linear signature hashing	4.37 pages	32.75 pages

Table 11: Sample standard deviation for diskspace

Recapitulating we can say that for diskspace the cost formulas are fully valid, although there are small deviations for the hashing index structures.

7.2 Query evaluation costs

In this section we compare the theoretical query evaluation costs with the measured query evaluation costs. In turn we look at queries with equality predicates, sub/superset predicates, and intersection predicates. For the calculation of the theoretical values we use the formulas appearing in section 4.3.

7.2.1 Equality predicates

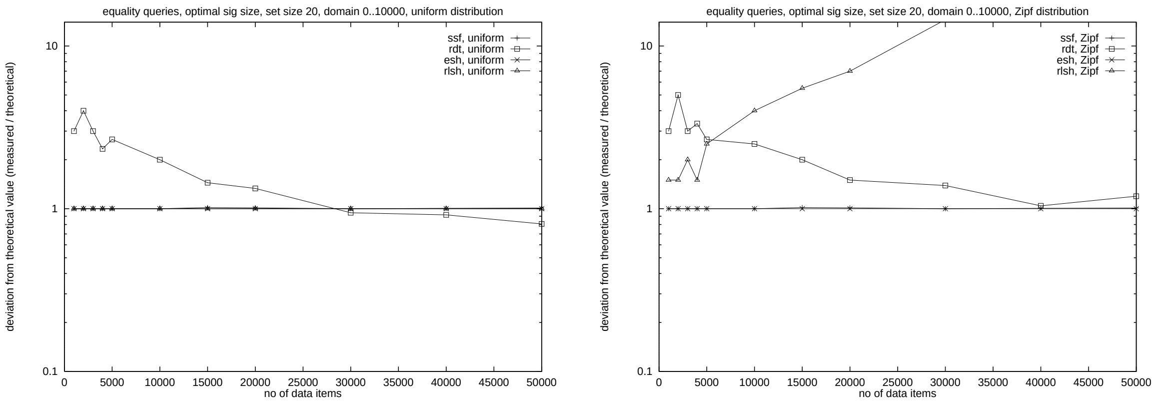

Figure 56: Predicted no. of page acc. vs. actual no. of page acc. (equality pred.)

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	0.84 page acc.	0.84 page acc.
Russian doll tree	4.23 page acc.	6.28 page acc.
extendible signature hashing	0 page acc.	0 page acc.
recursive linear signature hashing	0 page acc.	23.91 page acc.

Table 12: Sample standard deviation for queries with equality predicates

Figure 56 shows the ratio between the expected number of page accesses and the actual number of accessed pages for the evaluation of a query with an equality predicate. The number of page accesses were calculated using (71), (75), (76) and (77).

For uniformly distributed data (left diagram) the cost models are very accurate, the highest sample standard deviation observed was 4.23 page accesses for the Russian doll tree. The exploding costs of the recursive linear signature hashing index for skewed data (right diagram), due to the rapid growth of the number of recursive hash tables, has not been considered in the cost model, which assumes that almost all of the data items are stored in the topmost hash table. For uniformly distributed data this assumption is true, for skewed data this no longer holds. In table 12 all sample standard deviations for all index structures are listed.

7.2.2 Subset and superset predicates

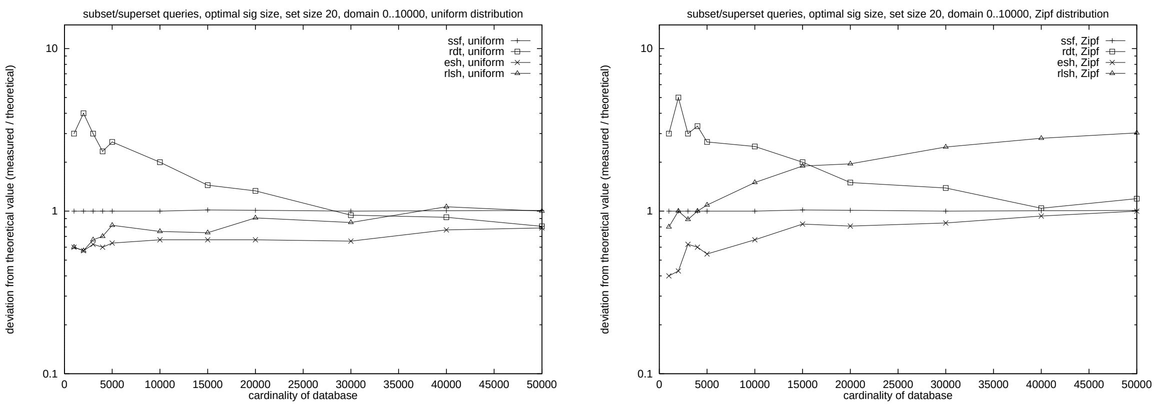

Figure 57: Predicted no. of page acc. vs. actual no. of page acc. (sub/superset pred.)

The curves for the cost ratios for queries with sub/superset queries can be found in figure 57. For the calculation of the theoretical values (78), (80), (101), and (116) were used. For uniformly distributed data (left part of figure 57) the predicted costs are not as

[1]only for queries with subset predicates

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	0.84 page acc.	0.84 page acc.
Russian doll tree [1]	4.23 page acc.	6.28 page acc.
extendible signature hashing	5.86 page acc.	3.81 page acc.
recursive linear signature hashing	3.16 page acc.	32.52 page acc.

Table 13: Sample standard deviation for queries with sub/superset predicates

accurate as for queries with equality predicates. This can also be seen, when looking at the sample standard deviation (in table 13). For a rough estimation of the query costs, though, the cost models are definitely suitable. For skewed data (right part of figure 57) this is not quite true. Although for the sequential signature file, the Russian doll tree, and the extendible signature hashing index the cost models are still valid, for the recursive linear signature hashing index the actual costs are much larger than the predicted costs. The sample standard deviation in table 13 confirms this impression.

7.2.3 Intersection predicates

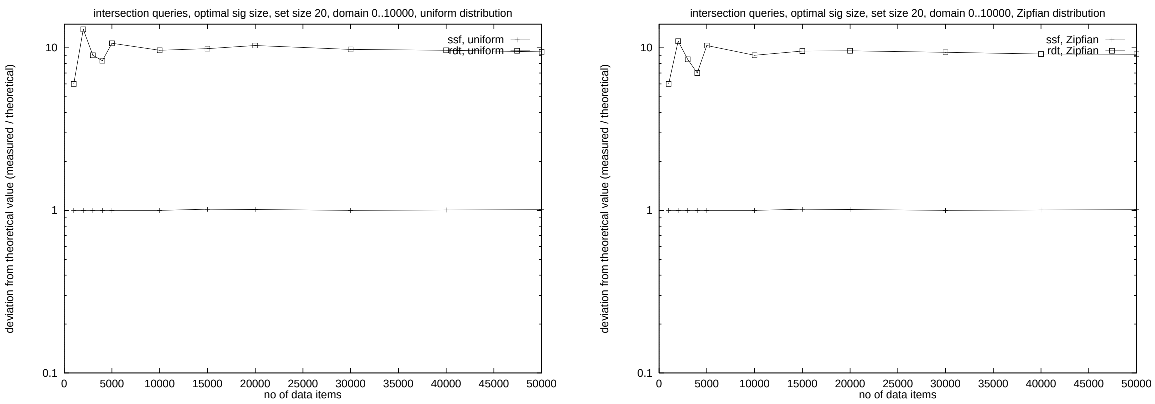

Figure 58: Predicted no. of page acc. vs. actual no. of page acc. (intersec. pred.)

For queries with intersection predicates we only have to check the cost models for the sequential signature file and the Russian doll tree. The ratio between the predicted and the actually measured query evaluation costs is plotted in figure 58. The costs of the sequential signature file are independent of the query type and the distribution of the data, so the expected costs for the sequential signature file (as calculated by (117)) are as accurate as for all other query types. The Russian doll tree performs totally different for queries with intersection predicates, however. The false drop probability for queries with intersection predicates is much higher than the false drop probabilities for queries with

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	0.84 page acc.	0.84 page acc.
Russian doll tree	126.60 page acc.	120.42 page acc.

Table 14: Sample standard deviation for queries with intersection predicates

equality or subset predicates. For the query costs of the Russian doll tree this means that almost all pages of the tree need to be accessed in order to evaluate a query. This is true both for uniformly distributed data and for skewed data. The query evaluation costs are slightly lower for skewed data, because the index is slightly smaller in this case. In table 14 the values for the sample standard deviation can be found.

7.2.4 Summary

Validating the cost models for uniformly distributed data and queries with equality, subset or superset predicates poses no problem. For skewed data, the cost model for the recursive linear signature hashing index becomes inaccurate, because the number of recursive hash tables gets out of control. For queries with intersection predicates neither the index structures nor the corresponding cost models are satisfactory.

7.3 Update costs

In this section we validate the cost models for the update operations. The first part covers the validation of the insertion costs, the second part is about the validation of the deletion costs.

7.3.1 Insertion costs

Figure 59 displays the comparison of the theoretical and actually measured costs for insertions. The theoretical costs were derived by applying (119), (123), (124), and (126). When comparing the predicted costs with the measured costs for uniformly distributed data (left part of 59), we only notice minor deviations. For skewed data (right part of 59), however, the situation is different. While the performance of the sequential signature file, the Russian doll tree, and the extendible signature hashing index is identical to the performance for uniformly distributed data, the insertion costs of the recursive linear signature hashing index differ greatly. The exact deviations for all index structures are summed up in table 15.

7.3.2 Deletion costs

In figure 60 the theoretical and actually measured deletion costs are compared. The theoretical costs were calculated by the formulas (127), (128), (129), and (131). The left part of figure 60 displays the cost ratio for uniformly distributed data and the right part of figure 60 displays the cost ration for skewed data. For the sequential signature file and

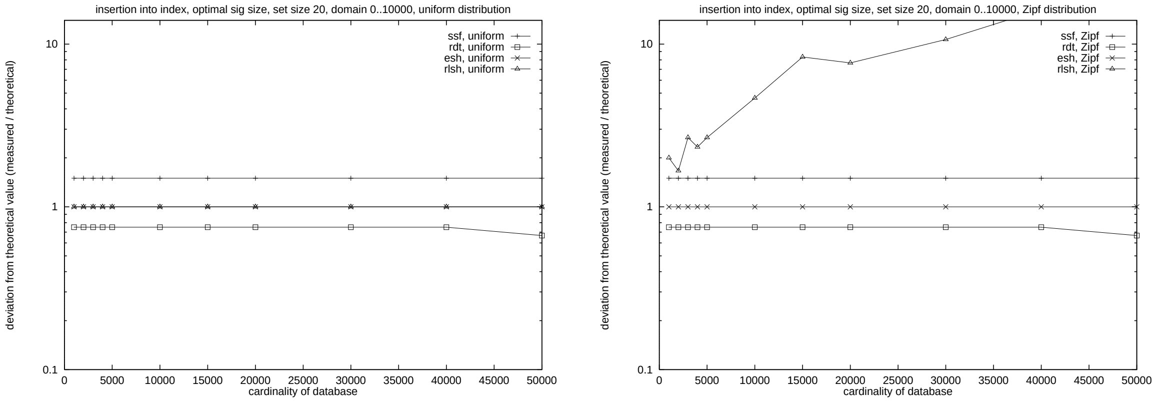

Figure 59: Predicted no. of page acc. vs. actual no. of page acc. for insertion

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	1.05 page acc.	1.05 page acc.
Russian doll tree	1.1 page acc.	1.1 page acc.
extendible signature hashing	0 page acc.	0 page acc.
recursive linear signature hashing	0 page acc.	25.58 page acc.

Table 15: Sample standard deviation for insertions

the Russian doll tree the predicted costs correspond to the actually measured costs well, whereas for the hashing index structures they almost fit perfectly. The exact values for the sample standard deviation can be looked up in table 16.

7.4 Creation costs

In this section we compare the theoretical creation costs (as computed by (134), (135), (136), and (137)) with the actually measured costs for creating the index structures. Figure 61 shows the comparison (for uniformly distributed data on the left hand side, for skewed data on the right hand side). The theoretical costs deviate considerably from the measured costs (the exact values of the sample standard deviation can be seen in table 17). This has two reasons. First, the inaccuracy of the insertion costs, although small, propagates itself when multiplied by the number of data items inserted during the creation of an index. The second reason is the neglection of the updates necessary during the restructuring of the index when an overflow occurs. A revised cost model for the creation costs should take this restructuring into account.

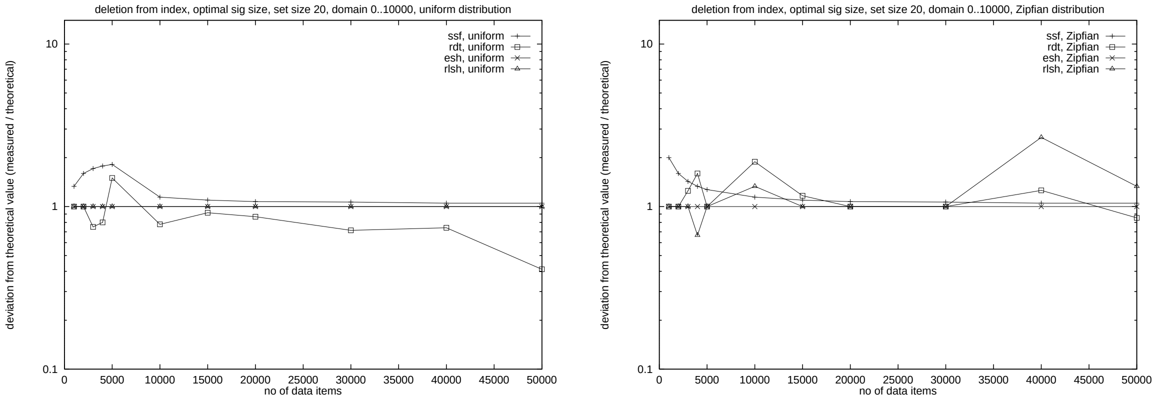

Figure 60: Predicted no. of page acc. vs. actual no. of page acc. for deletion

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	4.99 page acc.	3.59 page acc.
Russian doll tree	7.11 page acc.	3.90 page acc.
extendible signature hashing	0 page acc.	0 page acc.
recursive linear signature hashing	0 page acc.	1.67 page acc.

Table 16: Sample standard deviation for deletions

7.5 Summary of the validation

Summarizing the previous sections we can claim to have validated the cost models to a large degree. For uniformly distributed data the cost models are fully valid. For skewed data this is not necessarily the case, the recursive linear signature hashing index being the exception. It is not clear, however, if any effort should be made for the further analysis of the performance of the recursive linear signature hashing index for skewed data, because in the case of skewed data we have to dissuade from the use of the recursive linear signature hashing index. We also had some problems validating the cost models for the creation costs. The present cost model is a very rough estimation and needs some rework in the future.

8 Summary and Outlook

We presented four different index structures supporting queries involving set-valued attributes. Two of them, the sequential signature file [IKO93] and the Russian doll tree [HP94], were already known. The two other, extendible signature hashing and recursive linear signature hashing, were created by adapting and enhancing extendible hashing [FNPS79], recursive linear hashing [RSD84], and Quickfilter [LL89], [RZ90].

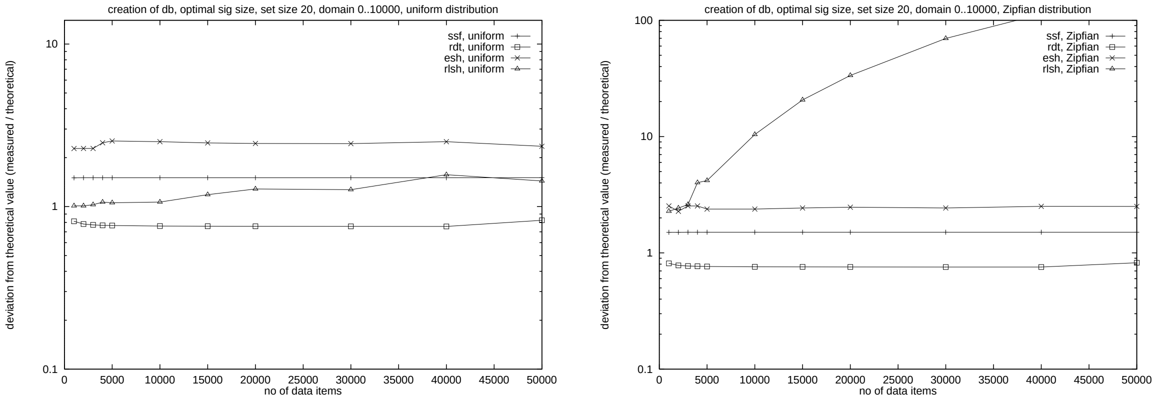

Figure 61: Predicted no. of page acc. vs. actual no. of page acc. for creation

index structure	*sample standard deviation*	
	uniform dist.	*Zipfian dist.*
sequential signature file	24322.64 page acc.	24322.64 page acc.
Russian doll tree	20487.77 page acc.	20525.41 page acc.
extendible signature hashing	102724.39 page acc.	107657 page acc.
recursive linear signature hashing	31586.37 page acc.	23696106.84 page acc.

Table 17: Sample standard deviation for creation

The index structures were implemented and compared intensively using mathematical methods as well as simulation. All of them were put through the tests on common grounds, so that none could gain an unfair advantage over the others. We noted several facts. For queries with equality predicates the extendible signature hashing index reigns supreme. The recursive linear signature hashing index would have been a serious contender, if it were not for its bad performance for skewed data (which it showed for all query types). For subset/superset queries the efficiency of the extendible signature hashing index was not as outstanding, but it was still able to compete with the Russian doll tree. Queries with intersection predicates generally seem to be a tough case. The hashing index structures do not even support it, while the Russian doll tree shows very poor performance. For update operations the extendible signature hashing index also performs very well (even if the sequential signature file has slightly faster insertion costs).

The simulations give a good impression of the performance of the index structures. Future research should try to confirm the results of the simulations in an actual application and try to close the small gaps still left in the cost models. Another area that needs further research is the efficient support of queries containing intersection predicates. Developing new index structures for this case using a different approach like inverted files [ZMR95] or nested indexes [BK89] may bring success.

References

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, 19(2):322–331, June 1990.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(4):290–306, 1972.

[BP94] A. Biliris and E. Panagos. EOS user's guide. Technical report, AT&T Bell Laboratories, 1994.

[Cat97] R. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[Com79] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[CS89] W.W. Chang and H.J. Schek. A signature access method for the Starburst database system. In *Proc. of the 15th VLDB Conference*, pages 145–153, Amsterdam, The Netherlands, 1989.

[CZ93] P. Ciaccia and P. Zezula. Estimating accesses in partitioned signature file organizations. *ACM Transactions on Information Systems*, 11(2):133–142, April 1993.

[Dep86] 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.

[FC84] 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.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD*, Boston, Mass., 1984.

[HP94] J.M. Hellerstein and A. Pfeffer. The RD-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison, 1994.

[IKO93] 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.

[KFIO93] H. Kitagawa, Y. Fukushima, Y. Ishikawa, and N. Ohbo. Estimation of false drops in set-valued object retrieval with signature files. In *Proc. of the 4th Int. Conf. on Foundations of Data Organization and Algorithms*, pages 146–163, Chicago, October 1993.

[KM92] A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison Wesley, Reading, Massachusetts, 1973.

[Lit80] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of the 6th VLDB Conference*, pages 212–223, Montreal, 1980.

[LL89] D.L. Lee and C.W. Leng. Partitioned signature file: Design considerations and performance evaluation. *ACM Transactions on Information Systems*, 7(2):158–180, April 1989.

[LL92] W.C. Lee and D.L. Lee. Signature file methods for indexing object-oriented database systems. In *Proc. of the 2nd Int. Computer Science Conf.*, pages 616–622, Hong Kong, 1992.

[MS86] 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.

[Oto84] E.J. Otoo. A mapping function for the directory of a multidimensional extendible hashing. In *Proc. of the 10th VLDB Conference*, pages 493–506, Singapore, 1984.

[PBC80] J.L. Pfaltz, W.J. Berman, and E.M. Cagley. Partial-match retrieval using indexed descriptor files. *Communications of the ACM*, 23(9):522–528, September 1980.

[Rob79] C.S. Roberts. Partial-match retrieval via the method of superimposed codes. *Proc. of the IEEE*, 67(12):1624–1642, December 1979.

[RSD84] K. Ramamohanarao and R. Sacks-Davis. Recursive linear hashing. *ACM Transactions on Database Systems*, 9(3):369–391, September 1984.

[RZ90] F. Rabitti and P. Zezula. A dynamic signature technique for multimedia databases. In *13th Int. Conf. on Research and Development in Information Retrieval*, pages 193–210, Brussels, Belgium, 1990. ACM SIGIR.

[SD85] R. Sacks-Davis. Performance of a multi-key access method based on descriptors and superimposed coding techniques. *Information Systems*, 10(4):391–403, 1985.

[SDR83] R. Sacks-Davis and K. Ramamohanarao. A two level superimposed coding scheme for partial match retrieval. *Information Systems*, 8(4):273–280, 1983.

[SM96] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1996.

[VM96] 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.

[XH94] 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.

[Yao77] S.B. Yao. Approximating block accesses in database organization. *Communications of the ACM*, 20(4):260–261, 1977.

[ZMR95] 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.

[ZRT91] P. Zezula, F. Rabitti, and P. Tiberio. Dynamic partitioning of signature files. *ACM Transactions on Information Systems*, 9(4):336–369, October 1991.