Performance Enhancements for Advanced Database Management Systems

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Informatiker
Sven Helmer
aus Heilbronn

Mannheim, 2000

Dekan: Professor Dr. Guido Moerkotte, Universität Mannheim
Referent: Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent: Professor Alfons Kemper, Ph.D., Universität Passau

Tag der mündlichen Prüfung: 22. Dezember 2000

Contents

Chapter 1

Introduction

Providing an abstract view of data is one of the main advantages of a database management system (DBMS). In DBMSs this is achieved by making available modeling concepts to map the real world to the database. These concepts are called the data model of a database. The predominant data model today is the relational model, which is ideal for modeling business applications. However, new applications have emerged that are difficult to map to existing data models. Therefore many data models supporting new applications have been introduced, like the object-oriented model [14], object-relational model [88], fuzzy data models [10, 11, 19], and models for semi-structured data (e.g. XML [96]). We investigate two areas where DBMSs are confronted with new demanding applications. First, we look at the introduction of set-valued attributes in DBMSs, an area that has attracted little attention so far. The negligence of this area is likely to change with the growing importance of commercial object-relational database systems, as many of them already provide set-valued attributes or will do so in the near future. Studies on data modeling allowing sets [73, 99] also support the notion of introducing set-valued attributes to the relational world, since it allows users to formulate many queries in a more natural way. This is the case in image [9, 52], genetic or molecular databases [2, 28]. Second, we investigate the challenges that have to be taken on in Data Warehouse environments. Data Warehouses aim at supporting the decision processes of managerial staff by holding huge amounts of historical data that can be analyzed to check the quality of decisions. From the viewpoint of DBMSs the sheer size of the tables in Data Warehouses poses a problem that needs to be addressed.

A lot of important work has been done on the logical level concerning new data models. However, the performance of actual DBMSs is a very important criterion for the acceptance of new data models. The introduction of the relational model by Codd in the early 1970s is a prime example for reluctant acceptance. The relational model was not able to replace the network model and become the dominant model until efficient algorithms and data structures had been developed. Experience has shown that to realize efficient DBMSs for new applications, we do not necessarily have to redesign and rebuild database systems from scratch. Nevertheless, just mapping new data models to older ones and using existing database systems does not get the job done, either. The mapping, if possible at all, results in an overhead to bring applications into line with DBMSs that have not even been optimized for them. This may lead to severe performance losses.

Thus we propose to modify the physical level of existing systems. Areas on the physical level of DBMSs that have contributed considerably to performance gains in the past are efficient join algorithms and methods for associative access. Ever since the invention of relational database systems, tremendous efforts have been undertaken to develop efficient join algorithms. The reason for this is that joins are frequent operations that are very expensive to evaluate during query processing. Another important prerequisite for the fast processing of data in a DBMS is the fast retrieval of this data from secondary storage. In order to accelerate the access, index structures are used. We focus on these two areas, join algorithms and index structures, in our work.

We investigate the efficient processing of set-valued attributes in object-oriented and object-relational databases. The main problem concerning sets is the fact that sets cannot be ordered totally. Many traditional algorithms and data structures in DBMSs, however, rely on ordered data (e.g., sort-merge join, B^+-trees). We cannot apply these techniques in our case; therefore we developed new, efficient join algorithms supporting equality, subset, and superset predicates and compared these algorithms to the naive nested-loop variant. We also devised new index structures for evaluating quickly queries with predicates involving set-valued attributes. We examined the performance of these access methods theoretically (by developing cost models for each index structure) and practically (by exposing the index structures to extensive experiments). For the Data Warehouse environment we dealt with join algorithms for joining very large relations, resulting in an efficient algorithm for 1:N relationships. This algorithm exploits special characteristics of data stored in Data Warehouses, the most important being the relation of content to location. We compared our new algorithm to traditional techniques demonstrating the effectiveness.

This thesis is organized as follows. First of all in Chapter 2 we introduce concepts and terms that are needed later on. Then we give a general overview of join algorithms in Chapter 3. Next we present our new join algorithms in Chapter 4 and Chapter 5. This is followed by an introduction to indexing in DBMSs in Chapter 6. We are then ready to present our work on index structures for set-valued attributes in Chapter 7. Chapter 8 concludes our thesis.

Chapter 2

Preliminaries

We introduce concepts and terms that are needed in later chapters. In Section 2.1, we show how to represent sets using superimposed coding and how this technique can be used to compare sets efficiently. For subset/superset comparisons, we also need a way to rapidly generate all subset/supersets of a given set. We deal with this matter in Section 2.2.

2.1 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 [62]. A code word created by superimposing bit fields is called a *signature* [27]. We use signatures to represent sets in our join algorithms and index structures. There are two advantages to signatures. One is their constant length; keys of constant length are easier to manage than keys of variable length. The other advantage 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.

2.1.1 Basic principles

A signature is a bit field of 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 is generated, in which exactly k bits are set. Afterwards all binary code words are superimposed using a *bitwise or* operation creating the final signature (see figure 2.1 for the algorithm). For a set s, let $sig(s)$ denote the signature of s.

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:

$$s\ \theta\ t \implies \mathrm{sig}(s)\ \theta\ \mathrm{sig}(t) \quad \text{for } \theta \in \{=, \subseteq, \supseteq, \cap\} \tag{2.1}$$

```
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) {
            do {
                rnd = random() % b;
            } while(rnd-th bit is set in tmpSig)
            set rnd-th bit in tmpSig;
            i++;
        }
        sig |= tmpSig;
    }
    return sig;
}
```

Figure 2.1: Algorithm for generating signatures

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 2.1.1 *Let us illustrate the technique of superimposed coding with an example taken from a course database. In this database we store information on students, in particular which courses students attend. Assume for a moment that Alice attends the courses "Computer Science 101", "Math 101", "Programming 101", Bob attends the courses "Math 101", "Physics 101", and Eve attends the courses "Computer Science 101", "Math 101". So we have three sets, S_{Alice}, S_{Bob}, and S_{Eve} with*

$$\begin{aligned}
S_{Alice} &= \{\ \textit{"Computer Science 101", "Math 101", "Programming 101"}\} \\
S_{Bob} &= \{\ \textit{"Math 101", "Physics 101"}\} \\
S_{Eve} &= \{\ \textit{"Computer Science 101", "Math 101"}\}
\end{aligned}$$

Let us now encode these sets. 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 have a hash function $H(x)$ at our disposal that maps the elements from our sets to binary code words.

$$\begin{aligned} H(\text{“Computer Science 101”}) &= 0100\ 1000 \\ H(\text{“Math 101”}) &= 1001\ 0000 \\ H(\text{“Physics 101”}) &= 0100\ 0100 \\ H(\text{“Programming 101”}) &= 0000\ 1100 \end{aligned}$$

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

$$\begin{aligned} sig(S_{Alice}) &= 1101\ 1100 \\ sig(S_{Bob}) &= 1101\ 0100 \\ sig(S_{Eve}) &= 1101\ 1000 \end{aligned}$$

As can be clearly seen in this example, $s\ \theta\ t$ only implies $sig(s)\ \theta\ sig(t)$ for $\theta \in \{=, \subseteq, \supseteq, \cap\}$. For example, $S_{Eve} \subseteq S_{Alice}$ and $sig(S_{Eve}) \subseteq sig(S_{Alice})$, but $S_{Bob} \not\subseteq S_{Alice}$ and $sig(S_{Bob}) \subseteq sig(S_{Alice})$. ◇

2.1.2 Query Evaluation with Signatures

Signatures can also be used to evaluate queries involving set-valued attributes. Assume that we have data items $o_1, o_2, \ldots, o_n$ in our database and all have a set-valued attribute A. Also consider a query set Q. We are interested in all data items for which $Q\ \theta\ o_i.A$ with $\theta \in \{=, \subseteq, \supseteq, \cap\}$ holds. Instead of comparing Q directly with each data item, we first compare $\text{sig}(Q)$ with each $\text{sig}(o_i.A)$. This is much faster, because signatures can be compared using a few, fast bit-operations. During the evaluation of a query, if $\text{sig}(Q)\theta\text{sig}(o_i.A)$ holds, we call o_i a *drop*. We then fetch o_i and compare Q directly to $o_i.A$. If $Q\ \theta\ o_i.A$ also holds, we have a *right drop*, else o_i is a *false drop*. False drops occur, because $Q\ \theta\ o_i.A$ only implies $\text{sig}(Q)\theta\text{sig}(o_i.A)$. We go into details on this matter in the following section.

2.1.3 False drop probabilities of signatures

The probability that a data item turns out to be a false drop—called *false drop probability* d_θ—has been studied intensively [27, 51, 60, 78, 80] and can be approximated by formulas (2.3) to (2.6). 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, i.e. the number of bits set to 1, can be estimated as follows. Assume we have a set M with cardinality $|M|$. The weight of its signature, $|\text{sig}(M)|$, can be approximated by

$$|sig(M)| \approx b \cdot \left(1 - \left(1 - \frac{k}{b}\right)^{|M|}\right) \tag{2.2}$$

where b is the number of bits in a signature and k the number of bits set in a single code word. $\frac{k}{b}$ is the probability that an arbitrary bit in a single code word is set to 1, so $1 - \frac{k}{b}$ is the probability that is set to 0. A bit is set to 0 in the final signature, if this bit is 0 in all of the $|M|$ superimposed code words. Subtracting the result from 1 yields the probability that a bit is set in the final signature. This formula is an approximation because we assume that all bits in a code word are set independently of the others, which is not always the case in real-world applications.

We now present the false drop probabilities for each comparison type as described in [60].

$$d_{=}(b,k,|Q|,|o_i.A|) = \frac{1}{\left(\begin{array}{c} b \\ |\mathrm{sig}(Q)| \end{array}\right)} \quad \text{for } |\mathrm{sig}(Q)| = |\mathrm{sig}(o_i.A)| \tag{2.3}$$

$$d_{\subseteq}(b,k,|Q|,|o_i.A|) \approx (1 - e^{-\frac{k}{b}|o_i.A|})^{k \cdot |Q|} \tag{2.4}$$

$$d_{\supseteq}(b,k,|Q|,|o_i.A|) \approx (1 - e^{-\frac{k}{b}|Q|})^{k \cdot |o_i.A|} \tag{2.5}$$

$$d_{\cap}(b,k,|Q|,|o_i.A|) \approx 1 - \sum_{j=0}^{k-1} \left(\begin{array}{c} |\mathrm{sig}(Q)| \\ j \end{array}\right) (1 - (1 - \frac{k}{b})^{|o_i.A|})^j (1 - \frac{k}{b})^{|o_i.A|(|\mathrm{sig}(Q)|-j)} \tag{2.6}$$

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

$$c_f = (n - c_r) \cdot d_\theta \tag{2.7}$$

We cannot distinguish between right drops and false drops by comparing only signatures. So all c_d data items which are drops have to be fetched 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 data items small. There is an optimal ratio between k and b for which d_θ becomes minimal.

For equality predicates $d_=$ becomes minimal, when

$$\frac{k}{b} = \left(1 - \left(\frac{1}{2}\right)^{|Q|}\right) \tag{2.8}$$

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

$$\frac{k}{b} = \frac{\ln 2}{|o_i.A|} \tag{2.9}$$

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

$$\frac{k}{b} = \frac{\ln 2}{|Q|} \tag{2.10}$$

For intersection predicates no closed formula exists to calculate the optimal ratio $d_{\cap}$ between k and b. Fortunately (2.6) only contains discrete values and for a fixed b an optimal value for k can be calculated relatively fast with a brute force algorithm, computing $d_{\cap}$ for all possible values of k.

2.2 Fast enumeration of subsets/supersets

For the algorithms and index structures presented in the next chapters, we also need a way to rapidly step through all subsets and supersets of a signature for a given set. We use the algorithm utilized by Vance and Maier in their blitzsplit join ordering algorithm [92]. The algorithm to generate all subsets of a given bit-string sig(s) is given below (& denotes *bitwise and* and ˜ denotes *bitwise complement*). When executed, x traverses through all possible subsets of sig(s). ($f(x)$ denotes a function applied to the current subset/superset.)

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

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

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

Chapter 3

Introduction to Join Algorithms

A join operation combines tuples from two different relations such that a join predicate is satisfied. Due to normalization in relational databases different relations often contain related data, so join operations are commonly used during query evaluation. However, join operations are not only found in relational database systems, but also in object-oriented database systems [55] and XML repositories [29]. Unfortunately, join operations are expensive to execute and difficult to implement efficiently. This explains the considerable effort that has been undertaken in order to develop efficient join algorithms. Starting from a simple nested-loop join algorithm, the first improvement was the merge join [8]. Later, the hash join [12, 22] and its improvements [54, 61, 71, 87] became alternatives. (For overviews see [70, 85] and for a comparison between the sort-merge and hash joins see [35, 36].) A lot of effort has also been spent on parallelizing join algorithms based on sorting [25, 67, 69, 82] and hashing [21, 32, 84]. Another important research area is the development of index structures that allow to accelerate the evaluation of joins [41, 57, 56, 72, 91, 95].

In this chapter we give a quick review of the most important join methods: nested-loop join in Section 3.1, sort-merge join in Section 3.2, hash-based joins in 3.3, and last but not least index joins in Section 3.4.

3.1 Nested-Loop Join

A naive strategy for joining two relations is simple iteration. When joining the relations R and S, we traverse the tuples of R (called the outer relation) and compare them to each tuple in S (called the inner relation). If a tuple r from R matches a tuple s from S, we add the pairing of the two tuples to the resulting relation. Figure 3.1 illustrates the algorithm and the concept of nested loops.

Fetching each tuple individually from secondary storage would lead to very low performance. Therefore, in practice, tuples are retrieved in blocks. Let us assume we have n blocks of main memory at our disposal. We read $n-1$ blocks of the outer, smaller relation. Then we read the inner relation block by block, joining the tuples in each block to the tuples in the blocks of the outer relation. When we reach the end of the inner relation, we read the next $n-1$ blocks of the outer relation and repeat the process. This

```
for each tuple r in R {
   for each tuple s in S {
      if r matches s {
         combine r and s
         add to result
      }
   }
}
```

Figure 3.1: Nested-loop join

version is called blockwise nested-loop join (see Figure 3.2). Note that the two innermost loops are performed completely in main memory.

The algorithm can be tuned further by reversing the direction in which the inner relation is retrieved each time we have completed one step of the outer loop. In this way we start a new step of the outer loop with the last retrieved block of the inner relation (no I/O is necessary to fetch this block).

```
for n-1 blocks B_r of R {
   for each block B_s of S {
      for each tuple r in B_r {
         for each tuple s in B_s {
            if r matches s {
               combine r and s
               add to result
            }
         }
      }
   }
}
```

Figure 3.2: Blockwise nested-loop join

3.2 Sort-Merge Join

An improvement to nested-loop joins is the sort-merge join. The algorithm is divided into two phases. In the first phase, both relations are sorted physically on the join attributes. In the second phase both relations are scanned in the order of the join attributes. If two tuples satisfy the join predicate, they are combined and added to the result. Figure 3.3 shows the algorithm (to simplify things, we assume a and b are the join attributes).

This algorithm has to be modified slightly when joining relations of non-unique attributes. When encountering a duplicate value $r(a)$ in the outer relation, we have to back up in

```
sort R on a
sort S on b

read first tuple s in S
for each tuple r in R {
   while s(b) < r(a) {
      read next tuple s in S {
         if r(a) == s(b) {
            combine r and s
            add to result
         }
      }
   }
}
```

Figure 3.3: Sort-merge join

the inner relation to the first value of $s(b)$ that is equal to $r(a)$. This results in higher execution costs for the algorithm, especially when the block of matching tuples in S does not fit into main memory. Without further modification, it is not possible to process non-equijoins, i.e. join predicates with other comparison operators than equality, with this algorithm.

When comparing the performance of sort-merge join to nested-loop join we notice that the improvement in performance of sort-merge join compared to nested-loop join stems from the fact that it avoids comparing tuples that cannot possibly match by ordering the relations. This fact, however, also restricts its use, as it is not possible to define a total order on all domains.

3.3 Hash Join

As we have seen in Section 3.2, we can improve the performance of a join algorithm by filtering out tuples that have no possible chance to be joined. A different approach (apart from sorting the relations) is to hash the tuples of the relations. Hashing can be employed in different ways, some of which will be discussed in the following subsections.

3.3.1 Simple Hash Join

In the most straightforward method of joining using hash tables, the tuples of the inner relation are divided into different buckets by using a hash function on the join attribute values. After that, we hash the join attribute value of each tuple of the outer relation using the same hash function. Each tuple of the outer relation is compared to all tuples in the corresponding bucket (which may be empty) and joined to the matching tuples. Figure 3.4 illustrates the algorithm. This algorithm is only viable, if the relation S fits

```
for each tuple s in S {
   hash s(b)
   put s in appropriate bucket
}

for each tuple r in R {
   hash r(a)
   for each tuple s in corresponding bucket {
      if r(a) == s(b) {
         combine r and s
         add to result
      }
   }
}
```

Figure 3.4: Simple hash join

completely into main memory. Otherwise, we traverse S blockwise, i.e., we load the tuples of each block into main memory, hash them, and then traverse all of R for each block. This is very inefficient for large relations, consequently better alternatives of hash join algorithms were developed (see next section).

3.3.2 Hash-partitioned join

In the simple hash join method, only the inner relation was partitioned into buckets. Techniques for hash-partitioned join partition both relations, thereby dividing the problem into smaller subproblems. One example is GRACE hash join [32, 85]. As it plays a role in Chapter 5, let us give a brief description. When joining two relations R and S, we partition them in a way such that the following two conditions are met. First, each of the partitions of the smaller relation fits into main memory. Second, matching tuples are always found in corresponding partitions of the other relation. The algorithm performs the steps depicted in Figure 3.5 (assuming R is the smaller relation). Obviously, the hash function `h2` for the main memory hash table is different from the hash function `h1`.

3.4 Index Join

There is still another approach to eliminating unnecessary comparisons of tuples that cannot possibly join. We can use an index to access the matching tuples (see Figure 3.6).

This algorithm assumes that an index has been created on the join attribute of relation S. This causes additional costs besides the join costs as the index needs not only to be created but also to be maintained. Also, the look-up costs can be prohibitive, if the join attribute in R takes on many different values, because we need to scan the index for each value.

```
for each tuple r in R {
   hash r(a) using hash function h1
   put r in appropriate output buffer R_i
}
flush all buffers to disk

for each tuple s in S {
   hash s(b) using hash function h1
   put s in appropriate output buffer S_i
}
flush all buffers to disk

for i = 1 to N {
   for each tuple r in R_i {
      hash r(a) using hash function h2
      put tuple in appropriate main memory hash table
   }
   for each tuple s in S_i {
      probe for matching tuples r in main memory hash table of R_i
      if r(a) == s(b) {
         combine r and s
         add to result
      }
   }
}
```

Figure 3.5: GRACE join

```
for each tuple r in R {
   look up matching tuples s in S using index
   combine r and s
   add to result
}
```

Figure 3.6: Index join

Chapter 4

Join Algorithms for Joins with Set Predicates

All of the algorithms presented in Chapter 3 concentrate on simple join predicates based on the comparison of two atomic values. Predominant is the work on equijoins, i.e., where the join predicate is based on the equality of atomic values. Only a few articles deal with special issues like non-equijoins [24], non-equijoins in conjunction with aggregate functions [17], and pointer-based joins [23, 86]. An area where more complex join predicates occur is that of spatial database systems. Here, special algorithms for supporting spatial joins have been developed [13, 38, 66, 49, 74].

Despite this large body of work on efficient join processing, apparently up to now there has been no work describing join algorithms for the efficient computation of the join if the join predicate is based on set comparisons like set equality ($=$) or the subset relation ($\subseteq$). These joins were irrelevant in the relational context since attribute values had to be atomic. However, newer data models like NF^2 [79, 83], object-oriented models like the ODMG-Model ([14]), or object-relational models [88] support set-valued attributes, and many interesting queries require a join based on set comparison. Consider for example a query looking for faithful couples. There we join persons with other persons on condition that they have the same children. Another example would be matching jobs to persons. In this case we join job offers with persons such that the set-valued attribute *required-skills* is a subset of the persons' set-valued attribute *skills*.

We propose several different main memory join algorithms for joining relations on set-valued attributes that are based on nested-loop, sort-merge, index and hash variants [47]. The design parameters for the different alternatives are discussed and evaluated thoroughly before comparing the algorithms to each other. The results of our experiments show that there are much better alternatives to the naive nested-loop algorithm.

The rest of this chapter is organized as follows. In the next subsection, we introduce some basic notions needed in order to develop our join algorithms. Sections 4.2 and 4.3 introduce and evaluate several join algorithms where the join predicate is set equality and the subset relation. Section 4.4 concludes the chapter.

4.1 Preliminaries

4.1.1 General Assumptions

For the rest of this chapter, we assume two relations R_1 and R_2 with set-valued join attributes a and b. We do not care about the exact type of the attributes a and b– that is, whether they are a relation, a set of strings, or a set of object identifiers. We just assume that the attribute values are sets and that the elements of these sets provide a way of checking whether they are equal.

Our goal is to compute the join expressions

$$R_1 \bowtie_{a=b} R_2$$

and

$$R_1 \bowtie_{a \subseteq b} R_2$$

efficiently. More specifically, we introduce join algorithms based on sorting and hashing and compare their performance with a simple nested-loop strategy. In addition to this, we describe a tree-based join algorithm and evaluate its performance.

For convenience, we assume that there exists a function m which maps each element within the sets of $R_1.a$ and $R_2.b$ to the domain of integers. The function m depends on the type of the elements of the set-valued attributes. For integers, the function is the identity, for strings and other types, techniques like folding can be used. From now on, we assume without loss of generality that the type of the set elements is integer. If this is not the case, the function m has to be applied before we do anything else with the set elements.

4.1.2 Set Comparison

The costs of comparing two sets using $=$ or $\subseteq$ differ significantly depending on the algorithm. Consider the case in which we want to evaluate $s \subseteq t$ for two sets s and t. We could check whether each element in s occurs in t. If t is implemented as an array or list, then this algorithm takes time $O(|s| * |t|)$. Set equality can then be implemented by testing $s \subseteq t$ and $t \subseteq s$, doubling the running time. For small sets, this might be a reasonable strategy. For large sets, however, the comparison cost with this simple strategy can be significant. Hence, we consider further alternatives for set comparison.

One obvious alternative is to have a search tree or a hash table representation of t. Since we assume that the representation of set-valued attributes is not of this kind but instead consists of a list or an array of elements, we abandoned this solution since the memory consumption and cpu time needed in order to construct this indexed representations are too expensive for a main memory algorithm in comparison to the methods that follow.

Another alternative for implementing set comparison is based on sorting the elements. Assume an array representation of the elements of the set, and denote the i-th element of the array representing set s by $s[i]$, then the following algorithm implements set comparison $s = t$, if the sets are sorted:

```
if(s->setsize != t->setsize)
  return false;
for(int i=0; i < setsize; i++) {
  if (s[i] != t[i])
    return false;
}
return true;
```

We comment on two details of this algorithm. First, note that we introduced a pretest by comparing the cardinality of the sets. This kind of pretest is used in every set comparison algorithm we implemented– also in the above mentioned trivial one. Second, when applying this sort-based algorithm for set equality within our join algorithms, we do not assume that the elements of the set are sorted. Instead, the sort is performed by the join algorithms explicitly. This way, the comparison with other join algorithms is not biased by additional assumptions. Note that this algorithm runs in time $O(|s|)$. Since we do not assume that the sets are sorted, we have to add $O(|s| \log |s| + |t| \log |t|)$ for sorting s and t.

A predicate of the form $s \subseteq t$ can also take advantage of sorting the sets. Again, we start by comparing the smallest elements. If $s[0]$ is smaller than $t[0]$, there is no chance of finding $s[0]$ anywhere in t. Hence, the search result will be negative. If $s[0]$ is greater than $t[0]$, then we compare $s[0]$ with $t[1]$. In case $s[0] = t[0]$, we can start comparing $s[1]$ with $t[1]$. The following algorithm implements this idea:

```
if(s->setsize > t->setsize)
  return false;
i=j=0;
while(i < s->setsize && j < t->setsize) {
  if(s[i] > t[j]) {
    j++;
  } else if (s[i] < t[j]) {
      return false;
  } else { /*  (s[i] == t[j]) */
      i++;
      j++;
  }
}
if(i==s->setsize)
  return true;
return false;
```

Note that the running time of this algorithm is $O(|s|+|t|)$. Again, since we do not assume that the sets are sorted, we have run time complexity of $O(|s| \log |s| + |t| \log |t|)$.

The third alternative we considered for implementing set comparisons is based on signatures. This algorithm first computes the signature of each set-valued attribute and then compares the signatures before comparing the actual sets using the naive set comparison algorithm. This gives rise to a run time complexity of $O(|s| + |t|)$. Signatures and their

computation are the subject of the next section. Furthermore, the next section introduces some basic results that will be needed for tuning some of the hash join algorithms.

4.1.3 Implementation of Signatures

As we are implementing main memory algorithms, the signatures have to be generated as fast as possible. Consequently we set only one bit within the signature for each element of the set whose signature we want to compute (i.e. $k = 1$, see also Section 2.1). Assuming a function m_{sig} that maps each set element to an integer in the interval $[0, b[$, the signature can be computed by successively setting the $m_{sig}(x)$-th bit for each element x in the set. Based on $m_{sig}(x)$ we can now give the algorithm to compute the signature sig(s) for a set s.

```
sig = 0;
for(int i=0; i < s.setsize; i++)
   sig |= 1 << m_sig(s[i]);
return sig;
```

The function m_{sig} can be implemented in several different ways. We investigated two principal approaches. The first approach uses a random number generator whose seed is the set element. The resulting number gives the bit to be set within the signature. In our implementation, we used two different random number generators: *rand*() of the C-library and *DiscreteUniform* of the GNU-Library. The second approach just takes the set element modulo the signature length. The advantage of the former is a reduction of the false drop probability, the advantage of the latter is a much better run time. We will investigate this trade-off experimentally.

4.2 Joins with Equality Predicates

This section describes the algorithms we evaluated for implementing a join of the form $R_1 \bowtie_{a=b} R_2$ where R_1 and R_2 are relations with set-valued attributes a of R_1 and b of R_2. We discuss several variants of four major approaches. First, we briefly evaluate three variants of the nested-loop approach. Subsequently, the sort-merge join and the tree-join are described. Last, we introduce the hash join variants.

4.2.1 Nested-Loop Joins

The nested-loop join is implemented by two nested loops ranging over the tuples of the inner and outer relation, respectively. We implemented the nested-loop join algorithm with the three different set comparison operations described above: the naive one, the one based on sorting the sets, and the comparison based on signatures.

In Figure 4.1, these three variants are evaluated. The top row shows the results for varying set sizes, the bottom row gives the results for fixed set sizes. Hence, the pretest on set

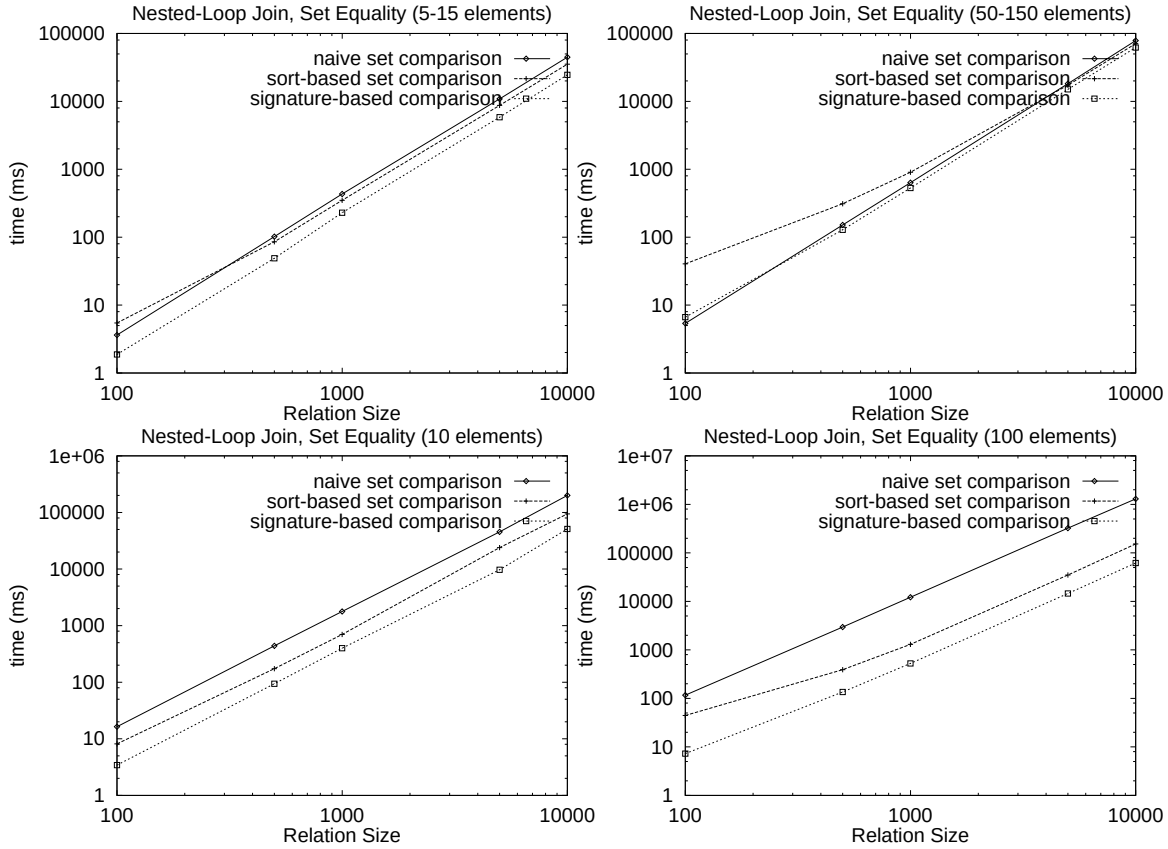


Figure 4.1: Performance of different nested-loop join algorithms (for equality predicates) [1]

sizes only accelerates the set comparison for the top row. In most cases–except for small relations and varying set sizes– the fastest comparison is that by signatures. Hence, we will use this a reference point for comparing subsequent algorithms.

4.2.2 Sort-Merge Join

The main idea of sort-merge join algorithms is to sort the relations to be joined on their join attributes and to merge subsequently the two sorted relations. If the join attributes have a simple, ordered domain, this approach is well known. However, ordering set-valued attributes seems less obvious on first sight. The idea is to use a lexicographical ordering on sets.

More specifically, our sort-merge join sorts the relations in two steps. In a first step, the

[1] In the top row, the number of elements per set varies uniformly between 5 and 15 for the left-hand side of the figure and between 50 and 150 for the right-hand side of the figure. In the bottom row, the number of elements is always 10 for the left-hand side figure and 100 for the right-hand side figure. Along the x-axes relation sizes are varied. The y-axes shows the cpu-time in milliseconds necessary for completing the join as measured on a Sun Sparc Station 20 with 64MB Main Memory. Note the logarithmic scale on both axes.

sets within the join attributes are sorted. In a second step, the relations themselves are sorted on their join attributes. We use a lexicographic ordering for sorting the tuples according to the set-valued join attributes. For example, the list of sets

$$\{1, 4, 7\}, \{1, 5, 7\}, \{2, 5\}$$

is lexicographically ordered, whereas the list of sets

$$\{1, 5, 7\}, \{1, 4, 7\}, \{2, 5\}$$

is not. At first glance this looks quite complicated and is not needed for equality predicates. However, as we want to use the same technique for subset predicates, we already introduce it right here.

Now, the merge phase proceeds as follows. For the outer relation–say R_1–exist two pointers (*low* and *high*) to tuples within it. These pointers indicate the lowest and the highest tuples within the sorted relations that are equal. A third pointer (j) is used to range over the inner relation–say R_2. The join attributes of the first two tuples t_1 and t_2 of the relations to be joined are compared. If the set a of t_1 lexicographically precedes the set b of t_2, then we can advance the *low* pointer of R_1. If the set b of t_2 precedes lexicographically the set a of t_1, then we advance the pointer (j) of R_2. If they are equal, a result tuple can be built. Further, we have to check subsequent tuples in R_1 whether they have the same a value. We do so by advancing the second pointer (*high*) and checking separately each tuple it points to until we meet a tuple whose a value is unequal to the current $t_2.b$ value. If the next tuple in R_2 does not match the tuples in R_1 between the low and high pointer, these tuples can be skipped. We also have to be careful that the loop variables stay in bounds.

```
low = 0;
high = 0;
j = 0;
while(low and j in bounds) {
  while(R1[low].a != R2[j].b && low and j in bounds) {
    if(R1[low].a < R2[j].b) {
      low++;
    }
    else {
      j++;
    }
  }
  do {
    build result tuple;
    high++;
  } while(R1[high].a == R2[j].b && high in bounds);
  j++;
  if(R1[low].a != R2[j].b) {
    low = high;
  }
}
```

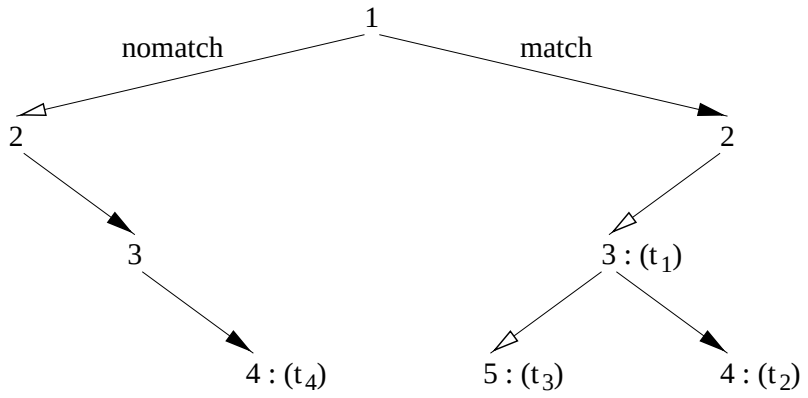
Since sorting has to be done before applying this algorithm, the set comparison algorithm is the one based on sorted sets. The performance evaluation of sort-merge join can be found in Section 4.2.5

4.2.3 Tree Join

The basic idea behind a tree-based join algorithm is to use a temporary index. We consider a binary tree as the temporary index structure (in the next section we look at hash tables). Let us give an example. Consider a relation containing the following four tuples with a set-valued attribute a:

1. $t_1.a = \{1, 3\}$
2. $t_2.a = \{1, 3, 4\}$
3. $t_3.a = \{1, 5\}$
4. $t_4.a = \{2, 3, 4\}$

The corresponding tree is



Every node in the tree consists of a key, a list, and two pointers:

```
struct treenode {
  int key;
  list tuplelist;
  treenode* match;
  treenode* nomatch;
}
```

The key is equal to a set element that can be found in at least one of the relation's tuples. Each node in the tree (denoted by its key in the diagram above) represents a different set. The node's list (in parantheses, following the colon) contains all tuples that have a set-valued attribute that is equal to this set. The pointers lead to further sets. All sets that contain the node's key are found by following the *match* pointer (black tipped

arrow). All sets that do not contain the node's key are found by following the *nomatch* pointer (white tipped arrow).

Finding all matching tuples for a given set-valued attribute of a query tuple is now an easy task. Starting with the smallest element in the sorted query set and the root of the tree, we compare the element with the key of the current node. If they are not equal, we try to find the element in the nomatch path of the node. If they are equal, we follow the match edge and continue searching for the next element of the set. If we have found all elements in the tree, then the list of the node that matched the last element of the set contains all matching tuples.

```
list find(tuple t) {
  treenode* currentNode = root;

  for all elements in t.a in ascending order {
    while(currentNode->key != element) {
      currentNode = currentNode->nomatch;
      if(currentNode == NULL) {
        /* matching tuples do not exist */
        return empty list;
      }
    }
    if(last element) {
      return currentNode->tuplelist;
    }
    else {
      currentNode = currentNode->match;
      if(currentNode == NULL) {
        /* matching tuples do not exist */
        return empty list;
      }
    }
  }
}
```

Building the tree is a little more complex. Before inserting the tuples into the tree, the elements within each set-valued attribute are sorted. (We do not need to sort the relations themselves as we did for the sort-merge approach.) The insertion of a tuple proceeds as follows. We start with the first set element of the tuple and go down the tree along the nomatch edges until we find a node with a key that is greater or equal to our element. If the key is greater, we know that the node we are looking for has not been inserted into the tree, yet. At this point we have already passed the correct place for insertion. So we have to create a new node with an empty list and a key equal to our current element. Then we back up and insert it between the current node and the parent of the current node. We have to be careful to insert it correctly, i.e. at the match or no match pointer. This depends on whether we followed a match or nomatch edge to reach our current node. Now we are at a node whose key is equal to our current element (regardless of inserting

the node or finding it already existing). If the current element is the last element of the set to be inserted, we insert the set an the current node. If not, we have to continue with the match edge and the next element of the set.

```
void insert(tuple t) {
  treenode* currentNode = root;

  for all elements in t.a in ascending order {
    while(currentNode->key < element) {
      if(currentNode->nomatch == NULL) {
        create new node with key = element and empty list;
        insert new node at currentNode->nomatch;
      }
      currentNode = currentNode->nomatch;
    }
    if(currentNode->key > element) {
      create new node with key = element and empty list;
      connect new node to parent of currentNode;
      connect new node to currentNode;
      currentNode = new node;
    }
    if(last element) {
      insert t into currentNode->tuplelist;
      return;
    }
    if(currentNode->match == NULL) {
      create new node with key = element and empty list;
      insert new node at currentNode->match;
    }
    currentNode = currentNode->match;
  }
}
```

4.2.4 Hash-Loop Join

The basic idea behind a tree join was to use a binary tree structure as a temporary index. Instead of a tree we now build a hash table for the tuples of the inner relation hashed on their join attributes. We have several techniques for hashing at our disposal. For the simplest alternative we hash the sets by their cardinality. This only yields a good distribution if the sets vary greatly in cardinality. However, since this is not applicable in general, we consider two other alternatives:

- direct hashing
- signature hashing

Direct hashing means that we map each set directly to a hash key (as opposed to signature hashing, where we map a set to a signature and then map the signature to a hash key). As we assume that the set elements have already been mapped to integers, we use the sum of the set elements modulo the hash table size as our hash key. We label this alternative with *sum*. When looking up sets in the hash table we resolve collisions by comparing the sets in the collision chain using the naive set comparison algorithm. We also implemented a variant with sorted sets to accelerate the comparison. However, the average collision chain length is too short for this approach to pay off. (see Table 4.1).

Signature hashing is based on superimposed coding (from Section 2.1). Several aspects have to be considered when using signatures for hashing:

- choosing the right signature length,
- allocating memory for signatures,
- computing signatures, and
- mapping signatures to hash values.

Signature Length

The performance of signatures depends on their size. For a fixed set size, the larger the signature, the smaller the false-drop probability. Hence, a large signature seems to be the best. However, several effects complicate the issue. First, the mapping of larger signatures is more expensive in terms of computing time (more data has to be processed). Second, if the size of the hash key remains constant for increasing signature sizes, the probability that two different signatures are mapped to the same hash key grows. This leads to a larger number of collisions. Third, we store the signatures in the hash table, because we not only use them as an intermediate step for computing hash keys (this would not pay off). We also employ signatures for resolving collisions during a lookup. We compare sets only if their signatures match. If the signatures become too large, the storage overhead would not be negligible anymore. Hence, we have a trade-off between lowering the false-drop probability on one hand and minimizing storage overhead on the other hand. Considering the formula for false-drop probabilities in case of set equality, we can use a signature of 32 bits to obtain a false-drop probability below 1.4e-06 for up to 60 elements per set. Beyond 60 elements, the false drop probability increases dramatically. Hence, we use 64 bits for sets containing up to 150 elements. The resulting false-drop probability is then less than 2.1e-09. These values are valid for sets with constant cardinalities.

Allocating Memory for Signatures

We have to distinguish two different cases when allocating memory for signatures. On one hand we can extend the tuples by a certain number of bytes to hold the signatures. On the other hand we can allocate the memory for the signatures dynamically and attach the signatures to their tuples. We implemented both alternatives in order to derive an upper bound on the performance loss of dynamic signature creation.

The question of dynamic versus static memory allocation applies to the tuples of the inner relation, which are inserted into the hash table with their respective signatures. For the tuples of the outer relation we allocate memory for one signature. As we go through the tuples of the outer relation this signature is cleared and reused for each tuple.

Computing Signatures

As mentioned in Section 4.1.3 there are several alternatives for computing signatures. We employ the following scheme to denote the different approaches:

ran for computing the signatures using the C-library *rand()* function

dis for computing the signatures using the GNU-library *DiscreteUniform* function

mod for computing the signatures by simply calculating the modulus of the element sum with respect to the signature length b.

Mapping Signatures to Hash Keys

The general problem of mapping signatures to hash keys is the smaller size of the hash keys. It is not unusual to have signatures containing a few hundred bits. Using a full signature as hash key would result in huge hash tables in this case. Consequently we have to map large signatures to smaller hash keys.

For our experiments we assume hash keys smaller than 2^{32}. When mapping signatures to hash keys we distinguish three different cases. If a signature fits into a 32-bit integer, we map it directly onto a hash key by evaluating signature modulo hash table size. If the signature is larger we consider two alternatives. Either we partition the signature into bitvectors of maximally 32 bits and fold these bitvectors with a bit-wise exclusive or operation obtaining a bitvector with maximally 32 bits (denoting this alternative by *fold*), or we just consider the lowest 32 bits of the signature and ignore the rest. This alternative is called *truncate*.

Implemented Alternatives

Having discussed the different approaches for implementing hash-based joins in the last few paragraphs let us now list the different combinations we have implemented. For direct hashing we have two alternatives *sum* and *sumS*. Both of them add the set elements to calculate the hash keys. Additionally *sumS* uses sorted sets to speed up the resolution of collisions.

sum sum of set elements modulo hash table size

sumS sum of set elements modulo hash table size and sorted sets

For signature hashing we have more variety. We implemented different signature sizes, 32 and 64 bits. Furthermore we realized all three variants of computing signatures: *ran*, *dis*, and *mod*. For signatures larger that 32 bits we had to decide how to fit them into 32 bits. Both variants, folding and truncating, were implemented. This gives us another nine different algorithms:

ran [32] 32 bit signature using *rand*

dis [32] 32 bit signature using *DiscreteUniform*

mod [32] 32 bit signature using *mod*

ran [64x] 64 bit signature using *rand*, folding (bit exclusive or) to 32 bit

dis [64x] 64 bit signature using *DiscreteUniform*, folding (bit exclusive or) to 32 bit

mod [64x] 64 bit signature using *mod*, folding (bit exclusive or) to 32 bit

ran [64t] 64 bit signature using *rand*, use only lowest 32 bits for hashing (truncate)

dis [64t] 64 bit signature using *DiscreteUniform*, use only lowest 32 bits for hashing (truncate)

mod [64t] 64 bit signature using *mod*, use only lowest 32 bits for hashing (truncate)

In addition to these variants we experimented with dynamically allocated signatures. We implemented two different techniques:

ZF [nn,32] arbitrary signature length, *mod* for building signature, folding (bit exclusive or) to 32 bit, *nn* denotes the actual signature length

ZT [nn,32] arbitrary signature length, *mod* for building signature, use only lowest 32 bits for hashing (truncate), *nn* denotes the actual signature length

Evaluating Hash-based Joins

We ran several experiments in order to evaluate the performance of the different hash-based alternatives. We start by investigating the two direct hashing variants, then take a close look at the signature-based techniques.

The main question that we want to answer for direct hashing is whether the overhead for sorting the sets pays off during the resolution of collisions in the hash table. Figure 4.2 shows typical results of some of the experiments we conducted. It can be clearly seen that the variant that does not sort the sets is always faster than the other algorithm. It is also interesting to note that for larger sets the performance of the non-sorting algorithm even improves. On account of the larger sets the sums of the elements have a greater variation which results, on average, in shorter collision chains. The sorting algorithm cannot capitalize on this, because it now has the additional overhead of sorting large sets.

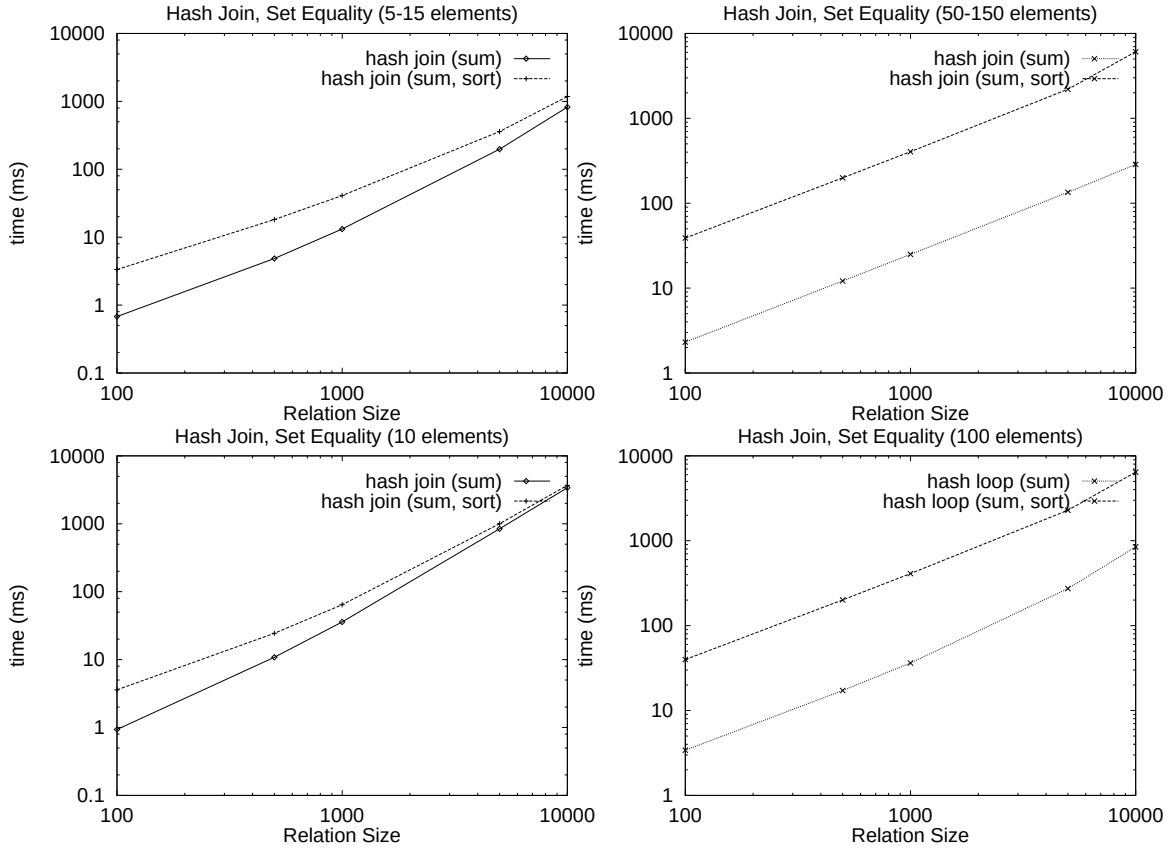


Figure 4.2: Direct hashing: sort-based versus naive set comparison (for equality predicates)

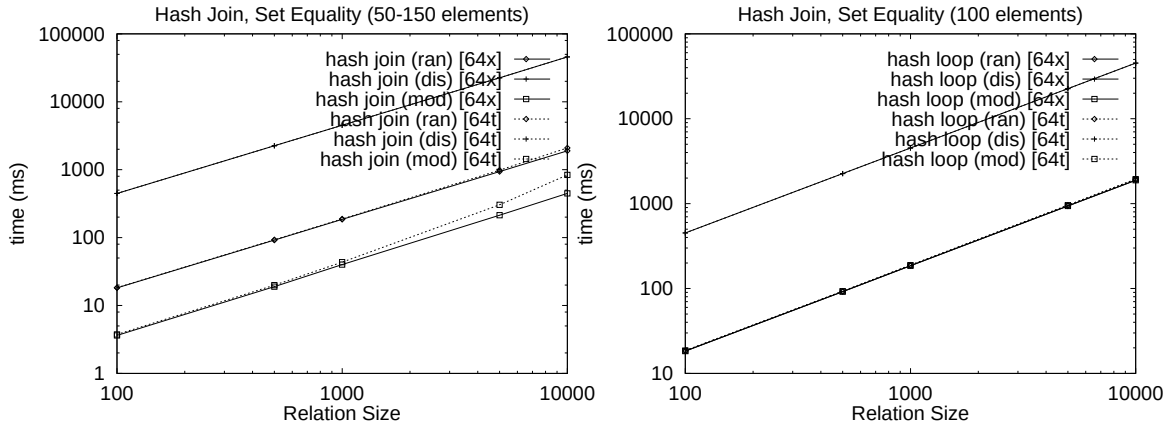


Figure 4.3: Signature hashing: folding vs. truncation (for equality predicates)

For future comparisons of direct hashing algorithms we restrict ourselves to the superior *sum* alternative.

For signature hashing we have several parameters that need to be examined. Let us

start with the mapping from signatures to hash keys. For large signatures we need to fold or truncate the signatures to 32 bits. Figure 4.3 show a comparison between folding and truncating. The performance of these two techniques is very similar, with a small advantage for folded signatures. Folding spreads out the hash key values slightly better resulting in a smaller collision chain length. We investigated the maximal chain length by inserting 500 tuples into a hash table for each alternative. Table 4.1 depicts the results for various set sizes. Usually the collision chain lengths for folded signatures are smaller than those for truncated signatures.

Set Size	sum	32bit sig			64bit sig, fold			64bit sig, 32bit truncate		
		dis	ran	mod	dis	ran	mod	dis	ran	mod
10	6	5	6	6	7	7	8	9	7	7
9-11	4	6	6	7	7	7	6	8	8	8
44	5	7	8	7	5	5	6	5	5	5
40-48	6	7	7	7	5	5	5	7	4	5
100	4	119	128	128	7	6	6	8	6	9
90-110	5	109	126	126	7	5	6	8	6	8

Table 4.1: Maximal collision chain lengths for different alternatives

Next we look at the different computation techniques for signatures (see Figure 4.4). The simple *mod* alternative is also the fastest. The quality of the signatures produced by the *dis* and *rand* algorithms in regard to collision chains is superior to *mod* (see Table 4.1), but they need much more time to achieve this goal. Thus for main memory environments the use of random number generators is too costly.

We were also interested in the performance of the join algorithms when allocating memory dynamically. As expected there is an observable overhead for the dynamic allocation. For small sets and small signatures (left hand side of Figure 4.5) static allocation of memory is about 2 to 3 times faster than dynamic allocation. For large sets and large signatures (right hand side of Figure 4.5) the overhead of dynamic allocation is below 10%, on account of the more complex computation of signatures, which consumes much more time than the creation. Using static memory allocation is quite reasonable, since the query optimizer determines the evaluation plan and is able to consider the additional space when allocating memory for the tuples.

In summary, we can say that there are several different ways to implement hash-based join algorithms and it is not always obvious which one is the most efficient (see Figure 4.6). For small sets the *mod* variant (with static memory allocation) performs best, whereas for large sets the *sum* variant catches up on the signature-based algorithms. For large sets the performance of the direct hashing algorithm improves slightly (see also Figure 4.2). By contrast, large sets have a negative influence on the signature-based algorithms. We are forced to use larger signatures, with the known consequences.

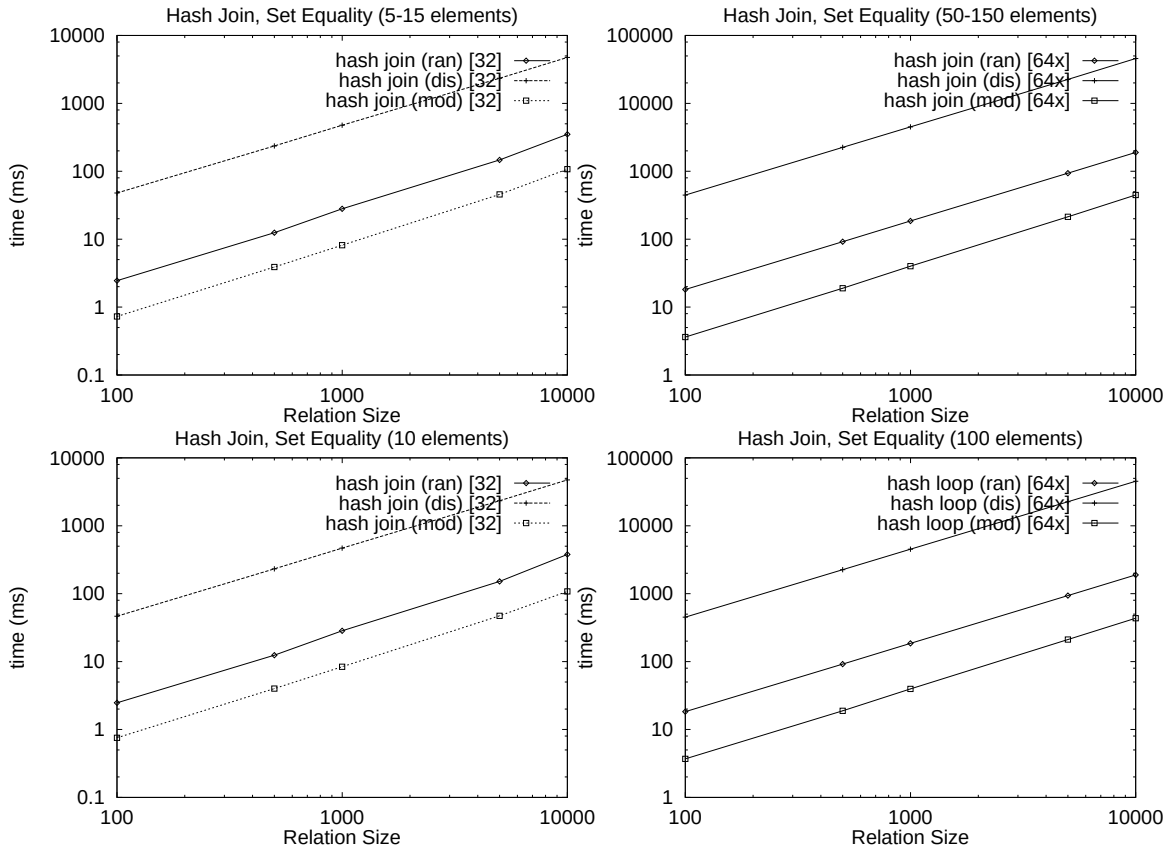


Figure 4.4: Signature hashing: alternative approaches generating signatures (for equality predicates)

4.2.5 Comparison of Algorithms for Equality Predicates

In Figure 4.7 we compare the nested-loop, sort-merge, tree and hash-based join variants. The signature-based *mod* alternative (with static memory allocation) is a very strong contender. Only for large sets is the direct hashing *sum* alternative able to catch up. If, for some reason, a hash-based join cannot be applied, then the sort-merge join is the algorithm of choice. The tree-join is close behind the sort-merge join, but there is not enough compensation for the significant storage overhead of the index. The nested-loop join seems to be an alternative for small relations only, in view of its high running time.

For us, the most important conclusion was that there exist algorithms which are much more efficient than the naive nested-loop variant. The alternative algorithms are surprisingly efficient. Joining two relations with 10,000 tuples each based on the equality of their set-valued attributes consisting of 100 elements within a second or two seems to be a reasonable result, especially when we consider that the best nested-loop variant needs almost one and a half minutes for the same task.

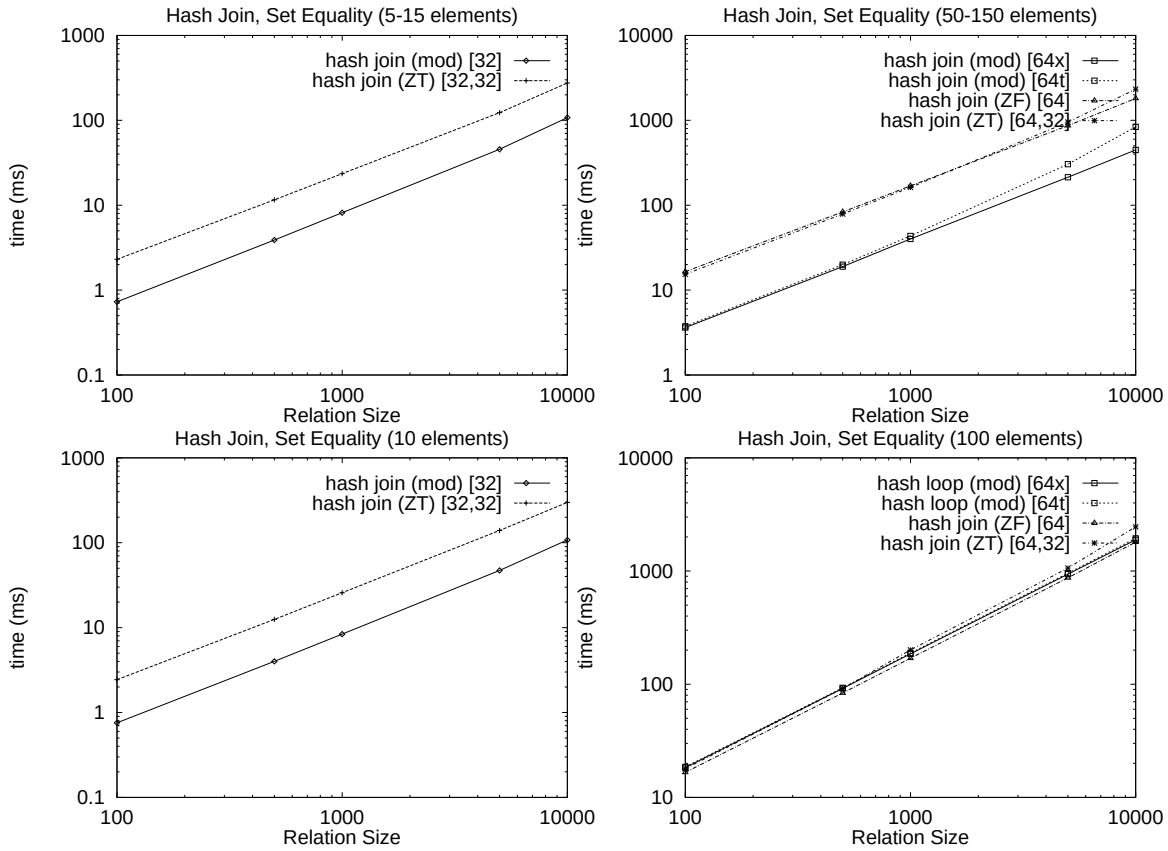


Figure 4.5: Signature hashing: overhead of dynamic signature creation (for equality predicates)

4.3 Joins with Subset Predicates

This section discusses algorithms to compute $R_1 \bowtie_{R_1.a \subseteq R_2.b} R_2$ for two relations R_1 and R_2 with set-valued attributes a and b. Obviously, these algorithms will also be useful for computing joins like $R_1 \bowtie_{R_1.a \supseteq R_2.b} R_2$, $R_1 \bowtie_{R_1.a \subset R_2.b} R_2$, and $R_1 \bowtie_{R_1.a \supset R_2.b} R_2$. For the latter two only slight modifications are necessary. Like in Section 4.2 we discuss different variants of nested-loop, sort-merge, tree-, and hash-join.

4.3.1 Nested-Loop Joins

We implemented three different variants of nested-loop joins. The first is based on naive set comparison, the second employs sorted sets, and the third utilizes signatures. For details on the different implementations of set comparisons with subset predicates see Section 4.1.2. We compared these algorithms experimentally. The results of these experiments (Figure 4.8) resemble those for equality predicates. The signature-based algorithm performs best, whereas the naive algorithm performs worst. Hence, we use the signature-based variant for further comparisons with other join algorithms.

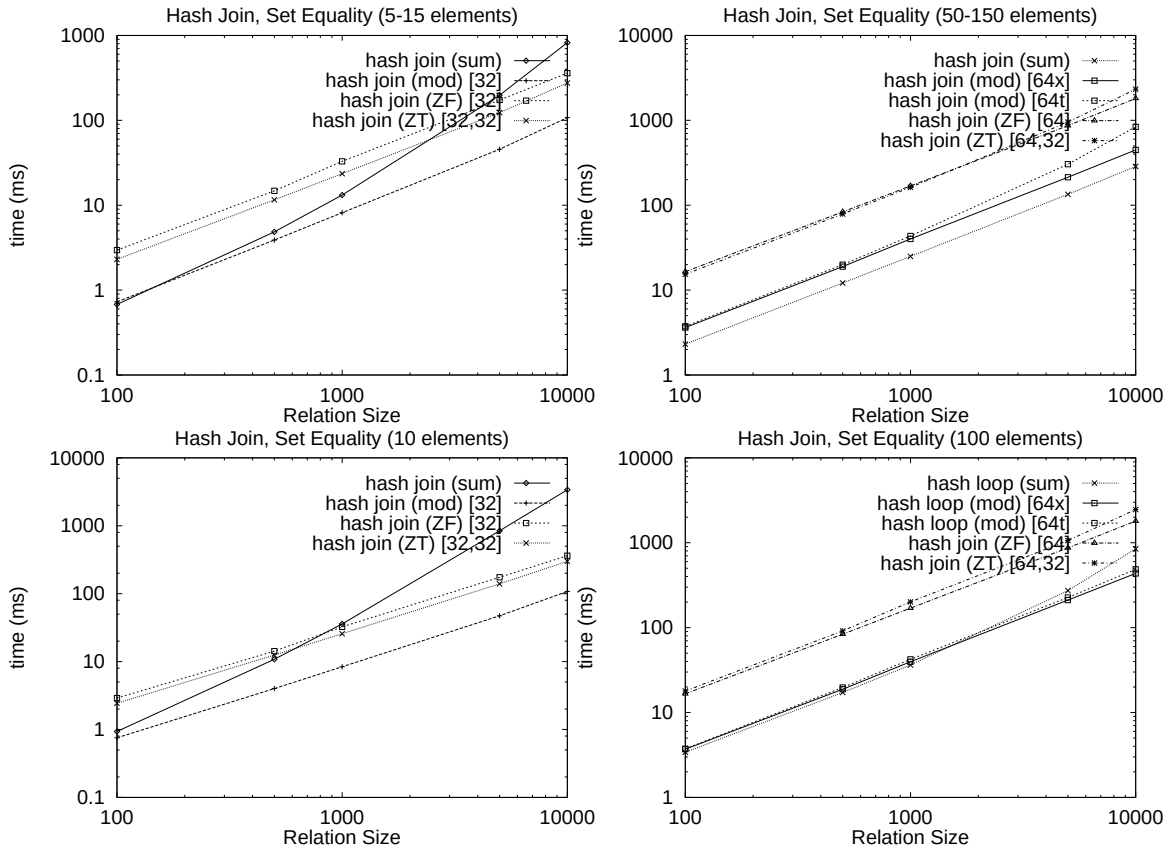


Figure 4.6: Signature hashing: comparison of the best alternatives (for equality predicates)

4.3.2 Sort-Merge Join

As already mentioned in Section 3.2 a simple sort-merge join cannot be used to join two relations on a non-equijoin predicate. Consequently, we have to modify the sort merge algorithm, arriving at a nested loop/sort-merge hybrid. Let us describe the modifications in detail.

In both relations R_1 and R_2 we sort each set-valued attribute. In addition to that, we sort the tuples of the inner relation R_2 lexicographically on the join attribute. We step through the (unsorted) outer relation R_1 tuple by tuple. For each tuple in R_1 we loop through R_2, aborting the loop as soon as the smallest element in the set-valued attribute of the current tuple in R_1 is smaller than the smallest element in the attribute of the current tuple in R_2. We can do this because the smallest element in the attributes of subsequent tuples in (the lexicographically sorted) relation R_2 will all be larger. We can also avoid unnecessary set comparisons by checking the largest elements in the set-valued attributes and comparing the size of the attributes. Finding the smallest and largest element in an attribute can be done in constant time as we have sorted the attributes and store the size explicitly. We summarize these ideas in the following code fragment.

```
sort each set in R1 and R2;
```

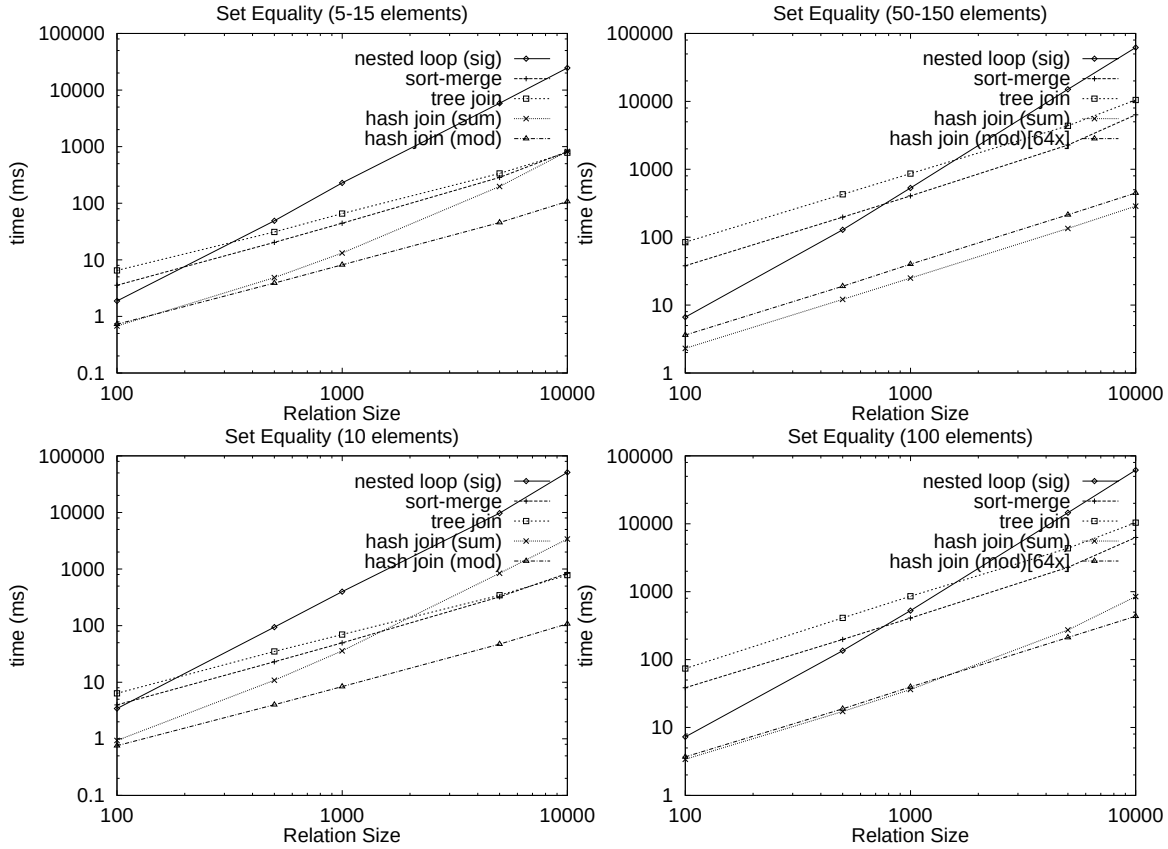


Figure 4.7: Performance of different join algorithms (for equality predicates)

```
sort R2 lexicographically;

for(i = 0; i < size of R1; i++) {
  for(j = 0; j < size of R2; j++) {
    if(smallest element of R1[i].a < smallest element of R2[j].b) {
      break;
    }
    if(largest element of R1[i].a > largest element of R2[j].b) {
      continue;
    }
    if(size of R1[i].a > size of R2[j].b) {
      continue;
    }
    if(R1[i].a is subset of R2[j].b) {
      build result tuple;
    }
  }
}
```

Unlike the sort-merge variant for equality predicates this algorithm has quadratic running

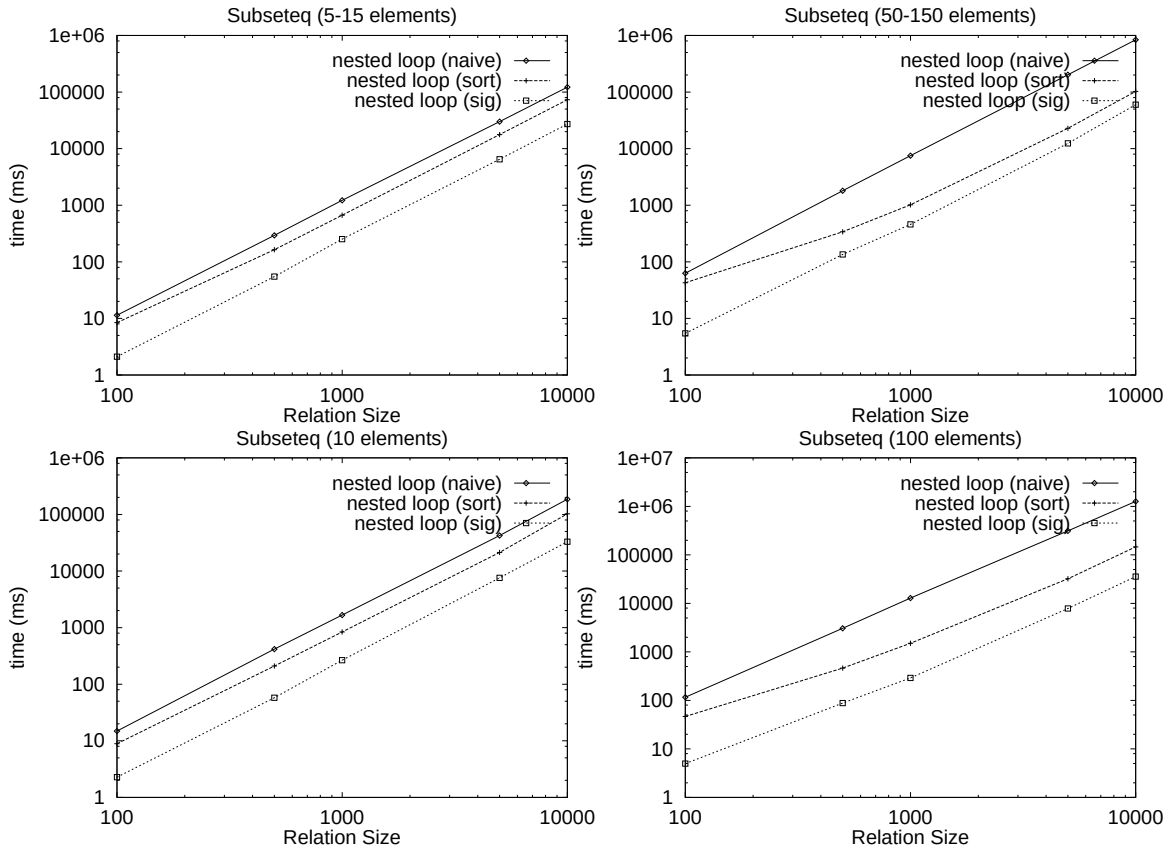


Figure 4.8: Performance of different nested-loop join algorithms (for subset predicates)

time ($O(|R_1| \cdot |R_2|)$ instead of $O(|R_1| \cdot \log |R_1| + |R_2| \cdot \log |R_2|)$). Nevertheless, with the added pretests that avoid expensive set comparisons we expect a better performance for this algorithm than for the naive evaluation of joins with subset predicates.

4.3.3 Tree Join

The tree-join algorithm for equality predicates can be easily adapted to joins with subset predicates. The procedure to insert tuples into the tree structure does not need to be changed at all. We modify the *find* procedure as follows. As we are looking for supersets of the set-valued attribute of the query tuple, we have to find a node in the tree for each element of the attribute. Contrary to set equality the set-valued attributes of the tuples we are searching for may contain additional elements.

Again we start at the root of the tree with the first element of the attribute of the query tuple. If we reach a node that has a key that is smaller than the current query element, we have found an additional element that is not present in the query set. In this case we have to follow the *match* and *nomatch* edge. If we reach a node with a key equal to the current query element, we check whether this is the last element we are looking for. If this is the case, we return the tuple lists of the current node and of all descendants of the current node. Otherwise we have to continue searching for the remaining elements of the

query tuple in the *match* branch. Finally if the key of the current node is larger than the current element, we know that the node we are looking for has never been inserted into the tree and we can stop searching in this branch of the tree. The following procedure is called with i equal to the index of the first element of $t.a$ and *currentNode* equal to the root of the tree.

```
list find(tuple t, int i, treeNode* currentNode) {
  if(currentNode == NULL) {
    return empty list;
  }
  if(currentNode->key < i-th element of t.a) {
    result = find(t, i, currentNode->match);
    concatenate find(t, i, currentNode->nomatch) to result;
    return result;
  }
  if(currentNode->key == i-th element of t.a) {
    if(last element of t.a) {
      result = currentNode->tuplelist;
      concatenate tuplelists of all descendants
                  of currentNode to result;
      return result;
    }
    return find(t, i + 1, currentNode->match);
  }
  /* currentNode->key > i-th element of t.a */
  return empty list;
}
```

It is important to note that the *find* procedure in most cases traverses more than one path in the tree. In the worst case we have to visit each node of the tree. The complete join algorithm proceeds as follows: first, the tree is built for the inner relation. Then, for each tuple in the outer relation, the *find* procedure is used to retrieve the joining tuples. For both relations, the set-valued attributes are sorted beforehand. Sorting the relations lexicographically on their join attributes is not necessary.

4.3.4 Signature-Hash Join

While the concept of hashing is well suited to joins with equality predicates, applying this concept to the evaluation of joins with subset predicates is not straightforward. Principally we have two ways to implement a hash join for subset predicates.

On one hand we could insert every tuple t of relation R_2 into a hash table redundantly, i.e. for every subset of $t.b$ t is inserted into the hash table with the hash key of the corresponding subset. During query evaluation we would then search with an ordinary equality predicate. In view of the exponential storage overhead and the dynamic allocation of the hash table we abandon this idea.

On the other hand we could leave it at inserting each tuple into the hash table once. During query evaluation we would have to generate the hash keys of all supersets of a given query set, which seems infeasible. Therefore we switch the inner and outer relations for hash joins, i.e., we compute $R_2 \bowtie_{R_2.b \supseteq R_1.a} R_1$ instead of $R_1 \bowtie_{R_1.a \subseteq R_2.b} R_2$. So we transform the problem of finding supersets of a given set to finding subsets. The number of subsets of a given set s, which is 2^s, is usually much smaller than the number of supersets of s, which is $2^{(D_s - s)}$ (D_s being the domain of the elements of s). Generating the hash keys of all subsets of a given set seem to result in an exponential runtime. Nevertheless, this approach is practicable because we use signatures to derive hash keys and not the elements of the sets (we elaborate on the precise costs later while discussing Formula (4.2)). If we can keep the signatures small enough, we are able to generate all subsets of the signatures fast enough by using the idea of Vance and Maier (see Section 2.2). The size of the signatures depends on the size of the given sets, so we have the problem of keeping signatures small for large sets. We solve this problem by using only part of a signature, more specifically the lowest d bits. We call this the *partial signature* of a set s, or abbreviated $partsig_d(s)$. This corresponds to the fixed prefix/suffix partitioning technique for signatures [64], except that we neglect the rest of the signature. The partial signature of its set-valued attribute is used as a hash key for a tuple. Given that the size d of a partial signature is crucial to the performance of the hash join algorithm, we address this issue after describing the join algorithm.

The Join Algorithm

We start by inserting the tuples t_i in R_1 into a hash table with hash keys equal to $partsig_d(t_i.a)$. For each tuple t_j in R_2 we generate the hash keys of all subsets of $partsig_d(t_j.b)$ and probe for matching tuples of R_1 in the hash table. We compare the full signature of the current tuple of R_2 with the full signatures of the tuples retrieved from the hash table. If the full signatures indicate a match, we compare the set-valued attributes directly to eliminate false drops.

When mapping partial signatures directly onto hash keys, the size of the hash table is restricted to powers of 2. If a hash table of another size, say n, is needed, we cannot employ this direct mapping. In this case we use $partsig(s)$ modulo n as hash key.

Tuning the parameters

As we have already mentioned, the size of partial signatures is a crucial parameter for the performance of the hash-join algorithm. The first step in optimizing partial signatures is to optimize the full signatures. Since we set only one bit per set element in a signature ($k = 1$) the false drop probability is controlled by the signature length b. If we choose a value of b that is too small, almost all bits in the full signature and partial signature will be set. On the other hand if we choose a value of b that is too large, we expect very sparsely filled signatures. Both of these cases lead to long collision chains in the hash table. The optimal value for b can be calculated by using formula 2.10 from Section 2.1.3. For our special case ($k = 1$) it translates to

$$b_{opt} = \frac{|R_2.b|}{\ln 2} \tag{4.1}$$

where $|R_2.b|$ is the average cardinality of the join attribute of relation R_2. Using the value of b_{opt} leads to signatures in which on average half of the bits are set. From the viewpoint of information theory these signatures contain maximal information. We can verify the theoretical findings with some experiments. For a set cardinality of 100 elements the value of b_{opt} is approximately 144 bits (according to (4.1)). Provided that on average half of the bits are set in a full signature, we also expect that half of the bits are set in a partial signature. The left hand side of Figure 4.9 shows the results for some partial signature sizes. We can clearly see that for the optimal full signature size b_{opt} about half of the bits are also set in the partial signatures. We also investigated the influence of the signature sizes on the collision chain lengths of the hash tables (right hand of Figure 4.9). For small full signature sizes the collision chains are quite long. With increasing signature sizes the lengths of the collision chains decrease. After passing the optimal point the chains slowly grow in length again. We have some leeway in choosing the full signature size, values from 100 to 200 all seem acceptable. Using slightly larger signatures seems attractive as less than half of the bits are set in partial signatures. On average fewer subsets would have to be generated and consequently fewer lookups would be needed. Although we would have more collisions, the collision chain length grows only slowly for increasing signature sizes (Figure 4.9). Experiments showed that using signatures that are 30% larger than b_{opt} resulted in running times that were 20% to 50% lower than those for b_{opt}. One more effect can be seen in Figure 4.9. As expected increasing the partial signature size decreases the collision chain length because we have a wider range of hash keys.

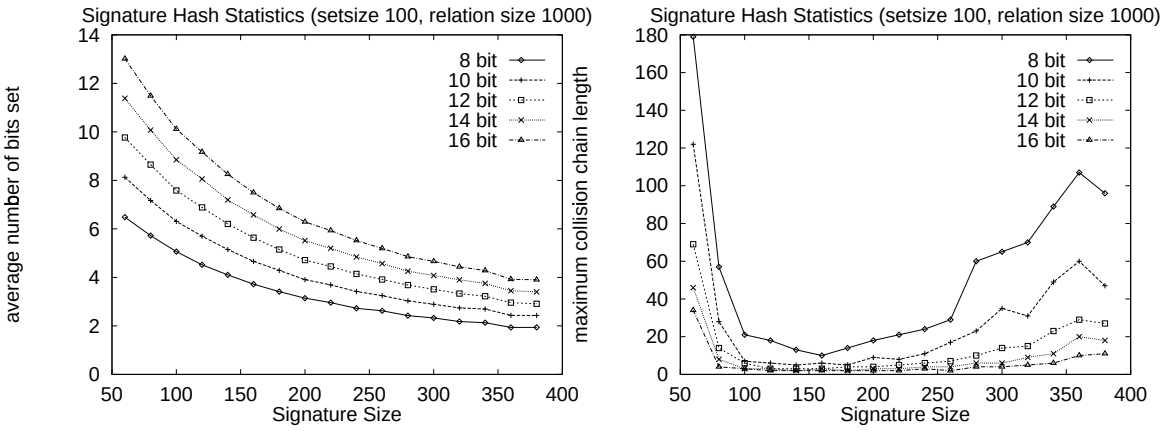


Figure 4.9: Tuning partial signature sizes (for subset predicates)

Next we wanted to find out how to choose d, the partial signature size. We want to keep it low to be able to rapidly generate and lookup subsets during query evaluation. Small partial signatures, however, lead to longer collision chains (witness Figure 4.9). So we have to balance these two factors.

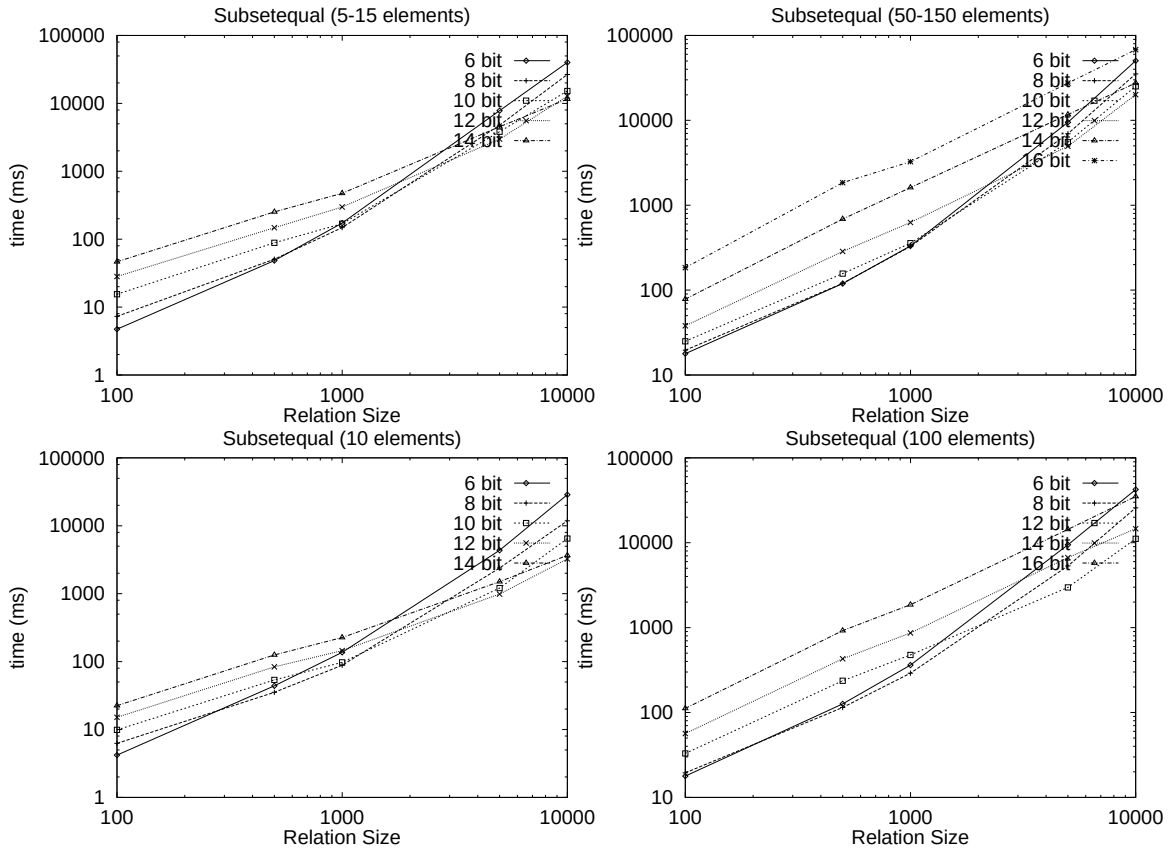


Figure 4.10: Performance of the hash join depending on the partial signature size (for subset predicates)

Figure 4.10 illustrates the experimental results using different partial signatures sizes. Small partial signature are very fast for small relations as few lookups with few collisions occur. For large relations, however, these partial signatures are not sufficient since the number of collisions increases dramatically. For larger relations larger partial signatures become more interesting. In general the partial signature size should be the largest d such that 2^d is approximately equal to the cardinality of the hashed relation. The theoretical costs $C(d)$ for retrieving all matching tuples from a hash table for a given query tuple are proportional to the number of lookups times the (average) length of the collision chains. On average $\frac{d}{2}$ bits are set in a partial signature. Assuming simple uniform hashing, one lookup takes time $\Theta(1 + \frac{n}{m})$ where n is the number of tuples and m the size of the hash table [18]. So $C(d)$ can be approximated by

$$\begin{aligned} C(d) &\approx 2^{\frac{d}{2}} \cdot \left(1 + \frac{|R|}{2^d}\right) \\ &= 2^{\frac{d}{2}} + \frac{|R|}{2^{\frac{d}{2}}} \end{aligned} \tag{4.2}$$

where $|R|$ is the cardinality of the hashed relation. These costs are minimal if $d = \log_2 |R|$

(that is $2^d = |R|$). For a proof see Appendix A. Inserting this optimal value into (4.2), we get:

$$\begin{aligned} C(\log_2 |R|) &= \left(2^{\log_2 |R|}\right)^{\frac{1}{2}} + \frac{|R|}{\left(2^{\log_2 |R|}\right)^{\frac{1}{2}}} \\ &= \sqrt{|R|} + \frac{|R|}{\sqrt{|R|}} \\ &= 2 \cdot \sqrt{|R|} \end{aligned} \tag{4.3}$$

As we can see, the costs only look exponentially at first glance. If the size of the partial signatures is carefully chosen, the lookup costs are actually proportional to the square root of the cardinality of the relations.

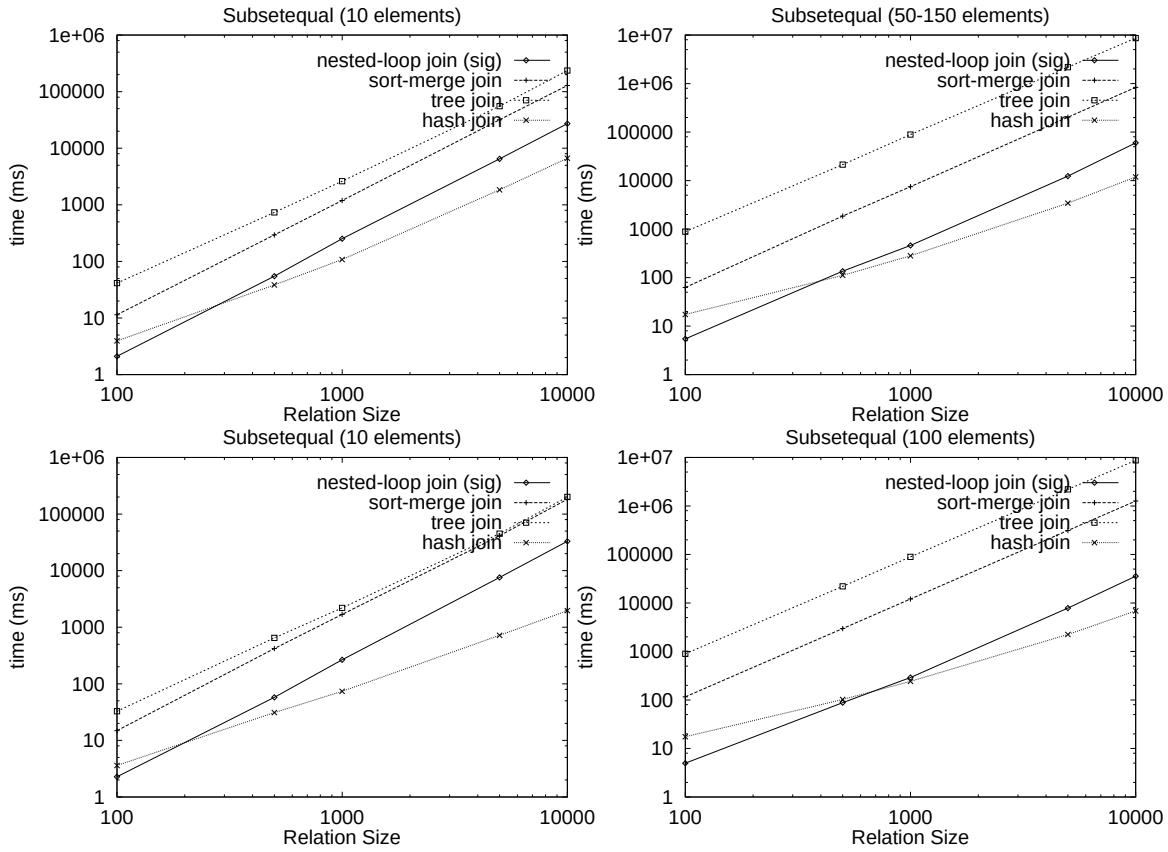


Figure 4.11: Performance of different join algorithms (for subset predicates)

4.3.5 Comparison of Algorithms for Subset Predicates

We compared the nested-loop, sort-merge, tree-, and hash join algorithms experimentally. The results of these experiments can be seen in Figure 4.11. The first thing that strikes

us is the performance of the nested-loop join compared to the performance of the other algorithms. The nested-loop algorithm itself has not changed much when compared to the version for equality queries. The performance of the other algorithms, however, has deteriorated significantly, so nested-loop join becomes competitive again. The numerous modifications to sort-merge and tree join to adapt them to subset predicates have influenced their performance severely. Hash join also does not perform as well as for equality predicates, but is still able to beat nested-loop join for large relations.

4.4 Conclusion and Outlook

We proposed and investigated several different main memory join algorithms for set equality and subset predicates. We have found efficient alternatives to the naive nested-loop algorithm. For set equality predicates the hash-based algorithms proved to be superior to all other algorithms. For subset predicates the hash join is still on top, while nested-loop join is only competitive for small relations. This is, however, not the naive variant but a modified and tuned algorithm based on signatures.

Although this has been an important first step in filling a gap, there are still interesting questions left for future research. Some of these include set-valued join algorithms with different predicates (e.g. set intersection), algorithms for secondary storage, and parallelization of the join algorithms.

Chapter 5

Diag-Join: A Join Algorithm for 1:N Relationships

Online analytical processing (OLAP) plays an ever more important role in the world of information processing. In contrast to online transaction processing (OLTP) it involves complex queries requiring the scanning of huge amounts of data, but seldomly updating or deleting. On account of the different requirements, special database systems called Data Warehouses have been developed. During the evaluation of queries in Data Warehouses, relations containing millions or even billions of tuples need to be joined. Joining these large relations (usually the central fact tables in Data Warehouses) is very costly. Evidently, fast join algorithms are very important in this environment.

The main strategy for lowering join costs is to filter out many non-qualifying tuples beforehand. Bit-vector indexing is predominantly used for this purpose, like in O'Neil's and Graefe's multi-table join [72]. However, it is not always possible to filter out a significant number of tuples. The join attribute may also take on many different values, leading to huge bit-vectors, so that the overhead of filtering does not pay off. We were wondering if properties of relations exist that can be exploited during a join operation. During our analysis we made the following observations. When inserting new tuples into a Data Warehouse, those tuples are usually appended to existing relations [50, 58]. Therefore time of creation is the predominant—though often implicit—clustering strategy. Another important observation was that in the context of data-warehousing, relations are typically joined on foreign keys [50, 58]. Backed by these observations, we developed a join algorithm—called *Diag-Join*— which takes advantage of these facts [48]. It exploits time-of-creation clustering for 1:n relationships.

Let us illustrate these two points by an example taken from [58]. All companies selling products have to ship these products to their customers. Assume that in the Data Warehouse of such a company a central fact table *Shipments* exists, that contains the data on all deliveries made. In a dimensional table *CustomerOrders* we store information on all orders that the company received. See Figure 5.1 for an illustration. Soon after appending an order from a customer, we expect the corresponding tuples to be added to *Shipments*, resulting in clustering by time of creation.

The Diag-Join exploits this clustering. In essence, Diag-Join is a sort-merge join without the sort phase. An important difference, however, is that the merge phase of Diag-Join

Shipments				
ProdKey	*Price*	*ShipDate*	*ShipMode*	*OrderNo*
123	24.00	10/12/96	Mail	K-323
234	35.00	10/13/96	Air	K-323
012	97.00	10/13/96	Air	K-323
635	1298.00	11/23/96	Truck	K-326
534	453.00	11/23/96	Truck	K-326
239	20.00	12/10/96	Air	K-351
978	10000.00	12/18/96	Rail	K-351
174	35000.00	12/20/96	Ship	K-351
...	...	...	...	...

CustomerOrders			
OrderNo	*CustomerID*	*TotalPrice*	*OrderDate*
K-323	1943	156.00	10/10/96
K-326	432	1751.00	11/20/96
K-351	129	45020.00	12/02/96
...	...	...	...

Figure 5.1: The relations *Shipments* and *CustomerOrders*

does not assume that the tuples of either relation are sorted on the join attributes. Instead, it relies on the physical order created by the (implicit) time-of-creation clustering strategy. More specifically, Diag-Join joins the two tables by scanning them simultaneously. The scan on the outer relation proceeds by moving a sliding window of adjustable size over the relation. Only within this window do we search for join partners for the inner relation. A special mechanism takes care of those tuples of the inner relation for which no join partner could be found in the window. They are called *mishits*. Though simple, this idea proves to be very effective. There are, however, some subtleties that are addressed later on. These are the buffer management, the window size, the organization of the window, and the sliding speed of the window. We also present a method which allows Diag-Join to join non-base relations (resulting from intermediate operations).

Diag-Join has two advantages over other join algorithms for appropriately clustered relations:

- Even if the relations do not fit into main memory, in many cases Diag-Join will be able to avoid the creation of large temporary files, unlike the sort-merge join [8], the hybrid hash join [22, 85], and the GRACE hash join [32, 85].
- Contrary to other join algorithms, output tuples can be produced right away without an interruption of the query evaluation pipeline.

The rest of this chapter is organized as follows. We present the Diag-Join algorithm in the next section. Section 5.2 contains performance evaluations and comparisons with blockwise nested-loop join, GRACE hash join, and index nested-loop join (for a brief sketch of these join algorithms, see Chapter 3). Section 5.3 concludes the chapter.

5.1 The Diag-Join

The first subsection briefly summarizes some preliminaries and notations used throughout the rest of the chapter. We then present a basic version of the Diag-Join explaining the principle of the algorithm in subsection 5.1.2. We proceed by giving an advanced version

Symbol	*Definition*
R_1	(smaller) relation to be joined
κ	key of relation R_1
R_N	(larger) relation to be joined (with foreign key κ)
$\|R_x\|$	cardinality of relation R_x in number of tuples ($x \in \{1, N\}$)
$\|\|R_x\|\|$	size of relation R_x in number of pages
$R_x[i]$	tuple at position i in relation R_x, $1 \leq i \leq \|R_x\|$
t	an arbitrary tuple
m_t	size of buffer/window in number of tuples
m_p	size of buffer/window in number of pages
l	size of array of hash tables
p	hash table size in pages ($= \lfloor \frac{m_p}{l} \rfloor$)
$interOp(R_x)$	intermediate operator on R_x

Table 5.1: Used symbols

of the algorithm illuminating implementation details in 5.1.3. We deal with the subtleties mentioned in the introduction. Further, we discuss how to join non-base relations (resulting from intermediate operations). The last two subsections contain a cost model and the derivation of formulas to calculate the *mishit probability* (i.e. the probability that a tuple turns out to be a mishit).

5.1.1 Preliminaries

For the rest of the chapter we use the symbols listed in Table 5.1. Given two relations R_1 and R_N to be joined, we assume that R_1 contains the key κ, which is a foreign key of R_N. That is, a 1:n relationship exists between R_1 and R_N. $|R_x|$ denotes the cardinality (in number of tuples) of a relation R_x (with $x \in \{1, N\}$), while $||R_x||$ stands for the size of R_x in pages. We further assume that the tuples in each relation are (implicitly) numbered by their physical occurrence. The i-th tuple in R_x is denoted by $R_x[i]$ with $1 \leq i \leq |R_x|$.

Let us assume that a tuple of R_1 and all matching tuples in R_N are created by the same transaction and are written to disk at the same time. We can easily figure out the physical position of the joining tuple in R_1 for a given tuple in R_N. We call this situation "perfect" clustering by time of creation. In the special case of 1:n relationships, i.e. every tuple in R_N joins exactly with one tuple from R_1, we expect for each tuple $R_N[i]$ to find the matching tuple in R_1 at position $\lceil \frac{i}{|R_N|/|R_1|} \rceil$. If the number of join partners of each tuple in R_1 varies, the calculated position is only an approximation. Figure 5.2 illuminates a perfect situation. On the x-axis we have the positions of the tuples in R_N, on the y-axis

the expected positions of their join partners in R_1. Here, each tuple in R_1 joins with exactly two tuples from R_N. Hence, the join partner of $R_N[5]$ is $R_1[3]$, because $\left\lceil \frac{5}{8/4} \right\rceil = 3$. It is important to note that, even for perfect clustering, the relations will almost certainly not be sorted on the join attributes.

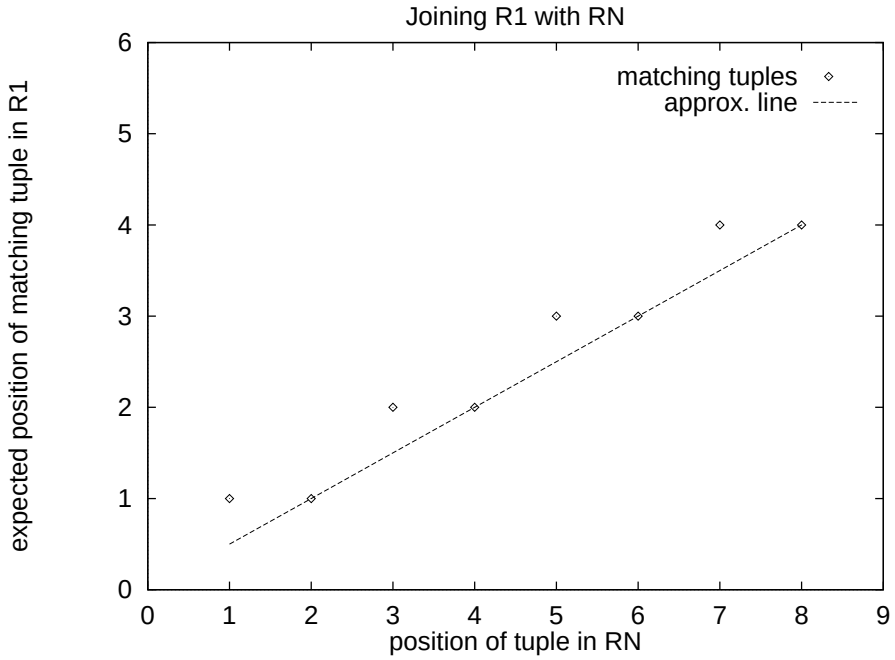


Figure 5.2: Expected positions of matching tuples

5.1.2 Basic Diag-Join

If the tuples in the relations are perfectly clustered, then a simple merge phase suffices to join the two relations. However, in reality this is not always the case. There may be some exceptions, because the number of join partners for each tuple in R_1 varies, the tuples are not inserted simultaneously into R_1 and R_N, or they are reorganized later (e.g. deletion of tuples, insertion of additional tuples, replacements). Hence we do not just look at one tuple of R_1 at a time, but hold m_t tuples—those in the vicinity of the expected position—in a buffer. We call the part of R_1 held in the buffer a *window* on R_1.

The basic Diag-Join algorithm works as follows. We initialize the window with $\left\lceil \frac{m_t}{2} \right\rceil$ tuples from $R_1[1]$ to $R_1[\left\lceil \frac{m_t}{2} \right\rceil]$. We expect the matching tuple for $R_N[1]$ to be at $R_1[1]$ or in the range from $R_1[-\left\lfloor \frac{m_t}{2} \right\rfloor]$ to $R_1[\left\lceil \frac{m_t}{2} \right\rceil]$. Since there are no negative positions in R_1, the interval from $-\left\lfloor \frac{m_t}{2} \right\rfloor$ to 0 is cut off. Then R_N is scanned sequentially starting with $R_N[1]$. No buffering is applied to R_N, except for the current tuple. For every tuple $R_N[i]$ we search the window for a matching tuple from R_1. If the lookup is successful (we call this a *hit*), we immediately produce an output tuple and go on to the next tuple in R_N. We can do this, because there can be at most one hit (1:n relationship). If the lookup fails

```
Diag-Join(R_1, R_N, m_t) {
   /* phase 1 */

   ratio  = |R_N| / |R_1|;
   curTup = m_t/2;
   fill buffer with R_1[1] to R_1[curTup];
   for(i = 1; i <= |R_N|; i++) {
      if(tuple t in buffer matches R_N[i]) {
         join t with R_N[i];
         output result;
      }
      else {
         write R_N[i] to tmpfile;
      }
      if(i % ratio == 0) {
         curTup++;
         if(buffer is full) {
            replace tuple with lowest position with R_1[curTup];
         }
      }
   }

   /* phase 2 */

   join R_1 with tmpfile using any standard join algorithm;
}
```

Figure 5.3: Basic Diag-Join algorithm

(called *mishit*), then $R_N[i]$ is written into a temporary file. Whenever $|R_N|/|R_1|$ tuples from R_N have been processed, we add the next tuple from R_1 to the window. If there is no free space left in the window, we replace the tuple with the lowest position. When we have finished scanning R_N, we join the tuples in the temporary file (which should be much smaller than $||R_N||$) with R_1 using some standard join algorithm. Figure 5.3 gives a summary of the basic Diag-Join algorithm.

Before presenting a more elaborate version of Diag-Join, let us briefly highlight some problems of the basic version. First, the algorithm is not very efficient, because it uses a tuple-oriented buffer, while most DBMSs use page-oriented structures. Second, the organization of the window is crucial for the efficiency and needs to be discussed. Third, the algorithm only works on base relations, e.g. no selections prior to the join are possible. We resolve these problems in the next section.

5.1.3 Advanced Diag-Join

We kept the algorithm in the last section very simple, because we intended to illustrate the basic principle of the algorithm. The implementation details are presented in this section.

Page-oriented Buffer

We change from a tuple-oriented buffer to a page-oriented buffer. We do not read single tuples into the window, but all tuples on the next p pages, which is much more efficient. We call p the *step size* of Diag-Join. As a consequence, we replace tuples in the window whenever $p \cdot ||R_N||/||R_1||$ pages have been scanned in R_N.

Hashing the Window

Searching the window sequentially for matching tuples is too expensive; therefore we use hash tables to look up join partners in the window. We have two alternatives to organize these hash tables. On one hand we can use one large hash table with a size of m_p pages, on the other hand an array of l hash tables with a size of $\left\lfloor \frac{m_p}{l} \right\rfloor$ pages each. Using only a single hash table is disadvantageous. If we apply a step size p equal to the window size m_p, we also replace a part of the vicinity inserted during the last step that is needed in the current step (i.e. the scrolling of the window on R_1 is very coarse). If we apply a step size p smaller than the window size m_p, we have to replace the oldest tuples in the hash table with those just read. Organizing a hash table to allow the deletion of individual tuples according to time of insertion is burdensome. Therefore we allocate an array of l hash tables. Each hash table has a size equal to $\left\lfloor \frac{m_p}{l} \right\rfloor$. We equate the hash table size with the step size, hence $p = \left\lfloor \frac{m_p}{l} \right\rfloor$. Then in each step we free an entire hash table, which is much cheaper than deleting individual entries. Figure 5.4 depicts an exemplary window organization. The window size is six pages, organized into three chunks of two pages each. Therefore the step size is also equal to two pages. The broken lines indicate how the pages are replaced when no free buffer space is left.

After describing the organization of the window let us now look at the algorithm. Sliding the window is done as follows. Whenever $p \cdot ||R_N||/||R_1||$ pages have been scanned in R_N, the least recently loaded hash table is cleared. Then the next p pages from R_1 are loaded into this hash table. How do we look up matching tuples in the hash table array? First of all we search the middle table at position $\left\lceil \frac{l}{2} \right\rceil$ in the array. If R_1 and R_N are perfectly clustered, we expect to find the matching tuple in this table. If we are not able to find it there, we search the table at position $\left\lceil \frac{l}{2} \right\rceil + 1$. On failure the tables at positions $\left\lceil \frac{l}{2} \right\rceil - 1$, $\left\lceil \frac{l}{2} \right\rceil + 2$, $\left\lceil \frac{l}{2} \right\rceil - 2$, and so on are searched. We call this technique *zig-zag search*. This is the best technique when the deviation of the relations from perfect clustering can be described by a normal distribution (see Figure 5.7 in Section 5.1.5). If the matching tuple is found, we join the tuples immediately and output the result. Otherwise the tuple from R_N is written into a temporary file. To speed up the algorithm, we could hold the mishits in a main memory buffer. Only if this buffer overflows, we flush it to disk. We

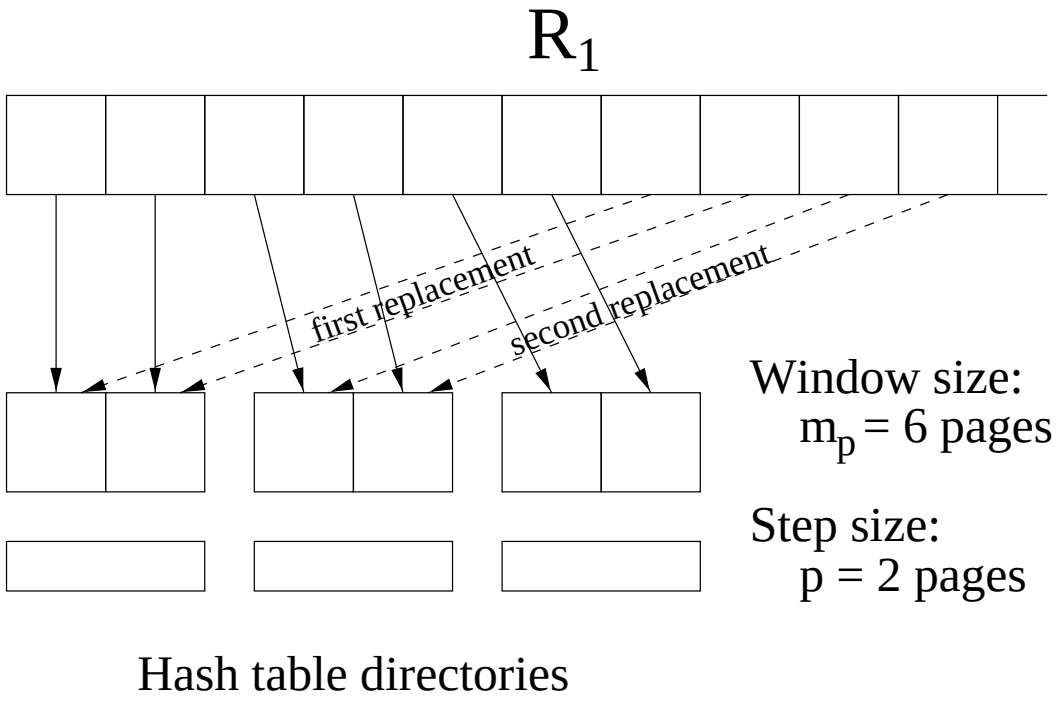


Figure 5.4: Window organization for Diag-Join

also recommend to use an odd number for l, so that the searching range for the lookups is symmetrical.

Joining non-base relations

We have to take special care when joining non-base relations. If we feed tuples from intermediate operators (working on R_N) straight into a Diag-Join operator, this may destroy the synchronization, i.e. we may slide the window on R_1 incorrectly. We solve this problem by using the Observer pattern described in [33]. The intent of the Observer pattern (also known as publish-subscribe) is to notify all dependent objects $o_1, o_2, \ldots, o_n$ of a state change in an object s. For a description in C++ notation see Figure 5.5.

The methods *attach* and *detach* connect and disconnect objects to a subject object s. When s changes its state, it calls the method *notify* which in turn calls the method *update* of all observer objects currently attached. In our case the operator accessing the tuples from R_N (scan, index scan, etc.) notifies Diag-Join about the position within R_N from which the current tuples are fetched. Then Diag-Join is able to slide the window with the right speed or even skip some pages of R_1. Note that this technique allows any intermediate operator to occur between the scan on R_N and Diag-Join, as long as it preserves the relative order of the tuples. The algorithm is summarized in Figure 5.6. Please note that the current middle table is not always at position $\left\lceil \frac{l}{2} \right\rceil$, because we reuse the hash tables in the array.

A similar technique can also be applied to handle intermediate operators between the scan

```
class Observer
{
   update(Subject*);
}


class Subject
{
   attach(Object*);
   detach(Object*);
   notify();
}
```

Figure 5.5: Observer Pattern

on R_1 and Diag-Join. When loading a hash table during the advancement of the window, it is always filled completely. If an intermediate operator discards many tuples, the scan on R_1 may hurry ahead in order to fill the hash table. If the scan on R_1 notifies Diag-Join of the positions of the currently scanned tuples, Diag-Join will be able to recognize this case. As a consequence, Diag-Join will delay the window sliding on R_1 until the scan on R_N has caught up. We would also like to draw attention to the fact that the tuples of R_1 that enter the Diag-Join operator need to be stored in a temporary relation for the second phase of the algorithm. Otherwise the scan on R_1 and all operations between it and the Diag-Join would have to be repeated.

The technique of modifying the sliding speed is not as effective on the R_1 branch of the join as on the R_N branch, however. If a tuple in R_1 is filtered out between the scan and the Diag-Join, it will not be placed into the hash table. Consequently potential matching tuples in R_N will not find it and be classified as mishits. We will not find out that they do not belong in the resulting relation until the second phase of the algorithm. If this happens to many tuple in R_N, we do a lot of unnecessary work while trying to join these tuples. A possible solution to this problem is to mark the filtered out tuples of R_1 as deleted but nonetheless pass them on to the other operators. Then in the Diag-Join operator, if a tuple in R_N finds a matching partner that is marked as deleted, it can be discarded and need not be treated as a mishit. When writing the tuples of R_1 to a temporary relation on disk for the second phase, we do not need to consider the tuples that are marked as deleted unless they are needed in yet another Diag-Join operator further on.

5.1.4 Cost model

Our cost model for Diag-Join is based on the cost models presented in [42]. The parameters needed for the cost model are shown in Table 5.2. The cost $C_{I/O}$ for transferring a set of

```
Diag-Join(R_1, interOp(R_N), m_p, l) {
   /* phase 1 */

   ratio = |R_N| / |R_1|;
   allocate array arr[l] of hash tables;
   fill arr[1] to arr[l/2] with tuples from R_1;
   do {
      t_N = next tuple from interOp(R_N);
      zig-zag search hash tables for matching tuple;
      if(matching tuple found) {
         join tuples;
         output results;
      }
      else {
         write t_N to tmpBuf;
      }
      if(notified from access operator on base relation R_N) {
         if(all hash tables are full) {
            clear least recently loaded hash table;
            load next pages from R_1 into cleared hash table;
         }
      }
   } while (tuples from interOp(R_N) remain);

   /* phase 2 */

   join R_1 with tmpBuf using any standard join algorithm;
}
```

Figure 5.6: Advanced Diag-Join algorithm

$||R_x||$ pages from disk to memory, or vice versa, through a buffer of size B_x is given by

$$C_{I/O}(||R_x||, B_x) = \left\lceil \frac{||R_x||}{B_x} \right\rceil \cdot T_k + ||R_x|| \cdot T_t \tag{5.1}$$

where T_k is the sum of the average seek and latency time and T_t is the cost for transferring a page between disk and memory.

The costs for Diag-Join consist of the costs for the first and the second phase.

$$C_{DIAG}(R_1, R_N) = C_{Phase1} + C_{Phase2} \tag{5.2}$$

In the first phase we have to read R_1 and R_N, hash all tuples of R_1, look for matching

Symbol	*Definition*
$C_{I/O}$	cost for transferring pages between disk and memory
B_x	arbitrary buffer
T_k	sum of average seek and latency time
T_t	time for transfer of one page
T_c	time for hashing a tuple
T_j	time for finding the join partner of a tuple

Table 5.2: Parameters for cost model

tuples and join them or write the mishits to disk.

$$\begin{aligned} C_{Phase1} &= C_{Read\ R_1} + C_{CreateHash} + C_{Read\ R_N} + \\ &\quad C_{Join} + C_{Write} \end{aligned} \tag{5.3}$$

The individual parts of C_{Phase1} are defined as follows:

$$C_{Read\ R_1} = C_{I/O}(||R_1||, p) \tag{5.4}$$
$$C_{CreateHash} = |R_1| \cdot T_c \tag{5.5}$$
$$C_{Read\ R_N} = C_{I/O}(||R_N||, 1) \tag{5.6}$$
$$C_{Join} = |R_N| \cdot T_j \tag{5.7}$$
$$C_{Write} = C_{I/O}(||tmpFile||, 1) \tag{5.8}$$

The costs in the second phase depend on the join algorithm used. In our case we applied GRACE hash join in the second phase (for cost models of GRACE hash join see [40, 42]), hence

$$C_{Phase2} = C_{GRACE}(R_1, tmpFile) \tag{5.9}$$

Even though we present an estimation for normally distributed tuples in Section 5.1.5 approximating $||tmpFile||$ will not be a trivial task. As the assumption of normally distributed tuples is probably not valid for all applications we recommend the following procedure. During times of low workload (or an issued run-stat command) a shortened version of the first phase of Diag-Join is processed. This shortened version is used to determine $||tmpFile||$ without actually creating the temporary file or any result tuples. Details on the estimation of the size of the temporary file and the shortened version of Diag-Join are subject of Section 5.1.5.

The query optimizer of a DBMS needs to be supplied with the above cost model and its parameters (especially an estimation of $||tmpFile||$) to enable it to make a decision about

Symbol	*Definition*
$N(a, b, \mu, \sigma)$	normal distribution
$n(x, \mu, \sigma)$	density function of normal distribution
$j(i)$	expected position of matching tuple
$m_{lo}(i)$	start position of middle hash table
$m_{hi}(i)$	end position of middle hash table
$w_{lo}(i)$	start position of window ($w_{lo}(i) \leq m_{lo}(i)$)
$w_{hi}(i)$	end position of window ($m_{hi}(i) \leq w_{hi}(i)$)
h_t	average number of tuples per hash table

Table 5.3: Parameters for mishit probability

the application of Diag-Join. The costs for joining base relations can be approximated by using (5.2) without modifications. If order-preserving intermediate operators occur, the standard techniques of the optimizer to estimate the costs of complex queries have to be applied (e.g. calculating the cardinalities of the intermediate relations and the size of Diag-Join's temporary files (tmpFile) with the help of selectivities).

5.1.5 Calculating the mishit probability

In this section we derive a formula for calculating the *mishit probability*, that is, the probability that an arbitrary tuple from R_N turns out to be a mishit. Table 5.3 summarizes the needed parameters.

With the help of this probability the size of the temporary file can be estimated:

$$|tmpFile| \approx Pr_{avg}(R_N[i]\text{is a mishit}) \cdot |R_N| \tag{5.10}$$

As already mentioned, we assume that the derivation of the relations from perfect clustering can be described by a normal distribution. The normal distribution $n(x, \mu, \sigma)$ with mean μ and standard deviation σ is defined as follows.

$$n(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \tag{5.11}$$

We also need to know the probability that x is in the range between a and b. This can be calculated by the distribution $N(a, b, \mu, \sigma)$.

$$N(a, b, \mu, \sigma) = \int_a^b n(x, \mu, \sigma)dx \tag{5.12}$$

Let us illustrate the concept of normally distributed tuples. For the tuple $R_N[i]$ at position i ($1 \leq i \leq |R_N|$) in relation R_N, we expect to find the matching tuple at position

$j(i) = \left\lceil i \cdot \frac{|R_1|}{|R_N|} \right\rceil$ in relation R_1, if the relations are perfectly clustered. There may be some deviation, however, as indicated by the bell-shaped curve in Figure 5.7. The curve indicates the probability that the matching tuple can be found at a certain position around $j(i)$ in R_1. The middle hash table in the window starts at position $m_{lo}(i)$ and ends at position $m_{hi}(i)$ (h_t is the average number of tuples per hash table):

$$m_{lo}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil - 1 \right) \cdot h_t + 1 \tag{5.13}$$

$$m_{hi}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil \right) \cdot h_t \tag{5.14}$$

$w_{lo}(i)$ and $w_{hi}(i)$ are the smallest and largest positions of the elements found in the window, respectively (we assume that l is odd):

$$w_{lo}(i) = m_{lo}(i) - \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \tag{5.15}$$

$$w_{hi}(i) = m_{hi}(i) + \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \tag{5.16}$$

Please note that for a better readability we have refrained from covering the special cases at the start and end of R_1.

The probability that $R_N[i]$ turns out to be a mishit is the probability that the matching tuple is not inside the window:

$$Pr(R_N[i] \text{ is a mishit}) = 1 - N(w_{lo}(i), w_{hi}(i), j(i), \sigma) \tag{5.17}$$

When scanning through R_N this probability changes, because $j(i)$ moves through the middle hash table from $m_{lo}(i)$ to $m_{hi}(i)$. Whenever $j(i)$ reaches $m_{hi}(i)$ the window slides down by the specified step size.

We are interested in attaining a mishit probability below a threshold value p_{accept}. This is tantamount to limiting the size of the temporary file. How large do we have to choose the window size m_t (and the step size h_t) to guarantee $Pr_{avg}(R_N[i] \text{ is a mishit}) \leq p_{accept}$? The mishit probabilities of the tuples in R_N repeat themselves for each window as $j(i)$ passes from $m_{lo}(i)$ to $m_{hi}(i)$. So the average mishit probability can be approximated by

$$Pr_{avg}(R_N[i] \text{ is a mishit}) = \sum_{j=\lfloor \frac{l}{2} \rfloor \cdot h_t + 1}^{\lceil \frac{l}{2} \rceil \cdot h_t} \frac{1 - N(1, m_t, j, \sigma)}{h_t} \tag{5.18}$$

This formula is very impractical as it can only be evaluated numerically and we still lack a way to determine σ precisely. Therefore, when estimating the needed window size, we recommend using histograms. Histograms can be built in a single scan through R_1 and

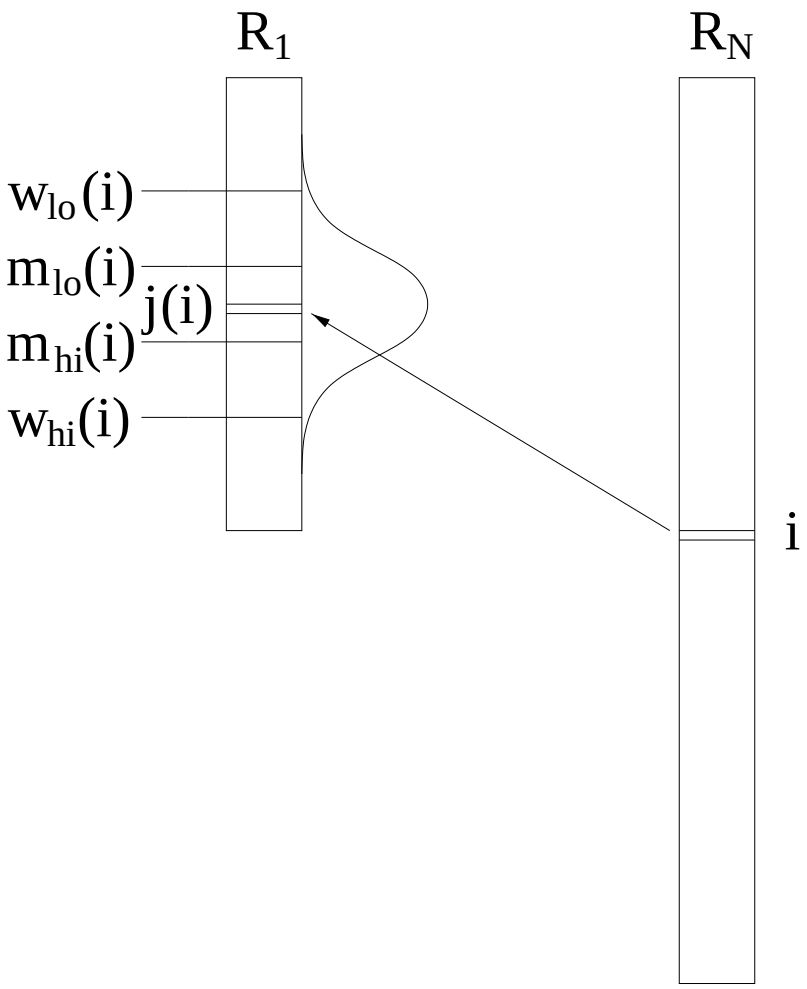


Figure 5.7: Normally distributed tuples

R_N with as large a buffer as possible. For each tuple in R_N the absolute value of the difference between the expected position and the actual position of the matching tuple in R_1 is inserted into the corresponding bucket of the histogram. Mishits are counted separately. The resulting histogram (for an example see Figure 5.8) can be used to approximate the smallest required window size for a given probability p_{accept}.

5.2 Benchmarks

This section is composed of two parts. Within the first part we describe the benchmark environment and how the benchmarks were run. In the second part we present the results and analyze them.

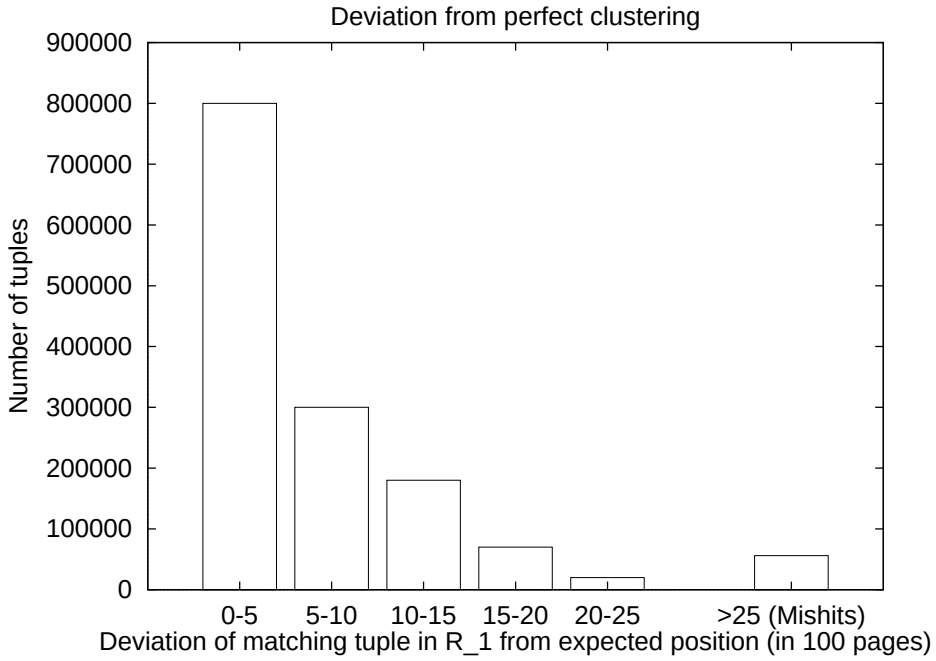


Figure 5.8: Histograms for measuring deviation from perfect clustering

5.2.1 Benchmark description

The benchmarks were executed on a lightly loaded Sun UltraSparc 1 (143 MHz) with 288 MByte main memory running under Solaris 2.5.1. The data we worked with were generated for a TPC-D benchmark with a scaling factor of 1 [90]. We joined the relations *Order* and *Lineitem* (see Figure 5.4 for the schemes). The relation *Order* was sorted on the attribute *orderdate*, *Lineitem* was sorted on *shipdate*. Note that this does **not** result in an ordering on the join attribute *orderkey* in the relations, but it models clustering by time of creation nicely. Moreover, in the TPC-D benchmark the positions of matching tuples are not distributed normally, but uniformly in an interval. Shipping dates are determined by adding 1 to 121 days randomly (uniform distribution) to the corresponding order date. As this is unrealistic, we expect Diag-Join to be even more efficient in practice.

The algorithm was implemented in C++ using the Sun C++ Compiler Version 4.1. It was integrated into our experimental Data Warehouse Management System AODB [93]. We buffered one page of mishits in main memory. For the standard join algorithm in the second phase of Diag-Join we used GRACE hash join [32, 85]. For the index nested-loop join we indexed the attribute *orderkey* on the relation *Order* with a B^+-tree using the Berkeley Database package [2].

In a first step we optimized some parameters of Diag-Join, e.g., the optimal number of hash tables. Then we compared the total costs, CPU-based costs and I/O-based costs of Diag-Join with blockwise nested-loop join, GRACE hash join, and index nested-loop join for different buffer sizes. We did not look at hybrid hash join, because for large relations

[2]Berkeley DB toolkit: http://www.sleepycat.com/

Order
orderkey
custkey
orderstatus
totalprice
orderdate
orderpriority
clerk
shippriority
comment

Lineitem
orderkey
partkey
suppkey
linenumber
quantity
extendedprice
discount
tax
returnflag
linestatus
shipdate
commitdate
receiptdate
shipinstruct
shipmode
comment

Table 5.4: Relations Order and Lineitem from TPC-D

relative to the size of main memory, GRACE hash join performs as well as hybrid hash join [42, 85]. Table 5.5 summarizes the parameters for the benchmarks. As can be seen the chosen buffer size is at most $\frac{1}{50}$ of the size of the relations. This is a realistic assumption for Data Warehouses in which huge relations can be found.

Parameter	*Value*
Page Size	4 KByte
Size of *Order*	44,475 pages
Cardinality of *Order*	1,500,000 tuples
Size of *Lineitem*	189,635 pages
Cardinality of *Lineitem*	6,001,215 tuples
Buffer size (window size) for Diag-Join	300 - 4000 pages (1.17 MByte - 15.62 MByte)
Step size (Window size/5)	60 - 800 pages
Buffer size for Nested-loop join (blockwise and index)	300 - 4000 pages (1.17 MByte - 15.62 MByte)
Buffer size for GRACE join	300 - 4000 pages (1.17 MByte - 15.62 MByte)

Table 5.5: Parameters used for benchmarks

5.2.2 Benchmark results

Tuning the Diag-Join algorithm

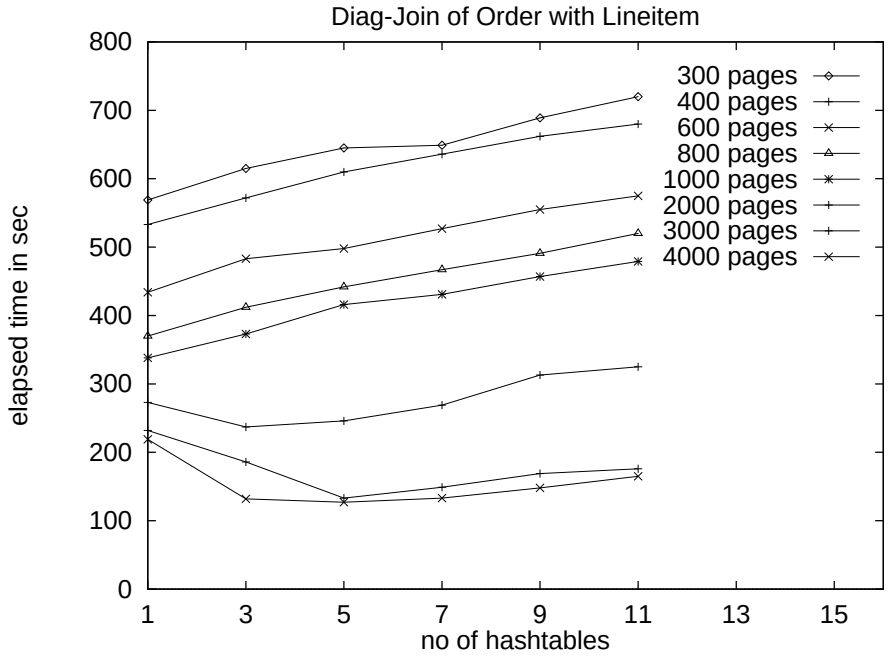


Figure 5.9: Granularity of hash tables

When joining relations with Diag-Join, we have to choose the right step size and buffer size for the window. Two effects have to be considered. If we use a large number of hash tables (small step size), we avoid cutting off matching tuples in the vicinity of the expected positions. However, the more hash tables we use, the longer the zig-zag search will take.

For small buffer sizes the step size is irrelevant, because the number of mishits caused by a large step size is small compared to the total number of mishits. For large buffer sizes, however, the number of mishits is relatively small and the step size has a noticeable effect. Decreasing the step size (i.e. increasing the number of hash tables) leads to a smoother scrolling of the window and thus fewer mishits caused by cutting off matching tuples (see Figure 5.9). Reducing the step size further does not improve the mishit ratio significantly. The run-time might even deteriorate as it is dominated by the search time for the zig-zag search in this case.

For our benchmarks we divided the window into five hash tables. In general this turned out to be a good compromise between optimizing the step size and the search time.

In Figure 5.10 the percentage of mishits in the relation *Lineitem* is depicted. The results of these measurements are straightforward. The more buffer we allocate, the lower the probability that a tuple from *Lineitem* will be a mishit, because the probability to find the matching tuple in a hash table increases. For large buffer sizes the effect of a large

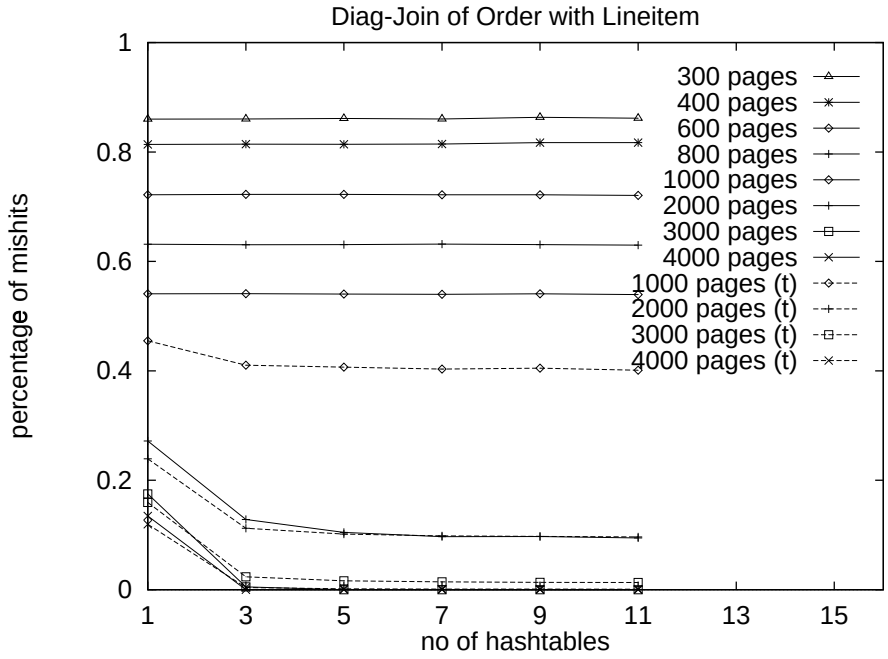


Figure 5.10: Percentage of mishits

step size can be clearly seen, as the percentage of mishits rises for a low number of hash tables. (The curves marked with (t) are theoretical values assuming that the deviation of the relations from perfect clustering can be described by a normal distribution (see Section 5.1.5).)

Comparison with other join algorithms

In this section we compare Diag-Join with blockwise nested-loop join, GRACE hash join, and index nested-loop join. The results for total runtime of all algorithms for joining the relations *Order* and *Lineitem* on the attribute *orderkey* are shown in Figure 5.11. Blockwise Nested-loop join is used as a reference, not as a serious competitor.

Total costs Blockwise nested-loop join performs worst. This comes as no great surprise, because the ratio between the buffer size and the relations' sizes is very unfavorable.

For sufficiently large buffer sizes (>3000 pages or 6% of $||Order||$) Diag-Join easily outperforms GRACE hash join, because in this case all tuples are joined in the first phase of Diag-Join and no additional phase for joining the mishits is needed. For medium-sized buffers (between 1000 and 3000 pages) Diag-Join is still faster than GRACE hash join and only for very small buffer sizes (<1000 pages or 2% of $||Order||$) GRACE hash join performs better. What are the reasons for this? The first phase of Diag-Join has a relatively low overhead, but is still able to join a certain number of tuples (see Figure 5.12). This takes at least some of the load off GRACE hash join in the second phase of Diag-Join.

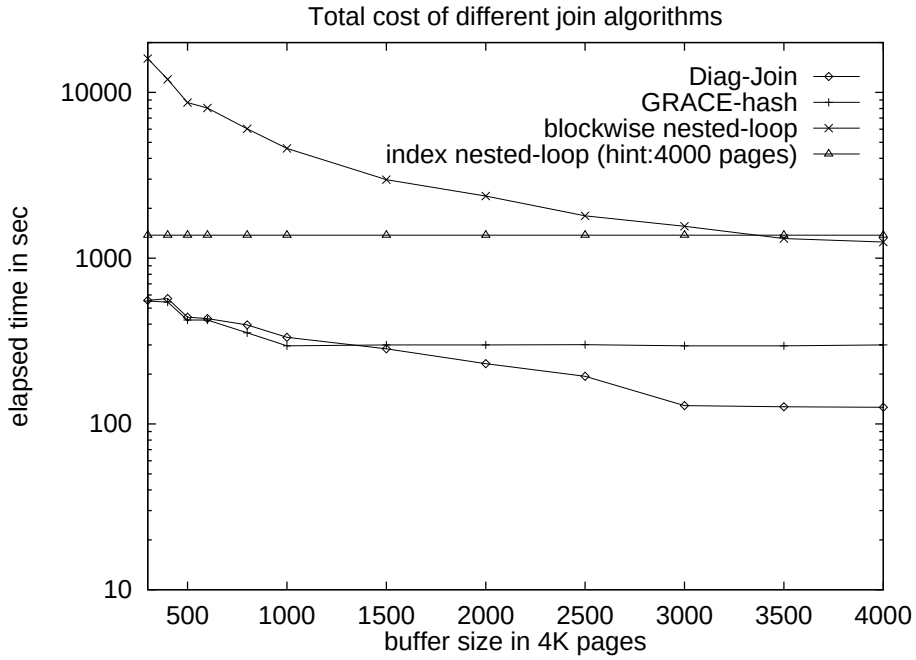


Figure 5.11: Total runtime of join algorithms

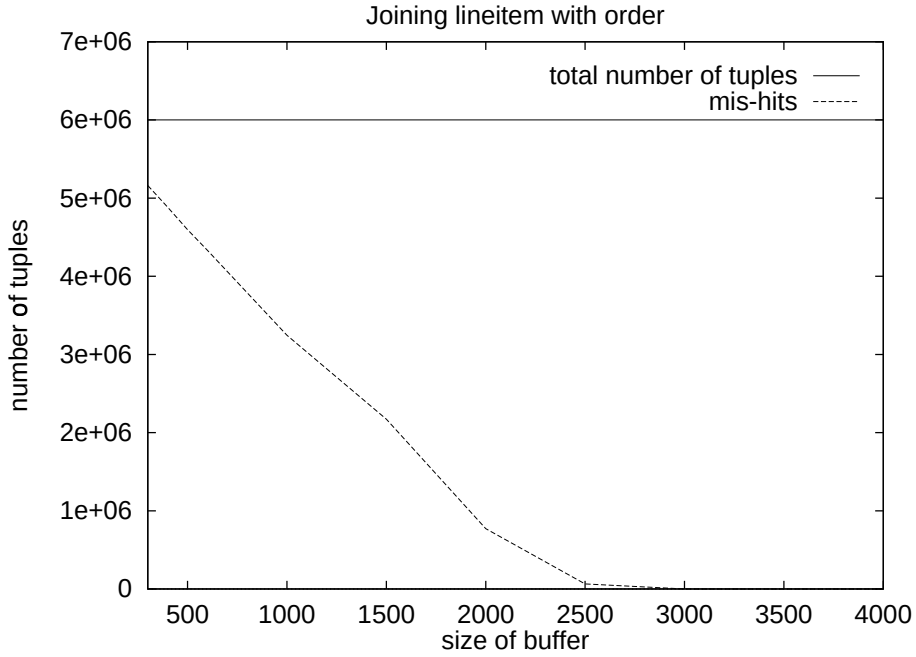


Figure 5.12: Total number of mishits

So the overhead for the first phase of Diag-Join is not as large as one might expect and even pays off for smaller buffer sizes.

Diag-Join also performs much better than index nested-loop join. Although index nested-loop join also profits from the clustering of *Order*, we have to access the tuples indirectly through a B^+-tree, which leads to a much higher overhead than hash table lookups. Since we used a generic B^+-tree (Berkeley DB toolkit) for the index join we had no guarantee on how much buffer was really allocated for the index lookups. Nevertheless the interface allowed to give hints, which we did by setting the buffer size to 4000 pages. Therefore we have only one measurement for the index join.

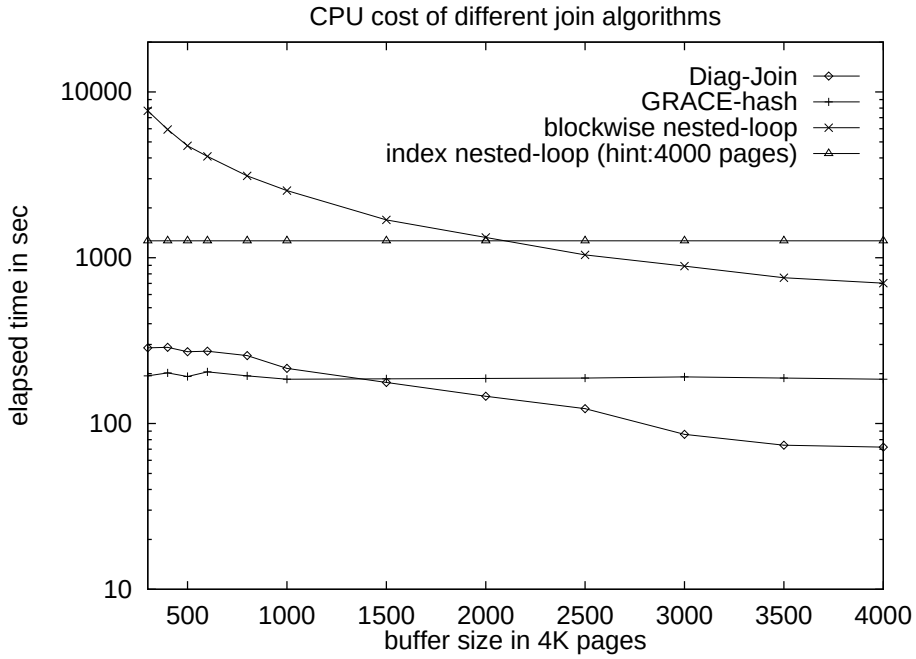


Figure 5.13: CPU costs of join algorithms

CPU-based costs Let us now have a look at the CPU-based costs of the join algorithms (see Figure 5.13).

The more memory we have available, the lower the costs of the blockwise nested-loop join are. This is obvious as the number of necessary loops decreases with increasing buffer size.

As long as it is sufficiently large, the size of the hash table directories is irrelevant for the CPU-based costs of GRACE hash join. The CPU-based costs for GRACE hash join are composed of the costs for hashing all tuples of *Order*, hashing all tuples of *Lineitem*, hashing all tuples of *Order* again during the merge phase, and doing $|Lineitem|$ lookups on this hash table. This leads to nearly constant costs.

The CPU-based costs for index nested-loop join are very high on account of the generic B^+-tree package we used. Some of these costs could be reduced by implementing a B^+-tree customized for AODB. However, this will not reduce the costs for searching the inner nodes of the tree, which will always be higher than (maximal) 5 lookups in hash tables.

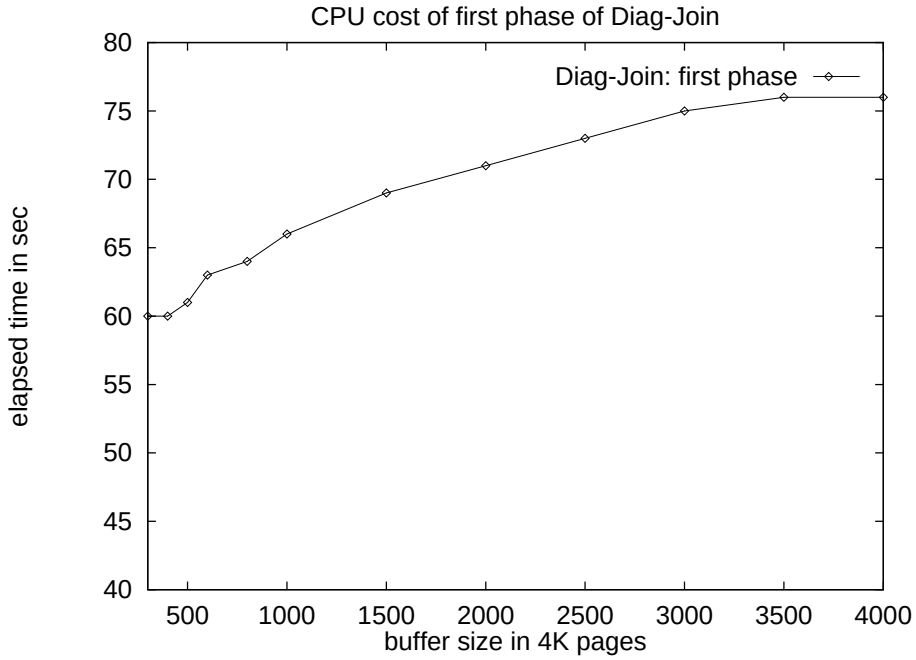


Figure 5.14: CPU-based costs for the first phase of Diag-Join

The CPU-based costs for Diag-Join for the first phase are almost constant regardless of buffer size, because *Order* and *Lineitem* are simply scanned (see Figure 5.14). The slight increase is caused by the costs for joining the tuples. The more available buffer there is in the first phase, the more tuples will find a join partner in this phase. (We did not write mishits to disk while measuring the CPU-based costs for the first phase.) The total decreasing CPU-based costs for Diag-Join are caused by falling costs of GRACE hash join in the second phase, as the number of tuples in the temporary file steadily decreases.

I/O-based costs The I/O-based costs are displayed in Table 5.15. For the blockwise nested-loop join we have the same behavior as for the CPU-based costs. The larger the buffer size, the smaller the number of loops and the lower the costs.

For GRACE hash join the I/O-based costs decrease with increasing buffer size. Beyond a certain buffer size, however, the seek and latency time becomes small and the costs for transferring the data dominate. As *Order* and *Lineitem* are always read twice and written once, a larger buffer does not change the transfer costs. Therefore the I/O-based costs level out.

Index nested-loop join also buffers pages of *Order* in main memory. When loading these pages into memory, however, they are not accessed sequentially. Therefore seek and latency time is considerably higher for index nested-loop join than for the other join algorithms.

When allocating large buffers for Diag-Join ($\geq$ 3000 pages, which corresponds to about 6% of the size of *Order*), all we have to do is to read *Order* and *Lineitem* once and we are finished. Hence we have low I/O-based costs in this case. For small buffers ($<$ 3000

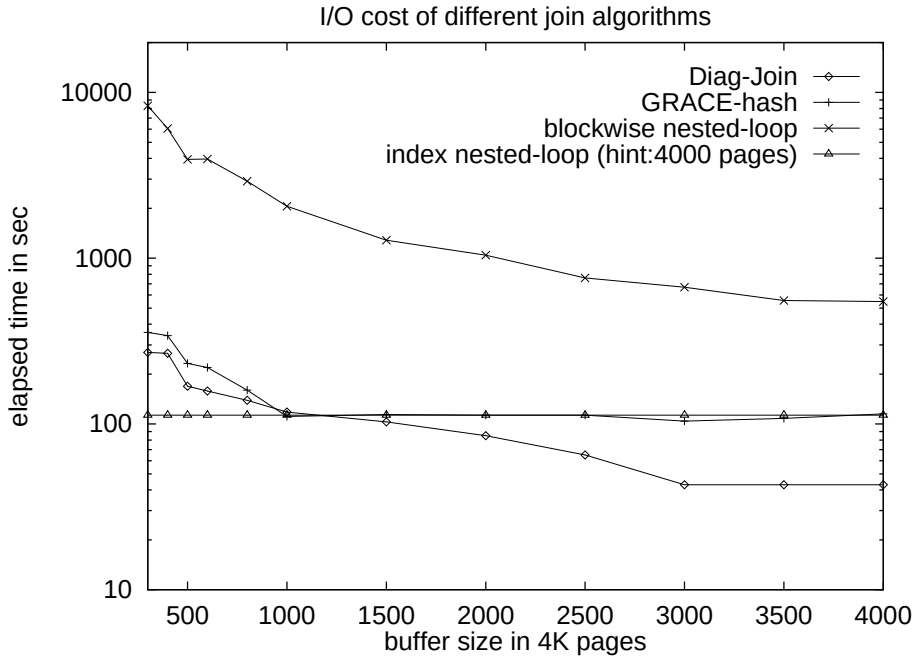


Figure 5.15: I/O-based costs of join algorithms

pages) all tuples of *Order* and *Lineitem* are read once in the first phase. Additionally, part of *Lineitem* is written into a temporary file, which is then joined with *Order*. When we decrease the buffer size, the temporary file will grow (because of a larger number of mishits) leading to higher join costs for GRACE hash join in the second phase.

5.2.3 Summary of Benchmarks

If we have a clustering of relations by time of creation, Diag-Join performs very well (up to two and a half times faster than GRACE hash join and considerably faster than blockwise/index nested-loop join). Diag-Join needs sufficient memory (about 6% of $||R_1||$ in our benchmark) to achieve the best case, but even for small buffer sizes the performance is still satisfactory.

Obviously, when joining relations that are not clustered by time of creation, i.e., relations with randomly placed tuples, Diag-Join will not perform better than other algorithms. In this case we expect a high rate of mishits as on average only $\frac{\text{buffer size}}{R_1} \cdot R_N$ of the tuples in R_N will find a matching tuple in the first phase. Diag-Join will not be a total failure in this case, however, as its overhead is not large and tuples that find a match need not be joined in the second phase, which decreases the costs of the join there.

5.3 Conclusion and Outlook

We developed a join algorithm, called Diag-Join, for any environment in which joining relations (or extents in object-oriented DBMS) clustered by time of creation takes place. We take advantage of the fact that incoming data is appended to the end of relations (or extents), resulting in a clustering of the tuples (or objects) by time of creation. When this is the case, often a single merge phase suffices for joining these large relations. This results in lower join costs than for any other join algorithm.

We implemented Diag-Join and integrated it into our experimental Data Warehouse Management System AODB. There we ran benchmarks based on the TPC-D relations *Order* and *Lineitem*. A careful analysis of the behavior of Diag-Join and the comparison with blockwise nested-loop join, GRACE hash join, and index nested-loop join revealed the impressive performance of our join algorithm. It ran two and a half times faster than GRACE hash join (the latter being on equal grounds with hybrid hash join in our case) and considerably faster than blockwise/index nested-loop join. However, we recommend that Diag-Join should only be used for at least loosely clustered relations, because for non-clustered relations the results are less favorable.

Diag-Join can be improved further by integrating it tightly into the join algorithm executed in the second phase. For example, the merging phase of Diag-Join can be coupled with the partition phase of GRACE hash join, i.e. all tuples that do not match are immediately partitioned. This would avoid the first scanning step of GRACE hash join.

An open question is the derivation of accurate (and not overly complex) methods for estimating the costs of a Diag-Join operator in a query-plan beforehand. This includes finding a measure for the degree of "clusteredness" of relations and the measurement of the effect of various other relational operators on the "clusteredness".

Chapter 6

Introduction to Index Structures

Like the aforementioned join algorithms index structures are transparent for the user, but play a key role in database performance. When searching for relevant data, the straightforward approach is to scan through all data and test every data item. This can be inefficient and cumbersome, especially if huge volumes of data are involved. Index structures allow fast access to data on secondary storage by content. The subject of indexing is a research topic of great interest as hundreds of different index structures have been proposed. Enumerating all these different index structures would go beyond the scope of this work. There are, however, a few underlying principles to all index structures, which we will describe in this chapter.

In the first section we give an overview of the general storage hierarchy in computing systems. Section 6.2 covers an abstract definition of queries, which we use to illustrate how index structures work. The basic principles of index structures are described in Section 6.3.

6.1 Storage Hierarchy

In every computing system, also in every DBMS, we have several layers of storage (see Figure 6.1). Generally the higher a memory type is positioned in this hierarchy, the faster, the costlier, and the smaller it becomes. The differences between the levels are usually several orders of magnitude. We divide this hierarchy into three sub-categories: *primary*, *secondary*, and *tertiary storage*. Primary storage consists of CPU-registers, cache memory, and main memory, the secondary storage comprises the disk level, and tertiary storage includes the tape level. We restrict ourselves to the levels that are most important for index structures in DBMSs: main memory and disks.

6.2 Data and Queries

As this is an introduction to index structures we discuss the subject in a simplified manner. Let us assume that we store a large number of data items in our database. We are particularly interested in a subset of data items that we want to index to allow for efficient

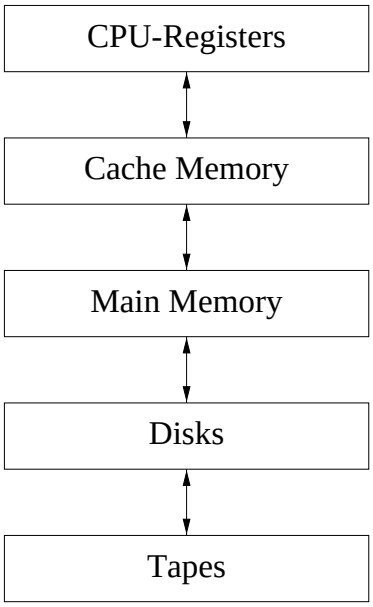


Figure 6.1: Levels of storage hierarchy

retrieval. Further we assume that the data items we want to index are all structured in the same way, i.e. they have attributes $a, b, c, \ldots$ with corresponding domains $D_a, D_b, D_c, \ldots$, so a data item o is in $\Phi = D_a \times D_b \times D_c \times \ldots$. We refer to the value of an attribute a of a data item o by $o.a$. We call the set of data items that are actually present in our database an *instance* Ω, i.e. $\Omega \subseteq \Phi$. Without loss of generality we assume that Ω consists of n data items: $\Omega = \{o_1, o_2, \ldots, o_n\}$. The data items in Ω are stored on pages in secondary storage (see Figure 6.2). During query evaluation, data items have to be fetched into main memory, which is considerably smaller than secondary storage.

A query is formulated in terms of a unary query predicate q. A unary predicate is defined as in first-order predicate calculus, so it is a mapping from Φ to $\{\text{true}, \text{false}\}$. When evaluating a query, we want to obtain all data items that satisfy the query predicate q, i.e. determine the answer set $A_q = \{o | o \in \Omega \wedge q(o) = \text{true}\}$. For example, let us assume that we have created a relation *Employee* in our database (see Figure 6.3). In this case the data items are the records representing the tuples of the relation. If we want to formulate a query that fetches all persons who are older than 30 years, we would use the query predicate $q_{\text{Age}>30}(o) := o.\text{Age} > 30$. The data items o_1, o_2, and o_4 will be found in the answer set, because they satisfy the predicate $q_{\text{Age}>30}(o)$.

How do we evaluate a query? The straightforward way is to fetch all data items and check the query predicate individually for each data item. This is quite expensive, as all data items have to be transferred from secondary storage to main memory and the predicate q has to be checked n times. Taking a closer look at DBMSs we notice the following:

- often only a fraction of the data items satisfy a given query predicate
- queries often use similar predicates

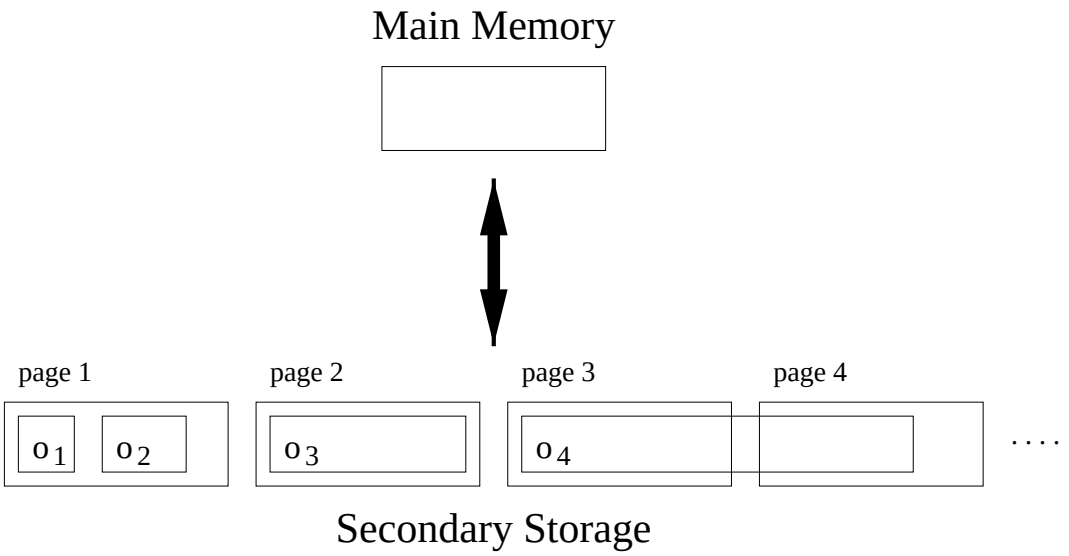


Figure 6.2: Transfer from pages into main memory

	Name	Salary	Age
o_1	Jones	45,000	32
o_2	Smith	50,000	45
o_3	Hayes	38,000	26
o_4	Green	75,000	53
o_5	Brooks	40,000	28
...	...	...	...

Figure 6.3: Employee relation

- secondary storage, like main memory, can be accessed randomly

Index structures exploit these properties of queries in order to reduce the accesses to secondary storage. This is a wide field of research, which is shown by the fact that hundreds of different index structures exist. Instead of discussing all these different index structures, we now explain what principles they work on. This is similar to the framework for trees presented by Hellerstein, Naughton, and Pfeffer in [44], though our approach is more general and more rigorous.

6.3 A General Framework for Index Structures

We want to avoid unnecessary transfer of data items from secondary storage to main memory, i.e. ideally we only want to fetch data items that satisfy the query predicate q. Often query predicates refer to certain properties of data items, e.g. the predicate $q_{\text{Age}>30}$ checks the value of the attribute *Age*. Our goal is to speed up query evaluation by

concentrating on the parts of data items that are essential for the evaluation of a query predicate q. We use predicates to describe properties of data items. Due to efficiency reasons, these predicates are not implemented explicitly in actual index structures, but help us in explaining the principles of index structures. In the following, we generalize the notion of search keys in terms of predicates. Furthermore, we explain how queries are evaluated on an abstract level using predicates. We then describe the general principles found in index structures (e.g. filters, tree structures, hashing techniques).

6.3.1 Describing predicates

We call a predicate that describes the property of a data item a *describing predicate*. It is defined as follows.

Definition 6.3.1 *A predicate p describes a data item $o \in \Omega$, iff $p(o) = true$.*

One possible describing predicate of data item o_2 in our employee relation (see Figure 6.3) is $p_2(o) := o.\text{Age} = 45$, because $p_2(o_2) = \text{true}$. Describing predicates generalize the concept of a search key, i.e. search keys in index structures can be seen as a way of representing describing predicates.

6.3.2 Query Evaluation

We assign a describing predicate p_i to a single data item or a set of data items[3], so that each data item is described by a predicate. During query evaluation, if we discover that $p_i \wedge q$ is unsatisfiable, we know that all data items described by p_i cannot possibly satisfy the query predicate q and we do not need to fetch them.

Definition 6.3.2 *Assume that p_i is a describing predicate and q a query predicate.*

$$p_i \wedge q \text{ is a contradiction } (p_i \wedge q \text{ is unsatisfiable}) \Leftrightarrow \forall\, o \in \Phi \;\; p_i(o) \wedge q(o) = \text{false}$$

At first glance this definition looks too general, as it involves all data items in Φ. Nevertheless, this approach makes sense. No matter how many data items are described by a predicate or how many new data items are inserted into our database (changing Ω), if $p_i \wedge q$ is unsatisfiable, we are guaranteed not to overlook any qualifying data items.

The general problem of determining whether or not $p_i \wedge q$ is a contradiction is intractable. That is one of the reasons why it is not possible to construct a multi-purpose index structure that is capable of supporting the efficient evaluation of each and every conceivable query. In practice, however, each index structure only supports a restricted set of query types. For these special query types the problem of detecting contradictions can be transformed into much simpler problems that can be solved very efficiently. In the following we describe different approaches of transforming the problem of determining contradictions, that is, we develop a general framework for index structures.

[3]This will become important in Section 6.3.5 on hierarchical index structures.

6.3.3 Projection

Let us look at the records of our relation *Employee* again. Assume we want to support queries with query predicates of the form $q_{Age}(o) := o.\text{Age}\ \theta\ c$ where $\theta \in \{=, <, >\}$ and $c \in D_{\text{Age}}$. In order to support the evaluation of queries concerning the attribute *Age*, we define the describing predicates as $p_i(o) := o.\text{Age} = o_i.\text{Age}$, where $o_i.\text{Age}$ is the value of the attribute Age of data item o_i. We assign a describing predicate p_i to each data item $o_i \in \Omega$ (the describing predicate have the same subscript as the corresponding data item). In this special case it is very simple to determine whether or not $p_i \wedge q_{Age}$ is a contradiction. All we have to do is to distinguish three different cases:

- θ is "=": $p_i \wedge q_{Age}$ is a contradiction $\Leftrightarrow o_i.\text{Age} \neq c$
- θ is "<": $p_i \wedge q_{Age}$ is a contradiction $\Leftrightarrow o_i.\text{Age} \geq c$
- θ is ">": $p_i \wedge q_{Age}$ is a contradiction $\Leftrightarrow o_i.\text{Age} \leq c$

We have transformed the problem into a simple comparison between values of D_{Age}. So it suffices to hold a copy of the value of the attribute *Age* of each data item, which represents the describing predicate, and a reference to the data item in the index (see Figure 6.4).

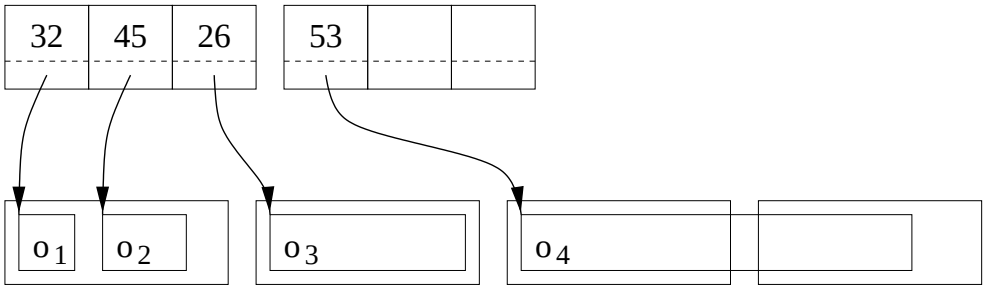


Figure 6.4: Example for storing predicates p_i for relation *Employee* compactly

When evaluating a query the constant c of the query predicate is compared to the value of *Age* of each data item in the index. Only if there is no contradiction, we fetch the data item. This speeds up query evaluation as the attribute values of *Age* can be stored more compactly in the index than the data items themselves. Therefore the index can be scanned much faster than the set of all data items and only qualifying data items are fetched. An index that works in this way is called *projection index* [31].

6.3.4 Filter

In our previous example we were able to transform the problem of determining if $p_i \wedge q$ is a contradiction to simple comparisons of numbers. Merely projecting onto an attribute is not always an efficient transformation though. Consider a geographical database storing

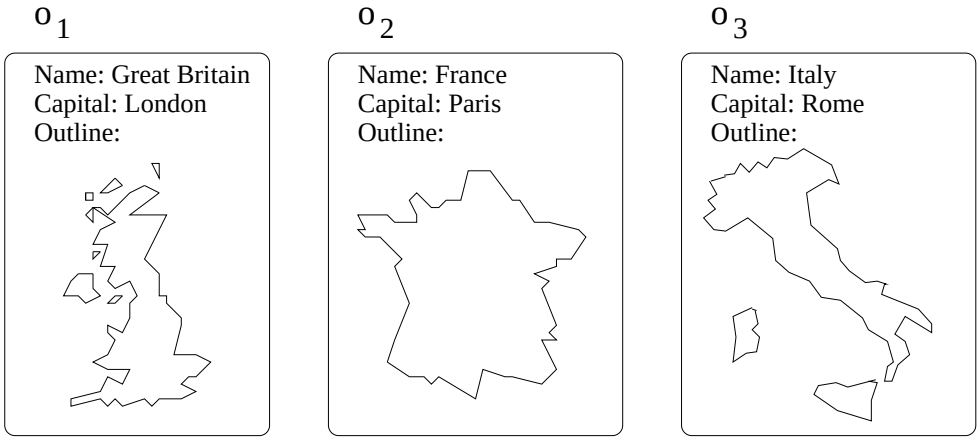


Figure 6.5: Outlines of countries in a geographical database

two-dimensional data (maps of countries). Each data item has the attributes *Name*, *Capital*, and *Outline* (see Figure 6.5). The attribute *Outline* contains lists of point coordinates describing polygons.

We want to support containment queries, which are typical in this context. Given a point $(x, y) \in \mathbb{R}^2$ we want to know if this point is contained in the outline of a country. We have query predicates of the form

$$q_{x\_y\_is\_in}(o) := \begin{cases} \text{true} & \text{if } o.Outline \text{ contains } (x, y) \\ \text{false} & \text{else} \end{cases}$$

Using the projection technique from the last section we would define our describing predicates as follows:

$$p_i(o) = \begin{cases} \text{true} & \text{if } o.Outline = o_i.Outline \\ \text{false} & \text{else} \end{cases}$$

Determining whether $p_i \wedge q_{x\_y\_is\_in}$ is a contradiction boils down to

$$p_i \wedge q_{x\_y\_is\_in} \text{ is a contradiction } \Leftrightarrow \ o_i.\text{Outline does not contain } (x, y)$$

Checking if a point is contained in a polygon can be costly for large polygons. Copying the values of the attribute *Outline* to a separate place on disk and scanning them is not more efficient than just scanning the data items themselves. So just using a projection index straightforwardly will not work in this case.

A technique used in geometrical applications to simplify the detection of contradictions are *bounding boxes*. A bounding box is defined as the smallest rectangle, whose sides are

parallel to the x- and y-axes, that completely covers the outline. Usually the coordinates of the lower left $(x^{\min}, y^{\min})$ and upper right corner $(x^{\max}, y^{\max})$ are given (see Figure 6.6). The resulting index structure is depicted in Figure 6.7. For each data item we store the coordinates of the lower left and upper right corner of the bounding box of its outline.

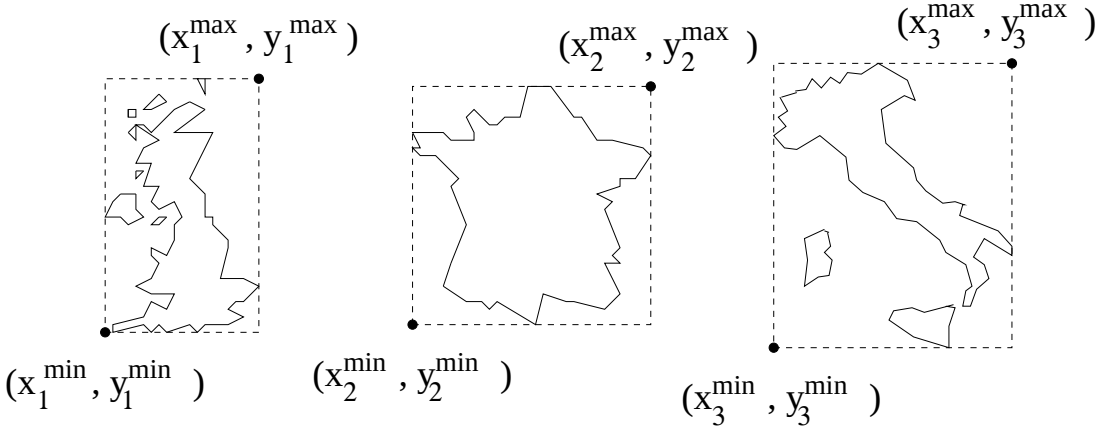


Figure 6.6: Bounding boxes

Instead of checking whether $o_i.Outline$ contains the point (x, y) we check whether the bounding box of o_i contains (x, y). Note that this is not equivalent to determining whether $p_i \wedge q_{x\_y\_is\_in}$ is a contradiction, this is just an implication:

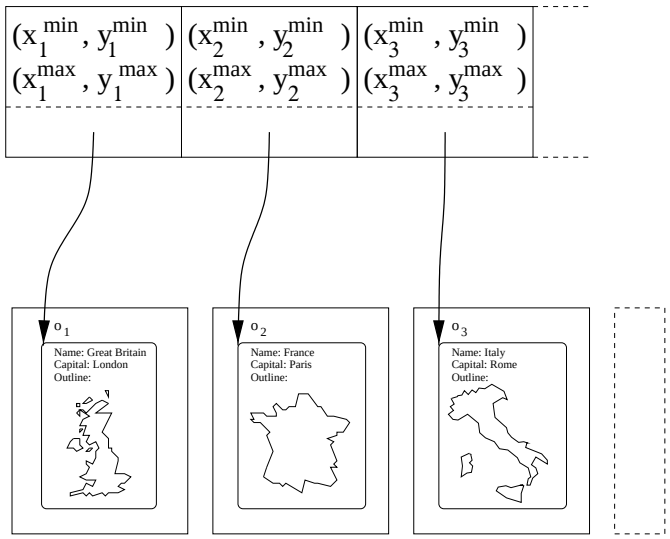


Figure 6.7: The complete geographical index

$$p_i \wedge q_{x\_y\_is\_in} \text{ is a contradiction} \quad \Leftarrow \quad (x, y) \text{ is not in bounding box of } o_i, \text{ i.e.}$$
$$x < x_i^{\min} \text{ or } x > x_i^{\max} \text{ or } y < y_i^{\min} \text{ or } y > y_i^{\max}$$

If (x, y) is not in the bounding box or o_i, then we are sure that $p_i \wedge q_{x\_y\_is\_in}$ is a contradiction. Let us prove this assertion. For a moment let us assume that the assertion is false, that is a data item o_i exists for which (x, y) is not in the bounding box of o_i and $p_i \wedge q_{x\_y\_is\_in}$ is not a contradiction. It follows that a data item $o_j \in \Phi$ exists for which $p_i(o_j) = \text{true}$ and $q_{x\_y\_is\_in}(o_j) = \text{true}$. As $p_i(o_j) = \text{true}$ we know that o_j has the exactly the same outline as o_i. Therefore $q_{x\_y\_is\_in}(o_i) = \text{true}$ also holds, i.e. the outline of o_i contains (x, y), too. If (x, y) is contained in $o_i.Outline$, then the bounding box of o_i also contains (x, y). This is contrary to our assumptions, therefore the assertion is true.

If we know that $p_i \wedge q_{x\_y\_is\_in}$ is a contradiction, this does not necessarily imply that (x, y) is not in the bounding box of o_i, however. So data items may exist for which $p_i \wedge q_{x\_y\_is\_in}$ is a contradiction, but (x, y) is contained in the bounding boxes of these data items. These data items are called *false drops* or *false positives*, because they do not satisfy the query predicate, but are fetched nonetheless. After fetching a data item we check if it really satisfies the query predicate $q_{x\_y\_is\_in}$ using more costly algorithms. By doing a quick pretest, however, we have avoided doing this for all data items. This technique is called *filtering* [15]. The main task in building effective filters revolves around finding transformations of the problem of detecting contradictions that can be evaluated efficiently without causing too many false drops.

Up to now we have avoided the (expensive) transfer of all data items from secondary storage to main memory. However, we still have to evaluate (at least) n predicates and we have some overhead administrating describing predicates. In the next section we show how we can improve the retrieval even further.

6.3.5 Hierarchical Organization

For fast query evaluation we want to avoid as much transfer of all kinds of data from secondary storage to main memory as possible. In addition to eliminating data items from the search process we want to access only describing predicates p_i that do not contradict the query predicate q. One way we can achieve this is by organizing the describing predicates hierarchically.

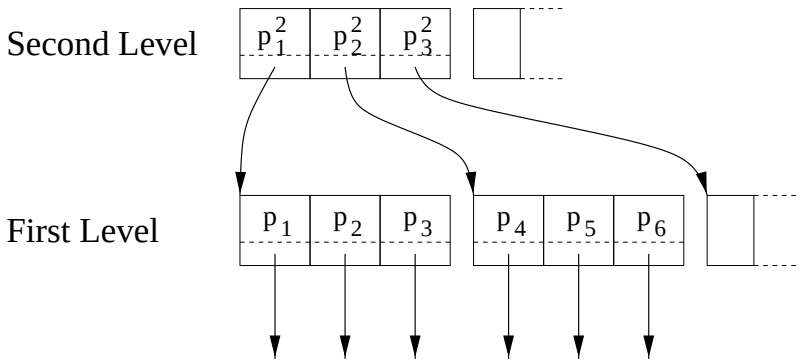


Figure 6.8: Two level hierarchy

General description

In the last sections we described techniques that help speed up query evaluation. One drawback of these techniques was the need to scan all index pages. It would be helpful to avoid all index pages that lead to data items whose describing predicates contradict the query predicate. We could describe index pages with predicates and fetch only those index pages whose predicate does not contradict the query predicate. Figure 6.8 illustrates the concept. For the sake of simplicity let us look at the index as if we stored the predicates in the index (in practice we would store representations of them). The predicates on the first level are ordinary describing predicates of data items (p_1 for o_1, p_2 for o_2, and so on). The predicates on the second level (annotated by p_k^2) describe sets of predicates on the first level. In Figure 6.8 predicate p_1^2 describes p_1, p_2 and p_3 and predicate p_2^2 describes p_4, p_5 and p_6. Usually all predicates found on one page are combined into a set.

How are the predicate on the second level chosen? We require that the following condition is satisfied by second level predicates.

Definition 6.3.3 *p_k^2 is a second level predicate of the first level predicates $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$, iff $p_{i_j} \Rightarrow p_k^2$ holds for all j, $1 \leq j \leq m$.*

When evaluating a query, we go through all predicates p_k^2 on the second level and check if they contradict the query predicate. Only if there is no contradiction we fetch the corresponding predicates from the first level and check those. That way we are sure that no data item will be overlooked. We assume without loss of generality that data item o_i satisfies the query predicate q. By definition $p_i(o_i)$ is true and since $p_i \Rightarrow p_k^2$ we know that $p_k^2(o_i)$ is also true. Therefore $p_k^2 \wedge q$ cannot be a contradiction, because $p_k^2(o_i) \wedge q(o_i)$ is true. As each p_i only implies p_k^2 false drops may exist, i.e. there could be cases where $p_k^2 \wedge q$ is not a contradiction, but for each p_i, $p_i \wedge q$ is a contradiction. So sometimes first level pages are fetched unnecessarily.

Let us return to the index for our relation *Employee*. We already defined in Section 6.3.3 the general form of query predicates and describing predicates. We construct second level predicates in the following way. Assume p_k^2 describes the first level predicates $P = \{p_{i_1}, p_{i_2}, \ldots, p_{i_m}\}$.

$$p_k^2(o) := minAge_k \leq o.\text{Age} \leq maxAge_k$$

with

$$minAge_k = \min(\{o_{i_j}.\text{Age} | p_{i_j} \in P\})$$
$$maxAge_k = \max(\{o_{i_j}.\text{Age} | p_{i_j} \in P\})$$

Now to determine, if p_k^2 contradicts a query predicate q_{Age} we do the following:

- θ is "=": $p_k^2 \wedge q_{Age}$ is a contradiction $\Leftrightarrow minAge_k > c \vee maxAge_k < c$
- θ is "<": $p_k^2 \wedge q_{Age}$ is a contradiction $\Leftrightarrow minAge_k \geq c$
- θ is ">": $p_k^2 \wedge q_{Age}$ is a contradiction $\Leftrightarrow maxAge_k \leq c$

All we need to know to check for contradictions of the query predicate with a second level predicate p_k^2 are the lower and upper bounds of the interval defined by p_k^2. Figure 6.9 shows a two level index for the records of the relation *Employee*. Note that we also exploited the fact that a total order exists on the domain D_{Age}. In this way we can avoid largely overlapping intervals.

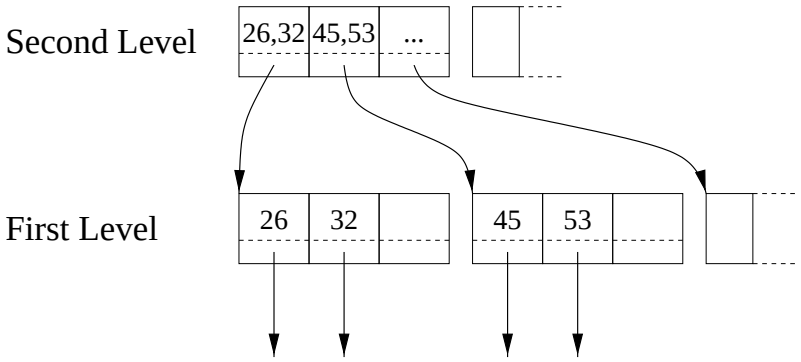


Figure 6.9: Example of a two level index

Inserting new data items

The advantages of this two-level organization come at a price. Updating these structures is more costly than updating a single level index. When inserting a new data item o_i we try to insert it in such a way that no second level predicate needs to be changed. That is we look for an ideal predicate p_k^2 for which $p_i \Rightarrow p_k^2$. If this is not possible we have to introduce a modified second level predicate $p_k'^2$ such that $p_k^2 \Rightarrow p_k'^2$ and $p_i \Rightarrow p_k'^2$. When doing this we have to be careful to cause few additional false drops.

Let us illustrate the insertion of a new data item with our example. We want to insert a new data item, o_9, into the structure depicted in Figure 6.9. The value of the attribute o_9.Age is equal to 35 and therefore $p_9(o) := o.\text{Age} = 35$. In this example we have two options to insert o_9, on the left index page with second level predicate $p_1^2(o) := 26 \leq o.\text{Age} \leq 32$ and on the right index page with second level predicate $p_2^2(o) := 45 \leq o.\text{Age} \leq 53$. Both options are not the ideal case. We can introduce a pseudo-metric for our index that measures the expected difference in efficiency for both options. As a quick reminder a pseudo-metric is defined as follows.

Definition 6.3.4 *Let M be a set. A function $d : M \times M \rightarrow \mathbb{R}$ is called a pseudo-metric, iff (assuming $x, y \in M$)*

1. $d(x, y) = d(y, x)$

2. $x = y \Rightarrow d(x, y) = 0$

3. $d(x, y) + d(y, z) \geq d(x, z)$

In our example we would like to keep the increase of the intervals of the second level predicates small in the hope that less false drops will occur. Assuming that domain $D_{Age} = \mathbb{N}$, a possible pseudo-metric could be

$$d(p_a^2, p_b^2) := |(maxAge_a - minAge_a + 1) - (maxAge_b - minAge_b + 1)|$$

Inserting o_9 on the left index page would result in the second level predicate $p_1'^2(o) := 26 \leq o.\text{Age} \leq 35$, inserting it on the right index page in $p_2'^2(o) := 35 \leq o.\text{Age} \leq 53$. As $d(p_1^2, p_1'^2) = 3$ is smaller than $d(p_2^2, p_2'^2) = 9$ we insert o_9 on the left side.

Finding the right place for insertion is not the only problem. There may not be enough space left on the first level page where we want to insert a data item o_i with a describing predicate p_i. We then have to split this first level page. We do this by dividing the representations of the predicates $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ present on this page and the representation of p_i into two approximately equal sized sets P_x and P_y. For these two sets we determine two second level predicates p_x^2 and p_y^2 such that for all $p \in P_x$ $p \Rightarrow p_x^2$ holds and for all $p \in P_y$ $p \Rightarrow p_y^2$ holds. When splitting up the describing predicates into P_x and P_y we have to be careful not to cause too many false drops.

Let us illustrate the splitting process with an example. Assume we have the access structure shown in the upper half of Figure 6.10. We want to insert a new data item o_{10} with $o_{10}.Age = 27$ and $p_{10}(o) := o.Age = 27$. This time we can find a describing predicate on the second level that fits perfectly, i.e. we do not need to modify any predicate on the second level. However, there is not enough space left on the first level page, so we have to split it. The upper half of Figure 6.10 shows the state before insertion, the lower half the state after insertion. Again we have exploited the total order on D_{Age} to attain small intervals for the second level predicates.

Eventually the representations of the predicates on the second level also have to be adjusted or moved. If we shift them to the right to make room for a new items, the technique is called ISAM (index-sequential access method) [34].

We could also describe predicates on the second level with predicates on a third level and so on. If we do not limit the hierarchical organization to two levels we get tree structures. Then we would split overflowing pages regardless of their level and propagate the changes to higher levels. Examples for index structures that work after these principles are B-trees [3], Quad-trees [30], R-trees [39], and S-trees [20]. Each uses different representations for the describing predicates and different strategies for handling insertions, but the underlying principles are the same.

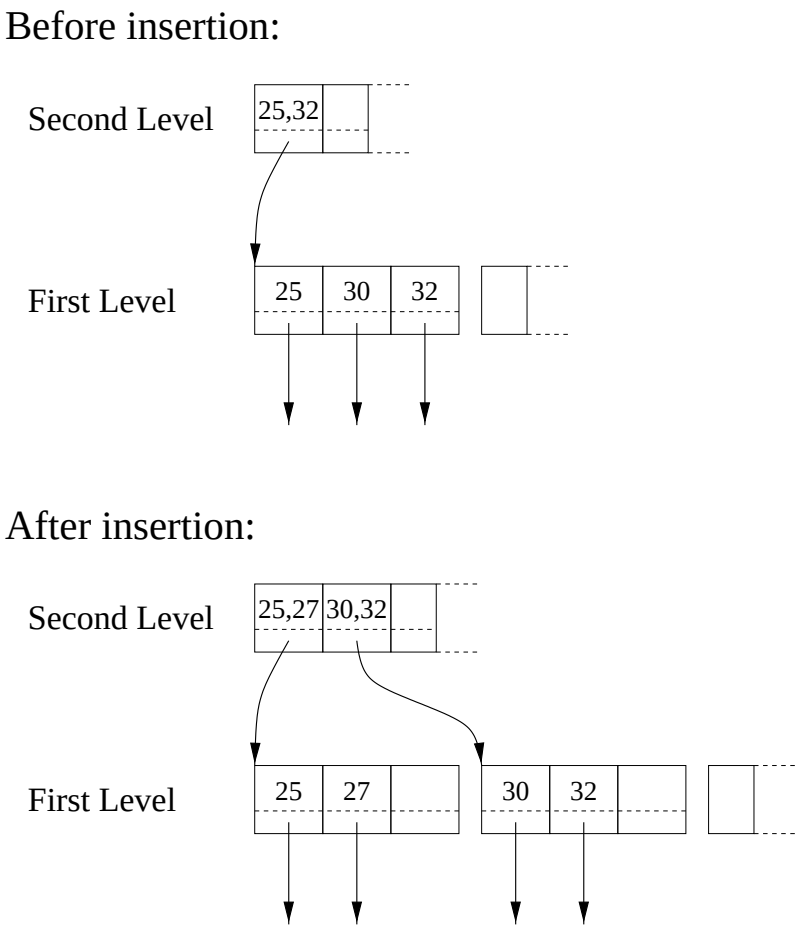


Figure 6.10: Splitting a predicate

6.3.6 Partitioning

The space required to store representations of describing predicates may still be quite large even when using weakened predicates. When we organize describing predicates hierarchically, we determine the predicates on the higher levels when needed, i.e. when we split pages. This is a very dynamic process; indeed, depending on what kind of data items we insert in which order, the describing predicates on higher levels can look very different from case to case.

For certain applications we can devise efficient access structures and not store explicit representations of describing predicates at all. If we have strict rules governing the form of the query predicates and the representations of the describing predicates, we can arrange the data items systematically. Given a query predicate it would be possible to calculate the positions of all data items whose describing predicate does not contradict the query predicate.

We can visualize this principle with our *Employee* relation. The describing predicates have the form given in Section 6.3.3. For now we allow only query predicates of the form $q_{Age}(o) := o.\text{Age} = c$. Supporting the efficient evaluation of queries with query predicates

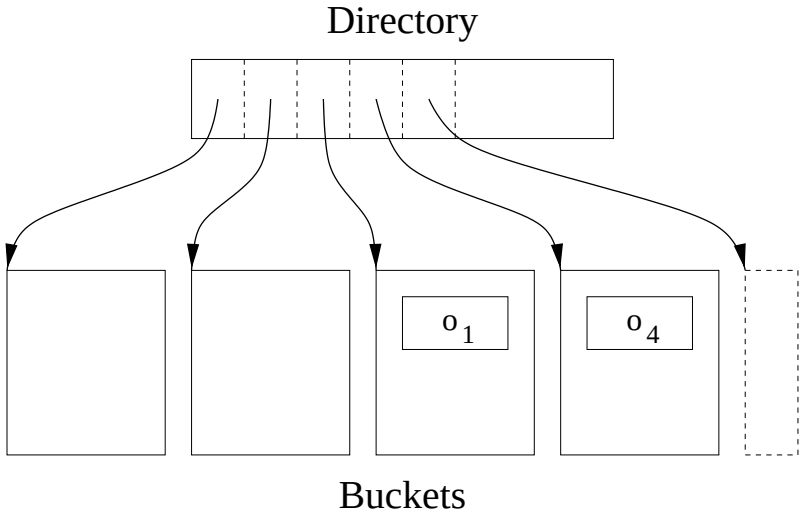


Figure 6.11: Partitioned index

having other comparison operators is not straightforward or not even possible. We could store all tuples for which o.Age mod 10 $= j$ in container j. We only need to store the references to the containers, the representations of the describing predicates are not stored anywhere. The place where the references are stored is called *directory*, the containers are called *buckets* (see Figure 6.11). Buckets are not necessarily restricted to one page. When searching for all persons of a particular age, we can calculate the position in the directory of the correct reference by c mod 10. We then scan the corresponding bucket for qualifying data items (using the above mapping data items with o.Age $= c + 10 \cdot j$ are also found in the same bucket). The technique described above is called *hashing*. Usually the function used for calculating the correct position, called *hash function*, is more sophisticated to allow a better distribution of the data items in the buckets.

The method of partitioning is only feasible if the buckets do not grow too large. Otherwise we have to do a costly sequential search in large buckets. However, we constantly insert new data items into a database. This is a fact we have to consider when creating hash indexes. When an overflow occurs, i.e. there is no space left in a bucket, there have to be rules that dictate how the index has to be restructured. Partitioned index structures that are dynamically restructured are called *dynamic hashing index structures*. Examples for dynamic hashing are linear hashing [63] and extendible hashing [26].

6.3.7 Summary

Although at first glance index structures seem to vary widely, they are based on a few underlying principles for reducing the transfer of data from secondary storage to main memory. The main task in building high performance index structures is finding appropriate describing predicates and transforming the problem of detecting contradictions into simpler problems that can be solved efficiently without loss of significance.

Chapter 7

Index Structures for Set-valued Attributes

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

Examples of applications involving set-valued attributes include keyword searches and queries in annotation databases containing information on images [9, 52], genetic or molecular data [2, 28]. Further, queries with universal quantifiers can be answered by set comparisons [16]. Many sets found in set-valued attributes are small, containing fewer than a dozen elements. Examples of applications where almost all sets are of low cardinality can be found in product and production models [37] and molecular databases [1, 94]. We focus on applications where sets are small enough to be embedded into the data items.

Indexing set-valued attributes is a difficult task for traditional index structures, because the data cannot be ordered and the queries are neither point queries nor range queries. Hellerstein, Koutsoupias, and Papadimitriou have shown in [43] that answering set inclusion queries is more difficult than answering 2-dimensional queries. We have combined many techniques from several different areas (partial-match retrieval, text retrieval, traditional database index structures, spatial index structures, join algorithms) to implement five index structures for set-valued attributes: sequential signature files, signature trees, extendible signature hashing, recursive linear signature hashing, and inverted files. Evaluating these index structures is very complex, since there is a huge number of parameters involved. We have done a mathematical modeling and comparison, but the complexity of the task and the weaknesses of theoretical tools for evaluating the performance of index structures motivated us to conduct additionally extensive experiments. We reveal implementation details of our index structures so that the results of the experiments become more transparent.

Related work

Work on evaluation of queries with set-valued predicates is few and far between. Several indexes dealing with special problems in the object-oriented [14] and the object-relational data models [89] have been invented, e.g. nested indexes [5], path indexes [5], multi indexes [68], access support relations [55], and join index hierarchies [95]. The predominant problem attacked by these index structures is the efficient evaluation of path expressions. With the exception of signature files [51] and Russian Doll Trees [45] the problem of indexing data items with set-valued attributes has been neglected by the database community. Searching for sets that are supersets of a query set is the predominant query form in text retrieval applications. Therefore the methods used in text retrieval, although at first glance similar to our methods, are optimized only for this partial-match retrieval. When indexing set-valued attributes, however, we have to look beyond that. In molecular databases, for example, searching for characteristic parts of a large molecule is a common query. So we need to support queries looking for subsets of a query set efficiently, too. Retrieving sets equal to a query set, although a simpler case, should also not be forgotten. Text retrieval techniques ignore the efficient evaluation of these two query types simply because they are not needed in that context.

Outline

This chapter is organized as follows. The next section covers preliminaries, i.e. a formal description of queries and a brief introduction to the storage manager we used to implement the index structures. In Section 7.2 we describe the index structures. We then devise a mathematical model and compare the index structures based on this model in Sections 7.3 and 7.4. A detailed description of the environment in which the experiments were conducted is the content of Section 7.5. We discuss the fine-tuning of the signature-based index structures for the experimental environment in 7.6. We present and analyze the results of the experiments in Section 7.7 for uniformly distributed data and in Section 7.8 for skewed data. Section 7.9 concludes the chapter.

7.1 Preliminaries

Before proceeding to the actual index structures, we need to explain some basics. Table 7.1 contains a summary of the symbols used throughout this chapter and their definitions. The remainder of this section presents definitions of set-valued queries and takes a quick look at the storage manager that was used.

7.1.1 Set-valued Queries

Our database consists of a finite set Ω of data items or objects o_i $(1 \leq i \leq n)$ having a set-valued attribute A with a domain D from which the elements of A are chosen. Let $o_i.A \subseteq D$ denote the finite value of the attribute A for some data item o_i. A *query predicate* q is defined in terms of a set-valued attribute A, a finite *query set* $Q \subseteq D$, and a

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

Table 7.1: Used symbols

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

7.1.2 Storage manager

The index structures were implemented atop the storage manager EOS. EOS provides key facilities for the fast development of high performance DBMSs [6, 7]. We exploited only a small part of the offered features, namely standard EOS objects and plain database pages. EOS objects are stored on slotted pages and are identified by an unique object identifier (OID) with a size of 8 byte. We stored all data items with set-valued attributes in EOS objects. Plain database pages (with a size of 4096 byte) belong to a particular storage area (a raw disk partition in our case) and do not contain any control information. We used these pages to implement our index structures.

7.2 The Competitors

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

Principally all signature-based index structures are filters (see Section 6.3.4) because we transform the comparison of sets (which is quite expensive as we have seen in Section

4.1.2) to a comparison of signatures. Basically we have three options to organize the signatures: sequentially [4], hierarchically, and partitioned. We examine sequential signature files (SSF) [51] for sequential organization, signature trees (ST) [20, 45] for hierarchical organization, and extendible signature hashing (ESH) [46] and recursive linear signature hashing (RLSH) [46] for partitioned organization.

7.2.1 Sequential Signature File (SSF)

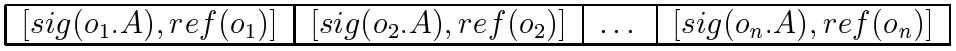


Figure 7.1: Sequential signature file index structure (SSF)

General description

A sequential signature file (SSF) [51] is a rather simple index structure, it is a filtering projection index. It consists of a sequence of pairs of signatures and references to data items. $[sig(o_i.A), ref(o_i)]$ (see Figure 7.1). During retrieval the SSF is scanned and all data items o_i with matching signature $sig(o_i.A)$ are fetched and tested for false drops. Inserting a new data item o_{n+1} into a sequential signature file is also straightforward. We have to generate a tuple containing the signature sig(o_{n+1}) and the identifier ref(o_{n+1}) and append this tuple to the signature file.

Name	*Description*
noOfEntries	the number of data items inserted into index
sizeOfSig	b, the size of the signatures (in bits)
noOfSetBits	k, the number of bits set in signature for each element
pageSize	the size of the pages in the database (in bytes)
sigAreaNo	the area number of signature list
sigPageNo	the starting page number of signature list
lastSigPageNo	the ending page number of signature list
lastSigPos	the first free position on last page of signature list
refAreaNo	the area number of reference list
refPageNo	the starting page number of reference list
lastRefPageNo	the ending page number of reference list
lastRefPos	the first free position on last page of reference list

Table 7.2: Root object of an SSF index

[4] This corresponds to a projection index.

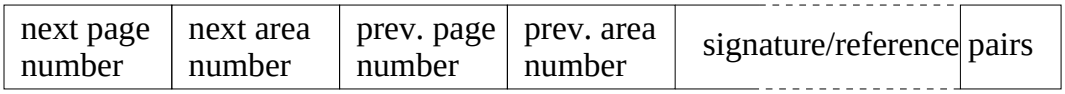


Figure 7.2: Signature and reference list node

Implementation details

An SSF index consists of a root object, which is copied into main memory when the index is opened, and pages containing the signatures and references. The entries of the root object are described in Table 7.2.

The pages containing the signatures and references are doubly linked. The first 8 bytes contain the link (page number, area number) to the next page, the next 8 bytes the link to the previous page (see Figure 7.2). For empty links the page and area numbers are set to 0.

7.2.2 Signature Tree (ST)

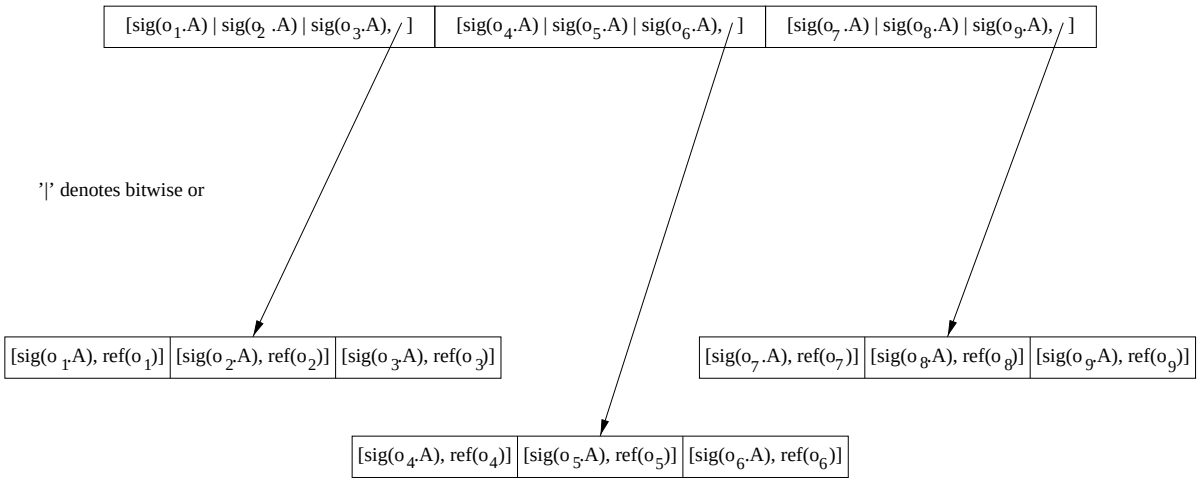


Figure 7.3: A Signature tree (ST)

General description

A Signature Tree (ST) is a hierarchical version of a filter index for sets. The internal structure of Signature Trees is very similar to that of R-trees [39]. The leaf nodes of ST [20, 45] contain pairs $[sig(o_i.A), ref(o_i)]$. In the leaf nodes of an ST we find the same information as in an SSF. We can construct a single signature representing a leaf node by superimposing all signatures found in the leaf node (with a bitwise or-operation, denoted by "|"). This corresponds to uniting the sets in the leaf nodes. We call a union of sets

from lower levels in the tree a bounding set. An inner node contains signatures of and references to each child node (see Figure 7.3).

When we evaluate a query we begin by searching the root for matching signatures. We recursively access all child nodes whose signatures match and work our way down to the leaf nodes. There we fetch all eligible data items and check for false drops. Matching signatures in leaf nodes are determined by the appropriate bitwise operation (see Section 2.1). Matching signatures in inner nodes are determined as follows. For equality and subset queries we check if $sig(Q) \subseteq sig(\text{child node})$. For superset queries there has to be a non-empty intersection, i.e. $|sig(Q) \cap sig(\text{child node})| \geq k$.

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 using the pseudo-metric

$$\text{diff}(sig(B_{ji}), sig(o_{n+1})) = |(sig(B_{ji}) \vee sig(o_{n+1})) \oplus sig(B_{ji})| \tag{7.1}$$

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

We descend the branch with the smallest value for $\text{diff}(sig(B_{ji}), sig(o_{n+1}))$. When we reach a leaf node m_l, the tuple $[sig(o_{n+1}), ref(o_{n+1})]$ is inserted and the signatures of the bounding sets in the parent nodes of m_l are modified.

When an overflow occurs during an insertion, i.e. there is no space left in leaf node m_l, m_l needs to be split. For the split of a node, we use 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 updated. This may lead to further splits, possibly all the way to the root node.

Name	*Description*
height	the height of the tree
sizeOfSig	the size of the signatures (in bits)
noOfSetBits	k, the number of bits set in signature for each element
pageSize	the size of the pages in the database (in bytes)
rootAreaNo	the area number of the root of the tree
rootPageNo	the page number of the root of the tree

Table 7.3: Root object of an ST index

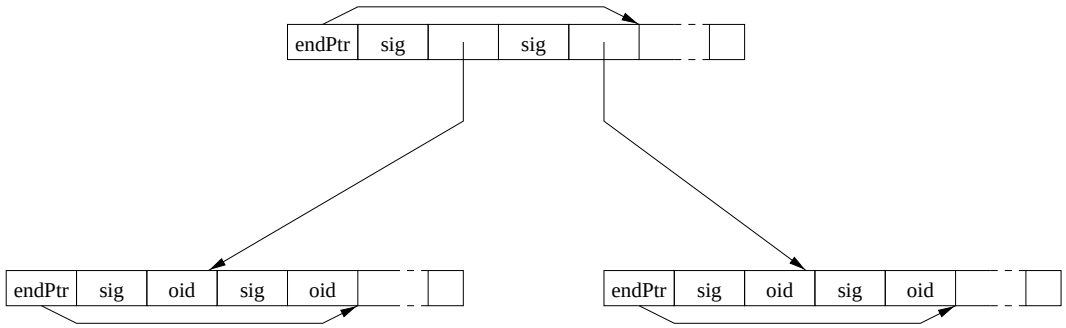


Figure 7.4: Pages of a signature tree

Implementation details

A Signature Tree includes a root object, which contains general information on the tree and is copied into main memory upon opening the index. Table 7.3 shows the contents of the root object.

We distinguish two different kinds of nodes in a signature tree, inner nodes and leaf nodes. The first 4 bytes of each page are identical. They are used to store the offset of the first free byte on a page. The pages of inner nodes contain pairs of signatures and 8 byte references (page numbers and area numbers) to child nodes. In leaf pages we store EOS object identifiers of data items instead of page references (see Figure 7.4).

7.2.3 Extendible Signature Hashing (ESH)

General description

An extendible signature hashing index (ESH) [46] is a partitioned filter index based on extendible hashing [26]. It is divided into two parts, a directory and buckets. In the buckets we store the signature/reference pairs of all data items. We determine the bucket into which a signature/reference pair is inserted by looking at a prefix $sig_d(o_i.A)$ of d bits of a signature. For each possible bit combination of the prefix we find an entry in the directory pointing to the corresponding bucket. The directory has 2^d entries, where

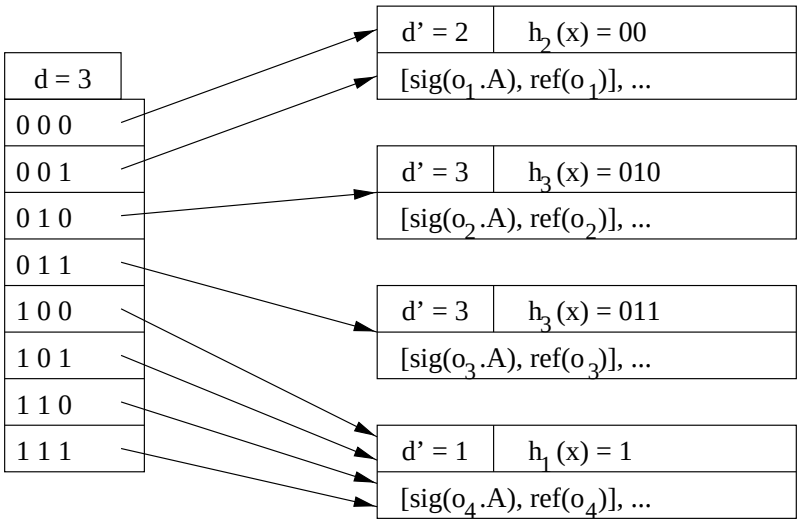


Figure 7.5: Extendible signature hashing (ESH)

d is called *global depth*. When a bucket overflows, this bucket is split and all its entries are divided among the two resulting buckets. In order to determine the new home of a signature the length of the inspected prefix has to be increased until at least two of the signature prefixes are distinct. The size of the current prefix d' of a bucket is called *local depth*. If we notice after a split that the local depth d' of a bucket is larger than the global depth d, we have to increase the size of the directory. This is done by doubling the directory. Pointers to buckets that have not been split are just copied. For the split bucket the new pointers are put into the directory and the global depth is increased (see Figure 7.5). We stop splitting the directory beyond a global depth of 20 and start using chained overflow buckets at this point, as further splitting leads to a too large directory, which in turn would slow down subset and superset queries considerably.

Name	*Description*
depth	the global depth of the hash table
sizeOfSig	the size of the signatures (in bits)
noOfSetBits	k, the number of bits set in signature for each element
pageSize	the size of the pages in the database (in bytes)
rootAreaNo	the area number of the root node of the directory
rootPageNo	the page number of the root node of the directory
maxPossDepth	the maximal global depth of the hash table (before chaining)

Table 7.4: Root object of an ESH index

The evaluation of an equality query is straightforward. We look up the entry for $sig_d(Q)$ in the directory, fetch the content of the corresponding bucket, check the full signatures, and eliminate all false drops. In order to find all subsets (supersets) of a query set Q, we determine all buckets to be fetched. We do this by generating all subsets (supersets) of

$sig_d(Q)$ with the algorithm by Vance and Maier (see Section 2.2). Then we access the corresponding buckets sequentially (by ascending page number), taking care not to access a bucket more than once. Afterwards we check the full signatures and eliminate the false drops.

ESH is similar to Quickfilter by Zezula, Rabitti, and Tiberio [100]. However, we use extendible hashing instead of linear hashing as our underlying hashing scheme and we also optimize the bucket accesses.

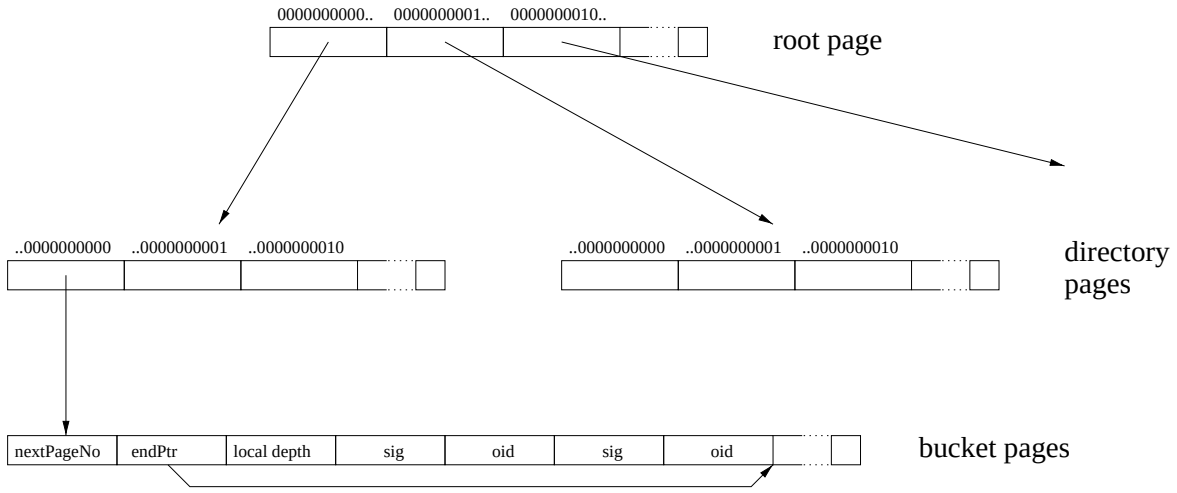


Figure 7.6: Pages of an ESH index

Implementation details

Like in the other index structures the general information of an ESH index is kept in a root object. The structure of an ESH index root object can be seen in Table 7.4.

We have limited the size of the directory to 2^{20} entries. Obviously this directory does not fit on a 4K page, therefore we implemented a hierarchical directory (see Figure 7.6). On the root page, which is always kept in main memory, we evaluate the first 10 bits of the query signature. Then we branch to the corresponding directory page to evaluate the other 10 bits of the query signature and fetch the corresponding bucket reference. A bucket page contains a reference to the next page of the bucket (in case of overflow records), a pointer to the end of the data on this page, the local depth of the bucket, and the signatures and references of all data items belonging to this bucket.

7.2.4 Recursive Linear Signature Hashing (RLSH)

General description

The fast growing directory of extendible hashing poses a considerable problem. Every time the global depth increases, the size of the directory doubles. This gave researchers

the impetus to find dynamic hashing techniques with directories that do not grow as rapidly. Litwin devised linear hashing [65], which was refined to recursive linear hashing by Ramamohanarao and Sacks-Davis [76]. First we give an introduction to linear hashing and then we describe recursive linear hashing and our modifications.

In linear hashing we start with a directory having 2^d entries. This is also called the d-th expansion, in which we reference 2^d buckets. Additionally we need a split pointer p that marks the next bucket to be split. Initially p points to the first entry of the directory. During an insertion if there is no space left in a bucket, an overflow bucket is created and chained to the original bucket. Whenever we have inserted L data items, we create a new bucket and extend the directory by adding an entry at the end of the directory at position $2^d + p$ that references this newly created bucket. Then the content of the p-th bucket is divided among the p-th and $2^d + p$-th bucket by rehashing the items. After that p is increased by 1. If p is equal to $2^d + 1$, i.e. the end of the current expansion has been reached, then we enter the $d + 1$-st expansion and reset p to 1.

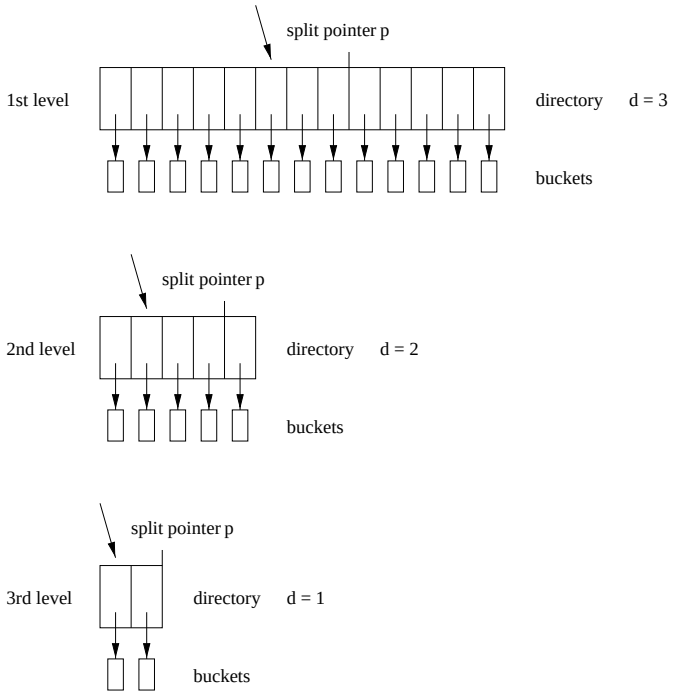


Figure 7.7: recursive linear hashing

Recursive linear hashing is a variant of linear hashing that avoids the use of overflow buckets by employing several hashtables on different levels (see Figure 7.7). Every time an overflow occurs, the inserted data item that caused the overflow is inserted into the hashtable on the next level. This rehashing continues recursively until a bucket with sufficient free space is found or no further hashtable exists. In the latter case a new hashtable is created (on the next-highest possible level) into which the data item is then inserted. Usually there are only two or three levels of recursive hash tables. Each level has its own value d_l for the expansion and its own split pointer p_l.

We modify recursive linear hashing by using signatures as hash keys, thus creating another partitioned filter index that we call recursive linear signature hashing (RLSH). The following recursive algorithm is used to search for a signature in an RLSH index:

```
search(level, sig(o_i)) {
    if(sig_{d_level}(o_i) ≥ 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 ≥ maxlevel) {
            data item not found;
        }
        else {
            search(level + 1, sig(o_i))
        }
    }
}
```

We start searching on the first level (calling *search* with parameter $level = 1$). If we do not find a qualifying signature on the current level, we go on searching on the next level. We continue until we have found the signature or no hashtables to search in are left. In the latter case we know that no data item with a qualifying signature is present in the database. When evaluation a subset or superset query, we have to generate all query signatures with the algorithm from Section 2.2 and initiate the corresponding subqueries.

Implementation details

The root object of a recursive linear signature hashing index can be found in Table 7.5. The physical layout of the index is illustrated in Figure 7.8.

We have separated the first level from others. There are two reasons for this. The first reason is that it is going to be larger than the other levels. So instead of storing the directory in a list (as we did with the directories of the recursive hashtables), we used a hierarchical structure. The second reason is that the first level is always accessed, regardless of the operation (retrieval, update, insertion or deletion). Therefore the reference to the top-level directory is not stored in the list accessible by *recLevel*, but directly in the root object. The data on the other levels (references to directories, values for expansions, split pointers, and number of inserted items) are stored in lists. As there are no overflow buckets, no information on chaining is needed in the buckets.

Name	*Description*
sizeOfSig	the size of the signatures (in bits)
noOfSetBits	k, the number of bits set in signature for each element
pageSize	the size of the pages in the database (in bytes)
rootAreaNo	the area number of the root node of the directory
rootPageNo	the page number of the root node of the directory
rootExpansion	d_1, expansion on the first level
rootSplitPtr	p_1, split pointer on the first level
rootNoOfEntries	number of data items inserted on first level
recLevel	ref. to list of refs to recursive hashtables
recExpansion	ref. to list of expansions of recursive hashtables
recSplitPtr	ref. to list of split pointers of recursive hashtables
recNoOfEntries	ref. to list with number of data items inserted on recursive levels
splitFactor	L, the number of insertions before expanding a directory

Table 7.5: Root object of an RLSH index

7.2.5 Inverted Files

General description

An inverted file (see Figure 7.9) consists of a directory containing all distinct values in the domain D and a list for each value consisting of the references to data items whose set-valued attribute contains this value. For an overview on traditional inverted files see [59, 81]). As done frequently we hold the search values of the directory in a B^+-tree, which makes it another hierarchical index. However, we modify the lists by storing the cardinality of the set-valued attribute with each data item reference. This enables us to answer queries efficiently by using the cardinalities as a quick pretest.

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

We use several well-known techniques to increase the performance of inverted lists. To reduce the size of the lists we compress them. We keep the lists sorted and encode the gaps using light-weight techniques [93]. It is also sensible to fetch the lists in increasing

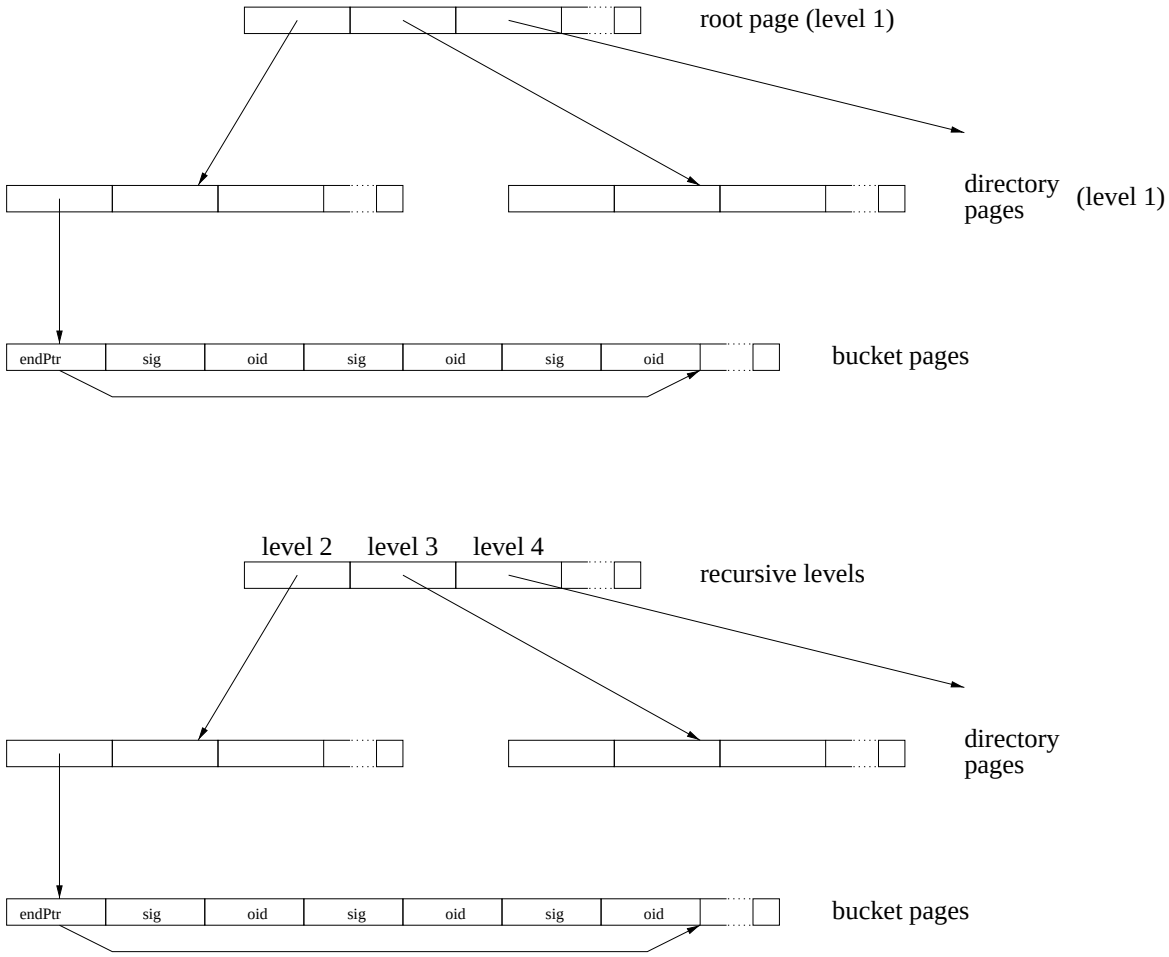


Figure 7.8: Pages of an RLSH index

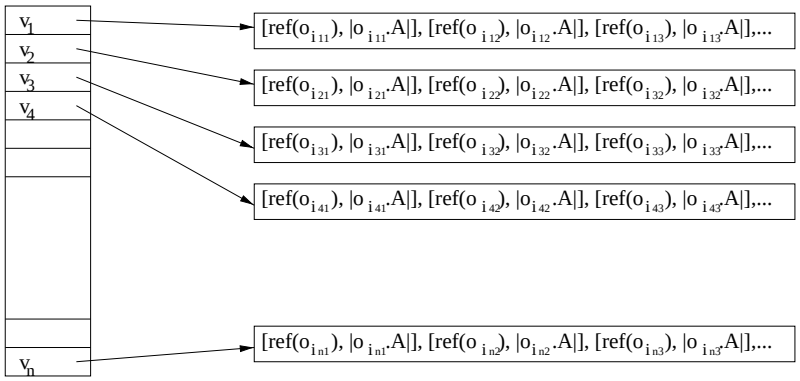


Figure 7.9: Inverted File

size and to use thresholding, i.e. instead of fetching lists beyond a certain threshold size (the number of pages needed to fetch all data items qualifying at the moment), we access

the data items immediately. For a more detailed explanation of these techniques see [103, 104].

Name	*Description*
noOfEntries	the number of data items inserted into index
depth	the height of the B^+-tree
pageSize	the size of the pages in the database (in bytes)
rootAreaNo	the area number of the root of the tree
rootPageNo	the page number of the root of the tree
maxNoOfEntries	maximal number of entries in a B^+-tree node

Table 7.6: Root object of an inverted file index

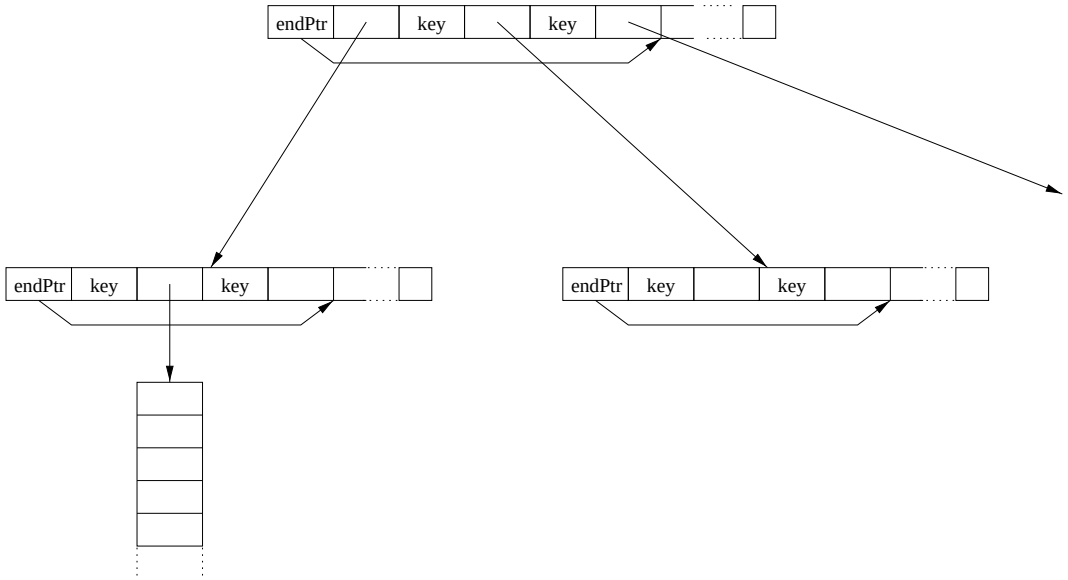


Figure 7.10: Pages of an inverted file index

Implementation details

The root object of an inverted file index and its description are depicted in Table 7.6.

The first 4 bytes of a node in the B^+-tree directory of an inverted file index are used to store the offset of the first free byte on a page. In leaf nodes we have search-keys along with the references to the corresponding list of data item references (which will be described later). In inner nodes we store references to child nodes which are separated by search keys. The size of search keys is 4 bytes, the size of references 8 bytes (see Figure 7.10).

Name	*Description*
compressionFlag	for a compressed list contains char 'c'
noOfOids	number of oids in the list
firstOid	first oid in uncompressed form
gapBits	number of bits used to code the gaps
setCardBits	number of bits used to code set cardinalities
oids	a list of compressed oids
set cardinalities	a list of compressed set cardinalitites

Table 7.7: Compressed list

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

Table 7.8: Uncompressed list

We distinguish two different types of lists, compressed and uncompressed. Lists containing fewer than 8 OIDs are not compressed to avoid overhead. Table 7.7 shows the internal structure of a compressed list.

An uncompressed list merely consists of a compression flag, which is set to 'u' in this case, and a list of oids and set cardinalities (see Table 7.8). The number of oids in the list is derived from the size of the list, which can be determined by an EOS function.

7.3 Mathematical Modeling

When comparing index structures, there are several principal avenues of approach: analytical approach, mathematical modeling, simulations, and experiments [102]. For the evaluation of the index structures we chose the following two approaches: a theoretical comparison by mathematical methods and a practical comparison by experiments. In order to be able to compare the index structures theoretically, we need to develop a mathematical model. In this section we present cost models for the index structures described in Section 7.2.

7.3.1 Preliminary Defininitions

Before discussing the cost models for the index structures we need to clarify some details. Table 7.9 summarizes all symbols used in the remainder of this section and their meaning.

All signature-based index structures store pairs of signatures and references, i.e. records of the form $[sig(o_i), ref(o_i)]$ have to be handled. We denote the size of such a record by

Symbol	*Definition*
α	utilization of space (ST)
avgsetsize	average cardinality of set-valued attributes (Inverted files)
β	utilization of space in B$^+$-tree directory (Inverted files)
b	number of bits in a signature
B	total number of pages needed to store all data items
bf	branching factor (ST)
card	space needed to store cardinality of a set (Inverted files)
comp	compression factor (Inverted files)
c_f	number of right drops (see Section 2.1.3)
c_r	number of false drops (see Section 2.1.3)
d	global depth of hash table (ESH)
D	domain from which elements for set-valued attribute A are chosen
d_l	expansion of level l (RLSH)
height	height of B$^+$-tree directory (Inverted files)
id	size of references in bits
key	size of key in B$^+$-tree in bits (Inverted files)
L	load factor (RLSH)
m	number of buckets (ESH)
maxlevel	number of levels (RLSH)
n	number of data items in the database
n_l	number of data items inserted into level l (RLSH)
P	number of bytes per page
p_l	split pointer on level l (RLSH)
r_l	number of data items actually stored on level l (RLSH)
u_l	number of buckets on level l (RLSH)

Table 7.9: Parameters used in cost models

S_{record}. The number of bytes occupied by such a record can be calculated as follows:

$$S_{record} = \left\lceil \frac{(b + id)}{8} \right\rceil \text{ bytes} \tag{7.2}$$

where b is the number of bits in a signature and id is the number of bits in a reference.

The cost models of all index structures have to consider the costs C_{fetch} for fetching data items during retrieval. We consider both right drops (c_r) and false drops (c_f). (For a quick reminder on how to calculate c_r and c_f see Section 2.1.3.) We define these costs beforehand to make the formulas in the following sections easier to read. We measure C_{fetch} in number of page accesses and approximate it by Yao's formula [98]. B is the total number of pages needed to store all n data items.

$$C_{fetch} = B \cdot \left(1 - \prod_{i=1}^{c_r+c_f} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1}\right) \tag{7.3}$$

7.3.2 Sequential Signature File

For each data item a record containing a signature and a reference is stored in a sequential signature file. The size of an SSF index, which we denote by S_{SSF}, can be computed easily (P is the number of bytes per page):

$$S_{SSF} = \left\lceil \frac{n \cdot S_{record}}{P} \right\rceil \text{pages} \tag{7.4}$$

Evaluating equality, subset, and superset queries with the help of a sequential signature file is straightforward. First we construct a signature sig(Q) for the query set Q. Then we traverse the signature file and compare sig(Q) with the signatures of all data items in the file. If a signature sig(o_i) of a data item matches the query signature sig(Q), we have to fetch the corresponding data item via the reference ref(o_i) and check for false drops.

We measure the costs for evaluating a query with an SSF index in number of pages that have to be accessed:

$$C^{SSF}_{=,\subseteq,\supseteq} = S_{SSF} + C_{fetch} \tag{7.5}$$

7.3.3 Signature Tree

We assume that nodes of a signature tree are mapped to disk pages and that each node is utilized to a degree α, i.e. $\alpha \cdot P$ bytes of each page are filled with data. It follows that the average branching factor, or fanout, of an inner node can be estimated by $bf = \alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor$. We approximate the size of an ST index similar to that of an R-tree [39]:

$$\begin{aligned} S_{ST} &= \sum_{i=1}^{\lceil \log_{bf}(n) \rceil} \left\lceil \frac{n}{\left(\alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor\right)^i} \right\rceil \text{ pages} \\ &= \left\lceil \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor} \right\rceil + \left\lceil \frac{n}{\left(\alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor\right)^2} \right\rceil + \ldots + 1 \text{ pages} \\ &\approx \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor - 1} \text{ pages} \end{aligned} \tag{7.6}$$

Let us now look at the query evaluation costs. We need to know the probability for each node m_j of an ST index that it will be accessed during query evaluation. The root

node is always accessed, the other nodes only if the signature of their bounding sets B_j match the query signature sig(Q). We have to distinguish between two different cases: equality/subset queries and superset queries. We will first look at the equality/subset case. The probability that a node will be accessed during query evaluation is

$$\Pr(sig(Q) \text{ retrieves } m_j \text{ in an equality/subset query}) = \Pr(sig(Q) \subseteq sig(B_j)) \quad (7.7)$$

Now the expected number of page accesses for a query with an equality or subset predicate can be estimated by

$$C^{ST}_{=,\subseteq} \leq 1 + \sum_{j=2}^{S_{ST}} \Pr(sig(Q) \subseteq sig(B_j)) + C_{fetch} \quad (7.8)$$

We always access the root node and the other nodes are accessed according to the assigned probability. We overestimate the actual costs, because in formula (7.8) we assume a child node can be accessed even if its parent node is not accessed.

The costs for superset queries are calculated similarly. We just have to exchange the probability for accessing a node by

$$\Pr(sig(Q) \text{ retrieves } m_j \text{ in a superset query}) = \Pr(sig(Q) \cap sig(B_j) \neq \emptyset) \quad (7.9)$$

Hence the costs for evaluating a query with a superset predicate can be estimated by

$$C^{ST}_{\supseteq} \leq 1 + \sum_{j=2}^{S_{ST}} \Pr(sig(Q) \cap sig(B_j) \neq \emptyset) + C_{fetch} \quad (7.10)$$

7.3.4 Extendible Signature Hashing

The size of an extendible signature hashing index S_{ESH} in pages depends on S_{DIR}, the size of the directory, and S_{BUCK}, the number of buckets actually created (some buckets will be shared, see also Section 7.2.3). For the cost model we assume that a bucket has exactly the size of a page.

$$S_{ESH} = S_{DIR} + S_{BUCK} \quad (7.11)$$

with

$$S_{DIR} = \left\lceil \frac{2^d \cdot id}{8 \cdot P} \right\rceil \text{ pages} \quad (7.12)$$

where d is the global depth of the hash table and

$$S_{BUCK} = m \text{ pages} \leq 2^d \text{ pages} \tag{7.13}$$

Estimating the number of buckets, m, in an ESH index may not be easy in practice. In [26] Fagin, Nievergelt, Pippenger, and Strong give an approximation for the average size of an extendible hashing index for uniform distribution of the hashing keys, which translates to ESH as follows.

$$\overline{S}_{ESH} \approx \left\lceil \frac{n \cdot S_{record}}{P} \right\rceil \cdot \log_2 e \text{ pages} \tag{7.14}$$

For the query evaluation costs we have to distinguish two cases: equality queries and subset/superset queries. We need very few page accesses within an ESH index when evaluating an equality query: just one access to the directory and one access to a bucket.

$$C_{=}^{ESH} = 2 + C_{fetch} \tag{7.15}$$

Next we discuss the query evaluation costs for subset/superset queries. Using optimized values for the signature parameters k and b, the expected weight of a signature is $\frac{b}{2}$. Assuming that the bits set in a signature are uniformly distributed, the expected weight of a prefix of d bits of a signature is $\frac{d}{2}$. So we expect to generate $2^{\frac{d}{2}}$ subqueries for subset/superset queries. As for each subquery we have to access the directory and a bucket, the total number of page accesses during a query evaluation can be estimated by

$$\begin{aligned} C_{\subseteq,\supseteq}^{ESH} &\approx 2^{\lceil \frac{d}{2} \rceil} \cdot 2 + C_{fetch} \\ &= 2^{\lceil \frac{d}{2} \rceil + 1} + C_{fetch} \end{aligned} \tag{7.16}$$

7.3.5 Recursive Linear Signature Hashing

The total size of the recursive linear hashing index can be determined by adding the sizes of the directories and buckets of all recursive hash tables.

$$S_{RLSH} = \sum_{l=1}^{maxlevel} (S_{DIR_l} + S_{BUCK_l}) \tag{7.17}$$

In order to calculate S_{DIR_l} and S_{BUCK_l} we have to know several parameters for each level: the number of buckets u_l, the expansion d_l (which corresponds to the global depth in ESH), and the split pointer p_l.

$$u_l = \left\lceil \frac{n_l}{L} \right\rceil \tag{7.18}$$

$$d_l = \lfloor \log_2(u_l) \rfloor \tag{7.19}$$

$$p_l = u_l \bmod d_l \tag{7.20}$$

Only r_l data items of the n_l items inserted at level l 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

$$n_{l+1} = n_l - r_l \tag{7.21}$$

We insert every data item into the first level hash table, so $n_1 = n$. For uniform distribution of the hash keys Ramamohanarao and Sacks-Davis present an approximation for r_l in [76]

$$r_l \approx 2 \cdot p_l \cdot \mathit{left}_l + (2^{d_l} - p_l) \cdot \mathit{right}_l \tag{7.22}$$

where left_l is the estimated number of data items per bucket on the left side of the split pointer, and right_l is the estimated number of data items per bucket on the right side of the split pointer. For details on how to calculate left_l and right_l see Appendix B. Assuming that the size of a bucket is one page, we can now calculate S_{DIR_l} and S_{BUCK_l}.

$$S_{DIR_l} = \left\lceil \frac{(2^{d_l} + p_l) \cdot id}{8 \cdot P} \right\rceil \text{ pages} \tag{7.23}$$

$$S_{BUCK_l} = u_l \text{ pages} \tag{7.24}$$

Regular recursive linear hashing with unique keys distinguishes between successful and unsuccessful searches, because when a matching key is found the search can be stopped immediately. On account of non-unique keys in RLSH, we cannot abort a search prematurely. We have to search all recursive hash tables.

While evaluating an equality query we have one access to the directory and one access to a bucket at each level, therefore the total costs are

$$C_{=}^{RLSH} = 2 \cdot \mathit{maxlevel} + C_{fetch} \tag{7.25}$$

We now take a look at subset/superset queries. Assuming optimized signature parameters we expect the query signature $\mathrm{sig}(Q)$ to have an average weight of $\frac{b}{2}$. At each level $\frac{d_l}{2}$ of these bits are relevant on average and cause $2^{\frac{d_l}{2}}$ lookups in the hash table on level l. As a lookup means one access to the directory and one access to a bucket we get

$$\begin{aligned} C_{\subseteq,\supseteq}^{RLSH} &\approx \left(\sum_{l=1}^{maxlevel} 2^{\lceil \frac{d_l}{2} \rceil} \cdot 2 \right) + C_{fetch} \\ &= \left(\sum_{l=1}^{maxlevel} 2^{\lceil \frac{d_l}{2} \rceil + 1} \right) + C_{fetch} \end{aligned} \tag{7.26}$$

7.3.6 Inverted Files

An inverted file consists of two parts, a directory, in our case a B$^+$-tree, and a set of lists. So we calculate the size of an inverted file by

$$S_{INV} = S_{TREE} + S_{LIST} \tag{7.27}$$

We derive S_{TREE} similarly to the size of a signature tree. In a signature tree we store records consisting of signatures and references. In a B$^+$-tree directory we have an entry for each distinct value in the domain D that occurs in an indexed set. In the inner nodes of the B$^+$-tree we store records of the form $[keyvalue, reference]$ with $keyvalue \in D$. The size of such a record is

$$S_{keyrecord} = \left\lceil \frac{key + id}{8} \right\rceil \text{ bytes} \tag{7.28}$$

We denote the size of the *keyvalues* in bits by key. Now we can estimate the size of the directory of an inverted file similarly to the size of a ST index. The utilization of the inner nodes is given by β.

$$\begin{aligned} S_{TREE} &= \sum_{i=1}^{|D|} \left\lceil \frac{n}{\left(\beta \cdot \left\lfloor \frac{P}{S_{keyrecord}} \right\rfloor\right)^i} \right\rceil \text{ pages} \\ &\approx \frac{n}{\beta \cdot \left\lfloor \frac{P}{S_{keyrecord}} \right\rfloor - 1} \text{ pages} \end{aligned} \tag{7.29}$$

Let us now look at the size of the lists. For each element in a set we insert a record consisting of the cardinality of the set and a data item reference into a list. Therefore the size of such a record is

$$S_{listrecord} = \left\lceil \frac{card + id}{8} \right\rceil \text{ bytes} \tag{7.30}$$

Due to compression we also have to consider a compression factor *comp*. We approximate the total size of the lists by

$$S_{LIST} = \left\lceil S_{listrecord} \cdot \frac{n \cdot avgsetsize}{P \cdot comp} \right\rceil \text{ pages} \tag{7.31}$$

We evaluate queries very similarly for all query types (equality, subset, and superset). For each element in the query set we have to traverse the B$^+$-tree directory and fetch the corresponding list. The average length of a list can be approximated by dividing S_{LIST} by $|D|$

$$C^{INV}_{=,\subseteq,\supseteq} \approx |Q| \cdot \left(\frac{S_{LIST}}{|D|} + height \right) + C'_{fetch} \tag{7.32}$$

An estimation for the height of a B$^+$-tree is given in [3]

$$\log_{\left\lceil \frac{P}{S_{keyrecord}} \right\rceil}(|D|+1) \leq height \leq \log_{\left\lceil \frac{P}{2 \cdot S_{keyrecord}} \right\rceil} \left(\frac{|D|+1}{2} \right) \tag{7.33}$$

We also have to modify the costs C_{fetch} for fetching data items, because in an inverted file index we do not have any false drops, so

$$C'_{fetch} = B \cdot \left(1 - \prod_{i=1}^{c_r} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1} \right) \tag{7.34}$$

7.4 Mathematical Comparison

After introducing cost models for all index structures in the last section we are now ready to compare the access methods theoretically. First of all we look at the size of the index structures, then we compare the retrieval costs. We measure the size of an index structure in pages, while we express the retrieval costs in number of page accesses, i.e. we assume that the needed CPU-time can be neglected.

7.4.1 Comparison of Index Size

We express the costs of all index structures in terms of the cost of sequential signature files to make the comparison easier.

Sequential Signature Files

Let us recall the storage costs of a sequential signature file index:

$$S_{SSF} = \left\lceil \frac{n \cdot S_{record}}{P} \right\rceil \text{pages} \tag{7.35}$$

Signature Trees

As we have already mentioned the organization of a signature tree is very similar to that of an R-tree. The typical storage utilization of an R-tree with uniform distribution of the keys and a linear split algorithm is about 64% [4]. So we can approximate S_{ST} by

$$\begin{aligned} S_{ST} &\approx \frac{n}{\alpha \cdot \left\lfloor \frac{P}{S_{record}} \right\rfloor - 1} \text{ pages} \\ &\approx \frac{n}{\alpha \cdot \frac{P}{S_{record}}} \text{ pages} \\ &= \frac{1}{0.64} \cdot S_{SSF} \\ &\approx 1.56 \cdot S_{SSF} \end{aligned} \tag{7.36}$$

Extendible Signature Hashing

Using the approximation (7.14) given in [26], we can estimate the size of an extendible signature hashing index as follows

$$\begin{aligned} \overline{S}_{ESH} &\approx \left\lceil \frac{n \cdot S_{record}}{P} \right\rceil \cdot \log_2 e \text{ pages} \\ &\approx 1.44 \cdot S_{SSF} \end{aligned} \tag{7.37}$$

Recursive Linear Signature Hashing

Before evaluating the size of a recursive linear signature hashing index we have to determine an appropriate load factor L. We have to consider two effects when increasing the load factor. On one hand the size of the index will decrease, as the space in the buckets is utilized better. On the other hand a high load factor also means a larger number of recursive hash tables, because buckets overflow more frequently. This in turn will lead to higher query evaluation costs. Usually, for a recursive linear hashing index a load factor that fills 80% to 90% of a bucket is chosen [76]. For a recursive linear signature hashing index, however, this is not quite optimal.

As we cannot abort a search prematurely and may generate many subqueries for subset/superset queries, it is even more important to keep the number of recursive levels at a minimum. This is a point in favor of a low load factor. We have to be careful not to overshoot the mark, however. A very low load factor results in more frequent splits of buckets, which in turn leads to larger expansions of the hash table directories. A large expansion size has a negative influence on the retrieval costs of subset/superset queries, as the query evaluation costs grow exponentially with the expansions size. We obtained the best results for a load factor that fills about 60% of a bucket (see Figure 7.11):

$$L = \frac{0.6 \cdot P}{S_{record}} \tag{7.38}$$

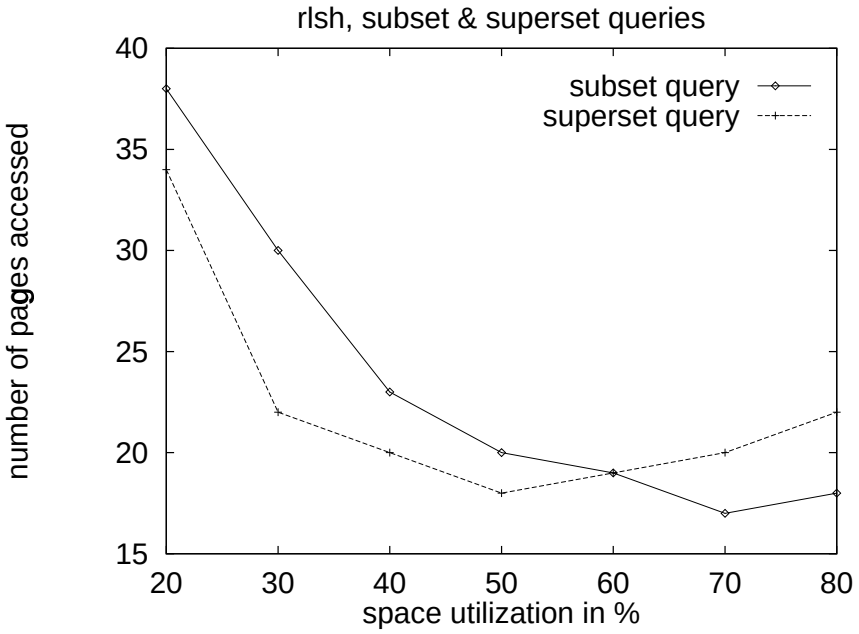


Figure 7.11: Query evaluation costs for different space utilizations

Having determined the load factor L we now continue to estimate the size of an RLSH index.

$$\begin{aligned} S_{RLSH} &= \sum_{l=1}^{maxlevel} (S_{DIR_l} + S_{BUCK_l}) \text{ pages} \\ &= \sum_{l=1}^{maxlevel} \left(\frac{u_l \cdot id}{8 \cdot P} + u_l \right) \text{ pages} \\ &= \sum_{l=1}^{maxlevel} \left(\frac{n_l \cdot id}{L \cdot 8 \cdot P} + \frac{n_l}{L} \right) \text{ pages} \\ &= \sum_{l=1}^{maxlevel} \left(1 + \frac{id}{8 \cdot P} \right) \cdot \frac{n_l}{L} \text{ pages} \end{aligned} \tag{7.39}$$

Ramamohanarao and Sacks-Davis show that there are at most two to three recursive hash tables in a recursive linear hashing index, which are very small compared to the hash table on the top level [76]. Almost all data items can be inserted into the intended hash table, i.e., $r_l \approx n_l$ and especially $r_1 \approx n_1 = n$.

$$S_{RLSH} \approx \left(1 + \frac{id}{8 \cdot P} \right) \cdot \frac{n}{L} \text{ pages}$$

$$
\begin{aligned}
&= \left(1 + \frac{id}{8 \cdot P}\right) \cdot \frac{n \cdot S_{record}}{0.6 \cdot P} \text{ pages} \\
&= \frac{\left(1 + \frac{id}{8 \cdot P}\right)}{0.6} \cdot S_{SSF} \qquad (7.40)
\end{aligned}
$$

We use pages with a size of 4096 bytes and references with a size of 64 bits. Inserting these values into (7.40) we get

$$S_{RLSH} \approx 1.67 \cdot S_{SSF} \qquad (7.41)$$

Inverted Files

Last but not least we estimate the size of an inverted file index in terms of an SSF index.

$$
\begin{aligned}
S_{INV} &= \frac{n}{\beta \cdot \frac{P}{S_{keyrecord}} - 1} + \frac{n \cdot avgsetsize}{P \cdot comp} \cdot S_{listrecord} \text{ pages} \\
&\approx \frac{n}{\beta \cdot P} \cdot S_{keyrecord} + \frac{n \cdot avgsetsize}{P \cdot comp} \cdot S_{listrecord} \text{ pages} \\
&= \frac{S_{record}}{S_{record}} \cdot \frac{n}{P} \cdot \left(\frac{1}{\beta} \cdot S_{keyrecord} + \frac{avgsetsize}{comp} \cdot S_{listrecord}\right) \text{pages} \\
&= \frac{1}{S_{record}} \cdot \left(\frac{1}{\beta} \cdot S_{keyrecord} + \frac{avgsetsize}{comp} \cdot S_{listrecord}\right) \cdot S_{SSF} \qquad (7.42)
\end{aligned}
$$

In [97] Yao gives an approximation of the asymptotic space utilization of a B^+-tree, it is about $\ln 2 \approx 69\%$. Furthermore we achieve a compression factor of about 8 with our simple compression technique. As we use keyvalues with a size of 32 bits, references with sizes of 64 bits, and 16 bits to store the cardinality of a set (7.42) simplifies to

$$
\begin{aligned}
S_{INV} &\approx \frac{1}{S_{record}} \cdot \left(\frac{1}{\ln 2} \cdot S_{keyrecord} + \frac{avgsetsize}{8} \cdot S_{listrecord}\right) \cdot S_{SSF} \\
&\approx \frac{17.31 + 1.25 \cdot avgsetsize}{S_{record}} \cdot S_{SSF} \qquad (7.43)
\end{aligned}
$$

Summary

Table 7.10 sums up our size comparison of the different index structures.

7.4.2 Comparison of Retrieval Costs

In this section we compare the retrieval costs of the different index structures. Again we use the costs of sequential signature files as a reference.

index structure	*size*
SSF	S_{SSF}
ST	$1.56 \cdot S_{SSF}$
ESH	$1.44 \cdot S_{SSF}$
RLSH	$1.67 \cdot S_{SSF}$
INV	$\frac{17.31+1.25 \cdot avgsetsize}{S_{record}} \cdot S_{SSF}$

Table 7.10: (Theoretical) size of index structures

Sequential Signature Files

As a reminder let us quote the retrieval costs for sequential signature files from Section 7.3.2. These costs are the same for all query types, we just have to traverse the signature file and access all drops.

$$C^{SSF}_{=,\subseteq,\supseteq} = S_{SSF} + C_{fetch} \tag{7.44}$$

Signature Trees

Equality and subset queries are evaluated differently from superset queries in signature trees, so we have to look at two different cases.

Evaluating a containment query in an R-tree is very similar to evaluating equality and subset queries in a signature tree. Guttman mentions in [39] that on average 10% of all pages in an R-tree are traversed during a containment query. So we can approximate the retrieval costs of equality and subset queries in signature trees by (assuming $\alpha \approx 0.64$)

$$\begin{aligned} C^{ST}_{=,\subseteq} &\approx \frac{1}{10} \cdot S_{ST} + C_{fetch} \\ &\approx \frac{1}{10} \cdot \frac{1}{\alpha} \cdot S_{SSF} + C_{fetch} \\ &\approx 0.156 \cdot S_{SSF} + C_{fetch} \end{aligned} \tag{7.45}$$

As long as the space utilization α is large enough, ST is going to be considerably faster than a sequential signature file.

For superset queries, however, signature trees lack performance. The percentage of pages we have to fetch during query evaluation is significantly higher than for equality and subset queries. In our experimental tests we found that 95% to 100% of the pages in a signature tree were accessed during query evaluation.

$$C^{ST}_{\supseteq} \approx S_{ST} + C_{fetch}$$

$$\begin{aligned} &\approx \frac{1}{\alpha} \cdot S_{SSF} + C_{fetch} \\ &\approx 1.56 \cdot S_{SSF} + C_{fetch} \end{aligned} \tag{7.46}$$

So for superset queries signature trees are even slower than sequential signature files, as α is always between 0 and 1.

Extendible Signature Hashing

We also have to distinguish between two cases for extendible signature hashing index structures. Equality queries are evaluated differently from subset and superset queries.

For equality queries we need only two page accesses to find the bucket containing candidate signatures. In a second step we have to access the corresponding data items and filter out false drops.

$$C_{=}^{ESH} = 2 + C_{fetch} \tag{7.47}$$

To figure out the retrieval costs of ESH for subset and superset queries we have to determine the global depth of the hash table:

$$C_{\subseteq,\supseteq}^{ESH} \approx 2^{\lceil \frac{d}{2} \rceil + 1} + C_{fetch} \tag{7.48}$$

We can approximate the global depth d by looking at the size of an ESH index structure.

$$S_{ESH} = S_{DIR} + S_{BUCK} = \frac{2^d \cdot id}{8 \cdot P} + m \approx \log_2 e \cdot S_{SSF} \tag{7.49}$$

In Section 7.3.4 we only gave a rough estimation for m, the number of buckets. That is to say we only determined an upper bound for m ($m \leq 2^d$). However, we can do better than that: we can also give a lower bound for m. A sequential signature file represents the smallest possible space in which we can store the signature/reference records of all indexed data items. So we know that

$$S_{SSF} \leq m \leq 2^d \tag{7.50}$$

Let us now look at the two extreme cases $m = m_1 = S_{SSF}$ and $m = m_2 = 2^d$ in turn.

$$\begin{aligned} \frac{2^{d_{m_1}} \cdot id}{8 \cdot P} + S_{SSF} &\approx \log_2 e \cdot S_{SSF} \\ \frac{2^{d_{m_1}} \cdot id}{8 \cdot P} &\approx (\log_2 e - 1) \cdot S_{SSF} \\ d_{m_1} &\approx \log_2(\log_2 e - 1) + \log_2(S_{SSF}) + \log_2(P) - \log_2\left(\frac{id}{8}\right) \end{aligned} \tag{7.51}$$

We use pages with a size of $P = 4096$ bytes and references with a size of $id = 64$ bits. Inserting these values into the above formula we get

$$d_{m_1} \approx \log_2(S_{SSF}) + 8 \tag{7.52}$$

Next we estimate d for $m = m_2 = 2^d$.

$$\begin{aligned}
\frac{2^{d_{m_2}} \cdot id}{8 \cdot P} + 2^d &\approx \log_2 e \cdot S_{SSF} \\
2^{d_{m_2}} &\approx \frac{\log_2 e \cdot S_{SSF}}{\left(\frac{id}{8 \cdot P} + 1\right)} \\
2^{d_{m_2}} &\approx \frac{\log_2 e \cdot S_{SSF} \cdot P}{\frac{id}{8} + P} \\
d_{m_2} &\approx \log_2(\log_2 e) + \log_2(S_{SSF}) + \log_2(P) - \log_2\left(\frac{id}{8} + P\right)
\end{aligned} \tag{7.53}$$

Inserting P and id:

$$d_{m_2} \approx \log_2(S_{SSF}) + 0.5 \tag{7.54}$$

Inserting d_{m_1} and d_{m_2} into (7.48) we get

$$\begin{aligned}
C_{\subseteq,\supseteq}^{ESH/m_1} &= 2^{\frac{(\log_2(S_{SSF})+8)}{2}+1} + C_{fetch} \\
&= \sqrt{S_{SSF}} \cdot 2^5 + C_{fetch} \\
&= 32 \cdot \sqrt{S_{SSF}} + C_{fetch}
\end{aligned} \tag{7.55}$$

and

$$\begin{aligned}
C_{\subseteq,\supseteq}^{ESH/m_2} &= 2^{\frac{(\log_2(S_{SSF})+0.5)}{2}+1} + C_{fetch} \\
&= \sqrt{S_{SSF}} \cdot 2^{1.25} + C_{fetch} \\
&\approx 2.38 \cdot \sqrt{S_{SSF}} + C_{fetch}
\end{aligned} \tag{7.56}$$

Having 2^d buckets is favorable, because in this case the keys are spread (uniformly) across all buckets and frequent overflows will be unlikely. Having S_{SSF} buckets is very unfavorable, because all keys are crammed into the lowest possible number of buckets, which means that the data is heavily skewed. This in turn leads to frequent overflows and a rapid growth of the directory.

Recursive Linear Signature Hashing

For the RLSH index the number of recursive levels ultimately determines the performance. Similar to the ESH index we look at two extreme cases. We get the best performance when we have no recursive tables at all and everything is stored on the top level. In the worst case scenario all data items have the same hash key, which means that all of them are mapped to the same bucket. So we have one completely filled bucket on each level and all other buckets are empty. This results in a total number of $\frac{n \cdot S_{record}}{P} = S_{SSF}$ levels. Let us analyze the costs for $maxlevel_1 = 1$ and $maxlevel_2 = S_{SSF}$.

For equality queries the best case is as good as ESH, we merely need two page accesses.

$$\begin{aligned} C_{=}^{RLSH/maxlevel_1} &= 2 \cdot maxlevel_1 + C_{fetch} \\ &= 2 + C_{fetch} \end{aligned} \tag{7.57}$$

The worst case behavior of RLSH, however, is even worse than SSF.

$$\begin{aligned} C_{=}^{RLSH/maxlevel_2} &= 2 \cdot maxlevel_2 + C_{fetch} \\ &\approx 2 \cdot S_{SSF} + C_{fetch} \end{aligned} \tag{7.58}$$

Let us now turn to the retrieval costs for subset/superset queries. For the best case the performance of RLSH keeps up with that of ESH.

$$C_{\subseteq,\supseteq}^{RLSH/maxlevel_1} = 2^{\frac{d_1}{2}} \cdot 2 + C_{fetch} \tag{7.59}$$

We have to determine the expansion d_1 of the top level, which depends on the number of buckets u_1 on the first level. (The formulas for d_1 and u_1 are taken from Section 7.3.5.)

$$d_1 = \log_2(u_1) \tag{7.60}$$

$$u_1 = \frac{n_1}{L} \tag{7.61}$$

As all data items are inserted into the top level hash table $n_1 = n$ holds. Inserting 7.60 into 7.59 we get

$$\begin{aligned} C_{\subseteq,\supseteq}^{RLSH/maxlevel_1} &= 2^{\frac{\log_2\left(\frac{n}{L}\right)}{2}} \cdot 2 + C_{fetch} \\ &= \sqrt{\frac{n}{L}} \cdot 2 + C_{fetch} \\ &\approx \sqrt{\frac{S_{SSF}}{0.6}} \cdot 2 + C_{fetch} \\ &\approx 2.58 \cdot \sqrt{S_{SSF}} + C_{fetch} \end{aligned} \tag{7.62}$$

In the worst case we have S_{SSF} hash tables, each with its own expansion d_l. As already mentioned, the expansion d_1 of the top level hash table is equal to $\log_s\left(\frac{n}{L}\right)$, because all data items are first inserted into the top level hash table. As each table stores $\frac{P}{S_{record}}$ less data items than its predecessor, we have to insert fewer and fewer data items into the lower tables. Generally

$$d_l = \log_2\left(\frac{n-(l-1)\cdot\frac{P}{S_{record}}}{L}\right) \tag{7.63}$$

We can now estimate the retrieval costs:

$$\begin{aligned}
C_{\subseteq,\supseteq}^{RLSH/maxlevel_1} &= \sum_{l=1}^{S_{SSF}} 2^{\frac{d_l}{2}}\cdot 2 + C_{fetch} \\
&= 2\cdot\sum_{l=1}^{S_{SSF}}\sqrt{\frac{n-(l-1)\cdot\frac{P}{S_{record}}}{L}} + C_{fetch} \\
&= 2\cdot\sum_{l=1}^{S_{SSF}}\sqrt{\frac{n}{L}-\frac{(l-1)\cdot\frac{P}{S_{record}}}{L}} + C_{fetch} \\
&\approx 2\cdot\sum_{l=1}^{S_{SSF}}\sqrt{\frac{S_{SSF}}{0.6}-\frac{(l-1)}{0.6}} + C_{fetch} \\
&\approx 2.58\cdot\sum_{l=1}^{S_{SSF}}\sqrt{S_{SSF}-(l-1)} + C_{fetch} \\
&= 2.58\cdot\sum_{l=1}^{S_{SSF}}\sqrt{l} + C_{fetch} \\
&\geq 2.58\cdot\sum_{l=\frac{S_{SSF}}{2}}^{S_{SSF}}\sqrt{\frac{S_{SSF}}{2}} + C_{fetch} \\
&= 2.58\cdot\sqrt{\frac{S_{SSF}}{2}}\cdot\left(S_{SSF}-\frac{S_{SSF}}{2}+1\right) + C_{fetch} \\
&\approx 2.58\cdot\sqrt{\frac{S_{SSF}}{2}}\cdot\frac{S_{SSF}}{2} + C_{fetch} \\
&= 2.58\cdot\left(\frac{S_{SSF}}{2}\right)^{\frac{3}{2}} + C_{fetch} \\
&\approx 0.91\cdot(S_{SSF})^{\frac{3}{2}} + C_{fetch}
\end{aligned} \tag{7.64}$$

In the worst case recursive linear signature hashing shows even worse behavior than a sequential signature file. The actual behavior of an RLSH index depends greatly on the kind of data that is inserted. This makes it very difficult to analyze the average case. As we will see in Sections 7.7 and 7.8 on experimental evaluation, RLSH shows close to best case behavior for uniformly distributed data, while performing very poorly for skewed data.

Inverted files

When comparing the retrieval costs of an inverted file index to a sequential signature file, we have to consider that inverted files do not yield false drops. Therefore the costs for fetching data items (denoted by C'_{fetch}, see (7.34)) will be lower.

$$
\begin{aligned}
C^{INV}_{=,\subseteq,\supseteq} &\approx |Q| \cdot \left(\frac{S_{LIST}}{|D|} + height \right) + C'_{fetch} \\
&= |Q| \cdot \left(\frac{S_{listrecord} \cdot n \cdot avgsetsize}{P \cdot comp \cdot |D|} + height \right) + \\
&\quad B - B \cdot \prod_{i=1}^{c_r} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1} \\
&= |Q| \cdot \left(\frac{S_{record}}{S_{record}} \cdot \frac{S_{listrecord} \cdot n \cdot avgsetsize}{P \cdot comp \cdot |D|} + height \right) + \\
&\quad B \cdot \frac{\prod_{i=c_r+1}^{c_r+c_f} \frac{n \cdot (1-\frac{1}{B}) - i + 1}{n - i + 1}}{\prod_{i=c_r+1}^{c_r+c_f} \frac{n \cdot (1-\frac{1}{B}) - i + 1}{n - i + 1}} - \\
&\quad B \cdot \frac{\prod_{i=c_r+1}^{c_r+c_f} \frac{n \cdot (1-\frac{1}{B}) - i + 1}{n - i + 1}}{\prod_{i=c_r+1}^{c_r+c_f} \frac{n \cdot (1-\frac{1}{B}) - i + 1}{n - i + 1}} \cdot \prod_{i=1}^{c_r} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1} \\
&= |Q| \cdot \left(\frac{S_{listrecord} \cdot avgsetsize}{S_{record} \cdot comp \cdot |D|} \cdot S_{SSF} + height \right) + \\
&\quad B \cdot \frac{\prod_{i=1}^{c_f} \frac{n \cdot (1-\frac{1}{B}) - c_r - i + 1}{n - c_r - i + 1} - 1}{\prod_{i=1}^{c_f} \frac{n \cdot (1-\frac{1}{B}) - c_r - i + 1}{n - c_r - i + 1}} + B \cdot \frac{1}{\prod_{i=1}^{c_f} \frac{n \cdot (1-\frac{1}{B}) - c_r - i + 1}{n - c_r - i + 1}} - \\
&\quad B \cdot \frac{1}{\prod_{i=1}^{c_f} \frac{n \cdot (1-\frac{1}{B}) - c_r - i + 1}{n - c_r - i + 1}} \cdot \prod_{i=1}^{c_r+c_f} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1} \\
&= |Q| \cdot \left(\frac{S_{listrecord} \cdot avgsetsize}{S_{record} \cdot comp \cdot |D|} \cdot S_{SSF} + height \right) + \\
&\quad B \cdot \left(1 - \prod_{i=1}^{c_f} \frac{n - c_r - i + 1}{n \cdot (1 - \frac{1}{B}) - c_r - i + 1} \right) + \\
&\quad \prod_{i=1}^{c_f} \frac{n - c_r - i + 1}{n \cdot (1 - \frac{1}{B}) - c_r - i + 1} \cdot B \cdot \left(1 - \prod_{i=1}^{c_r+c_f} \frac{n \cdot (1 - \frac{1}{B}) - i + 1}{n - i + 1} \right) \\
&= |Q| \cdot \left(\frac{S_{listrecord} \cdot avgsetsize}{S_{record} \cdot comp \cdot |D|} \cdot S_{SSF} + height \right) + \\
&\quad B - \prod_{i=1}^{c_f} \frac{n - c_r - i + 1}{n \cdot (1 - \frac{1}{B}) - c_r - i + 1} \cdot (B - C_{fetch})
\end{aligned}
\tag{7.65}
$$

The smaller the values for $|Q|$ (size of query set), for $S_{listrecord}$ (size of a record in a list), and for $avgsetsize$ (the average size of the indexed sets), the better the performance of an inverted file index when compared to a sequential signature file. Large values for S_{record}

(size of signature records), *comp* (compression factor), and $|D|$ (domain size) also favor inverted files.

Simplifying 7.65 for better comparability by inserting the values for $S_{listrecord}$ and *comp* we get:

$$C^{INV}_{=,\subseteq,\supseteq} \approx |Q| \cdot \left(\frac{1.25 \cdot avgsetsize}{S_{record} \cdot |D|} \cdot S_{SSF} + height \right) + C'_{fetch} \tag{7.66}$$

Summary

Table 7.11 gives an overview for the expected query evaluation costs for the different index structures.

index structure	*retrieval costs*		
	equality query	*subset query*	*superset query*
SSF	$S_{SSF} + C_{fetch}$		
ST	$0.156 \cdot S_{SSF} + C_{fetch}$		$1.56 \cdot S_{SSF} + C_{fetch}$
ESH	$2 + C_{fetch}$	best: $2.38 \cdot \sqrt{S_{SSF}} + C_{fetch}$ worst: $32 \cdot \sqrt{S_{SSF}} + C_{fetch}$	
RLSH	best: $2 + C_{fetch}$ worst: $2 \cdot S_{SSF} + C_{fetch}$	best: $2.58 \cdot \sqrt{S_{SSF}} + C_{fetch}$ worst: $0.91 \cdot (S_{SSF})^{\frac{3}{2}} + C_{fetch}$	
INV	$\|Q\| \cdot \left(\frac{1.25 \cdot avgsetsize}{S_{record} \cdot \|D\|} \cdot S_{SSF} + height \right) + C'_{fetch}$		

Table 7.11: Expected query evaluation costs

7.5 Environment of the Experiments

In addition to mathematical modeling we decided to do extensive experiments, because it is very difficult, if not impossible, to devise a formal model that yields reliable and precise results for non-uniform data distribution and average case behavior. A known problem of experimental evaluations is the sheer number of parameters and all their different combinations. As it is impossible to investigate all different aspects of the index structures (and for lack of a standardized benchmark for set-valued data) we present our specification of the experiments in detail in the following sections.

7.5.1 System Parameters

The experiments were conducted on a lightly loaded UltraSparc2 with 256 MByte main memory running under Solaris 2.6. The total disk space amounted to 10 GByte. All index structures were set atop the EOS storage manager, release 2.2, using the C++ interface of the manager [7]. We implemented the data structures and algorithms of

the index structures in C++ using the GNU C++ Compiler Version 2.8.1. The data structures were stored on 4K plain pages. The algorithms were not parallelized in any way. We allowed no buffering/caching of any sort, i.e. each benchmark was run under cold start conditions. We kept the storage manager from buffering pages read from disk by running the queries locally in the single-user mode of EOS (no client/server mode) and terminating all EOS processes after the processing of a query was done. For the next query EOS was restarted from scratch. We prevented the operating system from buffering by using RawIO instead of the file system. We cleared the internal disk cache of relevant pages by transferring 2 MBytes of data between the queries. Within a single run, the buffer was large enough to prevent accessing pages more than once.

7.5.2 Generating Data

parameter	*symbol*	*min value*	*max value*
database cardinality	$\|O\|$	50000	250000
set cardinality	$\|o_i.A\|$	5	15
domain cardinality	$\|D\|$	200	100000

Table 7.12: Parameters for generation of databases

As we designed the experiments we modeled the generated data on typical applications (like those mentioned in the introduction). We generated many different databases varying the cardinality of the database (in number of data items), the cardinality of the set-valued attributes (in number of elements contained), and the cardinality of the domain of the set elements. For a summary see table 7.12.

The data items in the databases were generated randomly. We investigated the performance of the index structures for uniformly distributed data and skewed data. For skewed data we decided to use a Zipf distribution (with $z = 1$), since various naturally occurring phenomena exhibit a certain regularity that can be described by it, e.g., word usage or population distribution [75]. A discrete Zipf distribution is defined by $P_z(x)$, which denotes the probability of event x occurring, with $x \in \{1, 2, \ldots, n\}$.

$$P_z(x) = \frac{1}{x^z} \cdot \frac{1}{H_n} \tag{7.67}$$

with

$$H_n = \sum_{i=1}^{n} \frac{1}{i^z} \tag{7.68}$$

Note that for $z = 0$ we obtain a discrete uniform distribution. Figure 7.12 shows the graphs of $P_1(x)$ and $P_0(x)$ for n=2000.

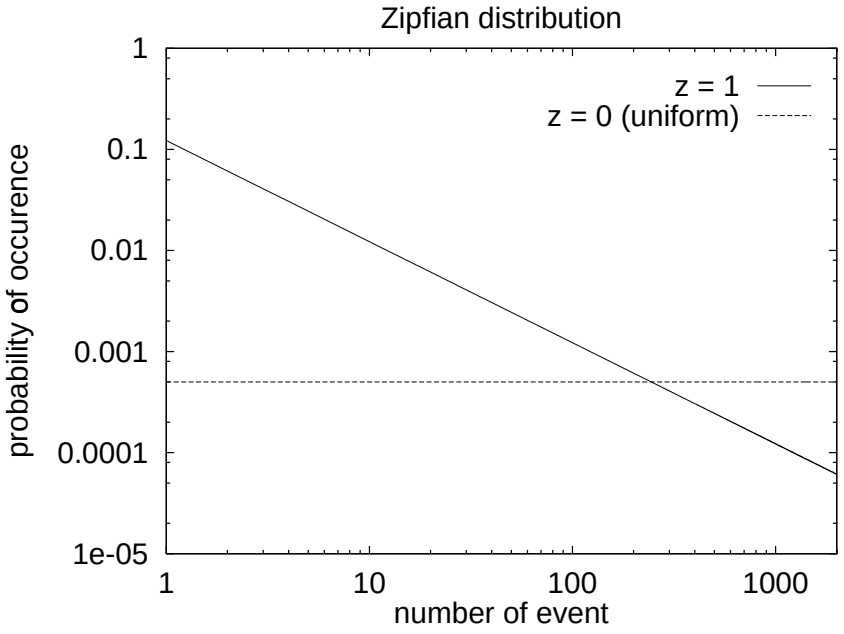


Figure 7.12: Zipf distribution

7.5.3 Generating Queries

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

The query selectivity is not a parameter we control explicitly. It is determined implicitly by the cardinality of the query set, the cardinality of the data sets, the cardinality of the domain, and the distribution of the data. Figure 7.13 shows the selectivities of different queries. Usually the selectivities are very low (around 10^{-5}). The exception to the rule are queries with subset predicates: for low query set cardinalities the selectivities are higher. This does not come as a great surprise. Assume uniformly distributed data for a moment. If we generate 100000 sets with an average cardinality of 10 out of a domain of 2000 elements (as we have done in Figure 7.13), we expect each element of the domain to appear in roughly 500 sets. So evaluating a containment query (query set cardinality = 1) will yield about 500 matches. For larger query set cardinalities the probability of finding the exact combination of elements as in the query set decreases rapidly. For skewed data the selectivities are much larger because a few elements of the domain are used frequently

in set-valued attributes of data items as well as in query sets.

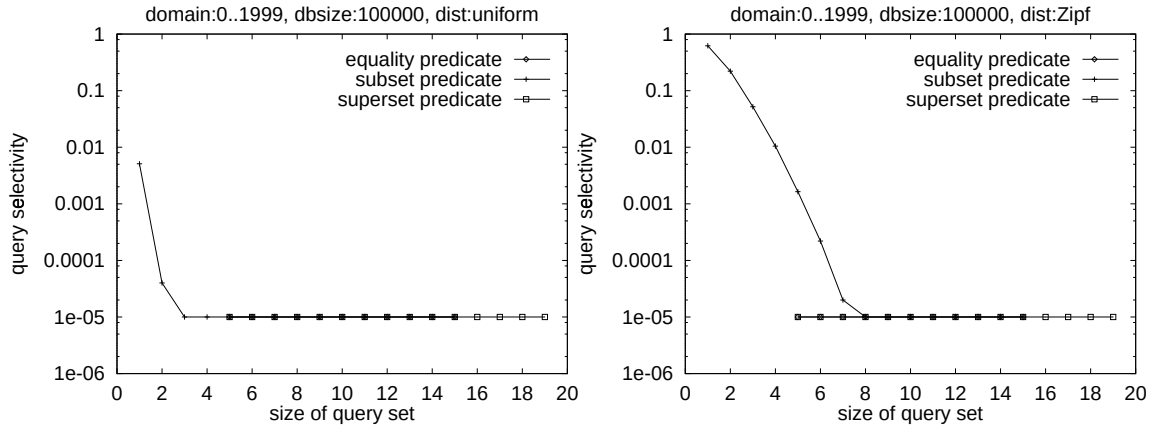


Figure 7.13: Selectivity of different queries

7.6 Tuning signature-based indexes

Signature-based indexes have to be tuned carefully to yield optimal performance. That is, the parameters k (the number of bits set per item) and b (the width of a signature) need to be chosen properly. Before comparing the indexes with each other, we determined the optimal parameters in our experimental environment for each query type. For each signature-based index structure we increased the width of the signature until no more performance improvement was visible. The optimal value for k was determined using the following formulas taken from [60]:

$$k_{opt=} = \frac{b \ln 2}{|Q|} \tag{7.69}$$

$$k_{opt\subseteq} = \frac{b \ln 2}{|o_i.A|} \tag{7.70}$$

$$k_{opt\supseteq} = \frac{b \ln 2}{|Q|} \tag{7.71}$$

Our observations are presented in the following sections.

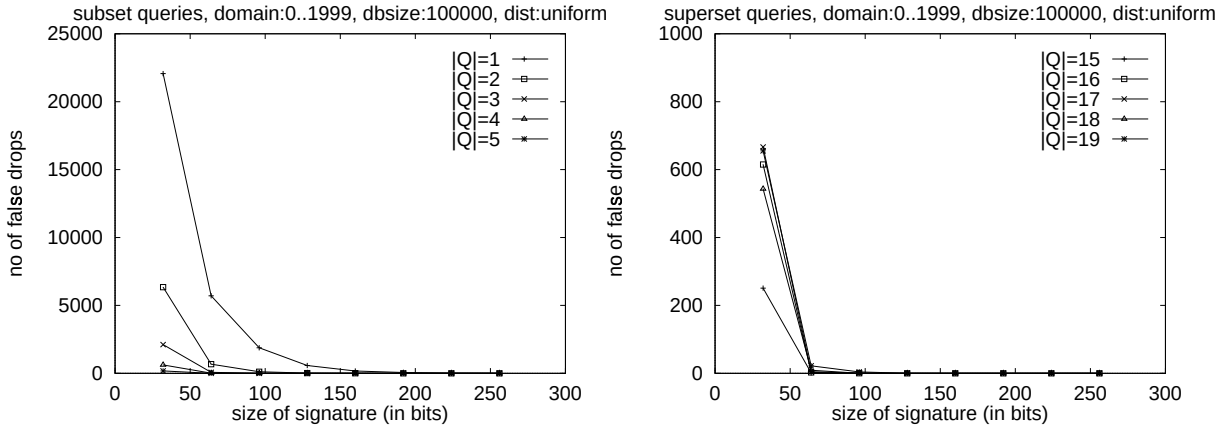
signature size (in bits)	32	64	96	128	160	192	224	256
index size (in 4K pages)	296	394	492	590	688	786	887	985

Table 7.13: Size of SSF index in 4K pages

7.6.1 Sequential signature files

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

The size of an SSF index increases proportionally to the size of the used signatures. This can be seen in Table 7.13.

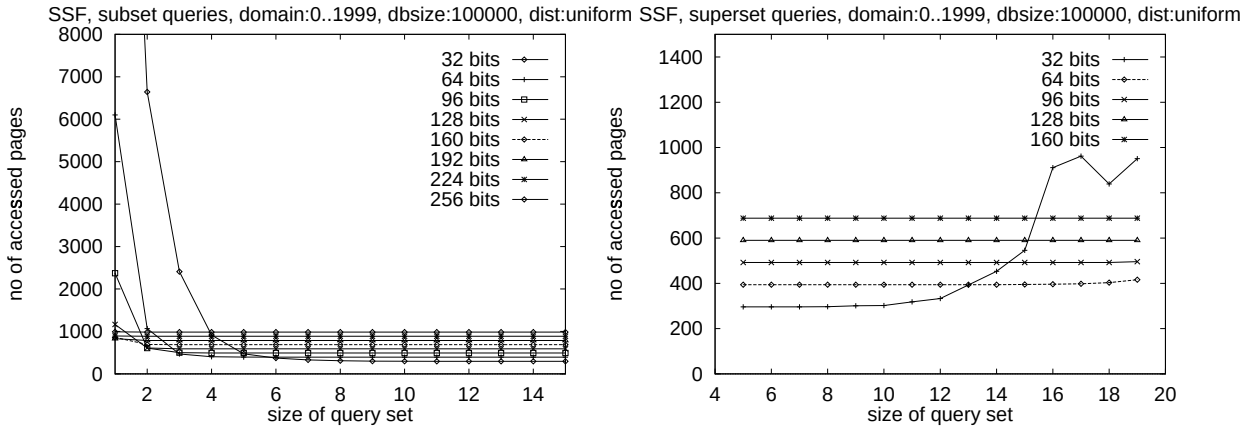


(a) Subset queries (b) Superset queries

Figure 7.14: Number of false drops for subset and superset queries

Let us now look at the query performance. For equality queries we had no false drops, regardless of the used signature size (32,64,96,128,160,192,224, or 256 bits). For subset and superset queries the number of false drops is depicted in Figure 7.14. As can be clearly seen, increasing the signature width decreases the number of false drops. The number of false drops also depends on the cardinality of the query set. Small query sets lead to "light" query signatures. For subset queries this increases the number of false drops (see Figure 7.14(a)). Large query sets show analogous behavior for superset queries (see Figure 7.14(b)).

The consequences for equality queries are clear. An index with a signature width of 32 bits suffices. For subset and superset queries we have to find a break-even point. Figure 7.15 shows the number of page accesses for subset and superset queries with different signature widths. Before drawing conclusions we should bring to mind that subset queries with small query sets and superset queries with large query sets are probably going to be predominant. Unfortunately, these are the cases where signatures show weak performance. For superset queries (Figure 7.15(b)) we found that a signature width of 64 bits gave us



(a) Subset queries (b) Superset queries

Figure 7.15: Number of page accesses for subset and superset queries (SSF)

the best overall performance. When averaging the page accesses for all query set sizes, the index with a signature of 64 bits comes out on top. When taking into consideration that larger query sets are more important, a signature size of 64 bits is by far superior to the other signature sizes. For subset queries (Figure 7.15(a)) the choice is harder to make. Applying a signature size of 96 or 128 bits yields the best average performance. However, this does not emphasize the important cases, i.e., small query sets, enough. A query set cardinality of 1 is crucial, as it represents containment queries (see Section 7.1.1). Therefore we chose a signature size of 160 bits.

7.6.2 Signature trees

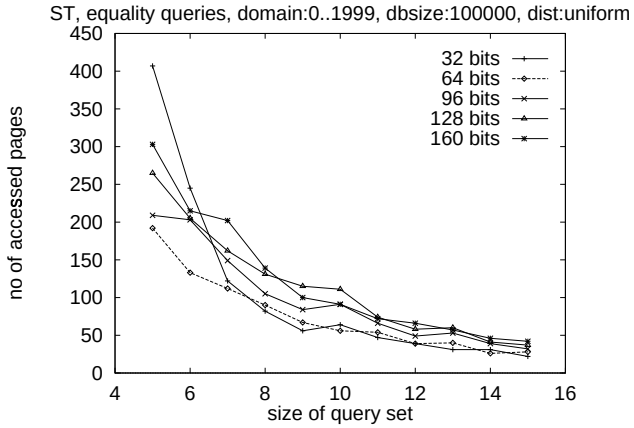
signature size (in bits)	32	64	96	128	160	192	224	256
index size (in 4K pages)	436	580	731	882	1010	1173	1296	1443

Table 7.14: Size of ST index in 4K pages

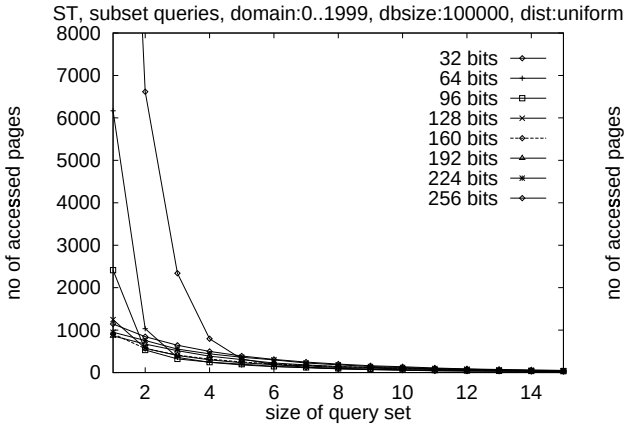
We observe similar effects for an ST index as for an SSF index. A larger signature width means fewer false drops, but the nodes of an index tree hold fewer signatures. We omit figures for the number of false drops as these are identical to those in Figure 7.14. We cannot eliminate false drops with signature trees either, because they are inherent to signatures.

Table 7.14 shows the size of an ST index depending on the signature size. Larger signatures mean that fewer signatures will fit on a page of the tree, which leads to a larger index.

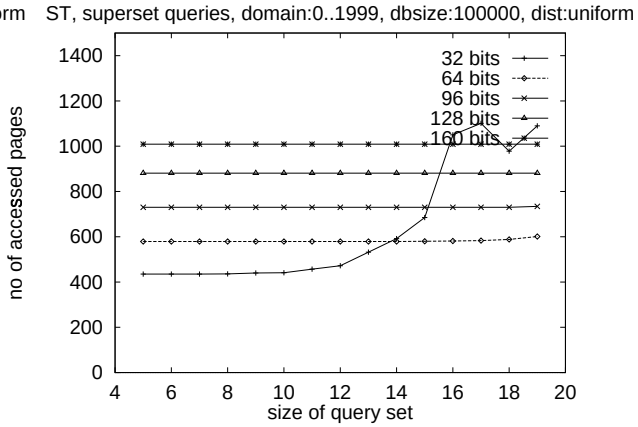
When evaluating the query performance we also have to consider equality queries this time (unlike for SSF), because searching the inner nodes of signature trees involves subset predicates (see Section 7.2.2). The optimal size for the signatures is 64, 160, and 64 bits



(a) Equality queries



(b) Subset queries



(c) Superset queries

Figure 7.16: Number of page accesses for equality, subset, and superset queries (ST)

for equality, subset, and superset queries, respectively. The results of the experiments can be seen in Figure 7.16.

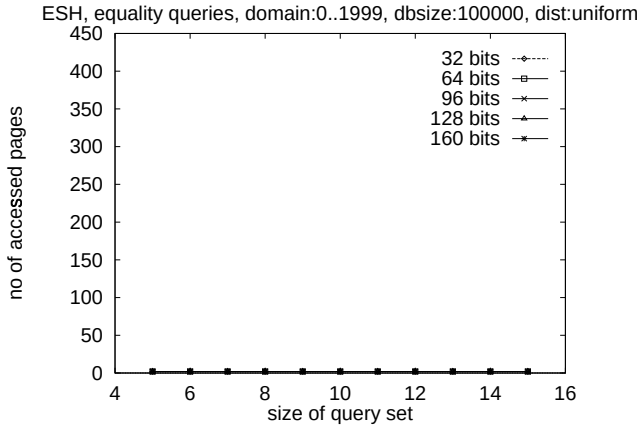
7.6.3 Extendible signature hashing

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

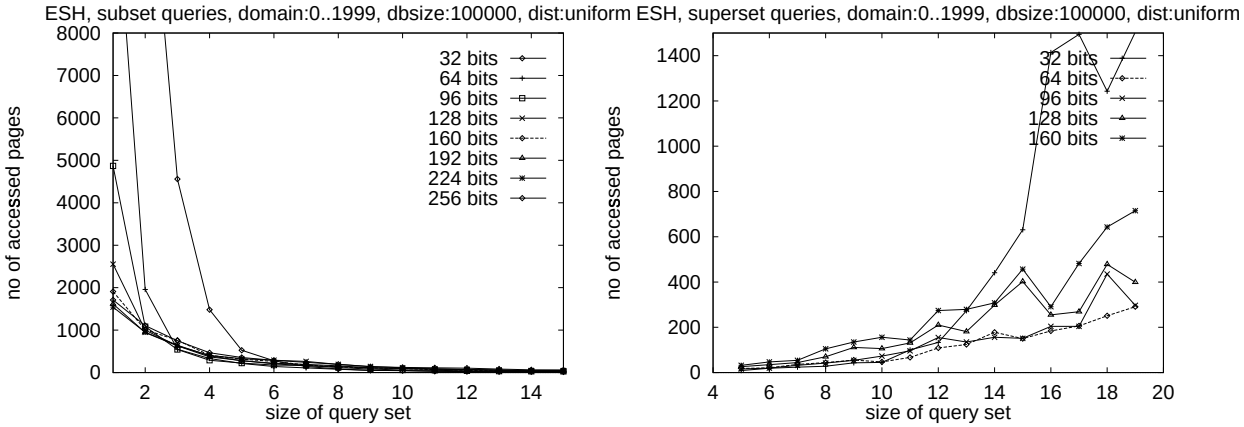
Table 7.15: Size of ESH index in 4K pages

As ESH also cannot filter out false drops, the number of false drops is not different from that of SSF (see Figure 7.14).

In order to restrict an otherwise limitless growth of the directory, we do not allow the directory to grow beyond 2^{20} entries (see Section 7.2.3). We use overflow buckets in this



(a) Equality queries



(b) Subset queries (c) Superset queries

Figure 7.17: Number of page accesses for equality, subset, and superset queries (ESH)

case instead of splitting the directory any further. Large signatures cause more frequent splits of the buckets and therefore lead to a faster expansion of the directory. This means that we reach the maximal size of the directory earlier and we start allocating overflow buckets sooner. As a consequence the internal fragmentation in the buckets decreases, which leads to a flattening of the growth of the index size (see Table 7.15).

Equality queries have a very low number of false drops and only one bucket needs to be accessed during the evaluation of a query. Therefore a long chain of overflow buckets is detrimental to the performance. The length of a chain depends on the size of the signature. Different signature sizes yield very similar results for equality queries (see Figure 7.17(a)). Therefore we choose a signature size of 32 bits, as it results in the smallest index structure without giving up performance. The query performance of subset and superset queries is in accordance with the performance of the other index structures, i.e., signature sizes of 160 and 64 bits, respectively, yield the best results.

7.6.4 Recursive linear signature hashing

As can be seen in Table 7.16, an RLSH index is larger than an ESH index (Table 7.15), because the buckets are split when a utilization of 60% is reached. So we have a higher internal fragmentation for RLSH.

signature size (in bits)	32	64	96	128	160	192	224	256
index size (in 4K pages)	1150	1525	1935	2325	2665	2995	3328	3708

Table 7.16: Size of RLSH index in 4K pages

Figure 7.18 illustrates the experimental results for tuning the query evaluation costs. For equality queries, an index with 32-bit signatures shows the best performance. For larger signatures we have more levels of recursive hash tables, which lead to a higher number of page accesses. The performance for subset and superset queries is very similar to the performance of ESH in these cases, i.e., we will also use 64-bit and 160-bit signatures, respectively. The costs of RLSH, however, are slightly higher than the costs of ESH. This is also due to the lower space utilization of the buckets, as splits occur more often than in ESH, resulting in a slightly larger directory or more recursive hash tables, which makes the evaluation of the queries more expensive.

7.7 Results for uniformly distributed data

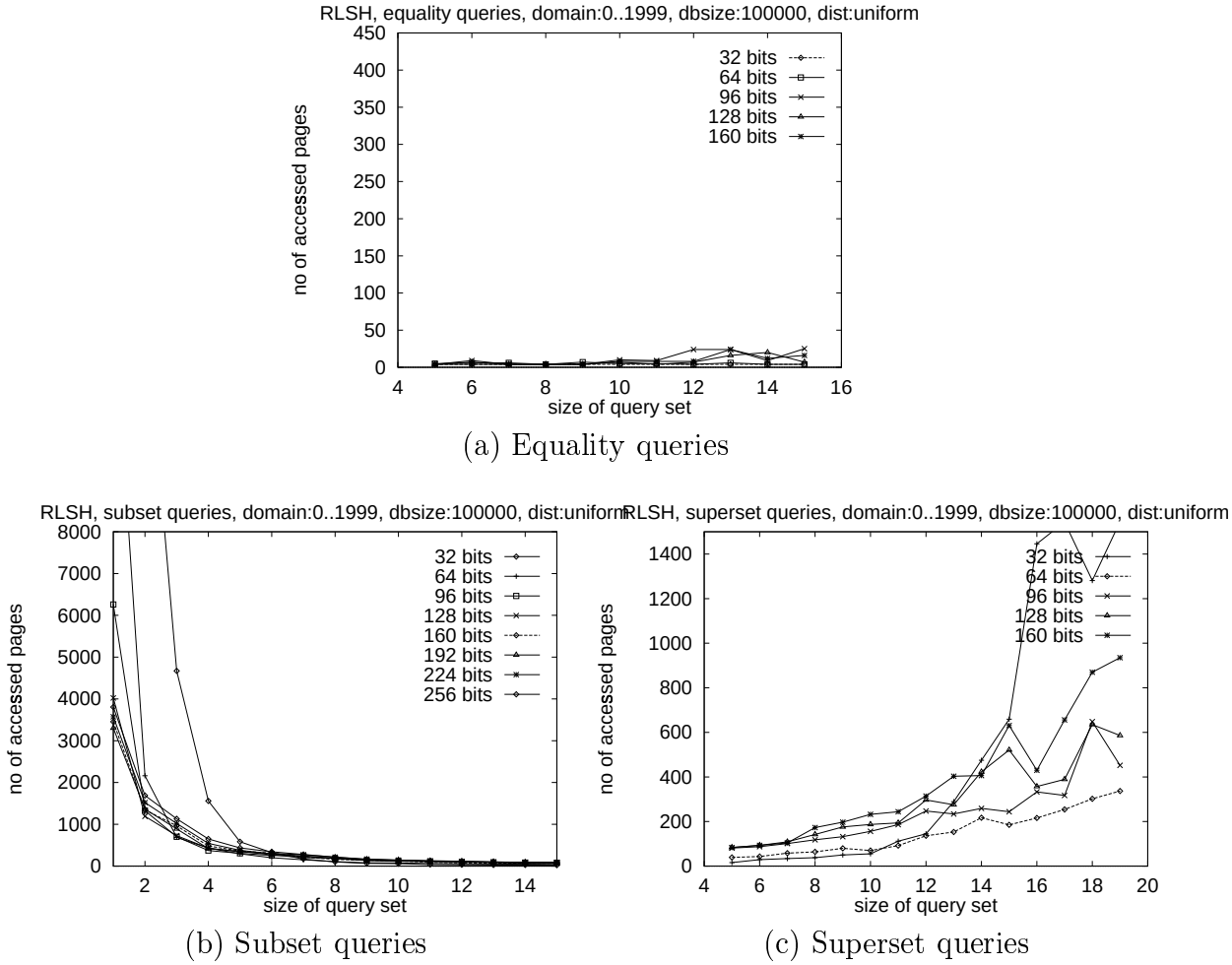
We present the most important results of our extensive experiments emphasizing query evaluation speed and index size. In this section we deal with uniformly distributed data, in Section 7.8 with skewed data. As unit of measurement we use the number of page accesses. As we are interested in the overhead of each index, we only count the page accesses needed to traverse an index structure during query evaluation. That means, we do not consider page accesses for accessing qualifying data items (as these are the same for all index structures). For signature-based index structures we take into account the accesses to false drops by subtracting the page accesses for qualifying items from the total page accesses.

7.7.1 Retrieval Costs

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

Influence of query set cardinality

Results of the experiments varying query set cardinality are found in Figure 7.19. The influence of query set cardinality on the retrieval cost of equality queries (Figure 7.19(a))



(a) Equality queries

(b) Subset queries

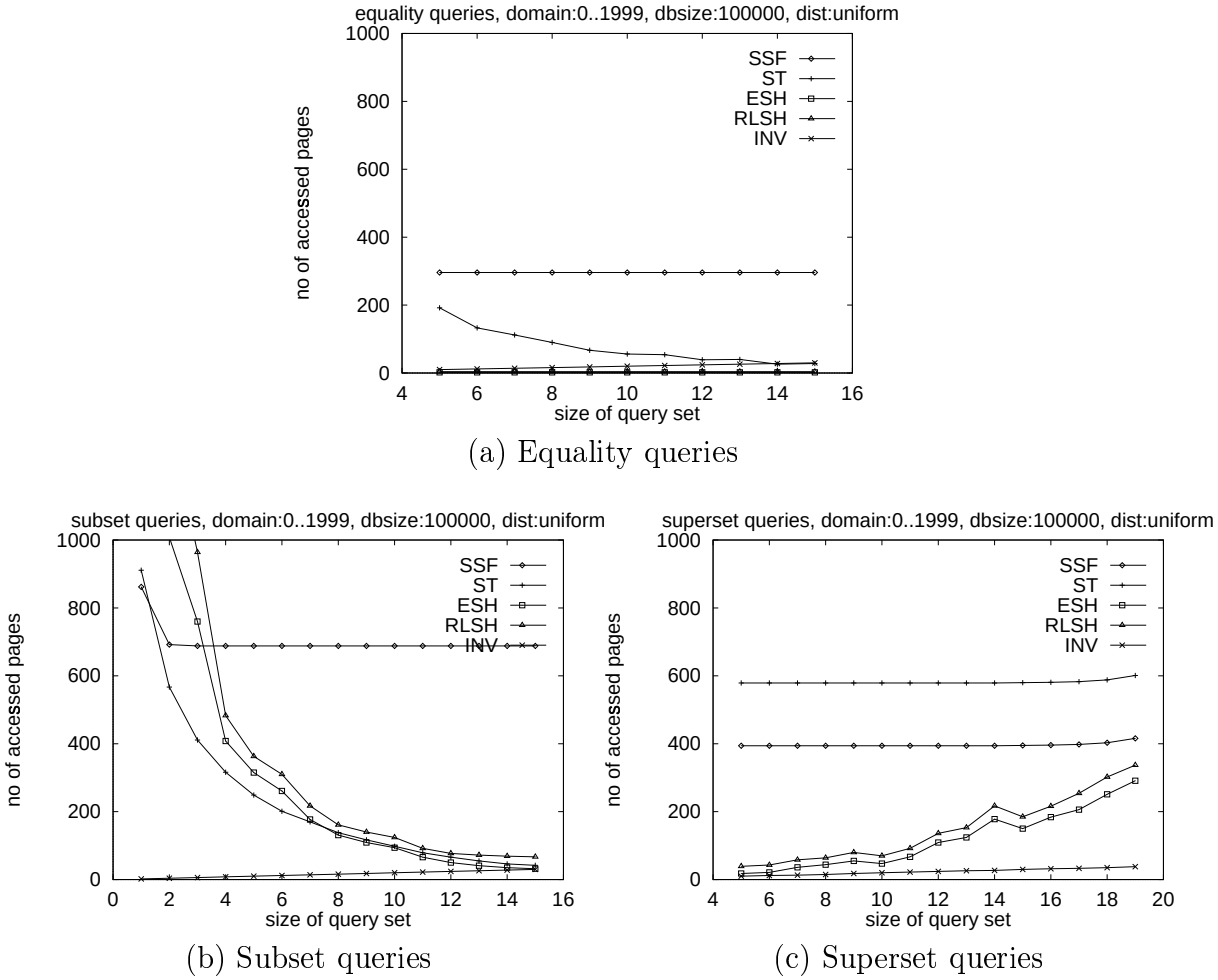
(c) Superset queries

Figure 7.18: Number of page accesses for equality, subset, and superset queries (RLSH)

is marginal. The only major exception is ST, as a subset query has to be performed on the inner nodes of the tree. Subset queries with a small query set cardinality are a difficult case, as we will see. There is also a slight increase in retrieval costs for inverted files for equality queries, as more lists have to be searched. In summary we can say that the hash table indexes are unbeaten for equality queries, as it only takes 2 page accesses to reach a data item.

The results for subset queries are depicted in Figure 7.19(b). Here the signature-based index structures have a severe disadvantage. Usually subset queries are formulated with query sets having a small cardinality. In these cases ST, ESH, and RLSH show their worst behavior, as ST needs to traverse many branches and ESH and RLSH need to visit many buckets. So for subset queries the inverted file index reigns supreme.

For superset queries the important cases are query sets with a large cardinality. As can be seen in Figure 7.19(c) the cardinality of query sets does not have much influence on SSF and the inverted file index. ST is not suited for superset queries at all (it is even worse than SSF) and the hash-based indexes have some problems with large query set cardinalities. Again the inverted file index is superior.



(a) Equality queries

(b) Subset queries

(c) Superset queries

Figure 7.19: Retrieval costs for uniformly distributed data, varying query set cardinality

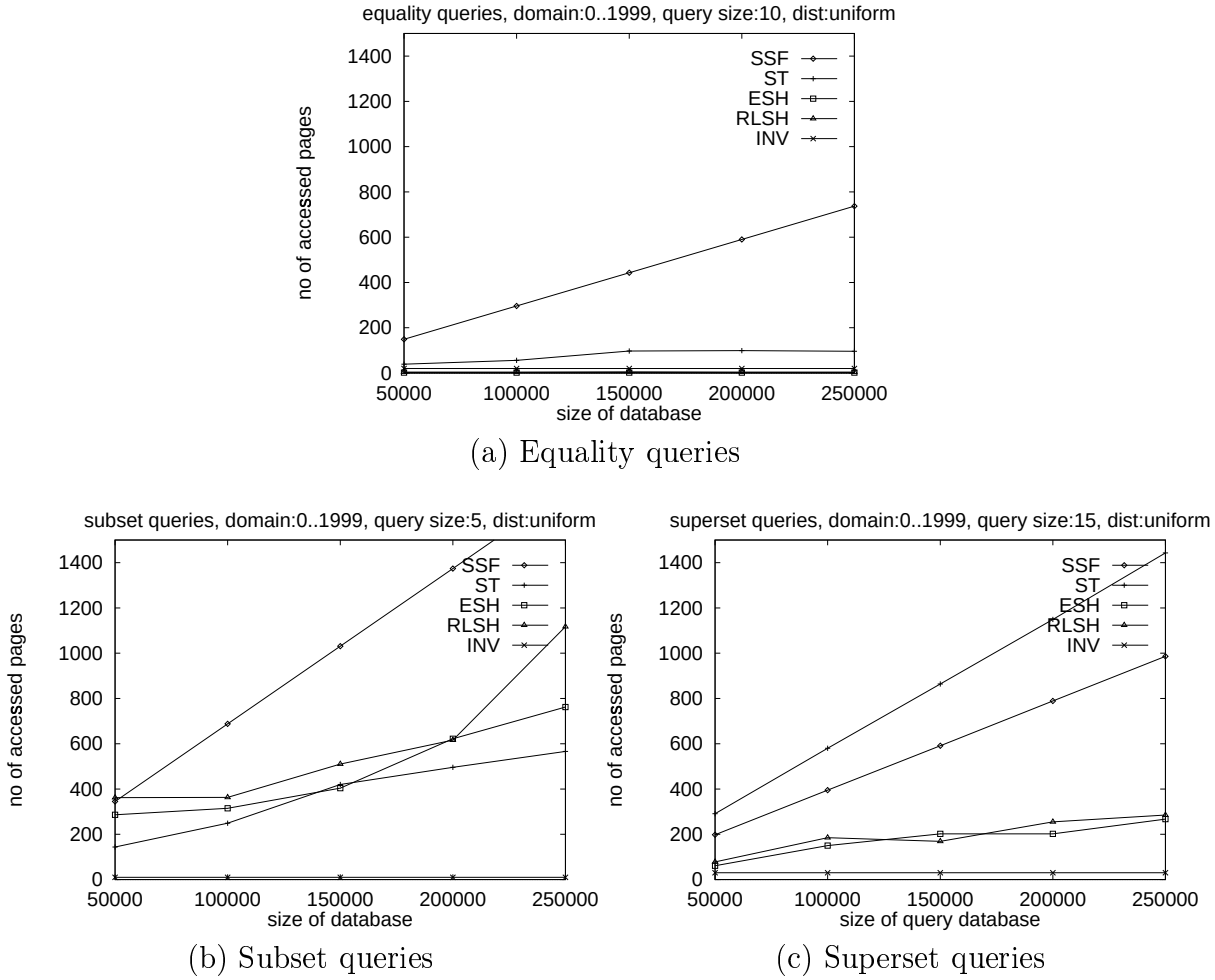
Influence of database size

Another important parameter we investigated is the size of the database (in number of data items). We wanted to know if the index structures scale gracefully. The results of our experiments can be seen in Figure 7.20 ((a) equality queries, (b) subset queries, and (c) superset queries). The retrieval costs for SSF are proportional to the database size, since the whole index has to be scanned. Given the different signature sizes for the signature-based indexes for each query type (see Section 7.6), the costs for equality (32-bit signatures), subset (64-bit signatures), and superset queries (160-bit signatures) differ.

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

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

The retrieval costs for an inverted file index grow very slowly for all query types. Thanks



(a) Equality queries

(b) Subset queries

(c) Superset queries

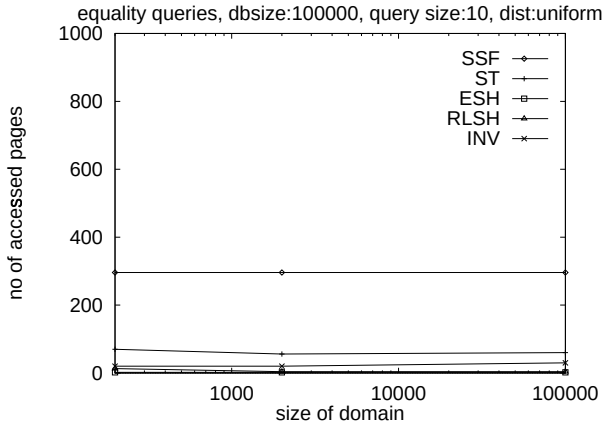
Figure 7.20: Retrieval costs for uniformly distributed data, varying database size

to the optimization techniques described in Section 7.2.5 we can keep the retrieval cost almost constant. Clearly the inverted file index shows the best overall scaling behavior of the examined index structures.

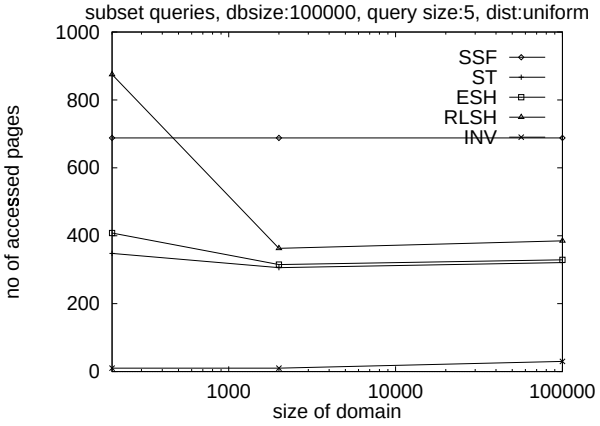
Influence of domain size

In Figure 7.21 we present the influence of domain size on the retrieval costs. Parts (a), (b), and (c) show the results for equality, subset, and superset queries, respectively. Except for RLSH this parameter has only marginal influence on the retrieval costs. Even for an inverted file index, which has to manage larger numbers of lists for increasing domain size, we cannot detect a noticeable increase in retrieval costs.

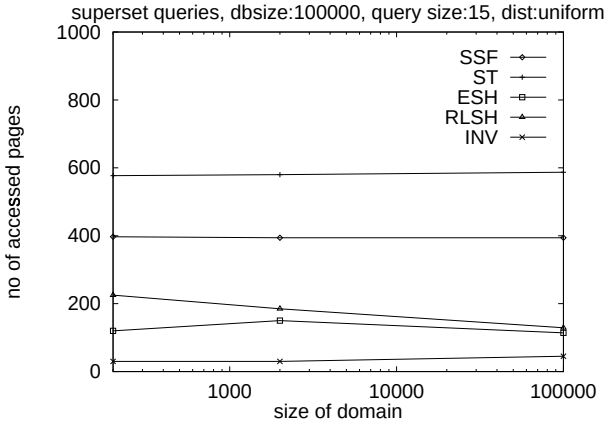
The large number of recursive hash tables for small domain sizes is responsible for the performance of RLSH. Figure 7.22 shows the number of recursive hash tables depending on the size of the domain and on the size of the signatures. For large signatures the number of recursive levels tends to go up (an effect we have also seen in Section 7.6.4 while tuning RLSH). For small domains this effect is amplified.



(a) Equality queries



(b) Subset queries



(c) Superset queries

Figure 7.21: Retrieval costs for uniformly distributed data, varying domain size

7.7.2 Index Size

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

Influence of database size

Table 7.17 shows the experimental results pertaining to the scalability of the index structures. All index structures seem to grow linearly with the database size. As already seen in Section 7.6 the signature size also has a direct influence on the index size. The compression of inverted files makes them competitive in comparison to the signature-based index structures. An earlier uncompressed version of inverted files we used was about 8 to 9 times larger, which would have been unacceptable.

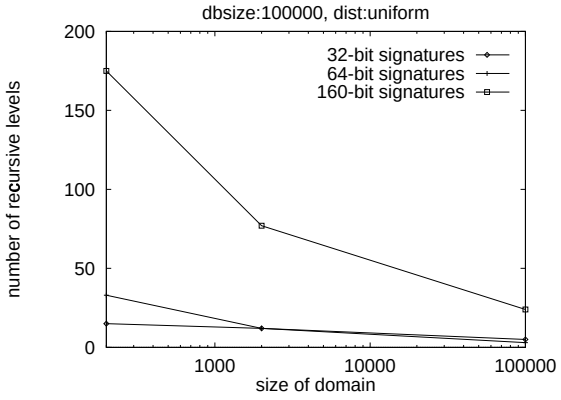


Figure 7.22: Number of recursive hash tables for RLSH

database size	50000	100000	150000	200000	250000
SSF (32 bit)	149	296	443	590	737
SSF (64 bit)	198	394	590	786	982
SSF (160 bit)	345	688	1031	1374	1717
ST (64 bit)	292	580	864	1147	1440
ST (160 bit)	502	1010	1503	2024	2554
ESH (32 bit)	435	872	1271	1747	2166
ESH (64 bit)	573	1160	1639	1956	2390
ESH (160 bit)	1045	1992	2663	3247	3962
RLSH (32 bit)	580	1150	1716	2292	2862
RLSH (64 bit)	769	1525	2307	3053	3773
RLSH (160 bit)	1354	2665	3975	5299	6574
Inverted file	268	530	788	1046	1303

Table 7.17: Index size in 4K pages for uniformly distributed data, varying database size

Influence of Domain Size

Table 7.18 shows the influence of domain size on the index size. As expected for the signature-based index structures, this influence is marginal. For inverted files we can say that the smaller the domain, the better the space requirement. The obvious reason for this is that for each value appearing in a set, a list has to be allocated. The total number of lists could be reduced by merging lists of values that appear infrequently into one list. This would lead to a better compression of the small lists. However, the retrieval costs would also rise as false drops would be introduced. We would then have to eliminate these false drops.

domain size	200	2000	100000
SSF (32 bit)	296	296	296
SSF (64 bit)	394	394	394
SSF (160 bit)	688	688	688
ST (64 bit)	575	580	587
ST (160 bit)	1012	1010	1018
ESH (32 bit)	874	872	886
ESH (64 bit)	1093	1160	1199
ESH (160 bit)	1639	1992	1992
RLSH (32 bit)	1171	1150	1154
RLSH (64 bit)	1536	1525	1532
RLSH (160 bit)	2770	2665	2662
Inverted file	369	530	1559

Table 7.18: Index size in 4K pages for uniformly distributed data, varying domain size

7.8 Results for skewed data

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

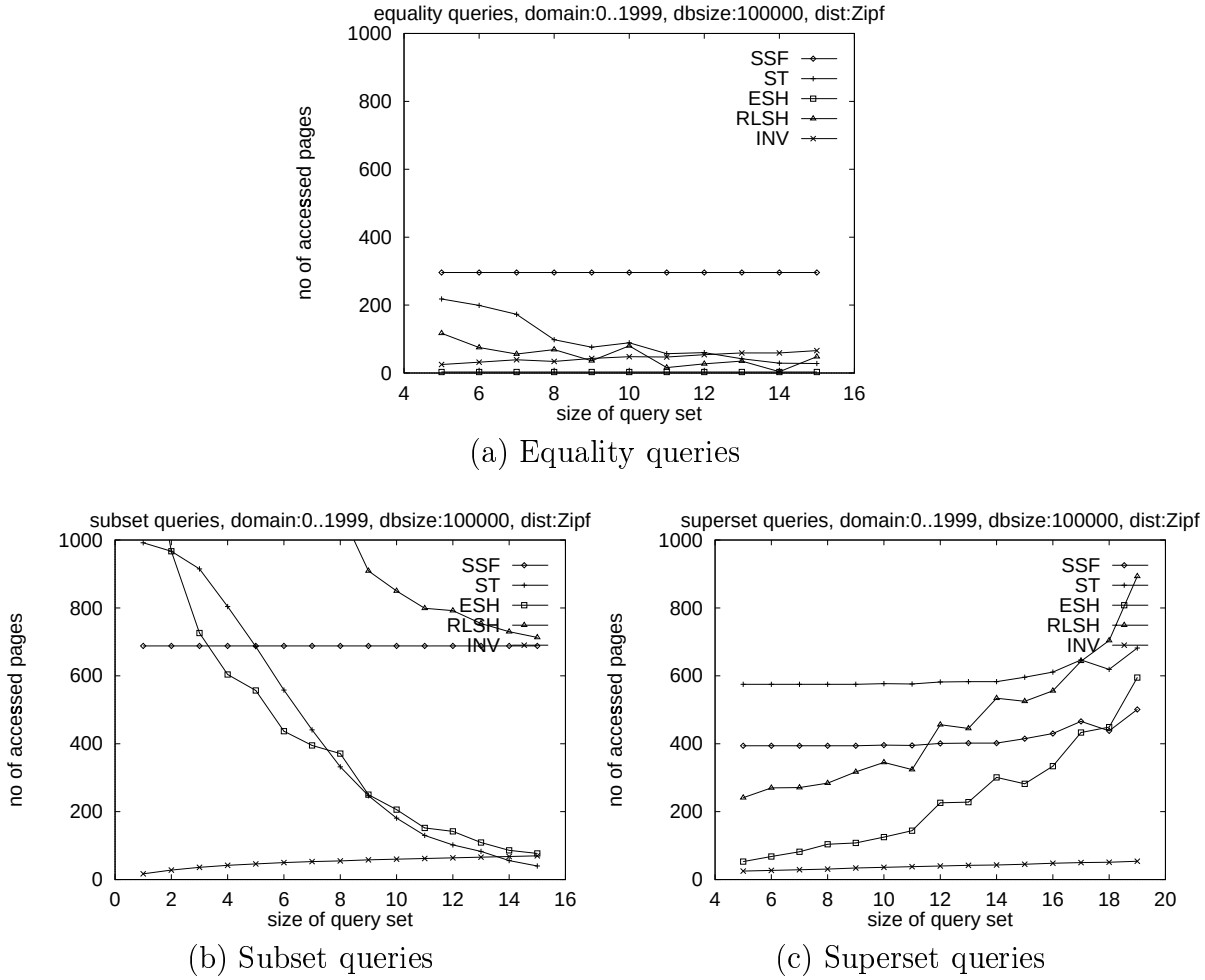
7.8.1 Retrieval Costs

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

Influence of query set cardinality

Varying the cardinality of the query sets yields the result depicted in Figure 7.23. For equality queries we obtained results very similar to those for uniformly distributed data. SSF and ESH are not influenced at all, while ST and RLSH show a slight increase in costs. We also noticed a small increase in costs for the inverted file index. This stems from the fact that the lists for the most common values are longer than the lists in the case for uniformly distributed data. Since these values also appear more often in queries, they are searched for more often during query evaluation. As for uniformly distributed data, ESH is still the fastest index for equality queries.

For subset queries (Figure 7.23 (b)) the performance of all index structures except SSF deteriorates. In the ST index more branches have to be traversed during query evaluation, while in the ESH index more buckets are accessed. The performance of RLSH is totally unacceptable. As we have already seen RLSH has difficulties with small domains (see Figure 7.21). Skewed data is comparable to small domains insofar as a small number of different elements appear in the sets. The decrease in performance for inverted files has



(a) Equality queries

(b) Subset queries

(c) Superset queries

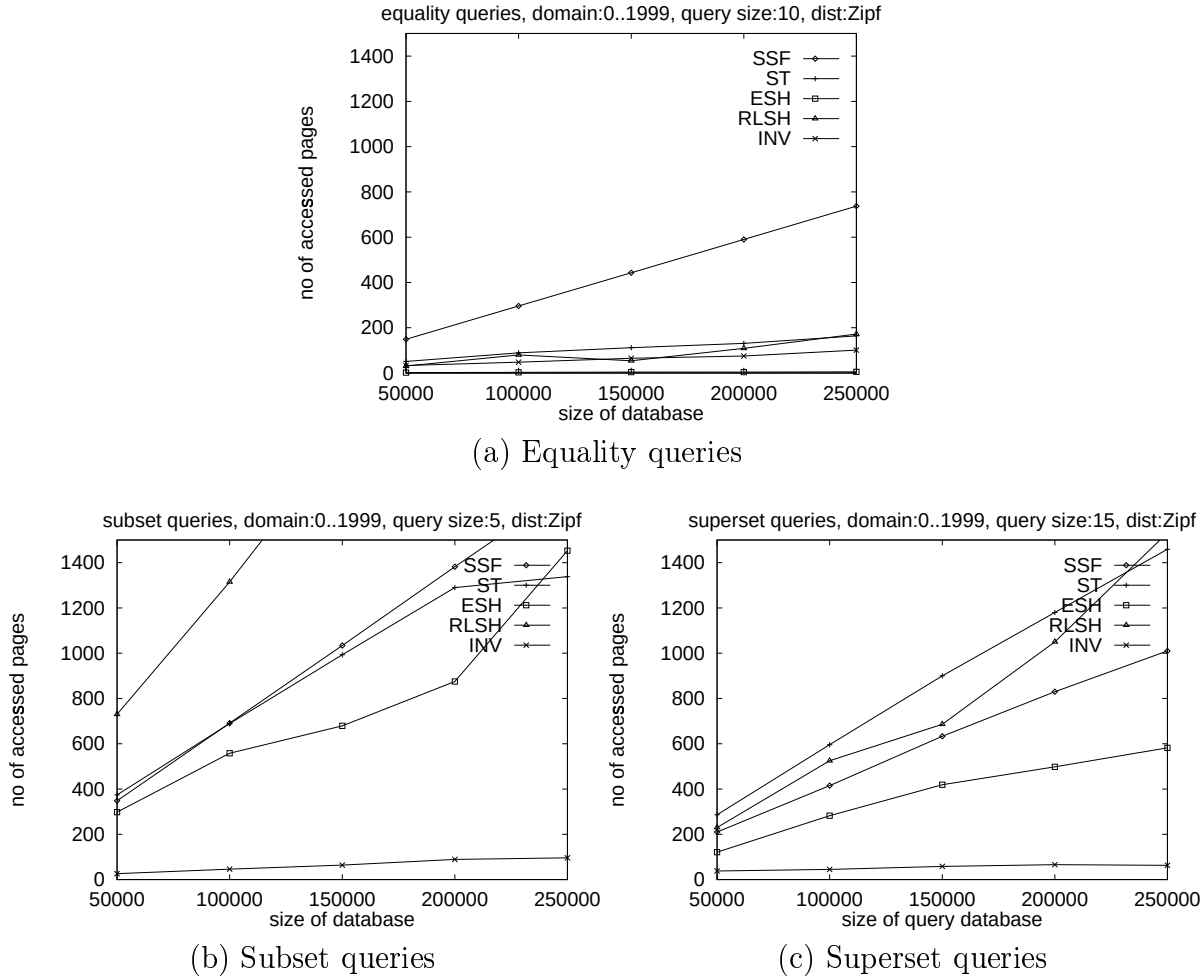
Figure 7.23: Retrieval costs for skewed data, varying query set cardinality

the same reasons already mentioned for equality queries. In spite of the deterioration the inverted file index is still the best index for subset queries.

The insights for SSF, ESH, and inverted files for subset queries also apply to superset queries. ST shows no further deterioration for superset queries for skewed data, because it already displays worst case behavior for superset queries for uniformly distributed data. The performance of RLSH is not as disastrous as for subset queries, but RLSH has trouble even keeping up with SSF. So for superset queries the inverted files are once more the index of choice.

Influence of database size

The influence of database size on the index structures is shown in Figure 7.24. We observed a linear growth of retrieval costs for SSF for all query types, i.e. skewed data has no influence whatsoever on the internal structure of SSF. In both cases (uniformly distributed and skewed data) SSF consists of sequential files with the same length, which are scanned sequentially during query evaluation. The performance of the other index



(a) Equality queries

(b) Subset queries

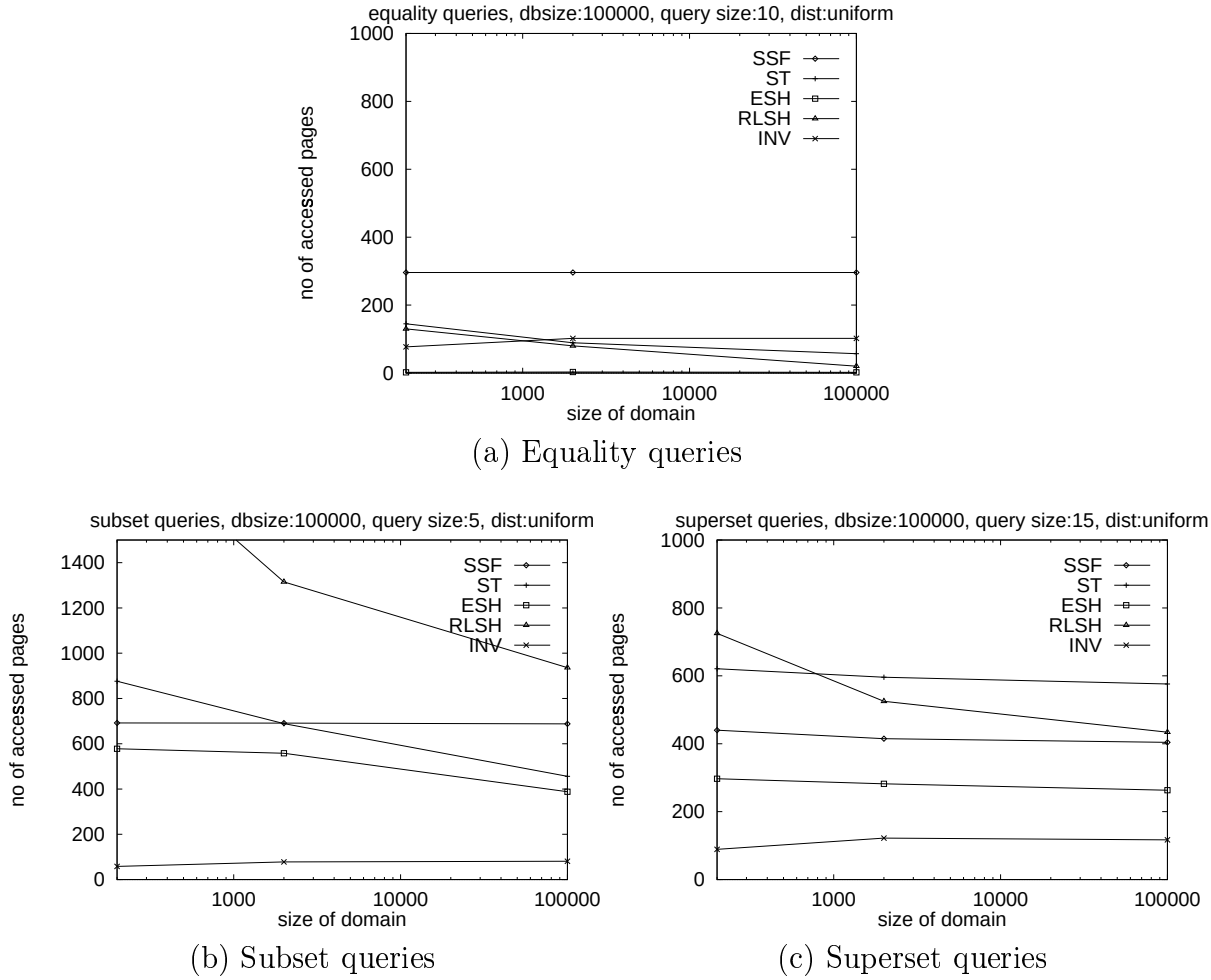
(c) Superset queries

Figure 7.24: Retrieval costs for skewed data, varying database size

structures for skewed data worsens compared to uniformly distributed data. Notable exceptions to this rule are ESH for equality queries (best case for hash-based indexes) and ST for superset queries (worst case for ST). ESH can still keep up the performance as the index structure for equality queries is relatively small because of the signature sizes (32-bit signatures). In this case, overflow buckets are rarely needed. The performance of ST does not deteriorate any further because it cannot get worse. Regardless of the data distribution ST needs to scan almost every single node in the tree. As expected, the performance of RLSH does not improve for larger databases. It does not seem to be suited for skewed data at all. Inverted files, although losing some performance compared to uniformly distributed data, still show the best overall behavior.

Influence of domain size

We summarized the results for the influence of the domain size in Figure 7.25. The domain size has no influence on SSF. For the other signature-based index structures and inverted files the performance deteriorates because small domains amplify the skewing of data. The



(a) Equality queries

(b) Subset queries

(c) Superset queries

Figure 7.25: Retrieval costs for skewed data, varying domain size

reason for this is that the variety of occurring values is lowered and the distribution of these values is more unbalanced (the probability of the most common value for $|D| = 200$ is 0.17, for $|D| = 2000$ it is 0.12, and for $|D| = 100000$ 0.08). RLSH, which already had difficulties coping with small domains for uniformly distributed data, suffers severe performance losses for skewed data. We can trace back these performance losses to the structure of RLSH, namely the number of recursive hash tables. The left hand side of Figure 7.26 shows the number of recursive hash tables for different domain sizes for skewed data. The right hand side of Figure 7.26 reiterates the numbers for uniformly distributed data from Figure 7.22 on a different scale for better comparison. The most important conclusion of this section, however, is that the performance of inverted files does not worsen significantly for large domains.

7.8.2 Index Size

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

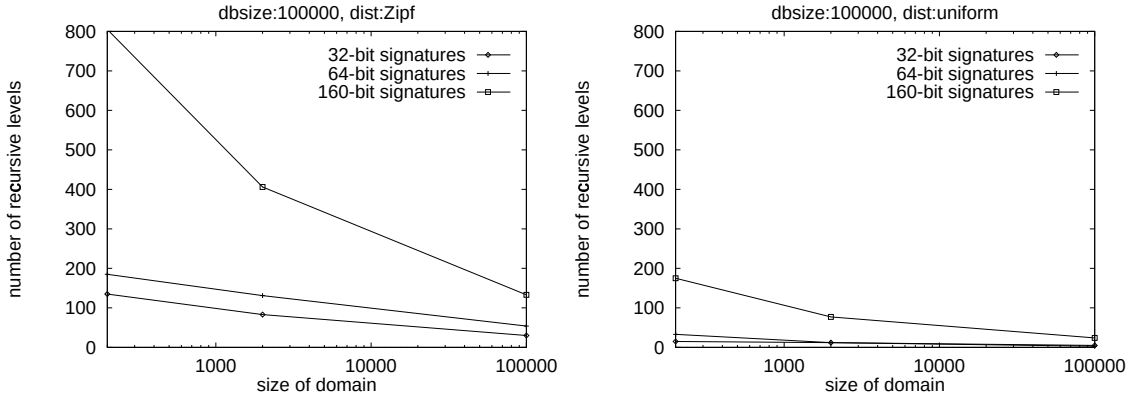


Figure 7.26: Number of recursive hash tables for RLSH

database size	50000	100000	150000	200000	250000
SSF (32 bit)	149	296	443	590	737
SSF (64 bit)	198	394	590	786	982
SSF (160 bit)	345	688	1031	1374	1717
ST (64 bit)	274	576	858	1137	1432
ST (160 bit)	492	993	1473	1964	2457
ESH (32 bit)	489	790	1086	1377	1654
ESH (64 bit)	473	859	1263	1642	2041
ESH (160 bit)	809	1490	2195	2884	3564
RLSH (32 bit)	577	1162	1767	2291	2832
RLSH (64 bit)	785	1589	2346	3187	3931
RLSH (160 bit)	1455	2895	4363	5778	7285
Inverted file	178	340	505	668	830

Table 7.19: Index size in 4K pages for skewed data, varying database size

Influence of database size

In Table 7.19 we depict the influence of skewed data on the size of the index structures when varying the database size. The values for SSF for skewed data correspond to those for uniformly distributed data (Figure 7.17). When comparing the other index structures, we notice that in some cases the index size is smaller for skewed data. In case of ST, skewed data leads to a higher fill degree of the nodes, which makes the index smaller. For ESH skewed data causes an unbalanced directory and many splits. Since the size of the directory is limited, we have to introduce overflow pages. Alongside many disadvantages, however, overflow pages have one advantage: the pages are better utilized. This is also the case for the inverted files, which do not need to fear comparison with the other index structures. The compression rate of the lists is better than for uniformly distributed data, because we have fewer, larger lists. An exception to this rule is RLSH, which increases

in size. The reason for this is a dramatic increase in the number of recursive hash tables along with the storage overhead for managing these tables (similar to the effect observed for small domains).

domain size	200	2000	100000
SSF (32 bit)	296	296	296
SSF (64 bit)	394	394	394
SSF (160 bit)	688	688	688
ST (64 bit)	576	576	567
ST (160 bit)	982	993	988
ESH (32 bit)	762	790	807
ESH (64 bit)	848	859	891
ESH (160 bit)	1468	1490	1539
RLSH (32 bit)	1140	1162	1150
RLSH (64 bit)	1720	1589	1598
RLSH (160 bit)	3295	2895	2828
Inverted file	266	340	961

Table 7.20: Index size in 4K pages for skewed data, varying domain size

Influence of domain size

Table 7.20 shows the influence of domain size on the index size. As usual, SSF is immune to changes in domain size. For most of the other index structures we can see that skewed data is stored more compactly than uniformly distributed data. An exception to the rule is, again, RLSH. Skewed data leads to a dramatic increase in the number of recursive hash tables. For the inverted file index we have found that the larger the domain, the larger the inverted file will be, but the inverted file index is still competitive when compared to the other index structures.

As we can see a smaller index size does not mean that the index structures are better suited for efficient retrieval. An inferior internal organization of the data is responsible for the deterioration of the retrieval costs, not a larger index size.

7.9 Conclusions

We studied the performance of several different index structures for set-valued attributes of low cardinality following two approaches to evaluate our work. The first approach was developing cost models for all index structures to compare them mathematically. Although we were able to determine the best/worst case behavior, the average case is very difficult to analyze. This persuaded us to evaluate the index structures experimentally in our second approach. For that reason we implemented sequential signature files, signature trees, extendible signature hashing and B^+-tree-based inverted files. We refitted the index

structures to support the evaluation of queries containing set-valued predicates (namely equality, subset, and superset predicates).

During the evaluation we observed the following. The inverted file index dominated the field clearly, though for equality queries the hash-based index structures (except for RLSH in the case of skewed data) were faster. Generally the signature-based index structures have difficulties with skewed data and some important query cases (small query sets for subset queries, large query sets for superset queries). The inverted file index showed robustness for skewed data and we were able to keep the space demands reasonable. Zobel, Moffat, and Ramamohanarao made a similar observation for the special case of text retrieval while comparing inverted files to signature files [101].

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

If there are significant breakthroughs in the area of high dimensional spatial index structures, set-valued queries will also be able to profit from them. Each set can be seen as a point in a $|D|$-dimensional space, where $|D|$ is the cardinality of the set domain. For example, assume we have the domain $D = \{1, 2, 3, \ldots, 10\}$ and a set $s = \{3, 4, 8\}$. Then s can be mapped to $(0, 0, 1, 1, 0, 0, 0, 1, 0, 0)$ in a 10-dimensional space. Set equality queries are simple point queries in this space, while we can transform subset and superset queries to range queries. We abandoned this straightforward approach because of the bad performance of high dimensional spatial access methods.

A common approach to handling high dimensional data is to transform the data to lower dimensions. So instead of using the sets themselves we could work with their signatures. Although at first glance it looks like this will make our job easier, as we now handle b-dimensional data (assuming a signature size of b) instead of $|D|$-dimensional data, this still overtaxes spatial index structures.

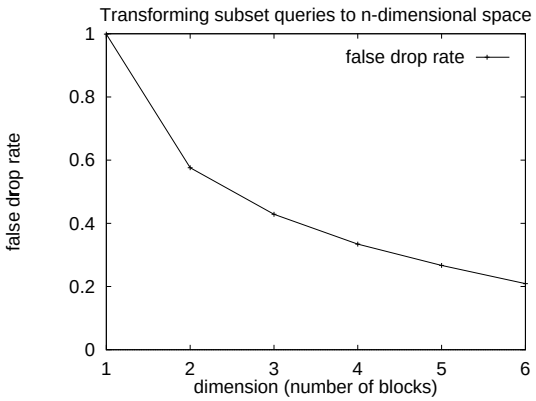


Figure 7.27: False drop rates for different block sizes

A way to scale down the dimensions arbitrarily is the following. We divide a bitvector representing a set into x blocks of size $\frac{b}{x}$ and count the number of bits set in each block. For example, assume a bitvector of size $b = 16$ and $x = 4$: 1001 1101 0001 0000. This bitvector

is mapped on the point $(2, 3, 1, 0)$ in 4-dimensional space. Again we can transform equality queries to point queries and subset/superset queries to range queries, but at the price of additional false drops. These false drops are due to the fact that the number of bits set in a block does not tell us which bits are set. That means we have introduced another filter, for which we can control the false drop rate by choosing the block size. Unfortunately, for large block sizes (low dimensional space), which can still be handled satisfactorily by spatial index structures, the false drop rate is too high to achieve favorable results (see Figure 7.27 for typical values in subset queries).

Chapter 8

Conclusion and Outlook

Developing new algorithms and access structures for the physical level of DBMS is always going to be an interesting subject, as we have to support the ever growing functionality of database applications. In our work we focused on supporting set-valued attributes and Data Warehouse applications. From a technical viewpoint join algorithms and index structures are two important research areas on the physical level. Join algorithms rank among the most expensive operations in DBMS, so they show a lot of potential for optimization. Accessing data efficiently via index structures can also contribute significantly to the performance of a database system.

Let us first turn to the join algorithms. On one hand we developed join algorithms for a new environment, namely joining relations on set-valued attributes. For these set-valued join algorithms we demonstrated that efficient alternatives to a naive nested-loop approach exist. We achieved the best results for a hash-based variant relying on superimposed coding for fast set comparison. On the other hand we devised an algorithm that works in a more traditional relational system, but uses knowledge about the data cleverly. We observed that often, related data is inserted at roughly the same time in a Data Warehouse. As we rarely delete or update tuples in a Data Warehouse, the original structure of older parts of a relation is not changed when inserting new data. Therefore there is a correlation between the location of related data in different relations. We developed an algorithm that exploits this fact while joining two relations containing related data. We have shown that this algorithm, called Diag-Join, is up to two and a half times faster than regular join algorithms used in this environment.

In the second part of our work we looked into the performance and robustness of index structures for set-valued attributes of small cardinalities. Indexing sets is a difficult problem, as we cannot impose a total order on the indexed data items. Furthermore, subset and superset queries cannot be transformed to point queries. We chose and modified five different index structures for set-valued retrieval: sequential signature files, signature trees, extendible signature hashing, recursive linear signature hashing and inverted files. The comparison of the access methods was done in two different ways. On one hand we developed (or supplemented existing) cost models and analyzed the behavior of the index structures based on these cost models. As it is very difficult to evaluate the performance for the average case and a non-uniform data distribution, we also decided to implement the index structures and subject them to extensive experiments. Our theoretical analy-

sis and experimental evaluation indicated a clear winner regarding robustness as well as performance: the inverted file index.

This thesis can be seen as a foundation for further work on processing set-valued attributes. As a matter of fact there is already work on secondary storage join algorithms based on our work. Ramasamy, Patel, Naughton, and Kaushik propose a partitioned set join algorithm in [77] while Jost investigates the applicability of Signature Trees and R-trees for the processing of set-valued joins [53]. There is still room left for additional optimization of set-valued join algorithms, so further publications in this area are likely. It is also not clear, how the existing index structures will behave for sets with large cardinalities. Despite all optimization, we expect the performance of inverted files to worsen, as the lists will grow. Signature-based index structures are not necessarily an answer, because for large sets we will have to increase the size of the signatures to keep the false drop probability low, which leads to performance loss. Comparing inverted files to signatures-based index structures for large sets calls for further experiments, the outcome of which will probably depend heavily on the type of indexed data. It is likely that in very large sets many dependencies between elements can be found. We could use this information to build highly specialized index structures, but devising an efficient, all-round access method will be a difficult task.

Moreover, it is not yet quite clear what problems arise when integrating our algorithms and index structures into real database systems. A query optimizer in a DBMS has to be able to choose correctly between different access methods and join algorithms to get the best query performance. In view of query optimization it is also important that parameterizing the algorithms is not too difficult a task. This may not be given for the signature-based techniques as superimposed coding is very sensitive to the processed data and needs to be finely tuned to perform optimally. A lack of tools for analyzing external memory algorithms (similar to O-notation for internal memory) does not help in this regard either, since accurate cost models help an optimizer tremendously.

Appendix A

Minimal Costs for Partial Signatures

In Section 4.3.4 we wanted to know for which value of d the costs $C(d)$ for retrieving all matching tuples from a hash table are minimal. As a reminder

$$C(d) = 2^{\frac{d}{2}} + \frac{|R|}{2^{\frac{d}{2}}} \tag{A.1}$$

The derivative $C'(d)$ $\left(\frac{dC}{dd}\right)$ is equal to

$$\begin{aligned} C'(d) &= \frac{\ln 2}{2} \cdot 2^{\frac{d}{2}} - \frac{\ln 2}{2} \cdot \frac{|R|}{2^{\frac{d}{2}}} \\ &= \frac{\ln 2}{2}\left(2^{\frac{d}{2}} - \frac{|R|}{2^{\frac{d}{2}}}\right) \end{aligned} \tag{A.2}$$

We now have to determine for which d $C'(d)$ is equal to 0:

$$\begin{aligned} C'(d) = 0 &\Leftrightarrow \frac{\ln 2}{2} \cdot 2^{\frac{d}{2}} = \frac{\ln 2}{2} \cdot \frac{|R|}{2^{\frac{d}{2}}} \\ &\Leftrightarrow 2^d = |R| \\ &\Leftrightarrow d = \log_2 |R| \end{aligned} \tag{A.3}$$

The derivative $C''(d)$ of $C'(d)$ is equal to

$$\begin{aligned} C''(d) &= \left(\frac{\ln 2}{2}\right)^2 \cdot 2^{\frac{d}{2}} + \left(\frac{\ln 2}{2}\right)^2 \cdot \frac{|R|}{2^{\frac{d}{2}}} \\ &= \left(\frac{\ln 2}{2}\right)^2 \left(2^{\frac{d}{2}} + \frac{|R|}{2^{\frac{d}{2}}}\right) \end{aligned} \tag{A.4}$$

Inserting $d = \log_2 |R|$ we get

$$C''(\log_2 |R|) \quad = \quad \left(\frac{\ln 2}{2}\right)^2 \cdot \left(\sqrt{|R|} + \frac{|R|}{\sqrt{|R|}}\right) \tag{A.5}$$

which for $|R| > 0$ is obviously larger than 0. Thus $C(d)$ becomes minimal for $d = \log_2 |R|$.

Appendix B

Approximations for the Cost Model of RLSH

In Section 7.3.5 we needed to estimate the the number of data items per bucket on the left side and on the right side of the split pointer. We approximate the probability that a bucket left of the split pointer is the home bucket of an arbitrary data item inserted at level l by

$$\lambda_l = \frac{p_l}{2^{d_l}} \tag{B.1}$$

The probability that the home bucket is located on the right hand side of the split pointer can be approximated by

$$\mu_l = 1 - \frac{p_l}{2^{d_l}} \tag{B.2}$$

The true probabilities for an individual bucket are dependent on the position of the split pointer at previous levels. Nevertheless, the values λ_l and μ_l serve as adequate approximations.

The maximum number of records that can be stored in a bucket is $\frac{P}{S_{record}}$, so the estimated number of records per bucket on the left hand side of the split pointer is

$$left_l = \sum_{j=0}^{n_l} v(j) \binom{n_l}{j} \lambda_l^j (1-\lambda_l)^{n_l - j} \tag{B.3}$$

where

$$v(j) = \begin{cases} j & \text{if } j \leq \frac{P}{S_{record}} \\ \\ \frac{P}{S_{record}} & \text{if } j > \frac{P}{S_{record}} \end{cases} \tag{B.4}$$

We obtain the estimated number of records on the right hand side of the split pointer similarly.

$$right_l = \sum_{j=0}^{n_l} v(j) \binom{n_l}{j} \mu_l^j (1-\mu_l)^{n_l - j} \tag{B.5}$$

Bibliography

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

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

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

[4] 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.

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

[6] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proc. 8th Int. Conf. on Data Engineering*, pages 301–308, Tempe, Arizona, February 1992.

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

[8] M. W. Blasgen and K. P. Eswaran. On the evaluation of queries in a relational database system. Technical Report IBM Research Report RJ1745, IBM, 1976.

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

[10] P. Bosc and H. Prade. An introduction to fuzzy set and possibility theory-based approaches to the treatment of uncertainty and imprecision in database management systems. In *Second UMIS Workshop (Uncertainty Management in Information Systems: from Needs to Solutions)*, Catalina, 1994.

[11] B. Boss and S. Helmer. Index structures for efficiently accessing fuzzy data including cost models and measurements. *Fuzzy Sets and Systems*, 108(1):11–37, November 1999.

[12] K. Bratbersengen. Hashing methods and relational algebra operations. In *Proc. of the 10th VLDB Conference*, pages 323–333, Singapore, August 1984.

[13] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 237–246, 1993.

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

[15] Bernard Chazelle. Filtering search: A new approach to query-answering. In *24th Annual Symposium on Foundations of Computer Science*, pages 122–132, Tucson, Arizona, 7–9 November 1983.

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

[17] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[18] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[19] V. Cross. Fuzzy information retrieval. *Journal of Intelligent Systems*, (3):29–56, 1994.

[20] 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.

[21] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 151–164, Stockholm, Sweden, 1985.

[22] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–8, 1984.

[23] D. DeWitt, D. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *PDIS*, 1993.

[24] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 443, Barcelona, Spain, 1991.

[25] D. DeWitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Fl, 1991.

[26] 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.

[27] 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.

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

[29] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB (Informal Proceedings)*, pages 41–46, 2000.

[30] R.A. Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[31] C.D. French. One size fits all. In *Proc. of the 1995 ACM SIGMOD*, pages 449–450, San Jose, 1995.

[32] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 209–219, 1986.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.

[34] S. P. Ghosh and M. E. Senko. File organization: On the selection of random access index points for sequential files. *Journal of the ACM*, 16(4):569–579, October 1969.

[35] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. IEEE Conference on Data Engineering*, pages 406–417, Houston, TX, 1994.

[36] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. on Data and Knowledge Eng.*, 6(6):934–944, Dec. 1994.

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

[38] O. Günther. Efficient computation of spatial joins. In *Proc. IEEE Conference on Data Engineering*, pages 50–59, Vienna, Austria, Apr. 1993.

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

[40] L.M. Haas, M.J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3):241–256, 1997.

[41] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. on Database Systems*, 3(3):285–298, 1978.

[42] E.P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *VLDB Journal*, 5(1):64–84, 1996.

[43] J.M. Hellerstein, E. Koutsoupias, and C.H. Papadimitriou. Towards a theory of indexability. In *16th PODS*, Tucson, Arizona, 1997.

[44] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of the 21st VLDB Conference*, pages 562–573, Zürich, 1995.

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

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

[47] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with subset join predicates. In *Proc. of the 23rd VLDB Conference*, pages 386–395, Athens, August 1997.

[48] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationship. In *Proc. of the 24th VLDB Conference*, pages 98–109, New York, August 1998.

[49] E. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 606–618, Zurich, 1995.

[50] W.H. Inmon. *Building the Data Warehouse (2nd ed.)*. John Wiley & Sons, New York, 1996.

[51] 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.

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

[53] A. Jost. Implementierung und Vergleich von Verfahren zur Berechnung des Teilmengen-Joins. Master's thesis, Philipps-Universität, Marburg, April 1999.

[54] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report CS-TR-96-20, University of Massachusetts, 1996.

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

[56] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 180–191, Santiago, Chile, Sept. 1994.

[57] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Massachusetts, 1989. Addison Wesley.

[58] R. Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, New York, 1996.

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

[60] 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.

[61] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–266, 1989.

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

[63] P. A. Larson. Linear hashing with partial expansions. In *Proc. of the 6th VLDB Conference*, pages 224–232, Montreal, 1980.

[64] D. Lee and C. Leng. Partitioned signature files: Design issues and performance evaluation. *ACM Trans. on Information Systems*, 7(2):158–180, Apr. 1989.

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

[66] M.-L. Lo and C. Ravishankar. Spatial hash-joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 247–258, Montreal, Canada, Jun 1996.

[67] R. Lorie and H. Young. A low communication sort algorithm for a parallel database machine. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 135–144, 1989. also published as: IBM TR RJ 6669, Feb. 1989.

[68] 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.

[69] J. Menon. A study of sort algorithms for multiprocessor DB machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 197–206, Kyoto, 1986.

[70] P. Mishra and H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[71] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 468–478, 1988.

[72] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, Sep 1995.

[73] G. Özsoyoğlu, Z.M. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[74] J. Patel and D. DeWitt. Partition based spatial-merge join. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 259–270, Montreal, Canada, Jun 1996.

[75] V. Poosala. Zipf's law. Technical report, University of Wisconsin Madison, 1995.

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

[77] K. Ramasamy, J.M. Patel, J.F. Naughton, and R.Kaushik. Set containment joins: The good, the bad, and the ugly. In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, August 2000.

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

[79] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

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

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

[82] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. FastSort: an distributed single-input single-output external sort. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 94–101, 1990.

[83] H.-J. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[84] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 469–480, Brisbane, 1990.

[85] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[86] E.J. Shekita and M.J. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, Atlantic City, New Jersey, May 1990.

[87] D. Shin and A. Meltzer. A new join algorithm. *SIGMOD Record*, 23(4):13–18, Dec. 1994.

[88] M. Stonebraker. *Object-Relational DBMSs, The Next Great Wave.* Morgan Kaufmann Publishers, San Francisco, California, 1996.

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

[90] Transaction Processing Council (TPC). TPC Benchmark D. http://www.tpc.org, 1995.

[91] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), 1987.

[92] 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.

[93] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3), September 2000.

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

[95] 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.

[96] Extensible Markup Language (XML) 1.0, W3C Recommendation, February 1998. available at `http://www.w3.org/TR/1998/REC-xml-19980210`.

[97] A.C. Yao. On random 2–3 trees. *Acta Informatica*, 9:159–170, 1978.

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

[99] C. Zaniolo. The database language GEM. In *Proc. of the 1983 ACM SIGMOD*, San Jose, California, 1983.

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

[101] 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.

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

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

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