S-Tree: A Dynamic Balanced Signature Index for Office Retrieval

Uwe Deppisch

Computer Science Department
Technical University Darmstadt
D-6100 Darmstadt, West-Germany

Abstract - The signature approach is an access method for partial-match retrieval which meets many requirements of an office environment. Signatures are hash coded binary words derived from objects stored in the data base. They serve as a filter for retrieval in order to discard a large number of nonqualifying objects. In an indexed signature method the signatures of objects stored on a single page are used to form a signature for that page. In this paper we describe a new technique of indexed signatures which combines the dynamic balancing of B-trees with the signature approach. The main problem of appropriate splitting is solved in a heuristic way. Operations are described and a simple performance analysis is given. The analysis and some experimental results indicate a considerable performance gain. Moreover, the new S-tree approach supports a clustering on a signature basis. Further remarks on adaptability complete this work.

1. Introduction

1.1 Background

In a modern office environment the retrieval of objects by content is one main requirement the data base system layer of an office information system should support [CHRI84, RABI85]. Generally these objects have a variable length and a complex structure. Besides simple data types like fixed length character strings, real and integer, there appear types of variable length; e.g. formatted data like multivalued attributes and especially unformatted data like text [SCHE81, SCHE82, GIBB83]. Figure 1 gives an example for these mixed type office objects. Here, *name* and *delivery* are atomic valued attributes, whereas the attribute *order* is of the set type and *order text* is of type text.

Single-level signatures are applied to support the retrieval of these office objects [CHRI84, FALO84]. Compared to alternative access methods [cf. SALT83b, RABI84, FALO85a], signatures offer the advantage to process partial-match, subset match, substring match and fuzzy match queries in an easy manner [cf. HARR71, RIVE76, SCHE77, SCHE78]. Because of their simple maintenance [cf. RABI84], signatures are well suited as an access path in data base systems in the environment of server-workstation networks.

Signatures are hash-coded binary words of fixed length; they represent abstractions of objects. Generally all bits of the signature are cleared to *null*, then a hash transformation is applied to the object's values to determine which bits are set to *one*. Several proposals have been made for the coding of signatures especially for textual attributes [e.g. HARR71, ROBE79, FALO85b, FALO85c], but their details are beyond the scope of this paper. Here, more emphasis is laid on the physical organization of the signatures. In single-level signature methods the signatures are stored separately from the objects and searched sequentially for retrieval purpose. Therefore, the same coding transformation is applied to the query to get the query signature. A scan of the signature file returns those object identifiers whose signatures contain *ones* in the positions the query signature does. After these match candidates are fetched from secondary storage a comparison with the query specifications finally eliminates false drops.

1.2 Organization of Signatures

In single-level signature methods every signature must be accessed and tested. This is done faster than sequential scan of the objects themselves, because the signatures are much smaller. Obviously the retrieval performance of the first step is linearly related with the number of objects. The more objects exist the more time is spent on scanning the signature file [cf. RABI84].

This research was supported by the Scientific Center Heidelberg of IBM Germany within the project "Databases in Server-Workstation Environment".

Permission to copy without fee all or part of this material is granted provided that the copyright notice of the "Organization of the 1986-ACM Conference on Research and Development in Information Retrieval" and the title of the publication and its date appear.
© 1986 Organization of the 1986-ACM Conference on Research and Development in Information Retrieval

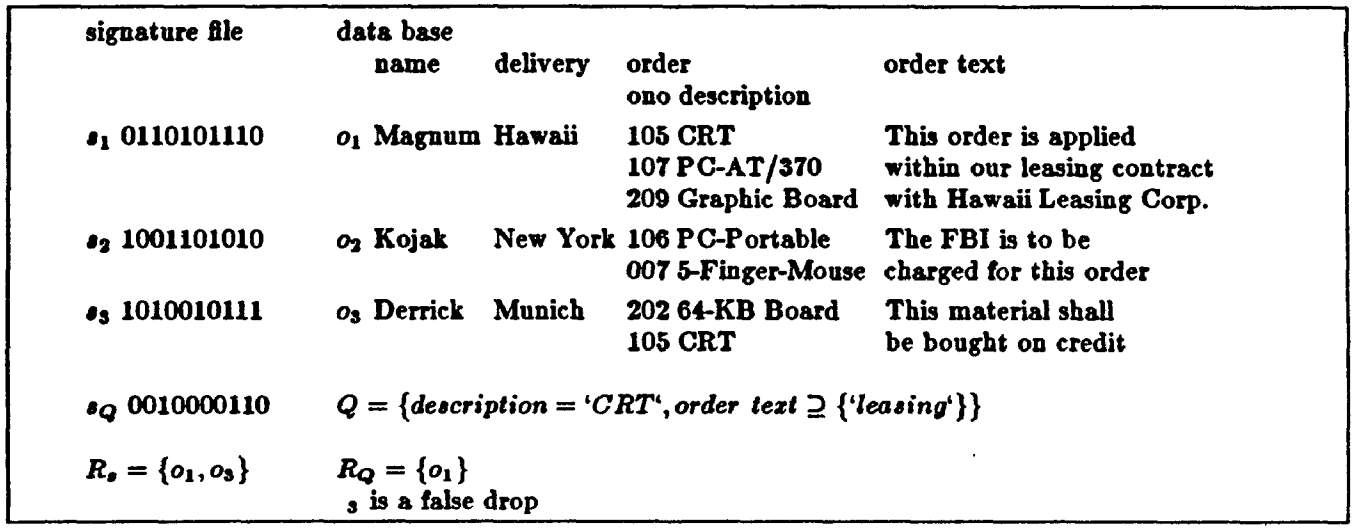

fig. 1: single-level signature method

To support a faster access, multi-level signature methods are suggested [PFAL80, SACK83]. In the *indexed signature* approach [PFAL80] a signature in the i-th level is created by superimposing (OR-ing) the signatures contained in one page of the (i-1)-th level. On the first level (leaves) we have one signature associated with one object or the object itself. Retrieval is processed by recursively searching the index tree with the query signature as a filter to cut off paths that cannot satisfy the query. In static environments this method provides a good retrieval performance if the file creation is done well [see PFAL80].

However, the office environment is dynamic [CHRI-84]. Therefore we need a method, which provides fast retrieval and which can also be balanced in the presence of insertions, deletions and updates. That is, no periodic reorganisation should be required. In this paper we want to show how dynamic restructuring as in B-trees [BAYE72] can be adopted to the signature approach. Therefore a new appropriate splitting technique is developed.

In the sequel we give a new approach of indexed signatures, called S-tree. The S-tree provides a considerable performance gain and achieves a clustering on a signature basis. In section 2 we show the motivation for the new approach, section 3 gives a definition of the S-tree. Section 4 presents the operations retrieve, insert, delete together with the maintenance procedures. The performance of retrieval is analysed in the following section 5. Section 6 gives some experimental results. In section 7 we outline the clustering feature; section 8 gives some remarks on adaptability. Finally, section 9 summarises the results.

2. Basic Concept of S-Tree

In the dynamic office environment retrieval and insertions are frequent operations. Applying the indexed signature approach here, means to store the signatures in the order the associated objects are inserted. That is, new signatures are entered into the last partially filled node, and ORed to the covering signatures above [cf. PFAL80]. Although [PFAL80] suggests to support the expectation of many insert operations by partially filling the nodes at creation time, performance may decrease because the signature tree is designed as a static structure. Moreover, a frequent processing of a time consuming periodic reorganization should be avoided in the dynamic office environment. Often new inserted signatures would not be inserted into the appropriate leaf (see fig. 2). Here, "appropriate" means the leaf where similar signatures are stored. "Similar signatures" are signatures with many set *ones* in same positions.

The superimposing of dissimilar signatures causes an increase of set *ones*. For example, consider the four signatures of node B2 in fig. 2, each with three set *ones*. By superimposing these signatures results the first signature in root B1 with seven set *ones*. Consequently, the nodes near the root become less selective and more paths of the tree must be searched for retrieval. Consider the query signature $s(Q) =< 10100010 >$ applied on the tree given in figure 2; all nodes must be accessed.

To avoid such a degeneration we should insert new signatures into leaves where similar signatures were already stored. If the appropriate leaf is already full, we will partition the signatures into two groups according to their similarity and put them into two nodes. This will prevent, that the superimposing of a single node's signatures to obtain the signature of the next level, causes a quick increase in set *ones*. Figure 3 shows such an S-tree, applied on the same object base as fig. 2. Consider the leaf N7; four similar signatures are ORed to form the first signature in node N3, which obtains four *ones*, just a single *one* more. If we apply the same query with $s(Q) =< 10100010 >$ we must search just one short path (N1, N3) to recognise that no match exists in the object base. A successful search is processed faster too, e.g.

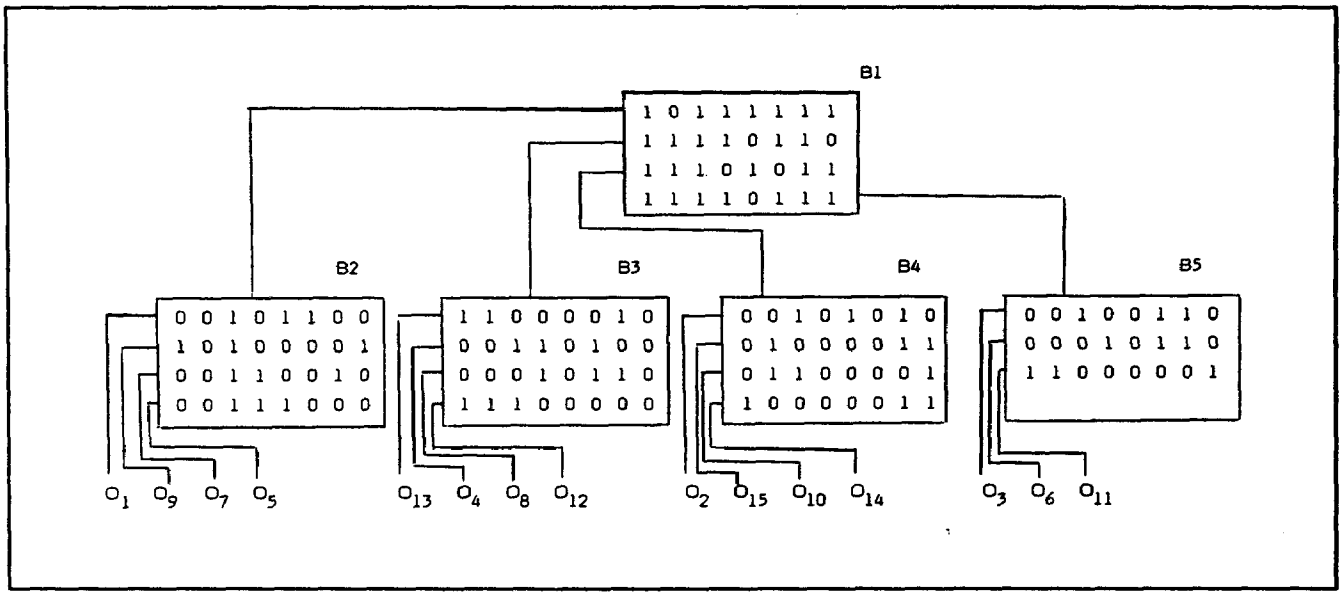

fig. 2: static indexed signature approach

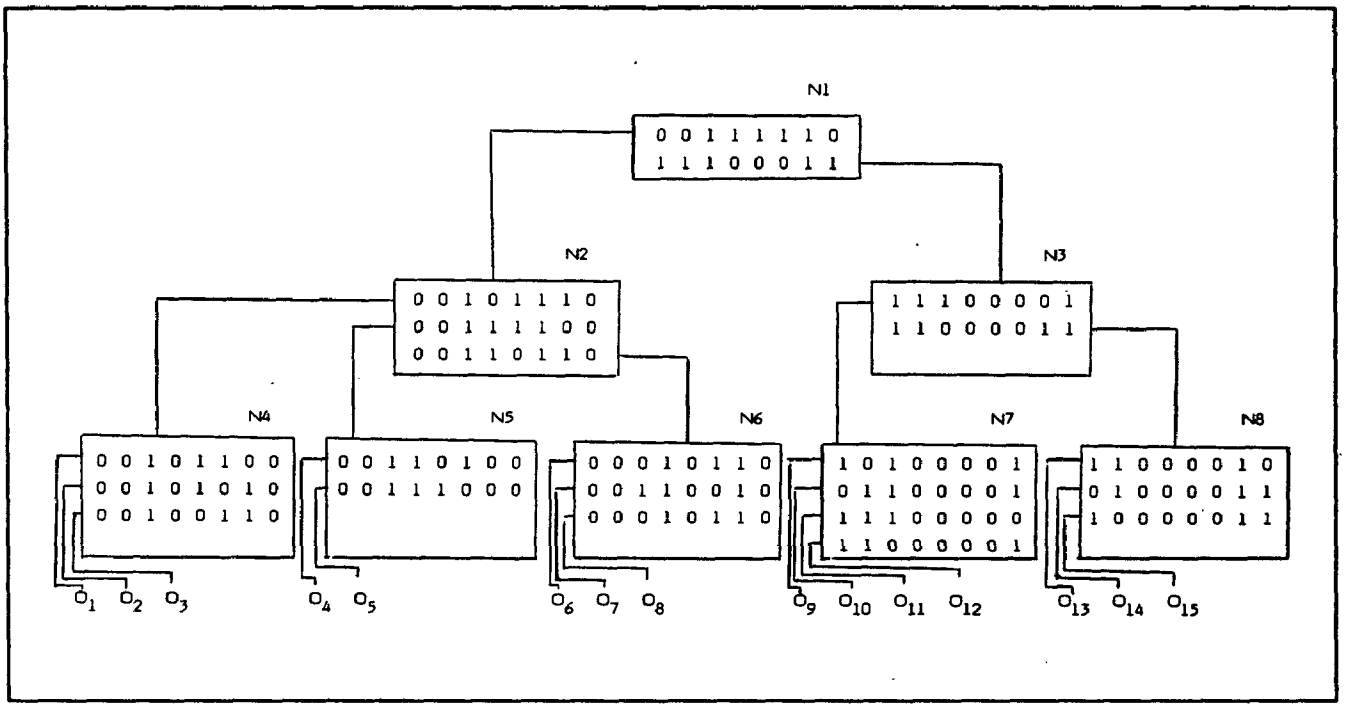

fig. 3: S-tree (4,2,3)

$s(Q) =< 11000010 >$. Moreover, due to the dynamic restructuring, the S-tree needs no periodic reorganisation to keep the performance.

3. Structure of S-Tree

3.1 Definition

Similar to a B^+-tree, an S-tree is a height balanced multiway tree, whose index part is managed like a B-tree [cf. BAYE72]. Each node corresponds to a page. The leaf nodes contain either the objects or object identifiers (*oids*). The former case we call an *immediate S-tree*, the latter a *mediate S-tree*. The leaves of a mediate S-tree contain entries of the form $< s, oid >$ where the object is accessed by the *oid*. The signature s is generated by applying an appropriate hash transformation on the object's attribute values, which maps them into a bit string $s = b_1|b_2|\ldots|b_L$ of fixed length L with $b_i \in \{0,1\}$. A signature in a non-leaf node is defined by superimposing the signatures contained in it's son node (via the signature operator σ). Therefore, entries E in non-leaf nodes have the form $< s, p >$ with the property

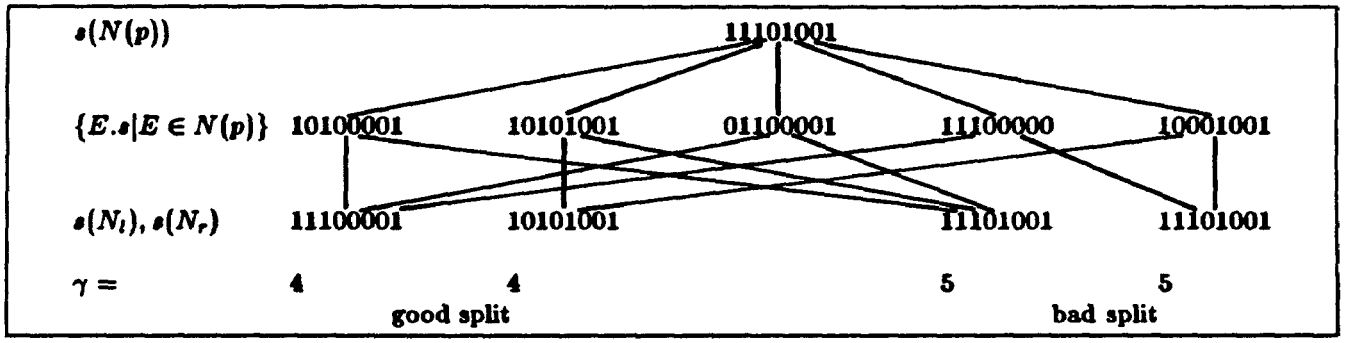

fig. 4: Splitting (K=4,k=2)

[1] $s = s(N(p)) := \sigma(\{E.s | E \in N(p)\}) := \bigvee_{E \in N(p)} E.s$

where $N(p)$ refers to the Node p and $E.s$ denotes the signature component of an entry E. Now we can define a *mediate S-tree* of the type (K, k, h), $K, k, h \in N_0$, with the following properties:

(1) Each path from the root to any leaf has the same length h (height).
(2) The root has at least 2 and at most K sons unless it is a leaf.
(3) Every node except the root has at least k and at most K sons.
(4) The signatures contained in each non-leaf node are minimal w.r.t. [1].

The height h of an S-tree for n objects is at most $\lceil log_k n \rceil - 1$. The minimum number of nodes in an S-tree is $\#_{min}(N) = 1 + 2\sum_{i=0}^{h-2} k^i = 1 + 2\left(\frac{k^{h-1}-1}{k-1}\right)$, the maximum number is $\#_{max}(N) = \sum_{i=0}^{h-1} K^i = \frac{K^h - 1}{K-1}$. The minimum space utilisation for all nodes is $\frac{k}{K}$. Contrary to a B-tree, there exists no order of entries within a node. Moreover, due to hash-coding it is possible that the same signature appears multiple times in an S-tree (see fig. 3: the signatures of object O_6 and O_8).

3.2 Problem of Splitting

According to the original B-tree we could choose the minimal branching factor $k = \frac{K}{2}$. However, the entries in the S-tree have not only a guidance function. Since the signatures are abstractions of sets, they contain already some more information about the objects. Due to superimposing, the signatures near the root contain more set *ones* than signatures near the leaves (see fig.3). In the retrieval process, it is not unlikely that nodes with many set *ones* are accessed, without leading to a hit (cf. the first example in section 2). Therefore a more flexible splitting technique is required which allows $1 \leq k \leq \frac{K}{2}$ (cf. analog problem for spatial searching [GUTT84]. To achieve a good split we must pay attention to partition the $K+1$ signatures belonging to one node to split, such that the signature weights

$$\gamma(s) := \sum_{i=1}^{L} b_i$$

of the two new nodes are low. That is, their signatures should have as few set *ones* as possible. Figure 4 illustrates an example for a good and a bad split. Here, the bad split produces not only heavier signatures, but also the same signature pattern. Consequently, retrieval proceeds either on both nodes or on none, i.e. the split does not increase selectivity. Let $s(N_l)$ and $s(N_r)$ be the signature of the left and the right brother node after splitting, we have to solve the following problem:

$$min! \quad \gamma = \gamma(s(N_l)) + \gamma(s(N_r))$$

with the constraint

$$k \leq \#(N_l), \#(N_r) \leq K.$$

However, since up to now no efficient algorithm for that problem is known (to the author), we solve this problem heuristically, and hence possibly suboptimal.

4. Operations of the S-Tree

4.1 Retrieval

Similar to the B-tree the search algorithm descends within the S-tree from the root down to the leaves. However, more than one subtree may be visited recursively [cf. PFAL80]. To demonstrate the retrieval, consider the following query signature $s(Q) =<$ 00110000 $>$ applied on the S-tree of fig. 3. The first signature of the root leads us to his son (N2), because it contains a *one* at the third and the fourth position. Processing proceeds here at the second and third signature. In the leaf N5 we find two match candidates (O_4, O_5), in the leaf N6 we have one (O_7). However, it is possible to gather all object identifiers first, sort them according to page numbers and fetch and check the match candidates in one pass, i.e. without visiting a single page twice.

4.2 Insertion

After inserting a leaf entry in a suitable leaf, the signature of that leaf must be updated. If the leaf signature has changed, this change must be propagated upwards within the S-tree. If the leaf node becomes overfull, it has to be split. Splits propagate upwards too. To find the appropriate leaf node, the CHOOSE function descends in the

```
Initialize NODE := ROOT.NODE; START := ROOT.DEPTH; DEPTH := START; s := s(Q);
Procedure RETRIEVE (START, DEPTH, NODE, s)
Fetch (NODE);
for each entry E in NODE do
    if s ∧ E.s = s
    then if DEPTH < HEIGHT
         then RETRIEVE (START, DEPTH + 1, E.p, s)
         else Check (E.oid)    (* check object and return *)
end;                           (* good drops to response set *)
if PAGE.NEXT(NODE) ≠ nil ∧ DEPTH = START          (* for NEXT option *)
then RETRIEVE (START, DEPTH, PAGE.NEXT(NODE), s); (* see section 8 *)
end; (* end RETRIEVE *)
```

```
Initialize STOP-DEPTH = HEIGHT;
Procedure INSERT (IE, STOP-DEPTH)    (* IE = entry to be inserted *)
DEPTH := ROOT.DEPTH;
if DEPTH = 1 then NODE := CHOOSE (ROOT, DEPTH, STOP-DEPTH, IE.s)
             else NODE := CHOOSE-NEXT (ROOT, DEPTH, STOP-DEPTH, IE.s);
if NODE is full then SPLIT (NODE, DEPTH, IE)
                else begin
                     Add IE to NODE;
                     ADJUST (NODE, DEPTH, IE.s);
                     end;
end; (* end INSERT *)
```

```
Function CHOOSE (NODE, DEPTH, STOP-DEPTH, s) returns NODE varifies DEPTH
Fetch (NODE);
if DEPTH = STOP-DEPTH then CHOOSE := NODE
                      else begin
                           Find entry E with minimal ε(s, E.s);
                           CHOOSE := CHOOSE (E.p, DEPTH + 1, STOP-DEPTH, s);
                           end;
end; (* end CHOOSE *)
```

```
Procedure ADJUST (NODE, DEPTH, s)
Fetch (Parent(NODE));
Find entry E with E.p = NODE;
s := E.s ∨ s;
if E.s ≠ s then begin
                E.s := s;
                if DEPTH > ROOT.DEPTH then ADJUST (Parent(NODE), DEPTH - 1, s);
                end;
end; (* end ADJUST *)
```

S-tree on one single path. This path is selected step by step by applying a similarity measure to the signatures in a node (s) and the signature to be inserted (s'). We may use the *Hamming distance metric*:

$$\delta(s, s') := \gamma(s \vee s') - \gamma(s \wedge s').$$

However, to achieve a high selectivity in each node we need a low weight $\gamma(s(N))$. Therefore it is better to choose that node to proceed which obtains the lowest weight increase. So we define the *weight increase distance* ϵ:

$$\gamma(s \vee s') = \gamma(s) + \epsilon(s, s')$$

$$\epsilon(s, s') := \gamma(s \vee s') - \gamma(s) = \gamma(\neg s \wedge s').$$

```
Procedure SPLIT (NODE, DEPTH, IE)
Add IE to NODE;   (* virtually *)
α := {(Find Entry E with maximal γ(E.s))};        (* find heaviest entry *)
β := {(Find Entry E with maximal ε(s(α), E.s))};  (* find entry with maximal weight increase *)
for each remaining Entry E do
    if #α + #(remaining entries) = k then Assign all remaining entries to α (* stop *);
    if #β + #(remaining entries) = k then Assign all remaining entries to β (* stop *);
    if ε(s(α), E.s) < ε(s(β), E.s) then Assign E to α
                                   else Assign E to β;
end;
Get a fresh page NEWNODE;
Set PAGE.NODE(NEWNODE) to β;
IE :=< ⋁_β E.s, NEWNODE >;
Set PAGE.NODE(NODE) to α;
if NODE ≠ ROOT.NODE then begin
                                   Fetch (Parent(NODE));
                                   Find entry E with E.p = NODE;
                                   E.s := ⋁_α E.s;
                                   NODE := Parent(NODE);
                                   if NODE is full then SPLIT (NODE, DEPTH, IE)
                                                   else begin
                                                        Add IE to NODE;
                                                        ADJUST (NODE, DEPTH, IE.s);
                                                        end;
                                   end;
else if CUT = false then begin
                         Get a fresh page ROOT;
                         Add < ⋁_α E.s, NODE > to ROOT;
                         Add IE to ROOT;
                         HEIGHT := HEIGHT +1
                         end;
                    else begin
                         PAGE.NEXT(NEWNODE) := PAGE.NEXT(NODE);
                         PAGE.NEXT(NODE) := NEWNODE;
                         end;
end; (* end SPLIT *)
```

Notice, the weight increase distance is not commutative and therefore no metric. By applying the weight increase distance, the adjustment propagation is better bound. For refinement we may use the Hamming distance as a secondary strategy.

4.3 Splitting

If a node into which an entry should be inserted is already full, it will be split. That is, the $K + 1$ entries should be partitioned into two nodes in such a way that the weight of the two new signatures is low and the Hamming distance between them is high. This reduces the probability that a query requires both nodes to be accessed. Instead of an exhaustive enumeration of all possible partitions, we give a simple heuristic, which needs only linear time.

In the first step the algorithm picks that entry that has the heaviest signature weight. Since this entry would be the heaviest in one of the two partitions too, and since it must be assigned to one of them, we choose this entry as *seed* α. As a second *seed* β we select that entry, which has the maximum number of set *ones* in positions where α has *nulls*, i.e. the entry providing the maximal weight increase to α. This is reasonable since superimposing the signatures to achieve the node's signature never reduces the weight. Thereafter the remaining signatures are assigned to that seed where they achieve the lowest weight increase, in equality to β because of it's lower weight. Further refinements are possible but not very promising because of the minimal node load k.

4.4 Deletion

Together with pure deletions we take updates into consideration that change the objects' signatures. Updates are managed via delete and insert. Deleting entries from an S-tree causes some maintenance to keep the desired tree properties. The original B-tree presents the catena-

```
Procedure DELETE (s, oid)
SEARCHSTOP := false;
SEARCHDEL (ROOT.NODE, ROOT.DEPTH, s, oid, SEARCHSTOP);
for DEPTH = ROOT.DEPTH until HEIGHT do
    if U(DEPTH) ≠ 0 then INSERT (U(DEPTH), DEPTH);
end;
end; (* end DELETE *)
```

```
Procedure SEARCHDEL (NODE, DEPTH, s, oid, SEARCHSTOP)
Fetch (NODE);
repeat for each entry E in NODE
        if s ∧ E.s = s
        then if DEPTH < HEIGHT
             then SEARCHDEL (E.p, DEPTH + 1, s, oid, SEARCHSTOP);
             else if E.oid = oid then begin
                                      Delete entry E;
                                      SEARCHSTOP := true;
                                      end;
until SEARCHSTOP;
if SEARCHSTOP = true and NODE has fewer than k entries and NODE ≠ ROOT
        then begin
             F := Parent(NODE);
             Fetch (F);
             Delete entry E with E.p = NODE in F;
             U(DEPTH) := NODE;
             end;
end; (* end SEARCHDEL *)
```

tion and underflow algorithm for that purpose [BAYE72]. In the approach of [GUTT84] underfull nodes are eliminated. This is propagated upwards, and then the entries with possible subtrees are reinserted at the correct level. Here we adopt the reinsertion strategy because updates and deletions are less frequent than retrieval or insertions, especially in the office environment. The reinsertion incrementally refines the S-tree structure according to adjacency of signatures. Furthermore the efficiency of reinsertion should be comparable the corresponding B-tree algorithms because pages requested are already fetched into primary memory due to the proceeding search [GUTT84].

5. Analysis

To estimate the performance we assume that a partial-match query with t set *ones* should be processed. We give a simple analysis in terms of node fetches (I/O). We make the following assumptions concerning the generation of the signatures:

(a1) each signature has the same signature weight $\gamma < \frac{L}{3}$,

(a2) hash-coding provides an equal distribution of set *ones*.

Notice that assumption (a1) does not fix the coding method. From assumption (a2) follows, that the $\binom{L}{\gamma}$ possible distinct signatures are equally probable. For a signature which is derived by superimposing λ signatures with weight γ, the probability that t prespecified positions contain *ones* is [ROBE79, SACK83]:

$$p(t) = \sum_{j=0}^{t} \frac{(-1)^j \binom{t}{j} \binom{L-j}{\gamma}^{\lambda}}{\binom{L}{\gamma}^{\lambda}}.$$

For the sake of simplicity Roberts derived in [ROBE79] the following approximation:

$$p(t) \approx \left[1 - \left(1 - \frac{\gamma}{L}\right)^{\lambda}\right]^{t}.$$

We can explain the approximation as follows: $\frac{\gamma}{L}$ is the probability that one prespecified bit position contains a *one*. Hence $1 - \frac{\gamma}{L}$ is the probability for a *null*. Raising to power with λ results the probability to keep the *null* by superimposing λ signatures. The correct probability that one prespecified position contains a *one* is 1 minus that term. By raising to power with t the approximation for t prespecified positions is achieved.

In the S-tree one signature in an i-th level node is generated by superimposing the signatures in the corresponding son. Therefore each signature in a node is the result from superimposing a subset of several objects' signatures. At the root this subset has the cardinality

$\lambda(1) = \frac{n}{\beta(1)}$, where n is the number of all objects and $\beta(1)$ describes the expected utilisation of the root. We obtain for depth d

$$\lambda(d) = \frac{n}{\prod_{i=1}^{d} \beta(i)}.$$

We keep the assumption that all L positions are set with equal probability for an estimation purpose, although the combining of similar signatures in deeper levels produces lower weights. For each depth d we obtain the following formula

$$p(t,d) \approx \left[1 - \left(1 - \frac{\gamma}{L}\right)^{\lambda(d)}\right]^{t}.$$

Now the number of matches at depth d is

$$\prod_{i=1}^{d} (\beta(i) \cdot p(t,i)),$$

and each match causes a node access of level $d+1$. Consequently the number of node accesses in the whole S-tree can be estimated by

$$\sum_{d=1}^{h-1} \left(\prod_{i=1}^{d} (\beta(i) \cdot p(t,i)) \right) + 1.$$

6. Experimental Results

We have implemented the S-tree [SEMS86] to compare its performance against the single-level signature method. In this section we present preliminary results of three experiments. Each experiment involves two steps, first building an S-tree of pseudo-random object signatures and second running a set of pseudo-random query signatures against it.

The generation of pseudo-random query signatures is designed independent of the object signatures. Each S-tree is tested with several query signature weights. For each single weight 60 retrieval operations with different signatures are executed and the average number of I/Os is presented. Each set of query signatures contains successful and unsuccessful searches. A special test set containing only successful query signatures requires up to four I/Os more than the presented results for the mixed set.

The following experiment parameters are set. For each experiment the page size is 2 K bytes, the size of a pointer is 4 bytes. Experiment 1 concerns signatures with length 256 bits and weight 40. The minimal node load k is 20, the maximal node load K is 56. Experiment 2 and 3 concern signatures with length 512 and weight 80 resp. 120, $k = 10, K = 30$. The lower part of the presented tables shows the average number of I/Os for the S-tree corresponding to the query weight. For comparison purpose we have computed the number of I/O-operations for scanning the one-level signature file, where the same signatures were stored sequentially and dense, i.e. without gaps. The results here are independent of the weight of the query signature.

In comparison with the single-level signature method the S-tree of experiment 1 achieves a better performance (see table 1). Moreover, performance is improved through the longer signatures. For example, the S-tree based on 256/40 signatures (table 1) needs 152 I/Os for a query with a 10 set *ones* (a quarter weight), while the S-tree with the 512/80 signatures (table 2) requires only 75 I/Os for a corresponding query with 20 set *ones*.

Compared to the single-level method the S-tree with a signature length of 512 bits improves the performance

number of objects	1000	2000	5000	10000
single-level method	18	36	90	179
S-tree				
query weight				
10	18	32	75	152
20	12	19	45	87
30	8	12	28	51
40	6	8	18	32

table 1: I/Os of experiment 1 (256/40,20,56)

number of objects	1000	2000	5000	10000
single-level method	34	67	167	334
S-tree				
query weight				
5	34	65	160	315
10	19	36	90	177
20	9	15	38	75
30	6	10	24	46
40	5	8	19	36
50	4	7	17	32
60	4	7	17	31
70	4	7	16	31
80	4	7	16	30

table 2: I/Os of experiment 2 (512/80,10,30)

number of objects	1000	2000	5000	10000
single-level method	34	67	167	334
S-tree				
query weight				
10	39	74	192	391
20	28	51	130	240
30	20	36	91	172
40	15	26	68	126
50	12	20	52	94
60	10	16	41	74
70	8	13	34	61
80	7	11	28	52
90	6	10	25	47
100	5	9	22	41
110	5	8	21	38
120	5	8	19	36

table 3: I/Os of experiment 3 (512/120,10,30)

even if the signature weight is increased (see table 3). The tables 2 and 3 show that the S-tree performs better for a query weight of 10% of object signature weight or more. The more bits are set to *one* in the query signature the less nodes are accessed. The height of the S-trees in experiment 2 and 3 is three, they increase the storage overhead of single-level signatures by factor 1.4 up to 1.9.

7. Clustering and Refined Retrieval

The immediate S-tree contains the objects in it's leaves. Their signatures may be stored with them, but they can also be computed by demand. To achieve nearly the same branching factor we may define a larger page size for the leaves, because the objects are larger than signatures. This approach groups objects together with similar signatures. Therefore the immediate S-tree realizes a *clustering* according to the hash-coding [cf. SALT78, RIJS79]. Performance is increased because it is probable that more objects of one leaf are match candidates. Due to the split operation, the S-tree keeps the clustering in a highly dynamic environment. There is no extra reorganisation and no extra cluster generation necessary. But the quality of clustering depends on the hash-coding. Here a hash-coding is required which preserves the similarity of the objects.

A second advantage of the S-tree is that it allows a refined retrieval. In the office environment a browsing capability is needed [TSIC85b]. This iterative browsing function can be supported by applying relevance feedback methods [cf. RIJS79, SALT83b] or nearest neighbor search on signature basis. One possibility may be that the user pinpoints those objects in the initial response set which seem to be relevant for him. Thereafter the signatures associated with these selected objects may be ANDed to discover their joint properties. The resulting signature may be applied on the object base for the next search iteration.

Nearest neighbor search on signature basis can be supported by extending the signature test with the Hamming distance or a distance ϱ which allows a certain number of set *one* positions to be *null*. The latter case can be done easily within the S-tree retrieval procedure. We just replace the test condition with $\gamma(s(Q) \wedge E.s) \geq \gamma(s(Q)) - \varrho$. Another refinement is the opportunity to embed a more sophisticated filter [e.g. similar to SALT83a] instead of one single query signature.

8. Adaptability

8.1 Light Queries

A partial-match query is defined as a *light query* if it's signature weight is lower than a given weight g

$$\gamma(s(Q)) < g.$$

In the case of a light query the probability is high, that in a big and high S-tree a great part of the tree has to be searched recursively. The signature of a light query has only a few set *ones* and therefore it is a less selective filter. Hence it is suitable here to search directly the leaves of a mediate S-tree or one level above in the case of the immediate S-tree. To support this, we chain the leaves containing first-level signatures (see fig. 5). This is similar to the one-level signature scan. As anchor we use the new parameter LEAF. To process a light query we call the RETRIEVE procedure with the parameters $(LEAF.DEPTH, LEAF.DEPTH, LEAF.NODE, s(Q))$. That is, the START parameter is equal to the depth of the leaves. Then the second part of the RETRIEVE procedure is processed which searchs sequentially the leaves.

8.2 Cutting the Top of the S-Tree

For very large object bases we cannot exclude that the root and nodes near to it have a heavy signature weight or even $\gamma(s) = L$. In such a case nearly all bits are set to *one* and the selectivity of these nodes is decreased. The reason for this is given in superimposing, heuristical splitting and the minimum node utilisation k. Here it is reasonable to cut off the top of the S-tree. Descending from the root those nodes are discarded, which contain mostly signatures heavier than a given threshold. By chaining the new roots we obtain a forest of S-trees (see fig. 6), which is more suitable for very large data bases. The consequences resulting from that approach are already taken into account in the given algorithms above via the use of the NEXT-pointer. Since we initialise the START parameter with the new depth of the root, the second part of the RETRIEVE procedure provides a search of the forest of S-trees. For insertion, the CHOOSE-NEXT function visits firstly every root and then descends the most suitable S-tree. Finally, in the split process the chaining of roots is controlled by the CUT variable.

9. Summary and Conclusion

Recently, several related approaches have been made. To build a tree of signatures, we could interprete the signatures as binary numbers and apply an ordinary B^+-tree (similar to the skd-B-tree in [OREN84]). But this would not perfectly utilise the feature that a signature due to hash-coding always describes a set of objects [DADA83]. The approach of [PRAB83] applies a signature for each node of an ordinary B-tree as an additional filter. In contrast to these approaches, we keep signatures as guides on the path and use the descriptive power of signatures to describe sets.

In this paper we have presented a new approach of dynamic indexed signatures for the retrieval of objects in an office information system. Due to heuristical splitting the S-tree is balanced and keeps adjacent signatures together. This increases retrieval performance in most cases. An important and desired feature is that the answer to a more specified query usually is found quicker than the answer to a less specified one. The S-tree im-

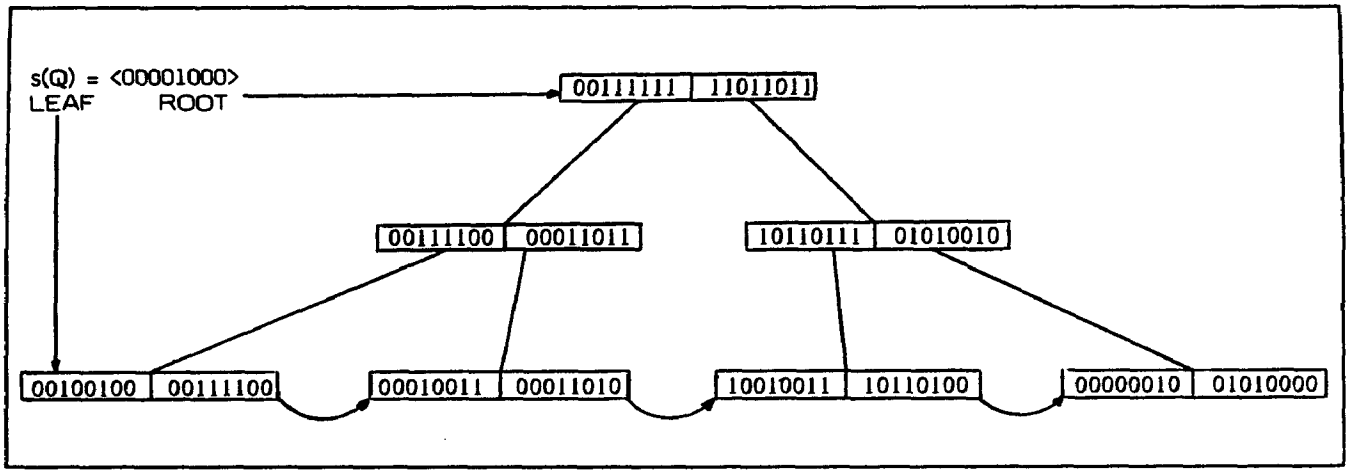

fig. 5: S-tree with light query

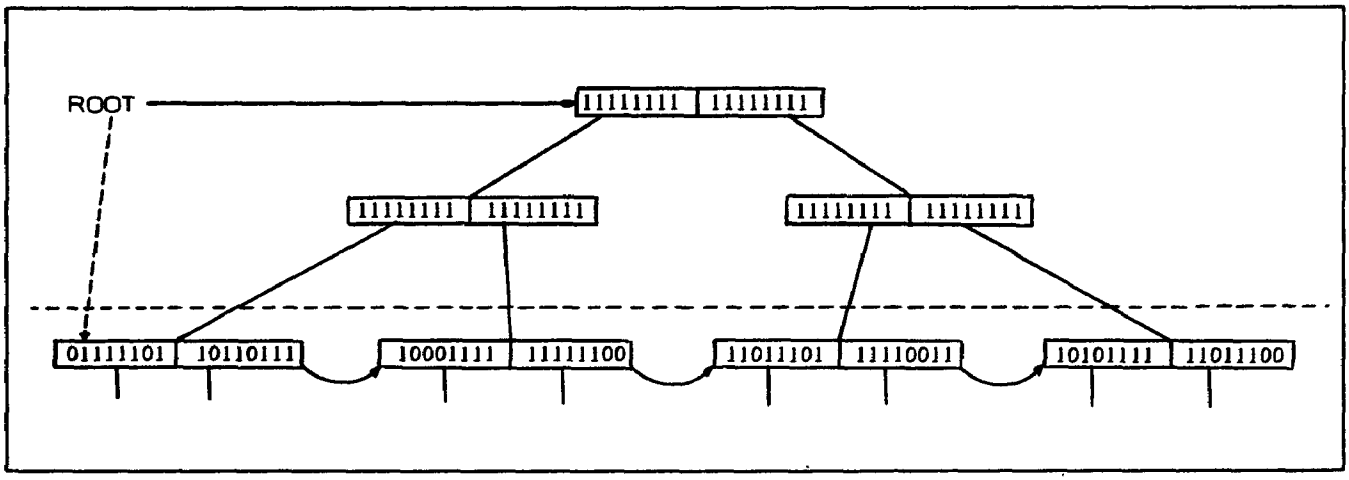

fig. 6: cutting the top of an S-tree (2,1,h)

proves this feature. Moreover, the S-tree supports clustering on a signature basis, since similar signatures are grouped together. This could be the basic idea for applying more refined retrieval methods. Because of the S-tree's adaptability it seems usable in many office environments. Therefore, we have started an implementation to prove it's quality in some experiments [SEMS86], and plan an integration into a data base system for advanced applications (DASDBS [DEPP85] or AIM [DADA86]).

Acknowledgements

Prof. H.-J. Schek provided many interesting suggestions and thoughtful comments. His help is gratefully acknowledged. Moreover the author wishes to thank F. Gebhard (GMD), P. Pistor (IBM HDSC), V. Obermeit, H.-B. Paul, M.H. Scholl, W. Waterfeld, G. Weikum, T. Semsroth and A. Wolf, who have contributed to clarifying discussions of the material. Thanks also to Lis Klinger who helped to preparate the figures.

References

BAYE72 Bayer, R., McCreight, E.M.: Organisation and Maintenance of Large Ordered Indexes, Acta Informatica 1972, pp. 173 - 189.

CHRI84 Christodoulakis, S. et al.: Development of a Multimedia Information System for an Office Environment, Proc. VLDB 1984, pp. 261 - 271.

DADA83 Dadam, P., Pistor, P., Schek, H.-J.: A Predicate oriented Locking Approach for Integrated Information Systems, in: Mason, R.E.A. (ed.): Information Processing 83, Proc. IFIP 1983, pp. 763 - 768.

DADA86 P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch: A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, accepted for SIGMOD 1986.

DEPP85 Deppisch, U., Obermeit, V., Paul, H.-B., Schek, H.-J., Scholl, M.H., Weikum, G.: The Storage Subsystem of a Data Base Kernel System (in German), Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications 1985, pp. 421 - 440; English version available as Technical Report DVSI-1985-T1, TU Darmstadt, 1985.

FALO84 Faloutsos, C., Christodoulakis, S.: Signature Files: An Access Method for Documents and its Analytical Performance Evaluation, TOOIS 1984, pp. 267 - 288.

FALO85a Faloutsos, C.: Access Methods for Text, Computing Surveys 1985, pp. 49 - 74.

FALO85b Faloutsos, C.: Signature files: Design and performance comparison of some signature extraction methods, Proc. SIGMOD 1985, pp. 63 - 82.

FALO85c Faloutsos, C., Christodoulakis, S.: Design of a Signature File Method that Accounts for Non-Uniform Occurence and Query Frequencies, Proc. VLDB 1985, pp. 165 - 170.

GIBB83 Gibbs, S., Tsichritzis, D.: A Data Modelling Approach for Office Information Systems, TOOIS 1983, pp. 299 - 319.

GUTT84 Guttman, A.: R-Trees: A dynamic index structure for spatial searching, Proc. SIGMOD 1984, pp. 47 - 57.

HARR71 Harrison, M.C.: Implementation of the Substring Test by Hashing, CACM 1971, pp. 777 - 779.

OREN84 Orenstein, J.A., Merrett, T.H.: A Class of Data Structures for Associative Searching, Proc. PODS 1984, pp. 181 - 190.

PFAL80 Pfaltz, J.L., Berman, W.J., Cagley, E.M.: Partial-Match Retrieval using Indexed Descriptor Files, CACM 1980, pp. 522 - 528.

PRAB83 Prabhakar, T.V., Sahasrabuddhe, H.V.: Signature Trees - A Data Structure for Index Organisation, Proc. IEEE Conf. on Systems, Man and Cybernetics 1983, pp. 1145 - 1147.

RABI84 Rabitti, F., Ziska, J.: Evaluation of Acces Methods to Text Documents in Office Systems, in: Rijsbergen, van, C.J. (ed.): Proc. Research and Development in Information Retrieval 1984, pp. 21 - 40.

RABI85 Rabitti, F.: A Model for Multimedia Documents, in: TSIC85a, pp. 227 - 250.

RIVE76 Rivest, R.L.: Partial-Match Retrieval Algorithms, SIAM Journal of Computing 1976, pp. 19 - 50.

RIJS79 Rijsbergen, van, C.J.: Information Retrieval, 2nd ed., London: Butherworth 1979.

ROBE79 Roberts, C.S.: Partial-Match Retrieval via the Method of Superimposed Codes, Proc. IEEE 1979, pp. 1624 - 1642.

SACK83 Sacks-Davis, R., Ramamohanarao, K.: A two-level superimposed coding scheme for Partial Match Retrieval, Information Systems 1983, pp. 273 - 280.

SALT78 Salton, G., Wong, A.: Generation and Search of Clustered Files, TODS 1978, pp. 321 - 346.

SALT83a Salton, G. Fox, E.A., Wu, H.: Extended Boolean Information Retrieval, CACM 1983, pp. 1022 - 1036.

SALT83b Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval, New York: McGraw-Hill 1983.

SCHE77 Schek, H.-J.: Tolerating Fuzziness in Keywords by Similarity Searches, Kybernetes 1977, pp. 175 - 184.

SCHE78 Schek, H.-J.: The Reference String Access Method and Partial Match Retrieval, IBM Scientific Center Report TR 77.12.008; partly contained in: The Reference String Indexing Method, in: Bracchi, G., Lockemann, P.C. (eds.): Proc. Information System Methodology 1978, Springer LNCS 65, pp. 432 - 459.

SCHE81 Schek, H.-J.: Methods for the Administration of Textual Data in Data Base Systems, in: Oddy, R.N. et al. (eds.): Information Retrieval Research, London: Butherworth 1981, pp. 218 - 235.

SCHE82 Schek, H.-J., Pistor, P.: Data Structures for an Integrated Data Base Management and Information Retrieval System, Proc. VLDB 1982, pp. 197 - 207.

SEMS86 Semsroth, T.: Experimental Studies on S-Trees, Master Thesis, TU Darmstadt, 1986, forthcoming.

TSIC85a Tsichritzis, D. (ed.): Office Automation, Berlin, Heidelberg: Springer 1985.

TSIC85b Tsichritzis, D., Christodoulakis, S., Lee, A., Vandenbroek, J.: A Multimedia Filing System, in: TSIC85a, pp. 43 - 65.