IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP5047

Constructing Signature Graphs for Signature Files

Yangjun Chen
Dept. Business Computing, University of Winnipeg, 515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9,
ychen2@uwinnipeg.ca

Yibin Chen
Dept. Business Computing, University of Winnipeg, 515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9, r.mcfadyen@uwinnipeg.ca

ABSTRACT

The signature file method is a popular indexing technique used in information retrieval and databases. It excels in efficient index maintenance and lower space overhead. However, it suffers from inefficiency in query processing due to the fact that for each query processed the entire signature file needs to be scanned. In this paper, we introduce a graph structure, called a signature graph, established over a signature file, which can be used to expedite the signature file scanning by one order of magnitude or more.

INTRODUCTION

In this paper, we propose a graph structure, called a *signature graph*, constructed over a signature file, which can be used as an index structure for documents, classes in object oriented databases and records in relational databases.

The signature file method was originally introduced as a text indexing methodology [Fa85, FLPS90]. Nowadays, however, it is utilized in a wide range of applications, such as in office filing [CTHP86], hypertext systems [FLPS90], relational and object-oriented databases [CS89, IKO93, LL92, SKRT95, YLK94], as well as in data mining [AB97]. Compared to the inverted index, the signature file is more efficient in handling new insertions and queries on parts of words. But the scheme introduces information loss. More specifically, its output usually involves a number of false drops, which may only be identified by means of a full text scanning on every text block short-listed in the output. Also, for each query processed, the entire signature file needs to be searched [CF84, Fa85, Fa92]. Consequently, the signature file method involves high processing and I/O cost. This problem is mitigated by partitioning the signature file, as well as by exploiting parallel computer architecture [CZ96, Le95, SK86].

During the creation of a signature file, each word is processed separately by a hashing function. The scheme sets a constant number (m) of 1s in the [1..F] range. The resulting binary pattern is called the word signature. Each text is seen to be composed of fixed size logical blocks and each block involves a constant number (D) of non-common, distinct words. The D word signatures of a block are superimposed (bit OR-ed) to produce a single F-bit pattern, which is the block signature stored as an entry in the signature file.

Fig. 1 depicts the signature generation and comparison process of a block containing three words (then D = 3), say "SGML", "database", and "information". Each signature is of length F = 12, in which m = 4 bits are set to 1. When a query arrives, the block signatures are scanned and many nonqualifying blocks are discarded. The rest are either checked (so that the "false drops" are discarded; see below) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for word signatures. The query signature is then compared to every block signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 1: (1) the block matches the query; that is, for every bit set in s_q, the corresponding bit in the block signature s is also set (i.e., $s \wedge s_q = s_q$) and the block contains really the query word; (2) the block doesn't match the query (i.e., $s \wedge s_q \neq s_q$); and

Fig. 1. Signature generation and comparison

text: … SGML … database …information …				matching
word signatures:		queries:	query signatures:	results:
SGML	010000100110	SGML	010000100110	match with OS
database	100010010100	XML	011000100100	no match with OS
information ∨	010100011000	informatik	110100100000	false drop
object signature (OS)	110110111110			

(3) the signature comparison indicates a match but the block in fact doesn't match the search criteria (false drop). In order to eliminate false drops, the block must be examined after the block signature signifies a successful match.

In this paper, we propose a method to speed up the (sequential) signature file scanning by introducing the concept of *signature identifiers* and establishing a graph structure, a *signature graph* for it just like a position tree for a text [AHU74]. But by the construction of a position tree, a *position identifier* is a continuous piece of character sequence, while by the construction of a signature graph a signature identifier is not a continuous piece of bit string.

A closely related work is the S-tree proposed in [De86]. It is in fact a R-tree built over a signature file. Thus, it can be used to speed up the locating of a signature in a signature file just like a R-tree for primary keys in a relational database. However, in the signature graph each path corresponds to a signature identifier which can be used to identify uniquely the corresponding signature in a signature file. It helps to find the set of signatures matching a query signature quickly.

Signature files can also be utilized as set access facility in OODBSs [IKO93]. Especially, according to the analysis of [IKO93], the bit-sliced signature file (BSSF) achieves a higher performance than the sequential signature file (SSF) by almost 50% (of time cost) in the best case. But the storage cost of BSSF doubles that of SSF and the update cost of BSSF triples that of SSF or more [IKO93]. Later on, we'll see that a signature graph has a much better time complexity and less update cost than BSSF but with almost the same storage cost.

SIGNATURE GRAPH

A first idea to improve the performance is to sort the signature file and then employ a binary searching. Unfortunately, this does not work due to the fact that a signature file is only an inexact filter. The following example helps for illustration.

Consider a sorted signature file containing only three signatures:

010	000	100	110
010	100	011	000
100	010	010	100

Assume that the query signature s_q is equal to 000010010100. It matches 100 010 010 100. However, if we use a binary search, 100 010 010 100 can not be found.

For this reason, we try a different way and organize a signature file into a graph, called a *signature graph*, which will be discussed in this section in great detail.

Definition of signature graphs

A signature graph working for a signature file is just like a *trie* [31, 34] for a text. But in a signature graph, each path visited to find a signature that matches a query signature corresponds to a signature identifier, which is not a continuous piece of bits, and quite different from a trie in which each path corresponds to a continuous piece of bits.

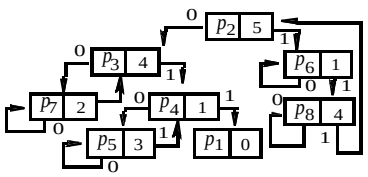
Definition 1. (*signature graph*) A signature graph G for a signature file $S = s_1.s_2 \ldots .s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = F$ for $k = 1, \ldots, n$, is a graph $G = (V, E)$ such that

1. each node $v \in V$ is of the form (p, *skip*), where p is a pointer to a signature s in S, and *skip* is a non-negative integer i. If $i > 0$, it tells that the ith bit of s_q will be checked when searching. If $i = 0$, s will be compared with s_q.
2. Let $e = (u, v) \in E$. Then, e is labeled with 0 or 1 and $skip(u) > 0$. Let $skip(u) = i$. If e is labeled with 0 and $i > 0$, the ith bit of the signature pointed to by $p(v)$ is 0. If e is labeled with 1 and $i > 0$, the ith bit of the signature pointed to by $p(v)$ is 1. A node v with $skip(u) = 0$ does not have any children.

In Fig. 2(b), we show a signature graph for the signature file shown in Fig. 2(a).

s_1- 1011 0110
s_2- 1011 1001
s_3- 1010 0111
s_4- 0111 0110
s_5- 0111 0101
s_6- 0101 1100
s_7- 1110 0100
s_8- 1010 1011

(a)



(b)

In Fig. 2, each p_i represents a pointer to a s_i (i = 1, ..., 8).

In the following, we first discuss how a signature graph is constructed in 2.2. Then, we discuss how to use signature graphs to speed up the search of signature files in 2.3.

Construction of signature graphs

Below we give an algorithm to construct a signature graph for a signature file, which needs $O(N \cdot F)$ time, where N represents the number of signatures in the signature file and F is the length of a signature.

At the very beginning, the graph contains an initial node: a node v with $p(v)$ pointing to the first signature and $skip(v) = 0$.

Then, we take the next signature to be inserted into the graph. Let s be the next signature we wish to enter. We traverse the graph from the root and each encountered node will be marked. Let v be a node encountered and assume that $skip(v) = i$. If v is not marked and $i > 0$, check $s[i]$ and mark v. If $s[i] = 0$, we go left. Otherwise, we go right. If $i = 0$ or v is marked, we compare s with the signature s' pointed to by $p(v)$. s' can not be the same as s since in S there is no signature which is identical to anyone else. (If there are two identical signatures s_1 and s_2, we remove s_2 and associate the oids of s_1 and s_2 with s_1.) But several bits of s can be determined, which agree with s'. Assume that the first k bits of s agree with s'; but s differs from s' in the $(k + 1)$th position, where s has the digit b and s' has $1 - b$. We construct a new node u with $skip(u) = k + 1$ and $p(u)$ pointing to s. Let $w_1 \rightarrow w_2 \ldots \rightarrow w_j \rightarrow v$ be the accessed path. Then, make u the left child of w_j if v is a left child of w_j; otherwise, make u the right child of w_j. If $b = 1$, we make v be the left child of u and the right pointer of u pointing to itself. If $b = 0$, we make v be the right child of u and the left pointer of u pointing to itself.

The following is the formal description of the algorithm.

Algorithm *sig-graph-generation*(*file*)
begin
  construct a root node r with $skip(r) = 0$ and $p(v)$ pointing to s_1;
  for j = 2 to n **do**
    call insert(s_j);
  end

Procedure *insert*(*s*)
begin
1 *stack* ← *root*;
2 **while** *stack* not empty **do**
3 {v ← pop(*stack*);
4 **if** v is not marked and $skip(r) \neq 0$ **then**
5 {i ← $skip(v)$; mark v;
6 **if** $s[i] = 1$ **then**
7 {let a be the right child of v; push(*stack, a*);}
8 **else** {let a be the left child of v; push(*stack, a*);}}
9 **else** (*v is marked or $skip(v) = 0$.*)
10 {compare s with the signature s' pointed to by $p(v)$;
11 assume that the first k bits of s agree with s';
12 but s differs from s' in the $(k + 1)$th position;
13 let $w_1 \rightarrow w_2 \ldots \rightarrow w_j \rightarrow v$ be the accessed path;
14 generate a new node u with $skip(u) = k + 1$ and $p(u)$ ointing to s;
15 make u be a child of w_j;
16 **if** $s[k + 1] = 1$ **then**
17 make v be the left child of u and u be the right child of itself;
18 **else** make v be the right child of u and u be the left child of itself;}
19 }
end

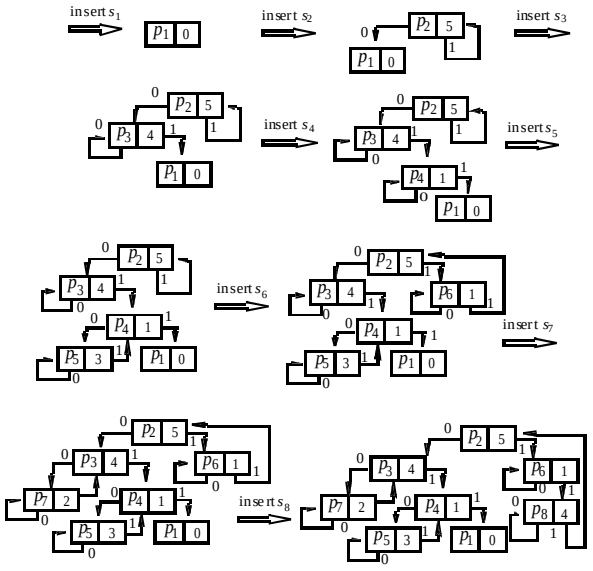
In the procedure *insert*( ), *stack* is a stack structure used to control the graph traversal.

In Fig.3, we trace the above algorithm against the signature file shown in Fig. 2(a).

Searching of signature graphs

In terms of the construction of signatures, the matching of signatures is a kind of 'inexact' matches. That is, for a signature s in S, any bit set to 1 in s_q, the corresponding bit in s is also set to 1, then we say, s matches s_q.

Fig. 3. Sample trace of signature graph construction



In the following, we first describe how to traverse a signature graph to find a signature in S which may be identical to s_q (exact matching). Then, we present an algorithm which is able to find all the signatures that may match s_q.

To find a signature in S that may be identical to s_q, see figure 3.

Algorithm *exact-matching*(G, s_q)

1. The search begins from the root.
2. Let v be the node encountered. Let $skip(v) = i$. If ith bit of s_q is 1, explore the right child of v; otherwise, explore the left child of v. v is marked.
3. The search ends up when a node v is encountered, which is marked or $skip(v) = 0$. In this case, compare s_q with the signature pointed to by $p(v)$.

In the following, we show the correctness of the Algorithm *exact-matching*( ). To do this, we introduce the concept of *signature identifiers*.

Consider a signature s_i of length m. We denote it as $s_i = s_i[1]s_i[2] \ldots s_i[m]$, where each $s_i[j] \in \{0, 1\}$ ($j = 1, \ldots, F$). We also use $s_i(j_1, \ldots, j_h)$ to denote a sequence of pairs w.r.t. s_i: $(j_1, s_i[j_1])(j_2, s_i[j_2]) \ldots (j_h, s_i[j_h])$, where $1 \le j_{k \le} m$ for $k \in \{1, \ldots, h\}$.

Definition 2 (*signature identifier*) Let $S = s_1.s_2 \ldots .s_n$ denote a signature file. Consider s_i $(1 \le i \le n)$. If there exists a sequence: $j_1, \ldots, j_h$ such that for any $k \neq i$ $(1 \le k \le n)$ we have $s_i(j_1, \ldots, j_h) \neq s_k(j_1, \ldots, j_h)$, then we say $s_i(j_1, \ldots, j_h)$ identifies the signature s_i or say $s_i(j_1, \ldots, j_h)$ is an identifier of s_i w.r.t. S.

For example, in Fig. 6(a), $s_8(5, 1, 4) = (5, 1)(1, 1)(4, 0)$ is an identifier of s_8 since for any $i \neq 8$ we have $s_i(5, 1, 4) \neq s_8(5, 1, 4)$. (For instance, $s_5(5, 1, 4) = (5, 0)(1, 0)(4, 1) \neq s_8(5, 1, 4)$, $s_2(5, 1, 4) = (5, 1)(1, 1)(4, 1) \neq s_8(5, 1, 4)$, and so on. Similarly, $s_1(5, 4, 1) = (5, 0)(4, 1)(1, 1)$ is an identifier for s_1 since any $i \neq 1$ we have $s_i(5, 4, 1) \neq s_1(5, 4, 1)$.)

Let $v_1 \rightarrow \ldots v_{k-1} \rightarrow v_k$ be the path explored. Let $skip(v_i) = j_i$ ($i = 1, \ldots, k$). Let s the signature pointed to by $p(v_k)$. Denote l_i the label for $v_{i-1} \rightarrow v_i$. Then, we have

$$s(j_2, \ldots, j_k) = s_q(j_2, \ldots, j_k) = (j_1, l_1)(j_2, j_2) \ldots (j_k\text{-}1, l_{k-1}).$$

But we don't have any other signature such that

$$s'(j_2, \ldots, j_k) = (j_1, l_1)(j_2, j_2) \ldots (j_{k-1}, l_{k-1}).$$

The path $v_1 \rightarrow \ldots v_{k-1} \rightarrow v_k$ is called the identifying path of $p(v_k)$.

Now we discuss how to search a signature graph to model the behavior of a signature file as a filter and to get all the signatures that may match s_q.

Denote $s_q(i)$ the ith position of s_q. During the traversal of a signature graph, the inexact matching can be done as follows:

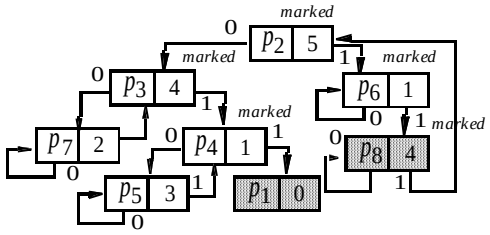
(i) Let v be the node encountered and $s_q(i)$ be the position to be checked.
(ii) If $s_q(i) = 1$, we move to the right child of v.
(iii) If $s_q(i) = 0$, both the right and left child of v will be visited.

In fact, this definition just corresponds to the signature matching criterion.

To implement this inexact matching strategy, we search the signature graph in a depth-first manner and maintain a stack structure $stack_p$ to control the graph traversal.

Algorithm *signature-graph-search*
input: a query signature s_q;
output: set of signatures which survive the checking;
1. $Set \leftarrow \varnothing$.
2. Push the root of the signature graph into $stack_p$.
3. If $stack_p$ is not empty, $v \leftarrow$ pop($stack_p$);
   else return(Set).

Fig. 4. Signature graph search



4. If v is not a marked node and $skip(v) \neq 0$,
   {$i \neg skip(v)$;
   mark v;
   If $s_q(i) = 0$, push c_r and c_l into $stack_p$; (where c_r and c_l are v's right and left child, respectively.) otherwise, push only c_r into $stack_p$;}
5. Compare s_q with the signature pointed by $p(v)$.
   (*$p(v)$ - pointer to a signature*)
   If s_q matches, $Set \leftarrow Set \cup \{p(v)\}$.
6. Go to (3).

The following example helps for illustrating the main idea of the algorithm.

Example 4 Consider the signature file and the signature graph shown in Fig. 2(a) once again.

Assume $s_q = 1011011$. Then, only part of the signature graph (marked with thick edges in Fig. 4) will be searched. On reaching a v that is marked or $skip(v) = 0$, the signature pointed to by this node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching since only 2 signatures (marked grey) need to be checked while a signature file scanning will check 8 signatures.

In general, if a signature file contains N signatures, the method discussed above requires only $O(N/2^l)$ comparisons in the worst case, where l represents the number of bits set in s_q and checked during the searching, since each bit set in s_q will prohibit half of a subgraph from being visited. Compared to the time complexity of the signature file scanning $O(N)$, it is a major benefit. We will discuss this issue in the next section in more detail.

MAINTENANCE OF SIGNATURE FILES

When a signature s is added to a signature file, the corresponding signature graph can be changed easily by running the algorithm *insert*( ) once with s as the input (see 2.2).

When a signature is removed from the signature file, we need to reconstruct the corresponding signature graph. To explain how to do this, we first establish a proposition.

Proposition 1. Let v be a node in a signature graph G. If v is the root or $skip(v) = 0$, the indegree of v is equal to 1 if $|G| > 1$, or 0 if $|G| = 1$; otherwise it is equal to 2.

Proof. We prove the proposition by induction on the number n of nodes in G.

Basis. When $n = 1$, G contains only a root r with $p(r) = 0$. The proposition holds. When $n = 1$, the proposition also holds.

Induction hypothesis. Assume that when $n = k$, the proposition holds. We consider the case when $n = k + 1$.

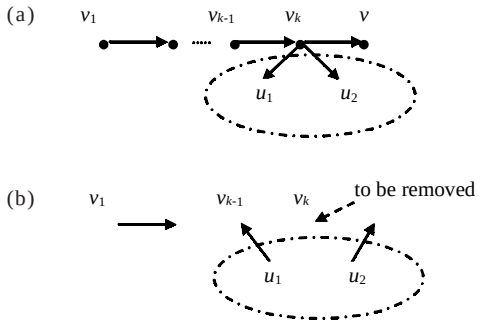
Let u be the last node inserted into G with $skip(u) = s$. Let $G' = G/\{u\}$. u is inserted into G' by comparing s with a signature s' pointed to by a $p(v)$ for some v in G. From lines 13 - 19 of Algorithm *insert*( ), we can see that v is not changed; and the indegree of u is equal to 2 since there is an edge from v's parent to u and another edge going from u to itself.

Proposition 2. Let v be a node in a signature graph G. If $skip(v) = 0$, the outdegree of v is equal to 0; otherwise it is equal to 2.

Proof. Similar to Proposition 1.

In terms of these two propositions, the deletion of a signature s from G can be done as follows.

Fig. 5 Illustration for deleting a signature



(i)Search G from the root until a node v is encountered, which is marked or $skip(v) = 0$. Compare $p(v)$ and s. If s matches $p(v)$ exactly, go to (ii); otherwise, nothing will be done.

(ii)Let $v_1 \rightarrow ... v_{k-1} \rightarrow v_k \rightarrow v$ be the path explored.

If $v_k \neq v$, replace $p(v)$ with $p(v_k)$. Let u_1 be another child of v_k (not on the path). Let u_2 be another parent of v_k (not on the path). Replace $v_{k-1} \rightarrow v_k$ with $v_k \rightarrow u_1$, and replace $v_k \rightarrow v$ with $u_2 \rightarrow v$. Remove v_k. Note that u_2 can be found by searching G from v_k with the target signature being $p(v_k)$. (See Fig. 5 for illustration.)

If $v_k = v$, replace $v_k \rightarrow v_k$ with $v_{k-1} \rightarrow u_1$. Remove v_k. (See Fig. 6. for illustration.)

In Fig. 6, to remove the signature pointed to by v, we remove v_k, a parent of v. It is because in v_k, $skip(v_k)$ now becomes useless. However, $p(v_k)$ must be remained. We do this by replacing $p(v)$ (in v) with $p(v_k)$, which will not change the signature identifier for $p(v_k)$. We have the proposition in figure 4.

Proposition 3. Let $P = v_1 \rightarrow ... v_{k-1} \rightarrow v_k \rightarrow \eth v$ be the path explored such that $p(v) = s$ (the signature found by (i)). (ii) will not change the signature identifier for $p(v_k)$.

Proof. If v is a node with $p(v) = 0$, then the path comprising $v_1 \rightarrow ... v_{k-1}$, $v_{k-1} \rightarrow u_1$, the path from u_1 to u_2, which is explored with the target signature being $p(v_k)$, and $u_1 \rightarrow v$, is the identifying path of $p(v_k)$. If v is marked (*i.e.*, visited for the second time), then there exists an i ($1 \leq i \leq k$) such that $v = v_i$. So the path comprising $v_1 \rightarrow ... v_{k-1}$, $v_{k-1} \rightarrow u_1$, the path from u_1 to u_2, which is explored with the target signature being $p(v_k)$, and $u_1 \rightarrow v_i$, is the identifying path of $p(v_k)$.

A similar analysis applies to the case of $v_k = v$.

CONCLUSION

In this paper, a new structure called a signature graph has been introduced, which can be used to speed up the search of signatures in a signature file. Given a query signature, we search the corresponding signature graph to find all the possible signatures that may match it. In this way, the sequential scanning of a whole signature file can be avoided, which improves the query processing efficiency significantly.

REFERENCES

AB97 H. Andre-Joesson and D. Badal, Using signature files for querying time-series data, in *Proc. 1st European Symp. on Principles of Data Mining and Knowledge Discovery*, 1997.

AHU74 Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Com., London, 1974.

BU77 R. Bayer and K. Unterrauer, "Prefix B-tree," *ACM Transaction on Database Systems*, 2(1), 1977, pp. 11-26.

Ca75 A. Cardenas, Analysis and performance of inverted data base structures, Commun. ACM 18, 5 (May), 1975, pp. 253-263.

CF84 S. Christodoulakis and C. Faloutsos, "Design consideration for a message file server," *IEEE Trans. Software Engineering*, 10(2) (1984) 201-210.

CR94 Crochemore, M. and Rytter, W., *Text Algorithms*. Oxford University Press, New York, 1994.

CS89 W.W. Chang, H.J. Schek, A signature access method for the STARBURST database system, in: *Proc. 19th VLDB Conf.*, 1989, pp. 145-153.

CTHP86 S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa and A. Pathria, Multimedia document presentation, information extraction and document formation in MINOS - A model and a system, *ACM Trans. Office Inform. Systems*, 4 (4), 1986, pp. 345-386.

CZ96 P. Ciaccia and P. Zezula, Declustering of key-based partitioned signature files, *ACM Trans. Database Systems*, 21 (3), 1996, pp. 295-338.

De86 U. Deppisch, S-tree: A Dynamic Balanced Signature Index for Office Retrieval, ACM SIGIR Conf., Sept. 1986, pp. 77-87.

DML98 D. Dervos, Y. Manolopulos and P. Linardis, "Comparison of signature file models with superimposed coding," *J. of Information Processing Letters* 65 (1998) 101 - 106.

Fa85 C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys*, 17(1), 1985, pp. 49-74.

Fa92 C. Faloutsos, "Signature Files," in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 44-65.

FLPS90 C. Faloutsos, R. Lee, C. Plaisant and B. Shneiderman, Incorporating string search in hypertext system: User interface and signature file design issues, *HyperMedia*, 2(3), 1990, pp. 183-200.

Go66 S.W. Golomb, Run-length encoding. *IEEE Trans. Inf. Theory IT-12*, 3 (July), 1966, pp. 399-401.

Ha81 R. Haskin, "Special purpose processors for text retrieval", *Database Eng.* 4, 1, 1981, pp. 16-29.

HFBL92 D. Harman, E. Fox, R. and Baeza-Yates, "Inverted Files," in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 28-43.

IKO93Y. Ishikawa, H. Kitagawa and N. Ohbo, Evaluation of signature files as set access facilities in OODBs, in *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C., May 1993, pp. 247-256.

Le92 D.L. Lee, Massive parallelism on the hybrid text-retrieval machine, *Inform. Process. Management*, 31 (6), 1992, pp. 281-289.

LL92 W. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," P*roc. ICIC'92 - 2nd Int. Conf. on Data and Knowledge Engineering: Theory and Application*, Hongkong, Dec. 1992, pp. 616-622.

MZ98 A. Moffat and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval", *ACM Transaction on Information Systems*, Vol. 14, No. 4, Oct. 1996, pp. 349-379.

SK86 C. Stanfill and B. Kahle, Parallel free-text search on connection machine system, *Comm. ACM*, 29 (12), 1986, pp. 1229-1239.

SKRT95 R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom. J. Zobel, Atlas: A nested relational database system for text application, *IEEE Trans. Knowledge and Data Engineering*, 7 (3), 1995, pp. 454-470.

YLK94 H.S. Yong, S. Lee and H.J. Kim, "Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases," *Proc. of 10th Int. Conf. on Data Engineering*, Houston, Texas, Feb. 1994, pp. 518-525.

ZMR98 J. Zobel, A. Moffat and K. Ramamohanarao, "Inverted Files Versus Signature Files for Text Indexing", *ACM Transaction on Database Systems*, Vol. 23, No. 4, Dec. 1998, pp. 453-490.