Data Base Engineering

Vol. 2 No. 2 Contents June 1978

A Quarterly Bulletin published by the IEEE Computer Society Technical Committee on Data Base Engineering

● RESEARCH PROJECT ABSTRACT ●†

1.	NAME OF PROJECT:	Knowledge Based Management System.
2.	ORGANIZATION(S):	Computer Science Department, Stanford University and SRI International. Stanford, CA 94305.
3.	PERSONNEL:	Professor Gio Wiederhold, CSD, Stanford University Dr. Daniel Sagalowicz, SRI International, Menlo Park, California 94025.
4.	KEYWORD(S):	Database Semantic Description, Conceptual Schema, Dynamic Access Structures, AI Technology.
5.	DESCRIPTION:	Project started April 1, 1978 and has as its focus to develop advanced methods of database description to make databases more useful. Existing natural language tools from Artificial Intelligence Research will be used to interface the system to the user [Sacerditu 77, Sagalowicz 77]. A high level semantic description will provide a model to validate update and analyze queries. A database structural model [Wiederhold 77] will support the semantic model, and provide guidance for implementation decisions. The techniques will be tested on a large database to access their appropriateness to real world database situations. Associated with these issues will be question of access structure management and data distribution.
6.	IMPLICATIONS:	We expect to gain an understanding about technology choices and contents for conceptual schemas, an evaluation of the required interfaces, and the power of existing AI technology to solve large scale problems. An important task will be to assure that only the database itself, but the data used for intelligent processing will be updateable.
7.	REFERENCES:	[Sacerdoti77a] E. D Sacerdoti, "Language Access to Distributed Data with Error Recover," SRI Artificial Intelligence Center Technical Note 140, February 1977. [Sagalowicz77] D. Sagalowicz, "IDA: An Intelligent Data Access Program," SRI Artificial Intelligence Center Technical Note 145, June 1977. [Wiederhold77a] G. Wiederhold, "Database Design," McGraw-Hill, 1977, Chapter 7. [Wiederhold78] G. Wiederhold, "Introducing Semantic Information into the Data Base Schema," CIPS Session 1978.

† For "Call for Abstracts in Data Base Engineering" and "Call for Abstracts of Reports, Working Papers and Theses on Data Base Engineering," see pages 10 and 12, respectively, in the March issue of Data Base Engineering Bulletin.

ACTIVITY-ORIENTED LIST

David S. Burris
University of Southern Mississippi
Hattiesburg, Mississippi

Kurt A. Schember
University of Texas at Arlington
Arlington, Texas

ABSTRACT:

In many applications, file references are not random, in fact a small percentage of the file may account for the majority of the files activity. Under these circumstances, a sequential search of a file ordered on frequency of reference may exhibit superior search statistics. The advantages of searching Activity-Oriented Lists (AOL) are explored and algorithms given for searching and maintaining AOL's as file reference patterns change. AOL's would enhance the flexibility and power of CODASYL SETS and other data-base products employing linked lists if included as an update and search option.

Introduction

A substantial amount of time in computing is spent searching list structures. An Activity-Oriented List (AOL) is one of a variety of data structures which, when used in an appropriate area, can substantially reduce both CPU and I/O overhead costs associated with list searches. An activity-oriented list is searched sequentially and ordered on frequency of reference from the most frequently accessed to least frequently accessed record. It may reside in main or auxiliary memory, and can be sequential or linked. The implementation must, however, allow reorganization of the list as frequencies or probabilities of reference change over time. Frequently referenced items will be located in a small number of probes, since they will be near the head of the list. Less frequently referenced items will require more probes, and a sequential search of the entire list is required to determine that a record is not present, an obvious drawback under some circumstances. Hence, activity-oriented lists are useful only if the majority of searches are to a relatively small subset of the file, and the desired record is present most of the time. Search time is frequently reduced for lists residing in auxiliary memory, since the number of physical records that must be read to obtain the desired logical record is often less than with other techniques. Records in AOLs may be inserted or deleted efficiently whether the list resides in main or auxiliary memory.

AOLs, in which most of the activity occurs in a small subset of records for relatively long periods of time, occur frequently. For example, compiler writers have noted that searches in compiler symbol tables are not random. References to small groups of keys tend to occur in clusters due to program logic. On a given day on the stock exchange, most of the activity will generally be restricted to a relatively small number of stocks. In an inventory system, as little as 20 percent of a file may account for as much as 80 percent of the search activity (1). Heising's "80-20" rule of thumb goes on to say that the same rule may be applied to the most active 20 percent of the file, i.e., 64 percent of the transactions deal with the most active 4 percent of the file, etc. Hence, a sequential search of a file ordered on frequency of use may yield better search statistics than techniques based on ordered lists or trees.

When an AOL is initially generated, it should be ordered in descending probability of reference. If the probabilities are not known a priori, then they can be approximated by maintaining a count field in each node of the list. The nodes can then be reordered periodically, based on the values of their count fields. File reorganization should be based on how often the frequencies of reference are expected to change. For example, a clothing store may only need to reorganize its files seasonally (see Figure 1).

The list can be made more responsive to changing probabilities by checking the count field during each reference. When it reaches a specified number, the record should be exchanged with the record preceding it in the list and the count field reinitialized. This allows records to move toward the front of the list as their activity increases.

When it is not desirable to maintain and check auxiliary count fields, the desired record may simply be interchanged with the preceding record after each successful search. Again, frequently used records will migrate ahead of those less frequently referenced. Note that the cost of maintaining the file must be paid after each successful search, but that the time is much less than that required to reorder a list based on a count field.

The previously mentioned methods of maintaining activity-oriented lists may not be satisfactory if the referenced subset of the list changes rapidly. Lists in which reference patterns change rapidly may be maintained by moving each successfully located record to the head of the list (4,6). This method can be efficient only for a linked list, since a substantial number of records would have to be moved in a sequentially allocated file to make room at the head of the list after each successful search.

Many commonly published (2,3) search algorithms, such as binary and Fibonacci for main memory and block and skip-chain searches for auxiliary memory, assume that the list is in key order. An attractive feature of these algorithms is that they place an upper bound on the number of probes required to locate a record, determine a record is not in the list, (2,3) or they exhibit a desirable expected number of probes to locate a record, under the assumption of equal probability of access. For example, a binary search has an upper bound of approximately $\log_2 N$ probes to locate a random record in a list of N records. For a block or skip-chain search, the expected number of probes is $\sqrt{N}$ when the block size or skip-chain length is $\sqrt{N}$ and all records have an equal probability of access (3). Search methods based on ordered lists

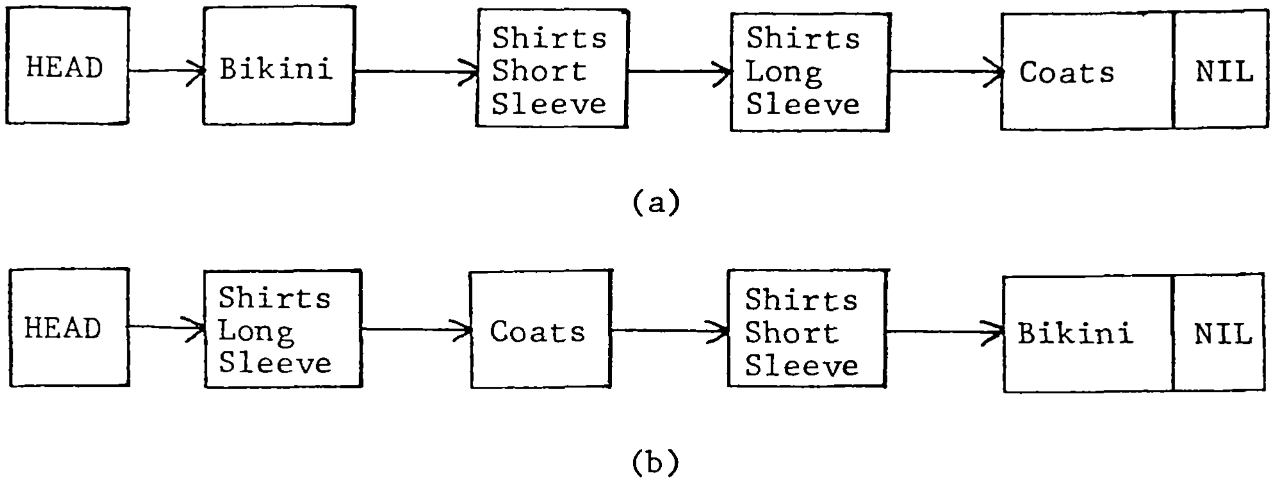

Figure 1: Inventory for a clothing store maintained as an AOL for (a) summer and (b) winter.

are frequently not optimal, however, if the file is subject to frequent record changes and/or if only a small percentage of the file accounts for most of the activity.

For sequential lists, new records can most efficiently be added at the end of the list. Linked insertions are most efficiently made at the front of the singly linked list or they may be easily inserted at the rear in circularly or doubly linked lists. If a list is ordered, it must first be searched to find where a new record should be inserted, then it must be rearranged to reflect the new order. For sequential lists, this implies that approximately half the records must be moved for an average insertion. For linked lists only the necessary pointers must be changed. A similar evaluation can be made for record deletion. While insertion of new records at the appropriate end of the list reduces overhead, the resulting list is no longer ordered and the search methods previously mentioned cannot be used. An alternative is to maintain a separate unordered list of insertions and execute a sequential search of the list. Periodically the ordered list may be updated from that insertion list (usually when the list is not available for other processing). Generally, records may be inserted more efficiently in an unordered list than an ordered list.

In many data processing applications, search activity is frequently restricted to a small subset of the total list for relatively long periods of time. Under these circumstances, the time required to locate a record in an unordered list using a sequential search can be competitive with, or superior to, search methods based on ordered lists. This is particularly true if the list is large and resides on auxiliary storage. During a sequential search, the key desired is compared to the key field of the current location in the list. If the keys don't match, the next logical record in the list is checked until either the desired record is found or the end of the list is reached. Binary and Fibonacci searches are not applicable to files in auxiliary storage due to the potentially large number of input/output operations, or in a virtual memory environment, where such a search may result in excessive page faults. While tree searches and block or skip-chain searches guarantee a reasonable expected number of probes, they do not give preferential treatment to frequently referenced records.

AOL Transaction Processing

The two broad categories of transaction processing addressed by AOLs are (A) list changes, such as additions and deletions; and (B) a record search followed by the possible updating of that record.

AOL Changes

Deleting a record from an AOL is similar to deleting a record from any other type of list. Once the determination to delete has been made, then the efficiency of such a process is a function of the storage medium and list organization (sequential vs. linked) and has little to do with the fact that it is an AOL.

Almost the same can be said for additions. They may be trivial or costly, but this is much more a question of organization and storage type than the fact that the list is an AOL. However, the application will determine, to a large degree, where the new record should be placed, since key order is not maintained.

Records should be inserted into the list based on their immediate probability of use. For example, if records have a high frequency of use immediately after insertion then they should generally be inserted at or near the front of the list amongst the most active records. As with deletions, random insertions or insertions at the front of the list can be most efficiently accomplished in a linked list. If there is no reason to suspect high activity after inserting a new record, then it can be inserted at the end of the list, which can be accomplished easily and efficiently in a sequential list. If insertions are to occur frequently at the rear of a linked AOL, then it may be desirable to maintain a pointer to the last record in the list, to aid in inserting new records. It may also be desirable to keep a count of insertions for lists which are only reordered periodically. The list can then be reordered before the normal reorganization period if the number of updates exceeds a threshold value that experience has shown impairs search efficiency.

Consider as an example, a new flavor (Imitation Bosenberry Mousse) being added to the current line of an ice cream store. An introductory offer is usually made, initially stimulating a high volume of business. Hence, the record for the new flavor should be inserted at the head of the AOL. If there are N flavors and M memory locations to hold the list ($N < M$), addition of a new flavor can be accomplished by storing the N older flavors in ascending order of frequency in list locations 1 to N. The new flavor would then be added in the N+1st location. Searching the list would proceed by probing at locations N+1, N, N-1, etc. until the desired item was either found or determined not to be in the list (see Figure 1). As long as space remains, it would be possible to efficiently make additions to the head of the list. The reverse of this strategy could be employed if new additions were expected to show very little initial activity. While additions can be made efficiently, deletions will almost always be relatively inefficient, since it is generally desirable to compress the storage representation of a sequentially allocated list after deleting an element.

AOL Searches

The efficiency of searching an AOL for a specific node is a function of list organization and storage type. However, two other factors that significantly influence search performance are the use of a count field and whether or not the probability of accessing a record is independent of previous searches. The questions related to dependency are discussed first, as considerations here are applicable to any list organization or storage medium. The question of count fields, however, is not. Both Rivest (5) and Knuth (2) note that the storage allocated to count fields could probably be put to use by other search methods in such a way as to improve overall performance, so that the need for AOL organization is frequently obviated for main memory searches. Hashing would be a strong alternative, as well as an explicit tree structure, particularly if reference probabilities are known a priori. Consequently, count fields may be more productively used with secondary storage. Examples of searching sequential linked activity oriented lists follow.

An algorithm for searching a sequential AOL for a particular value, say KEY, might appear as:

```
B:=TRUE;
FOR I:=N STEP -1 TO 1 WHILE (B=TRUE) DO
    BEGIN
    IF LIST (I) = KEY THEN
            BEGIN B:=FALSE;
                  CALL FOUND
            END;
    END;.
```

The routine FOUND would manipulate the record just located as desired.

Searching a linked AOL for the value KEY might be accomplished as (see Figure 2):

```
B:=TRUE
P:=HEAD
WHILE (P≠NIL) AND B DO
    IF P↑.INFO=KEY THEN B:=FALSE ELSE P:=P↑.LINK;
```

Note that the last node in the list has the value NIL for its link field to indicate that it is the last node, hence, the search will terminate properly if the desired item is not in the list. HEAD and P are pointer variables where HEAD points to the front of the activity-oriented list. P↑.LINK refers to the LINK field of the node pointed to by P.

Continual AOL Maintenance

Both methods described in this section update an AOL after every successful search. This allows the AOL to be more responsive to rapid changes than periodic reorganizational schemes.

Transposition

Using the transposition heuristic (Rivest) ($R_i \Leftrightarrow R_{i+1}$), a record is exchanged with the one in front of it following a successful search. Rivest has shown that this is the optimal search strategy, provided that the probability of searching for a given element is fixed, but independent of previously performed searches. A distinct advantage of this scheme over the move-to-the-front heuristic, is that the cost of reorganization after each search is essentially constant for either sequential or linked allocation. An algorithm to perform an AOL search with transposition on a linked list follows, assuming that a match will occur.

```
B:=TRUE;P:=HEAD;T1:=HEAD;T2:=HEAD;
WHILE (P≠NIL) AND B DO
    IF P↑.INFO=KEY THEN B:=FALSE
                   ELSE BEGIN T2:=T1;T1:=P;
                        P:=P↑.LINK END;
IF P≠HEAD THEN BEGIN
    T2↑.LINK:=P;
    T1↑.LINK:=P↑.LINK;
    P↑.LINK:=T1;
    IF T1=HEAD THEN HEAD:=P END;
```

Note that T1 is always one node behind P and T2 is always one node behind T1. Figure 3 shows an example, with the adjusted links as dotted lines. A special case exists if the desired record is the first or second record in the list. If the desired record is already at the head of the list then no exchange is required. If the desired record is the second physical record in the list, then it is necessary to set HEAD pointing to it after adjusting the link fields. This is accomplished by treating HEAD as the link field of a special node pointing to the first record in the list.

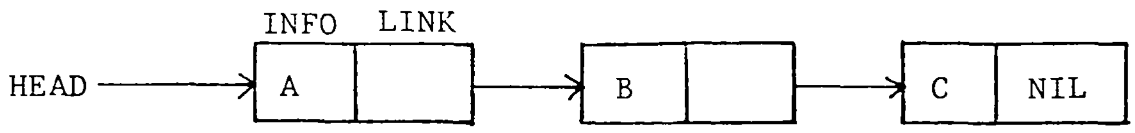

Figure 2.

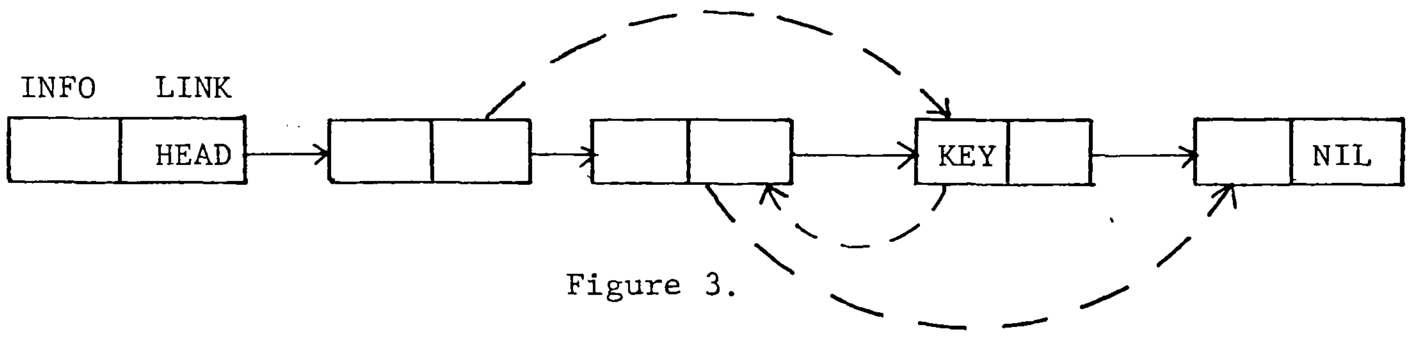

Figure 3.

Move To The Front

This search heuristic moves a record to the front of the AOL after each successful search (2,5). This action works reasonably well under the assumption of successive searches being independent of one another, but can be superior in an environment where one access implies that several more accesses to the same record are likely.

Implementing the move-to-the-front algorithm would incur about the same cost as transposition for a linked structure, but would be more costly to implement sequentially, since a portion of the file fron R_1 to R_{i-1}, where R_i is to become the first record, must be moved.

Periodic AOL Maintainence

Consideration is given here to methods for periodically reordering activity-oriented files based on count fields. If the file is sequentially allocated and resides entirely in main memory, any appropriate minimal memory sort such as shellsort, quicksort, or heapsort may be used to reorder the list based on the count fields. In a multilist system, where the entire file cannot reside in main memory at one time, several possibilities exist. It may be possible to extract the count fields from each record and store them in a sequential sort list in main memory along with pointers to the original records. If the resulting list fits in main memory, then an internal sort may be used and the

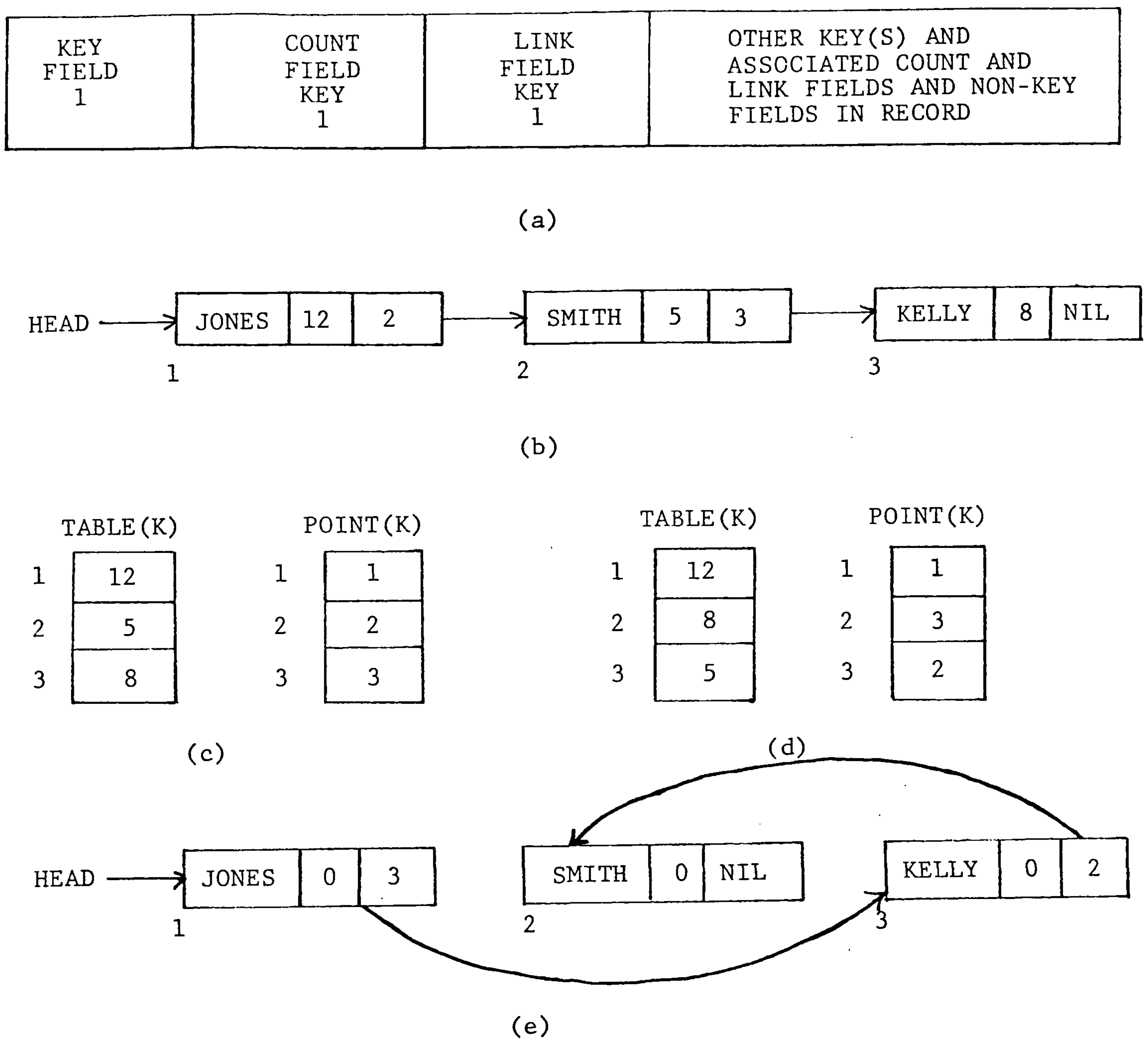

Figure 4: (a) general node format for a multilist file with several AOLs passing through each node (b) list before reorganization (c) TABLE and SORT arrays after Algorithm BST (d) TABLE and POINT after sorting (e) adjusted list after Algorithm AL.

pointers used to reorder the linked list. If the resulting list would exceed the available main memory, then the records themselves or the sort list may be sorted using an external sort or utility, with the pointers then used to reestablish the logical order of the list independently of any physical reordering. Note that any sublist passing through a multilist file may be reorganized independently of other sublists passing through the records. The following algorithm is given as an example for reordering a list based on the count field associated with the given key field in a multilist organization. For convenience it is assumed that a table can be generated with an entry for each record consisting of the value of the count field and pointer to the physical location of the record in auxiliary or main storage. It is further assumed that this table will fit in main memory. See Figure 4.

Algorithm BST (Build Sort Table)

```
BST1:  Set P←HEAD,K=0
BST2:  WHILE P≠NULL BEGIN
       K←K+1.
       TABLE(K)← KEY.COUNT(P)
       POINT(K)←P;
       P←LINK(P)
       END;
```

The algorithm for building the sort table assumes that the first record in the list is pointed to by HEAD. P is a pointer variable used to traverse the list and build the sequentially allocated sort table. The LINK field of the last record in the file contains a special value, NIL, to indicate that it is the last record in the file. The notation KEY.COUNT (P) refers to the count field associated with key KEY of the record indicated by P. Once generated, the sort table and associated table of pointers should be sorted into descending order on the count field. Now assuming a head of list variable, HEAD, the pointers in the file may be adjusted as follows:

<u>Algorithm AL</u> (adjust links)

```
AL1:  Set HEAD:=POINT(1);LINK(POINT(N)):=NIL;
          and KEY.COUNT(POINT(N)):=0;

AL2:  FOR I:=1 TO (N-1) BY 1
          BEGIN LINK(POINT(I)):=POINT(I+1);
                KEY.COUNT(POINT(I)):=0
          END;
```

After executing algorithm AL, the file will logically be reordered without physically moving any records or distributing other lists running through the same records. The count fields have been re-initialized for gathering frequency-of-use information for the next file reorganization.

Activity-oriented lists are particularly useful in data base applications where each record in the file has several sublists running through it. The sublists may be maintained in key order, as activity-oriented lists, or in any other order desired. Multilist file organizations allow records to appear in a different logical order on different lists independent of their physical order. Maintaining all or some of the sublists as activity-oriented lists may allow for economies in search time.

In large data base applications, the entire file frequently cannot be maintained in main memory or even high speed auxiliary memory. For example, a large manufacturing corporation may find it necessary to keep its inventory on tape files due to its size. However, most of the transactions during any given period would probably involve only a small portion of the list. Hence, when the inventory system is brought on-line, the most frequently accessed records can be kept in main memory and records with less activity in high speed direct access storage, and low activity records are allowed to remain on tape. Note that activity-oriented lists allow for the natural migration of records across the storage hierarchy based on their frequency of use.

Summary

In summary, activity-oriented lists allow for economies in the time required to search a file when a small portion of the file accounts for most of the activity. However, they are not the solution to every file searching problem. Unless search activity approximates Heising's "80-20 rule", or the list is very short, search efficiency degrades rapidly. This degradation is particularly noticable if all or part of the file resides in auxiliary memory and a large number of accesses to auxiliary memory are required to find the desired record. However, AOLs do give preferential treatment to highly active records in files where search activity can be restricted to a small subset of the file for relatively long periods of time. AOLs also exhibit the ability to maintain a high level of performance by adapting as the referenced subset of a file change.

References

1. W. P. Heising, "Note on Random Addressing Techniques", IBM Systems J., 2(1963), 112-116.
2. D. Knuth, Sorting and Searching, Addison Wesley, 1975.
3. J. Martin, Computer Data-Base Organization, Prentice-Hall, 1975.
4. J. McCabe, "On Serial Files With Relocatable Records", Operations Research, 13 (1965), 609-618.
5. R. Rivest, "On Self-Organizing Sequential Search Heuristics", CACM, 19(1976), 63-67.
6. G. Schay, Jr., and F. W. Dauer, "A Probabilistic Model of a Self-Organizing File System", SIAM J. Appl. Math., 15(1967), 874-888.

ACM-SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA

Y. Sagiv
Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

The ACM-SIGMOD International Conference on Management of Data was held in Austin, Texas, on May 31 through June 2. Approximately 250 people attended the conference, which for the first time provided parallel sessions: one session devoted to the presentation of papers, and the other to a panel discussion. This structure embodied a wide spectrum of interests ranging from practical issues to formal topics.

The papers discussed a variety of subjects and indicated the high quality of the program. Distributed databases were considered in a number of papers that covered such problems as concurrency, query processing, and integrated views. Several papers were devoted to languages, models and data dependencies that provide better tools for expressing data semantics. Other papers dealt with topics such as methods for improving the performance of databases (e.g. automatic design, performance evaluation of access paths, data structures, and query optimization), query processing and practical issues.

The panel discussions included sessions such as natural language interfaces, database machines, managing the data resources and more. The panels gave rise to stimulating discussions and contributed to the success of the program. The only problem with which some of the participants were faced was the wish to be in both parallel sessions at the same time. Unfortunately the proceedings of the conference (available from ACM) includes the papers, but does not have a detailed documentation of the panel discussions.

Finally, the local arrangements were excellent and earned the appreciation of everyone.

TC/DBE MEMBERSHIP APPLICATION/RENEWAL FORM

To become a member of the TC/DBE and be on the mailing list for the Data Base Engineering Bulletin, please return this form or a copy of it to:

IEEE TC/DBE
Department of Computer Science
University of Illinois, Urbana, IL 61801

NAME ______________________________
(please print)

INSTITUTION ______________________________

ADDRESS ______________________________

____

• ABSTRACTS OF REPORTS •[†]
WORKING PAPERS AND THESES

COMPILATION OF DATA BASE QUERIES

Randy H. Katz
Electronics Research Laboratory
University of California
Berkeley, CA 94720

This report describes the implementation of a compiler for data base queries. The notion of "levels of compilation" is introduced to classify existing query compilers and to propose a family of such compilers for the INGRES relational data base system. The implementation of one member of this family, a compiler which parses queries at compile-time, is described.

Comparisons of the time and space requirements for different levels of compilation are presented. A discussion of possible benefits and penalties of the compilation approach is presented. A description of the work required to implement a complete compilation oriented query processing system for INGRES is given.

This work was done in partial fulfillment of the degree of Master of Science. Research was supported by Army Research Office under Grant DAAG29-76-G-0245.

INTELLIGENT MAGNETIC BUBBLE MEMORIES

Mario Jino
Departamento de Engenharia Electrica/EL
Faculdade de Engenharia de Campinas-UNICAMP
C.P. 1170, 13100-Campinas, SP-BRAZIL

In this report, we are concerned with the design of intelligent magnetic bubble memories. It is our intent to explore ways of incorporating the novel bubble chip organizations and bubble movement operations in the design of such memories. In particular, we evaluate the performance of various file processing algorithms and memory organizations which can be achieved through their use. Retrieval times per word and per page are the parameters used to evaluate the different memory organizations. Performance of hierarchical memory systems using bubble memories is discussed. The unique features of magnetic bubble memories are used in designing algorithms for elementary file processing operations such as sorting, merging and clustering and for basic relational algebraic operations.

This work was done in partial fulfillment of the degree of Doctor of Philosophy at the University of Illinois, at Urbana-Champaign. It was published as a Technical Report UIUCDCS-R-78-900, Department of Computer Science, University of Illinois at Urbana-Champaign. Research was supported by the National Science Foundation under Grant MCS-77-27910 and by Universidade Estadual de Campinas and Fundacao de Amparo a Pesquisa do Estado de Sao Paulo.

Data Base Engineering Bulletin

A Quarterly Publication of the IEEE Computer Society
Technical Committee on Data Base Engineering

Data Base Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Data Base Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, data base architecture, data base machines, intelligent front ends, mass storage for very large data bases, distributed data base problems and techniques, data base software design and implementation, data base utilities, etc.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews and summaries, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Data Base Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in Data Base Engineering Technical Committee is open to IEEE Computer Society members, student members, and associate members. (Application form in this issue.)