error patterns and in discovering or at least correctly attributing improper activity, and this is a crucial deterrent capability.

System integrity refers to the proper functioning of the total system. In the case of hardware, this refers to the battery of error-prevention, error-detection, error-correction, and error-notification measures available today and under continuing development. In the case of software, this refers principally to freedom from surprises—the extent to which the software does correctly everything it is supposed to do, does nothing it is not supposed to do, and permits no unexpected behavior. Manufacturers are expending significant resources today in striving to enhance the integrity of computing equipment and of software.

Adequate physical security and thoughtfully developed and enforced operating procedures are most important aspects of data security. Achieving and maintaining appropriate protection from these elements of the security program is very much a matter of intelligent exposure analysis on the part of informed installation management, and the array of measures and techniques is well developed in the literature.

The Large Data Base File Structure Dilemma†

DAVID LEFKOVITZ

Moore School of Electrical Engineering,
University of Pennsylvania, Philadelphia, Pennsylvania 19174

Received December 13, 1974

This paper first presents a brief tutorial on the principal random file organization methods for handling two major applications—Transaction oriented systems and Information storage and retrieval systems. It then addresses a particular large data base dilemma, not satisfactorily resolved by any of these methods, and which is currently under active investigation. Two approaches to a solution are described. One is called the *hybrid inverted list;* the other is based upon an old technique called *super-imposed coding.* The former has been implemented and has recently been installed in an operational system. Some statistics related to file characteristics in this application are provided, but operational cost and performance statistics are not yet available.

Figure 1 classifies the principal types of information processing systems into two categories, *Transaction oriented systems* and *Information storage and retrieval systems.* Transaction systems are characterized by files that contain a diversity of record types, in which the records have relationships to one another, frequently referred to as Master-Detail relationships. These records may be arrayed in the same or different files. The structures that describe these record relationships are called trees, rings, and networks. Information storage and retrieval systems, on the other hand, tend to have records of similar type but which develop relationships among specific fields or components of these records called *keys.* The structures of files that support these various relationships are called Sequential, Indexed Sequential, Indexed Random, Mapped Random, Multi-list, and Inverted List. These two classes of file structures and the data processing that is characteristic of the respective systems present different requirements to the system designer. The data access requirements of the Transaction oriented system are considerably simpler than those of the Information storage and retrieval system, but accuracy with respect to the maintenance of data integrity is usually far more critical. The Information storage and retrieval systems, of which the bibliographic systems are one particular type, tend to deal with larger masses of diverse data types, and access to information tends to be more manifold; however, whether or not particular data items or records are or are not accessed for a given request tends to be less critical than in Transaction oriented systems.

†Presented in the "Conference on Large Data Bases," sponsored by the NAS/NRC Committee on Chemical Information, National Academy of Sciences, Washington, D. C., May 22-23, 1974. Supported in part by Contract NO1-CM-33719 from the Division of Cancer Treatment, NCI, NIH, DHEW.

Figure 2 graphically illustrates the tree, ring, and network file structures. The dots in the figure represent individual records. In the *tree,* a particular record may exhibit a master to detail relationship to one or more other records, and these other records may be in the same or a different file. Frequently, the detail records in a randomly organized file system will reside in a different file from that of the master records. The illustration shows a single master record at the top or root of the tree pointing to three detail records. One of these detail records is itself a master record and points to two other detail records. The two detail records at the third level will, in general, have a different record format and may themselves be contained in yet another file. The characteristic of the tree is that a given detail record may be pointed to by at most one master record. The *network* is a more general graphical structure, and no particular rules concerning the number of record relationships that can be established to or from a particular record exist. In the network, therefore, the master detail relationship may lose meaning. Any record may relate to any number of other records, and any number of records may relate to a particular record. The *ring* is a network in which all of the details emanating from a particular master are chained together, and the chain is closed on the original master. This forms "rings" of details hanging from a particular master, and a particular detail may itself, as a master, spawn its own "ring" of details, as shown in the illustration.

Figures 3 and 4 schematically illustrate the six basic file structures for keyed record relationships, characteristic of Information storage and retrieval systems. In Figure 3 the Sequential file is shown as a one-dimensional array of records. Access to each record is sequential. That is, in order to access, say, the third record, one would have to access

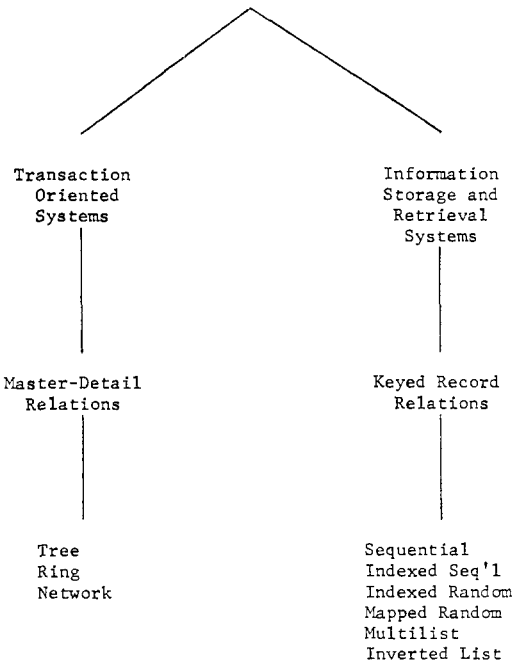

Figure 1. Classification of information processing systems.

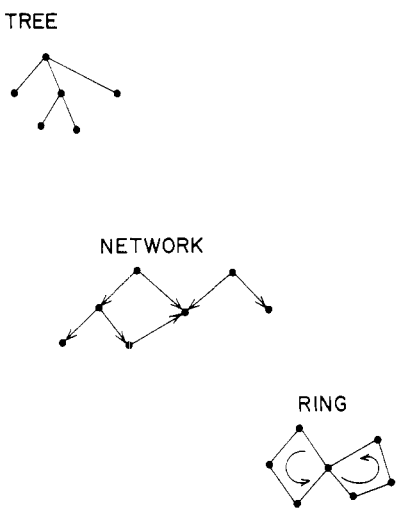

Figure 2. Structures for Master-Detail relations.

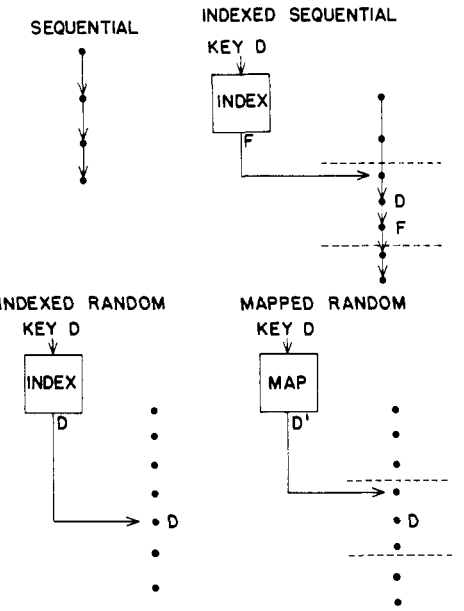

Figure 3. Structures for keyed record relations.

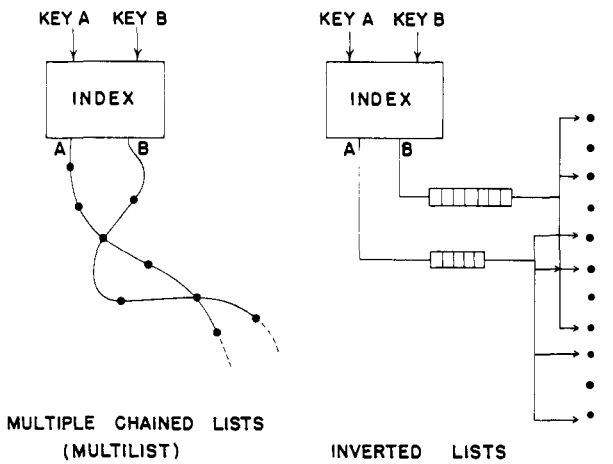

Figure 4. List structures.

the first and second before it. The Indexed Sequential file organization permits random access to a file, while retaining certain of the properties of a sequential file. An index, called a partial index, is constructed that will translate a value within a particular field of a record, called a *key value,* into a block address within the data file. The schematic shows three records within a block. The block is illustrated by two horizontal dashed lines and is shown to contain three records. The second record has a key called D and the third and last record of the block has a key called F. F, being the last record of the block, is the block identification entry in the partial index. If a look-up is made on key D, as shown in the diagram, it will translate through the partial index to the end of block key, F, which points to the beginning of the particular block, in which record F is located. The random access is, therefore, made to the beginning of this block and a sequential scan is made through the block looking for the record with key D.

The third type of file organization, called Indexed Random, drops the sequential nature of the file and stores all records at randomly located and arbitrarily chosen addresses throughout the storage medium. The index in this case is called a full index, since every record in the file must have an entry in it. Therefore, if key D is to be located, it will be decoded through the index to the record address of key D and will point uniquely to that record position in the file. This file cannot be accessed sequentially except by scanning the index; if the index is organized in a lexicographic order by keys, then it would be possible to produce the records in key sequence, but, in general, a random access of the file would be required for each record.

The fourth file organization is called Mapped Random. It is similar to Indexed Random in that the records themselves are distributed randomly throughout the file, but it is different in as much as they are not stored at arbitrarily selected locations. Rather, they are "mapped" into their locations by an algorithm that translates from the symbolic key into a numeric address. The map thus replaces the index as the look-up or addressing mechanism. These mappings are normally many-to-one in the sense that many different symbolic keys may map into the same numeric address. In recognition of this, an addressing element is normally selected as a block, or as is usually termed in the literature, a "bucket." The diagram shows a record with a key D as being located with two other records within a block. Key D is decoded through the map to a location called D', which is presumably the block address. Access is made to

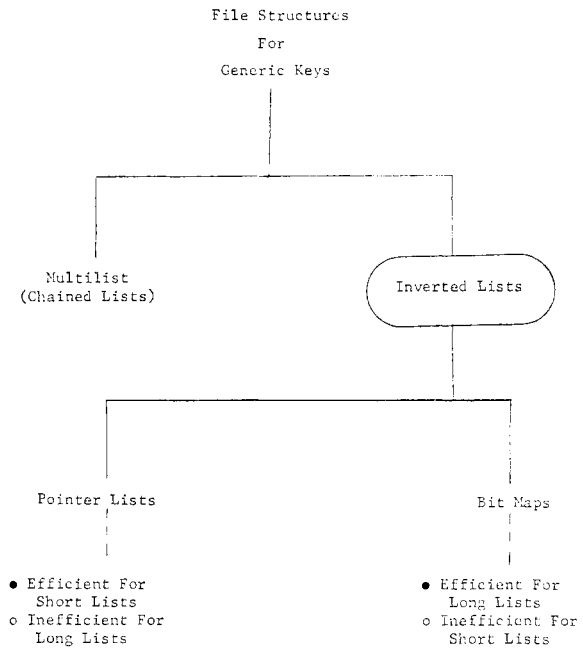

Figure 5. The IS&R large data base dilemma.

the beginning of the block, and the block is scanned sequentially to find the required record. Note that in the Mapped Random scheme it is impossible to retrieve the records sequentially, because neither the physical organization of the file nor the mapping recognizes any sequence to the keys themselves.

It is worthwhile at this point to discuss two properties or characteristics of keyed file structures. The first is that an individual record may be singly or multiply keyed. That is to say, a particular record of the file may contain a single key and always a single key, in which case it is said to be *singly* keyed, or it may contain more than a single key, in which case it is said to be *multiply* keyed. If a file can have multiply keyed records, then we say that it has multiple access points. The other property of a file is that a key may be *unique* or *generic*. If it is unique then there is one and only one record that contains that key. If it is generic then multiple records may contain that particular key. With respect to the four file structures just discussed, the Sequential file places no particular restrictions on a record or on a file with respect to either of these two characteristics, since the file is simply being scanned sequentially, and it would make no difference to the file structure as to whether a particular record contained one or more keys or whether one or more records contained a particular key. Only in the case where it might be desired to sort the Sequential file might it make a difference, in which case one would have to define multiple sort fields, if records contained multiple keys. With regard to the three random file structures, however, there is a difference, and, in fact, the reason for using one or the other of these file structures may relate directly to these particular characteristics of the given file.

The Indexed Sequential file may have generic keys but not multiply keyed records, the reason for this being that the file must be maintained in a particular key sequence and therefore the given record can only have one key, but any number of records could contain a particular key value. They would simply be repeated within the sequence. Certain implementations, though, of the Indexed Sequential method will also restrict the generic key characteristic for reasons of implementational simplification. The Indexed Random file structure, on the other hand, will permit multiply keyed records but not generic keys, while the Mapped Random file structure will, like the Indexed Sequential structure, permit generic but singly keyed records, because the record addresses are constrained by an algorithm, and hence can be based upon only one key within the record.

In order to allow file structures with both generic as well as multiply keyed characteristics to be constructed, a technique commonly called list-structuring is employed. Figure 4 illustrates the two common methods of list structuring. One is called the multiple-chained list or Multi-list[1] method; the other is called the Inverted List[2] method. The Multi-list organization is a variant on the Indexed Random structure, in which generic keys are introduced by assigning to each key field within the record another field called the *link-address*, which points to the next record in the file that contains the same key. In this way the full index points to the record in the file having the decoded key, and each record in the file then points to a successive record that contains the same key. The figure illustrates key A being decoded to list A, and key B being decoded to list B. Hence, a record may contain multiple keys, and multiple records can exist for the same key, and, as shown in the illustration, a particular record may lie at the intersection of two or more lists.

The Inverted List structure performs the same function but maps the data structure differently. In this case the linkages are removed from the records themselves and are stored in an auxiliary file called the Inverted List file. That is, the Inverted List technique identifies three separate file entities: the Index, the Inverted List, and the Data file. The Index points to the Inverted List, which is nothing more than a list of pointers to the records themselves.

If it is desired to perform Boolean logic on the keys in order to retrieve only those records that satisfy a specific logic specification, the access strategy for these two file organizations is somewhat different. In the case of Multi-list, the shortest list among a conjunction of terms is usually located (this is done by retaining the list length within the index for each key); the list is scanned and with each record read from the list the remainder of the Boolean logic applied. In the case of the Inverted List structure, the Boolean logic can be performed on the Inverted Lists themselves, without having to access any records in the Data file until the entire logical expression has been evaluated. This has advantages both in requiring fewer I/O operations as well as in being able to return a statistic that is sometimes useful to the requestor, particularly valuable in on-line systems, that will give him the opportunity to continue or to modify his search before actually accessing records from the Data file. There are certain simplicities in the programming of a Multi-list file both from the standpoint of file access and update, but these are normally overweighed by the efficiency of operation and the utility of the presearch statistic in the Inverted List file organization.

Figure 5, therefore, shows the two file structures for generic keys, with the Inverted List as the one normally selected for most large-scale operations. There are, however, two methods of implementation for the Inverted List file structure. One is by *pointer lists*, the other by *bit maps*. A pointer list is simply a list of the pointers to the records in the Data file, as illustrated in Figure 4. These pointers may either be direct disk file addresses or may be some other unique identification of the record, such as a unique key. In the latter case, the key itself would have to be subsequently translated to a unique address by either an index or a mapping technique. Such logical pointers, as they are sometimes called, are frequently used in order to decouple the data base itself from absolute addressing or positioning of data records in the file. This is desirable in many systems where records are moved around and repacked for storage and access efficiency. Such repacking or movement of the records will, therefore, not necessitate update or modification of the lists and indexes if the pointers are not themselves physical storage addresses. The slight loss in effi-

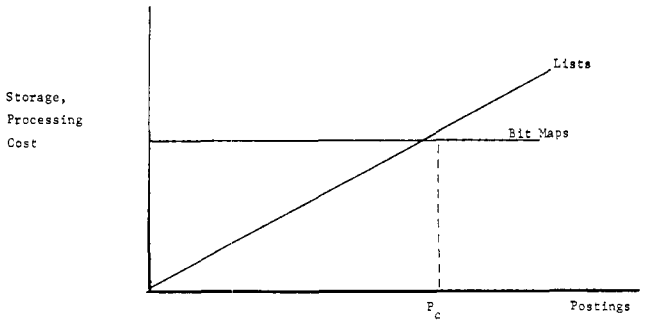

Figure 6. Inverted List characteristics—a graphic view of the dilemma.

- Hybrid Inverted List Processor
- Sequential Search - Superimposed Codes

Figure 7. Solutions to the dilemma.

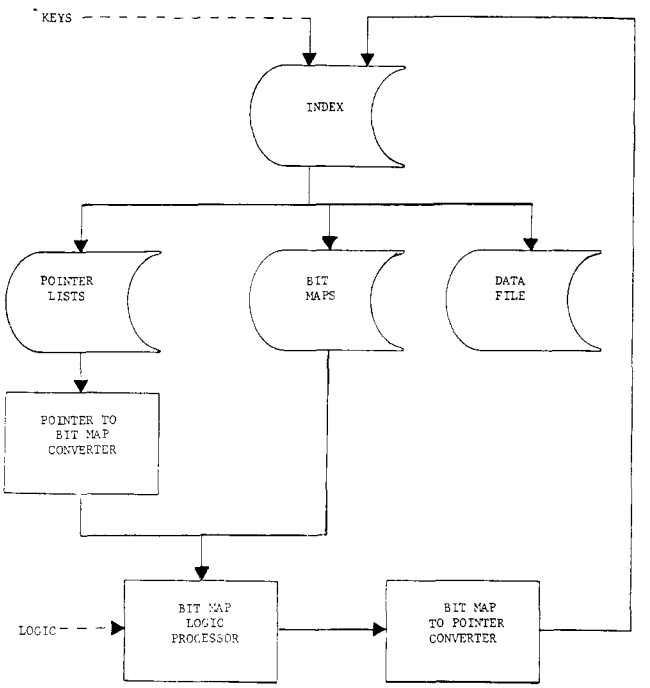

Figure 8. The Hybrid Inverted List Processor.

ciency at having to translate the pointers into physical addresses is normally compensated by the flexibility of record relocation.

The bit map approach is somewhat different. To each key is assigned a linear array of bits that contains as many bits as there are records in the file. The records themselves are then serial numbered, which can be viewed as the logical pointer of the record, as previously discussed. If the *i*th record in the file contains the particular key described by the bit mask, then the *i*th bit of the mask is set to 1; otherwise, it is set to 0. This has been called the "peek-a-boo" technique when implemented via cards with the bit locations arrayed in two dimensions, wherein a hole drilled through the card at a particular coordinate represents a logical 1. The bit array or mask can readily be subjected to Boolean logic by ANDing and ORing the entire masks. The results of such logical operations must then be decoded from bit positions back to pointer numbers and then translated to record addresses within the Data file.

These two implementations, the pointer lists and the bit maps, are, as indicated in Figure 5, complementary from the viewpoint of operational efficiency. The pointer lists are efficient for short lists but inefficient for long lists, while the bit maps are efficient for long lists and inefficient for short lists. Therein lies the dilemma for large-scale data bases, with hundreds of thousands and millions of records with generic keys.

Figure 6 graphically illustrates the dilemma. The characteristic curve for both storage and processing cost of the list *vs.* the number of postings in the list is shown in Figure 6. Both the storage and processing costs for the pointer list implementation will increase approximately linearly with the size of the list, while for the bit map method it is a constant. That is, the more entries on the list, the more it proportionately costs to store them and the more it proportionately costs to process them. The bit map, however, must dedicate the same number of bits per key (namely one bit per file record) regardless of whether the bit is 0 or 1, and logic operations must be performed on the entire bit mask regardless of the density of ones in the mask. One may employ certain compaction schemes that would alter the horizontal bit map curve, by replacing long strings of bit map zeroes by integers, but this just introduces further processing complexities, whose payoff would depend upon the actual density of ones in the file, or, to put it another way, the average number of postings per key. There will exist a crossover point, labeled P_c in the figure, below which it is less expensive to use lists and above which it is less expensive to use bit maps.

What is the solution? Figure 7 suggests two possible solutions. One is the *Hybrid Inverted List Processor;* the other is to revert to a sequential search and in particular to use a technique that is generically called *Superimposed Coding.* Each of these solutions is to be discussed in turn.

Figure 8 illustrates the Hybrid Inverted List Processor. There are four components of storage. One is the Index, which decodes both the symbolic keys as well as the logical serial number pointers. Then there are files corresponding to the Pointer lists, the Bit maps, and the Data file. That is, a system is conceived in which both the pointer lists and bit maps are used, and an attempt is made to optimize system performance and operation by the appropriate distribution of lists between pointers and bit maps based upon list length. Since the logic program is considerably different for pointer lists *vs.* bit maps one must choose one or the other to perform all of the logic processing. In this particular version, as is the case in the system that we are currently implementing, the logic is being performed via bit maps. Therefore, if a particular key resides in the system as a pointer list, which is to say that its list length is sufficiently short as to make pointer lists more efficient both from a storage as well as a processing standpoint, then the pointers in any list retrieved must be converted to a bit map before being fed to the Logic processor. The bit maps can be fed directly to the Logic processor, as shown in the illustration. Thus the list entries from both the pointer lists and the bit maps are fed to the bit map logic processor along with the logic of the retrieval specification. The result is a single bit map corresponding to the records that are to be retrieved. This bit map must then be converted to serial numbers which are fed back through the Index, which translates them into Data file addresses.

Figure 9 illustrates the optimization strategy that must be employed in allocating keys to bit maps *vs.* pointer lists in the hybrid system. Graph 9a shows storage cost *vs.* postings, as was previously illustrated. As shown immediately below it, in graph 9b, if one were to plot the storage cost *vs.* the crossover point, P_c, a curve with a minimum at the

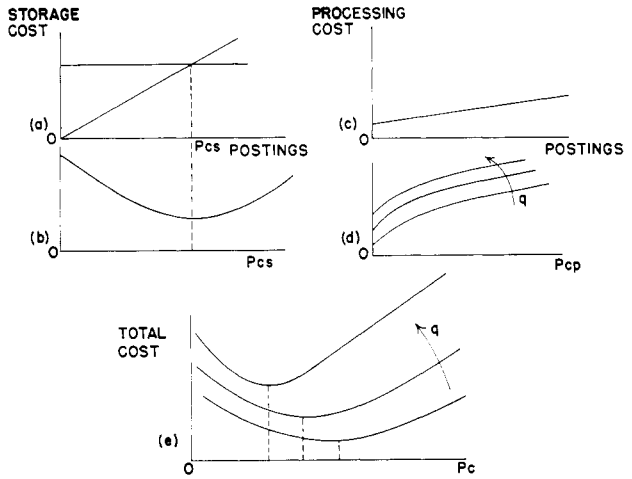

Figure 9. Storage-processing trade-offs.

Number of Records	=	213,061
Number of Keys	=	13,385
Number of Postings	=	5,884,109
Average Number of Keys/Record	=	28

No. of Postings (P)	No. of Keys With More Than P Postings	% of Total Postings
200K	1	4
150K	4	12
100K	9	24
50K	26	30
15K	81	50
5K	175	64
2K	328	70
1K	479	73

Figure 10. DR&D CIS inverted list statistics.

M = Number of Bits in Mask

p = Number of Bits in Key Code

$\binom{M}{p} \geq$ Total Number of Keys

Rule of Thumb:

No. of Keys/Record x p < ½M

Figure 11. Sequential search—the superimposed code.

point P_c in the upper curve would result. The processing cost is plotted in graph 9c for the hybrid system. There is a fixed minimum cost that must be sustained for the logic processing of the fixed length bit maps, and the cost increases linearly with postings when pointer lists are used. By varying the crossover point, as shown in graph 9d, in a hybrid system, a curve is obtained that begins with some fixed minimum cost, rises sharply, and then becomes linear with a positive slope. The reason for the sharp initial rise is that lists usually exhibit a Zipf characteristic and hence the preponderance of lists is short. A series of such curves will be produced, with increasing processing cost, as the number of questions processed over a given time period increases. This is an important factor, because it must be noted that processing costs will increase with the number of questions asked whereas storage costs will not. If one superimposes the two cost *vs.* P_c curves, Figure 9e results, where it is seen that the minimum will move downward in terms of P_c as the number of questions, or the processing, increases.

A chemical substructure search system for the Drug Research and Development Branch of the National Cancer Institute has been implemented with the Hybrid Inverted List technique because it severely faces the dilemma illustrated by Figure 5. A file of close to a quarter million compounds has been processed to produce the necessary search keys, and lists and bit maps have been generated. As of this writing, the system has been in operation for less than two months, so that operating costs are not yet available; however, the file characteristics, shown in Figure 10, plus formulas based upon the curves of Figure 9 indicate that the crossover from lists to bit maps should be somewhere between 1,000 and 12,000, which would require, as shown by Figure 10, approximately 80 to 500 bit maps. The system is designed so that the lists and bit maps are regenerated periodically and, hence, the crossover can be changed in response to system cost and performance or to changing file statistics. The system operation has been started at the low crossover end of 2000 postings producing around 400 bit maps. Even though insufficient data exist as of this writing to indicate whether the system is operating optimally, it is, nonetheless functioning satisfactorily. List processing response time to substructure searches normally ranges from a few seconds to several minutes depending upon system loading and query complexity, although under conditions of extreme loading combined with questions containing many "don't cares," an elapsed time of up to 45 minutes has been observed. This system is currently operating under VS 2 Release 1 on the IBM 370/168 with two other interactive applications and 15 batch queues running concurrently.

The second solution to the dilemma, suggested in Figure 7, is sequential search. This may appear to be a throwback, particularly within the framework of on-line systems, but it should be noted that the speed of the CPU combined with the extremely high data transfer rate from secondary storage of third and fourth generation computers has reached a point where one might profitably reexamine the potential of sequential processing. For large data bases it is still necessary to look into data compression, but with the combined approach of high CPU speed, high data transfer rate, and data compression, some mode of on-line search, and certainly an economic approach to batch processing, may be expected from a sequential search method. The technique that has been used to compress information for the purpose of sequential Boolean search is called superimposed coding. It has also been published under the name Zatocoding. A mask of M bits is constructed, and each key in the system's term or key vocabulary is encoded as a combination of p bits. Therefore, the total number of keys that can be accommodated by this superimposed code is the binomial coefficient $\binom{M}{p}$, as indicated in Figure 11. Each document or record in this system is thus encoded by superimposing the combinations of p bits for all keys within the record onto the M bit mask. If the total mask for a given record has too many ones, then there is likely to be a high number of false drops. For example, if a particular key has a code combination of positions 1 and 2, and another key has positions 2 and 3, then this record will have at least positions 1, 2, and 3 set. If there is some other key combination that corresponds to bit positions 1–3, and the key does not appear in this particular record, it would be considered a false drop. There is a rule of thumb that says that the number of keys per record times the number of bits in a key code should be less than one-half the mask. This will assure that less than half of the bits in the mask will be set and thus tend to reduce false drops. Using the DR&D file statistics (Figure 10), the size of the sequential search file would be minimized by choosing the smallest possible value of M such that

$$\binom{M}{p} \geq 14{,}000 \text{ and } 28p < \tfrac{1}{2}M,$$

since the number of keys in the DR&D system is close to 14,000, and the average number of keys per compound (record) is 28. The minimum value of M that will satisfy these two inequations is $M = 167$, where $p = 2$, which yields a key vocabulary of 14,000. However, the mask size of 167 turns out to be so close to the minimum value of M for $p = 3$, which is 168, that the optimal design values would actually be selected as

$$M = 168$$

$$p = 3$$

$$\binom{M}{p} = 2.46 \times 10^6$$

To get an idea of search file size in order to perform the key logic (exclusive of the connection tables or notations required to perform iterative search), assume that a record in the sequential file required 4 bytes for a compound identifier and 21 bytes for the 168 bit mask, yielding a fixed length record of 25 bytes. The total file size for 250,000 compounds would be 6.25 megabytes. A conservative estimate for the time to process the 168 bit mask would be 300 microseconds, which would yield a file search time of 75 CPU seconds. The time to transfer the search file from disk to core would be around 15 seconds, if head movement could be avoided; hence, the elapsed time would be at least 75 seconds, and in a normal multiprogramming mix might run 10 to 20 times as much. To this must be added the iterative (atom by atom) search time, and, since superimposed codes add their own coded false drops to those injected by the chemical screens, the amount of iterative searching would increase.

In conclusion, the superimposed code, employed in a sequential, highly compacted search file bears some further investigation even though at this moment it would appear as though, for interactive search of large files, it is not yet as efficient as the Hybrid Inverted list.

A particularly severe problem that the interactive system must face in the substructure search application is the "don't care." This condition is manifest as a series of OR's (disjunctions). A few "don't cares" each with several possible values can generate several hundred combinations, yielding an extensive logical sum of terms. The elapsed processing time in such a situation can be very long, as indicated above, particularly in a multiprogrammed operating environment. Such a system must be further optimized by a logic processor that will operate either upon list pointers or bit maps depending upon list length, and by retaining a bit map in core during the entire evaluation of a long sum of terms. This would be equivalent to retention of the top of the postfix logic processing stack in core rather than shuttling it back and forth to disk with each logic operation. This kind of optimization presents a time (CPU and channel) *vs.* space (core) trade-off. A file of a quarter million records must dedicate 32,000 bytes of core for the top of stack, while a file of 2.5 million must dedicate 320,000 bytes. Clearly, at some point some combination of zero compression, disk shuttling, and pointer list (as opposed to bit map) processing must be employed. These are questions and techniques for the future of such system developments to explore.

LITERATURE CITED

(1) Prywes, N. S., and Gray, H. J., *et al.*, "The Multi-List Type Associative Memory," Proceedings of Symposium on Gigacycle Computing Systems, AIEEE Publication, No. S-136, Jan 1962, pp 87–107.

(2) Lefkovitz, D., "File Structures for On-Line Systems," Spartan Books, New York, N. Y., 1969.

Large Data Base at the Lawrence Livermore Laboratory†

JOHN G. FLETCHER

Lawrence Livermore Laboratory, University of California, Livermore, California

Received December 2, 1974

The Lawrence Livermore Laboratory Octopus network includes a central storage facility of over 10^{12} bits. This facility is accessed through a directory mechanism which permits flexible organization and general sharing of files by many time-sharing users. The chief problems of the system are not unusual: the maintenance of high transfer rates, of reliability and availability, and of sufficient storage capacity.

The Lawrence Livermore Laboratory of the University of California (LLL) has one of the largest (if not the largest) concentrations of computing capability in the world. Processors include four CDC 7600's, one CDC 6600, two DEC PDP-10's, and numerous smaller machines; two CDC STAR-100's are on order. Input–output devices include two 15,000 lines/min Honeywell printers, two III FR-80 microfilm recorders, over 100 alphanumeric and raster display monitors, and over 600 teletypewriters and other interactive terminals. Data storage facilities include a terabit (10^{12} bit) IBM photodigital store, an IBM Data Cell, about 20 CDC 844 disk packs, and several other disks with transfer rates up to 40 MHz: there is also a vault containing over 30,000 magnetic tapes.

All these facilities (except the tape vault) are joined together into a single interconnected computer network called Octopus.[1] The organization of Octopus can be approximately characterized as a superposition of about eight *subnetworks*. Each subnetwork is centered on a computer called the *concentrator* of the subnetwork and performs a

† Presented in the "Conference on Large Data Bases," sponsored by the NAS/NRC Committee of Chemical Information, National Academy of Sciences, May 22–23, 1974. This work performed under the auspices of the U.S. Atomic Commission.