A SPECIALIZED COMPUTER ARCHITECTURE FOR TEXT RETRIEVAL

David C. Roberts

Central Intelligence Agency
Washington, D.C. 20505

Abstract

This paper describes a specialized computer architecture for text retrieval that provides a wide range of query capabilities, without the use of indexes of the material retrieved. A distributed approach is employed, with direct search processors. Each search processor is closely associated with one or more disk drives that store the data to be searched and each consists of a comparator for matching query terms, logic elements to combine query terms, a disk controller and a control minicomputer.

The key element of the architecture is the comparator, which can store over 64,000 characters of query terms and compare all of these terms simultaneously with data arriving from a disk continuously at more than one million bytes per second. Exact match, partical match, numeric comparison and context limitation are among the capabilities provided. The comparator is implemented as three similar universal finite-state automata, with special features to facilitate text retrieval and reduce hardware costs.

A breadboard version of the system with one searcher is operational and has undergone detailed evaluation. A prototype system with two searchers is being implemented and will be placed into regularly scheduled use.

INTRODUCTION

Requirements

This paper describes a specialized computer architecture for implementation of text retrieval systems. The system is intended for applications that serve moderate to very large user populations, with very large databases consisting primarily of text data, where the information requirements of users are varied and unpredictable.

Approach

The system employs the direct search approach, that is, the entire data base inits original form is searched for each batch of queries. User queries are entered into searchers, each of which holds up to 256 queries. Each searcher is associated with a part of the database, which it scans sequentially, comparing its part of the database with its batch of queries. Thus, no indexes to the data are employed, so that queries can include any words, partial words, or combinations of words that occur in the database; furthermore, new material can be added to the database as it arrives, without the overhead of index maintenance.

In order to provide for the implementation of systems of various capacities and performance requirements (such as response time, rate of query arrival and database size) a distributed architecture has been employed. Figure 1 shows a system configuration with three searchers for a database that occupies three disks. In operation, a

batch of queries is loaded simultaneously into all three searchers by the master control computer. Then the master issues a "start search" command, and each searcher begins a search of its part of the database, sequentially scanning its disk and comparing data as it is read with the batch of stored queries. Whenever a document in the database has been found to satisfy a query in the current batch, a hit report is sent to the master control computer. Depending on the application, such matches can be reported to users immediately, or accumulated and reported at the end of the search cycle.

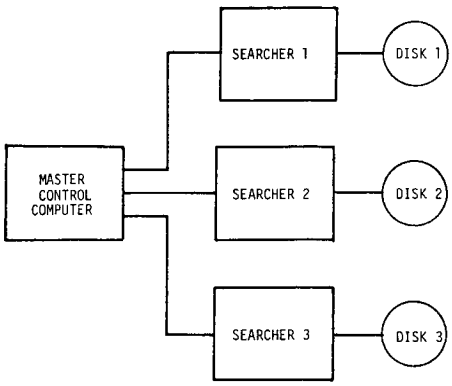


FIGURE 1. SYSTEM CONFIGURATION

The master control computer does not provide real-time control during a search; that control function is provided within each searcher. Rather, the master deals with the control of the overall system cycle, database maintenance, and communication of queries and results. Thus, master functions include such activities as query translation, database update, backup and recovery.

An important aspect of the searcher design is the partitioning of search functions. This approach enables special-purpose processors to be employed where they are clearly required, while still employing general-purpose hardware where it is most useful. The search process consists of two functions: term detection, the comparison of database text words with stored query words and partial words, and query resolution, which is evaluation of term match reports to determine whether they meet contextual requirements of queries. Each of these functions is performed by a separate specialized processor. These processors are interfaced to a small-scale general-purpose minicomputer, the search control computer, which furnishes data transfer, control and diagnostic services for the searcher.

Each searcher is comprised of four components. Figure 2 shows the functional elements of a searcher as connected to single disk storage device. The disk controller is a standard off-the-shelf item. The term detector is a special-purpose comparator, consisting of three universal finite state automata operating in a pipelined fashion, as described in detail below. The term detector stores query terms and query term phrases, and compares them with incoming data. It can store several thousand characters of query term data, and compare all of this data simultaneously with data transferred from disk at an average speed exceeding one million characters per second.

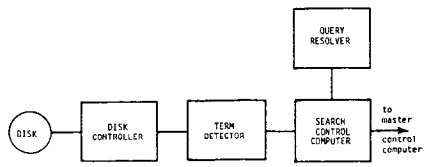


FIGURE 2. SEARCHER CONFIGURATION

Input to the term detector occurs at the maximum sequential transfer rate of the disk drive (a sustained rate up to one million bytes per second). The term detector forwards only reports of term and phrase matches to the query resolver, so the output data transfer rate from the term detector is typically two orders of magnitude lower than its input rate.

The query resolver receives term match reports, and determines whether those matches have occurred in combinations and context to satisfy queries in the current batch. The query resolver includes features for Boolean combination, word proximity, and various contextual restrictions such as sentence, paragraph and zone. The query resolver is implemented as a group of pipelined bipolar microprocessors, so that query resolution can keep up with the match report rate under conditions of extreme load.

Previous Work

Many earlier systems for document retrieval relied on the use of indexes, such as inverted files of various types, that are generated in advance of query arrival. Thus, when new documents arrive, indexes must be updated before those documents can be retrieved in response to queries. For very large databases, index update can be so complex that new information cannot be added immediately to the database.

The approach described here differs from such earlier work in that no indexes are used; the textual data itself, in its original form, is scanned completely for each batch of queries. This approach, called here the direct search approach, has been employed previously, implemented in software, hardware and firmware.

The COLTS system (1) is a software implementation of the direct search approach for the IBM System/370. For a single query, COLTS can search approximately 100,000 characters per second when running on a 370 model 158. COLTS is presently operational at several installations; because of its performance characteristics, it is best suited to small user populations and relatively small databases.

An early hardware direct search machine was the GESCAN developed by General Electric (2). This machine, which sold in 1969 for about $200,000, uses a special-purpose searcher with elements called "descriptors" that each emulate a single query character. Data is stored on magnetic tape, so search speed is limited to 120,000 characters per second, and only a single query at a time can be serviced. Although only a few of these machines were ever installed, they proved to be so useful that some are still in use. The searcher of the GESCAN machine performs both term detection and query resolution for a single query in its special-purpose comparator.

A later effort resulted in the Microtext system (3), which used microprogramming to customize a minicomputer for text searching. The machine was microprogrammed as a universal finite-state automaton, or fsa. When its memory was loaded with a representation of an fsa state graph, the minicomputer would emulate that fsa. A single fsa was used for term detection and query resolution. One of the findings of the Microtext project was that memory requirements for a single fsa for both term detection and query resolution were too great for any practical system.

A more immediate predecessor of the present system is the Associative File Processor (AFP) (4). This system partitions search functions into term detection and query resolution, and employs a single term detector to process all the terms from a batch of queries. The functional capabilities of the AFP are more limited than the present system. For example, the AFP can search directly only for exact matches of query terms, its maximum query capacity is smaller, and its query resolution capabilities are more limited.

Thus, several previous efforts have employed various technical aspects of the present system, although there has not been any reported use of a hardware implementation of a universal fsa, or the use of several universal fsa's in a pipelined configuration.

SYSTEM CONFIGURATION

The planned application for the text search system is for use with a host computer system which provides for user terminal interfaces. In this way, document search and retrieval can be integrated into other computer-based facilities made available to users. Figure 3 shows such a configuration.

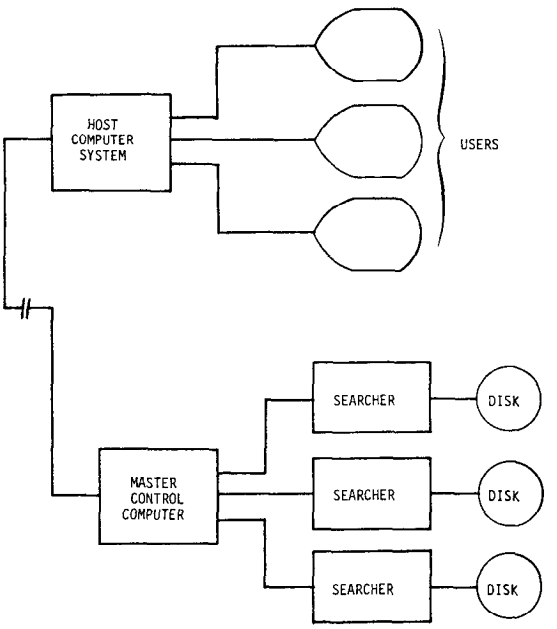


FIGURE 3. HOST AND TEXT SEARCH SYSTEM

The connection between the host system and the text search system can be made through a communication line or channel interface, depending on the speed requirements of the installation. This link is used to transfer queries, documents and commands to the text search system, which then responds with hit reports that give, for each query, the identities of all documents that satisfy it.

In most applications, the host system will have its own copy of the database to support delivery of documents to users. If query and response time requirements are such that any newly arrived queries can wait until retrievals are performed, then the text search system could perform both search and retrieval functions. However, the cost of a search unit is much greater than the cost of a disk; therefore,

whenever the query load is such that queries are often waiting, the use of a duplicate database stored by the host system appears to be desirable.

There are two determining factors for master control computer capacity, query translation and searcher load requirements. Query translation, the transformation of queries into state graphs for the term detector and tables for the query resolver, is very similar to program compilation. Because an entire query batch must be compiled at once, the memory and addressing capabilities for the master computer are determined primarily by the expected query batch size. As an example, a PDP 11/45 with 128K words of memory, with use of extended addressing features of RSX-11-M, is adequate for query batches averaging 10 queries of 15, 7-character terms.

The second determinant of master computer capacity is the need to load all the various tables in the searchers within a time interval that is short compared to the total system search cycle. As the number of searches in a system grows, this requirement influences the direct memory access (DMA) rate required of the master computer. A PDP 11/45 appears to have sufficient DMA capacity to load the searchers for a two-searcher system in less than fifteen seconds.

Configuration alternatives can be chosen to trade off response time, maximum database size, and maximum batch size. For example, if large-scale (300 million byte) disks are used, that take about five minutes to dump, then the average query will wait for 1 1/2 times five minutes, or about 7 1/2 minutes, to complete a search. Of course, hit information is available as soon as the first hit has been found, so that the average response time (until the first report) would be closer to 2 1/2 minutes. But if smaller disks are used (say 80 megabyte capacity) with the same transfer rate, then the system cycle time drops to about 1 1/2 minutes, with a possible average effective response time of 45 seconds. Of course, this response time improvement is gained at the cost of additional searchers, since the database would occupy nearly four times as many of the smaller-capacity disks.

SEARCHER DESIGN

The configuration of a single searcher is shown in Figure 4. The search control computer is a PDP 11/04; its UNIBUS is used to interface all the major searcher components. The bus switch is used so that the control computer can issue disk read commands, then disconnect the UNIBUS from the disk controller so that data entering the term detector does not tie up the UNIBUS.

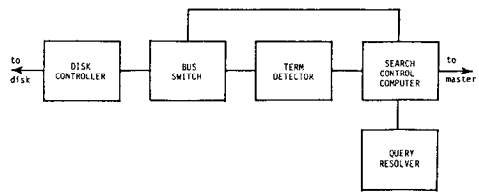


FIGURE 4. SEARCHER COMPONENTS

To start a search, the control computer loads term detector and query resolver memories, issues a disk read command, and opens the bus switch. Then data to be searched enters the term detector, and term match reports are forwarded directly to the query resolver via the UNIBUS. When processing of each document's hits has been completed, the control computer receives finished hit reports from the query resolver and forwards them to the master.

TERM DETECTOR

The term detector consists of a pipelined configuration of three universal finite state automata (fsa) that employ a memory organization and other special features so that sizeable query batches to be accommodated in reasonable amounts of memory. In the discussion below, the universal fsa is introduced first, followed by a detailed presentation of the specialized fsa design employed in the term detector.

Finite State Automaton

The finite state automaton is a simple conceptual model of a discrete sequential process, which has been applied to a number of problems in the computer field. An early use of fsa was in the study of computer design, particularly to minimize the number of components needed to construct a digital computer. This simple model of a computing machine has also been employed to model the elements of a computational process. An early theoretical result was the classification of formal languages based on the class of fsa needed to recognize the language.

A number of different formal definitions of an fsa have been developed. The definition presented below is based on one used by Gries (5) in connection with a discussion of lexical analyzers.

An fsa is defined here as a 5-tuple (A, S, M, B, F):

A is an alphabet called for input alphabet,
S is a set of elements called states,
M is a mapping from AxS into S,
B is a member of S, called the start state, and
F is a nonempty subset of S called the final states.

The fsa is initially in state B. With each arriving input character, mapping M is applied to cause a transition to another state. An input string that causes the fsa to enter one of the states that belongs to F is said to have been accepted by the fsa.

Mapping M for an fsa can be represented conveniently as a state graph. Figure 5 shows a state graph for an fsa that recognizes cat. Each state in set S is shown as a numbered circle; states in F are indicated as double circles. Mapping M is shown by arrows connecting states; for example, the arrow from state 1 to state 2 indicates that a c arriving while the fsa is in state 1 will cause it to enter state 2. In the figure * is used as a shorthand notation for transitions executed for all characters in the input alphabet other than those explicitly indicated.

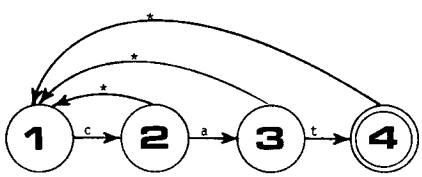


FIGURE 5. FSA TO RECOGNIZE CAT

The fsa of Figure 5 will recognize the string cat whenever it occurrs, even within a word. Figure 6 shows an fsa to recognize the word cat; the symbol indicates transitions that occur when a blank character is encountered.

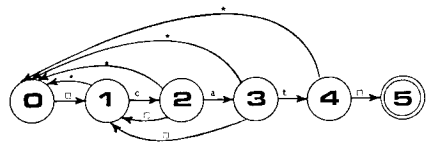


FIGURE 6. FSA TO RECOGNIZE THE WORD CAT

Now consider a query term including a don't care character. The simplest case is the fixed length don't care; suppose the original query term was @cat; that is, a search for any character followed by cat. This term would match acat, bcat, and other sequences of characters. A fixed-length don't care is handled easily by the use of * transitions, as shown in Figure 7.

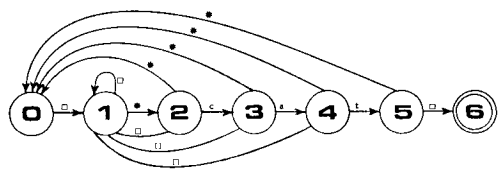


FIGURE 7. FSA TO RECOGNIZE @CAT

Variable-length don't care characters are inherently more difficult to process using the fsa approach, since the number of different sequences of characters that can match a single query term with a variable length don't care character is infinite. Consider the term ?cat, which would match any sequence of characters that ended with cat. This term would match all the four-letter sequences matched by @cat, above, plus five-letter sequences such as aacat, abcat, and so on, plus six-letter words such as aaacat, aabcat, etc.

Thus, during a search involving a variable-length don't care term, the term detector is looking for an occurrence of any of an infinite number of character sequences. This becomes particularly troublesome when some sequence of characters causes a failure of the search for one string, but may still be a successful match to another string. Consider the behavior of the fsa represented by the state graph of Figure 8 (for ?cat) when the sequence cacat is encountered. Here no additional states were required compared with Figure 5 but the * transitions, instead of going to state 0 to wait for start of a new word, continue the search by going to state 1.

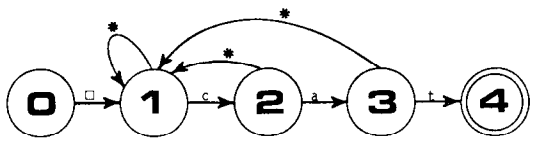
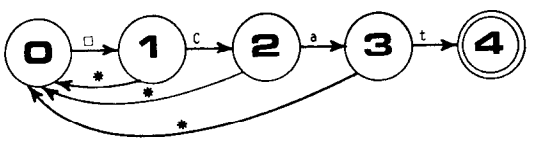


FIGURE 8. FSA TO RECOGNIZE :CAT

Thus, the processing of failure conditions differs for state graphs involving leading or imbeded vldc characters. In addition, study of the impact of vldc's on state graphs shows that as the number of terms detected by a given state graph grows, if that state graph includes imbedded vldc's then it will have many more states than a state graph for a collection of terms without imbedded vldc's.

In order to conserve memory space, as well as to permit the processing of * and transitions as defaults, the term detector contains separate, identical fsa's (called A and B), for term groups without and with imbedded vldc characters, respectively. There is also a third fsa, discussed later.



(A) FSA TO RECOGNIZE CAT?

	0	1	2	3	4
a	0	0	3	0	
b	0	0	0	0	
c	0	2	0	0	
t	0	0	0	4	
□	1	0	0	0	

(B) INDEXED REPRESENTATION

The fsa's employed utilize a specialized memory organization for representing state graphs that reduces the memory requirements. In earlier work with simulation of fsa's, generally two methods of representing state graphs in memory were employed: indexed states, for which a next state is stored for every possible input character, and list states, which store a list of input character-next state pairs for each state. The indexed representation is usually employed for states with many exiting transitions; the list representation for states with few exiting transitions.

Figure 9 shows memory structures for a software universal fsa to recognize cat?. To simplify the diagrams, the character set employed has been artificially restricted to a, b, c, t, and □. Of course, a practical character set would contain more symbols, including letters, numbers and other characters.

Figure 9(b) shows the indexed representation. For each state, one memory location for each possible input character is used to store the state to be entered when that input character is received. The representation of the input character is used as an index into the matrix of next states. Indexed representation permits very rapid operation (requiring only one memory access per state transition), at the penalty of inefficient use of memory when the character set is large and the matrix is sparsely occupied.

The list representation of Figure 9(c) is more efficient in its use of memory, but at the cost of slower execution speed, since a list search is required at each transition. For states that have many exiting transitions, the execution speed of the list representation can be prohibitive.

0	□	1	,	*	0
1	c	2	,	*	0
2	a	3	,	*	0
3	t	4	,	*	0
4					

(C) LIST REPRESENTATION

FIGURE 9. LIST AND INDEX MEMORY STRUCTURES FOR FSA

One approach for the implementation of an fsa for the present application would involve the use of indexed representation for states with many exiting transitions and list representation for states with few exiting transitions; that approach has been considered in some detail.

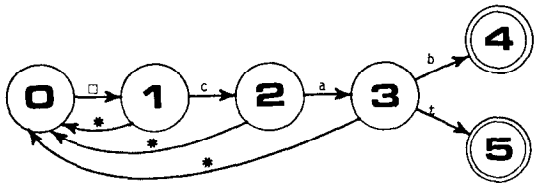
Rather than the memory organizations described above, a specialized memory organization has been developed for the term detector. This method, called Bird indexing (after R. Bird of Operating Systems, Inc., who suggested it), permits the representation of an indexed state using only one bit per character of the input alphabet.

Bird indexing is a form of indexed state. A single state word is used to represent each state; state words for

each state that can be reached in one transition from a given state must all be stored in adjacent memory locations. Each state word contains base and offset sections; the base is the memory address of the lowest-order state that can be reached from that state. All other states are reached by adding to the base an offset formed by using the offset part of the state word and the current input character.

A transition is executed as follows: the state word for the current state is held in a register for processing. When an input character arrives, the offset bit corresponding to that input character is checked; if it is set, then a new state number is formed. The number of offset bits set to the left of the character bit is counted, and added to the base part of the state word.

Figure 10 shows a Bird memory organization for an fsa to recognize cat and cab. In addition to the memory organization shown, a register called the default register stores a state number to be entered when an input character is encountered whose offset bit in the current state word is zero. The default register here would contain a zero; thus, in state 0, any input other than □ would cause the machine to remain in state 0.



	BASE	OFFSET a	b	c	t	□
0	1	0	0	0	0	1
1	2	0	0	1	0	0
2	3	1	0	0	0	0
3	4	0	1	0	1	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

FIGURE 10. BIRD MEMORY ORGANIZATION

Once a □ is encountered in state zero, then the number of ones in the offset portion of the state word to the left of the □ bit is counted (0) and added to the base part of the word (1+0=1). Thus, state 1 is entered when □ is encountered in state 0.

Now consider state 3; here either state 4 or 5 can be entered, depending on whether a b or t is encountered.

In addition to states like those above, called index states, there are three other state types: sequential states, jump states and retry states.

Sequential states are employed to save memory for portions of a state graph where there is no further branching. Figure 11 shows such a graph, following the branch at state 4. In that graph, once state 5 has been entered there are only two possible outcomes: the entire word catapult will be found, or a failure has occurred. Thus, all the alternatives available from index states are not needed for states 5 through 11. For sequential states, the characters being matched are packed into both parts of a state word, and the fsa compares input characters with the stored characters. Either the default state is entered for a failure, or a success is reported.

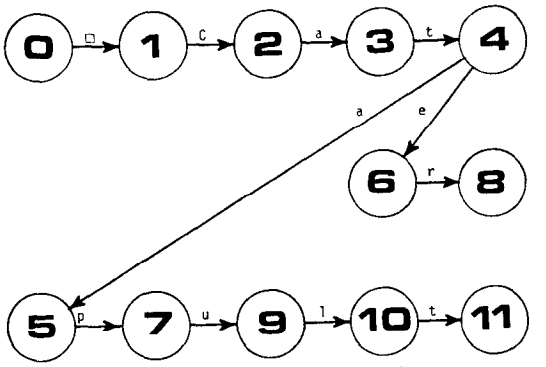


FIGURE 11. STATE GRAPH WITH SEQUENTIAL STATES

Jump states are used for transitions to a state that is not stored adjacent to the existing state. A jump state is equivalent to an indirect jump; that is, when a jump state is entered, the fsa immediately enters the state indicated in the base part of the state word, without further input. Jump states are useful primarily in certain complex cases involving imbedded vldc characters.

The retry state permits a jump to another state, with the character in the input register processed a second time for that state. Retry states provide the fsa with the power of a pushdown stack of depth one; they are used to reduce the number of states for complex state graphs involving imbedded vldc's.

Pipelining

A pipelined configuration of fsa's (Figure 12) is employed to save memory and to reduce the number of reports sent to the query resolver. A comparison of Figures 5 and 8 shows that the state graphs for terms involving leading (and imbedded) vldc characters differ from those without imbedded vldc's with respect to their handling of default and space transitions. If both types of the state graph were combined in a single fsa, constant changing of the registers used for these "implied" transitions would be required. In addition, state graphs involving imbedded vldc's tend to be much more complex than those without vldc's. For these two reasons, separate fsa's are used for state graphs for terms including imbedded vldc's (fsa B) and without imbedded vldc's (fsa A). Input is presented in parallel to fsa's A and B; each searches for different terms.

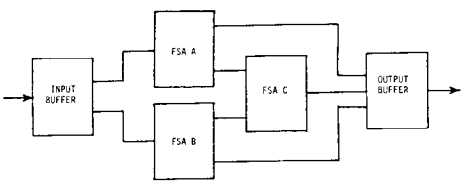


FIGURE 12. TERM DETECTOR CONFIGURATION

A third fsa (C) serves to reduce the volume of reporting to the query resolver, by detecting contiguous word phrases (cwp's). Outputs from A and B are addressed directly to the term detector output buffer (if they are present in queries as single-word terms), to fsa C (if they participate in cwp's), or both. FSA C, then, receives as input term hits from fsa's A and B, for terms that participate in cwp's and reports detection of complete cwp's to the query resolver.

As an example of the operation of fsa C, consider a search for weekly summaries not identified by any header information. In one example, such documents could be identified only by searching for occurrences of the cwp the week of. If the terms in this cwp were reported separately to the query resolver, then every occurrence of the and of in the database being searched would require processing by the query resolver. But by using fsa C to process cwp's, then only complete occurrences of the cwp are processed by the query resolver. This approach is useful *because very high frequency words such* as the, of and and are used in queries almost always as parts of cwp's and almost never by themselves.

Word Size

For an 8-bit character set, using Bird indexing each state word would be perhaps 300 bits in length. If the transitions were sparse, memory would be very inefficiently used. For this reason, the fsa's do not process characters; rather, they process half-characters, called nibbles. Using 4-bit nibbles, only 16 offset bits are required; the word length, including state type indicators, in only 45 bits. A switch-selectable capability to process 3-bit or 4-bit nibbles permits the term detector to process either 6-bit or 8-bit characters.

Additional Features

A number of additional features have been provided. A document (that is, the unit of data to which a single query applies) can be divided into up to 64 zones, and each of these into up to 64 subzones. Zone markers are automatically recognized by the term detector and stored in a register; the state graph for a query term can include a branch on the contents of the zone and subzone registers. Thus, a single "document" can consist of formatted and unformatted portions, all stored together, all processed by means of a single query.

Zone markers are intended to indicate type of information. Other markers, called sentence and paragraph, have been provided as contextual limits for querying. These markers are also recognized by the term detector, but they are counted, and current counts reported to the query resolver with each match report. Thus, a query or part of a query can be restricted so that it must be satisfied within a single paragraph or sentence of a document. Of course, use of such delimiters presumes the ability to recognize contextual boundaries when documents are entered into the system, so that markers can be inserted into the data in advance of search.

In addition to exact matches, state graphs can be constructed to perform numeric comparisons. Thus, queries can include numeric renges as well as complete and partial word matches.

To illustrate the use of these special features, if dates are represented in a standard code (say, Julian date), and stored in their own zone, then a query could request documents containing various combinations of words within the same paragraph and a mention of any date between two given dates.

CURRENT STATUS

A breadboard term detector has been in operation since January of 1978. Tests indicate that it can keep up with input data arriving at about 2 million characters per second, and can store up to 64,000 characters of term data in fsa A and 4,300 characters in fsa B. A software query resolver is now operational; a breadboard hardware query resolver is being built. An operational prototype two-searcher system is planned for installation in early 1979. Experience indicates that the cost to implement a term detector is reasonable; it is comparable to the price of a 300 MB disk drive with controller.

CONCLUSIONS

The work described here has shown the feasibility of employing universal finite-state automata for the implementation of large document retrieval systems that store and retrieve on the basis of full text rather than abstracts.

This work also indicates the possibility of processing text data and formatted data in a compatible fashion in the same system. In addition, the index-free operation of this system permits essentially on-line update, in that additions can be made to a database in the interval between searches.

BIBLIOGRAPHY

1. COLTS II: CRW On-Line Text Search User's Manual. Chase, Rosen and Wallace, Inc., Alexandria, Va., 1976.

2. Operator's Manual for GESCAN. General Electric Company, Apollo Systems, Space Division, Daytona Beach, Florida, 1970.

3. Bullen, R. H., Jr. and Millen, J. K. Microtext - The Design of a Micro programmed Finite State Search Machine for Full-Text Retrieval. AFIPS Conference Proceedings, Fall Joint Computer Conference, 1972, pp. 479-488.

4. Kadonaga, R. S., Associative File Processor Systems and Programmer's Manual. Operating Systems, Inc., Woodland Hills, Calif., 1978.

5. Gries, D. Compiler Construction for Digital Computers. Wiley, New York, 1971.

6. Roberts, D.C. (ed.) A Computer System for Text Retrieval: Design Concept Development. Central Intelligence Agency, Washington, D.C., 1977, Report No. RD-77-10011.