Information Technology: Research and Development (1983), *2* (155–168)

THE UTAH TEXT RETRIEVAL PROJECT*

L. A. HOLLAAR

Department of Computer Science, University of Utah, Salt Lake City, UT 84112, USA

(*Received 25 April 1983, revised 20 July 1983*)

ABSTRACT

The Utah Text Retrieval Project seeks well-engineered solutions to the implementation of large (over 50×10^9 characters), inexpensive (less than a dollar a query), rapid (average response time of 10 seconds) text information retrieval systems. It was established in 1980 in the Department of Computer Science at the University of Utah, and is an outgrowth of a similar project at the University of Illinois with which the author was associated.

At the present time, the project has three major components. Perhaps the best known is the work on the specialized processors, particularly search engines, necessary to achieve the desired performance and cost. The other two concern the user interface to the system and the system's internal structure. The work on user interface development is not only concentrating on the syntax and semantics of the query language, but also on the overall environment the system presents to the user. Environmental enhancements include convenient ways to 'browse' through retrieved documents, access to other information retrieval systems through gateways supporting a common command interface, and interfaces to word processing systems. The system's internal structure is based on a high-level data communications protocol linking the user interface, index processor, search processor, and other system modules. This allows them to be easily distributed in a multi- or specialized-processor configuration. It also allows new modules, such as a knowledge-based query reformulator, to be added.

1. INTRODUCTION

The advent of low-cost mass storage is increasing the interest in information retrieval from very large text databases. For example, in the mid-1960s, an IBM 2311 disk drive stored 7.25 million characters for approximately 0.4 cents per bit. Now personal computers contain larger disk drives, and drives smaller than a third of a cubic foot and holding more than 250 megabytes at a cost of about 0.0015 cents per bit are available. This savings in cost of over 250 times has been accompanied by bandwidth (number of bits transferred per second) increases of well over an order of magnitude.

This improvement in mass storage efficiency has been matched by more information becoming available in machine-understandable form due to the widespread use of word processing systems and computerized typesetting. Older

*This work is supported in part by the National Science Foundation under Grant MCS–8021116.

0144-817X/83/03 0155-14 $03.00

information can be added to a database using optical scanners. This increase in available information has occurred at the same time that fast, accurate retrieval has become more important, particularly in areas like science, management, and the law.

Text information retrieval systems can exist in many forms: large information utilities, electronic file cabinets, looseleaf services, and selective dissemination of information (SDI) or message routing systems (Hollaar *et al.*, 1983). The first three are similar, differing primarily in size, location of the database, and primary supplier of the information. The large information utilities are what generally comes to mind when the term 'information retrieval system' is used, and a number of systems storing tens of gigabytes of data. However, while examples of the various types of text information retrieval systems currently exist (and are rather successful), they are not as widespread as is desirable. Because of high costs and arcane, complex user interfaces, access is often limited to professional searchers, generally in libraries.

However, for information retrieval systems to achieve their maximum potential, they must be accessed by the ultimate users of the information, since these users know much better what information is desired to solve a particular problem. They also are the only people who can directly combine the retrieved information with other material to produce the necessary report or document. This means that the cost should be reasonable and, more importantly, the response time should be excellent. For many users of information, response times of a minute or more are intolerable.

1.1 *Differences from database management systems*

These text information retrieval systems differ substantially from the more common database management systems, even though both find information based on a user- or software-supplied query. The databases necessary for a useful text retrieval system are orders of magnitude larger than the more structured database management systems. While five million bytes would be a reasonably large inventory management or accounting database, it is fewer characters than in a single conference proceedings. The operations performed by the system users also differ. While the users of a database management system are allowed to add, delete, or modify information, for many text retrieval systems only the system management is allowed to modify the database.

Finally, the nature of the query for each system differs substantially, even though they appear similar at first glance. Both specify search terms connected with Boolean operators (such as AND, OR, and AND NOT). Both can specify that a term must appear in a particular context. For database management systems, this means the term or value appears in a particular field, while for less structured text, it might mean appearing in some logical context, such as a sentence, or in a specified proximity to another term.

Because the information in the text retrieval database is less structured than for a database management system, because the authors of different documents may have used different terms or phrases to describe the same concept, and because words may have inconsistent spelling ('color' and 'colour') or different meanings depending on their context ('will' and 'retort'), the simple, exact query of a database management system may not yield the desired result. Often many synonyms for the original search terms must also be used, either explicitly specified by the user or generated by a special system command or a thesaurus. The use of don't-care tokens in terms, which match either a single character, up to a specified number of characters, or an arbitrary number of characters, can be used to find all words with a common root or to handle spelling variants.

1.2 *Problems with conventional implementations*

The extremely large databases and complex query terms necessary for text information utilities make them difficult to implement efficiently using standard digital computers, although most major systems currently use large, conventional systems. Even if a conventional processor searches at the rate of one character per microsecond (the raw delivery rate of a disk drive), it would take over 8 hours to search a 30 gigabyte database. If the queries are complex, containing multiple terms or don't-care tokens, many of the fast search algorithms cannot be used, and the search times may increase by an order of magnitude. One study indicated that search times of 100 000 bytes per second were common for actual queries (Roberts, 1978). To complete a search of a 30 gigabyte database in a minute would require data input bandwidth of 500 megabytes per second, one to two orders of magnitude greater than available in large systems.

The common solution to this problem is not to exhaustively search the database, but to examine some surrogate to determine which documents satisfy the query. The use of an index or inverted file, however, increases the amount of storage the system requires, often more than doubling an already large number of disk drives (Bird *et al.*, 1978). Extra processing is required to handle each query, and additions to or deletions from the database require the restructuring of the index. Finally, if the point of inversion (document, sentence, individual word) does not match that specified in the query, additional processing is required. If the query specifies a lower context, such as a phrase when the index only indicates documents, a search must be performed, while if the index is to the word level, index entries must be intelligently combined to determine if two words are in the same sentence or paragraph.

2. PROJECT OVERVIEW

The goal of the Utah Text Retrieval Project is to produce systems capable of handling a variety of database sizes, ranging from hundreds of millions of characters for an electronic file cabinet in an office automation system to the 65 gigabytes required to handle United States Patents for the last thirty years (US Patent Office, 1983). Such systems should offer good response time and low cost. A rough calculation based on the data from the Patent Office shows that, by using specialized backend processors and clever indexing, response times of under ten seconds and costs of less than a dollar per query are possible (Hollaar *et al.*, 1983).

The project is an outgrowth of the EUREKA information retrieval project at the University of Illinois at Urbana-Champaign, initiated by Professor David Kuck. This project, started in the late 1960s, examined the use of specialized processors, particularly for handling index manipulations in an inverted file system (Stellhorn, 1977; Hollaar, 1978), and the effects of different commands on system and user performance (Rinewalt, 1976). A small testbed information retrieval system was also developed (Burket and Emrath, 1979).

The present project was started in 1980 when its project director left Illinois to join the Department of Computer Science at the University of Utah. It is primarily a systems engineering project, with three major activities. The best known is the work on the development of specialized backend processors like search or index manipulation engines, to improve system performance. The other two activities are recent additions. One is the development of a user interface that is easy to use, yet powerful, and which can easily be incorporated into a word processing system. The second is the design of the system's internal structure, allowing it to support a variety

of backend processors, and to be readily modified to include new processing modules, such as different index or search schemes, instrumentation, or enhancements.

3. BACKEND PROCESSOR SYSTEMS

Two different backend processors have been proposed to enhance the operation of a text information retrieval system: list merging processors to help manage index files (Stellhorn, 1977; Hollaar, 1978) and search engines to scan information from disk for a particular pattern (Hollaar, 1979). Both enhance the software-based information retrieval system by taking an operation that cannot be efficiently performed on a conventional computer because of bandwidth limitations or excessive control overhead and instead performing it on hardware specifically designed for the function.

3.1 *Backend Search Engine Development*

Since its inception, the Utah project has been developing an implementation of the search engine originally proposed by Haskin (Haskin, 1980; Haskin and Hollaar, 1983). This processor is based on a new form of finite state automation, and was designed to meet the goals of low cost, operation in an environment permitting limited indexing, and support for complex multiterm queries.

Most proposed hardware search systems (Bird *et al.*, 1977; Operating Systems, 1977; Manuel, 1981) use a single centralized search engine. While this provides satisfactory performance for the moderately-sized text databases for which these systems were originally designed (from 300 megabytes to a gigabyte), it can cause severe problems for very large databases. Even with the very high search speeds possible with these systems (millions of characters per second), it still takes thousands of seconds to search a database containing gigabytes of text. The solution is to divide the database into many small parts and assign a search engine to each part. This divide-and-conquer approach not only improves performance because many searches are being conducted simultaneously but, because more searchers are added as the database size increases, provides a uniform response time independent of database size.

A second means of improving search performance is to employ a limited index to reduce the amount of data that must be searched. While a complete inverted file may require substantially more storage than the actual text database (Bird *et al.*, 1978), a simple index indicating whether a word occurs in a document or disk track takes considerably less storage (often less than 20 per cent of the text database size) can dramatically improve system performance. For example, if the index can reduce the amount of data to be searched to 0.3 per cent of the database (which, for a 65 gigabyte database, is still close to 200 million characters) and the query arrival rate is less than one every two seconds, statistically only one query need be handled by each search engine at a time (Haskin, 1980). Furthermore, it is not necessary to have a searcher for each head of the drive, only one per drive, to give this performance. This is important, since advances in head positioners necessary to achieve a low storage cost per bit by having high bit densities will make it difficult and costly to provide data from more than one head simultaneously.

Clearly, implementing a system using a large number of search engines requires that the cost of each search engine, and its associated disk controller and other logic, be comparable to the cost of the disk drive. Disk drives currently available for between five and ten thousand dollars can store 250 megabytes. With today's technology, this

means the searcher must be implemented using mass-produced circuits whenever possible, and simple, medium-speed custom integrated circuits where they can improve performance or allow the use of a readily available component.

Figure 1 shows the major components of the text search engine. It consists of a conventional disk drive and its controller, a microprocessor search controller acting as the interface between the host system and the other components, a term comparator that examines the information from the disk and determines where terms from the query occur, and a query resolver to see if the terms occur in the specified context, proximity, or match the Boolean expression. While the term comparator must operate at disk transfer speed, the query resolver processes information only when a term is detected, and can be implemented using a fast microprocessor, possibly the same one as used for the search controller. Using a stored-program query resolver, rather than special hardware, allows a search engine to handle a variety of different query constructs (such as contexts, proximities, and Boolean expressions) simply by loading the correct program.

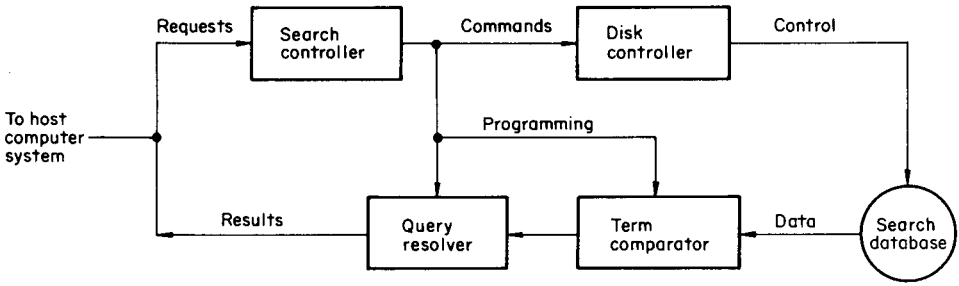

FIG. 1. Block diagram of the text search engine

3.2 *The partitioned FSA*

The term comparator is based on a new class of finite state automaton, the partitioned FSA (Haskin and Hollaar, 1983). Finite state automata provide excellent term comparator capabilities, since they can be readily programmed to handle a large number of terms, including those with fixed- or variable-length don't cares, by simply providing the proper state table. Figure 2 shows the state diagram to match the string DOG. The FSA starts in state 1, and transitions to state 2 when the input character is a D. From state 2, it goes to state 3 on an 0 (and back to state 1 on any other character, as indicated by the # on the transition arc) and to state 4 on a G, indicating that DOG has been found. Figure 3 shows a slightly more complex state diagram, with the transitions back to state 1 when an uninteresting character has been received removed for clarity. It is programmed to match three different terms with a common stem of GRE. Rather than checking for a single character of interest in each state, as in Figure 2, many characters can be specified to produce transitions from a state (as in states 5 or 6 in Figure 3).

If a term comparator can process incoming characters synchronously with their arrival from the disk drive, it is not necessary to include a buffer memory between the disk drive and the term comparator. This not only means that the system cost can be reduced, since no buffer is necessary, but that there is no possibility of a search failing because the buffer was not sufficiently large. For an FSA, this means that the

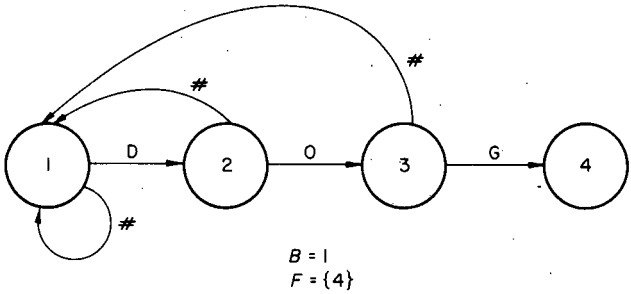

FIG. 2. FSA state diagram to match the string DOG

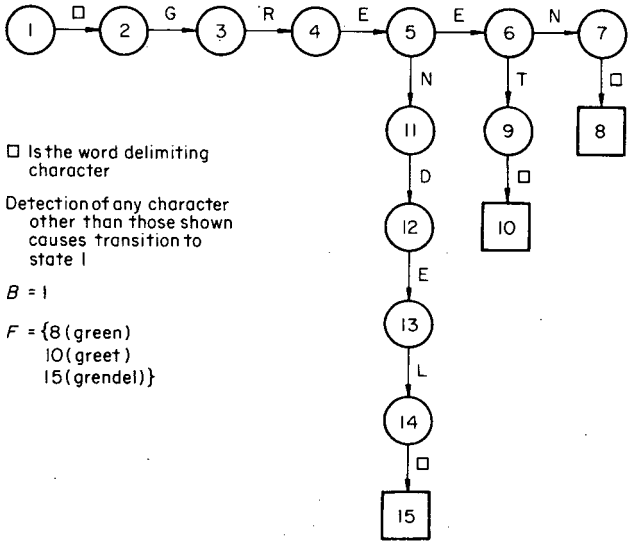

FIG. 3. FSA state diagram to match GREEN, GREET, or GRENDEL

determination of each transition must not take longer than the time it takes the next character to arrive from the disk. For states with only a single character of interest, this is quite simple to achieve. States with many characters of interest either require a very fast state table memory (if a sequential comparison against characters of interest is being made) or a memory large enough to store a transition for each possible character for every state. Both approaches are expensive.

3.2.1 *Basic PFSA configuration*

The partitioned FSA is based on comparing a single character of interest in each state, but permitting the FSA to be in more than one state at a given time. This is achieved by having a separate processor for each possible state the PFSA can be in simultaneously. These processors, called character matches or CMs, are all supplied with the input character from the disk over a broadcast bus, as shown in Figure 4. Additional bus lines are used to return the indication of a match (hit report),

consisting of the CM number and its current state table address, to the query resolver for additional processing.

The character matchers are also connected together to form a ring. As will be discussed later, approximately a dozen character matchers are necessary to handle queries with 100 different terms. Each CM is able to communicate with its two neighbors in the ring, but with no other CMs. This is used to implement forking, a process whereby a CM can force another CM into a specified state to assist in matching a pattern with alternative transitions from a state. For example, consider the

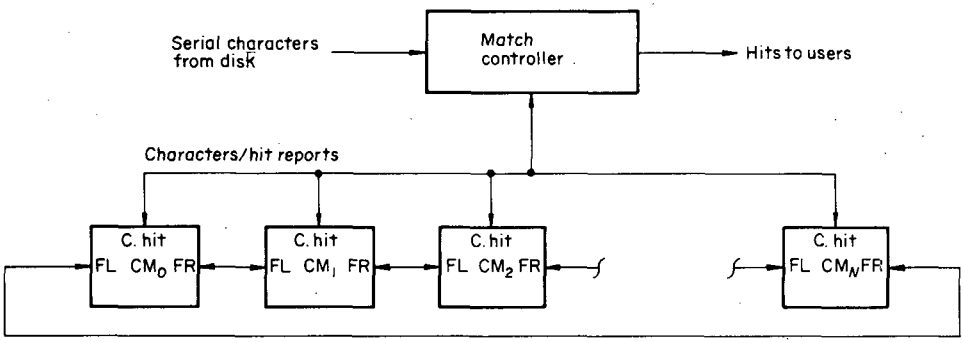

FIG. 4. The PFSA character matcher ring

pattern in Figure 3. A single CM can be used to match the initial substring GRE. At this point, the CM will transition to a state looking for N while forking to a neighbor CM, forcing it to look for E. At most one of these comparisons will be successful. If the original CM's comparison succeeds, it continues processing the NDEL to successfully match GRENDEL. If the CM started by the fork successfully matches its character (in this case, an E), it transitions to match an N while forking to the original CM (which is now idle, since it did not match its N), causing it to look for a T.

3.2.2 *An example PFSA state diagram*

Figure 5 shows a PFSA state diagram matching the seven terms #A?ISM#, IST#, #SCHISM#, #BEST, #BENT#, #BUNT, and #BUNTED# (where # indicates any valid word delimiter, such as a blank, and ? matches an arbitrary span of characters within a word). Three character matchers are required, as indicated by the three regions of the figure. Each CM is started in its idle state, indicated by the large circle, and it returns to this idle state either on an explicit transition (such as from state 7) or when the character specified by a state does not match the current input character. Forks are indicated by transitions crossing from one region to another. The idle state is left based on the type (alphabetic, delimiter, etc.) of the previous character, as indicated within the idle state's circle, and the value of the current input character. A complete description of the operation of the PFSA for these seven terms, including the state table entries, has been published elsewhere (Haskin and Hollaar, 1983).

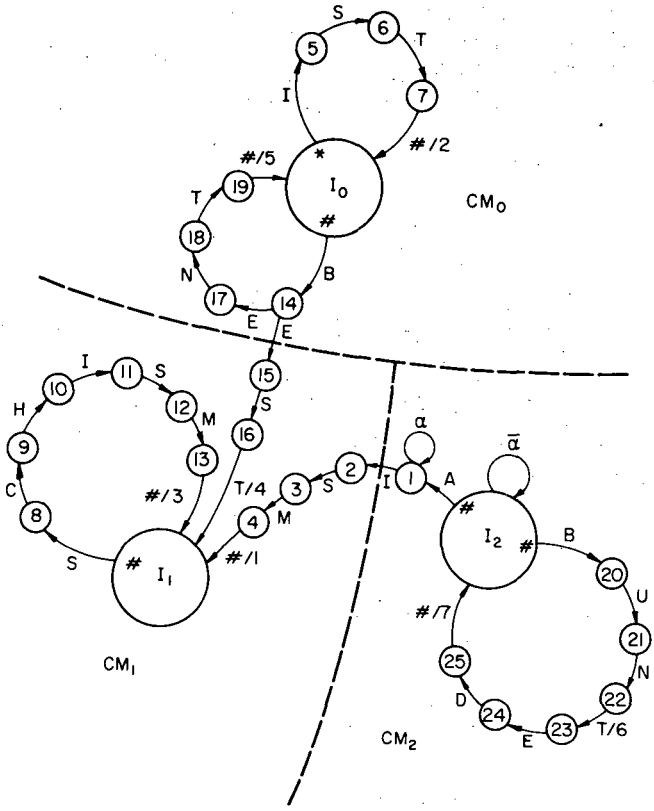

FIG. 5. A PFSA state diagram matching seven terms

Instead of specifying a particular character of interest in a state, a character type, defined by a mapping memory addressed by the current input character, can be specified. This allows a restricted form of don't-care match. For example, it is possible to specify a pattern that matches D, followed by any vowel, followed by G (matching DAG, DEG, DIG, DOG, or DUG). This type match is used to implement fixed- or variable-length don't cares. Another mode specifies a particular character to match, but indicates that it should match regardless of whether the input character is in upper- or lower-case.

3.2.3 *PFSA state partitioning*

It is clear that a conventional FSA state table must be partitioned in a way that assures that no input sequence will require that more than one state in each CM be active simultaneously. This can be done by comparing a state against another state to see if they are compatible (cannot occur at the same time for any input sequence). Obviously, for N states this will require N^2 comparisons, and can be extremely time-consuming for queries with many terms. However, it is possible to reduce the number of comparisons necessary by two different techniques. The first is to reduce the N^2 problem into a number of smaller problems by dividing the terms into different groups based on their initial character. Terms containing don't cares must be placed in

more than one group. As shown in Figure 6, this changes the number of tests required from curve 1 to curve 2.

A futility heuristic called tail removal (Haskin, 1980) can be used to stop further testing of states in a term when it becomes apparent that all its remaining states are compatible with another term. Unless don't cares occur, the remaining states are compatible if a state in one term is compatible with a state in another. Special rules handle the tail removal tests when either term contains a variable-length don't care.

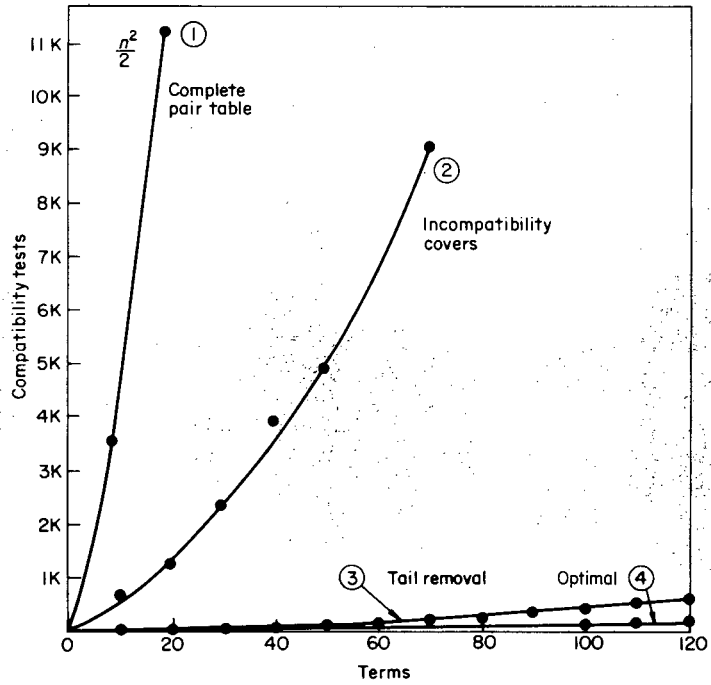

FIG. 6. Improvements to compatibility testing

As shown in Figure 6, tail removal provides a substantial reduction in the number of tests required. In fact, it is near the optimal curve, which is the number of tests required to check only those states that are incompatible. Using these techniques, it is possible to partition a state table with as many as 200 different terms in less than a second with a PDP11/40-class minicomputer.

3.2.4 *CM requirements*

The number of character matchers required to handle queries of various sizes is shown in Figure 7. It is based on twenty sets of randomly selected terms for each query size, and shows the number of CMs required for each set of terms. In most cases (close to 99 per cent in the figure), the number of CMs sufficient for T terms is $5 + T/16$. In the cases where sufficient CMs are not available, the state partitioning fails rapidly, and the intermediate results can be used to divide the query into two or more subqueries, whose results can be later combined.

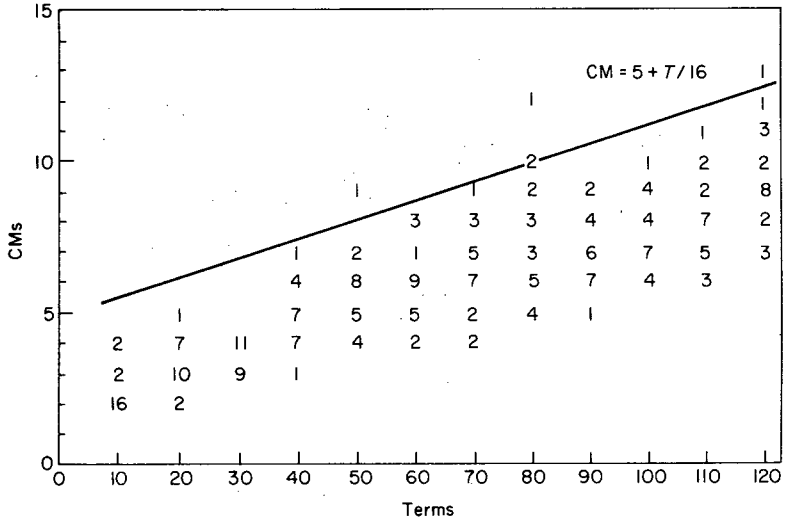

FIG. 7. Number of character matchers required for a variety of query sizes

3.3 *PFSA implementation status*

A demonstration system is currently being developed, and should be operational during 1983. It is based on a 250 megabyte eight inch Winchester technology disk drive. The entire search engine, including disk controller, search controller, query resolver, and PFSA character matchers fits on a single printed circuit board mounted with the disk drive electronics. Prototypes of the character matchers are currently being implemented as custom NMOS integrated circuits, with each CM taking a single 40 pin package. The search controller and query resolver are implemented using a conventional microprocessor.

Each search engine has a network communications port. State table information is broadcast to all search engines on the network from the host system, along with the areas on each disk to be searched. As searching proceeds, indications of documents matching the query are returned to the host from the search engines over the network. The host can also request a search engine to transfer a specified document or disk track over the network for display or additional processing, much like a conventional disk drive and controller.

Due to the small physical size of the 250 megabyte disk drive, approximately twenty search engines can be mounted in a single standard rack, giving a capacity of five gigabytes. The cost of each rack should be comparable to the price of currently available hardware search systems, but with about ten times the capacity and the ability to search the database in parallel, giving more than an order of magnitude increase in speed.

4. A COMMUNICATIONS-BASED RETRIEVAL SYSTEM

Work started in late 1982 on the development of an information retrieval system capable of taking full advantage of specialized backend processors, such as the search engine. It was recognized that retrieval of information from a database may be only a small part of the life cycle of an information request. The retrieved information needs

to be combined with other material to form a report. For existing retrieval systems, this may mean taking the printed results of one or more retrieval operations and entering them manually into a word processing system (or taping them together with handwritten passages between, for later typing).

The Utah system is adapting the work being done in workstation design to give a text handling system that encompasses conventional information retrieval and word processing. It includes a user interface which is easy and consistent to use, and which can easily be linked with a word processing system, and the use of a communications protocol-based approach, rather than simple subroutine calls, to provide a system that can readily support specialized backend processors. The system is based on readily available workstations, with high-resolution displays capable of handling multiple windows, although the system will function using a conventional cursor-addressable CRT display. (The prototype implementation is being done on an Apollo network workstation.)

The user interface to the various system components is through windows, with an editor process running in each window, allowing the user to scroll to any portion of the window and alter the information there. This allows the user not only to browse through the results of a query (although in this case the editor makes special calls on the database manager to bring in information as necessary, rather than operate on an actual file), but to examine and modify previously issued queries. Information can also be moved from one window to another, allowing the results of a query to be selectively incorporated into a document being prepared using the word processor.

Other features of the system include:

1. *Tailorability*. The users can tailor the system to fit their particular needs. This includes simple or complex modifications to the query syntax, specification of desired output formats, and support for local files. This is accomplished by the use of table-driven parsers and output formatters, with the provision to call user-supplied appendages when a particular function cannot be adequately described in the tables. Many of these modifications can be made by editing a special parser table window, using the same commands as for regular editing. In addition, it is possible to configure the system to achieve different cost and performance goals. This can be done by including a different implementation for a particular module, such as the index processor. The new module may use a different algorithm or make use of a special-purpose processor, like the searcher. As long as the communications protocol is followed, replacement of a module is straightforward.
2. *Extensibility*. The use of a standard communications protocol between modules also makes it easy to add new processing modules to the system or modify existing ones to a handle new functions. An example of this might be a rule-based query reformulator, which intercepts user queries and alters them based on known or inferred user patterns, perhaps using the other portions of the system to probe the database or its index. The standard calling sequences for key modules, such as the index processor or the searcher, allow these new modules to fully use the system, while including the appropriate table entries or appendages in the user interface, or defining new windows, allow the addition of the commands necessary to support the added modules. (While the development of such a reformulator is beyond the scope of the present project, other researchers can take the basic

system and incorporate their work, giving them a complete retrieval system to demonstrate their techniques.)

3. *Distributability*. The system can run on a single processor or a network of processors. It supports individual user workstations or a single frontend processor, and specialized backend processors, such as high-speed search units. This is done by using communications-based parameter passing between the various modules, allowing the system to run on one or more processors without changes to the high-level communications. (Of course, suitable low-level protocols must be provided.)
4. *Portability*. The system can run on a variety of different processors without extensive recoding. The use of a portable programming language implemented on a number of systems, such as C (Kernighan and Richie, 1978), and the centralization of operating system dependent actions into a single, multiple entry point module, minimizes the effort needed to port the modules to a new computer system.
5. *Instrumentability*. It is possible to add additional modules to the system to measure user or system performance, allowing it to be used as a testbed to determine the effects of different command structures or specialized hardware. This can be done without affecting system operation by passively monitoring the message traffic on the system network.

4.1 *System logical structure*

The system consists of a number of modules communicating on a network. The most obvious modules are the user interface task, which can be implemented on a large mainframe processor or as an individual workstation, and index file and search processors. These may be implemented as either software modules or specialized processors, as dictated by the requirements of the system.

For the user, the key module is the user interface. This contains the window manager and editors, command parser, and output formatters. Other submodules can be added to support word processing and other text functions. The actual retrieval system requires three key modules. An index processor takes the parsed query from the user interface, performs any necessary transformations (for example, those necessary to handle contexts that are either higher or lower than the index level) and optimizations, and uses the information stored in the index to form two lists of documents: those that are guaranteed to match the query based on information form the index (the hit list) and those that require further searching to see if there is a match (the maybe list). Documents not on either list (hopefully a major portion of the database) have no possibility of matching the query.

A variety of indexing schemes can be supported, ranging from no index at all (which simply returns a special token indicating all documents are included on the maybe list) through partial inversions to indexers that can completely handle a query, returning only a hit list. Online update of the index can be accommodated, with any documents not included in the index at the time of the query added to the maybe list for searching.

The remaining two modules of the retrieval system support the actual text database. The first one handles the searching of the text, based on the query and the maybe list generated by the index processor. The second provides access to the information for display or updating. In most applications, both of these modules would be implemented on the same machine (or parallel machines in the case of multiple search units), but they are separated to provide for improved system throughput by

eliminating the concurrent references by the search system and user browsing of retrieved documents, at the expense of database replication.

Other modules can be added to the basic system. A gateway provides communications with other information retrieval systems, either accepting a query and returning the result or translating the network protocol into a form that can be understood by an existing information utility, such as LEXIS or Westlaw for legal material. Interfaces may exist to other systems, such as a conventional structured database management system.

The structure of this communications-based approach allows a number of other significant advantages over existing retrieval systems, in addition to the ability to distribute the processing among a number of conventional or special-purpose processors. The entire system can function as a backend system driven by a number of workstations, since the design supports multiple host processors. There is no requirement that these workstations be identical, as long as they communicate on the network using the same protocol.

4.2 *Physical structures*

While the logical system design is one of separate processes communicating over a network, a variety of physical structures can be used when actually implementing the system, depending on what is selected for the lower communications protocol layers. If a local network such as an Ethernet is selected, the system may be structured like the logical model, or two or more processes combined in a single machine. Other processes may be replicated to provide parallel execution or divided into subprocesses. For example, if the search processor is implemented using the text search engine previously discussed, its actual structure may consist of a state table partitioning subprocess and many replications of the search engine subprocess.

In fact, the entire system can be implemented as a number of software processes running on a single host processor. The network communications is simulated using an interprocess communications facility or by simple subroutine calls from one module to another. With the exception of the code that handles the network communications, the programs are identical whether the system is implemented on a single processor, is fully distributed, or is anything in between these two extremes.

The overall system design will be completed during the summer of 1983, with a demonstration system operational by the end of 1983. It will include a test user interface, window manager, and editor, as well as word processing support. Simplified modules, operating on a database of about 100 documents, will simulate the operation of the retrieval system. During 1984, these simplified modules will be replaced by modules capable of handling very large databases and interfacing to specialized backend processors, such as the search engine. The system will also be ported to a variety of machines, including those running Unix and the IBM Personal Computer. Finally, a gateway to other retrieval systems will be implemented by the end of 1984.

5. SUMMARY

The Utah Text Retrieval Project seeks to produce a low-cost, high efficiency information retrieval system using state-of-the-art hardware and software techniques. The system will provide rapid response time by using special, small indexes and parallel searchers. Because of its network structure, it can be easily modified to incorporate new functions. Finally, its primary interface to the user will

be through a window in a word processing system, allowing the convenient copying of retrieved text into a final report or document.

The system can also aid research into information retrieval systems by providing a standard base for experiments on novel search and indexing schemes and, by changing the tables in the input parser, different query language constructs. By passively monitoring the traffic on the network, it is possible to measure the performance of the system or any of its components without affecting them.

ACKNOWLEDGEMENTS

A number of people have made substantial contributions to the overall system design of the project. Roger Haskin originally suggested the structure of the Partitioned FSA, and has continued to contribute toward its development. Kent Smith has been invaluable in the VLSI implementation of the search chip, which was designed by Wing Hong Chow. A number of the ideas on indexing are due to Perry Emrath of the University of Illinois. Shane Robison and Michael Zeleznik were responsible for the detailed design of the user interface and the internal communications.

REFERENCES

Bird, R. M., Newsbaum, J. B. and Trefftzs, J. L. (1978) Text file inversion: An evaluation. In *Proceedings of the Workshop on Computer Architecture for Non-Numeric Processing*. pp. 42–50.

Bird, R. M., Tu, J. C. and Worthy, R. M. (1977) Associative/parallel processors for searching very large textual data bases. In *Proceedings of the Workshop on Computer Architecture for Non-Numeric Processing*. pp. 8–16.

Burket, T. G. and Emrath, P. E. (1979) *User's Guide to EUREKA and EURUP*. University of Illinois: Department of Computer Science (Technical Report 79–956).

Haskin, R. L. (1980) *Hardware for searching very large text databases*. PhD thesis, University of Illinois at Urbana-Champaign.

Haskin, R. L. and Hollaar, L. A. (1983) Operational characteristics of a hardware-based pattern matcher. *ACM Transactions on Database Systems 8*.

Hollaar, L. A. (1978) Specialized merge processor networks for combining sorted lists. *ACM Transactions on Database Systems 3*, 272–284.

Hollaar, L. A. (1979) Text retrieval computers. *Computer 12*, 40–50.

Hollaar, L. A., Smith, K. F., Chow, W. H., Emrath, P. A. and Haskin, R. L. (1983) The architecture and operation of a large, full-text information retrieval system. *Advanced Database Machine Architecture* (D. K. Hsiao, ed.). Englewood Cliffs, NJ: Prentice-Hall. (Also in the *Proceedings of the International Workshop on Database Machines*, 1982.)

Kernighan, B. W. and Richie, D. M. (1978) *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall.

Manuel, T. (1981) Look-up chips check entire data base fast. *Electronics 54*, 42–46.

Operating Systems, Inc. (1977) *High speed test search design contract: interim report*. Woodland Hills, CA: Operating Systems, Inc. (Technical Report R77–002).

Rinewalt, J. R. (1976) *Evaluation of selected features of the EUREKA full-text information retrieval system*. PhD thesis, University of Illinois at Urbana-Champaign.

Roberts, D. C. (1978) A specialized computer architecture for text retrieval. In *Proceedings of the Workshop on Computer Architecture for Non-Numeric Processing*. pp. 51–59.

Stellhorn, W. H. (1977) An information file processor for information retrieval. *IEEE Transactions on Computers C26*, 1258–1267.

United States Patent and Trademark Office (1983) *P.L. 96–517, Section 9, Automation Plan*.