5-20-1987

On the implementation of E.R.I.K - Effective Retrieval of Information by Keyword: an information storage and retrieval research system

Eric Tyler

Rochester Institute of Technology

School of Computer Science and Technology

On the Implementation of

E.R.I.K. - Effective Retrieval of Information by Keyword

An Information Storage and Retrieval Research System

by

Eric Tyler

A thesis submitted to The Faculty of the
Graduate School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree
of
Master of Science in Computer Science

Approved by:

Jeffrey Lasky _____ Date 20 May 87
Professor Jeffrey Lasky, committee chairman

Peter Anderson _____ Date 20 May 87
Professor Peter Anderson

William Stratton _____ Date May 20 1987
Professor William Stratton

I wish to be contacted whenever reproductions of this document are requested. I may be contacted at the following address.

Eric J. Tyler Date 5/20/87

Eric J. Tyler

ABSTRACT

The accelerating production of information, often referred to as the information explosion, requires specialized methods for the storage and access of this wealth of knowledge. One solution to the problems of indexing, filing and retrieval of information has been Information Storage and Retrieval (ISAR) systems. The purpose of this thesis is to implement a functional experimental system (known as ERIK - Effective Retrieval of Information by Keyword) for research into various aspects of information storage and retrieval. An ISAR system has been created, incorporating numerous features found in both commercially available and recent research retrieval systems, as well as a few novel features. The design and implementation (originally done on a personal computer, now ported to an UNIX* environment) of the system is discussed as well as some of the background of the problems faced in the information storage and retrieval systems domain. An exciting implementation of an information retrieval system utilizing a newly developed massively parallel computer is also discussed, and its' approach to the problem of information retrieval is contrasted with existing solutions.

KEY WORDS and PHRASES

information retrieval, pattern matching, data base, query, search strategy, bibliographic search, full text, boolean, vector processing, proximity, text search, conflation, relevance feedback

CR CATEGORIES

H.3.1, H.3.3, H.3.5, 1.2.7 (Information Storage and Retrieval, On - Line Information Services).

* UNIX is a trademark of A.T.&T. Bell Laboratories

DEDICATION

For my wife, Cynthia Tyler.
Her love and encouragement sustained me.

ACKNOWLEDGEMENTS

I am indebted to my family and friends who were always there to listen, and encourage me to attain my goals. I wish to thank my thesis advisor, Jeff Lasky, and my thesis commitee for their contributions. I also want to thank the Eastman Kodak Company for their support.

CONTENTS

CHAPTER 1

INTRODUCTION

The accelerating production of information, often referred to as the information explosion, requires specialized methods for storage of and access to this wealth of knowledge. Information Storage and Retrieval, a branch of the Computer and Information sciences, attempts to address this problem by supporting the retrieval of documents (in the realm of information retrieval, "document" refers to a collection of words - which are variously called terms, keywords or concepts) possessing content relevant to a particular topic, while non-relevant documents are ignored. There are many difficulties associated with the storage and retrieval of information in text form. The major problems that persist in Information Storage and Retrieval are those associated with the quality of information and its meaning. Two such problems pinpointed more than twenty years ago still exist. The first is the problem of identifying the content of a document, while the second is determining whether, or to what degree given

documents are relevant to a specific topic.

Information retrieval today involves searching for relevant information located somewhere in a mass of data. The databases, the total body of information that we wish to search, are kept on-line on direct access devices.

1.1 QUERIES

For the retrieval of information, a structured question pertaining to a particular topic must be furnished to the system. This question is referred to as a "query". The answer to the query is a list of documents contained in the database whose contents, hopefully, satisfy the conditions imposed by the query.

1.2 PRECISION AND RECALL

Computer based information retrieval systems offer the opportunity to create queries with differing levels of precision. "Precision" represents the total number of relevant documents retrieved out of the total number of documents retrieved. "Recall" represents the proportion of relevant documents retrieved from the total of relevant documents available in the database. Ideally one wants queries to result in high precision, high recall answers. However, a number of tradeoffs are immediately encountered. A

too precise query will retrieve few or no irrelevant documents, but most likely only part of all of the possibly relevant ones. A more general query will not only retrieve all of the relevant documents, but will also retrieve a large number of non-relevant ones. Figure 1 below shows a contingency table relating the relevance versus non-relevance of retrieved documents as determined by the user, and indicates how recall and precision values are generated for determining the performance of an information retrieval system.

Documents	Relevant	Nonrelevant	
Examined	A	B	A + B
Not Examined	C	D	C + D
	A + C	B + D	A + B + C + D

$$\text{Recall} = \frac{A}{A + C} \qquad \text{Precision} = \frac{A}{A + B}$$

Figure 1

1.3 EXISTING SEARCH STRATEGIES

Over the past twenty years, several strategies for the

satisfaction of information searches have been formulated. They include:

1. Boolean

2. Vector Processing

3. Full Text

4. Proximity Searching

5. P-norms

6. Document Clustering

7. Signature Files

Each strategy requires specialized storage structures and retrieval algorithims. Based upon the structure imposed upon the information databases, various query types will be more effective than others for information retrieval. The strategies and their related structures are described in more detail in the BACKGROUND section.

1.4 THE SMART SYSTEM

The most advanced research into ISAR systems has been performed at Cornell University by G. Salton, et.al. For the past twenty years, Salton and his group have been developing an experimental information storage and retrieval system known as the SMART system. SMART was originally defined as a research tool for investigating the possibilities of automating the indexing and retrieval of textual data. The current implementation is designed to meet the two goals of providing a flexible experimental system for research in information retrieval and a portable interactive environment for actual users. While the system meets the stated goals, it suffers from some of the shortcomings experienced by practically any software system which attempts to be totally "general" or "flexible". The processing performance of the system when carrying out searches is described by one of its implementors as "not blindingly fast, but it is quick enough to be used as an actual retrieval system" [1]. The system does not allow for the mixing of search strategies during queries. That is, a single query cannot consist of elements from more than one of the above listed strategies. These shortcomings have the effect of constraining the ability of the system to act as a research tool for those search strategies which are currently implemented in commercially available retrieval systems.

SMART does, however, contain a number of features which increase the flexibility of interaction with the system on the part of the researcher. The system provides a pre-parser which enables a document to be segmented into specific sections such as title, author, subject, etc. Each section of a document may be parsed, indexed and displayed separately by possibly different methods. Different storage representations of each document are possible to enable research into varying retrieval strategies for the same information. Modularity of the system enables researchers to quickly substitute an experimental routine for practically any facet of the system in order to test hypotheses.

1.5 THESIS OBJECTIVE

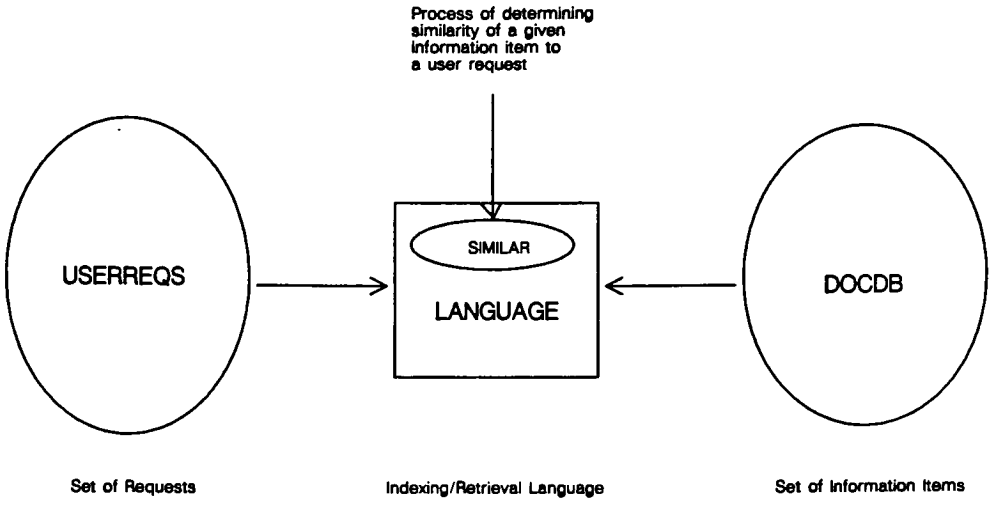
The objective of this thesis is to develop a base information storage and retrieval research system which addresses the shortcomings of purely research oriented ISAR systems such as SMART. It is intended that the system will be robust in terms of indexing and retrieval performance. Multiple search strategies will be supported, as well as the ability to mix all available strategies within a single query.

CHAPTER 2

BACKGROUND OF INFORMATION RETRIEVAL

2.1 OVERVIEW OF ISAR SYSTEMS

Every information retrieval system may be logically divided into three major components: the information item set (documents), requests against the document set and some mechanism for determining which, if any, of the documents satisfy the conditions imposed by requests submitted to the system. In the following figure (Figure 2), the document set is referred to as DOCDB, the request set as USERREQS and the similarity mechanism (contained in LANGUAGE, the indexing and retrieval language) as SIMILAR. The diagram depicts the relationship of these components to one another.



Logical Overview of Information Retrieval

Figure 2

SIMILAR represents a relationship operator which maps specific requests to particular information items. As shown in Figure 2, the determination of the relevance of a particular document to a given query is, in practice, not performed directly. Instead, the documents are first processed via LANGUAGE and converted to a form (see SUPPORTING FILE STRUCTURES below) which enables effective processing. The requests consist of keywords and operators suitable to the search strategies supported by the particular system extracted from user queries by LANGUAGE. The set of keywords which may be contained in a given request or document representation is either pre-specified (controlled), or taken freely from the text of the user query or source document (un-controlled).

The similarity mapping process produces identifications of information items that are potentially relevant to a given request. In some instances, the system places the retrieved identifiers in order of probable relevance to the request. Additional functionality provided by such systems may include the browsing/printing of source documents, recall and editing of previous queries or the ability to perform relevance feedback processing.

Discussions of currently implemented search strategies, and their supporting file structures are provided below. Examples of how a number of such systems map to the simple model described above are available in [2].

2.2 EXISTING SEARCH STRATEGIES

As noted above, there are currently seven primary search strategies found in current ISAR systems, with related search algorithims and supporting database structures. Commercially available retrieval systems primarily utilize Boolean, Proximity and Full Text strategies [2],[12],[13],[14]. A discussion of each of these, and the remaining strategies follows.

1. Boolean - These retrieval systems are designed to retrieve all documents exhibiting the precise combination of keywords found in the query related by Boolean operators. When two query terms are connected by AND, both terms must be present in a document in order for it to be retrieved. When OR connectors are used, at least one of the terms must be present in a document in order for retrieval to take place. Boolean retrieval systems generally provide for high retrieval performance as expressed in terms of processing efficiency, as well as the levels of precision and recall that are available to experienced users [2],[3].

2. Vector Processing - The terms comprising a document are represented in vector form with each term of the document appearing in the vector only once (e.g. D = (T1, T2, T3, ... , Tn) where the Ti are unique terms). Weights may or may not be applied to the terms within the vectors. In the binary case, these weights may take on the values of 1 or 0. Other weight values are derived by various methods [3],[4] to indicate the importance of the individual term to the parent document and the database as a whole. In this context, "importance" is meant to imply the contribution this particular

term makes to the subject matter of the document. Non content-conveying terms such as "by", "more" and "these" usually do not contribute to the subject of a document, while more meaningful terms which appear with a relatively high frequency in a particular document, such as "computer" and "filesystem" will almost certainly pertain to a documents' subject.

The weight determining methods most generally employed by today's ISAR systems are primarily statistical in nature. No semantical, contextual or artificial intelligence techniques are applied. Simple counts, averages and frequency calculations are performed, with the frequency of the occurance of a term in a given document becoming the primary indicator of the importance of the particular term in a document. Query vectors consisting of terms deemed inportant by the user are compared against the vectors of all of the documents in the database. A ranking of the relevance of all documents to the query is determined by performing calculations (generally a form of inner product) upon the weights of the terms appearing in both the query and document vectors.

An example would compare query Q1, with a definition of

Q1 = (T1, T4, T9)

against document vector D56 with term content of

D56 = (.78T1, .67T5, .91T9, .43T23, ... T89)

The resulting relevance ranking calculated by the vector inner product is

(1 X .78) + (1 X .91) = 1.69

In this example, the calculated value, 1.69, would be compared against a user specified threshold. Those documents yielding relevancy values that exceed the threshold are presented to the user in ascending sort order of that value. The user may then experiment with various combinations of query terms to generate suitably sized ranked lists of documents with increasing relevancy values.

The determination of relevance rankings enables vector processing systems to engage in "relevance feedback" with the user. Relevance feedback allows a user to indicate those documents which appear to best satisfy a query from the total list of documents returned in response to the query. The system then examines the vectors pertaining to those documents, modifies the weightings of the query terms based on the weightings of those terms in the vectors and processes the modified query automatically.

For example, if the user had submitted Q1 as described above and had decided D56 was, indeed, relevant to their query, a modified query that would be automatically resubmitted to the query processor might appear as:

Q2 = (.78T1, T4, .67T5, .91T9, .43T23, ... T89)

where the modification method was simple addition of the weights of the matching query and document terms. In this way, the user can iteratively tune his query by

providing additional information. Experiments have shown that incremental retrieval gains in recall and precision due to relevance feedback are initially significant, but decrease rapidly after the first feedback cycle.

3. Full Text - The entire text of each document/abstract is kept on-line and queries are formulated in order to locate specific "patterns" or phrases of text. This method is restricted by the hardware and software currently available, as huge amounts of text must be brought into the computer from mass storage devices and then searched in order to process the query [5],[6]. Such performance considerations reduce the usefullness of this strategy for on-line information retrieval, generally relegating queries utilizing such methods to batch processing.

4. Proximity Searching - In these systems, location restrictions for query terms may be established such that two terms may be required to be physically adjacent to one another (phrase searching), within N words or in the same sentence, etc. Systems with

this search capability enhance individual word index entries to include positional information such as paragraph number, sentence number and location within sentence [2]. This particular strategy is often found in legal information retrieval systems such as LEXIS and WESTLAW.

5. P-norms - This strategy, described in [7], is an extension of conventional Boolean retrieval systems. The Boolean operators are adhered to more or less strictly depending upon the value of a special parameter, known as a p-norm, which is assigned to the AND/OR operators within a query. The assigned values modify how the search algorithims view the operators. AND operators may be modified to the extent that some but not not all terms connected by AND need necessarily be present in order for a document to be retrieved, thereby increasing output size and hopefully increasing recall. Conversly, OR operators may be "restricted" such that more than one of the OR-connected terms need be present in order for a document to be retrieved, thus reducing the output size and hopefully increasing precision. The evaluation equation discussed in [7] is defined such that by setting the operator modifier value equal to one (1) for all operators, a standard

vector processing model is defined. Setting the p-norm modifier for all operators equal to infinity [and limiting the query and term weights to zero (0) or one (1)] yields an equation which models conventional Boolean retrieval processing. The p-norm modifiers may take on values intermediate to zero and infinity, with the resultant operations tending toward more or less strict adherence to the specific Boolean operators contained within the query on an operator by operator basis. Rankings of the relevancy of documents to the query may also be performed, thus aiding the user in reducing unnecessary processing [7],[8],[9].

6. Document Clustering - Preprocessing is performed upon the documents as they are input to determine whether they may be clustered into groups of documents which are considered relevant to particular topics. A representation of the topic(s) that each cluster is relevant to is generated and stored as part of the cluster. For vector systems, the representation takes the form of a group vector. Algorithims for the generation of such representations are available in [16] and [19]. The weights of the terms contained within the group vector are averages of the weights of the specific

term within the documents represented by the group vector. When a search is conducted, the query is compared against each of the document cluster representations. If the query matches one of the representations acceptably (the similarity calculation yields a value which exceeds a specified threshold), the query is then compared against all of the documents located in that cluster to determine which specific documents are relevant to the query. In this way, large numbers of document within the data base can be ignored, and presumably the search is efficient [2],[3],[10].

7. Signature Files - This type of retrieval system performs the storage and retrieval of "messages". A message consists of two distinct sections: the header and the body. The header consists of formatted data such as author, date, destination, title, etc. The body consists of the text of the document itself. The resulting message is stored in a simple sequential file. A representation of the message, known as a "signature" is created and stored in a logical (signature) file when the message is stored. The signature file entry consists of: message type (installation dependent), a pointer to the actual message in the general file,

the formatted data and a signature of the body of the message. The signature of the text is a sequence of bits that approximately represents the significant words contained in the body of the document. Common words are discarded during processing.

To access the documents, a user constructs a query by filling in a template relating to the desired message type. The user fills in some selection criteria relating to the formatted header data, and may also input a "pattern" which may consist of a complex Boolean expression. Messages that satisfy the query are those that satisfy the header section selection criteria, plus the Boolean query against the message body signature [10],[11]. Such systems exhibit poor retrieval processing performance on large message data bases as the signature file must be read sequentially in order to satisfy input queries.

2.3 SUPPORTING FILE STRUCTURES

Numerous file structures have been proposed to support each of the search strategies (with the exception of Full Text - only the text files themselves being necessary). File structures vary in complexity from those possessing little or no organization to those structures maintaining numerous explicit relationships between information items. Linked lists of document terms, inverted indices and sequential record file representations have all been used in various systems. The following discussion details those file structures normally utilized to support the previously discussed search strategies.

Historically, Boolean search strategies have best been supported by some sort of inverted file structure. All terms which contribute to the subject(s) of the documents (high frequency terms such as "a", "the", etc. are excluded) are placed in an index, with pointers indicating all of the documents in which specific terms appear. During a search, the index is accessed and all documents containing the terms defining the query are located. Logical operations are then performed on the document lists based upon the Boolean operators present in the query. The single resulting document list is presented to the searcher as the result of the query.

P-norm search strategies may also be supported by inverted file schemes. Documents containing the query terms are identified in the process described above for Boolean systems. Determination of the strictness with which each Boolean operator is to be viewed and calculation of document relevancy is performed by the retrieval language provided by the system. Document - query similarity equations necessary for these calculations are available in [7] and [8].

One extension (with small modifications) to the conventional Boolean search file structures would provide for supporting the majority of the remaining search strategies of Vector Processing, Proximity Searching and, in one plausible implementation, Document Clustering. This extension would involve the implementation of a file structure which maintains, for example, all of the terms appearing within a document, the number of times the term appears, positional information (e.g. each line that the term appears on), and weighting factors which indicate the frequency with which the term appears in the document (presumed to be an indication of its contribution to the document intent). The following discussions describe how the

listed strategies might be satisfied by such a structure.

Simple Vector Processing systems incorporate file structures which maintain the term weighting factors described above. As stated in the previous description of Vector Processing systems, a query consisting of terms is compared against all of the vectors stored in the proposed file, and an overall determination of document relevancy is calculated. Simple methods such as inner product calculations, or vector matching cosine similarity equations may be used to determine the relevancy of a given document to the query [2].

For Proximity Searching systems, the positional data maintained for each term in the proposed file is employed. Queries consisting of desirable term adjacency criteria (for example, phrase construction) are compared against the information to determine those documents

possessing the correct term/positioning combinations.

One implementation of a Document Clustering retrieval system could be based on Vector Processing as described above. All documents upon input would have a vector created for them. A clustering algorithim would determine those documents belonging to a given cluster and would then generate a vector which would act as a representation of all of the documents in the cluster. During a search, queries would be compared against the representative cluster vectors. If a cluster was deemed promising by the initial comparison, the vectors relating to all of the individual documents located in the cluster would then be compared to the query and relevance determinations performed.

Finally, Signature File systems require system file structures unlike those described above. Indeed, the file structure requirements are much less

stringent in this case. As stated in the strategy descriptions, the actual message text is stored in a sequential file, with a pointer to its location within the file. Then a signature of the message is constructed and entered into the signature file. The access method to the signature file is strictly sequential, using message type as the initial screening parameter. If the message type is matched, the formatted data is accessed and compared for matches. Finally, the signature of the message is compared against the Boolean query. If the message is still deemed interesting, the actual message may then be retrieved from the message file. Very simple file structures will suffice to support this strategy. An example structure would provide a record for each message containing the structured data, length of the message signature and a pointer to the actual message in the message file. Following the message record would be the variable length message signature. Access would consist of reading each record and performing comparisons between the query and the structured data. Those messages showing promise would subsequently have their signatures accessed for further comparisons. Favorable evaluation of the signature would result in the message itself being retrieved from the message file. The signatures of messages deemed

uninteresting by the structured data evaluation would be passed over by "seeking" in the file based on the signature length value contained in the message record.

CHAPTER 3

ERIK IMPLEMENTATION

3.1 OVERVIEW

ERIK was originally implemented on a Digital Equipment Corporation Rainbow 100 personal computer containing two 400 Kbyte floppy disks and 896 Kbytes of memory. Computer Innovations' C86 'C' compiler was used in program development. Approximately 8500 lines of commented 'C' source code were generated for this project. An additional 500 - 600 lines of comments are also contained in the source files. The source code has been downloaded to one of the UNIX based super-minicomputers at the Rochester Institute of Technology Graduate Department of Computer Science and recompiled for the purpose of demonstrating the viability of the project.

Of the seven previously discussed search strategies, ERIK provides the following subset:

Boolean (and / or)

Vector Processing

Proximity Searching (phrase construction and line adjacency)

Full Text (using the Boyer-Moore string searching algorithim)

Precedence of operators is modifiable via parentheses.

The three remaining strategies - document clustering, p-norms, and signature files were excluded for various reasons. The increased retrieval performance with respect to precision and recall that may have been realized by incorporating p-norms into ERIK was not deemed significant enough to justify the amount of additional implementation effort required. Support for signature files was not provided as such support would have required additional file structures and access

methods, as well as extensive modifications to the query processing routines in order to handle mixed strategies incorporating signature file searches. Another reason is the previously described poor performance in terms of processing speeds encountered with signature file systems (see EXISTING SEARCH STRATEGIES, above). Finally, document clustering was not incorporated since debate still continues over the usefullness of this particular approach [19], [20]. Again, processing performance provides a second reason for not supporting clustering as current clustering algorithims are system resource intensive (both CPU and input-output).

An important contribution to ISAR research provided by ERIK is the ability to mix any of the above strategies in queries submitted to the system. Coupled with this ability is the capability to perform relevance feedback experiments based on those previously submitted queries which contained vector processing elements. Finally, ERIK provides the ability to "capture" the query session for further investigation at a later time. Discussion of these and further capabilities follows.

3.2 ERIK FUNCTIONAL SPECIFICATION

The following list of features provide a high level indication of the functionality provided by ERIK.

1. ERIK enables a user to create as many data bases as desired.

2. Once a specific data base is created, ERIK provides the user with the ability to populate the data base via either of two methods: interactively or in a batch fashion.

3. Interactive update also provides the user with the ability to see how the system processes and formats the document source (the "diagnostic" input option).

4. After the data base is populated in some manner, ERIK enables the user to perform queries against the data base consisting of mixtures of the supported search strategies.

   Examples of ERIK queries appear below. The

base elements of the syntax of ERIK queries consist of the following:

Each query must end with a period ('.')

Vector terms may or may not possess weights, contained in brackets ('[', ']'). Examples would be: computer[0.6] and fields[0.89]

An entire vector is indicated by curly braces ('{', '}')

Full text strings are delimited by double quotes (")

The word and line adjacency operators are denoted by w#X and l#X respectively, where X is the number of words or lines which constitutes the proximity limit

EXAMPLES

computer and {files[0.9] system[0.7] disks}
or "computer file systems".

computer w#1 files.

(computer or systems) and programming w#1
languages.

5. In addition to the query language support, several features available on commercial ISAR systems are provided.

6. The ability to save the current context of the retrieval session for later continuation is provided.

7. The user is able to view a documents' source as well as the source and results of previous queries.

8. The user may direct query source and results to an external file for later perusal or printing.

9. The user may limit the number of documents reported as the result of a given query.

10. The user may specify a "trace" wherein the system details some of the processing it is performing.

11. A dictionary capability is provided whereby the user may determine if a particular word or series of words have been indexed within the data base. This allows the user to utilize terms existing in the data base to structure their queries.

12. If a previous query contained a vector clause, the system will enable relevance feedback for that query: providing positive, negative and combinations of feedback possibilities.

13. A simple "help" facility is provided to aid users as they interact with the system.

14. Several utilities are also provided to enable a user or data base administrator to view the contents of the various system files for informative reasons.

Further discussion of the capabilities of the system is provided below in the section on ERIK program design.

3.3 ERIK CONSTRAINTS

Information Storage and Retrieval data bases are static by nature. What is meant here is that ISAR data bases tend to grow over time, with little or no deletion or modification to their contents. The primary concern with such systems is their retrieval performance in processing time and the ability to provide results meeting query criteria. As such, systems of this nature are generally multi-user for retrieval purposes only, and single-user for update purposes. ERIK is such a system. Numerous users may access the system for retrieval purposes at any

given time. However, only one user should be executing the population program at one time, and that should occur when no other users are accessing the data base in question. There is no danger of data base corruption if users performing queries are operating when the data base is being populated. However, erroneous results are possible for those users reading the database.

As stated above, ERIK enables the user to create as many data bases as may be desired. Each simply requires a unique name. Currently, the data base name must be no longer than 15 characters. The only "sizing" limits are the number of terms and lines per document, and the total number of documents per data base. These values are currently limited to 65535. The remaining files may grow as needed to fulfill input requirements.

A "term" to ERIK is any sequence of alphanumerics separated by any non-alphanumerics. A date of the format 12/13/86 would result in three ERIK terms: 12, 13 and 86. Single letter terms are currently retained, unless specified in the stopword list. Therefore "don't" would yield "don" and "t" as terms. As terms are input, they are conflated and/or truncated. Conflation yields terms that have been reduced to their morphological root.

Terms such as computation, compute, computable will be reduced to a common stem in order to "broaden" the term and increase the number of hits in the data base during queries. Terms exceeding 15 characters in length, after conflation, are truncated to that length.

Since data base file names are limited to 15 characters in length, accessing data bases residing in another directory requires the entire path name to fall within the 15 character limit. A compromise is that data base files must all reside within one directory. ERIK programs, when executing, may then assume the files reside within the current working directory.

ERIK currently does not provide the ability to delete documents from a data base. File re-organization utilities are not supplied. Actually, only one file may really be considered a candidate for reorganization - the .trm file - as discussed below.

Due to the size of the project, certain capabilities possessed by purely research oriented systems such as document pre-parsers are not within the scope of this project. Such pre-parsers enable the indexing and

retrieval of prespecified sections of a document by different means. For example, the header of a document may be indexed in one fashion, while the body of the text may be indexed in another. Such distinctions may be desirable from a processing or retrieval performance point of view. Within ERIK, each document is read sequentially, a character at a time, with terms (determined as described above) derived from the source text and indexed within the database. AI techniques, while especially applicable to development of "natural language" query processors to act as the user interface, have also not been addressed.

CHAPTER 4

THE ERIK SYSTEM ARCHITECTURE

4.1 OVERVIEW

One important point to keep in mind when reading the following material is that a primary goal of the implementation of ERIK was to minimize constraints placed on the user of the system, while providing acceptable response time for update and retrieval requests. Because of that goal, ERIK allocates and deallocates memory continously during its processing. ERIK generates binary trees of input terms for both update and query purposes. The system generates linked lists for all of its search strategies. The allocated memory is deallocated as soon as its useful life has ended. This enables ERIK to run on a large variety of machines of varying power and memory sizes. Dynamic memory allocation was chosen as a reasonable compromise between constraining the user with respect to number of terms per document, number of

documents to be retrieved per search, etc. (which would have allowed the implementor preallocation of arrays) and the truly dynamic utilization of space by substituting magnetics for scratch space with the attendant performance degradation.

In light of the previous discussions regarding search strategies and supporting file structures, and the desirability to provide an architecture which will support multiple search strategies with satisfactory performance, the following system architecture was used.

ERIK consists of a set of seven (7) programs and seven (7) files. The files (where "database" indicates the user specified name of the data base) and their contents are described in ERIK FILE STRUCTURES. The programs are detailed in the ERIK PROGRAMS section.

4.2 ERIK FILE STRUCTURES

"database".stp - The stopword list. Contains insignificant terms which are NOT to be entered into the

data base. Examples are be, to, more, etc. It is a simple text file and may be created by any editor. It ideally contains insignificant terms, in random alphabetical order, one per line. It may be added to, or modified at any time. The modified contents will be used at the next invocation of the data base population program.

"database".doc - The document vector file. Contains all significant terms contained in the source document with related data for each term. Stored data includes the word and line position of each term in the source document, total number of times the term appears and a frequency weight factor. The format of the file is as follows:

Information stored for each document (its vector) will be comprised of:

Administrative data for the vector

- document identification

- number of terms in document

- number of unique terms in document

- spares

Data for each term

- term itself

- weighting factor

- number of times the term appears

- list of positional data (word and line 's)

Additionally, the file will maintain administrative data about itself:

Total number of vectors in the file

Total number of terms contained in the file

Offset to the start of the next available vector

Spares

It should be apparent that this file will be utilized for Vector Processing and Proximity search strategies. Modifications to the proposed structure would enable Document Clustering strategies based upon document vector techniques.

"database".off - The document vector offset file. Contains the offset within the .doc file where a particular documents' vector begins. The file is formatted as a simple sequential record file where each record contains:

document identifier

offset in the .doc file for that documents' vector

This file is used to support Relevance Feedback by enabling the system to locate a given documents' vector in the .doc file based on its' identifier.

"database".trm - The term list file. Contains data which indicates for each term in the data base which documents the term appears in. This file provides a mapping of

each term to those documents within which the term resides. Data stored in this file consists of:

File administrative data

- number of unique terms cataloged by the file
- number of records currently in use
- number of continuation records in use
- number of records remaining
- dates of last update, etc.
- spares

Data maintained for each term cataloged consists of one or more records which contain administrative data and then a series of "buckets" which hold information about those documents within which the term appears.

Term record data

- term itself

- continuation record number

- continuation record pointer

- number of buckets currently used

- actual buckets

- spares

Each bucket holds three pieces of information, the document identification number, the offset in the database.doc file for that document's vector and the offset within the .doc file where data about the term starts. If proximity searching is being conducted, this last data enables the algorithims to access the positional data directly without having to read the rest of the vector.

The reasons for constructing this file of fixed length records relate to tradeoffs made as a result of considering the factors of retrieval performance versus the efficient utilization of disk space. It cannot be

known how many entries will be necessary for each term, but utilizing some sort of variable-length record scheme would be unwieldy in terms of performance (especially in the UNIX environment). By fixing the size of the records, efficient paging schemes may be implemented, while hopefully not wasting too much disk space on those terms that do not appear often enough to fill integral numbers of the records. Periodic re-organization of this file can locate related records physically adjacent to ensure acceptable performance over time.

"database".idx - The data base index file. This file comprises a B+ Tree index. All terms cataloged in the .doc and .trm files have an entry in this file at the leaf node level. Each of the entries is of a fixed length (15 characters currently, as discussed above). This allows for very fast search and update performance over a variable length entry scheme. The tradeoff, of course, is that on average, terms will not be the length currently specified. Some space may be wasted on terms of a shorter length, which is not a great concern in light of the prevailing technological trend towards less expensive, higher capacity secondary storage. Or, terms may have to be truncated to the specified length. However, most terms of a great length are reduced via the conflation process anyway. Very few actually get

truncated.

Associated with the entry is a pointer that indicates the offset within the .trm file where this term appears. Therefore, it is a simple matter to access any of the information contained within the database files relating to a term by entering the index file with the term and getting its .trm file pointer, then accessing the .trm file and determining the documents (and their .doc file offsets) that the term appears in. As stated, the file is a B+ Tree index, with all of the terms having an entry in a leaf node. All of the leaf nodes are also chained together in sequential order enabling one to do a sequential scan of all terms cataloged by the database in alphabetic order. Also, all of the nodes at each level of the tree are chained to their predecessor to enable reconstruction of the file in case of corruption.

Administrative data contained in the file consists of:

File administrative data

- file offset to the root page

- offset of the first sequential page

- number of levels in the tree currently

- pointers to initial page of each of the levels

- spares

Each data or index page also contains administrative data:

Page administrative data

- page type (index or data)

- number of entries in page

- offsets to prior and succeeding sibling pages

- level of tree page belongs on

- file offset for this page

- spares

"database".err - The database error log file. If any software errors occur during the execution of eriki or erikq throughout the life of the data base, they are recorded in this file. The file may simply be "dumped" to the screen. If the file grows too large, it may be deleted and recreated with zero (0) length.

"database".inp - A batch input file. Optional. A simple text file containing the names of document source files which are to be input to the specified data base. Currently source file names are limited to 15 characters in length.

4.3 ERIK PROGRAMS

The programs comprising ERIK and their various purposes are described below. The vast majority of the functionality supported by ERIK is provided by two of its programs: eriki and erikq. Of the two, erikq provides the most functionality.

dbcreate - Creates an ERIK data base. Builds an instance of each of the above files, except the .stp and .inp files.

eriki - Provides the functionality to populate the data base.

erikq - User interface for manipulation of the data base.

idxdmp - Generates a dump of the contents of the data base index file.

trmdmp - Generates a dump of the contents of the data base term file.

docdmp - Generates a dump of the contents of the data base document vector file.

offdmp - Generates a dump of the contents of the data base document vector offset file.

Further specification of each of the programs is provided in the following section.

4.3.1 ERIK PROGRAM DESIGN

The designs of the programs eriki ("ERIK input") and erikq ("ERIK query") are discussed in detail below. The remaining ERIK programs - idxdmp, trmdmp, docdmp, offdmp and dbcreate are extremely simple in their design. Each of the "XXXdmp" programs simply utilize the appropriate data structures to sequentially scan the related file ("XXX" implies .off, .idx, etc.) and dump its contents.

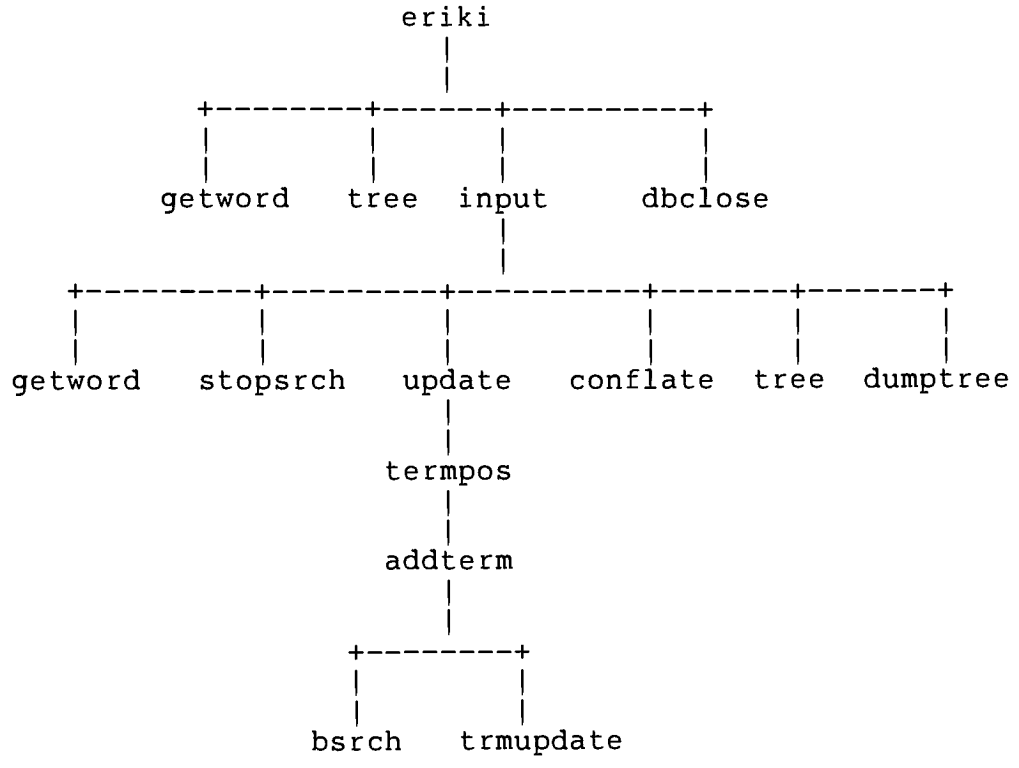
dbcreate simply constructs an instance of each of the data base files except the .stp and .inp files. For the .doc and .trm files it simply writes the file administrative data and closes the file. The .err and .off files are created with zero (0) length and closed. Finally, to create the .idx file, dbcreate writes file administrative data, creates one (1) data page: the "root" of the B+ Tree and closes the file.

4.3.2 DESIGN OF Eriki

eriki is the data base population routine. It reads document source files and updates the data base files

appropriately. Two types of update are possible: interactive and batch. If the user indicates interactive updates this program prompts the user for source file names and inputs those specified. If batch update is requested, a file of document source file names is read and each document is added to the data base.

The following page displays a structure chart of the major modules contained within eriki. The diagram shows which module calls which, in roughly call sequence order. System and/or run - time library calls (e.g."read" or "write") are not shown on the diagram.


eriki
getword
tree
input
dbclose
getword
stopsrch
update
conflate
tree
dumptree
termpos
addterm
bsrch
trmupdate

The type of processing that occurs, and where it occurs consists of:

Input the stopword list and create a binary tree of the terms (performed in eriki proper by calls to getword and tree). getword forces input text to lowercase, and generates terms as defined above.

Determine update type desired - batch or interactive. Either type is handled within eriki proper.

Call update to read the source file and update the data base files.

update calls getword to read the file. It then calls stopsrch to see if the input term is in the stopword list. If not, it conflates/truncates the input term via the conflate call and creates a binary tree of the document terms by calling tree.

conflate performs conflation on the input term utilizing a conflation algorithim and conflation rules provided in [15]. If, after conflation, the term length still exceeds the system maximum, the term is truncated.

Once the entire file has been read, update determines whether the user wanted "diagnostic" input (available only under interactive input). This functionality enables the user to see the terms in the document, the term count in the document and the terms' positional data. This is accomplished via the call to dumptree which simply performs an in - order traversal of the binary tree, dumping the data.

If the user wanted to update the data base, input calls update.

update determines which document this is and updates the data base files by calling termpos.

termpos performs an in - order traversal of the

binary tree. It writes each term to the .doc file followed by its' weight (as calculated by termpos), followed by the count of the term in the document and finally all of the positional data for the term in line , word order. Once the .doc file is updated for each term all of the memory dynamically allocated for the term in the binary tree is freed and the .idx and .trm files are updated via a call to addterm.

addterm checks to see if the term is in the .idx file. If it is, it calls trmupdate indicating this is a repeat term. Otherwise, addterm calls trmupdate indicating a new term and adds the term to the B+ tree. addterm performs caching of the index and data pages comprising the tree. The root and last accessed page from each level of the tree are saved to speed access for subsequent searches/updates. An outcome of this technique is that the data bases are therefore restricted to single-user updates.

During the process of searching/updating the

tree, addterm calls bsrch to perform a binary search of the contents of the tree node pages.

trmupdate updates the .trm file depending on what addterm said about the term being a repeat or new one. If repeat, it takes the record position that addterm passed it and updates the term record. If it's a new term, the routine allocates a new record, initializes it and updates the file. If the term has occurred frequently in the document collection, trmupdate may have to follow and/or create chained continuation records in order to track all of the documents that the term resides in.

Once all of the source files are read and input to the data base eriki calls dbclose to close the data base prior to exiting.

If, at any time during their processing, any of the above modules encounters a system error, the error is logged in the .err file. A traceback of the routines in the call stack is provided to enable the rapid pinpointing of the

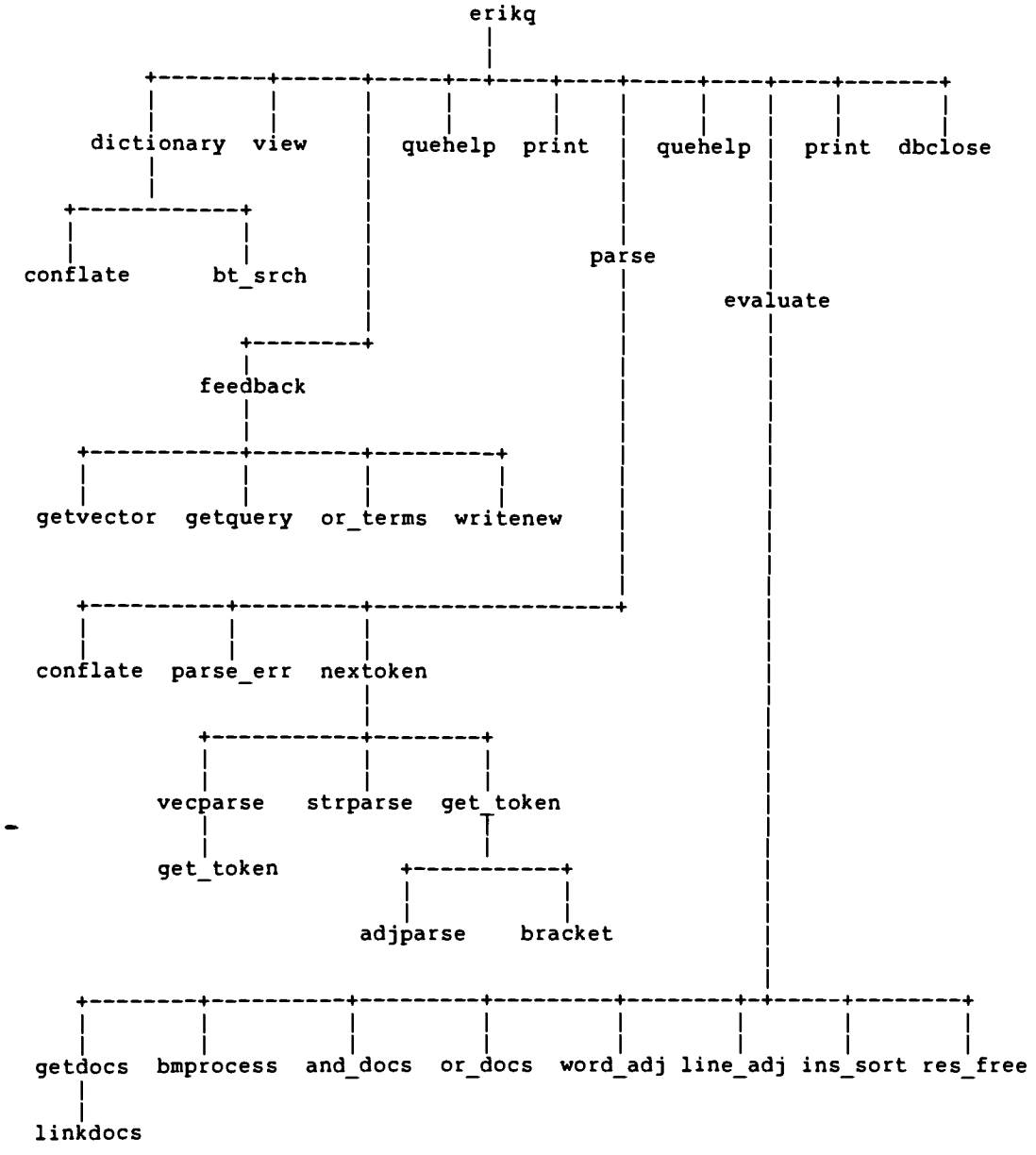
actual error.

4.3.3 DESIGN OF Erikq

erikq is the data base query routine. In addition to supporting the creation and evaluation of a wide variety of query formats, erikq provides the user with a number of features available on commercial systems. See the Functional Specification section for further details.

The following page displays a structure chart of the major modules contained within erikq. The modules indicated generally provide the functionality available as commands at the command level within erikq. Therefore, the diagram does not necessarily indicate any call sequence order. System and/or run - time library calls (e.g."read" or "write") are not shown on the diagram.

ERIKQ STRUCTURE CHART



The type of processing that occurs, and where it occurs consists of:

erikq checks to see if a previous session file has been indicated. If so, the file is opened and necessary context information is read. The code checks to see if the indicated file was, indeed a previous session file by looking for an indicator. If a previous session was not indicated, erikq creates a temporary session file for its' use. Each query, as it is successfully processed, is indexed with its file offset within the session file to enable later access to its text and results. Additionally, this module establishes an important data structure - the query evaluation stack. The evaluation stack is used by erikq to accept parsed queries and act as a "scratch pad" during query evaluation.

If the user requests help, erikq calls quehelp (query help) to provide this functionality. quehelp simply reads a help text file and dumps the text available there for the command the user indicates.

If the user wants dictionary information, erikq calls dictionary. dictionary prompts for either a single term or a range of terms. If a single term is indicated, dictionary conflates/truncates the term using conflate and using btsrch searches the .idx file for the terms' existence. If a range is indicated, dictionary verifies it is, indeed, a range and then outputs all terms in the data base lexically within the indicated range (the indicated terms need not be in the data base).

If the user wishes to view a document, or the source or results to a previous query, erikq calls view. This routine prompts for the output desired and which query or document is wanted. If a document is indicated, view opens the file .txt where " " is the document identifier provided by the input program eriki. If a query or its results is desired, view accesses the session file based on query number to retrieve the requested data.

If the user wishes to "print" a query's source or results, erikq calls print. print prompts

for the type of data desired and which query. It also prompts for an output file name. The routine then accesses the session file based on query number to retrieve the requested data and directs it to the indicated file.

The default command during the session is to evaluate a query. To initiate this activity erikq calls parse. This routine comprises an operator left precedence parser that has been modified to process an enhanced Boolean retrieval language. parse establishes a small stack for reducing the input query to reverse polish notation within the evaluation stack established by erikq main. The routine then accepts user query input until a period (.) is input to end a query, or a '' is input to abort that query. If the query is aborted, the user is returned to the query input phase. If the user wishes to return to session control they may enter 'exit' anywhere in their query. parse processes the query by making a series of calls to nextoken, conflating/truncating any terms by calling conflate and creating the RPN version of the query. If any syntactical or lexical errors are detected, the user is

notified via parseerr. If a query parses correctly, the query is indexed within the session file for later use.

When parse calls nextoken, the routine determines if any more input is available in an input token queue it maintains. This queue is provided in order to support vector processing by ensuring an entire vector may be processed as a single entity. nextoken parses vector and string clauses directly via calls to vecparse and strparse. For processing of Boolean and other tokens, nextoken calls gettoken.

gettoken performs the bulk of query input and parsing. It delimits input tokens and determines which are plain words, operators or possibly higher level clauses (e.g. vectors). The routine inputs characters until white space or a special charcter is encountered. Depending on the special character gettoken will call either adjparse to parse a proximity operator or bracket to determine weighted vector terms.

Once a query has been successfully parsed and control is returned to erikq, evaluate is called. This routine, like parse, establishes a small stack for a scratch pad as it traverses the evaluation stack populated by the parser. The routine runs down the evaluation stack interrogating each entry there and performing processing as necessary. Interim results are pushed onto and popped off of the scratch pad stack as required. If the current token is a simple term, evaluate calls getdocs to acquire a linked list of identifiers which indicate documents within which the term resides. If the token type indicates a string clause, the top result on the scratch pad is popped and bmprocess is called to access the documents indicated within the list and determine whether or not they contain the indicated full text string. If the token on the evaluation stack is either of the Boolean operators, the top two (2) results (which had best be document lists from processed terms) are popped from the scratch pad. The lists are then submitted to the appropriate routine (anddocs or ordocs) and a result list is returned and pushed onto the scratch pad. If the token is an adjacency token, the top two (2) results are popped as

for Boolean operators. The two lists are ANDed by calling anddocs since in order to be adjacent, the terms must reside in the same document. The appropriate adjacency routine (wordadj or lineadj) is then called and a resulting list of document IDs is returned. Once the entire stack has been evaluated and a single result list has been generated, the results are output to the session file following the query that it is related to. If the query contained a vector, the result list is first sorted in rank order based on the document weights generated during processing. The sorting is accomplished via a call to inssort which performs an in-place insertion sort of the resultant document linked list. Finally, throughout the evaluation process as interim result lists are no longer necessary, evaluate frees up the memory taken by the list by calling resfree.

The Boolean operator routines - anddocs and ordocs determine the lengths of the two lists to be operated upon. The shorter of the two lists is chosen as the base list to work from in order to reduce comparisons, thereby

reducing processing time and boosting performance.

The adjacency routines access the .doc file to derive positional data for the supplied term. Ordering of the terms in the query is maintained. That is, the first query term must reside in the document before the second term and it must meet the indicated proximity criteria (within two words, three lines, etc.). Only a single instance of meeting the criteria is required. An appropriate linked list of document identifiers is returned.

The routine getdocs performs a number of activities. Its' primary function is to determine which documents contain a given term. It accomplishes this by calling linkdocs which reads the .trm file and creates the document linked list. If the input term is a vector term, a weighting factor provided during the parsing is applied to the weight of that term in each document it resides in as maintained in the .trm file.

The routine bmprocess enables full text processing. The routine is an adaptation of an implementation of the Boyer-Moore string searching algorithim created by S. Recard 28-Feb-84 and forwarded to J. Lasky. It generates a linked list of those documents which contain the input text string.

Finally, with respect to system functionality, erikq will call feedback if the user indicates a desire to perform relevance feedback on a previous query. The user is prompted by the system to input the desired query. The query number is error checked against the session index. If acceptable, the system then prompts for a document identifier for a relevant and/or a non-relevant document. If either type of document is indicated, a multiplier value to modify the weight of the term in the subsequent query will be requested. The multiplier defaults to one (1) if no input is received. For each document indicated (one relevant and/or one non-relevant) feedback calls getvector to get all of the terms contained in that document. The terms are placed in a linked list for further processing. The terms

contained in the previous query are then retrieved by a call to getquery. A linked list of terms results. The lists are then or'd together via a call to orterms to create the query that will be resubmitted to the system on the behalf of the user. The query is written to the session file and indexed by writenew and control returns to erikq which calls parse to initiate the evaluation of the query.

getquery accesses the .doc file to gather all of the terms that that document contains. The multiplier received in feedback is applied to each terms' weight as they are retrieved from the file.

getvector accesses the session file to read the initial query and gain the vector information contained therin. A linked list of terms is returned.

Once the user indicates the desire to exit a session and erikq has created the save file (if requested), erikq closes the data base by

calling dbclose and exits.

CHAPTER 5

IMPLEMENTATION EVALUATION

5.1 SYSTEM SHORTCOMINGS

This section describes approaches that would be taken if the opportunity to "do it all over again" were presented to the implementor.

The current implementations of the query evaluation stack and query input token queue utilize pre-allocated arrays for the data structure. They are too inflexible and possibly wasteful of memory resources. In order to support reasonably sized queries, the arrays would have to be allocated to sufficient sizes. Those values can not be known a priori. Therefore, either the software will report an error and exit (it checks for array overflow in both cases), or the arrays are allocated to such a large value that, for the majority of queries,

memory is wasted.

There is currently a limit of 65535 on the number of documents that may be stored in a given data base. This results from the fact that an unsigned short ('C' language storage declaration) variable is used for tracking acceptable document identifiers. Use of a 32 bit data type, whether signed or unsigned should result in sufficient document identifiers to effectively make the data base "limitless". An interesting related topic is that the current limit on the number of words and lines in a document is also 65535 (same reason as above). This limit does not need to be increased. Not many documents approach that size, let alone any document that may be a candidate for information storage and retrieval. Most likely, abstracts of large documents would be submitted to such a system, as opposed to the documents themselves.

The current algorithims for the update of the .trm file (trmupdate) operate on a record basis. It may make sense to rewrite this code to operate on a "page" basis. It is realized that a page is nothing more than a record, however, the philosophy of processing might be changed for the better if the data stored in this file were

stored in pages and disk i/o were optimized appropriately as opposed to reading/writing a less than optimal number of bytes for a single record operation.

As with any data base product, the performance bottleneck results from the requisite disk i/o. This unfortunate fact is magnified in ISAR systems since such massive amounts of data may be required to be accessed and processed in order to satisfy an user query. Realizing this fact from the start, i/o operations have been optimized or eliminated as much as possible in general (remember the query term tree in erikq used to reduce i/o). There are, however, a few spots in the code (termpos called within eriki is such an example) where disk operations may conceivably be optimized further.

The stoplist search is currently implemented via a binary tree. Overall document input processing performance could be boosted by substituting a better performing algorithim such as surrogate encoding of the list, as used in numerous spelling checker programs.

5.2 FUTURE EXTENSIONS

This section briefly describes feasible future enhancements to the ERIK ISAR system.

A major extension would be the addition of multi-user support. The primary impact of such an effort would be felt in the B+ Tree update algorithims. A major stumbling block to such an implementation is that there is currently a lack of some sort lock arbiter on Unix systems.

The ability to delete documents, and the subsequent word deletions would be desirable. Again, an extensive effort would be necessary in the B+ Tree algorithims. Particularly if multi-user support is desired.

Currently only one file is really a candidate for a re-organization utility - "database".trm. Re-organization of that file would result in improved performance for query evaluataion. An utility would have to be provided.

Data encryption could be envisioned wherby the words maintained within the various data base files are encrypted, and thus less susceptible to perusal via file dumps, etc.

When processing queries, most systems (if not all) do not report the success of the system in processing part of the query. What is suggested is that a means be provided wherby the user is told how well the system had done in processing the query to a point if the remainder of the query causes the overall query to fail. An example would be a query of the nature:

.... or computers and avacados ....

The system may have been dutifully processing the query through "or computers", but the "and avacados" with the restrictions caused by the Boolean "and" may cause the current result list to be lost. If the system could report to the user that the query worked to a certain point, and what the results were, the user may be able to generate more effective subsequent queries.

Query evaluation support for date retrievals would be an extremely useful addition. Currently users can not

request all documents that meet the query criteria, and were generated before January 15, 1985, etc.

Support for document clustering may be added. It may be that the inclusion of a "database".clu file possessing roughly the same structure as the .doc file, but containing the clustered document representations may be sufficient from a file support perspective. Query software would need to recognize a cluster request and sequentially scan the .clu file performing appropriate comparisons. If a reasonable match were found, the .clu file could indicate which vectors in the .doc file to investigate for greater detail. Appropriate update software would, of course, be required.

Security features may be added. Logins to an ERIK data base, with requisite passwords would be relatively easy to implement.

5.3 RELATED THESES TOPICS

Listed are several thesis topics which, if implemented, could possibly enhance/complement this project.

1. Implementation of efficient lock management under Unix systems. The implementation need not result in a generalized lock manager. Availability of a lock manager would enable ERIK to be modified such that it could support concurrent readers and updaters while maintaining database integrity. The lock manager and subsequent lock algorithims within ERIK should be designed such that they alleviate as much as possible the performance penalties normally associated with concurrently accessed databases.

2. Investigation of hardware solutions to the ISAR problem set. Particularly the processing of full text queries. Some companies (GEISCO is an example) have designed hardware approximating "vector processors for text". It may be possible to provide the fast Boyer-Moore algorithim [6] on a board which may be pipelined on a DMA disk channel.

3. Use of parallelism in addressing the problem set presented by information retrieval. One use might entail processing distinct portions of a given query on separate processors. A scenario would be to have one processor access the data base for documents containing one word, while another processor is performing the same activity for another word in the same query. This implementation would accomodate the full suite of functionality found in commercial and research ISAR systems, including proximity searching, conflation, word frequencies, etc.

4. A recently developed example of the use of massively parallel computers to address the information retrieval problem is of such importance that it is discussed in the next chapter.

CHAPTER 6

THE CONNECTION MACHINE

6.1 INTRODUCTION

This chapter discusses the Connection Machine (CM), an important example of massively parallel computer architectures, and its application to the problem of information retrieval. A recent article [21], describes the development of an ISAR system based on the capabilities of the CM. The system presents a new means for processing Boolean and simple term queries utilizing multiple processors and high bandwidth I/O capabilities. This system also provides an user interface incorporating relevance feedback techniques. The implementation of the system is described below.

6.2 THE CONNECTION MACHINE ARCHITECTURE

The CM is a highly parallel computer whose current version consists of between one and four modules of 16,384 proprietary 32 bit processors. A maximum configuration therefore results in 65,536 processors. Each processor possesses 512 bytes of memory and a 1 bit wide arithmetic and logical unit (ALU). Each module is controlled by a single microcontroller which executes "macroinstructions" originating from a host computer. That is, a single, possibly conventional computer (e.g. VAX) executes program instructions, broadcasting those to be processed in parallel to the multiple processors in the CM.

The modules can be configured to run the same program simultaneously, or to execute independently, thus processing up to four programs concurrently. All processors contained within a given module execute the same program, with each operating on the contents of its' own memory. The ALU for each processor contains a context flag which indicates whether or not this processor is to be participating in this particular execution of a program. Three independent communications networks are provided: a 1 bit wide global OR, a 2 dimensional grid and a 12 dimensional hypercube with 16 processors at each vertex.

6.3 DOCUMENT REPRESENTATION IN THE CM

The basis of the ISAR system as implemented on the CM is the representation of a document within the memory of each of the CM's multiple processors. The primary requirement in this implementation is the ability to detect the presence of a given word. Additionally, the document representation must be compact. The method chosen for the representation of documents was originally developed for use in spelling checker programs and is known as surrogate coding [18]. Advantages include rapid detection of the presence of a word, minimal processing overhead and a compact representation. Disadvantages are the loss of positional and word frequency information and the possibility of false positive word detection.

A surrogate coded table consists of an array, N bits in length. To represent a given document, each word contained in the document is hashed via K independent hash functions, with the resulting numeric viewed as a bit to be set in the bit table. For example, if the word to be represented is "parallel", and K = 5, the word would be hashed 5 times, and the resulting bits would be set in the table. When a word is submitted in a query, the same 5 hash functions are calculated, and the resultant bits are AND'ed within the bit table. If the

result is positive, the word exists in the table, and therefore the document.

It is possible for a probe to return a false positive. The probability of a false hit may be calculated by determining the probability that a given bit in the table will be 0. That is, to reduce the error rate, each bit should take values of 0 and 1 with equal probability. For W words, the probability, P that a given bit in a table of size N will be 0 is equal to:

```
P (approximately) = e ** (-KW/N)
```

P becomes 1/2 when :

```
K = (N/W) log 2
```

The probability of false acceptance can then be shown to approximate:

```
(1 - P) ** K = 2 ** -K
```

Depending on the false hit rate that is considered acceptable for a given application, the values of N, K and W may be tuned to yield optimal performance.

6.4 USE OF THE CM IN INFORMATION RETRIEVAL

As mentioned earlier, each processor in the CM contains 512 bytes of memory. A number of those bytes are reserved for scratch memory, reducing the number of bytes available for document representation to 384 bytes. This memory is generally "partitioned" into six tables of 512 bits, or three tables of 1024 bits. The number of words to be inserted into each table must be limited as the above discussion indicates. For a 512 bit table, 15 to 30 words may be accomodated reasonably. If a given document contains more than the alloted number of words, several tables are used for each document. Each of the tables are generally placed in different processors, which are "chained" together, and report to a designated "head processor".

Probes for words in a document are not performed in isolation, but as components of Boolean queries, or as components of simple queries where each word is given a weighting factor.

6.4.1 BOOLEAN QUERY PROCESSING

Boolean queries are built from primitive word probes that are combined via Boolean operators (AND/OR). To compute the AND or OR of a set of word tests, each table is probed independently by the processor containing the document in its' memory, and the results are passed to the head processor. At the head processor, the appropriate Boolean operation is applied to the provided data, and a final result is arrived at.

6.4.2 SIMPLE QUERY PROCESSING

For simple queries, weight values are assigned for each word in a query, and each document in the data base is then scored by adding up the scores for each of the query terms that the document contains. This is accomplished by totaling the scores of the words that each processor contains, and then passing that value on to the head processor. The retrieval process may be halted when a specified number of documents have been retrieved, or when only those documents exceeding a particular score have been retrieved.

6.5 BENCHMARKS PERFORMED

Benchmarks of a information retrieval system were performed on a CM configured with 16,384 processors in September of 1985. The test data base was composed of 31,994 documents, totaling 18 megabytes. Each CM processor was configured with two bit tables of 64 bytes each. The packing density of each table was limited to 35 words. A number of trial queries, composed of between 25 and 20,000 words were evaluated. Query execution times varied between 0.004 seconds for a 25 word query and 0.295 seconds for a 20,000 word query. Extrapolation of the benchmarks indicates that a fully configured CM (65,536 processors) could process identically sized queries in the same time frames while the data base size could grow to greater than 71 megabytes.

6.6 USE OF RELEVANCE FEEDBACK

The user interface designed for the CM ISAR system incorporates relevance feedback techniques. Both positive and negative feedback indications are allowed. The authors of [21] feel that "relevance feedback leads to both high precision and high recall as a result of the large number of words employed in the search process".

The interface utilizes a "seed" approach where the user is requested to provide an initial query consisting of a small number of words. The system performs the appropriate Boolean or simple query and returns a list of documents initially considered relevant to the query. The user is then presented with a multi-windowing user interface which allows them to browse the previously retrieved documents and indicate those documents that the user may feel are particularly relevant or non-relevant to their query. A subsequent query is conducted automatically, utilizing all of the words contained in the documents marked appropriately by the user. Multiple iterations of this process are possible. Results of the techniques as described by the authors reach very favorable levels of recall and precision. Values provided indicate levels of precision and recall approaching 0.85 and 0.95 - 1.0, respectively for specific queries.

6.7 EVALUATION OF THE CM ISAR SYSTEM

While it is readily apparent that the Connection Machine is an exciting contribution to the field of Information Retrieval, this author does not feel that the CM should be viewed as a panacea for the problems presented by the

world of information retrieval. It is true that the CM is capable of the incredibly fast searching of large document data bases for specific words, and simple Boolean connectives of such words. It is also true that the user interface currently available with the CM system incorporates features desirable from both query refinement and user-friendliness standpoints. However, this author feels that the sheer brute-force searching of documents for specific terms will not prove sufficient for answering all aspects of the information retrieval problem set. The loss of positional and word frequency data is felt to be a significant lack in the overall capabilities of the CM ISAR system. Phrase construction is impossible, and more sophisticated techniques for determining document relevance based on word weighting factors are unavailable. Techniques available in current ISAR systems to "broaden" search terms, such as conflation and thesaurus capabilities are also not supported. Finally, the fact that future artificial intelligence or other contextual techniques for the analysis of the content of documents appear to be precluded in the currently available system is viewed as a serious disadvantage.

CHAPTER 7

BIBLIOGRAPHY

[1] Buckley, C. "Overview of SMART Implementation". Cornell University. (3/22/84).

[2] Salton, G. and McGill M. "Introduction to Modern Information Retrieval". McGraw-Hill, 1983.

[3] van Rijsbergen, C.J. "Information Retrieval". Butterworths, 1979.

[4] Sparck Jones, K. "A Statistical Interpretation of Term Specificity and its Application in Retrieval". Journal of Documentation. Vol 28, (1972).

[5] Aho, A. and Corasick, M.J. "Efficient String Matching. An Aid to Bibliographic Search". Communications of the ACM. Vol. 18, 6 (6/75).

[6] Boyer, R. and Moore, J.S. "A Fast String Searching Algorithim". Communications of the ACM. Vol. 20, 10 (10/77).

[7] Salton, G., et.al. "Extended Boolean Information Retrieval". Communications of the ACM. Vol. 26, 12 (12/83).

[8] Salton, G. "The Use of Extended Boolean Logic in Information Retrieval". Cornell University Depart of Computer Science Report number TR 84-588.

[9] Salton, G. and Voorhees, E. "Automatic Assignment of Soft Boolean Operators". Cornell University Depart of Computer Science Report number TR 84-608.

[10] Christodoulakis, S. and Faloutsos, C. "Signature Files". ACM Transactions on Office Information Systems. Vol. 2, 4 (10/84).

[11] Christodoulakis, S. and Tsichritzis, D. "Message Files". ACM Transactions on Office Information Systems. Vol. 1, 1 (1983).

[12] Lefkowitz, D. "File Structures for On-Line Systems". Spartan Books, 1969.

[13] Martin J. "Computer Data-Base Organization". 2nd. edition. Prentice- Hall, 1977.

[14] Wiederhold, G. "Database Design". 2nd. edition. McGraw-Hill, 1983.

[15] Gerrie, B. "Online Information Systems". Information Resources Press, 1983.

[16] Salton, G. (Editor) "The SMART System". Prentice-Hall, 1971.

[17] Bayer, R. and McCreight, E. "Organization and Maintenance of Large Ordered Indices". Acta Informatica 1, 1972.

[18] Mc Ilroy, M.D. "Development of a Spelling List". IEEE Transactions on Communications. January, 1982.

[19] Voorhees, E. "The Cluster Hypothesis Revisited". Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. 1985.

[20] Buckley, C. and Lewit, A.F. "Optimization of Inverted Vector Searches". Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. 1985.

[21] Stanfill, C. and Kahle, B. "Parallel Free-Text Search on the Connection Machine". Communications of the ACM. Vol. 29, 12 (12/86).

APPENDIX A

ERIK USERS GUIDE

A.1 INTRODUCTION

ERIK is a functional Information Storage and Retrieval (ISAR) system. It provides the ability to create, populate, query and maintain data bases of textual information. Contained within the system are features which are normally found only in experimental ISAR systems. This document, The ERIK User's Guide provides an overview of ERIK, and describes the usage of the system.

A.1.1 INTENDED AUDIENCE

The intended audience of this document is the user and/or administrator of ERIK data bases as well as those who are curious about the operation of that class of data base management systems known as Information Storage and Retrieval Systems.

A.2 OVERVIEW

ERIK is a specialized version of a data base management system. It has been developed and optimized for the storage and manipulation of textual data. Systems of this nature are used to manage the massive amounts of textual information which is generated at an

ever-increasing pace. ERIK consists of a set of seven (7) programs and seven (7) files. The files (where "database" indicates the user specified name of the data base - currently limited to 15 characters in length) and their contents are:

"database".stp - The stopword list. Contains insignificant terms which are NOT to be entered into the data base. Examples are be, to, more, etc. It is a simple text file and may be created by any editor. It ideally contains insignificant terms, in random alphabetical order, one per line. It may be added to or modified at any time. The modified contents will be used at the next usage of the data base population program.

"database".doc - The document vector file. Contains all significant terms contained in the source document with related data for each term. Stored data includes the word and line position of each term in the source document, total number of times the term appears and a frequency weight factor.

"database.off" - The document vector offset file. Contains the offset within the .doc file where a particular documents' vector begins.

"database".trm - The term list file. Contains data which indicates for each term in the data base which documents the term appears in.

"database".idx - The data base index file. Contains each term stored on the system and a pointer to the position in the .trm file where that term's document data is stored.

"database".err - The database error log file. If any software errors occur during the execution of eriki or erikq throughout the life of the data base, they are recorded in this file. The file may simply be "dumped" to the screen. If the file grows too large, it may be deleted and recreated with zero (0) length.

"database".inp - A batch input file. Optional. A simple text file containing the names of document source files which are to be input to the specified data base. Currently source file names are limited to 15 characters in length.

The programs comprising ERIK and their various purposes are:

dbcreate - Creates an ERIK data base. Builds an instance of each of the above files, except the .stp and .inp files.

eriki - Provides the functionality to populate the data base.

erikq - User interface for manipulation of the data base.

idxdmp - Generates a dump of the contents of the data base index file.

trmdmp - Generates a dump of the contents of the data base term file.

docdmp - Generates a dump of the contents of the data base document vector file.

offdmp - Generates a dump of the contents of the data base document vector offset file.

Further specification of the use of each program is provided below.

A.3 ERIK CONSTRAINTS

Information Storage and Retrieval data bases are static by nature. What is meant here is that ISAR data bases tend to grow over time, with little or no deletion or modification to their contents. The primary concern with such systems is their retrieval performance in processing time and the ability to provide results meeting query criteria. As such, systems of this nature are generally multi-user for retrieval purposes only, and single-user for update purposes. ERIK is such a system. Numerous users may access the system for retrieval purposes at any given time. However, only one user should be executing the population program at one time, and that should occur when no other users are accessing the data base in question.

ERIK enables the user to create as many data bases as

he/she may desire. They simply require unique names. Currently, the data base name must be no longer than 15 characters. The only "sizing" limits are the number of terms and lines per document, and the total number of documents per data base. These values are currently limited to 65535. The remaining files may grow as needed to fulfill input requirements.

As mentioned above, data base and input file names are limited to 15 characters in length. As a result, accessing data bases residing in another directory requires the entire path name to fall within the 15 character limit. A compromise is that data base files must all reside within one directory. ERIK programs, when executing, may then assume the files reside within the current working directory.

ERIK currently does not provide the ability to delete documents from a data base. File re-organization utilities are not supplied. Actually, only one file may really be considered a candidate for reorganization - the .trm file.

A.4 ERIK PROGRAMS AND THEIR USAGE

A.4.1 Dbcreate

dbcreate creates the files necessary for an ERIK data base. The command line to execute dbcreate is:

```
dbcreate "database"
```

where "database" is the name of the data base to create, currently limited to 15 characters. If a data base of the same name already exists, dbcreate will notify the user and exit. When creation is complete, the user is notified via a simple message and execution terminates.

A.4.2 Eriki

This program is the primary interface for populating the data base. It is invoked via the following command line:

eriki [bi] "database"

Either 'b' or 'i' must be provided at invocation. A 'b' indicates the desire to perform batch input. The program will open the "database".inp file and assume the contents to consist of document source file names. Upon completion of the processing of each source file, a message is output indicating the success or failure of the system to process the document.

An 'i' indicates that the user wishes to interactively input documents. the program then prompts the user for source document file names, and the type of input processing desired - 'i' or 'd'. An 'i' here (the default) indicates that the user wishes the document to be immeadiately processed. A 'd' implies "diagnostic" processing - the system will read the document and output the processed source on the screen and NOT update the data base. The format of the diagnostic output is:

Header information for the document, followed by output for each term:

term itself, followed by:

term count in document, followed by:

the line and word positional data for the term

example:

NUMBER OF UNIQUE TERMS IN DOCUMENT: 3
TOTAL NUMBER OF TERMS IN DOCUMENT: 5

document 2 1 1 2 4

information 2 1 2 2 3

retrieval 1 3 5

A "term" to ERIK is any sequence of alphanumerics seperated by any non-alphanumerics. A date of the format 12/13/86 would result in three ERIK terms: 12, 13 and 86. Single letter terms are currently retained, unless specified in the stopword list. Therefore "don't" would

yield "don" and "t" as terms. As terms are input, they are conflated and/or truncated. Conflation yields terms that have been reduced to their mophological root. Terms such as computation, compute, computable will be reduced to a common stem in order to "broaden" the term and increase the number of hits in the data base during queries. Terms exceeding 15 characters in length, after conflation, are truncated to that length.

As a result of processing a source file, the system will rename the source file using a format of:

X.txt

where X inplies a number from 1 - 65535. This enables the system to track the document for later processing, such as accessing the source for query purposes if necessary. When the processing of any source is complete, the system reports the success of the processing and the new name of the source file.

If, at any time during its' processing this program encounters a serious error, it reports that error in the "database".err file and exits gracefully. A "traceback" of where the program was in its' processing when the error occurred is provided to enable rapid determination of the exact location of the error.

A.4.3 Erikq

This program is the user interface for performing queries against the data base and as such provides the majority of the functionlity supplied by the system. The program is invoked by a command line like:

```
erikq "database" [session file name]
```

where session file name is an optional parameter that indicates the name of a file that contains the user dialog from a previous query session.

When a user enters the program, the following information is presented:

available commands: dictionary feedback help limit print query review trace view enter 'help' for more detailed descriptions for each command

A simple 'help' facility is provided that outputs some text on the screen for each of the available commands. Each command may be indicated by simply entering the first character of the desired command. The commands, and what they accomplish are:

A.4.3.1 Dictionary -

This command allows the user to see what terms are currently indexed in the data base. The user may search for a specific term, or they may input two terms, and the system will output all terms within the data base that are within the range delimited by the terms. Remember that terms are conflated in the data base. This routine conflates the user input and then performs the searches.

A.4.3.2 Feedback -

This command gives the user access to the relevance feedback features of the system. ERIK provides three types of relevance feedback - positive feedback, negative feedback and a combination of the two. The user is prompted to input the numbers of the previous query to be modified and re-submitted to the query processor. If the indicated query is not in an acceptable range, the user is notified. Relevance feedback only makes sense when processing queries containing vectors. The system checks to ensure that the query to be re-submitted did, indeed, contain a vector before continuing processing.

The user is then asked to input the number of a document that they considered relevant to the initial query.

Obviously this document identifier should come from the results list provided from the initial query - the system does not check to ensure this. If the user does provide input, they are asked if they wish to provide a multiplier that will be applied to the weights of each of the documnet vector terms as the modified query is generated. This multiplier defaults to one (1). Whether or not the user provided input to the first prompt(s), they are asked to indicate a document that was non-relevant to the first query. Again, if they provide input they are prompted for a multiplier value. The multiplier again defaults to one (1). If the user solely input a relevant document identifier, positive feedback occurs and a new query is generated and submitted to the query processor. If the user solely input a non-relevant document identifier, negative feedback occurs and a new query is generated and submitted to the query processor. If the user provides both, a combination of positive and negative feedback occurs.

A.4.3.3 Help -

This functionality provides the simple help system supplied. It requests the command that the user wishes help on, and accesses the help file and dumps the text found there for the command onto the screen. It stays

within its' processing loop until a carriage return is hit.

A.4.3.4 Limit -

This enables the user to limit the size of the results list reported for a processed query. This is particularly useful for queries containing vectors which get ranked and reported in ranked order. The user may say they wish to see only the 10 top ranked documents.

A.4.3.5 Print -

This allows a user to direct the text or results of a previous query to a file (they are prompted for a name) such that upon leaving the system they may print or otherwise peruse the contents.

A.4.3.6 Query -

The session command processor defaults to this command. It places the user in the parser input routine and accepts the user query. The user must terminate his/her query with a period (.) followed by a carriage return.

If the user wishes to abort the current query and remain in the parser, they may enter a '' followed by a carriage return. If they wish to return to the command processor, they may enter 'exit' anywhere within a query (NOTE: this implies queries should NOT contain the word 'exit'!!). Acceptable query terms are sequences of alphanumerics delimited by white space (line feeds, tabs, spaces). The parser is an operator left precedence parser that has been modified to accept an enhanced retrieval query set. Acceptable queries may contain the following clauses and operators (note that all query input should be in lowercase):

Operators:

and - Boolean AND

or - Boolean OR

w X - word adjacency. The X indicates the number of words that the operands of this operator must be within one another to meet the criteria. Ordering is maintained. That is, operand 1 MUST appear first, and occur within X words of operand 2.

l X - line adjacency. The X indicates the

number of lines that the operands of this operator must be within one another to meet the criteria. Ordering is maintained. That is, operand 1 MUST appear first, and occur within X lines of operand 2.

For user interest, the adjacency operators possess the highest precedence.

() - left and right parentheses. These act as would be expected to indicate a higher precedence of those clauses appearing within their range.

Clauses:

strings - delimited by " and ". The query processor will utilize the Boyer-Moore string searching algorithim and open documnet source files searching for a single occurance of that string.

vectors - delimited by { and }. Vectors consist of optionally weighted terms separated by white space. The weights are indicated within [and]. Weights may be negative and

may contain decimal points. Weights default to one (1).

An example would be:

{document[.23] information[-0.89] retrieval}.

Queries may consist of mixtures of operators and clauses with some restrictions.

Those resrictions are:

strings - Only one string may be input per query. It must appear at the end of the query and be preceeded by an 'and' operator. This enables the query processor to subset the number of document source files it must access to improve performance.

terms - Terms must be acted upon by operators (except terms within a vector). It is not possible to simply list terms. A query MAY consist of a single term followed by a period (.).

parentheses - Only the Boolean operators may act upon clauses/terms delimited by parentheses. Adjacency

operators will not be accepted.

vectors - Only one vector may be entered per query. It may not be the only entity delimited by parentheses. Vectors may not be the LAST entity delimited by parenstheses.

The user is encouraged to experiment with the various combinations made possible by the operators, clauses and restrictions indicated above. A large number of query formats are possible. The parser will tell the user what it doesn't like and the user may try again.

A.4.3.7 Review -

This simply reports the number of queries that have successfully been processed so far.

A.4.3.8 Trace -

This enables the user to follow the processing of the query and its' subsequent evaluation. The parser reports the results of its' operation, and the evaluator reports the number of documents that each unique term resides in

as the query is evaluated.

A.4.3.9 View -

This enables the user to view on the screen the text or results of a previous query, or the text of a source document file.

As was implied in the descriptions above, any user query that parsed sucessfully is saved in a session file, followed by the results of the evaluation of that query. When the user exits the session, the file may be saved for continuation of that retrieval session at a later date. An indicator is placed in the saved file to indicate that it is an accepatble session file. If the system, upon executing with an old session file indicated dtermines that the file wasn't an ERIK session file, erikq will exit.

If, at any time during its' processing this program encounters a serious error, it reports that error in the "database".err file and exits gracefully. A "traceback" of where the program was in its' processing when the error occurred is provided to enable rapid determination of the exact location of the error.

The following programs are general maintenance/utility programs. They enable the curious user or system administrator to view the contents of each of the data base files. The insight gained from their output may be must to further understand the workings of the system and/or aid in detecting file corruption.

A.4.4 Idxdmp

This program generates a dump of the contents of a data base index file. An ERIK index file comprises a B+ - tree index. The data generated includes administrative information such as number of levels to the B+ - tree, file offset to the root page, etc. Following that are the contents of each page in sequential order. Header data for each page is dumped, followed by the terms and file offsets. If the page is an index page (upper level of the tree) the offsets point to child pages within the index file. If the page is a data page (leaf node of the tree), the offsets are pointers into the .trm file for the related term. Right sibling data contained in the data pages is used for sequential scans of the indexed terms (why it's a B+ - Tree). Left sibling pointer data may be used for file reconstruction in the event of file corruption.

The program is invoked with the following command line:

```
idxdmp "database"
```

example:

(Note that the total number of terms indexed for this database doesn't exceed the capacity of a single B - tree node page)

```
ADMINISTRATIVE DATA FOR FILE: eric1.idx
OFFSET TO ROOT PAGE: 1024
OFFSET TO SEQUENTIAL SCAN PAGE: 1024
OFFSET TO NEXT AVAILABLE PAGE: 0
NUMBER OF TREE LEVELS: 1
OFFSET TO TREE LEVEL 0: 0
OFFSET TO TREE LEVEL 1: 0
OFFSET TO TREE LEVEL 2: 0
OFFSET TO TREE LEVEL 3: 1024
NUMBER OF PAGES IN FREE CHAIN: 0


ADMINISTRATIVE DATA FOR PAGE: 1
SCANNED PAGE OFFSET IN FILE: 1024
STORED PAGE OFFSET IN FILE: 1024
OFFSET TO LEFT SIBLING: 4294967295
   (-1 printed as an unsigned value)
OFFSET TO RIGHT SIBLING: 4294967295
   (-1 printed as an unsigned value)
PAGE TYPE (INDEX or DATA): 1
   (1 indicates DATA page)
TREE LEVEL OF PAGE: 3
NUMBER OF ENTRIES IN PAGE: 14


bsrch  198
btre  396
c86s2  594
confl  792
d  990
dbopen  1188
```

```
erik  1386
front  1584
input  1782
loger  1980
outtre  2178
termcod  2376
testtre  2574
updat  2772
```

A.4.5 Trmdmp

This program dumps the contents of the .trm file. It outputs some file administrative data and then reads each record sequentially and outputs their contents. Each record contains "buckets" holding data about which document the various terms reside in. The number of buckets per record is fixed for performance reasons, so the records are chained if a term resides in a number of documents that is greater than the number of buckets per record.

example:

```
ADMINISTRATIVE DATA FOR FILE: eric1.trm
NUMBER OF TERMS IN FILE: 14
NUMBER OF USED RECORDS: 14
NUMBER OF CONTINUATION RECORDS: 12067
   (nonsense value for now)
NUMBER OF REMAINING RECORDS: 65520
   (nonsense value for now)
OFFSET TO NEXT AVAILABLE RECORD: 198
```

```
ADMINISTRATIVE DATA FOR RECORD: 1
RECORD OFFSET IN FILE: 198
TERM ENTRY: bsrch
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
   (0 since   buckets not exceeded yet)
CONTINUATION RECORD POINTER: 0

(1 says "resides in document 1, 40 is .doc file
offset position for document 1. 82 is the .doc
file offset position within document 1
where this terms' positional data begins)


1 40 82
2 502 544
3 964 1006



ADMINISTRATIVE DATA FOR RECORD: 2
RECORD OFFSET IN FILE: 396
TERM ENTRY: btre
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 112
2 502 574
3 964 1036



ADMINISTRATIVE DATA FOR RECORD: 3
RECORD OFFSET IN FILE: 594
TERM ENTRY: c86s2
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 142
2 502 604
3 964 1066



ADMINISTRATIVE DATA FOR RECORD: 4
RECORD OFFSET IN FILE: 792
TERM ENTRY: conf1
```

```
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 172
2 502 634
3 964 1096



ADMINISTRATIVE DATA FOR RECORD: 5
RECORD OFFSET IN FILE: 990
TERM ENTRY: d
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 202
2 502 664
3 964 1126



ADMINISTRATIVE DATA FOR RECORD: 6
RECORD OFFSET IN FILE: 1188
TERM ENTRY: dbopen
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 232
2 502 694
3 964 1156



ADMINISTRATIVE DATA FOR RECORD: 7
RECORD OFFSET IN FILE: 1386
TERM ENTRY: erik
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 262
2 502 724
3 964 1186
```

```
ADMINISTRATIVE DATA FOR RECORD: 8
RECORD OFFSET IN FILE: 1584
TERM ENTRY: front
NUMBER OF ENTRIES IN RECORD: 5
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 292
2 502 754
3 964 1216
4 1426 1498
5 1588 1660



ADMINISTRATIVE DATA FOR RECORD: 9
RECORD OFFSET IN FILE: 1782
TERM ENTRY: input
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 322
2 502 784
3 964 1246



ADMINISTRATIVE DATA FOR RECORD: 10
RECORD OFFSET IN FILE: 1980
TERM ENTRY: loger
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 352
2 502 814
3 964 1276



ADMINISTRATIVE DATA FOR RECORD: 11
RECORD OFFSET IN FILE: 2178
TERM ENTRY: outtre
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 382
```

```
2 502 844
3 964 1306


ADMINISTRATIVE DATA FOR RECORD: 12
RECORD OFFSET IN FILE: 2376
TERM ENTRY: termcod
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 412
2 502 874
3 964 1336


ADMINISTRATIVE DATA FOR RECORD: 13
RECORD OFFSET IN FILE: 2574
TERM ENTRY: testtre
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 442
2 502 904
3 964 1366


ADMINISTRATIVE DATA FOR RECORD: 14
RECORD OFFSET IN FILE: 2772
TERM ENTRY: updat
NUMBER OF ENTRIES IN RECORD: 3
CONTINUATION RECORD NUMBER : 0
CONTINUATION RECORD POINTER: 0


1 40 472
2 502 934
3 964 1396
```

A.4.6 Docdmp

This program dumps the contents of the .doc file. It first outputs file administrative data, followed by the contents of each document vector. The output for each vector consists of vector administrative data followed by the following for each term in the document (in alphabetic order):

term itself

term document weight

term count in document

repeating instances of line and word positional data (term count times)

example:

(You'll note that the same document was indexed three times)

```
ADMINISTRATIVE INFORMATION FOR FILE: eric1.doc

NUMBER OF VECTORS IN FILE: 3
TOTAL NUMBER OF TERMS IN FILE: 50
FILE OFFSET TO WRITE NEXT VECTOR: 1426
```

```
ADMINISTRATIVE INFORMATION FOR VECTOR: 1

FILE OFFSET FOR VECTOR: 40
TOTAL NUMBER OF TERMS IN VECTOR: 14
NUMBER OF UNIQUE TERMS IN VECTOR: 14

(term  weight count line word)

bsrch  0.071  1   1 7

btre  0.071  1   1 5

c86s2  0.071  1   5 14

confl  0.071  1   2 12

d  0.071  1   5 13

dbopen  0.071  1   2 10

erik  0.071  1   1 6

front  0.071  1   1 1

input  0.071  1   1 2

loger  0.071  1   1 8

outtre  0.071  1   1 3

termcod  0.071  1   2 9

testtre  0.071  1   2 11

updat  0.071  1   1 4

ADMINISTRATIVE INFORMATION FOR VECTOR: 2

FILE OFFSET FOR VECTOR: 502
TOTAL NUMBER OF TERMS IN VECTOR: 14
NUMBER OF UNIQUE TERMS IN VECTOR: 14


bsrch  0.071  1   1 7

btre  0.071  1   1 5

c86s2  0.071  1   5 14

confl  0.071  1   2 12

d  0.071  1   5 13
```

```
dbopen  0.071  1   2 10

erik  0.071  1   1 6

front  0.071  1   1 1

input  0.071  1   1 2

loger  0.071  1   1 8

outtre  0.071  1   1 3

termcod  0.071  1   2 9

testtre  0.071  1   2 11

updat  0.071  1   1 4

ADMINISTRATIVE INFORMATION FOR VECTOR: 3

FILE OFFSET FOR VECTOR: 964
TOTAL NUMBER OF TERMS IN VECTOR: 14
NUMBER OF UNIQUE TERMS IN VECTOR: 14


bsrch  0.071  1   1 7

btre  0.071  1   1 5

c86s2  0.071  1   5 14

confl  0.071  1   2 12

d  0.071  1   5 13

dbopen  0.071  1   2 10

erik  0.071  1   1 6

front  0.071  1   1 1

input  0.071  1   1 2

loger  0.071  1   1 8

outtre  0.071  1   1 3

termcod  0.071  1   2 9

testtre  0.071  1   2 11

updat  0.071  1   1 4
```

A.4.7 Offdmp

This program dumps the contents of the data base .off file. It simply reads records sequentially and outputs their contents (document vector number and .doc file offset).

example:

```
DUMP OF DATA BASE FILE: eric1.off
document ID  .doc file offset

        1              40
        2             502
        3             964
```