FAST-INV:
A Fast Algorithm for Building Large Inverted Files

By Edward A. Fox and Whay C. Lee

TR 91-10

FAST-INV:
A Fast Algorithm for
Building Large Inverted Files*

Edward A. Fox and Whay C. Lee

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg VA 24061-0106

ABSTRACT

Inverted files are widely used in building bibliographic and other types of retrieval systems. In order to investigate the utility of advanced information retrieval methods for improving access to large online library catalogs, it was necessary to extend the SMART system in a variety of ways. One particular problem was to develop a fast method to produce an inverted file from hundreds of thousands of (partial) MARC records. The FAST-INV software was developed in 1986, taking advantage of the large primary memories available on modern computers and the order inherent in the input data. Using the new algorithm, processing in primary memory for N basic data elements has time complexity O(N), and processing of files that will not fit in primary memory can be accomplished in a fixed number of passes. Performance studies show this approach to be (at least) an order of magnitude faster than commonly used techniques. It is hoped that these findings will be of interest to database providers and will help them reduce costs relating to the building of inverted files, as we have been doing for the last five years.

CR Categories and Subject Descriptors: E.2 [**Data Storage Representations**]: *Inverted file representations;* H.2.2 [**Database Management**]: Physical Design - *access methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage - *file organization*

General Terms: Algorithms, Performance.

Additional Keywords and Phrases: Access methods, CD-ROMs, document retrieval, indexing, information retrieval, inverted files.

 Project was funded in part by grants from the National Science Foundation (IST-8418877) and the Virginia Center for Information Technology (INF-85-016), and aided by an AT&T Equipment Donation.

1. Introduction*

Inverted files are the most commonly used means for providing rapid access to large bibliographic and other types of databases. In the last thirty years, the number of electronic databases has steadily grown, with over 3000 publicly available by the mid 1980's [WILL 85a]. These computer readable databases cover a wide variety of subjects, and provide references on demand to vast stores of published literature [WILL 85b].

Access to that information, heretofore only possible through the aid of (networked) connection to the computing equipment of large service companies, can now be accommodated on personal computers as well, because of the revolution in optical storage devices [FUJI 84] and advances in magnetic storage units. These developments has long been of interest to information professionals [GOLD 84]. They provide an opportunity to some researchers to ensure that the advances made by the information retrieval research community may become an integral part of new systems [FOXE 86a]. Though extravagant claims have been made regarding the effect of CD-ROM technology [ADAM 85], it is already the case that thousands of large databases have been published to fit in the more than 550 million bytes of storage space available on each CD-ROM.

It is necessary to build inverted files for most databases, whether managed by a service company or mastered onto a CD-ROM. Commonly used textbooks in the area of information storage and retrieval (e.g., [HEAP 79], [VANR 79], [SALT 83]) do discuss inverted files, but provide few insights regarding how to efficiently build them. While service companies no doubt have their own (proprietary) algorithms, it is now also of value to those considering the expenses of pre-mastering a database for CD-ROM to learn of efficient methods for building inverted files.

2. Background

This research has been carried out as part of a larger effort to investigate the utility of advanced information retrieval methods in the context of an online public access library catalog. The SMART experimental retrieval system had to be adapted to handle a very large subset of the full collection of records that is managed by VTLS Inc. software to serve the needs of the Virginia Tech library.

2.1 Purpose of library study

There have been a number of investigations regarding the use of online public access catalogs in libraries [COCH 83]. Since there are so many different types of people using those catalogs, it may be especially important to adapt such systems to individual characteristics [BORG 85]. New designs should be considered, to better serve the needs of library users by applying additional sources of knowledge and expertise, and by making current software more intelligent [BATE 86]. Since Boolean logic query forms are used in many catalog systems, either directly or indirectly, it seemed appropriate to investigate the effect of extended Boolean processing methods, including the use of "p-norm" queries, automatic Boolean query construction, and Boolean feedback [FOXE 83a]. A front end system to VTLS was developed to help with initial studies [FOXE 86b]. However, more thorough testing required changing the retrieval host system as well.

---

*This paper was written in 1986. Because of interest in our work, with only slight modifications, we are now releasing that unpublished manuscript as a technical report. See also related discussion in a chapter entitled "Inverted Files" by Harman, Fox, and Baeza-Yates, in a new text on information retrieval algorithms edited by Frakes and Baeza-Yates, published by Prentice-Hall.

2.2 Use of SMART

Data from the Virginia Tech library computer, which runs VTLS Inc. software, was dumped in 1986 so that an inverted file could be built for use with the VTLS routines. That data was made available, and the SMART retrieval system was chosen to handle the data for experimental studies. Based on prior involvement with the implementation of SMART [FOXE 83b], the decision was made to use the currently released form [BUCK 85a] and to then make enhancements. Efforts commenced to develop a front end to SMART that would be suitable for running experiments with Virginia Tech students enrolled in freshman English.

Since the normal routines used in SMART are geared to expect small to medium size collections, a number of other modifications were needed as well. Inverted files were previously built and accessed in primary memory, so changes were made to allow the inverted file to be stored on secondary memory and for parts to be read in only as necessary. This approach makes optimization of use of inverted files [BUCK 85b] even more important, but before facing that problem it was necessary to build the large inverted file of library data in the first place.

3. Inverted Files and Related Methods

A library patron will often want to obtain a number of books, journals, or other items based on partial knowledge. That is, given one or more items of information regarding the author(s), title, subject, or call number, it is necessary to identify catalog entries that relate. Inverted files can help with this process, but before focussing on them it is appropriate to consider some alternate schemes.

3.1 Alternative approaches

Searching and sorting problems are involved in numerous uses of computers. However, most of the attention given to this topic has until recently dealt with providing access based on one "key" attribute (at a time) [KNUT 73]. There are many data structures and algorithms that have been developed for this purpose [GONN 84]. For example, B-trees [BAYE 72] have been widely used in a variety of adaptations [COME 79] for dictionary access problems and other applications.

If one has knowledge about the relative importance of each of the classes of attributes, such as when the objects being searched for have some special characteristics that suggest certain properties will be identified before others, multiple attribute tree schemes may be of particular value [RAOS 85]. Superimposed coding methods may also be of use when partial knowledge is available, such as to help with matching in a Prolog database [SACK 85]. To handle text files, this approach is one that may deserve particular attention in a variety of situations [FALO 85]. For special processing needs, such as range searching, several data structures can be used, and each have their own set of costs, advantages, and disadvantages [BENT 79].

Hardware advances provide a new set of alternatives to address the problem of finding items based on partial knowledge in very large collections [HASK 80]. Experimental systems have been developed to compare the utility of differing approaches [HOLL 85]. One particular architecture, that of using massively parallel systems, has been shown to perform very well [STAN 86], though very expensive equipment is needed to carry this out.

3.2 Indexing and inversion

Given a raw collection of documents, it is necessary to process the given text in several stages in order to obtain an inverted file. In the SMART system, documents are often provided in

a short form, with title, author, abstract, and descriptors serving as a surrogate for the full text. Automatic indexing procedures build a terse representation of each document as a "vector" of terms with associated weights [SALT 83]. Words and phrases in the text (that are not on a "stop-word" list of common words) are entered into a system dictionary, usually by hashing, and thereby associated with a "concept number." The current SMART system allows separate handling of different "concept types" such as author or title word or bibliographic citation [FOXE 83a, b, c], but this refinement can be ignored in the present discussion. Similarly, to simplify the explanation, the handling of weights associated with terms in documents will be generally ignored.

For an online catalog system, where a small number of access points are usually available, the searchable representation of a document is often limited to the title, author(s), Library of Congress subject headings, and the call number. For a bibliographic collection, a longer representation results due to the presence of abstracts. For full text collections, even longer document vectors are common. The size of the corresponding inverted file grows according to the amount of information recorded per document.

If one associates a dimension in a T-dimensional space with each of the T terms present in the collection, then the N documents in a collection can be represented as an NxT matrix. While conceptually clear, however, this matrix is impractical to use with N on the order of thousands or millions, and T on the order of tens or hundreds of thousands. Typically, most entries of the matrix will be zero, so it is wise to store the information in a more compressed form. Thus, the document file can be built from the full matrix by omitting zeroes from each row, and the inverted file can be built by omitting zeroes from columns.

To build an inverted file from the Virginia Tech library data, one can first index the documents to obtain document vectors, and then invert the resulting document file. This process is shown in Figure 1 for a small example collection that will be referred to in later figures as well. A document is simply represented as a list of concept numbers, while the inverted file is represented as a list of document numbers associated with each concept. Since the (hashed) dictionary may have gaps, there may be unused concept numbers, so the inverted file may also have gaps between successive concept number entries.

In SMART, a hashed dictionary and document vector file are built directly from a text collection. Usually, after stop words are removed, the remaining words (in text sections) are stemmed, using a fast implementation of an extension of the algorithm given in [LOVI 68]. Since documents are not very long, placing the concepts of a document in a sorted order can be done in primary memory in connection with improving efficiency of looking up the concepts in the dictionary. The size of the hashed dictionary is often estimated to be about twice the number of concepts expected, so the range of concepts is typically in the tens of thousands, even for fairly large collections. The high concept number (HCN) in the dictionary is then an upper bound on the number of concepts in the inverted file.

4. File Inversion

Little has been published regarding algorithms for building inverted files. Since inverted files for CD-ROMs are static, and since inverted files for online databases can be loaded from static files, issues relating to updating or dynamic data structures for inverted files will be ignored. As can be seen in Figure 2, constructing an inverted file from a document file can be viewed as a special type of sorting process. Whereas formerly entries were ordered first on document and second on concept number, sorting reverses the importance of the keys so that concept number is primary and document number is secondary. It is interesting to look at and compare the different approaches to file inversion used in some well-known experimental information retrieval systems.

Fig. 1 INDEXING & INVERSION

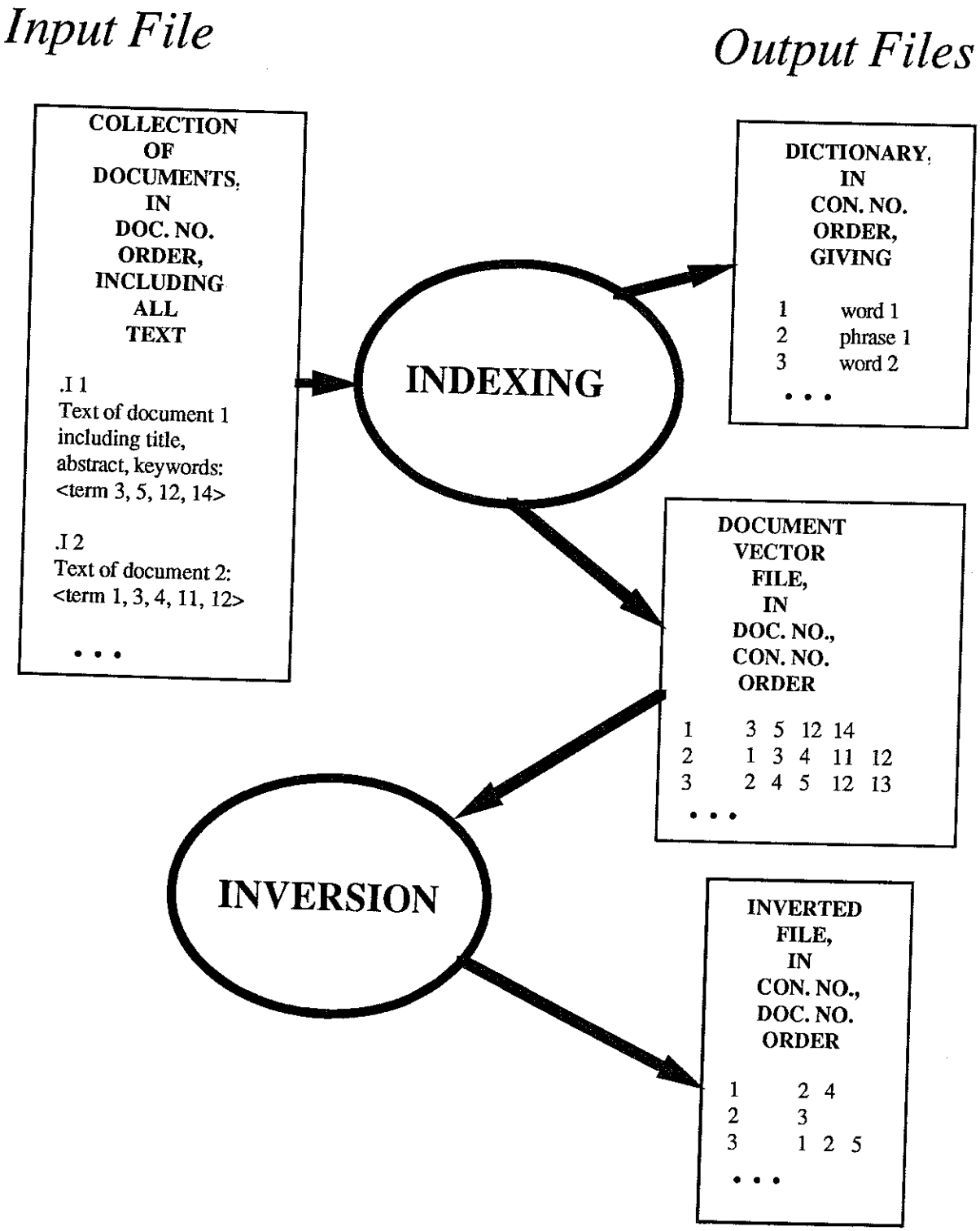
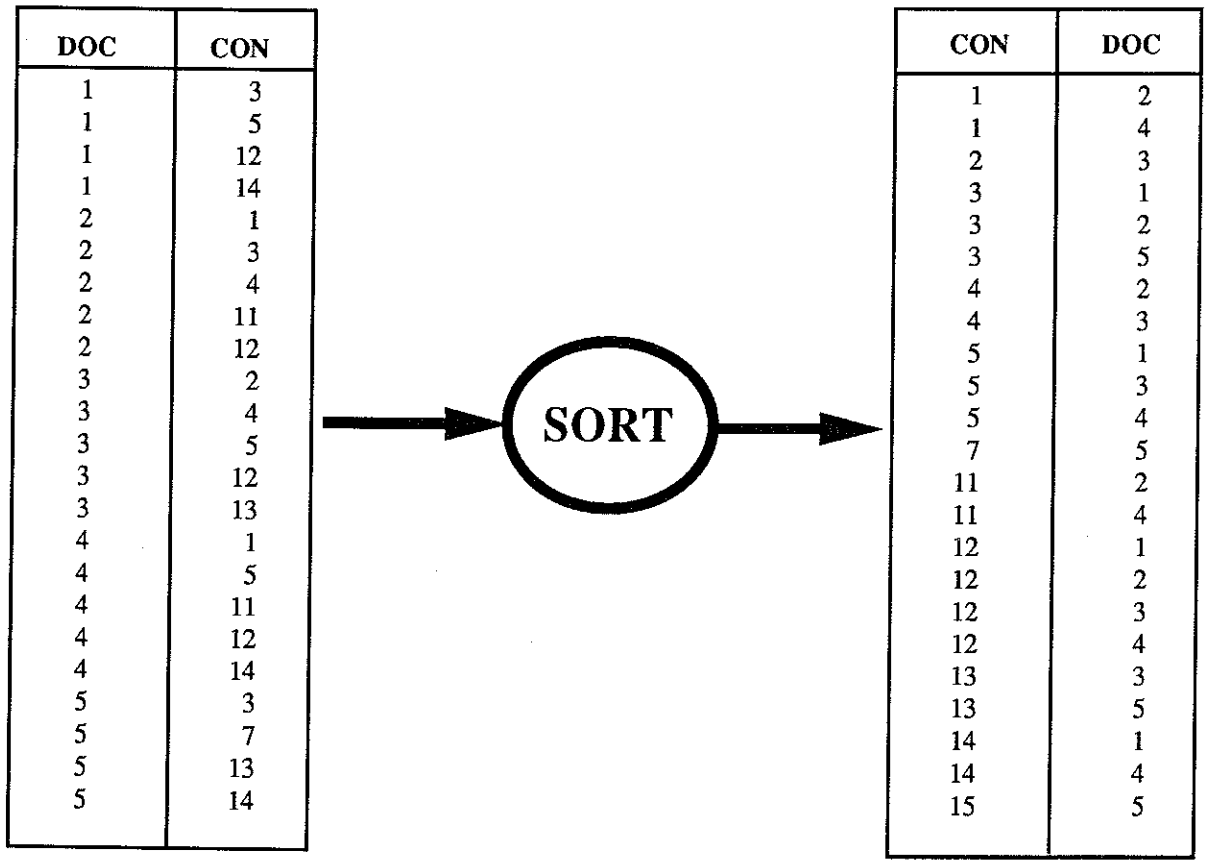


Fig. 2 RELATION INVERSION PROCESS

Input File *Output File*



4.1 SIRE

The SIRE experimental system was developed to allow the integration of advanced retrieval methods, such as similarity computations and ranking of the retrieved set, with inverted file methods [NORE 77]. A commercial version has been marketed for several years, and has been used to show that practical enhancements to Boolean retrieval systems can be made [FOXE 88]. The commercial version of SIRE was programmed in the C language and ran on a variety of computers. So that SIRE (™ of KNM, Inc.; later Personal Librarian from Personal Library Software Inc.), can function on personal computers as well as mainframe computers, standard system sorts were used to build the dictionary and inverted file. While this makes the software easy to port, it leads to considerable expense when a large inverted file must be built.

4.2 SMART

Since the collections heretofore used with SMART typically included no more than about 40,000 documents, and since the software runs on computers running some version of Berkeley UNIX (™ of AT&T Bell Laboratories), it was assumed that inverted files could be built and accessed in (virtual) memory. To understand this process it is necessary to be aware of the data structures involved.

Since there are often a number of concepts for each document and a number of documents that contain each concept, it is inefficient to normalize either the document or the concept files. Thus, instead of the relational form shown in Figure 2, a scheme with pointers can be used as can be seen in the top portion of Figure 3. The pointers are actually implemented using byte offsets and/or counts stored with the pointing (or "offset") file, as shown on the bottom half of the figure. In any case, the pointing files are small compared to the pointed to files (especially since weights are stored in the latter). In the subsequent discussion, the four files involved, considered from left to right in the top part of Figure 3, will be referred to as: docptr, conlist, conptr, and doclist.

In SMART, doclist is built during inversion as a collection of linked lists. The document vector file is read in sequential order, using docptr and conlist, giving values as would be obtained by sweeping through the file on the left half of Figure 2. For each new <doc#,con#> pair, the linked list for that con# is accessed, and the doc# (and weight) is inserted.

4.3 Complexity of inversion

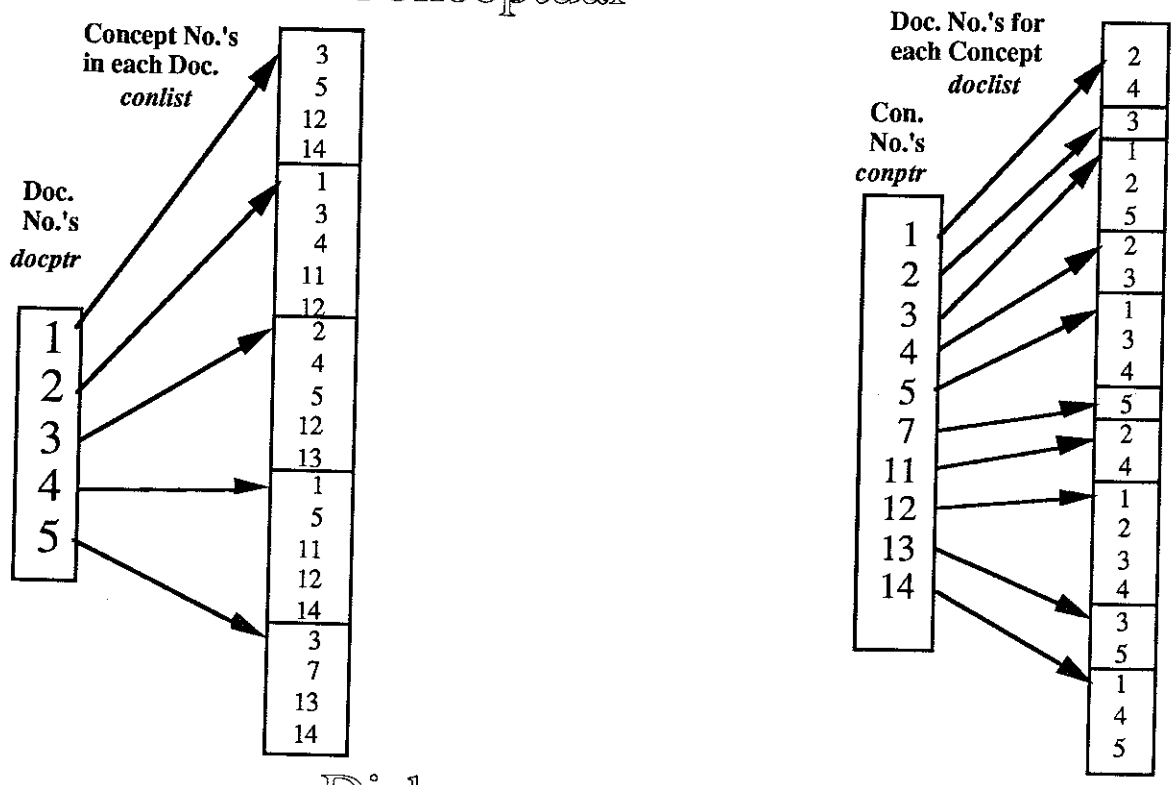
The cost of inversion can be measured in terms of a variety of resources involved. The SIRE algorithm uses a disk sort, so I/O operations are significant. The SMART algorithm uses primary memory for large files, so page faults can be expensive. To see this more clearly some definitions and equations are necessary.

Let N = number of documents, often in the thousand to million range
T = number of terms, often in the tens of thousands
HCN = high concept number in dictionary
L = number of document/concept (or concept/document) pairs in the collection
k = average number of terms per document
c = average number of documents per term
M = available primary memory size, in bytes

Fig. 3 SMART INVERSION PROCESS

Input Files *Output Files*

Conceptual



Disk Representation

Doc. No.	Off-set	No. Con.
1	0	4
2	4	5
3	9	5
4	14	5
5	19	4

Con. No.	Off-set.	No. Docs.
1	0	2
2	2	1
3	3	3
4	6	2
5	8	3
7	11	1
11	12	2
12	14	4
13	18	2
14	20	3

Then

HCN = high concept number in dictionary, usually < 2 * T
L << N*T
|docptr| = N
|conptr| = T
|conlist| = |doclist| = L
k = L/N
c = L/T

When N >> T, as is the case for large collections,
L = k*N >> T
c >> k

Based on these definitions, it can be seen that the cost of sorting with SIRE is O (L log L), which can be very expensive, especially when heavy use of disks is made during the sort.

Regarding the SMART approach, the cost is based on several components:

Comparisons
$L * c = L^2 / T$
Page Faults
L, in the worst case, since when L >> M, a page fault may be needed for each insertion

Thus, both the SIRE and SMART methods of building an inverted file can be very expensive.

5. Fast Inversion Algorithm

In order to build an inverted file using the Virginia Tech library data, it was necessary to extend the SMART system and to use an approach that did not require the entire inverted file to fit into (virtual) memory. Two insights have enabled the development of a very fast inversion algorithm, which is called FAST-INV:

1. use the large primary memories available on today's computers
2. use the inherent order of the input data to avoid sorting

The first principle is important since personal computers with 4 Mbytes of primary memory are common, now that a megabyte can be obtained for less than $100, and since mainframes may have more than 100 Mbytes of memory. Even if databases are on the order of 1 Gbyte, if they can be split up into memory loads that can be rapidly processed and then combined, the overall cost will be minimized.

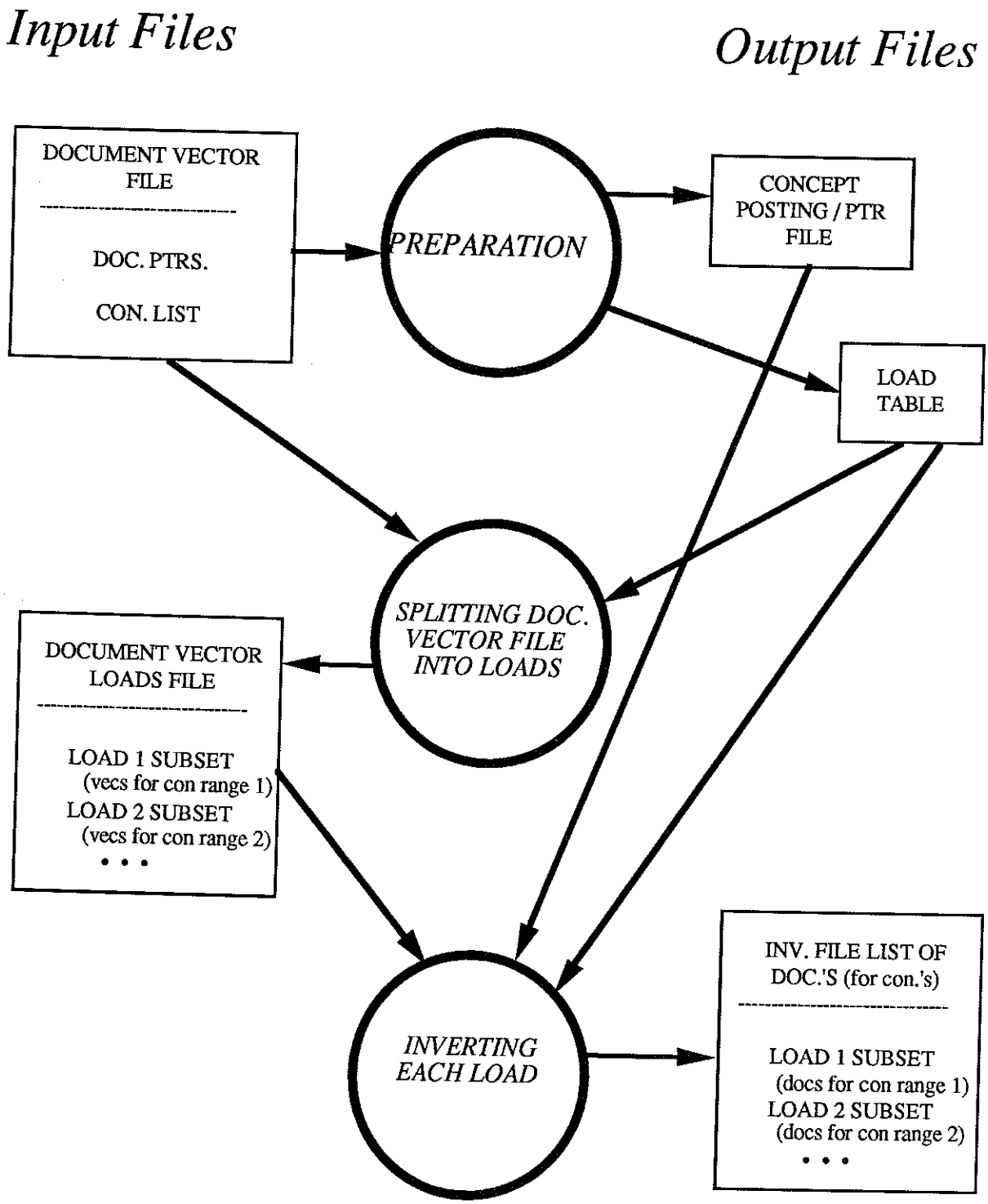
The second principle is crucial since with large files it is very expensive to use polynomial or even N log N algorithms. These costs are further compounded if principle 1 is not followed, since then the cost is of disk operations.

The FAST-INV algorithm follows these two principles to use primary memory in a close to optimal fashion, by processing the data in three passes. The overall scheme can be seen in Figure 4.

5.1 Preparation

In one pass, the document file can be read and two special files produced: conptr and a load table. It is assumed that M >> HCN, so the conptr file eventually needed can be built in primary

Fig. 4 FAST-INV USE OF LOADS



memory. However, it is assumed that M < L, so several primary memory loads will be needed to process the document data. Because entries in the doclist file will be grouped on concept number, with those concepts in ascending order, it is appropriate to see if the conlist data can be somehow transformed beforehand into j parts such that:

- L / j < M, so that each part will fit into primary memory
- HCN / j concepts, approximately, are associated with each part

This allows each of the j parts to be read into primary memory, inverted there, and the output to be simply appended to the (partially built) doclist file.

Specifically, preparation involves the following:

1. Allocate an array, con_entries_cnt, of size HCN, initialized to zero.
2. For each <doc#,con#> entry in the document vector file:
   Increment con_entries_cnt[con#].
3. Use the just constructed con_entries_cnt to create a disk version of conptr.
4. Initialize the load table.
5. For each <con#,count> pair obtained from con_entries_cnt:
   If there is not room for documents with this concept to fit in the current load,
   then create an entry in the load table and initialize the next load entry;
   otherwise update information for the current load table entry.

At this time, after one pass through the input, the conptr file has been built and the load table needed in later steps of the algorithm has been constructed. Note that the test for room in a given load enforces the constraint that data for a load will fit into available memory. Specifically:

Let LL = length of current load (i.e., number of concept/weight pairs)
S = spread of concept numbers in current load (i.e., end concept - start concept +1)
8 bytes = space needed for each concept/weight pair
4 bytes = space needed for each concept to store count of postings for it

Then the constraint that must be met for another concept to be added to the current load is

$$8 * LL + 4 * S < M$$

5.2 Splitting document vector file

The load table indicates the range of concepts that should be processed for each primary memory load. There are two approaches to handling the multiplicity of loads. One approach, which is currently used, is to make a pass through the document vector file to obtain the input for each load. This has the advantage (needed in the context of our library experiment) of not requiring additional storage space (though that can be obviated through the use of magnetic tapes), but has the disadvantage of requiring expensive disk I/O.

The second approach is to build a new copy of the document vector collection, with the desired separation into loads. This can easily be done using the load table, since sizes of each load are known, in one pass through the input. As each document vector is read, it is separated into parts for each range of concepts in the load table, and those parts are appended to the end of the corresponding section of the output document collection file. With I/O buffering, the expense of this operation is proportional to the size of the files, and essentially costs the same as copying the file.

5.3 Inverting Each Load

When a load is to be processed, the appropriate section of the conptr file is needed. An output array of size equal to the input document vector file subset is needed. As each document vector is processed, the offset (previously recorded in conptr) for a given concept is used to place the corresponding document/weight entry, and then that offset is incremented. Thus, the conptr data allows the input to be directly mapped to the output, without any sorting. At the end of the input load the newly constructed output is appended to the doclist file.

5.4 An Example

Figure 5 illustrates FAST-INV processing of the sample data previously shown in Figures 1 through 3. The document vector input files (docptr, conlist) are read through once to produce the concept postings/pointers file (stored on disk as conptr) and the load table. Three loads will be needed, for concepts in ranges 1-4, 5-11, and 12-14. There are 10 distinct concepts, and HCN is 14.

The second phase of processing uses the load table to split the input document vector files and create the document vectors loads files. There are three parts, corresponding to the three loads. It can be seen that the document vectors in each load are shortened since only those concepts in the allowable range for the load are included.

The final phase of processing involves inverting each part of the document vectors loads file, using primary memory, and appending the result to the inverted file. The appropriate section of the conptr file is used so that inversion is simply a copying of data to the correct place, rather than a sort.

6. Performance Results

Based on the discussion above, it can be seen that FAST-INV is a linear algorithm in the size of the input file, L. The input disk files must be read three times, and written twice (using the second splitting scheme). Processing in primary memory is limited to scans through the input, with appropriate (inexpensive) computation required on each entry.

Thus, it should be obvious that FAST-INV should perform well in comparison to other inversion methods, such as those used in the SIRE and SMART systems (see section 4 above). To demonstrate this fact, a variety of tests were made.

6.1 Preliminary runs

When the library catalog data was not yet available, and since other test collections had been used with the regular versions of SIRE and SMART, it was decided to use the CACM and INSPEC collections (see explanation of their history in [FOXE 83a, Appendix C] and [FOXE 83c]). Statistics about these collections are given in Table 1. It can be seen that the CACM collection is roughly 1/4 of the size of the INSPEC collection, except that the CACM dictionary is 1/2 the size of the INSPEC dictionary (because authors are included in CACM but not in INSPEC, and because the growth in size of a dictionary slows down after several thousand documents are processed). This later point reinforces the argument made earlier, that $T << M$.

Fig. 5 FAST-INV EXAMPLE

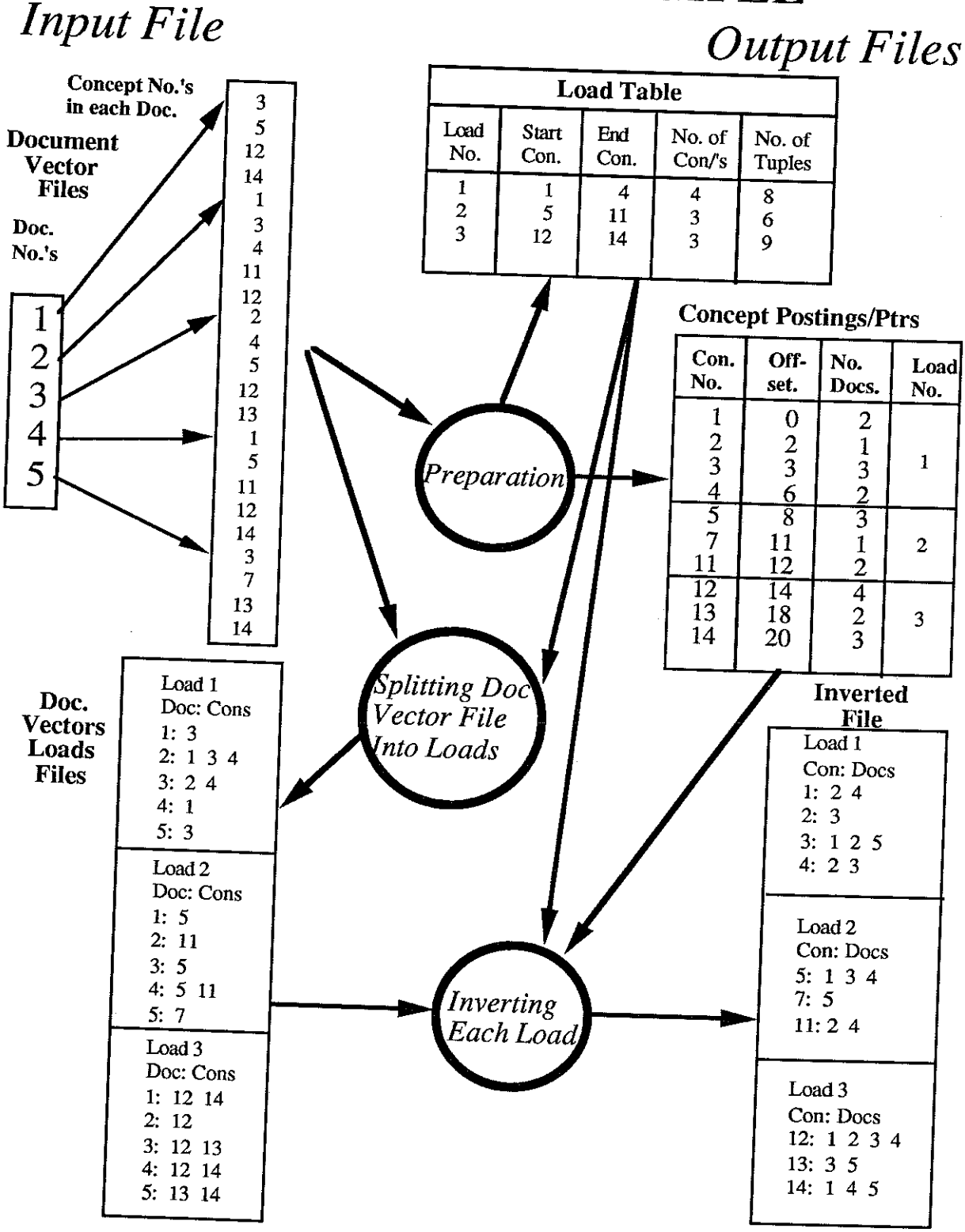


Table 1. CACM and INSPEC Document Collection Statistics

Description	CACM	% of INSPEC	INSPEC
No. of Documents	3,204	25	12,684
Bytes of Original Text	2,187,742	25	8,680,495
Bytes of Document Pointers	38,480	25	152,240
Bytes of Concept Lists	713,120	21	3,348,776
Bytes of Dictionary	178,902	50	355,894

An initial study was made using the CACM collection, to ascertain if simple sorting would suffice to build an inverted file for the library data. The results are shown in Table 2. The SMART run used the regular inversion program, carried out in primary memory. The Load/Sort method divided the collection into loads that would fit into primary memory, and then used quicksort to process the data for each memory load. There were expenses involved in formatting the data before and after the sorting, so that a simple transformation could be carried out, as was shown earlier in Figure 2. It is easily seen that the Load/Sort method was more expensive than the SMART approach.

Table 2. Comparison of CACM Inversion Times for SMART and Load/Sort Methods

Method	Real (secs.)	User	System	User+System
SMART	65.5	21.8	31.7	53.5
Load/Sort	125.8	46.8	54.2	101.0

Further testing was done with the larger INSPEC collection. To demonstrate the effects of different collection sizes, that collection was subdivided so that 1/4, 1/2 and full size runs could be made. Statistics for files involved in each of these versions are given in Table 3.

To provide comparison, the SMART inversion program was used with the three sizes of INSPEC collection. Results are given in Table 4. It can be seen that paging goes up rapidly with the size of the input files. Probably because of the paging, disk reads are very numerous for the full collection. It is harder to explain the processing time utilization. Most likely the increase from 1/4 to 1/2 size collection is less than a doubling because of overhead in the processing, while the increase from 1/2 to full collection reflects the more than linear complexity of the algorithm.

This run and others were made using a lightly loaded VAX-11/785 computer, with 4 megabytes of primary memory, running the ULTRIX version of Berkeley UNIX (VAX and ULTRIX are ™ of Digital Equipment Corporation). The "time" system command provided information regarding user time (time spent on the job with the processor in user mode), system time (additional time spent on the job with the processor in kernel mode, i.e., processing I/O or other system calls), disk reads (in units of 8K blocks), disk writes, page faults, etc. While real time varies widely depending on load, these other measures are not greatly affected, and in many cases represent a run whose values are in the middle of a set of replications.

Table 3. Sizes of Inversion Files from 1/4, 1/2, and Full Size INSPEC Collections

File/Collection	1/4 Size/% of Full		1/2 Size /% of Full		Full Size
INPUT (bytes)					
Document Pointers	39,116	26	77,324	51	152,240
Concept Lists	837,540	25	1,674,780	50	3,348,776
OUTPUT (bytes)					
Concept Pointers	87,696	50	123,408	71	174,876
Document Lists	824,512	25	1,649,016	50	3,298,040
CONCEPT COUNTS					
High Concept No.	20,997	54	20,998	54	38,639
No. Distinct Concepts	7,308	50	10,284	71	14,573

Table 4. Statistics for Inversion of INSPEC Using SMART Inversion Method

Measure/Collection	1/4 Size/% of Full		1/2 Size /% of Full		Full Size
Page Faults	15	0.4	613	16.8	3658
Disk Reads	222	12.8	440	25.3	1738
Disk Writes	293	30.4	507	52.6	963
User Time	28.2	21.3	51.9	39.2	132.5
System Time	48.7	32.6	66.5	44.5	149.3
User+System Time	76.9	27.3	118.4	42.0	281.8

6.2 FAST-INV parameters

Since FAST-INV makes extensive use of primary memory, it was appropriate to buffer all disk read and write operations with the minimum size buffer, thereby freeing remaining memory for important uses. Table 5 indicates how the size of I/O buffers effects the total time required for inverting the three sizes of INSPEC collection. All runs were made under the assumption that 2 megabytes of primary memory were freely available. It can easily be seen that using 8K buffers is the wisest course. Buffers of that size were used in all subsequent runs.

The next parameter of interest was the amount of available memory, M. Table 6 below shows the values for the load tables created during runs with M set to 1, 2, and 3 megabytes. There is the expected decrease in number of loads as M goes from 1 to 2 megabytes, but the 3 megabyte run required the same number (i.e., two) of loads as the 2 megabyte case (because 3 megabytes was just slightly too small for 1 load to suffice). The relationship between the number of distinct concepts and the highest concept number, caused by hashing of the dictionary, can be readily seen.

FAST-INV was coded in the C programming language, run through the optimizing compiler, and tuned to minimize memory and processing requirements. The amount of primary memory allowed, M, was used to allocate a block of space that was then managed by the program. Routines " oalloc" and "ofree," our versions of the system allocation and free commands, were easy to develop since memory arrays were obtained and released in a last in first out fashion.

Table 5. Effect of Buffer Size on User+System Time for Inversion with FAST-INV (2Mbytes)

BufferSize/Collection	1/4 Size/% of Full	1/2 Size /% of Full	Full Size
2K BUFFERS	12.0 23.2	21.5 41.6	51.7
4K BUFFERS	10.4 22.7	20.6 44.9	45.9
8K BUFFERS	10.3 22.8	19.6 43.5	45.0
64K BUFFERS	10.5 23.1	20.3 44.7	45.4

Table 6. Effect of Memory Size on Loads in FAST-INV

Load+Measure/Memory	1M	2M	3M
LOAD 1			
No. Tuples	124,619	249,559	374,292
No. Distinct Concepts	4,614	8,862	13,408
Highest Concept No.	6,731	12,607	19,072
LOAD 2			
No. Tuples	124,940	162,696	37,963
No. Distinct Concepts	4,248	5,711	1,165
Highest Concept No.	12,607	38,639	38,639
LOAD 3			
No. Tuples	124,733		
No. Distinct Concepts	4,546		
Highest Concept No.	19,072		
LOAD 4			
No. Tuples	37,963		
No. Distinct Concepts	1,165		
Highest Concept No.	38,639		

As noted earlier, the splitting approach taken in this version of FAST-INV was to produce the subset of the document vector file for a given load by making a pass through the input. This caused extra disk reads, but that had little effect on the processing time reported in Table 7 below. For the small 1/4 size collection, memory size also had little effect. For the 1/2 size collection, processing time for the 2 and 3 megabyte cases was slightly less, probably due to there being half as many passes. For the full collection, processing time with 2 megabytes was less than for 1 megabyte, again probably because of having fewer loads. However, using 3 megabytes for the entire period of time needed to process the full collection led to many page faults (on a system with only 4 megabytes of real memory), forcing the processing time to exceed that for the 2 megabyte

memory run.

Table 7. Effect of Memory Load Size on INSPEC Inversion with FAST-INV (8K buffers)

Measure/Collection	1/4 Size/% of Full		1/2 Size /% of Full		Full Size
1 MBYTE MEMORY					
Loads	1		2		4
Page Faults	1		1		7
User+System Time	10.3	17.7	22.7	38.9	58.3
2 MBYTE MEMORY					
Loads	1		1		2
Page Faults	1		1		1
User+System Time	10.3	22.8	19.6	43.5	45.0
3 MBYTE MEMORY					
Loads	1		1		2
Page Faults	1		5		587
User+System Time	10.3	21.5	19.6	40.9	47.9

Based on these runs, the most appropriate situation for running FAST-INV was to have 8K buffers and 2 megabytes of memory available. For this case, the performance as shown by a variety of measures is reported in Table 8. One load is needed for the 1/4 and the 1/2 size collection, and 2 loads were needed for the full collection. Paging was negligible. Disk reads are as predicted based on the number of passes and loads. Disk writes are primarily for producing the output files. Processing time increases in roughly a linear fashion (except that for the full size, due to the method of splitting shown, there is a slight extra expense because of the added disk I/O).

Table 8. Statistics for Inversion of INSPEC Using FAST-INV (2M load, 8K buffers)

Measure/Collection	1/4 Size/% of Full		1/2 Size /% of Full		Full Size
Loads	1		1		2
Page Faults	1		1		1
Disk Reads	427	16.9	856	34	2521
Disk Writes	273	29.7	483	52.6	919
User Time	5.0	21.8	9.7	42.4	22.9
System Time	5.3	24	9.9	44.8	22.1
User+System Time	10.3	22.8	19.6	43.5	45.1

Finally, given the FAST-INV case of Table 8 above, it was appropriate to compare FAST-INV with SIRE and SMART in terms of processing the full INSPEC collection. To give the most useful measure, all three runs were made on the same computer with no other users logged on. It is clear from Table 9 that the FAST-INV method was much better than the SMART and SIRE approaches. However, it is important to note that the cost of building the dictionary is included in the SIRE time for inversion, not in the indexing time. Nevertheless, it is clear that the time for inversion with FAST-INV is considerably less than that of either of the other two programs.

Table 9. Real Time Requirements (min.) for SIRE, SMART, FAST-INV (on INSPEC)

Method	Comments	Indexing	Inversion
SIRE	Dictionary built during inversion	35	72
SMART	Dictionary built during indexing	49	11
FAST-INV	Dictionary built during indexing	49	1:14

7. Conclusions and Further Work

In order to test the effectiveness of advanced retrieval algorithms with a large real life test collection, it was decided to use a large subset of the data from the Virginia Tech library online public access catalog. The SMART retrieval system has been extended to handle such a large collection. In the process, a more effective means for inverting large document files was sought.

A new algorithm for building an inverted file has been developed that makes use of the large amounts of primary memory available on modern computers, and exploits the inherent order of the input data. Whereas other approaches are O(N log N) or polynomial, and require an extensive amount of disk I/O or paging for large collections, FAST-INV only requires a small number of passes through the input file and involves no sorting. Performance studies show it to be roughly an order of magnitude faster than the SMART system, which on a medium size collection is in turn faster than the SIRE approach.

FAST-INV involves a splitting of the input file. While performance data supplied is based on splitting method 1, that avoids creating a large disk file at the expense of requiring additional passes through the input, lower overall cost is expected for large collections when splitting method 2 is used. Note that FAST-INV has been used with a library catalog database of roughly 500,000 entries - proof of its utility for large collections.

Acknowledgements

Sharat Sharan, serving as a graduate assistant and working on his M.S. project, programmed an early version of the algorithms described and made some preliminary runs using SIRE and SMART. Jeff Pache of The Institution of Electrical Engineers provided a copy of the INSPEC document collection to Cornell University, enabling use of the document vectors in these studies. Robert Dattola provided an early form of the CACM collection. Gerard Salton and his students have provided versions of SMART and of test collections. Paul Gherman, Director of Virginia Tech libraries, and Vinod Chachra, President of VTLS Inc., along with their staffs, have aided by preparing special dump routines and by providing data for the library experiment.

Bibliography

[ADAM 85] Adams, Michael Q. CDROM: Myths, Realities & Prospects. In *ASIS '85, Proc. 48th ASIS Ann.Mtg.*, 54-58, Oct. 1985.

[BATE 86] Bates, Marcia J. Subject Access in Online Catalogs: A Design Model. *J. Am. Soc. Inf. Sci.*, 37(6):357-376, Nov. 1986.

[BAYE 72] Bayer, R. and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173-189, 1972.

[BENT 79] Bentley, J.L. and J.H. Friedman. Data Structures for range searching. *ACM Comp. Surveys*, 11(4):397-409, 1979.

[BORG 85] Borgman, Christine L. Designing an Information Retrieval Interface Based on User Characteristics. In *Res. & Dev. in Inf. Ret., Eighth Annual Int. ACM SIGIR Conf.*, Montreal, 139-146, June 1985.

[BUCK 85a] Buckley, C. Implementation of the SMART Information Retrieval System. TR 85-686, Cornell Univ., Dept. of Comp. Sci., May 1985.

[BUCK 85b] Buckley, Chris and Alan F. Lewit. "Optimization of Inverted Vector Searches." *Res. & Dev. in Inf. Ret., Eighth Annual Int. ACM SIGIR Conf.*, Montreal, 97-110, June 1985.

[COCH 83] Cochrane, Pauline A. and Karen Markey. Catalog Use Studies — Since the Introduction of Online Interactive Catalogs: Impact on Design for Subject Access. *Library and Information Science Research*, 5(4): 337-363, 1983.

[COME 79] Comer, D. The Ubiquitous B-tree. *ACM Comp. Surveys*, 11(2):121-137, June 1979.

[FALO 85] Faloutsos, C. Access Methods for Text. *ACM Comp. Surveys*, 17(1):49-74, March 1985.

[FOXE 83a] Fox, E.A. Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types. Dissertation, Cornell University, University Microfilms Int., Ann Arbor MI, Aug. 1983.

[FOXE 83b] Fox, E.A. Some Considerations for Implementing the SMART Information Retrieval System under UNIX. TR 83-560, Cornell Univ., Dept. of Comp. Sci., Sept. 1983.

[FOXE 83c] Fox, E.A. Characterization of Two New Experimental Collections in Computer and Information Science Containing Textual and Bibliographic Concepts. TR 83-561, Cornell Univ., Dept. of Comp. Sci., Sept. 1983.

[FOXE 86a] Fox, E.A. Information Retrieval: Research into New Capabilities. In *CD-ROM: The New Papyrus,* Steve Lambert and Suzanne Ropiequet (eds.), Microsoft Press, 1986, 143-174.

[FOXE 86b] Fox, Edward A. and Sandra Birch. "UNIX Micros for Students Majoring in Computer Science and Personal Information Retrieval." *Microcomputers for Information Management,* 3(1): 15-29, March 1986.

[FOXE 88] Fox, E.A. and Matthew B. Koll. Practical Enhanced Boolean Retrieval: Experiences with the SMART and SIRE Systems. *Information Processing and Management,* 24(3): 257-267.

[FUJI 84] Fujitani, L. Laser Optical Disk: The Coming Revolution in On-Line Storage. *Commun. ACM*, 27(6):546-554, June 1984.

[GOLD 84] Goldstein, C.M. Computer-Based Information Storage Technologies. *ARIST*, 19: 65-96, 1984.

[GONN 84] Gonnet, G.H. *Handbook of Algorithms and Data Structures.* Addison-Wesley,

Reading, MA, 1984.

[HASK 80] Haskin, R. Hardware for Searching Very Large Text Databases. In *Proc. 5th Workshop on Comp. Arch. for Non-Numeric Proc.*, ACM, New York, 49-56, March 1980.

[HEAP 78] Heaps, H.S. *Information Retrieval, Computational and Theoretical Aspects.* Academic Press, New York., 1978.

[HOLL 85] Hollaar, L.A. A Testbed for Information Retrieval Research: The Utah Retrieval System Architecture. *Proc. 8th Annual. Int. ACM SIGIR Conf. on R&D in Inf. Ret.*, Montreal, 227-232, June 1985.

[KNUT 73] Knuth, D.E. *The Art of Computer Programming: Vol 3/ Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[LOVI 68] Lovins, B.J. Development of a Stemming Algorithm. *Mech. Trans. and Comp. Ling.*, 11(1-2):11-31, March-June 1968.

[NORE 77] Noreault, T., M. Koll and M.J. McGill. Automatic Ranked Output from Boolean Searches in SIRE. *J. Am. Soc. Inf. Sci.*,28(6):333-339, Nov. 1977.

[RAOS 85] Rao, S.V. Nageswara, S. Sitharama Iyengar, and C.E. Veni Madhavan. A Comparative Study of Multiple Attribute Tree and Inverted File Structures for Large Bibliographic Files. *Inf. Proc. & Mgmt.*, 21(5):433-442, 1985.

[SACK 85] Sacks-Davis, Ron. Performance of a multi-key access method based on descriptors and superimposed coding techniques. *Inform. Systems,* 10(4), 391-403, 1985.

[SALT 83] Salton, G. and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[STAN 86] Stanfill, Craig and Brewster Kahle. Parallel Free-Text Search on the Connection Machine System. *Commun. ACM,* 29(12): 1229-1239, Dec. 1986.

[VANR 79] Van Rijsbergen, C.J. *Information Retrieval: Second Edition.* Butterworths, London, 1979.

[WILL 85a] Williams, Martha E. Electronic Databases. *Science* 228(4698): 445-456, 26 April 1985.

[WILL 85b] Williams, Martha E., ed. *Computer-Readable Databases.* American Library Assoc., Chicago 1985.