Surrogate Subsets: A Free Space Management Strategy for the Index of a Text Retrieval System

F. J. Burkowski

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada

ABSTRACT

This paper presents a new data structure and an associated strategy to be utilized by indexing facilities for text retrieval systems.The paper starts by reviewing some of the goals that may be considered when designing such an index and continues with a small survey o various current strategies. It then presents an indexing strategy referred to as surrogate subsets discussing its appropriateness in the light of the specified goals. Various design issues and implementation details are discussed. Our strategy requires that a surrogate file be divided into a large number of subsets separated by free space which will allow the index to expand when new material is appended to the database. Experimental results report on the utilization of free space when the database is enlarged.

INTRODUCTION

This paper will present a new data structure and an associated strategy to be utilized by indexing facilities for text retrieval systems. Typical applications [FAL87D], [CHR86] include the archiving and retrieval of natural-language documents contained in very large databases such as automated law [DEF88] and patent databases, electronic encyclopedias, abstracts, medical libraries, automated office filing and newspaper databases. In the more sophisticated systems, a computer network is used to communicate user queries to a document server which responds by sending a selection of documents back to the use workstation. Queries typically involve the inclusion of words or phrases in a syntax which defines a Boolean or relevance search that selects a hopefully limited set of documents tha are germane to the information needs of the user. Query syntax may support the stipulatio of words within various text elements of a document, for example, the query may reques all newspaper articles containing "acid rain" WITHIN the headline AND "river pollution WITHIN the main text.

Typically, during retrieval operations, words or phrases extracted from the query ar presented to an index facility which maintains the text locations of all the significant word in the database. Depending on the needs of the application, the precision used to specif the location or address of a word may be extremely narrow (byte displacement in a text file or very wide (the address of some text element, perhaps the document itself). In the forme case we will consider the index entry for a word to have an address granularity of a byt while in the latter case the address granularity is some larger extent specified by the type o some particular text element (for example, a sentence, paragraph, chapter or the documen itself). Naturally, the granularity of the address has an effect on the size of the index an on the nature of the queries that may be handled efficiently.

In response to a query the retrieval system will access the index, extracting from it all required address lists and it will then perform various manipulations on these lists in an attempt to determine the address of words which meet the constraints imposed by the query. These manipulations typically involve intersections, union and sorting operations. We will avoid any further discussion of these manipulations since this paper will deal primarily with index techniques.

This paper will assume that the words selected for indexing have been determined by the needs of the application. Techniques which consider the appropriate strategies for index word selection generally fall into one of two categories which serve to support either free-text search or keyword searching. These issues are discussed briefly in [STA86] and more extensively in [SAL86]. While not advocating any particular strategy we will generally assume index word selection is being done to support a free text search allowing the user to retrieve information from the database after defining queries which incorporate arbitrary combinations of document words. This assumption places the heaviest demands on the size and performance requirements of the index, keyword searching typically requiring fewer index entries.

ASSUMED ENVIRONMENT

The following list of assumptions will summarize the text retrieval environment that this paper considers:

1) Multi-user large database

   The multi-user environment will make frequent demands on the system which should strive to provide short response times while dealing with an index that is very large.

2) Dynamic Growth

   The database will grow with the addition of new material.

3) Flexibility

   Indexing techniques should *potentially* allow text inversion on every word of the database. The system should efficiently handle queries that contain word proximity constraints and phrases.

GOALS

1) Fast retrieval

   List manipulation aside, most of the time spent in retrieval arises from disk accesses to the index and so indexing techniques should focus on achieving low I/O counts.

2) Fast loading and appending

   The initial database load and subsequent database append operations should be done quickly and in a fashion that will least compromise retrieval ability.

3) Efficient Free Space Management

During append operations the index will extend into free space areas. This should be done in a way that does not seriously compromise retrieval performance, append performance or the efficient utilization of the storage area.

It can be seen that these three goals are in a type of dynamic tension in that design tradeoffs which favour one goal tend to weaken the achievability of the other goals.

OTHER ISSUES: INDEX SIZE

An important consideration is the size of the index. This paper will assume that the primary use of the index will be the retention of word addresses within the text file. In this case, the size of the index is very dependent on the address granularity that is utilized. Small granularity addressing (say down to the text word or character level) will give a much larger index but the system functionality is greatly enhanced since proximity searching and detection of phrases are handled in a much more effective fashion.

Let us consider this dependency in more quantitative terms. We will develop formulae that establish the ratio of index size to overall text size under two scenarios:

A) address granularity is a single byte

B) address granularity is a text element with a minimum size of TE_S bytes.

The following notation will be used:

DB_S	=	database size in bytes
w_S	=	average size of word (4.87 bytes)
TE_S	=	text element size in bytes
TE_V	=	average number of distinct words in a text element of size TE_S
R_{sb}	=	ratio of index size to text size when address granularity is a single byte
R_{te}	=	ratio of index size to text size when address granularity is a text element of size TE_S

The value of w_S was obtained by calculating the average word length in a newspaper data base containing 250 megabytes of text. If we include the delimiter after each word, the word accounts for $1 + w_S = 5.87$ bytes on average.

With an address granularity of a single byte and assuming each address is stored in a sequence of bytes we get $DB_S\lceil(\log_2 DB_S)/8\rceil/(1 + w_S)$ as the size of the index (ignoring any contribution necessary to define the structure of the index). Thus,

$$R_{sb} = \lceil(\log_2 DB_S)/8\rceil/(1 + w_S) \quad (1)$$

Now with the wider granularity it is possible to save on the size of the address since it need only identify the text element to specify a word's location. Consequently the address size is $\lceil(\log_2(DB_S/TE_S))/8\rceil$. More significantly we can eliminate duplicate words in the text element during the load activity, indexing only the distinct words. The number of distinct words V in a sequence of N consecutive words can be derived from the formula

$$V(\gamma + \ln V) = N \tag{2}$$

where $\gamma = 0.57721$ (Euler's constant). This formula (derived in [FAL84]) assumes that the distribution of words in text follows a Zipf distribution. Thus the size of the contribution made by the text element to the index is $TE_V \lceil(\log_2(DB_S/TE_S))/8\rceil$ where $TE_V(\gamma + \ln TE_V) = TE_S/(1 + w_S)$ and so

$$R_{te} = \lceil(\log_2(DB_S/TE_S))/8\rceil / ((1 + w_S)(\gamma + \ln TE_V)). \tag{3}$$

As an example, let us consider 1 gigabyte of text stored on an optical disk. Equation (1) indicates that with single byte address granularity the index has a size that is 68% of the size of the text. In practice, since the addresses are in ascending order, we can use a simple front end compression scheme on the addresses to drop this to a smaller percentage providing a final size that is considerably lower than the frequently cited extreme of 300% mentioned in [HAS81].

If we consider a text element size TE_S to be equal to 1024 bytes then $TE_V = 40.72$. In this case equation (3) indicates an index of size 12% and again with front end compression this can be reduced even further.

SEARCHING FOR PHRASES WHEN THE ADDRESS GRANULARITY IS LARGE

With the above observations in mind it becomes extremely tempting to do a <u>partial</u> inversion of the text by employing this larger address granularity. Savings would be significant, especially for large multi-gigabyte text collections. The major problem is that these schemes have a good average response time but an unpredictable worst case response time [DEF89], very extended response times occurring when searching for a phrase which must be detected by a final document inspection since the search itself can only retrieve documents containing the phrase constituents in any order and in any location within the document.

Many examples of such an anomaly can be given, the worst situation being a phrase which is relatively rare in occurrence but which is comprised of words that are frequently used elsewhere in the text. We have performed indexing experiments using document level address granularity and have discovered many queries that support this claim. For example, in a newspaper collection of 95,000 articles a search for documents containing the phrase "John Turner" produces a very fast response fetching the first of 646 articles in less than 5 seconds. However, a search for documents with the phrase "the new John Turner" takes over 3 minutes because of the extensive document scanning (typically the word "new" appears in roughly one out of every three newspaper articles).

A Brief Overview of Current Access Methods

We will start by discussing some of the indexing techniques that have been used for text. This discussion is not meant to be a tutorial, but will instead describe these strategies from a perspective that will consider the performance and goals stated earlier. A more comprehensive description of the following strategies appears in [FAL85].

Inverted Lists

Inverted lists [TEO82] (pg. 344) can be implemented using a database dictionary [FAL85], and a postings file. A word from the query is found in the dictionary. The dictionary entry contains a pointer which selects a list of addresses specifying the text locations containing that word. All such lists are stored in the postings file. Organization of the dictionary can be done using a variety of techniques such as B-trees [BAY72], TRIES, or hashing. In most cases the technique will allow rapid updates to the dictionary when the database increases in size due to the appending of new documents. Accomodating list expansion in the posting file is a more challenging problem especially if one is concerned about space consumption on secondary storage.

Discussion

Inverted lists can exhibit very fast retrieval times provided the address list for a particular word is stored in contiguous areas of the disk. During load or append operations one or more addresses will be appended to various lists in the postings file and since these are spread over a large area of the disk there will be many disk seeks. Typically, one of two strategies is used:

1) The address lists in the postings file are retained in a series of chained buckets. When a bucket is filled, a pointer to a new bucket allows the list to expand. In addition to internal fragmentation the scheme has the drawback of extending the retrieval time due to the extra disk seeks across the noncontiguous buckets.

2) In an effort to maintain fast retrieval each list can be stored in a contiguous extent which has a length equal to 1 or more adjacent buckets. When new entries are about to overflow the extent the list can be copied to a longer free extent made available from a free space list. The original extent is returned to the free list manager. Retrieval is not compromised but the load process becomes quite extended due to the copying and overhead associated with the free list management.

Signature Files

Signature file techniques [BER87], [CHR84], [FAL87S], [STA86] typically require that each document be divided into "logical blocks" each containing a constant number D of distinct, non-common words. Each word is mapped to a word signature which is a bit pattern of length F (F=512, for example) with m bits set to "1", the rest being 0. Positions of the m "1" bits are determined by hashing techniques. The word signature derived from the D distinct words are then OR-ed together to form the *block signature* corresponding to the logical block. In the next figure we use small values (F=12, m=4, D=3) to illustrate the technique.

Word	Signature		
phantom	0 0 0 1	1 0 0 0	0 1 1 0
opera	0 0 0 1	0 1 1 0	0 0 0 1
webber	1 0 0 0	0 0 1 0	0 0 1 1
block signature	1 0 0 1	1 1 1 0	0 1 1 1

Illustration of a Block Signature for Three Words

Searching for a word is done by creating the word signature and then examining each block signature to see if that word signature has been included. Because of the hashing and due to the superimposition of the word signatures it is possible that a block signature appears to contain a word signature even though the corresponding word is not actually in the document. This occurence of a "false drop" will happen very infrequently if appropriate choices are made for m, F and D [TSI83], [CHR84].

Discussion

The main advantage of a signature file is the rapid update capability. Appending new information to the database will result in a simple extension of the signature file into the free space that follows it. Utilization of the free space is excellent since there is no possibility of fragmentation. Size of the index is small, quoted at 5 to 10% in [CHR84] when the logical block contains about 40 distinct words. Since the address granularity is essentially the size of the logical block, the signature file exhibits a size which is not surprisingly low. It is comparable to the size that would be achievable with inverted list schemes using the same address granularity (see equation (3)).

Superimposed encoding implies that proximity and phrase searches must be done with document filtering.

Retrieval is quite slow because of the extensive scanning. For example, a signature file that is 5% of the size of the text file will be 30 megabytes in length if the text is 600 megabytes long. Since disk transfer rates are typically one megabyte per second, the I/O time for accessing the signature file will be half a minute. Various modifications to superimposed encoding have been implemented [FAL87D], [ROB79] most with the goal of speeding up the retrieval strategy while trying to maintain the ease of update.

Bit-Slice Signature Files

Roberts [ROB79] stored signature files in a "bit-slice" fashion. The large array of bits comprising the signature file is transposed and stored in such a way that the bits from the i-th position of all the signatures are stored contiguously in a bit-slice. Since we need not check the j-th bit of any signature in the file if the j-th bit of the signatures derived from all the query words is "0", we can reduce the amount of scanning that is to be done. We need only scan those bit-slices that correspond to the bit positions in the query word signatures that are set to "1". This considerably speeds up the scan operation but the cost per query word is still rather high since it involves one disk seek for each "1" bit set in the word signature plus the bit-slice transfer time which may be longer than the seek time depending on the size of the signature file. Ease of update is still retained, but only if documents are appended in batches and the signature bits are buffered in main memory before being written out to the transposed signature file.

Signature Files with Concatenated Signatures

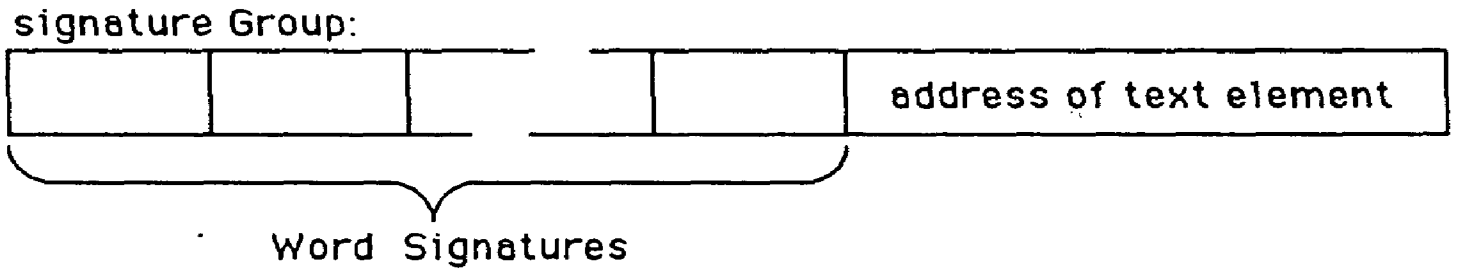
The surrogate file (essentially a signature file with concatenated word signatures) is a sequence of integers (bit strings of fixed length), each integer representing the *word signature* of a significant word contained in the main text of the database. Creation of the surrogate corresponding to a text element (for example, the entire document) involves three steps [LAR83]:

1. Common words are removed using a list of *stop* words.

2. A signature word is computed for each remaining word in the text element. In most cases, this is simply a hash function that maps words (character strings) onto integer values that are m bits long. A reasonable value for m would be between 16 and 24.

3. Duplicate word signatures are eliminated.

Thus the surrogate file is a series of *signature groups* (see next figure below) appearing in the same order as the corresponding text elements. Each signature group is comprised of a series of word signatures derived from the words in the corresponding text element followed by the address of that element. When a user query is to be satisfied, words from the query are converted into word signatures and the surrogate file is scanned for these word signatures. Whenever a match occurs the address at the end of the group is extracted so that the corresponding text element can be eventually located. Since the hash encoding is not guaranteed to be a 1-to-1 mapping, it is possible that more than one text word maps to the same signature value. During a scan operation this multiple map can produce a *false drop* and an unrelated document of no interest to the user may be retrieved. This can be detected and rectified by having software check documents before they are passed to the user. As noted in [LAR83] false drops can be made to occur with very small frequency if signature words are long enough.



Structure of a Signature Group

Note that if more documents are appended to the text portion of the database, the surrogate file is similarly extended by the appending of additional document signature groups. Retrieval operations are expected to be slow since the size of the surrogate file results in extensive transfer times and hence long scan times.

SURROGATE SUBSETS WITH ANTICIPATORY EXPANSION SPACE

We will now progress to the main content of this paper which describes an index strategy that we have dubbed *surrogate subsets*. The approach is intended to serve many of the goals presented earlier while maintaining both fast retrieval (without false drops) and fast update.

In an effort to minimize the scan time while retaining the property of an easy update, we adopt a technique which is a compromise strategy in that it has some of the properties of both inverted files and signature files with concatenated signatures. Our approach is most easily described as an extension of the concatenated signature scheme with the following modifications:

1) As in signature files a word is represented in the index by a fixed length value or identifier which we will call a *marker*. A marker has the same appearance and functionality as a word signature. We use this different terminology to stress the fact that a marker is <u>not</u> created using hashing techniques but rather it is <u>assigned</u> during a database load, the assignment technique guaranteeing uniqueness. This will avoid the false-drop problem.

2) The file of concatenated markers is subdivided into a reasonably large number of *subsets*. During the creation of the file a word marker is mapped to a particular subset and this subset will be the one that retains the marker group. During retrieval operations the same mapping is used when we wish to select the subset to be scanned when presented with the marker value derived from a given query word. The subset designation and marker value for each word are kept in a <u>database dictionary</u>. Note that within a subset there is a one-to-one mapping between markers and words.

3) Each subset is followed by free space that allows for subset expansion during subsequent database appends.

This simple overview of the technique avoids any discussion of the free space assignment strategy which works by taking advantage of the predictable nature of the Zipf distribution of word frequencies in the database. This word frequency distribution is essentially determined by an initial <u>load</u> of a portion of the database and it is then used to predict free space requirements, the objective being to drastically reduce the occurrence of overflows of subsets. The final section of this paper outlines an experiment which reports on the effectiveness of the strategy.

We now describe the load, append and retrieval operations in more detail. In these discussions we will use the following notation:

$\{W_i\}_{i=1}^{N}$	is the set of distinct words in the text that are to be indexed. Each distinct word W_i from the database will have an entry at location i of the dictionary. This entry initially keeps track of the word frequency and later will retain the marker value and subset identifier assigned to the word.
$\{S_j\}_{j=1}^{\sigma}$	is the set of identifiers for the subsets in the surrogate file.

SID[i]	This dictionary entry retains the subset identifier S_j assigned to the word W_i.
MRK[i]	This dictionary entry holds the marker value assigned to the word W_i.
CNT[i]	This temporary dictionary entry retains the number of times W_i appears in the text of the initial load.
ESZ[]	This array keeps track of the current estimated size of all the subsets.
IXMIN(ESZ)	This function returns the index of the minimum value in the array ESZ.

Other functions and arrays will be defined as we progress in the discussion.

The Load Activity

The initial load builds the index for an initial portion of the database. For example, in our experiments, we used the first 40 megabytes of a text database that was 250 megabytes long. From this first portion the loader can determine with reasonable accuracy the statistical properties of the word frequencies. While we could get by with a smaller initial load, it is more effective to use a larger portion. However, as the size increases, it become more difficult to handle other memory resident structures such as the word dictionary. The load activity progresses through the following phases:

1) Pass I

   The loader processes the text extracting from it the distinct words $\{W_i\}_{i=1}^{N}$ that are to be indexed. While doing this it builds a word dictionary which retains CNT[i] for each word W_i. When a word is taken from the text stream, the loader will attempt to find it in the dictionary. If found it increments CNT[i], if not found, a new entry is created with CNT[i]=1.

2) Assignment of Subsets

   The loader initializes all ESZ[] entries to 0. Entries in the dictionary are now processed in descending order with respect to the CNT[] value. This can be done through a sort vector SV[k] k=1,2,...,N created just prior to this step. SV[k] will be the index of the dictionary entry that is in position k after a descending order sort of CNT[].

   For k=1,2,...,N the loader executes:

```
SJ_TEMP = IXMIN(ESZ)
SID[SV[k]] = SJ_TEMP
increment ESZ[SJ_TEMP] by CNT[SV[k]].
```

   When this is completed, CNT[] locations in the dictionary may be used for other purposes such as retention of the subset ID and marker values computed in the next steps. The assignment of subset identifiers is essentially analogous to the packing of a collection of boxes with articles of various sizes. We place the largest articles in the largest available space within the boxes, followed by the lesser size items (always using the largest available box space) until we finish with the smallest articles. This heuristic approach will help ensure that subsets are fairly well balanced.

3) Assignment of Markers

The loader progresses through the dictionary assigning a marker value to each word W_i. Marker values will be unique within subsets (and hence there is never a false drop problem). The array NXTMRK[] keeps track of the next marker value to be assigned to a particular subset. Initially, all its entries are 0.

For k=1,2,...,N the loader executes:

```
MRK[k] = NXTMRK[SID[k]]
increment NXTMRK[SID[k]] by 1.
```

Marker values should be at least one byte in length, two if the number of subsets is small. The database designer must consider the anticipated vocabulary of the system and choose accordingly.

4) Pass II

In this second pass of the text, the surrogate subsets are created. For each word, the loader consults the dictionary to determine the subset selection and marker values to be used. For each word W_i in the text stream, the loader creates the pair MRK[i], ADRS[i] where MRK[i] is extracted from the dictionary and ADRS is the address of the word in the text stream. This information is appended to the end of the list held in the subset designated by SID[i]. In practice, the loader will record in another array the last MRK[ ] that was placed in a subset and if it is the same as the current MRK[i] then the MRK[i] value can be omitted. The loader also keeps track of the last address in a subset and enters ADRS[i] using an encoding that provides a type of front-end compression. In our experiments, address entries had a length of 1, 2, or 4 bytes the average length being between 1 and 2 bytes. Steering bits in the most significant part of an entry are used to distinguish markers and addresses of particular lengths.

5) Free Space Allocation

The ESZ array used in step 2 is also used in step 4 to provide a rough estimate of the size requirements for the subsets. The appropriate amount of free space can be preallocated for each subset in proportion to the values held in ESZ (ESZ[i]*6 is sufficient at this time). It is impossible to establish the exact space requirements that each subset will need to accomodate the first load since we cannot anticipate the space savings due to marker omission and front-end compression.

In this step, the loader can redefine the free space needs based on the true size of the subsets thus far. Let $FS[S_j]$ denote the length of the free space that is to follow subset S_j and let the current length of S_j be given by $CL[S_j]$. Define

$$R = TL/PL \qquad (4)$$

where TL is the total length of the text that is to be indexed (this includes the portion of the text just loaded) and PL is the length of the text portion just loaded. The success of our allocation policy rests on the assumption that, on average, future appends will cause the subsets to grow to a final size which is R times larger than the current size, that is:

$$(CL[S_j] + FS[S_j]) / CL[S_j] = R. \quad (5)$$

Equations (4) and (5) essentially deal with average behaviour. Some subsets will not totally use the free space, others will need more. In our effort to reduce the overhead associated with overflow of the free space we can extend the free space by some small fraction of the free space size recommended by (5). With this approach

$$FS[S_j] = (R-1)\ (1+XTR)\ CL[S_j] \quad (6)$$

where XTR provides the extra extension of the free space. In a future section we report on recommended values of XTR. Once all the $FS[S_j]$ are determined the subsets can be shifted so that each one gets the calculated free space allotment.

If both index and text are to be kept on a single volume the loader can be informed about the volume capacity and will use this to calculate TL the maximum allowable length of text that can be both indexed and stored on the volume. With a little calculation it can be determined that:

$$TL = PL(VC + XTR*SC) / (PL + (1 + XTR)*SC) \quad (7)$$

where VC represents the volume capacity and SC is the sum of the lengths of the current subsets that is

$$SC = \Sigma\ CL[j].$$

The Append Activity

An append is similar to a load except that a two pass procedure is not needed. When a word is extracted from the text stream there are two possibilities:

1) The word is in the dictionary
In this case the append program must recognize the commitment made by the prior assignation. If the word is found at location k then the pair MRK[k], ADRS[k] is entered into the subset just as in step 4 of the load activity.

2) The word is not in the dictionary
In this case the word is entered into the dictionary at index i, for example, and SID[i] is assigned the subset identifier which designates the subset having the shortest current length. As before NXTMRK[] is used to determine the marker value to be assigned to MRK[i]. Having done this the subset entry can be made in the usual fashion.

It is important to realize that loading and appending can proceed at a very fast rate if suitable buffering is used. In our experimental database 4096 subsets were used. Each subset was given a buffer area equal in length to the size of a disk sector. When this buffer is filled it is written out to disk. Total buffer space required is 4096 x 512 bytes = 2 megabytes. The size of the buffer area was the primary restriction on the number of subsets used by the indexing scheme. Since the buffer area represents a "wavefront" of subset extension activity the disk writes tend to be more localized using this technique than that experienced in updating the typical inverted file.

Disk writes are also far less frequent. Considering front end compression and redundant marker omissions a word in the text will contribute about 3 bytes (on average) to the index. This means that on average we can process about 170 words in the text stream before a disk write to the index is required. The index scheme tends to be much less I/O bound than indexing done with inverted lists.

Retrieval Activity

The main concern during retrieval activity is the generation of an address list for each word in the query. When presented with a query word the search administrator task will consult the dictionary to find the subset and marker value assigned to that word. It will then scan the designated subset looking for any occurrence of the marker. Whenever the marker is found addresses following it are extracted and returned as part of the required address list.

Discussion

The surrogate subsets scheme just described has the following advantages:

1) Fast Loading and Appending

   This is due to the buffering capability just described.

2) Fast Retrieval

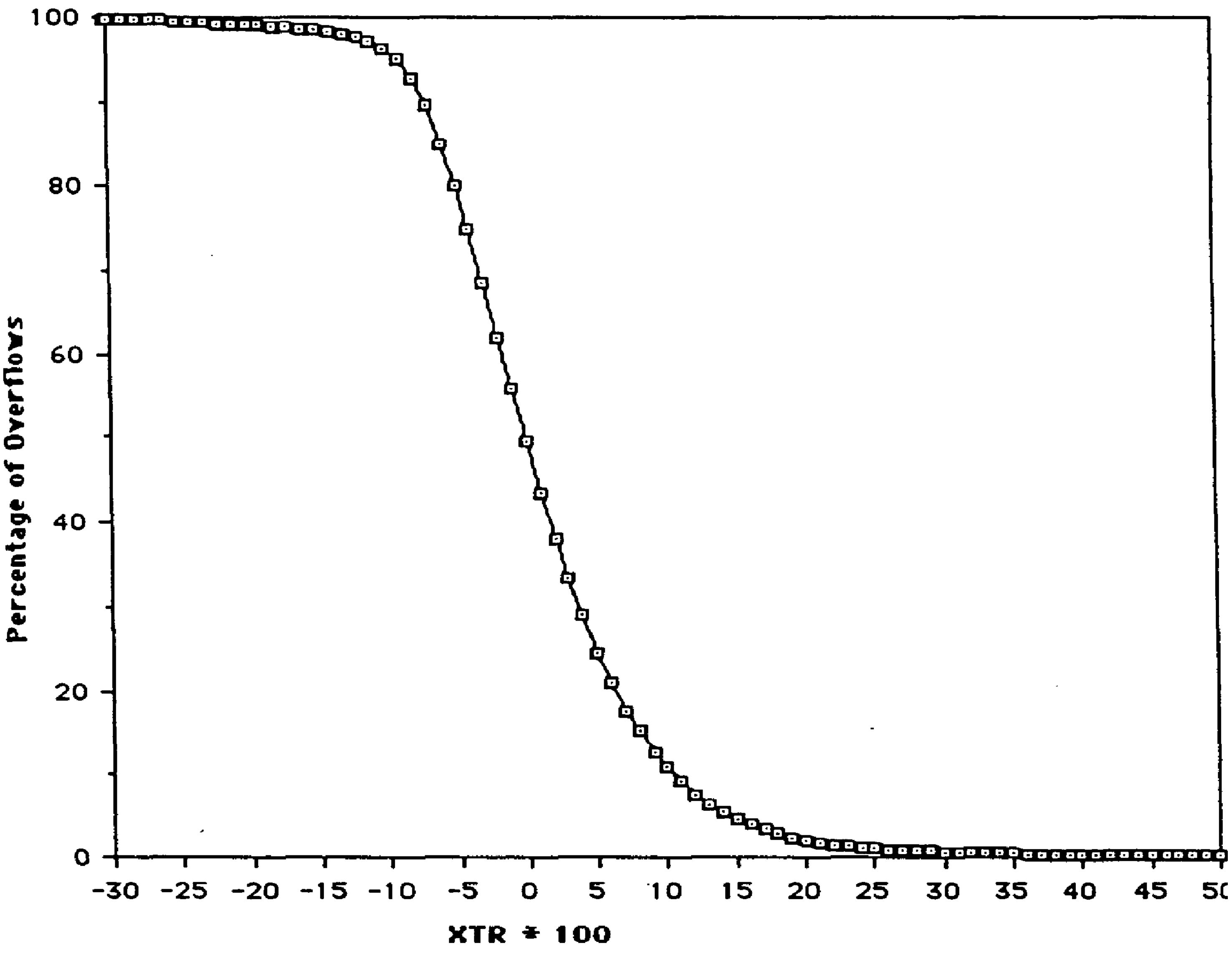
   Typically, the derivation of an address list for a query word requires one disk seek to pull out the dictionary entry, another disk seek to get to the subset and the equivalent of one more seek to scan the subset. As an example, consider a 200 megabyte text file to be stored on a 300 megabyte optical disk. With the subset building techniques described above, we can create an index that uses word level address granularity while occupying space that is about 40% of the size of the text. Actual size will depend on various decisions regarding exclusion of stop words from the index. The index will then be about 80 megabytes in size. This is distributed over 4096 subsets giving 20K bytes per subset (on average). With a disk transfer rate of 1 megabyte/second or 1K bytes/ms we can transfer this in 20 ms which is comparable to the time to do a disk seek.

3) Good Space Utilization

   Because of the marker values, the total size of the index tends to be longer than that possible with inverted lists. However, the extra space required is somewhat ameliorated by the following considerations: Since index entries for low frequency words are intermingled with one another within a subset the average difference between successive addresses is smaller than the average difference seen when the addresses have their own "private" lists as in typical inverted files. This means that front end compression can be used more effectively introducing a compression factor which tends to offset the regular appearance of marker values in these subsets. Subsets holding high frequency words have far fewer marker values since there is a higher chance that successive index entries are for the same word and so the marker value is omitted as previously described.

Experimental Results: Choosing XTR

In the following experiment we sought to determine how the number of overflows varied with respect to XTR. A newspaper database comprised of 250 megabytes was created using one load activity of 44 megabytes followed by five append operations each adding in about 41 megabytes each. Subsets were established using the strategy defined in the previous sections but the free space areas following the 4096 subsets after the initial load were made extra large so that no overflows occurred during the subsequent appends. After the last append, data defining the final lengths of the various subsets were extracted and passed to a program which evaluated the number of overflows that would have been realized for various values of XTR in the range [-0.3, 0.5]. Results are summarized in the following plot:



When XTR = 0, the number of overflows was 1940, slightly less than half the subsets. The significant feature about this plot is that the number of overflows rapidly decreases as XTR climbs to about 0.20 with relatively small gains after that. For example, with XTR = 0.10, 0.15 and 0.20 the number of overflows is 444, 188 and 86 respectively corresponding to 10.8%, 4.6% and 2.1% of the subsets. These figures indicate that small increases in the index size allocation produce substantial returns in reducing overflows. For example, if the size of the index is derived from equation (5) and amounts to 40% of the size of the text, then making it 48% the size of the text represents a 1/5 increase, ie. XTR = 0.20 and so we expect the overflows to drop to less than 2.1%.

Handling Overflows

Overflows will occur. For example, in a newspaper database the sudden popularity of a name or topic will generate many index entries for a subset that was perhaps initially considered to exhibit slow growth. Various strategies can be used to accomodate this situation:

1) Spare Subsets

Some small percentage of the subsets can be set aside as overflow subsets. Situated at the end of the subset sequence, they can have their free space reallocated as the situation requires. The overflowing subset can then have its contents moved to this subset or the high frequency entries can be weeded out and moved to a spare subset.

2) Subset Shift

When an overflow is about to occur the current subset length statistics can be used to reassess the free space allocation, the space can be reallocated and the subsets shifted accordingly.

If this strategy is put in place we can essentially eliminate the initial load. A small but comprehensive "standard" dictionary for a previously established database in conjunction with its recommended free space allocation can be used to define the free space for subsets in a newly started database. Appends can procede directly but the chances of an early overflow are somewhat increased since the word frequencies are not likely to be similar. However, when this happens there should be enough loaded data to do the reassessment just described.

SUMMARY

High frequency words in the text have a frequency distribution that follows a Zipf distribution. This gives us an advantage that we can use in indexing. Because of the distribution, such word occurrences are fairly predictable and we can use this fact to judiciously provide free space for the future growth of a subset. However, we have to exercise caution when assigning entries to subsets. Placing entries associated with two or more high frequency words into the same subset would cause inordinate growth within that subset after a succession of future appends. This would result in extended scan times which would be counter productive to fast retrieval. Consequently, we rely on an initial load activity to do some preassessment of the text, the objective being to determine with reasonable accuracy, those words that are high frequency and those that are not. The high frequency words can then be established in their own "private" subsets.

Because of the strategy used for the load activity, a low frequency word will share a subset with other low frequency words. There is an immediate benefit. While the growth of a subset containing one low frequency word (as in an inverted list scheme) is rather unpredictable, the growth of a subset with many low frequency words can be more accurately assessed since the average occurence of low frequency words in the database is much more predictable.

The net result is a tendency to build subsets that have a predictable growth. Furthermore, the subsets are as uniform as possible in length (within the constraints imposed by the uneveness of the distribution corresponding to the high frequency words). This not only aids storage utilization and scan time but also allows us to devise address compression strategies that are appropriate for each frequency level.

CONCLUSION

The indexing scheme presented in this paper provides retrieval times and storage utilization which is competitive with typical inverted list schemes but it provides an update capability which is significantly faster since it allows an easy expansion of lists into available free space. This is done without significantly compromising execution times for either retrieval or load activities. Furthermore, indexing can be done with word level address granularity thus promoting the rapid handling of queries dealing with proximity and phrases.

ACKNOWLEDGEMENTS

The author wishes to thank The Montreal Gazette for their provision of a large newspaper text collection which has been used in our experiments. I also wish to express gratitude to Mr. Dennis Ablett of Infomart in Toronto for his help in arranging the acquisition of this text. Finally I wish to thank Kent Chow of Sigscan Systems (Toronto) for work done in the programming and testing of the surrogate subset retrieval engine thus proving its effectiveness in various real-life text database applications.

REFERENCES

[BAY72] BAYER, R. AND MCCREIGHT, E., Organization and maintenance of large ordered indexes, *Acta Informatica*, vol. 1, no. 3, 1972, pp. 173-189.

[BER87] BERRA, P. B., CHUNG, S. M. AND HACHEM, N. I., Computer architecture for a surrogate file to a very large data / knowledge base, *IEEE Computer*, vol. 20, no. 3, 1987, pp. 25-32.

[CHR84] CHRISTODOULAKIS, S. AND FALOUTSOS, C., Design considerations for a message file server, *IEEE Trans. Software Engineering*, Vol. SE-10, No. 2, Mar. 1984, pp. 201-210.

[CHR86] CHRISTODOULAKIS, S. AND FALOUTSOS, C., Design and performance considerations for an optical disk-based, multimedia object server, *Computer*, Vol. 19, No. 12, Dec. 1986, pp. 45-56.

[DEF88] DEFAZIO, S. AND GREENWALD, C., The Mead information retrieval system, *IEEE Compcon 88*, Feb. 1988, pp. 431.

[DEF89] DEFAZIO, S., Private communication.

[FAL84] FALOUTSOS, C. AND CHRISTODOULAKIS, S., Signature files: an access method for documents and its analytical performance evaluation, *ACM Transactions on Office Information Systems*, Vol. 2, No. 4, Oct. 1984, pp.267-288.

[FAL85] FALOUTSOS, C., Access methods for text, *Computing Surveys*, Vol. 17, No.1, Mar. 1985, pp. 49-74.

[FAL87D] FALOUTSOS, C. AND CHAN, R., Fast text access methods for optical disks: designs and performance comparison, UMIACS-TR-87-66, CS-TR-1958, Dept. of Comp. Sci. and Inst. for Adv. Comp. Studies, Univ. of Maryland, Dec. 1987, 29 pages.

[FAL87S] FALOUTSOS, C. AND CHRISTODOULAKIS, S., Optimal signature extraction and information loss, *ACM Transactions on Database Systems*, Vol. 12, No.3, Sept. 1987, pp. 395-428.

[HAS81] HASKIN, R. L., Special purpose processors for text retrieval, *Database Engineering*, Vol. 4, No. 1, Sept. 1981, pp. 16-29.

[LAR83] LARSON, P. A., A method for speeding up text retrieval, *Proceedings of ACM SIGMOD Conference*, May, ACM, New York, 1983, pp. 117-123.

[ROB79] ROBERTS, C. S., Partial-match retrieval via the method of superimposed codes, *Proc. IEEE* , 67,12 , Dec. 1979, 1624-1642.

[SAL86] SALTON, G., Another look at automatic text retrieval systems, *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 648-656.

[STA86] STANFILL, C. AND KAHLE, B., Parallel free-text search on the connection machine system, *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1229-1239.

[TEO82] TEORY, T. J. AND FRY, J. P., *Design of Database Structures*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[TSI83] TSICHRITZIS D. AND CHRISTODOULAKIS, S., Message files, *ACM Trans. Office Information Systems,* Vol. 1, No. 1, Jan. 1983, pp. 88-98.