GRAB – Inverted Indexes with Low Storage Overhead

Michael Lesk Bellcore

ABSTRACT: A searching command (*grab*) for maintaining indexes combines acceptably fast searching with very low storage overhead. It looks like *grep* except that it demands a preindexing pass, looks only for whole words, and runs faster. As an example of performance, consider the time to search for single words in a 7.8 Mbyte file (the Brown corpus of English). The times below are in seconds on a DEC 8600 running Ultrix; the space overhead is given as a percentage of the original file.

		EGREP		GRAB		B-Trees	
Word	No. uses	Time	Space	Time	Space	Time	Space
Shakespeare	29	31.3		0.4		0.1	
Dickens	3	31.6	0	1.2	7%	0.1	97%
Chaucer	1	31.4		0.8		0.1	

Time and Space Costs

In these examples, *grab* gets 98% of the time savings for only 7% of the space costs of using B-trees. The preprocessing time for *grab* is also less, about 1/3 of that required for the B-trees.

Grab contains the following three ideas: (a) storing an inverted file by hash codes rather than words; (b) keeping a bit vector of blocks in which the items appear, rather than pointers to bytes; and (c) using a fixed length codeword compression scheme on the sparse vectors.

1. Usage

Suppose you happen to have the full text of *A Study in Scarlet* in a file H1, about 269,000 bytes long. If no one else has made the index, you type `imake H1` and about 1 minute later there is a file *H1.sg* which contains 14,000 bytes. Then you type `grab tobacco H1` and, 0.28 seconds of CPU time later, two lines are typed on your terminal:

> 019#06 the ground. You don't mind the smell of strong tobacco, I hope?"
>
> 033#23 at a glance the ash of any known brand either of cigar or of tobacco. It is just in

If, instead, you had used *egrep* you would not have had to make the index in advance, but the search would take 14 times as long (That may sound bad, but is only 4 seconds. This file is really too short to need *grab*). If, instead, you had used B-trees (a similar package is available, with indexer named *bmake* and searcher *bgrab*), the search would run in 0.08 CPU seconds but the index file would take 201,750 bytes and making it requires 3 minutes. Admittedly the search is faster, but can you tell the difference between 0.3 seconds and 0.1 seconds when the shell overhead is about 1.0 seconds? More important, would you rather have the search go faster than 0.3 seconds, or save 187,000 bytes of disk space?

There are a few options to these programs. When executing *grab*, the `-y` option folds upper and lower case in searching. Specifying `-n` prints the name of the file from which each printed line comes. And timing information on the search is printed when `-r` is specified.

For *imake*, the `-bN` option sets the block size to N, and the `-hN` option makes N the hash table size (see below for an explanation). The ordinary user shouldn't need these options.

There is also a set of library routines using the same software. This permits use of the same kind of data retrieval within a program. The entry points and their usage are:

```
fd = gopen(filename); char *fd, *filename;
```

gopen opens a file which should have an associated *filename.sg* index file, made by the *imake* routine. It returns NULL on failure and nonzero on success.

```
p = gseek (word, fd); int p; char *word, *fd;
```

gseek returns a pointer to the first byte position within *filename* where *word* is found. It returns `-1` if *word* is not found in the file.

```
p=gnext (fd); char *fd; int p;
```

gnext returns a pointer to the next byte position within the file where the key is found. It returns `-1` when it runs out of places.

```
gclose (fd); char *fd;
```

gclose closes the files.

Thus a typical loop to find all references to *cat* in a file named *dog* (and on which `imake dog` has previously been run) is:

```
int p;
char *gf;
gf = gopen("dog");
for (p=gseek("cat", gf); p>=0; p=gnext(gf))
    . . .
gclose(gf);
```

As another example of timings, the full *American Academic Encyclopedia* is 54 Mbytes long; making a set of indexes that use 4.7 Mbytes, or 8% more storage, results in the following search costs:

Word	No. Hits	CPU time (on an 8650) Grab	Egrep	Grep	Fgrep
qwerty	0	5.8	131.9		
quarto	3	9.8	127.5	224.1	281.2

Unfortunately, although the ration of real time to CPU time (lightly loaded system) for *egrep* is about 2:1, the ratio for *grab* is about 6:1, so that the response time advantage for *grab* is only a factor of 4 or 5 instead of 15. Just finding out the line where the word appears isn't that useful; you'd really like to know the title of the encyclopedia article. Using the *grab* library routines, a 76 line C program sufficed to look backwards for the article name, and it finds the article reference plus the contents of the line in a similar 1.1 minutes of CPU time. That may not be real-time response, but it is a lot better than *grep*, and the output is more useful.

2. *Introduction and Discussion*

We have a number of large text files in which we would like to look up words quickly. These include the Brown corpus of English (7.8 Mbytes), the *World Almanac* (10.6 Mbytes) the *American Academic Encyclopedia* (56 Mbytes), the LATA Switching System Generic Requirements (4.7 Mbytes), and two collegiate-level dictionaries (15.6 Mbytes for the Webster's 7th New Collegiate and 6.5 Mbytes for the *Oxford Advanced Learner's Dictionary*).

The traditional UNIX solution is to use *grep*. The UNIX religion teaches that *grep* is all you need for a DBMS, except for really large files where you might need *egrep*. However, "large" in that sense does not include the files above. Even on a DEC 8600, for example, to *egrep* through the Brown corpus on an idle machine takes 32 seconds. Thus it is desirable to have some kind of index. This is not exactly new, and various packages have been written in the past (including by the author) to make and maintain such inverted indexes [Lesk 1977].

The usual inverted index, however, roughly doubles the storage space required. The Weinberger B-tree package, [Weinberger 1981] for example, is easily adapted to store a word index

to large texts, but the overhead of the tree is roughly equal to the size of the text file originally being stored. Since most UNIX systems are always running out of disk space, it is desirable to avoid such high space overheads. The uses of bit vectors for such purposes has been studied before, although usually with the aim of imitating edge-notched cards for modern hardware. A good study of a comparable system to this one is given by Choueka [Choueka et al. 1986].

Thus we come to *grab*. It combines low storage overhead (<10%), fast response time (a square root function, able to search a 36 Mbyte file in less than 1 second), and simple code.

Some basic facts about grab:

1. It requires preprocessing to make an index; thus it is best on relatively static files, and is not appropriate for files which are changing frequently.
2. It only looks up words. It does not do substring searches. (Since some suffixing is done, *grab* is on occasion better suited than *grep*).
3. It uses very little additional storage; about 5% of the file size suffices for an index to a 1 Mbyte file.
4. Response time is quite fast. Usually *grab* searches less than 10% of the text and it can do that in a few tenths of a second on megabyte-size files.

3. How It Works

What *grab* tries is to do a *grep* through part of a file. Thus it trades time for space savings. The index file stores a bit vector indicating in which blocks in the file the words appear; thus, only those blocks need be searched to answer a query. The remainder of this section is very detailed bit-level description, and may not be of interest to readers only interested in using the program.

The details of *imake*, the procedure which produces the index file, are as follows:

1. Isolate each word in the initial file. Convert upper case letters to lower case.

2. Determine if this is one of the 287 most frequent words in English. If so, ignore it (below it is explained that should one request a search for such a word, *grab* just invokes *egrep*).
3. Remove the suffixes *-s, -es, -ed, -ing,* and *-ly.* If appropriate, a final *-e* is restored or a doubled final consonant made single.
4. Compute a hash code for the word, typically in the range 0-1009.
5. Compute the current block number. Typically the blocks are 1024 bytes long; they should not be made much smaller if efficiency is to be preserved.
6. Sort the resulting list, to make a list of block numbers in which each hash code can be found. Represent this list as a bit vector, with one bit per block. If the bit is 1, the hash code is found in the corresponding block.
7. Since the resulting bit vectors are sparse (see next section for why), compress them using a 10-bit to 4-bit fixed vocabulary code table. The compression is optimized for sparse vectors: it uses the 16 possible code words to represent the cases of (a) all bits in the original 10 zero; (b) one bit on, all others zero (10 cases); (c) however many bits are on, they are in half of the 10-bit string (4 cases); (d) some other combination of 1-bits, represented as all bits on. Note that when the vector is not sparse, the mistakes are fail-safe; *grab* may have to search too much of the file, but it will not mistakenly skip a block.
8. Store the resulting compressed vectors.

It should now be clear how the search is going to work. The steps involved in *grab* itself are:

1. Take the search word, convert to lower case, and see if it is a common word. If it is a common word that was not indexed, invoke *egrep* to do the search.
2. Suffix the word, compute the hash code, and find the appropriate bit vector.

3. Decompress the vector coding and find the blocks that contain the right hash code (and therefore may possibly contain the word sought).
4. Fetch each such block, and scan through it for the word actually wanted, checking the upper/lower case distinctions (if the `-y` option was not given).
5. When a match is found, print the containing line.

Note that the results of searching with *grab* are not the same as searching with *grep*. *Grab* looks for whole words; thus it will not find words that are substrings of longer words, the way *grep* will. On the other hand, it will find suffixed versions of the words, even if slightly changed (*groping* from *grope*). Also, the behavior of *grab* under certain exceptional circumstances (enormously long lines) may not be the same as that of *grep*.

4. Performance Estimates

The following section contains estimates of performance and can be ignored by ordinary users. It is an explanation of how fast it ought to run. *Grab* is designed to process English files; hence, all the estimates for performance are based on English. They will not be enormously different for programming languages, but may be quite inappropriate for tables of numbers or other kinds of data.

There are three main variables in this section: N, the number of bytes in the original file; H, the number of distinct hash slots; and B, the size of a block in bytes. N is not under our control, but H and B are parameters the program can choose.

The average English word (including one space) is about 6 bytes long in running text. Thus, the number of words in N bytes is N/6. However, many of these words are on the common word list; so the number of surviving words to hash is about N/10. For those wanting justification for back-of-the-envelope calculations, here are actual numbers for several text files:

Text	N	No. words	No. non-common
A Study in Scarlet	268,000	51,000	27,000
Pride and Prejudice	687,000	124,000	51,000
Moby Dick	1,242,000	218,000	103,000

How should B and H be chosen? Well, B, the block size, should be chosen to be large enough that the overhead in reading blocks is low. Here is a table of the time, in microseconds per byte, required to read from a file using various block sizes:

CPU time vs. block size

Block size	11/780	11/785	Pyramid
16384	1.5	1.3	1.2
8192	1.5	1.3	1.4
4096	1.5	1.3	1.3
2048	2.0	1.5	1.6
1024	2.4	2.1	2.1
512	3.8	2.8	3.1
256	6.5	4.8	4.9
128	12.0	9.6	9.7
64	21.8	16.9	17.2
32	43.2	32.3	32.8

Clearly, B should be at least 1024, which is the default. Making B too large is no good either, because it increases the amount of file to be scanned after the hash table has identified a block.

As for H, it has to be large enough to give reasonable dispersion of the words over hash codes; after all, the increase in search speed comes from the need to examine only 1/H of the stored items. Making it too large, however, creates a risk that the hash table will be so sparse that space is being wasted. For files of the sizes intended for *grab*, there are likely to be a minimum of several thousand different words; but not more than several tens of thousands. A value of H in the range of 1,000 to 10,000 is likely to be effective.

Then how big is the index file? The number of blocks in the file is N/B, and for each block H hash code bits must be remembered; but these have been through a 10-bit to 4-bit compression, so the size of the index in *bytes* is $H* N/20B$. A simple strategy is to take H about the same as B. It can not be exactly the same,

since B should be a power of two (to match system block sizes) while H should be prime (to simplify dispersion in hashing). But for B of 1024 choosing H of 1009 makes sense. If H is about the same size as B, the index file can be expected to be about 5% of the original file.

And how long will it take to search? It takes some time to fiddle around with the index file, but the major contribution to search time is looking through part of the main file for the actual words. How much has to be looked through? There are N/10 words, of which 1/H are likely to have the right hash codes; for each of these, however, we have to look through a whole block of B bytes. Thus the number of bytes to examine is $B* N/10H$. Again, if B ≃ H, the expected amount of searching is N/10. So, compared to *grep*, for a 5% space penalty we get a 90% speedup, on average.

Is the choice of B ≃ H sensible as the files get larger? Not really. As the search time increases, we are likely to care more about it, and to be more willing to give up a little more space to get faster searching. Shrinking B, although in theory increasing search speed by decreasing the number of blocks to look at, costs too much in system overhead (see the table above). It makes more sense to increase H. A reasonable choice is to take H roughly equal to the square root of N. This gives H = 1,000 for N = 1 Mbyte, a reasonable choice (and also a reasonable minimum H). For larger files, taking H as the square root of N and B as 1024 gives space costs of $N^{1.5}/20{,}000$ with search times of $100* N^{0.5}$ (in terms of the number of bytes to be examined).

Remember that making H too big is not efficient either, since the hash table will waste space if it is not reasonably full. Here is an indication of the number of distinct words in some texts. To use all the spots in the hash table the number of distinct words should be at least double the size of the table.

Text	Size (bytes)	No. words	No. roots
Pride and Prejudice	687,000	7,144	4,601
Tristram Shandy	1,236,000	16,348	11,740

The second column gives the number of distinct words before eliminating common words and removing suffixes; the last column

gives the number of words after those operations (the number which will actually be presented to the hashing function).

The reader may also wonder why the code compression chosen was 10 bits into 4. Why not some other values? The reason for a target size of 4 bits is for convenience in manipulating 8-bit bytes, of course. But why 10 bits to start with? Clearly, the longer the input data segment to be coded, the greater the compression; but also, the greater the probability that you will have to search a whole block because two bits were on instead of 1 bit. Here is a table showing this tradeoff. The first column is the length of the bit string used for input to the compression algorithm. The second column is the resulting size of the final index file (this is merely the first column divided by 9). The third column is the total number of bits signalled as 'on' in a typical file (it is thus related to the fraction of the file you might have to search, except that the program does somewhat better because it does not waste the unused code bits in the code words, using them to represent some of the possible 2-bit combinations). The last column is the fraction of code words that represent the "all bits on" configuration because more than one bit was on. The actual data are from a 1.2 Mbyte file, the text of *Tristram Shandy*.

Length	Size	*F-s*
8	0.50	0.145
9	0.44	0.156
10	0.40	0.168
11	0.36	0.180
12	0.33	0.192
13	0.31	0.204
14	0.29	0.214

From this table it appears that the tradeoff curve is smooth, and therefore there is no best value; it depends on the importance given to space vs. time. The 10-bit input string seemed like a good compromise, and 10 is a nice round number. The actual compression vector used allocates the possible 16 codewords to the bit patterns in the list below (with . representing any bit), using the more restrictive patterns in preference to the ones lower down:

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
.	.	.	.	.	0	0	0	0	0
0	0	0	0	0	.	.	.	.	.
0	0	0	.	.	.	.	.	0	0
.	.	.	0	0	0	0	0	.	.
.	.	.	.	.	.	.	.	.	.

5. *Performance Data*

In contrast to the estimates, here are some numbers showing actual performance on a CCI Power-6 machine. The tests were run on 5.6 Mbytes of data (the First Folio text of Shakespeare, stored as two files). In the next table, the average time required to search for words is shown as a function of hash table size.

Hash size	Space cost	CPU time for word frequency... Not found					Few	Some	Many	Total
503	2.5%	9.42	7.23	7.78	11.98	8.90	7.65	5.52	21.97	80.45
1009	5.0%	1.88	0.82	1.57	5.68	4.83	1.20	1.57	10.55	28.10
2003	9.8%	6.29	0.42	2.15	0.20	3.52	0.99	0.81	10.02	24.40
3001	14.7%	0.90	0.49	0.24	0.20	3.49	0.99	0.81	15.21	22.33
4001	19.6%	0.86	5.75	0.18	0.29	5.29	0.99	0.84	10.01	24.21
5003	24.5%	0.80	0.35	0.17	0.18	3.60	1.17	0.93	10.37	17.57
6007	29.5%	0.74	0.46	0.18	0.18	3.28	0.93	0.78	9.87	16.42
7001	34.4%	0.76	0.49	0.17	0.18	3.49	0.98	0.77	9.95	16.79

The words not found are *railway, airplane, steamship, spaceship,* and *shuttle;* the next three columns represent searches for *cat* (5 occurrences), *dagger* (16 occurrences) and *sword* (144 occurrences). Note from this table that

1. The times required to search for words are quite variable. Because of the hash table effects, a search for a not found word can take quite different amounts of time and it is not possible to say which words will be fast and which slow; it varies for different hash numbers.
2. Having many hits greatly slows the program. The efficiency advantages are greatest when looking for something that's not there.
3. Overall, for these files of several megabytes there is little advantage to hash tables more than 3000 or so slots long.

Another interesting question is whether it is important to have prime values as the size of the hash table. Here are some paired comparisons: in general prime numbers are better, but the difference is not always large and there is considerable variability from word to word.

Hash size	Prime?	CPU time for word frequency								
		Not found					Few	Some	Many	Total
501	No	13.94	8.97	6.93	7.69	10.04	2.38	11.76	15.36	77.07
503	Yes	9.42	7.23	7.78	11.98	8.90	7.65	5.52	21.97	80.45
1001	No	3.24	2.34	6.02	9.49	3.91	1.88	3.44	10.57	40.89
1024	No	5.73	13.35	18.32	19.00	15.03	22.20	13.55	15.32	122.50
1009	Yes	1.88	0.82	1.57	5.68	4.83	1.20	1.57	10.55	28.10
2001	No	1.41	0.47	0.18	0.24	3.37	0.88	2.29	9.62	18.46
2048	No	5.87	6.22	1.73	4.28	8.72	9.00	4.63	10.23	50.68
2003	Yes	6.29	0.42	2.15	0.20	3.52	0.99	0.81	10.02	24.40
2011	Yes	6.43	0.34	1.20	0.16	3.45	0.89	0.77	9.81	23.05
4096	No	0.83	2.65	0.70	3.92	7.72	1.78	2.18	10.20	29.98
4001	Yes	0.86	5.75	0.18	0.29	5.29	0.99	0.84	10.01	24.21
5001	No	0.79	1.82	0.85	3.09	3.26	0.88	0.92	9.71	21.32
5003	Yes	0.80	0.35	0.17	0.18	3.60	1.17	0.93	10.37	17.57
6001	No	0.75	0.49	3.27	0.20	3.29	0.98	0.83	9.87	19.68
6007	Yes	0.74	0.46	0.18	0.18	3.28	0.93	0.78	9.87	16.42

Opening files is expensive if they are short, but not too bad when they are megabytes. In all the data above, the Shakespeare text was stored as two files; in the following table this is compared with one 5.6 Mbyte file.

Hash size	No. Files	CPU time for word frequency... Not found					Few	Some	Many	Total
1001	2	3.24	2.34	6.02	9.49	3.91	1.88	3.44	10.57	40.89
1001	1	3.35	2.08	6.10	9.75	3.55	1.90	3.82	10.72	41.27
2001	2	1.41	0.47	0.18	0.24	3.37	0.88	2.29	9.62	18.46
2001	1	1.15	0.47	0.20	0.18	2.98	1.00	2.03	9.95	17.96
3001	2	0.90	0.49	0.24	0.20	3.49	0.99	0.81	15.21	22.33
3001	1	0.67	0.48	0.10	0.17	2.92	0.82	0.87	15.50	21.53
4001	2	0.86	5.75	0.18	0.29	5.29	0.99	0.84	10.01	24.21
4001	1	0.73	5.63	0.12	0.23	5.00	1.08	0.83	10.25	23.87
5001	2	0.79	1.82	0.85	3.09	3.26	0.88	0.92	9.71	21.32
5001	1	0.45	1.78	0.83	2.95	2.80	0.83	0.80	9.85	20.29
6001	2	0.75	0.49	3.27	0.20	3.29	0.98	0.83	9.87	19.68
6001	1	0.60	0.38	2.75	0.18	3.17	0.98	0.73	9.90	18.69
7001	2	0.76	0.49	0.17	0.18	3.49	0.98	0.77	9.95	16.79
7001	1	0.58	0.35	0.12	0.22	2.90	0.97	1.00	9.88	16.02

And, of course, this is all still faster than *egrep:*

Program Test	CPU times No hits				Few	Some	More
grab -h3001	0.9	0.5	0.2	0.2	3.5	1.0	0.8	15.2
egrep	38.4	39.9	41.4	40.8	40.9	39.2	38.7	39.3

6. *Miscellaneous Comments*

This program is not a panacea. If you need a DBMS, get a DBMS. If you need to find "lexicographically closest" items instead of exact matches, get B-trees. If your files are not at least several hundred Kbytes, you might as well use *grep*.

Because this program is intended for use with big files, some warnings about *imake* and what it requires. The final output files are only about 5% of the input, so there is probably enough space for that. But the program makes and sorts a temporary file about 50% of the size of each input file. Thus, there must be enough space on */tmp* to hold the biggest input file; and there must be enough space on */usr/tmp* to hold just about the size of the biggest

input file. If you need these directories moved, you can recompile.

Grab is written in C and contains about 600 lines of source code. The *imake* routines are about the same size (and have a 200 line suffixing program in common).

As a side effect of this project, the programs *bmake* and *bgrab* were written. They look like the *grab* the newer routines, but use B-trees. If the additional secondary storage is of no concern, they are faster and simpler. The *bmake* program requires enough space on */tmp* to hold about the same size file as the largest file you are indexing; and it requires twice as much space on */usr/tmp*. These routines require Weinberger's B-tree package.

References

M. E. Lesk, *Inverted Indexes on UNIX*, UNIX manuals, Version 7 and 4.2BSD, part 2 (Supplementary Documents), 1977.

P. J. Weinberger, *UNIX B-Trees*, 1981.

Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal, Improved hierarchical bit-vector compression in document retrieval systems, *Proc. 9th ACM SIGIR conf.* (1986).

[*submitted Jan. 14, 1988; revised June 7, 1988; accepted Sept. 23, 1988*]