# Ron's Indexing Progam
# (RIP)
# User Manual

*by Ronald S. Burkey*

## Document Mods

| | |
|---|---|
| 03/01/97 RSB: | Began |
| 03/10/97 RSB: | ... continued. |
| 01/08/02 RSB: | Prettified a little for first GPL release. |

## Table of Contents



## Introduction

RIP is a program used to maintain and access primarily English-language plaintext-only databases. The strong suit of RIP is that it can handle *very large* databases with a minimum of effort by the user. Databases of essentially unlimited size can be used simply by providing an adequate amount of mass storage for them. At present (03/10/97), for example, my own database is about 1.2G bytes, and increasing in size at over 100M per month.

RIP is an indexing and archiving system. When a text file is entered into the RIP database, it is compressed and indexed. The file can later be browsed or searched without removing it from the database, but since is it stored in RIP's own format cannot be accessed by other text-based applications. However, the file can be extracted at a later time and restored exactly to its original format. After compression and indexing, a typical 400K text file (the average size of a novel) will occupy about 300K. Smaller files (short stories, for example) are less efficient and bigger files (the Bible, for example) are more efficient.

Compared to familiar programs, RIP is like a CD-ROM encyclopedia with elements of an Internet search engine thrown in. The 'entries' in the RIP database are text files rather than encyclopedia articles, but like the encyclopedia, full-text searches can be performed to find phrases when you don't know what file (article) they occur in. For example, the database can be

searched for all files containing 'Albert Einstein', and then those files can be browsed or extracted. RIP is unlike an encyclopedia in that the databases dealt with can be much larger. For example, the entire Encyclopaedia Britannica CD-ROM, if the articles were restored to plain-text format, would probably be <300M, while lower-end encyclopedias such as Groliers or Compton's are only 1/2 to 1/3 that size. Furthermore, commercial products such as encyclopedias are only updated infrequently, and so can employ very time-consuming programs for producing a large but very fast index for their products. It is not important to Grolier's, for example, if it takes a day of computer time to re-index their encyclopedia (since it is only done once a year), and if the resultant index is larger than the original text was (since the text didn't come close to filling up a CD anyhow). Incidentally, I don't know anything about Grolier's procedures, so don't go around quoting me.

RIP is aimed instead at databases that are very large, but still under constant revision (in the sense that new text files are added to it) and hence in need of occasional re-indexing. One advantage of RIP is that only a relatively small amount of indexing data is saved (about 35% of the size of the original text data), but this is also its disadvantage: Text searches take longer than with an equivalently-sized CD-ROM encyclopedia. Search speed is traded for ease of update and compactness of index files.

I wrote RIP because my hobby is to download etext from the Internet. I have collected thousands of files from many sources. There are so many files that I personally don't even have enough time to make a list of the titles and authors of all the files. That's what's great about RIP. I don't need to know what I have in my database, because RIP can find it even if I don't know it's there. I can scarf files at will and I don't need to make any effort at all to track them. Other than the downloading itself, the database requires essentially no maintenance.

RIP presently does *not* have the capability of 'grading' files for relevance in searching the database the way Internet search engines do, although a limited capability for this could be added. RIP is *not* suitable for anything but plaintext: Graphic inserts are not supported; Adobe Acrobat files (PDF) cannot be used; word-processor, SGML, and HTML file support is spotty at best. Someday, it would be nice to have full support for HTML, including graphics and links.

Languages other than English, but which use the latin character set and don't differ statistically too drastically from English (such as Latin, French, German, Spanish, Italian) can be used without much difficulty. But if the language differs too much from English, or if the database consists primarily of the alternate language, the RIP statistical table needs to be altered as described in the final section below. For example, if 90% of the files are in English and 10% in French, it's not a problem. If they're all in Greek, transliterated into latin characters -- well, who knows? Try it and see. The program will certainly continue to work, but the efficiency of the file compression (which is optimized for English) will suffer.

At present, RIP is a DOS program. It can be under Windows, but is not a Windows program. Eventually it would be nice to convert it to Windows, to have an attractive browser and to allow background searches.

# License

RIP is copyrighted by Ronald S. Burkey, but is released as free software under the GNU General Public License (GPL), which you can view at www.fsf.org.

# Technical Limitations

Maximum text-file size: 512M bytes.
Maximum number of text-files per database partition: 64K.
Maximum number of database partitions: 100.

The theoretical maximum database size is therefore 512M*64K*100, which is fairly large. The maximum number of partitions (100) was just chosen out of nowhere, though, and could be increased with a stroke of a key.

# Simple Setup

You must set aside a dedicated directory for RIP (called, perhaps, \RIP), in which you will have the RIP program (RIP.EXE), statistical file (RIP_ALLF.DAT), and probably the RIP master index (RIP.FIL and RIP.IND). You can also put your text files here. Your text files can be put anywhere, but it is certainly most convenient to put them in a single directory structure (possibly with subdirectories) containing them and nothing else. For a database under construction (i.e., not on a CD), it is probably best just to put all the text files in subdirectories of the \RIP directory. Thus, you might have \RIP\DICKENS for books by Charles Dickens, \RIP\TWAIN for books by Mark Twain, and so forth. I wouldn't do it this way myself, because it's too much effort to have to classify books by author, but you get the idea. The scheme of classifying by author (which is something a lot of Internet sites do) conflicts with the desire for minimum-effort of mainenance. A better scheme for my purpose is to classify by source, so that each Internet site I am in the habit of scarfing files from gets its own subdirectory.

One important point that you can't ignore is that all text files which are candidates for inclusion in a RIP database must have the filename extension '.TXT'.

Creating a RIP database, or adding to an existing one, is a multi-step process, and not a particularly user-friendly one if you insist on typing out all the commands yourself, or arranging your files in some weird way. (But a batch file can be easily set up to do all the work with just a single command.) That's because the way I use the program, adding files by the hundreds, an interactive user interface would be a nuisance. A DOS command-line interface, in which all the work can be done by a batch file, is preferable.

If you've arranged everything the way I say, and don't want to know how it all works, just skip down to the final paragraph of this section. If you want more of the details, read on:

The first step is to get a list of all the files you want to add. For some uses, you could just type in all these names yourself (saving the list, for example, as another file called FILES.LST); make sure you include full pathnames such as 'C:\RIP\TWAIN\TSAWYER.TXT', rather than just 'TSAWYER.TXT'. But if you've organized your files the way I suggested (for example, in various subdirectories of \RIP), you can make the DOS DIR command do all the work for you:

```
DIR /S /B \RIP\*.TXT >FILES.LST
```

This finds all the '.TXT' files in all subdirectories of \RIP, and lists them in FILES.LST.

The next step is to compress/index the files:

```
RIP C <FILES.LST
```

or you can even avoid creating FILES.LST in the first place by just using

```
DIR /S /B \RIP\*.TXT | RIP C
```

This step takes every '.TXT' file and *replaces* it with a file of the same name but the extension '.RIP'. For example, if you start out with TSAWYER.TXT, you'll end up with TSAWYER.RIP--and no TSAWYER.TXT. So until you get confident that RIP won't destroy your files, you might want to make backup copies of your TXT files before doing this. I have personally never lost a file just relying on RIP, and I no longer make any .TXT backups myself. (And besides, I tell you in advance that you're doing it at your own risk: I'm not liable for any losses, okay?) In case you're wondering, the need for replacing *.TXT by *.RIP (as opposed to just having both types of files present simultaneously) was carefully thought out. First, for big databases, it may be hard to find room for both. Second, if the original TXT file isn't deleted, you will keep adding it to the database time after time, whenever you update the database.

The '.RIP' files are typically 70% of the original in size, but this varies from file to file.

The final step is to create a 'master index' of all your '.RIP' files:

```
DIR /S /B \RIP\*.RIP | RIP I
```

This creates files called RIP.FIL and RIP.IND. RIP.IND can be quite a large file, but still only 5-10% of the size of your plaintext. The 5% applies to very large databases, and the 10% to smaller databases. Also, your disk needs 3-4 times this as free space to create RIP.IND. The creation process can be time consuming. On a 90 Mhz Pentium, a 300M database (uncompressed) will take 9 minutes, and the time increases approximately linearly with the database size. On the other hand, the time will go down as computers get faster. For this 300M database, the master index might be 20M, and so you'll want to allow 80M of free space before starting.

As promised, all that ugliness can be beautified by creating a batch file to do the whole job for you:

```
DIR /S /B \RIP\*.TXT | RIP C
DIR /S /B \RIP\*.RIP | RIP I
```

Just type the name of the batch file from the DOS prompt, and go drink some coffee while it works.

## Simple Usage

You can interactively search or browse the database, simply by using the command

```
RIP
```

This presents a menu of options, which I won't explain in great detail (since they're reasonably self-explanatory) but I'll say a few words about each.

There are four basic ways of accessing the database. Each is a different way of choosing a file, and then viewing it with the browser. Operation of the browser is very simple, so we might as well begin with it.

The browser makes every attempt to display the file just as it would have appeared if typed to the screen from DOS, except that it performs automatic word-wrap. In the DOS version, there are no adjustments you can make (such as font, text-size, margins, etc.), except that the next section explains how to configure the text color and background color. If you want to use a really nice browser (such as QREAD), you'll have to extract the file from the database. Personally, after agonizing over this point a long time, I found to my surprise that with proper choice of color, a DOS-text browser can be a very comfortable way to read books.

Most of the allowed browser commands are listed at the top of the browser screen. You can move up or down, line-by-line (arrow keys) or page-by-page (PgUp, PgDn), or to the beginning or end of the file (Home, End). You can extract the file from the database into plaintext format (Alt-S). (The file remains in the database, but an uncompressed version of it, identical to the original, is created in the C:\UNRIP directory.) You can search for text (F or S), or for the next occurrence of a previous search (N). The 'F' text search, simply finds the next occurrence of any of the words you type in. Thus if you do an 'F'-search for 'albert einstein', it finds the next occurrence of either 'albert' or 'einstein', and uses whole words only. The 'S' text-search is the more usual kind of thing: it just searches for whatever you type in, ignoring case, including spaces, punctuation, and special characters. You can set a bookmark (B). And that's about it for the browser, except that various of the menu commands described below (R,+,-) are also accepted by the browser. The escape key gets you out of the browser and back to the main menu.

As I said before, the main menu simply provides alternate ways of chosing the files to be browsed.

The simplest (and surprisingly quite useful) method is the random selection (R). This just chooses a random file and jumps to a random point in it. The random jump is similar to browsing in an actual library of books, where you just wander around and pick up anything that looks interesting. This is the feature that's missing from the vast etext reserve of the Internet. The book you want might be out there, but you need to know the name or author. You can't just browse through books at random: Randomly browsing web sites is much easier than randomly browsing the books contained in them.

The bookmark command (B) returns you to the last bookmark you've set. Or, if like me you are reading several books simultaneously, the '0'-'9' keys take you to any of the last 10 bookmarks you've set. '0' is the earliest bookmark, '1' the next earliest, and so on.

The file command (F) takes you to a specific filename. You don't need the full path, just the name. For example, to read d:\rip\twain\tsawyer.rip, you would just enter 'tsawyer'.

The search command (S) searches the database for specific text. The system doesn't search for exact phrases, but rather for words in proximity. For example, if you search for 'albert einstein' (note the case insensitivity), RIP doesn't search for exactly this phrase, but rather for the words 'albert' and 'einstein' appearing within a few lines of each other. The program ignores common words such as 'a', 'me', 'the', etc. (You can type them in, but they won't affect the search.) For the most part, you can assume that all words of 4 characters or more affect the search, while shorter words do not. There are some exceptions. All words containing the letters 'J', 'Q', 'X', or 'Z' affect the search, regardless of length. Thus, if you search for 'the wizard of oz', the words 'the' and 'of' will be ignored, wherease 'wizard' and 'oz' are searched for. Unfortunately, this makes it impossible to search the database for phrases like 'to be or not to be,' in which every word is discarded, even though the entire phrase is very distinctive.

I will undoubtedly add more sophisticated checking over the course of time, but that's how it works now.

An 'S'-search of the database will typically find quite a few files matching the search criterion. As it finds each file, it pops it into the browser. You can then proceed to the next file with the '+' command, or to the prior file with the '-' command. The 'S'-search from the main menu locates matching files by a process of successive refinement. First it examines the master index, to eliminate most files from the search. Then it examines the remaining candidates one by one, first checking the index information embedded in the file itself, and then proceeding to an actual textual examination. The file does not appear in the browser until having passed all these tests.

# Advanced Setup

Two optional files are used to modify the operation of RIP somewhat. RIP.CFG configures the browser. RIP.LST configures the database.

The only characteristic of the browser that can currently be altered is the set of screen colors.

To set the background/foreground colors for the main menu, the browser command bar, and the browser text, RIP.CFG would contain lines of this kind:

| | |
|---|---|
| BCK_MENU=color | (color of main-menu background) |
| CLR_MENU=color | (color of main-menu foreground) |
| BCK_COMMAND=color | (color of browser command-bar background) |
| CLR_COMMAND=color | (color of browser command-bar foreground) |
| BCK_TEXT=color | (color of browser text background) |
| CLR_TEXT=color | (color of browser text foreground) |

The colors are the usual allowed IBM PC colors 0-15:

| | |
|---|---|
| 0 | BLACK |
| 1 | BLUE |
| 2 | GREEN |
| 3 | CYAN |
| 4 | RED |
| 5 | MAGENTA |
| 6 | BROWN |
| 7 | LIGHTGRAY |
| 8 | DARKGRAY |
| 9 | LIGHTBLUE |
| 10 | LIGHTGREEN |
| 11 | LIGHTCYAN |
| 12 | LIGHTRED |
| 13 | LIGHTMAGENTA |
| 14 | YELLOW |
| 15 | WHITE |

You have to use the numbers, and not the words; for example, 'CLR_MENU=15'. Also, you should generally use only 0-7, since otherwise there will be no difference between highlighted and non-highlighted areas. By default, for text I use a black background with brown foreground. This seems a bit odd at first, but I find that it makes for very relaxing reading.

RIP.LST can be used to configure how the database is partitioned.

It is sometimes convenient to partition the database into smaller databases (for example, if the database is stored on CD, each CD might be one partition of the database) and to have a separate master index for each partition. Another reason to do this is that there is an absolute limit of 64K files per partition, so if you surpass 64K total files, you will need to partition the database. RIP.LST is primarily a master list of these partitions, with one line for each partition. In its simplest form, each record just is the base name of the master index for the partition. When the master index is generated, it is always called RIP.FIL/RIP.IND (so the basename is 'RIP'), but if there are separate partitions, the master indices clearly need to be renamed or at

least stored in separate directories. For example, if we copied all the indices (*.FIL and *.IND) into a single directory and called them VOL1.FIL/VOL1.IND, VOL2.FIL/VOL2.IND, ..., we could simply have a RIP.LST like this:

```
VOL1
VOL2
.
.
.
```

The databases would be searched in the order shown.

More typically, the different partitions might be on CD's (or one on hard disk, and one on each CD.) A couple of difficulties arise in this case. First, some earlier versions of RIP used a master index named RIP.FIL/RIP2.IND (rather than RIP.FIL/RIP.IND) and once committed to CD cannot be renamed. To add the "2" to the IND-file name, we put a second field of "*" in RIP.LST:

```
C:\RIP\RIP
D:RIP *
```

In this example, there are two partitions: C:\RIP\RIP.FIL and C:\RIP\RIP.IND on hard disk, and D:RIP.FIL/D:RIP2.IND on CD.

Another difficulty is that the drive designators in CD-based master indices are generally wrong. The RIP.FIL file contains a list of all the text (.RIP) files in the database, by full pathname. But since the CDs are moved from computer to computer, these won't always be accurate. Thus, RIP.FIL on a given CD might give a filename as D:\TWAIN\TSAWYER.RIP, because on the computer used to generate the database, D: was the CD-ROM. But if the CD is moved to a computer in which the CD-ROM is E:, then the system won't be able to locate the file. Or for that matter, if (for speed) the master index was generated while the database was on hard disk and then merely copied later to the CD, even the beginning part of the pathname could be wrong. For example, RIP.FIL might give the filename as C:\RIP\TWAIN\TSAWYER.RIP. This can be handled by adding a couple of separate fields to RIP.LST:

```
RIP
D:RIP * C:\RIP D:
```

This would simply replace C:\RIP by D: in the leading positions of the filenames found in E:RIP.FIL.

# Additional Features

We have already discussed file compression ('RIP C'), master-index creation ('RIP I'), and interactive search/browsing ('RIP'). The RIP program also has the following command-line functions:

'RIP D': File extraction. This is the opposite of 'RIP C', taking a .RIP file and replacing it with a .TXT file. The command-line syntax is identical for the two:

```
RIP D <filelist
```

'RIP T': Test the database. This checks the integrity of .RIP files:

```
RIP T <filelist
```

'RIP %': This gives you the compression statistics for your .RIP files (that is, compressed size versus uncompressed size), including a summary for the database as a whole. The compression ratio is much worse than for a program such as PKZIP; the reason is that the text data is compressed reasonably efficiently (to about 40% of its original size), but then index data is added to the file (about 30%), so the overall compression is usually around 30% (meaning that the .RIP file is about 70% the size of the .TXT file).

```
RIP % <filelist
```

'RIP n' ('n' being a number, such as 20): Extract file header. What this does is to uncompress the first 'n' lines of the .RIP file. This is useful for producing a summary of the database, since the titles and authors of the files are generally mentioned somewhere in the first few lines of the file:

```
RIP n <filelist >summary.TXT
```

'RIP H': Hashcode computation: This is an interactive mode useful only for program debugging. It can compute the 'hashcode' for any given word. See the technical theory section below if you're interested in what this means.

# Large Database Management

I will illustrate the use of RIP in maintaining a large database by describing my own application.

I collect etext files from the Internet. Generally, you can think of each file as being a 'book'. The average file size is 300-400K, though some files are as short as just a few K, and others are around 10M. The Bible (a very long book indeed) is 4.5M.

While existing books in the database are sometimes revised, usually the only process by which the database is modified is the addition new books. I have to guard a little against duplication (since once a book is on the Internet it often migrates to several sites), but this isn't a big problem, particularly if you collect files primarily from a single source, such as Project Gutenberg, EWTN, Project Perseus, etc.

Initially, books are collected on hard disk. But as the database grows, it eventually becomes convenient to free up some disk space by partitioning the database into two parts and placing part of it on CD. If the database continues to grow, it is partitioned into three parts (two on CD and one on disk), and so on. Basically, therefore, you monitor the size of the hard-disk portion of the database until it reaches a good CD-size (such as 600-650M), and then just make a CD with it, clear the .RIP files off of the hard disk, make a new partition in RIP.LST, and start collecting more files in the (now-empty) hard-disk partition.

Helpful Hint: In really large databases, when you are simply trying to find a specific book by title or author, a free-form search of the entire database is a hassle. It would be more convenient to just search a title/author list. Of course, in a no-maintenance approach, there is no such thing. (You could, of course, create one manually.) However, the "RIP n" command can be used to generate a master list of the beginnings (say, the first 20 lines) of all the files, and if this master list itself is treated just like any old text file (i.e., compressed and indexed), and is the very first file in the first index, then the system will be able to complete title/author searches much quicker, since we are guaranteed that most titles and authors are in the first first file searched. This allows the user to determine the desired filename, and from there to fetch it with the 'F' command rather than the 'S' command.

# Technical Theory of Operation

The RIP system performs two basically independent functions. First, it provides a file-compression service. Second, it provides an index/search service. There is little necessary relation between these services, in that either service could be provided in a different way (or not at all), and therefore they will be discussed separately.

## *Compression.*

The file-compression ability of RIP is inferior to that of a program such as PKZIP or GZIP, and so one might wonder why this capability is present, merely than relying on these fine programs. The reason is that PKZIP, GZIP, and all other really fine compression utilities are file-based rather than block-based. By this I mean that once the file is compressed (for example, with PKUNZIP), it is necessary to uncompress the entire file before it can be accessed; if you just want to get a certain paragraph from the file, you are out of luck. However, the RIP system requires the ability to fetches random sections of files without the enormous speed penalty implied by the need to uncompress the entire file. This is why we provide our own compression algorithm.

To provide this block-addressable capability, the system divides the original text-file (.TXT) into fixed-size blocks of 8K bytes. Near the beginning of the output .RIP file is a table of pointers to these compressed datablocks. The compression itself is a context-sensitive (but not adaptive) Huffman compression. An adaptive method is not used because the block size is too small to allow an adaptive algorithm enough time to adapt. Instead, a pre-analysis of 'typical' english-language text was performed to determine the frequency of occurrence of each ASCII symbol under the following conditions: following the letter 'A', following the letter 'B', ..., following the letter 'Z', following a digit, following punctuation ('.', ',', ';', ':', '?', or '!'), or following any other symbol. Using these 29 separate sets of frequencies, 29 separate Huffman codes were generated, and stored in the file RIP_ALLF.DAT (which must be present for RIP to function). Compression is a straightforward replacement of characters by the appropriate context-sensitive Huffman code, except that the context is reset to 'other' at the beginning of each 8K block. In other words, suppose the string 'the quality of mercy' was being compressed. Consider the word 'quality'. The Huffman code for 'q' is taken from the 'other' context, that for 'u' is taken from the 'following-Q' context, that for 'a' is taken from the 'following-U' context, and so on. I find that the trick of using context-sensitive Huffman codes gives me about an extra 10% compression over using just a plain Huffman code.

For english-language text, this achieves a compression ratio (compressed text about 40% of uncompressed) close to that of an adaptive algorithm, though still slightly inferior. For some datasets, such as long strings of digits, the file is actually enlarged, since the frequencies of occurrence of each character are so different from that of typical text. For intermediate cases, such as other languages (French, German, etc.) the system is still workable but compression efficiency is reduced.

## *Indexing a file* .

Although indexing a file has a number of subtleties, conceptually it works as follows. Each 8K block of text in the file is analyzed, and a list of all unique words in the block is compiled. (For example, even if the word 'centennial' appears 7 times in the block, it will only appear on the list once. After the entire file is analyzed, we have a list of blocks, and for each block, a list of the unique words in the block. This list is then re-sorted so that it is a list, by word, of which blocks the word appears in. This re-sorted list, the 'index', is then appended to the compressed text data.

Conceptually, therefore, to determine if a phrase is in the file, we first check the index for each word in the phrase we are searching for. For example, suppose we are searching for 'albert einstein'. We look up the index entry for 'albert'. If there is no entry (i.e., if no 8K blocks contained 'albert') then the phrase doesn't appear in the file. Otherwise, we fetch the list of all blocks containing 'albert'. Then we look up the index entry for 'einstein'. Again, if there are no blocks containing 'einstein', then the phrase 'albert einstein' certainly doesn't appear in the file. If some blocks do contain 'einstein', we fetch the list of blocks. We then compare the block-list for 'albert' against that for 'einstein'. If there is no overlap, then 'albert einstein' doesn't appear in the file. If there *is* overlap, it doesn't mean that the phrase appears, just that both words are in the same 8K block. At that point, we uncompress just that block and do a regular text search on it to see if the phrase actually appears. At present, we don't really do a text search, but just

check to make sure that the individual words all appear within 100 characters or so of each other. Thus, we find 'albert einstein' or 'einstein albert' or 'einstein was named albert', etc.

As I said, though, there are some subleties. For one thing, in creating the list of unique words, we ignore case. For another, we count any set of contiguous characters as a word. Thus, 'einstein's' is treated as a word, and is not identical to 'einstein'.

Another point is that we need to guard against the case in which the phrase 'albert einstein' spans two blocks. For example, 'albert' might be the last word in one 8K block, and 'einstein' the first word in the next. We take care of this by enlarging the 8K blocks slightly so that they overlap. (This is done only for indexing, and not for compression.)

Also, it is very difficult in practice to actually deal with 'words' and 'unique words'. It is rather simple to do for small files, but large files such as the Bible contain enormous numbers of unique words. Several hundred thousand unique words exist in the english language as a whole, and even fitting this list into the memory of most current computers (under DOS, anhow) is a daunting task, not to mention the enormous amount of CPU time needed to maintain and enlarge this list during the indexing process. Even the data structures needed for this task tend to be inefficient, primarily because not all words have the same numbers of characters in them.

Because of this, we don't actually compile lists of unique words, nor index by word as I have been saying. Instead of the words themselves, we actually generate hashcodes and work with the hashcodes instead of the words. The hashcode is a 4-byte number generated from the ASCII codes of the words by a numerical process. There are approximately 4 billion possible hashcodes and only a few hundred thousand words in the language, so the intention is that there should be very little duplication by the hashcode algorithm. In other words, there should be few distinct words that generate the same hashcode. (We can tolerate some duplication, since the final step of every search is actually a full-text comparison.) Thus, every place I have spoken of 'words' above, I really meant 'hashcodes of words'.

Even with hashcodes instead of words, some steps of the indexing process are difficult to handle. Consider the Bible, for example. This is a 4.5M (uncompressed) file of nearly a million 'words'. The file will consist of about 560 8K-blocks. An 8K-block will contain a little over 1000 words, of which we may suppose that 500 of the words are unique (unique within the block, that is). Each unique word is assigned a 4-byte hashcode. Therefore, the index alone is 560*500*4=1.12M in size. As we casually noted above, this index must be re-sorted to give a list of blocks by word rather than words by block. This can't be done entirely in memory, since the index itself doesn't even fit in memory! And even if it could be handled in memory (by use of extended memory, for example), we couldn't guarantee that it would fit into memory on every computer. Therefore, disk-based external sorting routines have been developed that don't require the index to be in memory. The sorting routine, a mergesort, has a running time proportional to N *log* N, where N is the filesize.

The entire point of the re-sorting step is to keep us (during the search process) from having to examine the entire index (which is nearly half the .RIP-file size). Instead, we can do a binary search on the index to isolate those parts of it pertaining just to relevant words, and then

examining only that part of the index. A binary search can't be performed directly on the index as described above, though, since the records for each word are not the same length. (The record for each word is a list of blocks in which the word appears, and this clearly varies in size word by word.) Fixed-length records are necessary for a binary search. Therefore, to the index as already described we append a table of pointers to the records. Each entry in this pointer table consists of the hashcode of the word, along with a pointer into the index. Since these records are of fixed length, during the search process we can perform a binary search on the pointer table, and thereby determine which parts of the index to load and examine.

*Master index* .

The master index (RIP.IND) is just like the indices of the .RIP files, except that it gives the unique words by file rather than by block. Each file in the database partition is numbered, 0-65535, and RIP.FIL is used to relate the filenames to the numbers. However, the structure of RIP.IND and algorithms used to search RIP.IND are just like those of the indices to the individual RIP files.

The master index for a CD-sized database partition is about 30M, so all the comments above about the difficulty of sorting this file in-memory apply 30-fold.

## Preparation of Alternate-Language Databases

As mentioned earlier, if the statistical character of the characters in the text of the database differs significantly from that of 'typical' English, you will want to use a set of Huffman codes tailored to the particular language you are using. This entails replacing the RIP_ALLF.DAT file with a different file. In doing so, any new .RIP files you generate will be incompatible with standard files, and hence will be unreadable using a standard RIP_ALLF.DAT file. Similarly, if you already have some .RIP files, they will be unreadable with your new RIP_ALLF.DAT file.

You don't actually have to generate a new RIP_ALLF.DAT file, but rather files called

RIP_A.DAT
.
.
.
RIP_Z.DAT
RIPDIGIT.DAT
RIPPUNCT.DAT
RIPOTHER.DAT

These files will contain the Huffman codes for the various contexts. (They are in ASCII format, so on the off-chance you're interested, you can actually read them.)

Erase your existing RIP_ALLF.DAT file. The first time you run RIP after that, it will load the other *.DAT files (RIP_A.DAT etc.) and combine them to produce a new RIP_ALLF.DAT file (which is simply a much more compact form of the other files). You can then delete all the *.DAT files except RIP_ALLF.DAT.

How do you produce the *.DAT files? You use the HUFFMAN1 program. First, you have to produce a large ASCII file of typical text in your language. Each character must consist of single-byte codes. Two-byte character codes cannot be used. Make the file as large as you can, and don't just choose a single work. Combine a bunch of works to create the pattern file. The HUFFMAN1 program must then be run 29 times, once for each context (i.e., once for RIP_A.DAT, once for RIP_B.DAT, and so on).