GLIMPSE: A Tool to Search Through Entire File Systems

Udi Manber and Sun Wu

TR 93-34

October 1993

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

To appear in the
1994 Winter USENIX Technical Conference

GLIMPSE: A Tool to Search Through Entire File Systems

Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

Sun Wu

Department of Computer Science
National Chung-Cheng University
Ming-Shong, Chia-Yi, Taiwan
sw@cs.ccu.edu.tw

ABSTRACT

GLIMPSE, which stands for GLobal IMPlicit SEarch, provides indexing and query schemes for file systems. The novelty of *glimpse* is that it uses a very small index — in most cases 2-4% of the size of the text — and still allows very flexible full-text retrieval including Boolean queries, approximate matching (i.e., allowing misspelling), and even searching for regular expressions. In a sense, *glimpse* extends *agrep* to entire file systems, while preserving most of its functionality and simplicity. Query times are typically slower than with inverted indexes, but they are still fast enough for many applications. For example, it took 5 seconds of CPU time to find all 19 occurrences of Usenix AND Winter in a file system containing 69MB of text spanning 4300 files. *Glimpse* is particularly designed for personal information, such as one's own file system. The main characteristic of personal information is that it is non-uniform and includes many types of documents. An information retrieval system for personal information should support many types of queries, flexible interaction, low overhead, and customization, All these are important features of *glimpse*.

---

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, by NSF grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Part of this work was done while the author was visiting the University of Washington.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

1. Introduction

With the explosion of available information, especially through networks, organizing even one's own personal file system is becoming difficult. It is hard, even with a good organization, to remember things from a few years back. (Now where did I put the notes on that interesting colloquium two years ago?) After a while, not only the content of a piece of information can be forgotten, but also the *existence* of that information. There are two types of tools to search for patterns: grep-like tools, which are fast only if the search is limited to a small area, and index-based tools, which typically require a large index that needs to be computed ahead of time. *Glimpse* is a combination of the two approaches. It is index-based, but it uses a very small index; it assumes no structure from the data; and it allows very flexible queries including approximate matching.

The most common data structure used in information retrieval (IR) systems is an inverted index [SM83]. All occurrences of each word are stored in a table indexed by that word using a hash table or a tree structure. To reduce the size of the table, common words (e.g., *the* or *that* in English) are sometimes not indexed (although this cannot be done when the text is multi-lingual). Inverted indexes allow very fast queries: There is no need to search in any of the texts, only the table needs to be consulted and the places where the word occurs are retrieved immediately. Boolean queries are slower, but are still relatively fast.

The main drawback of inverted indexes for personal file systems is their space requirement. The size of the index is typically 50%-300% of the size of the text [Fa85]. This may not be a major drawback for commercial text databases, because disk space is relatively cheap, but it is a major drawback for personal information. Most users would not agree to double their disk cost for the benefit of indexing. Indeed today most personal file systems are not indexed. But, due to an increased availability of digital information through networks, many personal file systems are large enough to require IR capabilities. Signatures files [Fa85, GB91] have been suggested as an alternative to inverted indexes with good results. Their indexes are only 10%-30% of the text size. Their search time is slower than with inverted indexes, and since they are based on hashing, their parameters must be chosen carefully (especially for many files of different sizes) to minimize the false drops probability.

The second weakness of inverted indexes is the need for exact spelling (due to the use of hashing or tree structures to search for keywords in the index efficiently). If one is looking for all articles containing Schwarzkopf for example, any article with a misspelling will be missed (not to mention that the exact spelling is needed to form the query). The typical way to find misspelled words is to try different possibilities by hand, which is frustrating, time consuming, and is not guaranteed to succeed. In some cases, especially in large information bases, searching provides the only way to access information. A misspelling can cause a piece of information to be virtually lost. This is the digital equivalence of dropping a folder behind the file cabinet (except that there is no limit to the room behind this 'file cabinet,' and hardly anyone ever cleans there). This problem is expected to become more acute as more information is scanned by OCR (Optical Character Recognition) devices, which currently have an error rate of 1-5% [BN91, RKN92].

Glimpse requires a very small index, in most cases 2-4% of the original text, and it supports arbitrary approximate matching. As we will describe in detail in the next section, glimpse's engine is *agrep*, a search program that we developed [WM92a, WM92b], which allows the user to specify the number of allowable errors (insertions, deletions, substitutions, or any combination). The user can also search for the best match, which will automatically find all matches with the minimum number of errors. Because we use a small index, our algorithms are usually slower than ones using inverted indexes, but their speed is still on the order of a few seconds, which is fast enough for single users.

In some sense, *glimpse* takes the opposite extreme to inverted files in the time vs. space tradeoff (with signature files being in the middle). For some applications, such as management of personal information, speed is a secondary issue. Most users would rather wait for 10-15 seconds, or even longer, for a query than double their disk space. Even for IR systems, such as library card catalogs, where high throughput is essential, our scheme can be used as a secondary mechanism to catch spelling errors. For example, we found several spelling errors in a library catalog (see Section 4). We believe that this capability is essential in all applications that require a high level of reliability; for example, medical labs could miss information on patients due to misspelling.

We call our method *two-level searching*. The idea is a hybrid between full inverted indexes and sequential search with no indexing. It is based on the observation that with current computing performance, sequential search is fast enough for text of size up to several megabytes. Therefore, there is no need to index every word with an exact location. In the two-level scheme the index does not provide exact locations, but only pointers to an area where the answer may be found. Then, a flexible sequential search is used to find the exact answer and present it to the user. (Some other IR systems, such as MEDLARS [SM83] and STATUS [Te82], allow sequential search as a postprocessing step to further filter the output of a query, but the search relies on inverted indexes.) The idea of two-level searching is quite natural, and, although we have not found references to it, it has most probably been used before. The use of *agrep* for both levels — searching the index and then searching the actual files — provides great flexibility, in particular it allows approximate matching and regular expressions, as we will discuss later.

We have been using *glimpse* for several months and we are finding it indispensable. It is so convenient that we use it many times even when we know where the information is and can use *agrep* directly: It is just easier to quickly browse through the output than to `cd` to the right directory, `ls` to find the exact name of the file, `agrep` it, and `cd` back. We found ideas that we wrote many years ago, and not only forgot where we put them, but forgot about writing them in the first place. We found that we sometimes save information that we probably would have discarded without *glimpse*, because we now have some confidence of ever finding it again. An interesting example, conveyed to us by someone who tested a beta version of *glimpse*, was to find an e-mail address of a friend who moved recently; *glimpse* found it not in any mail messages, but in a call for papers! *Glimpse* can sometimes truly find a needle in a haystack.

2. The Two-Level Query Approach

In this section, we describe our scheme for two-level indexing and searching. We start with the way the index is built.

The information space is assumed to be a collection of unstructured text files. A text consists of a sequence of *words*, separated by the usual delimiters (e.g., space, end-of-line, period, comma). The first part of the indexing process is to divide the whole collection into smaller pieces, which we call *blocks*. We try to divide evenly so that all blocks have approximately the same size, but this is not essential. The only constraint we impose is that the number of blocks does not exceed $2^8 = 256$, because that allows us to address a block with 8 bits (one byte). This is not essential, but it appears to be a good design decision.

We scan the whole collection, word by word, and build an index that is similar in nature to a regular inverted index with one notable exception. In a regular inverted index, every occurrence of every word is indexed with a pointer to the exact location of the occurrence. In our scheme every word is indexed, but not every occurrence. Each entry in the index contains a word and the block numbers in which that word occurs.

Even if a word appears many times in one block, only the block number appears in the index and only once. Since each block can be identified with one byte, and many occurrences of the same word are combined in the index into one entry, the index is typically quite small. Full inverted indexes must allocate at least one word (4 bytes), and usually slightly more, for each occurrence of each word. Therefore, the size of an inverted index is comparable to the size of the text. But our index contains only the list of all unique words followed by the list of blocks — one byte for each — containing each word. For natural language texts, the total number of unique words is usually not too large, regardless of the size of the text.

The search routine consists of two phases. First we search the index for a list of all blocks that may contain a match to the query. Then, we search each such block separately. Even though the first phase can be done by hashing or B-trees, we prefer sequential search using *agrep*, because of its flexibility. With hashing or B-trees, only keywords that were selected to be included in the data structure can be used. With sequential search we can get the full power of *agrep*. Since the index is quite small, we can afford sequential search.[2]

Agrep is similar in use to other grep's, but it is much more general. It can search for patterns allowing a specified number of errors which can be insertions, deletions, or substitutions (or any combination); it can output user-defined records (e.g., paragraphs or mail messages), rather than just lines; it supports Boolean queries, wild cards, regular expressions, and many other options. Given a pattern, we first use *agrep* to find all the words in the index that match it. Then, using *agrep* again, we search the corresponding blocks to find the particular matches. The same procedure holds for all types of complicated patterns such as ones that contain wildcards (e.g., U..nix), a set of characters (e.g. count[A-E]to[W-Z], where [A-E] stands for A, B, C, D, or E), or even a negation (e.g., U[^l]..ix, where [^l] stands for any character except for l). These kinds of patterns cannot be supported by the regular hashing scheme that looks up a keyword in the table, because such patterns can correspond to hundreds or even thousands of possible keywords.

Boolean queries are performed in a similar way to the regular inverted lists algorithm. Suppose that the query is for *pattern1* AND *pattern2*. We find the list of all blocks containing each pattern and intersect them. Then we search the blocks in the intersection. Notice that even if we find some blocks that contain *pattern1* and *pattern2*, it does not mean that the query is successful, because *pattern1* may be in one part of the block and *pattern2* in another. *Glimpse* is not efficient for Boolean queries that contain very common words. The worst example of this weakness that we encountered was a search for ''linear programming.'' This term appeared in our files in several blocks, but to find it we had to intersect all blocks that contain the word 'linear' with all blocks that contain the word 'programming' which in both cases were almost all blocks.

The idea of using *agrep* to search the index can also be integrated with regular inverted indexes. It is possible to separate the list of words in an inverted index from the rest of the index, then use *agrep* to find the words that match the query (e.g., a query that allows some errors), then use the regular inverted index algorithm to find those words. We are not familiar with any system that provides approximate matching in this fashion.

In summary, we list the strengths and weaknesses of our two-level scheme compared with regular inverted indexes:

---

[2] Although the index is not an ASCII file, because the block numbers use all 256 byte values, the words themselves are stored in ASCII so *agrep* can find them.

Strengths

1. Very small index.
2. Approximate matching is supported.
3. Fast index construction.
4. No need to define document boundaries ahead of time. It can be done at query time.
5. Easy to customize to user preferences.
6. Easy to adapt to a parallel computer (different blocks can be searched by different processors).
7. Easy to modify the index due to its small size. Therefore, dynamic texts can be supported.
8. No need to extract stems. Subword queries are supported automatically (even subwords that appear in the middle of words).
9. Queries with wildcards, classes of characters, and even regular expressions are supported.

Weaknesses

1. Slower compared to inverted indexes for some queries. Not suitable for applications where speed is the predominant concern.
2. Too slow, at this stage (but we're working on it), for very large texts (more than 500MB).
3. Boolean queries containing common words are slow.

3. Usage and Experience

We have been using *glimpse* for a few months now, mostly on our own file systems. We have tried it on several other data collections, ranging in size from 30MB to 250MB. In this section, we discuss some of the current features of *glimpse* and our experience with it.

The current version of *glimpse* is a collection of many programs of about 7500 lines of C code. It is geared towards indexing a part of a file system. The statement

```
glimpse_index [-n] dir_name [dir_names]
```

indexes directory dir_name (or several directories) and everything below it. The most common usage is `glimpse_index ~`. The `-n` option indexes numbers as well. Indexing is typically done every night, and it takes about 6 minutes (elapsed time) to index 50MB of text or about 7-8 seconds per 1MB (using a DEC 5000/240 workstation).

Before indexing a file, the program checks whether it is a text file. If the file is found to have too many non-alphanumeric characters (e.g., an executable or a compressed file), it is not indexed, and its name is added to a `.log` file. Other formats, for example, 'uuencoded' files and 'binhexed' files, are excluded too. Determining whether a file is a text file is not easy. Texts of languages that use special symbols, such as an umlaut, may look like binary files. There are tools that do a good job of identifying file types, for example Essence [HS93], and we will eventually incorporate them in *glimpse*. On the other hand, some ASCII files should not be indexed. A good example is a file containing DNA sequences. We actually have such files and found them to cause the index to grow significantly, because they contain a large number of essentially random strings. We should note that the two-level scheme is not suitable for typical biological searches, because they require more complicated types of

approximate matching. Indexing numbers is useful for dates and other identifying numbers (e.g., find all e-mail messages from a certain person during July 1991). But large files with numeric data will make the index unnecessarily large. *Glimpse* allows the user to specify which files should not be indexed by adding their names to a `.prohibit` file. We plan to add more customized features to the indexing process.

The partition into blocks is currently done in a straightforward way. The total size of all text files is computed, and an estimate on the desired size of a block is derived. Files are then combined until they reach that size, and a new block is started. We plan to improve this scheme in the future in two ways: 1) the partition should be better adapted to the original organization of the data, and 2) the user should have the ability to control how the partition is done.

The user interface for *glimpse* is similar to that of *agrep*, except that file names are not required. The typical usage is

```
glimpse information
```

which will output all lines *in all indexed files* that contain *information*;

```
glimpse -1 HardToSpell
```

will find all occurrences of *HardToSpell* with one spelling error.

```
glimpse -w 't[a-z]j@#uk'
```

will find all email addresses in which a 3-letter login name starts with t and ends with j (it is typical not to know the user's middle name) and uk is somewhere in the host name (in *agrep* the symbol # stands for arbitrary number of wild cards, and the -w option specifies that the pattern must match a complete word).

Glimpse supports Boolean queries the same way that *agrep* does (with ; serving as AND):

```
glimpse -1 'Lazoska;Zahorian'
```

will find all occurrences of both names (allowing one spelling error, which is needed).

```
glimpse 'Winter Usenix'
```

will first search the index for the two words separately and find all the blocks containing both words, then use *agrep* for the whole phrase.

Another nice feature of *glimpse* is to limit the search to files whose names match a given regular expression.

```
glimpse -F 'haystack\.c$' needle
```

will find all needles in all haystack.c files. Sometimes, you want to search everywhere for that elusive definition:

```
glimpse -F '\.h$' ABC_XYZ
```

will search all .h files.

```
glimpse -F '\.tex$' 'environment;^\\'
```

will find all occurrences of *environment* in a line that starts with \ (useful if you want to see environment as part of a definition). *Glimpse* is in fact so flexible that it can be used for some file management operations; for example,

```
glimpse -F mbox -h -G  . > MBOX
```

will concatenate all files whose name is mbox into one big one;

```
glimpse -c -F 'pattern\.h$' '\#define'
```

will provide a list of all pattern.h files, each followed by the number of #define's it has. We even allow errors in the regular expressions for the file names, using, for example, `-F1` for one error.

4. Performance

All the performance numbers given below are anecdotal. The point of this section is not to compare the running time of *glimpse* to other systems, but to argue that it is fast enough for its purposes. Query times depend heavily on the type of indexed data and the query itself (not to mention the architecture and the operating system). If the pattern appears in only one block, the search time will be almost always fast no matter how large the data is or how complex the pattern may be (an unsuccessful query is the fastest because only the index needs to be searched). If the pattern matches a large portion of the blocks, the query may be slow. Searching in a large information space requires a careful design of queries. No matter how fast the search is, if the number of matches is large, it will take too long to sift through them. However, we found that even matching to common patterns can be useful, because one starts to see many matchings quite fast, and one can then stop and adjust the pattern. In some cases, even though there were many matches, the first few ones were sufficient to get the answer we were looking for. (We also added an option -N that tells the user the number of blocks that would need to be searched.)

Below we list some running times (on a DEC 5000/240 workstation) for a personal file system containing 69MB of text in 4296 different files. It took 4.9 minutes of CPU time (9 minutes of elapsed time) to index it (including the time to determine that 857 other files were not text files) and the index size was 1.9MB, which is 2.7% of the total. 205 blocks were used. A typical search takes from 2-10 seconds. For example, a search for *biometrics* took 1.6 seconds. (The numbers listed here are the sum of the user and system times.) A search for *incredible* took 2.8 seconds (there were 27 matches in 22 files). A search for *Checoslovakia* allowing two errors took 3.6 seconds. A search for *Usenix* took 13.9 seconds because there were 93 matches divided among 70 different blocks. A Boolean search can sometimes help by limiting the number of scanned blocks (which are only those that match all patterns), but again it depends on the number of matches. For example, searching for Usenix AND Winter took only 4.5 seconds since there were only 19 matches. A search for Lazowska AND Zahorjan with one error took 4.7 seconds. A search for protein AND matching took 7.7 seconds because both patterns were very common. The search for the e-mail address mentioned above (t[a-z]j@#uk) took 30 seconds (although we believe that some of it is due to a bug that for some complicated regular expressions causes search in some unnecessary blocks), but this is still much better than any alternative.

For a much larger information space, we indexed an old copy of the entire library catalog of the Univeristy of Arizona. This experiment and the next one were run on a SUN SparcStation 10 model 512 running Solaris. The copy we used lists approximately 2.5 million volumes, divided among 440 files occupying 258MB. The index size was 8MB, which is 3.1%. The index would have been much smaller for regular text; the catalog contains mostly names and terms, in several languages, and since it uses fixed records, quite a few of the words are truncated. It took 18 minutes (all times here are elapsed times) to index the whole catalog. A search for Manber took 1 second (one match). A case insensitive (-i) search for usenix yielded one match in 1.8 seconds. A search for UNIX took 3.9 seconds (138 matches in 8 blocks). A search for information AND retrieval took 14.9 seconds; there were 38 matches in 9 blocks, but 31 blocks had to be searched (because they included both terms).

Searching for ‘algorithm’ allowing two errors took 16.3 seconds, finding 269 matches, 21 of which did not match the pattern exactly due to a misspelling (alogorithm, Alogrithms) or a foreign spelling (e.g., algoritmy).

An example that highlighted the ‘‘best case’’ for *glimpse* was a directory containing the archives from comp.dcom.telecom. They occupy about 71MB divided among 120 files. The small number of files minimizes the overhead of opening and closing files making *agrep*, and as a result *glimpse* very fast. Even without *glimpse*, searching for Schwarzkopf allowing two misspelling errors took *agrep* only 8.5 seconds elapsed time. It took *glimpse* 3:38 minutes to index the directory and the index size was 1.4MB (about 2% of the total). With *glimpse*, the Schwarzkopf search took 0.4 seconds (all matches were in one file). A search for Usenix (no error allowed), which appeared in 5 blocks, took 0.9 seconds. A search for glimpse took 3.7 seconds (15 matches in 9 blocks).

5. Future Work

We list here briefly some avenues that we are currently exploring.

5.1. Searching Compressed Text

If the text is kept in a compressed form, it will have to be decompressed before the sequential search can be performed. This will generally slow down the search considerably. But we are developing new text compression algorithms that actually *speed up* the search while allowing compression at the same time. The first algorithm [Ma93] allows *agrep* (and most other sequential search algorithms) to search the compressed file directly without having to decompress it first. Essentially, instead of restoring the text back to its uncompressed form, we modify the *pattern* to fit the compressed form and search for the modified pattern directly. Searching the compressed file directly improves the search time (as well as space utilization, of course), because there is less I/O. In preliminary tests, we achieved about 30% reduction in space and 25% decrease in search time. This allows files to be kept in a compressed form indefinitely (unless they are changed) and to be searched at the same time. In particular, we intend to compress the index used in the two-level scheme, because it is searched frequently. We are working on another compression scheme that will be integrated into *glimpse*. This work is still in a very preliminary stage. The compression rates, for natural language texts, seem to be in the 50% range. It also allows fast search without decompression, although it is too early to predict its speed.

5.2. Incremental Indexes

The two-level index is easier to modify and adapt to a dynamic environment than a regular inverted index, because it is so much smaller. To add a new file to the current index, we first add the file to an existing block or create a new block if the file is large enough or important enough to deserve it. Then we scan the index and add each word in the new file that does not already appear in the block. Since the index is small, reading it from disk and writing it back can be done very fast. Deletion of a file is slightly more time consuming because for each word we need to check the whole block to determine whether that word appears somewhere else (and thus should not be deleted). Fortunately, *agrep* contains a very powerful algorithm for multi-pattern matching. It can search for a large collection of words (up to 30,000) concurrently.

Incremental indexing will be essential for indexing newsfeed. We are considering adapting *glimpse* to allow searching usenet netnews. Currently, the total size of a typical usenet server is from 500-800MB. Quite a bit of it is not text but images and programs, so it is a size we can handle. The problem is that this kind of newsfeed consists of a large number of small files (individual email messages) stored at random places on the

disk. A better data organization will probably be required for reasonable performance.

5.3. Customization

There are several ways to let the user improve the indexing and searching procedures by customizing. The user should be able to decide whether certain patterns are indexed or not. We provide an option to index numbers, but we should allow more flexibility. One way is to provide a hook to an external filtering program, provided by the user, which will decide what to index based on contents, type of file, name of file, etc. We also plan to allow the user to store a large set of multi-words phrases (e.g., "fiber optics," "Jonathan Smith, "May 1992") that will be indexed together, allowing quick search for them without the need for Boolean queries. Another option is to partition the collection of files into categories and build separate indexes for each (e.g., correspondence, information from servers, program source codes). This is not needed for small to medium size file system, but may be essential for large file systems. We also plan to support access to any special structure or additional information associated with the text. For example, some searches may specify that the desired information starts at column 30 on the line or in the second field. The user may want to search only small files (say, below 20KB) or recent files (say, after August 1992).

There should be more ways to view the output. For example, for queries that give many matches, the user may be interested first in a rough idea of where those matches are (e.g., only directory names). We currently have an option [-c] to list only the files containing a match along with the number of matches, and another option [-N] to output only the number of blocks with potential matches. These options, and many more, can be incorporated in *glimpse* rather easily. We believe that customization is the key to better search facilities.

Acknowledgements

Burra Gopal found and corrected several bugs in the code. We thank Greg Andrews, George Forman, and Trevor Jenkins for comments that improved the manuscript.

References

[BN91]

Bradford R., and T. Nartker, ‘‘Error correlation in contemporary OCR systems,’’ *Proc. of the First Int. Conf. on Document Analysis and Recognition*, (1991), pp. 516–523.

[Fa85]

Faloutsos C., ‘‘Access methods for text,’’ *ACM Computing Surveys*, **17** (March 1985), pp. 49–74.

[GB91]

Gonnet, G. H. and R. A. Baeza-Yates, *Handbook of Algorithms and Data Structures* (Chap. 7.2.6) Second Edition, Addison-Wesley, Reading, MA, 1991.

[HS93]

Hardy D. R., and M. F. Schwartz, ‘‘Essence: A resource discovery system based on semantic file indexing,’’ *Proc. of the USENIX Winter Conference*, San Diego (January 1993), pp. 361–374.

[Ma93]

Manber U., ‘‘A text compression scheme that allows fast searching directly in the compressed file,’’ technical report 93-07, Department of Computer Science, University of Arizona (March 1993).

[RKN92]

Rice, S. V., J. Kanai, and T. A. Nartker, ‘‘A report on the accuracy of OCR devices,’’ Technical Report, Information Science Research Institute, University of Nevada, Las Vegas, 1992.

[SM83]

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

[Te82]

Teskey, F. N., *Principle of Text Processing*, Ellis Horwood Pub., 1982.

[WM92a]

Wu S. and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ *Usenix Winter 1992 Technical Conference*, San Francisco (January 1992), pp. 153–162.

[WM92b]

Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the ACM* **35** (October 1992), pp. 83–91.