

# File indexing and searching for Plan 9

*Francisco J Ballesteros  
Laboratorio de Sistemas  
Universidad Rey Juan Carlos*

## ABSTRACT

In Plan9 most resources are provided as files, besides regular on disk stored files. When the set of files grow, it is important to be able to quickly locate files based on their contents. This paper describes the set of tools documented in *tags(1)*, which provides file indexing and content based searching for Plan 9, using a file system to provide the search interface.

## Introduction

It is common today in most systems to be able to search files based on their content. MacOSX has the Spotlight tool, Google supplies a file indexing and searching tool for various systems including Windows and Linux, and UNIX have had since long ago tools like *whereis* and *Glimpse* that permit searching files by name or by content.

The native version for Plan 9 of such tool was missing, although APE can be used to port others from other systems. We describe here our second attempt at providing content based file searching for Plan 9, after early experiments in Plan B with *adb(1)*, a simple tag database.

Content based searching requires solving two problems: (1) indexing file contents and (2) searching the indexes to execute queries. How both things are done depends on the type of searches to be supported. The tools described here provide only exact word matching. Approximate text matching would require different techniques and has not been considered yet. You can refer to tools like descendants of *Glimpse* to learn such techniques.

For exact word queries, it is important to be able to extract the words of interest (probably all) from a file. The appropriate way of doing this is specific for each type of file. Therefore the indexer has to be modular and permit the addition of particular word extractors for different file types.

Regarding searching, there is a compromise between maintaining the database in memory, leading to fast searches and to high memory consumption, and maintaining most of it on disk, which leads to slower searches but minimizes memory consumption.

Another issue is that on Plan 9, as of today, it is not uncommon for a user to have multiple terminals. At the very least, it is very common (by design) to have multiple terminals sharing a central file server and several CPU servers. This may be exploited to use otherwise idle machines to help in file searching, while keeping most of the terminal resources available for other uses.

It is also important for the indexer to support quick updates to the indexes, otherwise changes to the file system would not be able to be incorporated quickly to the database and the users would miss most of the recent changes made to the file system. Considering that many searches refer to things just made, the importance of this point

increases.

In Plan 9 users rearrange their namespaces so that they incorporate possibly many file trees, from different servers. However, the indexing tool must keep file paths in a consistent way for all the name spaces. Also, it is important to run the indexer utility close to the file server, to reduce I/O latencies. As a result, the search database is best associated to a particular file tree (at a single file server), instead of being associated to a namespace. In that way the database used determines the file tree where the files reside, and paths are not ambiguous anymore (because they are not a function of the client's name space).

### Data structures

The set of tools described in *tags(1)* are centered around two data structures:

- 1 A trie that maps from words to file qids.
- 2 A hash table that maps from qids to file names.

Keeping the index and the file names appart was made because Qids are more compact. Keeping just them as the values in the trie instead of keeping strings permits the Trie to be more compact. This is important because the memory occupied by the database is significant, and the more compact it is, the better. We can do this because the database is to refer to a file tree at a particular server, to keep paths meaningful as said in the introduction.

The trie is stored in a single file, e.g., `/lib/sys.trie.db`, and is fully read on main memory by the tools using it. The hash table is kept at a separate file, e.g., `/lib/sys.hash.db`, and is also read fully onto main memory by the tool supporting it.

All the words used to lookup files are kept in the trie. This means all the words contained in text files, and all the words extracted from other file types. As an aid, each file is considered to contain the path elements present on its name. This permits looking for `sys src` to focus searches on system source files, for example.

Each node in the trie represents a prefix (or a full word). For example, the root node corresponds to the empty word. A trie node is described by the following structure:

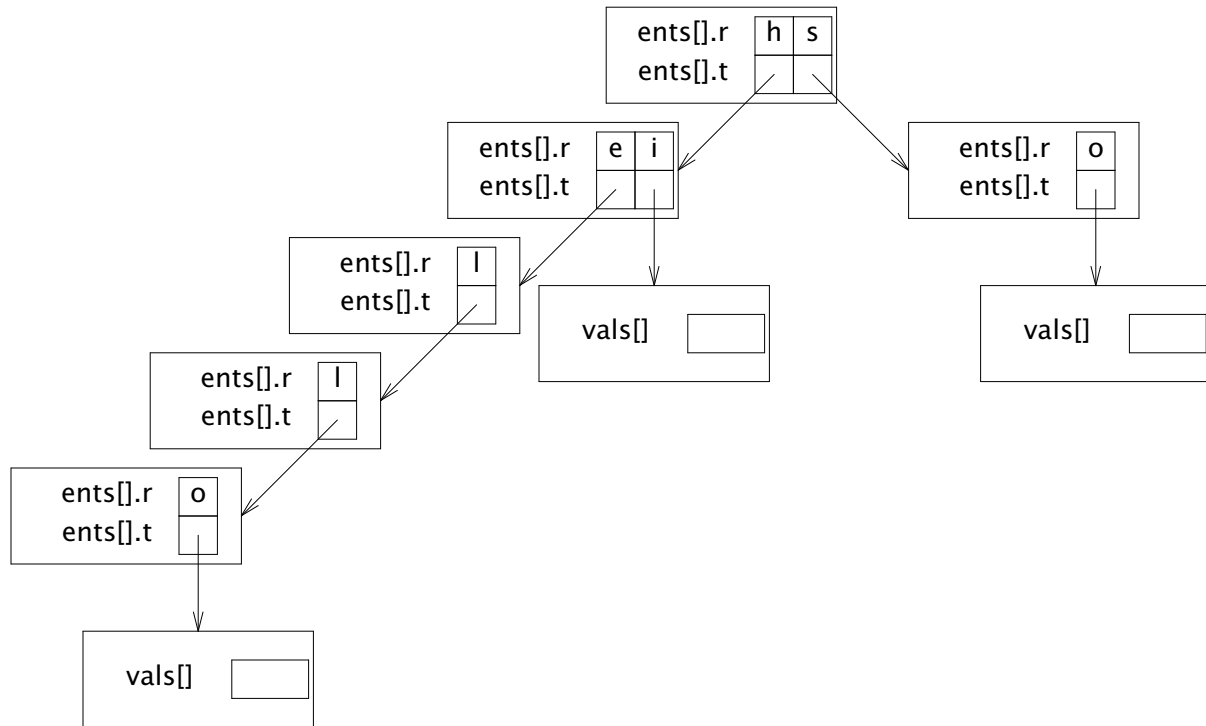
```
typedef struct Trie Trie;
typedef struct Tent Tent;

struct Tent {
    Rune    r;
    Trie*   t;
};

struct Trie {
    Tent*   ents;    // ents[i].r are runes for childs
    int     nents;   // ents[i].t are children
    int     aents;   // # of ents allocated
    uulong* vals;    // values for this node prefix
    int     nvals;   // # of values in use
    ulong*  svals;   // small values (fit in a long)
    int     nsvals;  // # of small values in use
};
```

A node `t` maintains pointers (in `t->ents[i].t`) to child nodes that represent a longer words sharing the prefix that `t` represents. Each link to a child node is labelled (by `t->ents[i].r`) by the rune that has to be added to the prefix represented by `t` to obtain the prefix represented by the child. For example, the prefix `a` representing

either the word *a* or all the words starting with *a* would be represented by the child  $t \rightarrow \text{ents}[i].t$  of the root node, provided that  $t \rightarrow \text{ents}[i].r$  in the root node contains the rune *a*. Figure 1 depicts an example trie keeping the words (and their prefixes) *hi hello* and *so*.



Because the trie is used to map words to Qids, each node  $t$  that (besides a prefix) represents a word valid as a index for a set of files, contains in  $t \rightarrow \text{vals}$  an array of Qids. For example, in the figure, the two nodes holding the *o* rune would point to further trie nodes, used just to contain the Qids for files tagged with *hello* and *so*. In the same way, the entry for *i* in the left child of the root node would point to another trie node used just to contain the Qids for files tagged with *hi*. Note that all nodes in the figure are similar but in each node, we do not depict arrays that are empty, for clarity.

Both the *ents* and *vals* arrays are grown dynamically, as more space is needed. For *ents*, *nents* records the number of entries used and *aents* records the number of allocated entries (because in the future we might allow to delete entries in this array). For *vals* we grow the array in chunks of *Incr* nodes (a constant in the program) and there is no need to keep *avals*. The array  $t \rightarrow \text{svals}$  is an optimization, discussed later.

Note that this implementation leads to a flat trie, whose depth is only as long as the longest word. Other implementations would use inner trees (mixed within the trie) to keep children conceptually contained at a single node in the trie.

We thought that it was best to keep the entire database in memory, to permit fast searching. That is considering that memory is cheap and that we may also be able to use the memory of a shared machine to keep the database there. Nevertheless, the database must be kept as compact as feasible while on memory to avoid consuming all the memory available.

Therefore, using a single array to keep all the pointers to the children (and all the values) seemed a sensible thing to do. It permits a compact representation where space for further pointers is not necessary.

Both arrays are kept sorted, which means that searches on them are still logarithmic. Additions to the trie are not that frequent and do not require fast response times, compared to searches.

Many Qids fit in a `long` value, and the trie stores them in `svals` instead of doing it in `vals`. That way we use half the size for such Qids, at the expense of maintaining two more words at each node in the Trie, to maintain the array. We tried both with and without this optimization and the difference is about 50 Mbytes of main memory for our system database. In the future, if many Qids use the high long of their path, the two extra words may turn into a penalty. Searching time is not affected by this optimization.

The data structure mapping Qids to paths is a simple hash table. There is not much to say about it, other than showing the structure itself:

```
enum {
    Nhash = 512 * 1024,    // # of buckets
};

typedef struct Ent Ent;

struct Ent {
    uulong   qid;
    char*    path;
    Ent*     next;    // in hash
};

Ent*       hash[Nhash];
```

Searching for lines matching the given query relies on a double search. First, the inverted index implemented by the trie reports the files relevant to the query. Second, `grep(1)` is used to search such files and show the lines relevant.

One point of interest in both data structures is that neither one supports removal of entries. Removing qids from the trie would require iteration over all the nodes in the trie, which is utterly expensive. Instead, the tool implementing the hash table checks that the files looked up exist, before printing their paths. If they are removed they are simply discarded.

Also, if a file no longer contains a tag, it is still indexed by the tag. In any case, the tag is related to the file (because it did contain it). The search interface relies on `grep(1)` to show lines that match the query on the files retrieved from the database. If a tag is no longer in a file, no lines will be shown for such tag. This makes the problem of old tags mostly irrelevant.

From time to time, (say, once per month), the database is to be regenerated to clean it up. That is the price for avoiding the time to remove entries while re-indexing files.

## Tools

The software for indexing and searching files is split in different tools. This is also described in `mktags(1)`. This is the synopsis:

```
[ DB= dbpath ] looktags [ -n ] tag...
mktags [ -d ] dbpath file...
tagfiles [ -d ] triepath file...
rdtrie triepath [ tag... ]
qhash [ -dv ] hashpath [ qid... ]
qhash [ -dv ] -a hashpath [ qid path... ]
```

**qhash** [ **-dv** ] **-c** *hashpath file...*

**tagfs** [ **-abcD** ] [ **-s** *srv* ] [ **-m** *mnt* ] *triepath*

The first two programs are Rc scripts providing the primary user interface. The other programs provide the actual software for indexing and searching.

Mktags creates a database named *dbpath* that maps from tags (words) to file names. Only given *files* are indexed (including subdirectories as well). Any word in the path name for a file, and any word contained in the file (for most files) is a valid search tag for the file. A database is made of two files: a trie and a hash table. The name of the trie has the suffix *.trie.db* and the name of the hash has the suffix *.hash.db*. The path to the database (files) without any suffix is considered the name of the database.

By convention, there is a system wide data base at */lib/sys* (that is, */lib/sys.trie.db* and */lib/sys.hash.db* ) and a per-user data base at *\$home/lib/\$user* (that is, *\$home/lib/\$user.trie.db* and *\$home/lib/\$user.hash.db*).

Looktags searches the system and the user databases for files that match the query specified by its arguments. By default, only file names are printed. Flag **-n** instructs looktags to run *grep(1)* to print some of the matching lines.

A query is made of lists of tags separated by the ":" character, each as a distinct argument. A file matches the query if it is associated (contains) to all the tags on one of the lists. For example,

```
looktags a b c : d e
```

would search for files either matching all of a, b, and c or matching all of d and e.

Looktags can be instructed to use a different database by defining the DB environment variable to contain a list of names for the databases to be used (without any file name suffixes).

To speed up searches, the trie part of the database can be kept in memory using tagfs. When using a database named */a/b/dbname* the program looktags searches first for a file named */srv/dbname.tagfs* (to reach a server holding an in-memory version of the trie part of the database), and uses it instead if available. Otherwise, looktags looks for the host identified by *\$search* in the *ndb(6)* database. Should it be found, looktags imports its */srv* to look for */srv/dbname.tagfs* on it. This is used to share an in-memory database among several machines sharing a network. Only as a last resort would looktags read the database by itself to execute the query.

Tagfiles tags every *file* mentioned (recurring for directories) as an argument using the Trie stored in the file *trie*. Here, *trie* must include the *.trie.db* suffix if any. Mktags relies on this program.

For each file indexed, tagfiles uses every word in its path name as a tag to search for the file. Also, tagfiles looks at the file name suffix and uses *file(1)* to determine the type of file and pick a particular indexing method. For text files, tagfiles reads entire file contents and associates each word contained in the file as a tag to search for the file. For other types of file, tagfiles tries to execute external programs to extract the list of tags for each file. Should the appropriate external program not exist, tagfiles would still try to index the file as text when appropriate.

The following programs may be executed by tagfiles to obtain tags for files. They are expected to write tags for the file given as an argument, one per line:

- tagc to tag C source.
- taglimbo to tag Limbo source.

- `taghtml` to tag HTML files.
- `tagman` to tag manual pages
- `tagrc` to tag Rc scripts
- `tagtroff` to tag roff source.
- `tagdoc` to tag Microsoft Office documents, including rich text format.
- `tagpdf` to tag Adobe PDF files.
- `tageps` to tag Adobe EPS files.
- `tagps` to tag PosctScript files.

`Rdtrie` can be used to inspect and query the Trie in the database. The Trie data structure keeps all the known tags in a trie, maintaining a list of Qids for each tag.

Without any *tag* argument in the command line, `rdtrie` reads and prints the entire Trie file, *trie*. Otherwise, `rdtrie` reads *trie* and then interprets any following arguments as a query. The Qid matching the query are printed in the standard output. See above for the syntax of queries. `Looktags` relies on this program to execute its query.

`Qhash` maintains a file name hash table in the database. This data structure is used to translate Qids into file names. In what follows, Qid means actually the `Qid.path` field of the file's Qid, in base 16. Also, the argument *hash* is mandatory and has to be the path for the hash file in the database, including the `.hash.db` suffix.

The first invocation syntax (without using flags `-a` or `-c`) can be used to retrieve the path names for the given *qids* in the command line. This is used by `looktags` to retrieve the paths for matching files.

Under flag `-a` the program `qhash` adds the following argument pairs (each with a *qid* and *path*) to the *hash* file.

Under flag `-c` `qhash` retrieves Qids and (absolute) path names for *file(s)* mentioned as arguments (recurring for directories), and adds them to the database. This is used by `mktags` to create/update the hash file in the data base.

In any of the programs above, flags `-d` and `-v` (when available) enable certain debug messages to track problems while using the programs.

The program `tagfs` can be used to update a Trie and is an alternative to `rdtrie` to perform searches by keeping the entire Trie in memory. It is a file system that serves by default the pipe at `.tagfs/srv/triename` (where *triename* is the base name of the *triepath* without suffixes), mounting itself at `/mnt/tags`. Flags `-s` and `-m` can be used to instruct `tagfs` to serve *srv* instead or to mount itself at *mnt* instead.

The single directory served contains a `ctl` file that can be read to gather statistics about the Trie and can be written to modify the trie. A write of the string `sync` writes the in-memory database back to its file. A write of the form `tag qidpath tag ...` adds any *tag* to *qidpath* in the trie (but does not update the on-disk database).

A query can be made by creating a file, writing the query into it (being careful to separate different tags and `:` characters with white space), and then reading from the same file the list of qids that match the query. The query file is removed as soon as it is closed after having read from it.

## Examples of use

Create the per-user and the system database:

```
; mktags $home/lib/$user $home /mail/box/$user/msgs
; mktags /lib/sys /cfg /rc /sys
```

Look for files mentioning either list append or queue append, then repeat query but using an alternate database kept at `/lib/other.trie.db` and `/lib/other.hash.db`:

```
; looktags list append : queue append
; DB=/lib/other looktags list append : queue append
```

Add (or update!) tags for files under `/usr/prof` to the personal database:

```
; tagfiles $home/lib/$user.trie.db /usr/prof
; qhash -c $home/lib/$user.hash.db /usr/prof
```

Place the system database in memory so that looktags can be faster, and add the tag `yoyoba` to file with `qid 8345f`

```
; tagfs /lib/sys.trie.db
; echo tag 8345f yoyoba >/mnt/tags/ctl
; echo sync >/mnt/tags/ctl
```

Make the system database at `whale.lsub.org` available to other hosts: First, edit `/lib/ndb/local` to contain `search=whale.lsub.org` for the network entry. Second, at `whale`:

```
whale% tagfs /lib/sys.trie.db
whale% chmod a+rw /srv/sys.tagfs
```

Now from other hosts, looktags may use Whale's in-memory database.

### Heuristics

The most important heuristic is the one used by `tagtext` within `tagfiles` to determine which pieces of text are words. In particular, words of less than three characters are ignored. Also, pieces of non-blank text of more than 50 characters are considered as non-text (see for example encoded attachments in mails). The remaining text is parsed to locate alphanumeric words to be used as tags.