# A Text Retrieval Package for the Unix Operating System

*Liam R. E. Quin,*
*SoftQuad Inc. (lee@sq.com)*

**Abstract**

This paper describes *lq-text*, an inverted index text retrieval package written by the author. Inverted index text retrieval provides a fast and effective way of searching large amounts of text. This is implemented by making an index to all of the natural-language words that occur in the text. The actual text remains unaltered in place, or, if desired, can be compressed or archived; the index allows rapid searching even if the data files have been altogether removed.

The design and implementation of *lq-text* are discussed, and performance measurements are given for comparison with other text searching programs such as *grep* and *agrep*. The functionality provided is compared briefly with other packages such as *glimpse* and *zbrowser*.

The *lq-text* package is available in source form, has been successfully integrated into a number of other systems and products, and is in use at over 100 sites.

## 1. Introduction

The main reason for developing *lq-text* was to provide an inexpensive (or free) package that could index and search fairly large corpora, and that would integrate well with other Unix tools.

*Low Cost Solution:* There are already a number of commercial text retrieval packages available for the Unix operating system. However, the prices for these packages range from Cdn$30,000 to well over $150,000. In addition, the packages are not always available on any given platform. A few packages were freely available for Unix at the time the project was started, but generally had severe limitations, as mentioned below.

*A tool for searching large corpora:* Some of the freely available tools used *grep* to search the data. While *lq-text* is $O(n)$ on the number of matches, irrespective of the size of the data, *grep* is $O(n)$ on the size of the data, irrespective of the number of matches. This limits the maximum speed of the system, and searching for an unusual term in a large database of several gigabytes would be infeasible with *grep*. Other packages had limits such as 32767 files indexed or 65535 bytes per file. Tools available on, or ported from, MS/DOS were particularly likely to suffer from this malaise.

*Unix-based:* Unix has always been a productive text processing environment, and one would want to be able to use any new text processing tools in combination with other tools in that environment.

### 1.1 Results and Benefits

Timings are given in detail below, along with a perspective on how the above goals were met. As a brief example, a search over the SunOS 4.1.1 manual pages for “\<core dump\>” on a Sun 4/110 took 52 seconds with *grep*, and 1·3 seconds with *lq-text*. In addition, *lq-text* automatically finds matches of a phrase even if there is a newline or punctuation in the text between two words. It is also possible to combine *lq-text* searches, finding documents containing all of a number of phrases. Complex queries can be built up incrementally.

## 2. Design Goals

The main goals of any text retrieval package were outlined briefly in the introduction. *lq-text* was developed with some more specific goals in mind, described in the following sections.

### 2.1 Limited Memory

Developed originally on a 4MByte 386 under 386/ix, *lq-text* does not assume that *any* of its index files will fit completely into memory. As a result, indexing performance does not degrade significantly if the data does not fit into main, or even virtual, memory.

### 2.2 Offline Storage

Once files are indexed, *lq-text* does not need to consult them again during the searching process. The result of a query is a list of matching files, together with locations within those files. The original text files are needed if the user wants to view the matched documents, but it turns out that the file names and document titles are often sufficient.

### 2.3 Matching With Certainty

Packages that use hashing or probablistic techniques often return results that *might* match the user's query. A 'bad drop scan' is then used to reject the false hits [Lesk78]. These techniques are incompatible with the Offline Storage requirement, since a bad drop scan may not be possible.

### 2.4 Accurate Phrase Matching

The package should be able to match a phrase word for word, including getting the words in the right order and coping with uneven whitespace. It should also be able to match capitalisation and punctuation at least approximately, so that the user can constrain a search on someone's name, *Brown* for example, to match *Brown* but not *brown* in the text. Only the first letter of each word is inspected for capitalisation, in order to minimise the data stored in the index. This is sufficient for most queries.

In the literature, the terms *recall* and *precision* are used to refer to the proportion of relevant matches retrieved and the proportion of retrieved matches that are relevant, respectively; the goal of *lq-text* is to have very high precision, and to give the user some control over the recall. These terms are defined more precisely in the literature [Clev66], [Salt88], and are not discussed further in this paper. The term *accuracy* is used loosely in this paper to refer to literal exactness of a match—for example, whether *Ankle* matches *ankles* as well as *ankle*. In an inaccurate system, *Ankle* might also match a totally dissimilar word such as *yellow*. The information that *lq-text* stores in the database enables it to achieve a high level of accuracy in searching for phrases.

### 2.5 Updatable Index

It should be possible to add new documents to the index at any time. It should also be possible to *unindex* documents, or to *update* documents in place, as well as to inform the package when documents are renamed or moved.

### 2.6 Unix toolkit approach

It should be possible to manipulate search results with standard Unix tools such as awk and sed. This must be done in a way consistent with the Offline Storage requirement. For example, the user should be able to see the first seven matches using `head -7` or `sed 7q` without having to search the documents themselves.

### 2.7 Summary

This paper shows how some of the goals have been met, and indicates where work is still in progress on other goals. All of the original design goals are still felt to be relevant. With continued improvements in price/performance ratios of disks, the offline storage goal may become less important, but the design philosophy is still very important for CD/ROM work.

## 3. Technology Overview

This section gives a brief background to some of the main approaches to text retrieval.

### 3.1 Signatures

Packages based on *signatures* keep a hash of each document or block of each document. The idea is to reduce I/O by identifying those blocks which *might* contain a given word. This method does not store enough information to match phrases precisely, and software relying on it needs to scan documents to eliminate bad drops [Salt88], [Falo85], [Falo87a], [Falo87b]. A widely distributed publicly available system, *zbrowser*, uses a cross between block signatures and the Document Vector method discussed below. This system can answer proximity-style queries (these two words in either order, near each other) fairly well, but does not handle searching for phrases [Zimm91].

### 3.2 Full Text Inverted Index

A *full text inverted index* consists of a record of every occurrence of every word, and hence is generally the largest in size of the indexes discussed; the index also allows the highest *accuracy* (but not necessarily highest precision, see Future Work below). The larger index increases I/O needed for searching, but on the other hand there is no need to scan documents for bad drops [Salt88], [Mead92].

### 3.3 Document Vector

This strategy keeps a record of every file in which each word appears; one could call it a *partial inverted index*. This is usually much smaller than a full text inverted index, but cannot be used to find phrases directly without a bad drop scan. A recent example is *Glimpse*; this and other examples are mentioned in [Salt88], [Mead92], and [Orac92]. *Glimpse* also appears to be restricted to searching a single file system at a time on a local machine.

### 3.4 Relational Tables

One way of implementing a full or partial inverted index is by storing each occurrence of each word in a cell of a relational table. This is generally by far the least efficient of the strategies discussed, with indexes typically three times the size of the data, but it is also the easiest to implement robustly.

### 3.5 DFA or Patricia Tree

These systems store a data structure representing a deterministic finite state automaton that, when executed against the query, will reach an 'accept' state representing all matches. They are usually byte rather than word oriented, although they can be written either way. It is difficult to allow updates to such an index, and the algorithms are fairly complex. Knuth describes Patricia trees in some detail [Knut81]; a sample in-memory implementation, *Cecilia*, was described by Tsuchiya [Tsuc91]; PAT, the Oxford English Dictionary (OED) software, also uses Patricia trees [Bray89], [Fawc89].

### 3.6 Other

Many other approaches are possible. There are several schemes that actually replace the original data with, for example, a multiply-threaded linked list, so that the data can be recreated from the index. This has an unfortunate and well-known failure mode, in which the reconstituted text uses incorrect words. Other schemes include sub-word indexing, either on individual bytes or on *n-grams*, although these usually fall into the 'DFA' category above in terms of their characteristics.

## 4. The *lq-text* Design

A full text inverted index was chosen to meet the design goals. In particular, this is the only strategy which allows accurate matching of phrases without reverting to a bad drop scan.

In order to make the index smaller, however, the list of matches for each word is compressed, as described in detail in the Implementation section below.

The package is implemented as a C API in a number of separate libraries, which are in turn used by a number of separate client programs. The programs are typically combined in a pipeline, much in the manner of the probabilistic inverted index used by *refer* and *hunt* [Lesk78].

The *lq-text* package includes a set of *input filters* for reading documents into a canonical form suitable for the indexing program *lqaddfile*, to process; a set of search programs; and programs that take search results and deliver the corresponding text. There are also wrappers so that users don't have to remember all the individual programs.

## 5. The lq-text Implementation

This section describes the implementation of *lq-text*.

### 5.1 Information Stored

Information is stored about the documents indexed, and about each distinct *word-form* (for example, information about occurrences of *sock* and *socks* is all stored under *sock*; see the discussion of stemming under *Indexing* below).

Three distinct kinds of index are used. The first uses *ndbm*, a dynamic hashing package [Knut81]. The second type of index is a fixed record size random-access file, read using *lseek*(2) and a cache. Finally, the third index contains linked lists of variable-sized blocks; the location of the head of each chain is stored in one of the other indexes, as detailed below. This third index is used to store the variable-sized data constituting the list of matches for each word-form. The combination of the three index schemes allows fast access and helps to minimise space overhead.

*Dynamic Hashing Packages:* These have a number of desirable properties, including *minimal disk access*, since usually only two disk accesses are needed to retrieve any item, and *automatic expansion*, since the hash function simply gets wider as the database grows, allowing updates at any time. In addition, the technology is widely available, since many Unix systems include *ndbm*, and there are also much faster *ndbm*-clone implementations available.

Two *ndbm*-clone packages are distributed with *lq-text*. One of these, *sdbm*, has been widely distributed and is very portable [Yigi89]. The other, *db*, is part of the 4.4BSD work at Berkeley, and is described in [Selt91]. A general discussion on implementing such packages was distributed in [Tore87]. See also *ndbm*(1).

Two *ndbm* databases are used: one maps file names into File Identifiers (FID), and the other maps natural-language words into Word IDentifiers (WID).

*Fixed Record Indexes:* A word identifier (WID) as obtained from the *ndbm* word map is taken as a record number into a fixed size record file, `widindex`. The information found in this record is described later; for now, it suffices that one of the fields is a pointer into the linked lists of blocks in the final file, `data`.

*Linked Lists of Blocks:* Each word can occur many times in any number of files. Hence, a variable-sized data structure is needed. A linked list of 64-byte disk blocks is used. However, where adjacent blocks in the data correspond to the same thread, they are coalesced, rather as in the Berkeley Fast File System [McKu83]. Although an *lseek*(2) and a *read*(2) may be required for each 64-byte block, the Unix block buffer cache makes this arrangement relatively efficient. A Least Recently Used (LRU) cache holds a number of 16 Kbyte segments of these 64-byte blocks, giving a significant speedup, especially over NFS, where *write*(2) is synchronous.

### 5.2 Per-file Information

Each document is assigned a File Identifier when it is indexed. A File Identifier, or *FID*, is simply a number.

Conceptually, this number is then used to store and retrieve the information shown in Table 1.

| Field | Example |
|---|---|
| location | `/home/hermatech/cabochon/letters` |
| name | `tammuz.ar(june-14.Z)` |
| title | *Star Eyes watched the jellycrusts peel* |
| size | 1352 bytes |
| last indexed | 12th Dec. 1991 |

Table 1: Per-file Information

In the table, `/home/hermatech/cabochon/letters` is an absolute path to a directory; if a relative or unrooted path is given, *lq-text* will search along a document path specified in the database configuration file, or by the `DOCPATH` environment variable. The search is performed on retrieval, so that the prefix to the path needn't be stored in the database. The document name is here given separately to emphasise that once a document has been indexed, it can be moved, or even compressed and then stored as a member of an archive, as here: *lq-text* will automatically extract `june-14.Z` from the *ar*-format archive `tammuz.ar` and run *uncompress* on the result to retrieve the desired document.

The size and date fields shown in the table are used to prevent duplicate indexes of the same file, so that one can run *lqaddfile* repeatedly on all the files in a directory and add only the new files to the index (no attempt is made to detect duplicated files with differing names). In addition, the file size allows *lq-text* to make some optimisations to reduce the size of the index, such as reserving numerically smaller file identifiers for large files. The reasoning is that larger file are likely to contain more, and more varied, words. Since the file identifier has to be stored along with each occurrence of each word in the index, and since (as will be shown) *lq-text* works more efficiently with smaller numbers, this can be a significant improvement.

In fact, all of the above information except the document title is stored directly in the *ndbm* index. Using a multi-way trie would probably save space for the file locations, but the file index is rarely larger than 10% of the total index size. The `DOCPATH` configuration parameter and Unix environment variable supported by *lq-text* allow the file names to be stored as relative paths, which can save almost as much space as a trie would, and at the same time allows the user to move entire hierarchies of files after an index has been created. The document title is kept in a separate text file, since users are likely to want to update these independently of the main text.

## 5.3 Per-word Information

For each unique word (that is, for each *lemma*), *lq-text* stores the information shown in Table 2.

| Field | Bytes |
|---|---|
| Word length | 1 |
| the Word itself | $(wordlength)$ |
| Overflow Offset | 4 |
| No. of Occurrences | 4 |
| Flags | 1 |
| (Total) | $10 + word\text{–}length$ |

Table 2: Per-word Information

The word length and the word are stored only if the WordList feature is not disabled in the database configuration file. The Overflow Offset is the position in the data overflow file (described below) at which the data for this word begins. The remainder of the word index record is used to store the first few matches for this word. In many cases, it turns out that all of the matches fit in the word index, and the Overflow Offset is set to zero.

The flags are intended to be a bitwise 'and' of all the flags in the per-word entries described below, but this is not currently implemented, and the space is not reserved. When implemented, this will let *lq-text* detect whether all occurrences of a word have a given property, such as starting with a capital letter. This information can then be used to recreate the correct word form in generating reports, and when searching the vocabulary.

## 5.4 Per-occurrence Information

For each occurrence of each lemma (that is, for each *match*), the information shown in Table 3 is stored.

| Field | Size in Bytes |
|---|---|
| FID (file identifier) | 4 |
| Block Number | 4 |
| Word In Block | 1 |
| Flags | 1 |
| Separation | 1 |
| (Total) | 11 |

Table 3: Per-occurrence Information

Since, on average, English words are approximately four characters long, and allowing one character for a space between words, one might expect the index to be approximately double the size of the data from the per-match information alone. Many commercial text retrieval systems do as badly, or worse [Litt94].

In fact, the information is stored compressed. The matches are stored sorted first by FID and then by block within file, then by word in block. As a result, for all but the first match it is only necessary to store the difference from the previous FID. Furthermore, matches for a single FID are grouped together, so that it isn't necessary to repeat the FID each time. The information ends up being stored as follows:

> ΔFID (difference from previous File IDentifier)
> Number of following matches using this FID
>> Block In File
>> Word In Block
>> Flags (if present)
>> Separation (if not 1)
>>
>> ΔBlock In File
>> Word In Block
>> Flags (if different from previous)
>> Separation (if not 1)

Storing lower-valued numbers makes the use of a variable-byte representation particularly attractive. The representation used in *lq-text* is that the top bit is set on all but the last byte in a sequence representing a number. Another common representation is to mark the first byte with the number of bytes in the number, rather like the Class field of an Internet address, but this means that fewer bits are stored in the first byte, so that there are many more two-byte numbers.

The flags are stored only if they are different from the previous match's flags. This is indicated on the disk by setting the least significant bit in the Word In Block value; this bit is automatically bit-shifted away on retrieval. Further, the separation is only stored if the flags indicate that the value is greater than one (whether or not the flags were stored explicitly). The combination of delta-coding, variable-byte representation and optional fields reduces the size of the average match stored on disk from eleven to approximately three bytes. For large databases, the index size is about half the size of the data. Furthermore, since *lq-text* has enough information stored that it can match phrases accurately without looking at the original documents, it is reasonable to compress and archive (in the sense of *ar*(1)) the original files. When presenting results to the user, *lq-text* will fetch the files from such archives automatically.

## 6. Programs and Algorithms

This section describes the programs used first to create and update *lq-text* indexes, and then to retrieve data, and outlines the main algorithms used.

### 6.1 Indexing

*Storing the Per-File information:* Each input file is read a word at a time. A word is defined to be a `WordStart` character followed by zero or more `WithinWord` characters. In addition, a character of type `OnlyWithinWord` may occur any number of times within a word, but if it occurs two or more times in a row, the first occurrence is taken as the last character in the word. This allows for the apostrophe in possessives (*the James'*) as well as in such words as *can't*. Words shorter than `MinWordLength` are rejected, and the next word read successfully will have the `WPF_LASTHADLETTERS` flag set. Words longer than `MaxWordLength` are truncated to that length. In addition, if any punctuation was skipped in looking for the start of the word, the `WPF_UPPERCASE` flag is set on this word.

The word is then looked up to see if it was in the common words file. If so, it is rejected and the next successfully read word will have the `WPF_LASTWASCOMMON` flag set. In addition, whenever the start of this word is more than one character beyond the start of the previous successfully read word—as in the case that a common word, or extra space or punctuation, was skipped—the next successfully read word will have the `WPF_HASSTUFFBEFORE` flag set, and the distance will be stored in the Separation byte shown in Table 3.

Common word look-up uses linear search in one of two sorted lists, depending on the first letter of the word. Using two lists doubled the speed of the code with very little change, but if more than 50 or so words are used, the common word search becomes a significant overhead; a future version of *lq-text* will address this.

After being accepted, the word is passed to the *stemmer*. Currently, the default compiled-in stemmer attempts to detect possessive forms, and to reduce plurals to the singular. For example, *feet* is stored as a match for *foot*, rather than in its own entry; there will be no entry for *feet*. When the stemmer decides a word was possessive, it removes the trailing apostrophe or *'s*, and sets the `WPF_POSSESSIVE` flag. When a plural is reduced to its singular, the `WPF_WASPLURAL` flag is set.

Other stemming strategies are possible. The most widespread is Porter's Algorithm, and this is discussed with examples in the references [Salt88], [Frak92]. Such an alternate stemmer can be specified in the configuration file. Porter's algorithm will conflate more terms, so that there are many fewer separate index entries. Since, however, the algorithm does not attempt etymological analysis, these conflations are often surprising.

The converted words are then mapped to a WID as described above, and stored in an in-memory symbol table. Whenever the symbol table is full, and also at the end of an index run, the pending entries are added to the index, appending to the linked-list chains in the `data` file. The ability to add to the data for an individual word at any time means that an *lq-text* index can be added to at any time.

*Compression:* As mentioned above, numbers written to the index are stored in a variable-byte representation. In addition, the numbers stored are the difference between the current and previous values in a sequence.

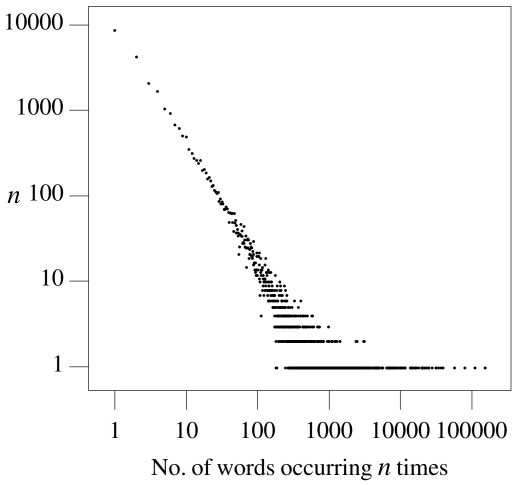



Figure 1: Vocabulary Distribution

Figure 1 shows the vocabulary distribution graph for the combined index to the King James Bible, the New International Version, and the complete works of Shakespeare, a total of some 15 megabytes of text (the NIV is in modern English; the other two are in 16th-century English). It can be seen from the graph that a few words account for almost all of the data, and almost all words occur fewer than ten times. The frequency $f$ of the $n$th most frequent word is usually given by *Zipf's Law*:

$$f = k\,(n + m)^s \quad . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \quad [1]$$

where $k$, $m$ and $s$ are nearly constant for a given collection of documents [Zipf49], [Mand53]. As a result, the optimisation whereby *lq-text* packs the first half dozen or so matches into the end of the fixed-size record for that word, filling the space reserved for storing long words, is a significant saving. On the other hand, the delta encoding gives spectacular savings for those few very frequent words: 'the' occurs over 50,000 times in the SunOS manual pages, for example; the delta coding and the compressed numbers reduce the storage requirements from 11 to just over three bytes, a saving of almost 400,000 bytes in the index. Although Zipf's Law is widely quoted in the literature, the author is not aware of any text retrieval packages that are described as optimising for it in this way.

## 6.2 Retrieval

This section describes the various programs used to retrieve information from the index, and some of the algorithms and data structures used.

*Simple Information:* The *lqfile* program can list information about files in the index. More usefully, *lqword* can list information about words. For example, the script

```
lqword -A | awk '{print $6}' | sort -n
```

was sufficient to generate data for a *grap*(1) plot of word frequencies shown above (see Figure 1).

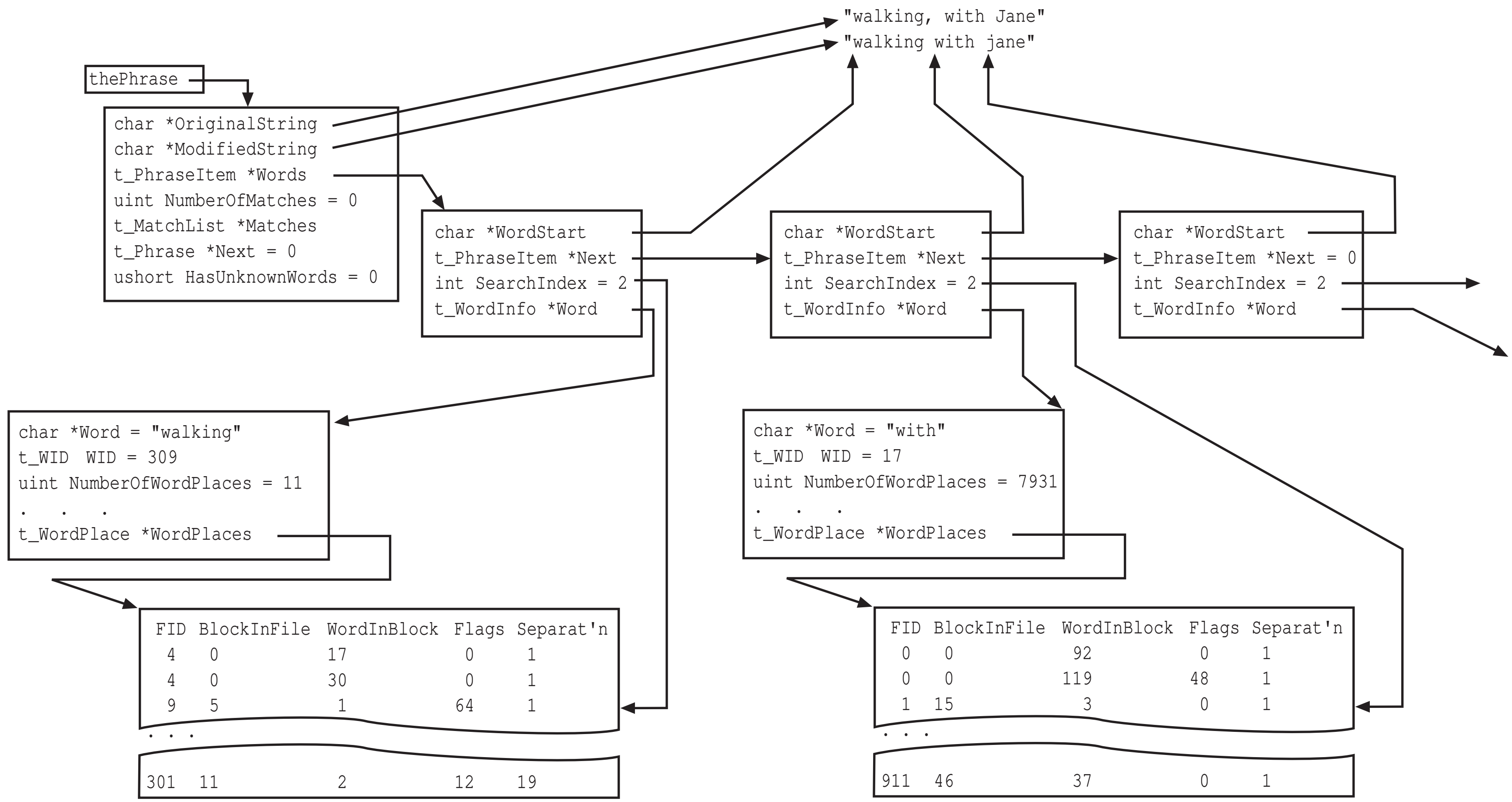
It is also possible to pipe the output of *lqword* into the retrieval programs discussed below in order to see every occurrence of a given word.

*Processing a Query:* A query is currently a string containing a single phrase. The database is searched for all occurrences of that phrase. In order to process a query, a client program first calls `LQT_StringToPhrase()` to parse the query. This routine uses the same mechanisms to read words from the string as the indexer (*lqaddfile*), and the same stemming is performed.

The client then calls `LQT_MakeMatches()` , which uses the data structure to return a set of matches. A better than linear time algorithm, no worse than $O$(total number of matches) is used; this is outlined in Algorithm 1, and the data structure used is illustrated in Figure 2.

This appears $O(m^w)$ at first sight, if there are $w$ words in the phrase each with $w$ matches, but the search at [3] resumes at the 'current' pointer, the high-water mark reached for the previous word at [1]. As a result, each match is inspected no more than once. For a phrase of three or more words, the matches for the last word of the phrase are inspected only if there is at least one two-word prefix of the phrase. As a consequence, the algorithm performs in better than linear time with respect to the total number of matches of all of the words in the phrase. In addition, although the data structure in the figure is shown with all of the information for each word already read from disk, the algorithm is actually somewhat lazy, and fetches only on demand.

*Sample Program:* The *lqphrase* program takes its arguments one at a time, treats each as a query, and processes it in turn as described. The main part of the source for *lqphrase* is shown in the Appendix, to illustrate this part of the C API.




**Notes:** The `WordInfo` for each word in the phrase also has its own `WordPlace`, with `BlockInFile`, `WordInBlock`, `Flags` and `Separation` set as if the query string was itself an indexed file. These values are interpreted more or less strictly depending on the current Phrase Match Precision Level.

The arrays of `t_WordPlace` structures shown here are usually very large. The `SearchIndex` pointers (actually integers)are indexes into the last row of each array that has been inspected for a match. This index allows the algorithm to resume each search where it left off, so that the complexity is no worse than linear.

**Figure 2 – Data Structure used in Matching a Phrase (simplified)**

```
[1] For each word in the first phrase {
  [2] for each subsequent word in the phrase {
    [3] is there a word that continues the match
    rightward? {
      Starting at current pointer, scan forward for a
      match in the same file;
      Continue, looking for a match in the same
      block, or in an adjacent block;
      Check that the flags are compatible
      If no match found, go back to [1]
      Else, if we're looking at the last word {
        Accept the match
      } else {
        continue at [2]
      }
    }
  }
}
```

Algorithm 1: Phrase matching

### 6.3 Ranking of Results

A separate program, *lqrank*, combines sets of results and sorts them. Currently, only boolean 'and' and 'or' are available. Quorum ranking, where documents matching all of the phrases given are ranked above those that match all but *n* phrases, will be in the next release.

Statistical ranking—for example, where documents containing the given phrases many times rank more highly than those containing the phrases only once—is also planned work. See [Salt88]. Statistical ranking and document similarity, where a whole document is taken as a query, should also take document length into account, however; this is an active research area [Harm93].

The initial implementation of *lqrank* used *sed* and *fgrep*. This was improved by Tom Christiansen to use *perl*, and then coped with larger results sets (*fgrep* has a limit), but was slower.

The current version is written in C. For ease of use, *lqrank* can take phrases directly, as well as lists of matches and files containing lists of matches.

The algorithms in *lqrank* are not unusual; the interested reader is referred to the actual code.

### 6.4 Presentation of Search Results

The retrieval programs discussed so far—*lqword*, *lqphrase*, and *lqrank*—return an ASCII match format as follows:

[1] Number of words in query phrase
[2] Block within file
[3] Word number within block
[4] File Number
[5] File Name and Location

(Items [4] and [5] are optional, but at least one must be present.)

In itself, this is often useful information, but the programs described below—*lqkwic* and *lqshow*—can return the corresponding parts of the actual documents.

*Key Word In Context listings:* The *lqkwic* program fetches the data from the original documents referred to by the given matches. It presents the results in the format used by a permuted, or 'key word in context', index. *lqkwic* has a built-in *little language* [Bent88] to control the formatting of the results, and uses lazy evaluation to maximise its efficiency. This program can be used to generate SGML concordances, for example, or even simply to expand the file name in each match into an absolute path. See the Appendix for sample *lqkwic* output.

*Converting to line numbers:* For use with editors and pagers such as *ex*, *nvi*, *less* and *more*, the *lqbyteline* program converts matches to (file, line-number) pairs. Unfortunately, although all of these editors and pagers understand a *+n file* option to open the named file at the given line number, the options applies only to the first file opened subsequent files are opened at the first line (*nvi* goes so far as to convert subsequent *+n* options to *–n* options, but then tries to edit a file of that name).

*Text In Place:* *lqshow* is a *curses*-based program to show part of a file, with the matched text highlighted.

### 6.5 Combined Interfaces

Two interfaces to *lq-text* are currently included in the distribution. These allow a combination of searching and browsing in a single interactive session.

*lqtext:* This is a simple curses-based front end.

*lq:* This is a shell script that combines all of the *lq-text* programs with a simple command language. It requires the System V shell (with shell functions and the ability to cope with an 815-line shell script!).

The original purpose of *lq* was to demonstrate the use of the various *lq-text* programs, but *lq* is widely used in its own right. A brief example of using *lq* to generate a keyword in context listing from the netnews `news.answers` newsgroup is shown in the Appendix.

## 7. Performance

This section describes some work that was done to measure and improve the performance of *lq-text*, and then gives some actual measurements and timing comparisons with other systems.

## 7.1 Profiling

*lq-text* originally took over 8 hours to index the King James Bible on a 25 MHz under 386/ix. Extensive profiling, and careful tuning of cache algorithms, improved performance dramatically: the time to index the Bible has been reduced to under five minutes.

*Function Calls:* Although most C compilers have a fairly low function call overhead these days, it's still not trivial. Functions called for every character of the input were folded inline, and those called for every word were made into macros in many cases. Understanding *why* each function was called the number of time it was proved a big help both in speeding up the programs and in debugging them.

At one point, *lqaddfile* was spending over 40% of its time in `stat`(2). It turned out that it was opening and closing an *ndbm* database for every word of input, which was suboptimal.

Now, most of the routines spend more than half of the time doing I/O, and no single function accounts for more than 10% of the total execution time.

## 7.2 Timings

The performance of *lq-text* is compared with SunOS 4.1 *grep*, GNU *grep* (*ggrep*), and Udi Manber's *agrep*. The *agrep* timings reflect only the simplest use of that program, since the goal was to generate comparable results. For *lq-text*, the time to build the index is also reported. Recall that the index only needs to be built once.

The following searches were timed; since the results for the various forms of *grep* were always very similar for any given set of files, the *grep* timings are only given once for each collection of data.

*0. Not There:* something not in the index at all, a nonsense word; the time was always 0·0 for *lq-text*, as reported by *time*(1), irrespective of the size of the database. This timing is therefore omitted from the table.

*1. Not Found:* a phrase made up of words that do occur, but not in the order given (if 'gleeful' and 'boy' each occur, but 'gleeful boy' does not, 'gleeful boy' would be such a search).

*2. Common:* a phrase that occurs infrequently, but includes a relatively frequent word.

*3. Unusual:* a word or phrase that occurs infrequently

The following corpora were used:

*Man Pages:* The on-line manuals from SunOS 4.1, a total of twelve megabytes.

*Bibles:* The King James and New International bibles, and the Moby Complete Works of Shakespeare, a total of over 15 megabytes.

*FAQ:* The netnews `news.answers` newsgroup of approximately 1150 articles, totalling over 30 megabytes.

*Timing Environment:* A SPARCstation 10/30 (1 processor, 30 MHz, no cache) with 64 MBytes of memory was used for the timings. The system was not equipped with Wide SCSI disks. The timings are given in real time, using *time*(1), as this is the most important in practice. Each timing was performed several times, and an average taken of all but the first. This favours the *grep* algorithms somewhat, since it reduces the impact of the I/O that they do.

| **Index** | **egrep** | **ggrep** | **agrep** | **lq-text** | | |
|---|---|---|---|---|---|---|
| | | | | **[1]** | **[2]** | **[3]** |
| man | 41·5 | 37·1 | 45·9 | 1·8 | 0·0 | 0·3 |
| Bible | 60·2 | 55·5 | 67·7 | 1·8 | 1·2 | 0·1 |
| FAQs | 181·9 | 179·2 | 115·7 | 0·3 | 1·0 | 0·2 |

Table 4: Timings

| **Index** | **Size** | | **Creation Time** | | |
|---|---|---|---|---|---|
| | **Data** | **Index** | **real** | **user** | **sys** |
| man | 12M | 6·5M | 267·4 | 124·9 | 64·5 |
| Bible | 15M | 6·6M | 351·3 | 136·6 | 51·0 |
| FAQs | 41M | 20·5M | 1598·8 | 851·9 | 375·8 |

Table 5: Index Statistics

The *lq-text* timings do not include the time to produce the text of the match, for example with *lqkwic*. However, running *lqkwic* added less than one second of run-time for any except the very large queries, even when the data files were accessed over NFS.

The index overhead is approximately 50% of the size of the original data. This can be controlled to some extent using stop-words; the `news.answers` database used 79 stop-words, reducing the database by about 2 Megabytes. In addition, single-letter words were not indexed, although the presence of a single-letter word was stored in the per-word flags. The other databases used no stop words, and indexed words from 1 to 20 characters in length—the differences are because the FAQ index is accessed by a number of other users on our site.

*Results:* As one would expect, *lq-text* easily out-performs the *grep* family of tools. For queries producing a lot of matches, such as 'and the' (1790 occurrences in the SunOS manual pages), the time taken to print the matches dominates the run-time of *lqphrase*.

## 8. Ongoing and Future Work

This section describes speculative, planned and ongoing work.

## 8.1 The C API

The *lq-text* libraries (`liblqerror`, `liblqutil` and `liblqtext` itself) each provide a clear set of functions forming an Application Programmer's Interface (API). The process of tidying up the API is under way:

*Documentation:* The API is currently documented only by function prototypes in header files and by examples. Clearly this needs to change.

*Completeness:* The API isn't complete yet. For example, in `liblqerror` there's an `Eopen()` function which works like Unix *open*(2), except that it provides error messages and can be made to exit on error. However, there is no `Eclose()` function yet.

*Consistency:* The structure of the API needs to be clear enough that one would be able to guess which library contains any given function; this is largely but not completely true now. Almost all functions have a prefix, such as `LQU_` in `LQT_ObtainWriteAccess()`, for example, for functions from `liblqtext`. A very few functions don't do this, and a few others are actually defined in client programs rather than in the library.

*Configuration and Testing:* Configuration is currently a case of editing a `Makefile` and a `C` header file, but several people have asked for something like the GNU auto-configuration package.

An *ad hoc* test suite is included with the *lq-text* distribution, but this needs to be made more formal, and to be run automatically when the software is built.

## 8.2 A User Interface

*lq-text* is primarily a text retrieval engine suitable for integration into other systems. However, experimental user interfaces have proved popular, and it is certainly expected that better interfaces will be provided in the future.

*X11 interface:* An X11 client based on the Fresco toolkit is planned, building on the work of Marc Chignel [Golo93], Ed Fox *et al.* [Fox93] and others. However, this work is awaiting the distribution of the Fresco toolkit with X11R6.

## 8.3 Functionality

In addition to the user interface, there are some specific features that are wanted:

*Approximate matching:* currently, *lq-text* can perform *egrep*-style matches against the vocabulary in the index; it would be interesting to extend this to *agrep*-style approximate patterns, and to integrate it into the main query language, so that "`core /^dump.*~/`" might match 'core dumped', using approximate matching only for the second word in the phrase.

*Complex queries:* It it desired to support queries that are themselves complex, or that refer to the structure of documents stored marked up in SGML format [Stan88], perhaps building on the work of Forbes Burkowski [Burk92]. Allowing a more complex syntax in a query has to be done carefully, so that the language is both straightforward and general. Handling structured documents also entails an extended query parser. At the same time, Fuzzy Logic [Zade78] and limited recognition of anaphoristic references is proceeding. It may also be possible to perform experiments in clustering, in the manner of some of the recent work at Xerox [Cutt93].

*Performance:* Although *lq-text* is already pretty fast at both retrieval and indexing, it could certainly be made faster. Experiments with *mmap*(2) and with alternate cache algorithms are ongoing.

*Run-time configuration:* New parameters will include user-defined stemming (perhaps using stemming algorithms described by W. Frakes in [Frak92]), and allowing a partial (document-vector) index.

## 9. Acknowledgements

Richard Parry helped with the original design, and walked through code. Mark Brader has contributed `C` advice. Mark Bremner contributed the code to uncompress documents on the fly. Mark Moraes helped with porting and installation issues. Henry Spencer, Geoff Collyer and Ian Darwin helped with the licence and copyright. Many, many people have sent feedback and bug fixes. Ozan Yigit, Bob Gray and Kate Hamilton helped with the paper.

## 10. Conclusions

The *lq-text* package is freely available, and hence clearly meets the criterion of low cost given at the start of the Introduction above. The largest database indexed by the author at the time of writing (March 1993) occupies about 100 Megabytes, and it remains to be determined how suitable *lq-text* is at indexing and searching larger bodies of text, which was the second main goal given in the Introduction. The package does provide fast retrieval, and meets all of the Design Goals given above except for the abilities to unindex and update documents. These last two features are expected in the summer of 1994.

The source for *lq-text* is available for anonymous ftp from `ftp.cs.toronto.edu` in /pub. Updates are announced on a mailing list, `lq-text-request@sq.com`.

## 11. References

[Bent88] Bentley, Jon, "Little Languages," in *More Programming Pearls*, Addison-Wesley, 1988. A clearly-written rationale for the use of little (or embedded) languages. This column first appeared in Comm. ACM. in August 1986.

[Bray89] Bray, Tim, *Lessons of the New Oxford English Dictionary Project,* pp. 137–199, Usenix, Winter, 1989.

[Burk92] Burkowski, Forbes J., "An algebra for hierarchically organized text-dominated databases," *Information Processing & Management*, **28**, p. 333, 3, 1992.

[Clev66] Cleverdon, C. W., Mills, J., and Keen, E.M., *Factors Determining the Performance of Indexing Systems, Volume 1 – Design,* Aslib Cranfield Research Project, Cranfield, 1966.

[Cutt93] Cutting, Douglas R., Karger, David R., and Pedersen, Jan O., "Constant Interaction-Time Scatter/ Gather Browsing of Very Large Document Collections," in *Proc. 16th ACM SIGIR*, pp. 126–131, 1993. One of a number of papers reporting work at Xerox Parc on information retrieval

[Falo85] Faloutsos, Christos, "Access Methods for Text," *Computing Surveys*, **17**, 1, pp. 49–74, March 1985. Compares text retrieval methods for office systems

[Falo87a] Faloutsos, Christos and Christodoulakis, Stavros, "Optimal Signature Extraction and Information Loss," *ACM Trans. on Database Systems*, **12**, 3, pp. 395–428, Sept. 1987.

[Falo87b] Faloutsos, Christos and Christodoulakis, Stavros, "Description and Performance Analysis of Signature File Methods," *ACM Trans. on Office Systems*, **5**, 3, July 1987. A good overview of signatures.

[Fawc89] Fawcett, Heather, *PAT User's Guide,* Open Text, 1989.

[Fox93] Fox, Edward A., France, Robert K., Sahle, Eskinder, Daoud, Amjad, and Cutter, Ben, "Development of a Modern OPAC: From REVTOLC to MARIAN," TR 93-06, Virginia Polytechnic Institute and State University, 1993. A client-server Online Public Access Catalogue for a library, using the NeXTStep GUI.

[Frak92] Frakes, William B. and Baeza-Yates, Ricardo, *Information Retrieval: Data Structures and Algorithms,* Prentice-Hall, 1992. An excellent introduction to the issues in implementing information retrieval systems. Examples in `C` for Unix, available by ftp from `ftp.vt.edu` in `pub/reuse/ir-code`.

[Golo93] Golovchinsky, G. and Chignell, M.H., "Queries-R-Links: Graphical Markup for Text Navigation," in *Proceedings of INTERCHI '93, Amsterdam*, pp. 454–460, ACM Press., N.Y., April 1993. Presents a conceptually simple way for users to add and subtract terms from text retrieval queries, and raises issues about the trade-offs between pre-determined hypertext links and live text retrieval queries.

[Harm93] Harman, Donna, "Overview of the First TREC Conference," *Annual ACM SIGIR Conf.*, **16**, p. 36, 1993. At the SIGIR conference in 1993, some of the TREC participants reported that they had had difficulties using similarity techniques on long documents.

[Knut81] Knuth, Donald, *The Art of Computer Programming, Vol III: Sorting and Searching,* Addison-Wesley, 1981.

[Lesk78] Lesk, M. E., "Some Applications of Inverted Indexes on the Unix System," in *V7 Unix Programmers' Manual, Vol 2A*, Bell Laboratories, 1978.

[Litt94] Littman, Dan, "AppleSearch 1.0," *Macworld*, May 1994. A review of Apple's 'easy to administer, easy to use' text retrieval software. Mentions that 'the indexing process requires more than double the disk space of the original documents'.

[Mand53] Mandelbrot, Benoit, "An informational theory of the statistical structure of language," in *Communication Theory*, ed. Willis Jackson, pp. 486–502, Butterworths, 1953.

[McKu83] McKusick, Marshall Kirk, Joy, William N., Leffler, Samuel J., and Fabry, Robert S., "A Fast File System for Unix," CSRG Technical Report 83-147, 1983.

[Mead92] Meadow, Charles T., *Text Information Retrieval Systems,* Academic Press, Toronto, 1992. Gives clear descriptions of full-text retrieval data structures and algorithms, although with a bias towards indexing only abstracts of books or of library catalogue entries.

[Orac92] Oracle Corporation,, *SQL*TextRetrieval Version 2 Technical Overview*, 1992.

[Salt88] Salton, Gerald, *Automatic Text Processing,* Addison-Wesley, 1988.

[Selt91] Seltzer, Margo and Yigit, Ozan, *A New Hashing Package for Unix,* Usenix '91, Dallas, TX, 1991.

[Stan88] (ISO), International Organization for Standardization, "Information Processing—Text and office systems—Standard Generalized Markup Language (SGML)," ISO8879, 1988.

[Tore87] Torek, Chris, *Re: dbm.a and ndbm.a archives,* netnews `comp.unix` newsgroup, 1987.

[Tsuc91] Tsuchiya, Paul F., *A Search Algorithm for Table Entries with Non-contiguous Wildcarding,* Bellcore, 1991. Unpublished(?) description of Cecilia, a package using in-memory Patricia trees with efficient update and deletion.

[Yigi89] Yigit, Ozan, *How to roll your own dbm/ndbm,* Unpublished Manuscript, 1989.

[Zade78] Zadeh, L. A., "PRIF—a meaning representation language for natural languages," *Int. J. Man-Machine Studies*, **10**, pp. 395–460, 1978. One of many of L. A. Zadeh's papers arguing for modeling the 'pervasive imprecision of natural languages' (p.396).

[Zimm91] Zimmerman, Mark, *Zbrowsr implementation*, 1991. Article in *para* mailing list, unpublished.

[Zipf49] Zipf, Georke K., *Human Behavior and the Principle of Least Effort,* Addison-Wesley, Cambridge, MA., USA, 1949.

## 12. Appendix 1—Source for *lqphrase*

```
PRIVATE void
MatchOnePhrase(Phrase)
    char *Phrase;
{
    t_Phrase *P;

    if (!Phrase || !*Phrase) {
          /* ignore an empty phrase */
          return;
    }

    if ((P = LQT_StringToPhrase(Phrase)) == (t_Phrase *) 0) {
          /* not empty, but contained no plausible words */
          return;
    }

    /*  PrintAndAcceptOneMatch() is a function that prints
     *  a single match.  It is called for each match as soon as it is
     *  read from the disk.  This means that results start appearing
     *  immediately, a huge benefit in a pipeline.
     */
    if (LQT_MakeMatchesWhere(P, PrintAndAcceptOneMatch) <= 0L) {
          return;
    }
}
```

## 13. Appendix 2—Sample *lq* session

This listing shows part of a session using *lq*, a shell-script that uses *lq-text*. The *faq* command invokes *lq* after setting up environment variables and options to use the `news.answers` database.

```
: sqrex!lee; faq
Using database in /usr/spool/news/faq.db...
| Type words or phrases to find, one per line, followed by a blank line.
| Use control-D to quit.  Type ? for more information.
> text retrieval
>
Computer Science Technical Report Archive Sites == news/answers/17480
  1:v.edu> comments: research reports on text retrieval and OCR orgcode: ISRI
Interleaf FAQ -- Frequently Asked Questions for comp.text.interleaf == news/answers/17607
  2:g with hypertext navigation and full-text retrieval. 1.2. What platform
alt.cd-rom FAQ == news/answers/17643
  3:and Mac. 56. Where can I find a full text retrieval engine for a CDROM I a
  4:======== 56. Where can I find a full text retrieval engine for a CDROM I a
  5: I am making? Here is a list of Full-Text Retrieval Engines from the CD-PU
OPEN LOOK GUI FAQ 02/04: Sun OpenWindows DeskSet Questions == news/answers/18078
  6:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
OPEN LOOK GUI FAQ 04/04: List of programs with an OPEN LOOK UI == news/answers/18079
  7: Description: Networked, distributed text-retrieval system. OLIT-based fr
  8:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
[comp.text.tex] Metafont: All fonts available in .mf format == news/answers/18080
  9:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
OPEN LOOK GUI FAQ 03/04: the XView Toolkit == news/answers/18082
 10:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
Catalog of free database systems == news/answers/18236
 11:------------------ name: Liam Quin's text retrieval package (lq-text) vers
 12: are bugs. description: lq-text is a text retrieval package. That means
| Type words or phrases to find, one per line, followed by a blank line.
| Use control-D to quit.  Type ? for more information.
> :less 9  (brings up the 9th match in a pager)
> :help
| Commands
|
| :help
```

```
     Shows you this message.
  :view [n,n1-n2,n...]
     Use :view to see the text surrounding matches.
     The number n is from the left-hand edge of the index;
     :set maxhits explains ranges in more detail.
  :page [n,n1-n2,n...]
     Uses less -c (which you can set
     in the $PAGER Unix environment variable) to show the files matched.
  :less [n,n1-n2,n...]
     The same as :page except that it starts on the line number
     containing the match, firing up the pager separately on
     each file.
  :index [n,n1-n2,n...]
     This shows the indexfor the phrases you've typed.
  :files [n,n1-n2,n...]
     This simply lists all of the files that were matched,
     in ranked order.
  :prefix prefix-string
     Shows all of the words in the database that begin with that prefix,
     together with the number of times they occur.
  :grep egrep-pattern>
     Shows all of the words in the database that match that egrep pattern,
     together with the number of times they occur (not the number of files
     in which they appear, though).
  :set option ...
     Type :set to see more information about setting options.
  :shell [command ...]
     Use /home/sqrex/lee/bin/sun4os4/msh  to run commands.

  Commands that take a list of matches (n,n1-n2,n...)
  only work when you have generated a list of matches.  If you don't give
  any arguments, you get the whole list.

> :set
  :set match [precise|heuristic|rough]
       to set the phrase matching level,
       currently heuristic matching (the default)
  :set index
       to see a Key Word In Context (KWIC) index of the matches
  :set text
       to browse the actual text that was matched
  :set database directory
       to use the database in the named directory.
       Current value: /usr/spool/news/faq.db
  :set rank [all|most|any]
       all presents only documents containing all the phrases you typed;
       most shows those first, and then documents with all but one of
       the phrases, and so on.
       any doesn't bother sorting the matches; this is the default,
       because it's the fastest.
       Currently: find documents containing all of the phrases
  :set tags string
       determine whether SGML tags are shown or hidden in the KWIK index
  :set tagchar string
       set the character used as a replacement for hidden SGML tags
  :set maxhits n|all
       show only the first n matches (currently 200)
  :set prompt string
       set the prompt for typing in phrases to string
> ^C
Interrupted, bye.
: sqrex!lee;
```