An Object-Oriented Architecture for Text Retrieval

Doug Cutting, Jan Pedersen, and
Per-Kristian Halvorsen

Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, California

Abstract

For almost all aspects of information access systems it is still the case that their optimal composition and functionality is hotly debated. Moreover, different application scenarios put different demands on individual components. It is therefore of the essence to be able to quickly build systems that permit exploration of different designs and implementation strategies. This paper presents a software implementation architecture for text retrieval systems that facilitates (a) functional modularization (b) mix-and-match combination of module implementations and (c) definition of inter-module protocols. We show how an object-oriented approach easily accommodates this type of architecture. The design principles are exemplified by code examples in Common Lisp. Taken together these code examples constitute an operational retrieval system. The design principles and protocols implemented have also been instantiated in a large scale retrieval prototype in our research laboratory.

1 Introduction

It is good design practice in building any large software artifact, such as a text retrieval system, to decompose it into modules that reflect identifiable pieces of functionality. This is especially the case if the modules are to be reused or combined in a variety of different ways to form new systems. In particular, we argue that text retrieval systems benefit from this design strategy and that object-oriented programming is the appropriate method for abstracting the observed functionality. To illustrate this point we propose an object-oriented text retrieval architecture that captures the variability in a wide variety of text retrieval systems.

Modularity leads to robustness and flexibility through the careful definition of protocols which serve as the sole interconnections between modules. Since modules need only be plug-compatible, they may be replaced when appropriate without disturbing the remainder of the system. This also implies that system variability may be expressed by supplying multiple implementations of the same module, one of which is selected at system construction time. This can be made explicit in object-oriented programming languages, which are designed to support multiple protocol implementations.

Text retrieval systems exhibit a large number of possible time/space trade-offs. There is also variation (and controversy) in the appropriate combination of components to form complete systems, as well as research and evaluation efforts which demand flexibility in the choice of components. Thus, the ability to select subsystems from a range of options should be particularly useful in this context.

2 Desiderata

In considering the composition of an appropriate text retrieval architecture, we concentrate on the following major sources of variation:

- Corpus Retargeting

  Text retrieval is always relative to some collection of documents, or corpus, yet corpora come in different formats, reside on different storage media, etc. Indeed, some corpora may require complex computations to be performed (e.g. decompression [23]) before text is available for processing.

- Text Analysis

  Automatic indexing implies that the source text must be analyzed to some degree, if only to extract word tokens. Specific corpora may employ domain-specific jargon or sub-languages that require special handling. Text may be stemmed or normalized by morphological analysis [13, 14]. Additionally one may wish to experiment with higher level linguistic analysis, such as part-of-speech identification [4, 17] and phrase parsing [8, 4]. These analysis modules must be parametrized by source language in a multilingual environment.

- Indexing Strategies

  Indices are used to accelerate search. The degree of acceleration can often be traded for smaller storage requirements by varying the indexing granularity. Signature techniques require validation [9], others are only appropriate for static corpora [11].

- Storage Substrate

  Some systems may store their indices in private file-based data structures [12], others may employ a standard relational database accessed over a local-area network. Some may be required to store their indices on optical disks [3]. These alternatives have very different performance characteristics and demand different storage layout strategies.

- Search Methods

  A variety of search methods have been proposed for use in information retrieval [21, 20, 19]. Some are preferred in commercial environments, while others are still undergoing validation in the research community. Each search method places a different demand on the underlying database layer, although most can be accommodated through variations on the basic inverted index.

- User Interface

  A retrieval system may be presented to the user in numerous different ways, ranging from a line-oriented tty-based approach to 3D animated information visualizations [2]. Often the mode of interaction places strict requirements on the performance of the underlying search engine.

3 An Architecture

An appropriate text retrieval architecture will naturally account for the sorts of variation outlined above in a way that affords maximal flexibility with minimal overhead. We will adopt the position that there should be one module for each expected source of variation. However, the interconnections between these modules remains unresolved.

One approach that helps discover these interconnections, and thus leads towards protocol definitions, is to consider the control and data flow in two tasks central to a text retrieval system:

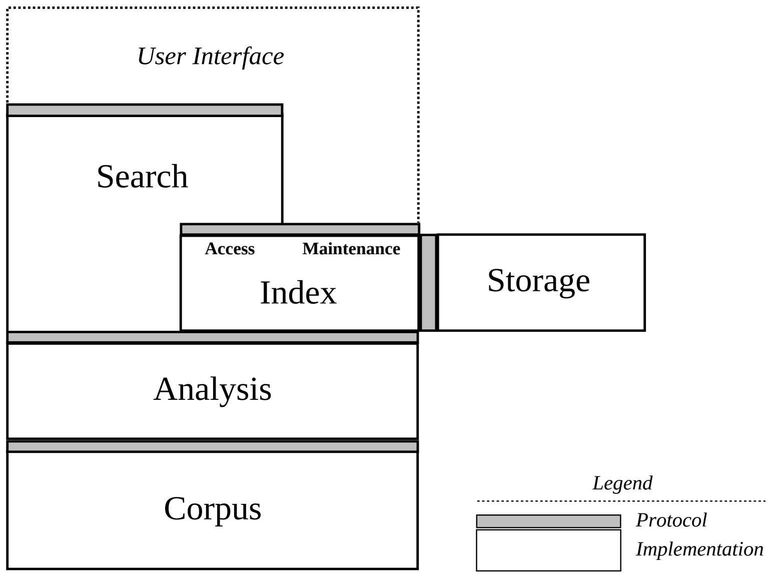

Figure 1: System Architecture

- Index construction

  Source text must be noticed and analyzed to enable accelerated search. This may be a one-time computation, in the case of a static corpus, or a continuing incremental task in the case of a dynamically changing corpus.

  Control here begins with the user interface calling the index maintenance module. This employs the analysis module which in turn invokes the corpus module to get raw text. The analysis module provides this text as index terms, and the index module stores them with calls to the storage module.

- Query resolution

  Search algorithms are the keystones of a retrieval system, and the major clients of all system components. Control in a search task begins with a search algorithm, which may invoke the analysis module to extract search terms from a query. These search terms are then used as keys for calling into the index access module which returns posting previously stored in the storage module. On occasion, the index access module may be required to refer back to the original text source to completely resolve the postings query. These results are then employed by the search algorithm as desired.

These tasks suggest a hierarchical module arrangement, as illustrated in Figure 1. Note that for most purposes the corpus is only seen through the analysis module and that the indexing module is the only client of the storage layer. The search module rests above with access both to the indexing and analysis modules. User interfaces are logically separable from the indexing and search engine, with primary access to the search module, and an occasional need, for display reasons, to peek directly through to the corpus abstraction. We will not further address the complex issue of user interface in this paper.

We have successfully employed this architecture in the construction of a high performance prototype text retrieval system, incorporating approximately 25,000 lines of Common Lisp code [6]. The use of an object-oriented programming style to implement this architecture incurs negligible run-time overhead since method calls are confined to the protocols which interconnect modules, each of which offers substantial functionality. [1] In fact, performance is improved because the flexibility

[1] In languages such as C++ even this cost is reduced since method selection is often resolved at compile-time.

```
(defclass string-corpus () ((strings :initarg :strings)))

(defmethod open-document ((corpus string-corpus) id)
  (with-slots (strings) corpus
    (make-string-input-stream (nth id strings))))
```

Figure 2: String Corpus Implementation

to supply alternative implementations allows for application-specific optimizations which would not be appropriate in a generic system. This has been demonstrated in our prototype which has proven performance in excess of industry standards over a 64 Megabyte corpus. We have also found that this architecture enables experimentation both in the ways envisaged (replacement of modules) and in ways not envisaged.

The remainder of this paper illustrates how this architecture can be implemented using object-oriented techniques, focusing on the definition of protocols through the appropriate attachment of methods to class objects. Accompanying the text is a simple, demonstration implementation of the suggested design, written in Common Lisp [22]. Since the emphasis is on architecture not algorithms, the demonstration system is unencumbered with optimized module implementations; interconnections are emphasized rather than specific functions. We note in the text where improved algorithms are appropriate. Nonetheless, the demonstration system is fully functional and runable. We conclude with the description of a sample run of the demonstration system.

The demonstration system only uses classical methods (specialized only on the first argument), but does rely heavily on multiple inheritance. Thus it should be a straightforward matter to translate it to, for example, C++, but would be somewhat more complicated to implement in Smalltalk, which does not support multiple inheritance.

Method invocations that are part of one of the protocols are highlighted with an underline. Each module is abstracted as an object with an associated protocol. It is intended that each application defines a class which is a subclass of implementations of each protocol, so that it inherits the appropriate method definitions. The programmer is then able to mix and match implementations to build an application with the desired characteristics.

4 Corpus Abstraction

A corpus is, for our purposes, a collection of documents, each with a textual component. Documents may have other components (e.g. titles, authors and dates) and super-structure (e.g. volumes and chapters) but these properties are extraneous for the purpose of textual access. Access to these non-textual properties is outside the scope of this architecture (though it is also amenable to an object-oriented treatment [10]) and is better layered on top of the retrieval subsystem.

The purpose of the corpus module is to map from abstract document identifiers (ID's) to text. The corpus protocol is intentionally kept simple to minimize the burden placed on each application, and so maximize the applicability of the retrieval system. The corpus protocol is a major interface (along with search and indexing) to client applications.

A corpus is implemented by a class. Thus a corpus is an object with some private state variables (for example, a table mapping from ID's to file names), and on which methods may be specialized. Methods must be defined for each corpus class which, given an ID, provide the text of the indicated document. All access to the text of a document for the rest of the system is through these methods. Character streams are used to represent the text of documents. These are objects which primarily just support sequential access to the characters of the text of a document.

Each corpus need not be implemented from scratch. A library of generic corpus implementations can be developed which enables one to quickly access common corpus formats and representations. Such a library might include:

```
(defclass virtual-corpus () ((sub-corpora :initarg :sub-corpora)))

(defmethod open-document ((corpus virtual-corpus) id)
  (with-slots (sub-corpora) corpus
    (let ((modulus (length sub-corpora)))
      (open-document (nth (mod id modulus) sub-corpora)
                     (floor id modulus)))))
```

Figure 3: Virtual Corpus Implementation

- a file corpus implementation

  Many corpora contain all documents in a single file. Documents consist of stretches of text within these files. Clients of this generic corpus need only specify start and end positions for each document to fully implement the above protocol.

- a directory corpus implementation

  A facility can be provided which allows the maintenance of corpora where each document is in a separate file, often all in one directory. Here the file system implements most of the details, and, in the simple case, all clients need specify is the name of a directory.

The demonstration system implements a minimal corpus protocol that represents document ID's as integers and supplies a single text access method `open-document`, which returns the text as a character stream (see Figure 2). In this implementation corpora are represented simply as a list of a strings, one for each document. ID's supply the position in the list.

Additionally one may define corpora in terms of other corpora. One might, for example define a corpus which represents the union of some number of other corpora. This can be accomplished by just renumbering ID's on access to retain uniqueness, as illustrated in Figure 3. Here the virtual corpus ID's encode both a corpus selector and a document selector in the same integer.

5 Text Analysis

Analysis converts text into objects which form the basis of search. Documents are analyzed prior to the generation of index terms, and queries are analyzed to yield search terms.

The analysis on queries and documents being indexed need not be the same, although they must produce terms in the same domain. For example, one might delay replacement of words with synonym sets until query time, as such replacement is risky and may require human intervention. In general, pre-indexing analysis should be restricted to that which can be done automatically without undue loss of potentially pertinent information.

To convert text into terms we establish a protocol which traffics in proto-terms, or *tokens*. Implementations of this protocol are typically composed of a pipeline of processing elements.

At the start of the pipeline is a *tokenizer* which extracts tokens from the text. Subsequent stages act as *filters* on these base tokens. Stop lists, stemmers, part-of-speech taggers [4, 17] and phrase spotting can be implemented as filters. Tokens emitted at the end of the pipeline are terms for indexing and search.

Token pipelines may be implemented as concatenated token streams. The example code illustrates a technique for doing this in object-oriented languages with multiple inheritance.

Figure 4 provides an implementation of some basic pipeline elements: a tokenizer, a normalizing filter and a stop-word filter. The tokenizer parses the character stream, emitting a token for each contiguous sequence of alphabetic characters. The normalizing filter just lowercases tokens, and the

```
(defclass tokenizer ()
    ((char-stream :initarg :char-stream)))
(defmethod next-token ((token-stream tokenizer))
  (with-slots (char-stream) token-stream
    (with-output-to-string (string-stream)
      (let ((in-token-p nil))
        (loop (let ((char (read-char char-stream nil)))
                (cond ((null char) ; EOF
                       (if in-token-p (return) (return-from next-token nil)))
                      ((alpha-char-p char)
                       (write-char char string-stream)
                       (setq in-token-p t))
                      (t (if in-token-p (return)))))))))

(defclass normalizer () ())
(defmethod next-token ((token-stream normalizer))
  (let ((token (call-next-method)))
    (if token (string-downcase token) nil)))

(defclass stop-list ()
    ((stop-words :initform '("an" "and" "by" "for" "of" "the" "to" "with"))))
(defmethod next-token ((token-stream stop-list))
  (with-slots (stop-words) token-stream
    (loop (let ((token (call-next-method)))
            (cond ((null token) (return nil)) ; EOF
                  ((member token stop-words :test #'string=))
                  (t (return token)))))))
```

Figure 4: Pipeline Component Definition

```
(defclass simple-analysis-pipeline (stop-list normalizer tokenizer) ())

(defclass simple-analyzer () ())

(defmethod make-token-stream ((analyzer simple-analyzer) char-stream)
  (make-instance 'simple-analysis-pipeline :char-stream char-stream))
```

Figure 5: Pipeline Definition

```
(defclass appending-token-stream () ((streams :initarg :streams)))

(defmethod next-token ((token-stream appending-token-stream))
  (with-slots (streams) token-stream
    (if streams
        (or (next-token (first streams))
            (progn (setf streams (rest streams))
                   (next-token token-stream))))))
```

Figure 6: Appending Token Stream

stop-word filter removes words which appear on a small stop list. Note that the filters access token stream elements by invoking the next method in the method inheritance.

Here tokens are just character strings. A more advanced implementation might have different types of tokens (e.g. dates, numbers, punctuation, phrases) and possibly annotate tokens with typographic information.

Figure 5 shows how these elements can be composed by defining a class which inherits from each of them. The order of pipeline processing is determined by the precedence of the classes in the inheritance. Here tokens flow right-to-left through the superclasses.

This implementation technique has the feature that all processing elements are top-level objects in the protocol, i.e. individual elements can support operations without requiring other elements to pass the message down the pipeline. This is particularly valuable in the case of the tokenizer, as clients may inquire from the tokenizer where emitted tokens occurred in the source character stream. This facilitates the construction of user interfaces which wish to show fragments from the source text in query results. [5]

We also see here the definition of `simple-analyzer`, the class which embodies the analysis protocol. This is used later in Figure 10 when an application is defined.

Figure 6 defines a token stream which appends the contents of a list of other token streams, thus exhibiting a token stream which is not a pipeline, but is rather defined in terms of other token pipelines. This technique is useful in the definition of corpora in terms of other corpora for experimental purposes. We also use it in the implementation of relevance feedback (see Figure 9).

6 Storage

The storage module provides a generic means for accessing persistent store. The purpose of this module boundary is to allow systems to store their indices in different manners: some may wish to store their indices in an existing relational database; others may require that indices be stored on optical disks. We would like to be able to accommodate these sorts of variation with little change to other parts of the system. [2]

Support of indexing is a broad goal, as there are many different strategies for indexing. We have however identified a few generic facilities which we hope satisfy this goal:

- Maps

  One would like to be able to store small records, composed of strings and integers, and then recall them given distinguished components, or *keys*. Often it is desirable to be able to enumerate such records, in key order. Such *maps* can be supported in many different ways. B-trees were designed specificly to solve this problem for ordered sets which are so large that they must be paged to secondary storage [1]. BIM-trees are similar to B-trees but were designed for use with CIV optical storage [3]. Hashing does not usually enable efficient ordered enumeration, but is a good implementation technique when this is not required [16]. Most commercial database management systems, relational and otherwise, also provide this functionality (which is usually implemented internally as B-trees).

- Blocks

  One would also like to be able to associate blocks of binary data with map entries. Thus it should be possible for components of maps to be pointers to such blocks, which may be read and written. An variety of allocation strategies for this sort of storage are covered in [15], and, again, most commercial database systems provide access to this sort of functionality.

[2] The primary goal of the storage module is to support the storage of indices, and the design of a protocol must be certain to support this. However it would be fortuitous if this module were also able to generically handle the storage requirements of applications, e.g. maintaining author and date indices. This is by nature rather ill-defined usage, and we thus do not attempt to further specify it here.

```
(defclass hash-store ()
    ((table :initform (make-hash-table :test #'equal))))

(defmethod get-mapping ((store hash-store) term)
  (with-slots (table) store
    (gethash term table)))

(defmethod (setf get-mapping) (value (store hash-store) term)
  (with-slots (table) store
    (setf (gethash term table) value)))
```

Figure 7: Hash Table Storage Implementation

Dynamic and static inverted indices can be implemented entirely with such structures [7, 12]. Terms are typically stored in a map, potentially with frequency information, while postings are stored in blocks. Signature techniques typically have similar requirements, with signatures or bit-slices being stored as blocks indexed by some map.

As the storage module is entirely hidden behind the indexing module, implementations of the indexing protocol may be tempted to use their own storage. However when reusability and variability are highly valued, the protocol should be amended rather than circumvented.

The storage implementation presented in Figure 7 provides access to an unordered mapping in the form of Common Lisp's built-in hash table facility. Keys are assumed to be strings, and values are pointers. Because this is not a persistent store (and for the sake of brevity) a block access implementation is not shown. Clients can store pointers to arbitrarily large structures directly in the map.

7 Indexing

An index is a cache used by search engines. One must thus have some notion of what search strategies are to be employed before an index can be specified. However most search methods may be implemented by treating terms as atomic entities (indeed, this observation is exploited by the analysis protocol). Indices are typically used to accelerate the enumeration of statistics about these terms, such as their frequency, and the documents which contain them. In the case of some signature techniques, lookup is not by individual terms, but rather by sets of terms.

The protocol for accessing the index consists of the procedures for reporting these statistics, as well as those for creating and maintaining the index in the face of a changing document base. (It is however, the responsibility of the application, not the index, to invoke these maintenance routines when necessary.)

A given indexing implementation will actually record only certain statistics. These may not always match the requirements of the desired search strategy. While reconciliation is not always possible, this confict may often be viewed as a time/space tradeoff. Details which are not stored in the index can be extracted directly from the text at query time. A particular application, i.e. a given corpus on given hardware, can vary the indexing detail to tune the index for reasonable response while minimizing storage.

For example, if an inverted index contains term offsets then searches involving term proximity [21] may be resolved with reference only to the index. However, if term offsets are not recorded then they may be recovered at search time by a scanning the text of documents known to contain the term of interest, albeit somewhat more slowly.

Some index optimizations are only applicable to static corpora. For example, postings for high-frequency terms may be efficiently represented as bit-vectors [11]. Indices for dynamic corpora require somewhat more complex representations and maintenance strategies [7].

```
(defun map-tokens (function token-stream)
  (loop (let ((token (next-token token-stream)))
          (if token (funcall function token) (return)))))

(defclass binary-index () ())

(defmethod index-document ((index binary-index) id)
  (map-tokens #'(lambda (token) (pushnew id (get-mapping index token)))
              (make-token-stream index (open-document index id))))

(defmethod get-binary-postings ((index binary-index) term)
  (get-mapping index term))

(defmethod get-term-frequency ((index binary-index) term)
  (length (get-binary-postings index term)))

(defmethod get-frequency-postings ((index binary-index) term)
  (mapcar #'(lambda (id)
              (let ((freq 0))
                (map-tokens #'(lambda (token)
                                (if (string= token term) (incf freq)))
                            (make-token-stream index (open-document index id)))
                (cons id freq)))
          (get-binary-postings index term)))
```

Figure 8: Binary Index Implementation

Indices which will reside on read-only media have special requirements as well. Here the implementation which creates the index is not the same as that which accesses it. This fractures the indexing module into separate creation and access modules. These presumably have much in common, but we do not have experience with this problem and will not speculate about an appropriate sub-architecture.

The sample index implementation shown in Figure 8 stores only binary posting information. For each term a list of all the documents which contain it is recorded in a map provided by the storage module. It can thus support access to binary postings directly as an access to the map. Term frequency is not stored directly, but can be computed on demand without reference to the text by measuring the length of the binary postings. Access to within-document frequencies requires a scan of the documents named in the binary postings, counting occurrences.

8 Search Algorithms

Search algorithms are the major clients for most system components. The search task encompasses query specification, which may include query text parsing and analysis, and index access for term postings. Search methods are distinguished through their specification and manipulation of the query; most are term-based, although each places a different demand on the index. For example, classical boolean search simply performs set operations on postings lists, where only the presence or absence of a term need be noted in the index. Elaborations, such as proximity search, that employ nearness constraints require sequential placement information [21]. Ranking methods, such as extended boolean [20], fuzzy boolean [18], and relevance search [19] introduce weights, typically based on term frequencies.

The proposed text retrieval architecture supports the implementation of at least these search paradigms. This capability is actually a property of the index access protocol described above since the determining factor is what information can be extracted from that database. At least two strategies are possible. Methods may be defined whose contracts are to deliver term statistics of each desired sort, with an understanding that if that information is not immediately available in the index itself, a computation may be performed over the original source text to recover it (as in `get-frequency-postings` above). Alternatively, indices need only implement those access methods that can be serviced efficiently, and a constraint can be placed on the pairings of indices and search methods that can coexist in a complete system. This constraint is enforced by simply allowing the object system to note that no binding is provided for the required access method in the given system.

The demonstration system follows the first strategy by supplying access methods for binary postings (implemented as lists of document identifiers), and frequency annotated postings (implemented as lists of document identifiers paired with frequencies). These two methods are sufficient to support simple implementations of boolean search without negation and relevance search with inverse frequency term weights (see Figure 9). Note that the query input to `relevance-search` is simply a token stream, which is pumped for tokens in the usual manner. This allows for the possibility that the caller may apply a different parsing strategy on the query than the one supplied with the corpus. This feature is exploited to easily implement `relevance-feedback` by simply passing down an appending token stream, which effectively concatenates the contents of the provided document set.

Since the search algorithms are top-level entry points to the demonstration system there would be only marginal utility in organizing them as method protocol on a search object. Instead they are presented as procedures (which, in Common Lisp, simply makes them methods on the "anything" class `t`). There would be an advantage in specifying a search protocol if it was desirable to provide different implementations of the same search method tuned to different index implementations.

9 Sample Session

The disparate pieces of the demonstration system are brought together in a sample application (see Figure 10). A class `demo` is defined which mixes together a corpus implementation `string-corpus`, an analyzer `simple-analyzer`, a storage layer `hash-store`, and an index `binary-index`, and hence inherits the methods associated with each of these classes. It also uses an initialization protocol to load the corpus object with a string representation of each file in a given directory. The directory in question contains a collection of biographies donated by members of our laboratory, with each biography in a separate file. The `:after` method on `initialize-instance` notices and indexes each document accessible through the corpus object. Hence, simply creating an instance of `demo` class will perform all the computations required prior to search.

Figure 11 illustrates the output of some sample searches over this application. The first example evaluates a boolean search with the expression "information and (access or retrieval)". Results are returned in an unspecified order (which in this case is document ID order). The second example executes a relevance search with the textual query "information access". Here the results are presented in scored rank order. Finally, a feedback step over the document ID's 70, 86, and 27, yield the final results, also in similarity score order.

```
(defmethod boolean-search ((app t) expr)
  (labels ((resolve (x)
             (if (listp x)
                 (case (first x)
                   (and (intersection (resolve (second x)) (resolve (third x))))
                   (or (union (resolve (second x)) (resolve (third x)))))
                 (get-binary-postings app x))))
    (resolve expr)))

(defmethod relevance-search ((app t) query &optional (threshold 10))
  (let ((terms ())
        (scores ()))
    (map-tokens #'(lambda (token) (pushnew token terms :test #'string=)) query)
    (dolist (term terms)
      (let ((weight (/ 1.0 (get-term-frequency app term))))
        (dolist (freq-pair (get-frequency-postings app term))
          (let* ((id (car freq-pair))
                 (freq (cdr freq-pair))
                 (score-pair (assoc id scores)))
            (unless score-pair
              (setq score-pair (cons id 0.0) scores (cons score-pair scores)))
            (incf (cdr score-pair) (* weight freq))))))
    (mapcar #'car (subseq (sort scores #'> :key #'cdr) 0 threshold))))

(defmethod relevance-feedback ((app t) ids)
  (relevance-search
   app
   (make-instance 'appending-token-stream
                  :streams (mapcar
                            #'(lambda (id)
                                (make-token-stream app (open-document app id)))
                            ids))))
```

Figure 9: Generic Search Implementations

```
(defclass demo (string-corpus simple-analyzer hash-store binary-index) ()
  (:default-initargs
      :strings (mapcar #'file-to-string (directory "~/demo-corpus/"))))
(defmethod initialize-instance :after ((app demo) &key &allow-other-keys)
  (dotimes (id (length (slot-value app 'strings)))
    (index-document app id)))

(defun file-to-string (pathname)
  (with-output-to-string (string-stream)
    (with-open-file (file-stream pathname)
      (loop (let ((char (read-char file-stream nil)))
              (if char (write-char char string-stream) (return)))))))

(defmethod print-titles ((app demo) ids)
  (dolist (id ids (values))
    (format t "~&~3D ~A~%" id (read-line (open-document app id)))))
```

Figure 10: An Application

```
> (setq app (make-instance 'demo))
#<DEMO 27212576>

> (print-titles
   app
   (boolean-search app '(and "information" (or "retrieval" "access"))))

 20 Daniel M. Russell - System Sciences Laboratory
 22 George G. Robertson - System Sciences Laboratory - User Interface Research
 27 Jan O. Pedersen - System Sciences Laboratory
 43 Jock Mackinlay - User Interface Research
 50 Julian Kupiec - System Sciences Laboratory
 61 Herb Jellinek - User Interface Research
 70 Per-Kristian Halvorsen - System Sciences Laboratory, Natural Language
 86 Douglass R. Cutting - System Sciences Laboratory
 89 Stuart K. Card - System Sciences Laboratory / User Interface Research
 92 Francoise Brun-Cottan

> (print-titles app
                (relevance-search app
                                  (make-token-stream app
                                                     (make-string-input-stream
                                                      "information access"))))
 20 Daniel M. Russell - System Sciences Laboratory
 27 Jan O. Pedersen - System Sciences Laboratory
 70 Per-Kristian Halvorsen - System Sciences Laboratory, Natural Language
 22 George G. Robertson - System Sciences Laboratory - User Interface Research
 89 Stuart K. Card - System Sciences Laboratory / User Interface Research
 92 Francoise Brun-Cottan
 86 Douglass R. Cutting - System Sciences Laboratory
 61 Herb Jellinek - User Interface Research
 13 Mark Stefik -- System Sciences Laboratory
 50 Julian Kupiec - System Sciences Laboratory

> (print-titles app (relevance-feedback app '(70 86 27)))

 70 Per-Kristian Halvorsen - System Sciences Laboratory, Natural Language
 86 Douglass R. Cutting - System Sciences Laboratory
 27 Jan O. Pedersen - System Sciences Laboratory
 76 Dan Gerson - System Sciences Laboratory/Collaborative Systems Area
  7 John W. Tukey - System Sciences Laboratory - Consultant
105 John Batali -- SSL -- NLTT
 68 Pat Hayes - Embedded Computation Area
 36 Scott Minneman - Design, Use, and Shared Spaces Area
 16 Jeff Shrager
 97 Daniel G. Bobrow - System Sciences Laboratory
```

Figure 11: Sample Session

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[2] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *CHI'91 Conference Proceedings*, pages 181–187. ACM SIGCHI, ACM Press, April 1991.

[3] S. Christodoulakis and D. A. Ford. File organizations and access methods for CLV optical disks. In *Proceedings of the Twelfth Annual International ACMSIGIR Conference on Research and Development in Information Retrieval*, pages 152–159, June 1989.

[4] K. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 1989.

[5] D. R. Cutting, P.-K. Halvorsen, J. O. Pedersen, and M. Withgott. Information theater versus information refinery. In *AAAI Spring Symposium on Text-based Intelligent Systems*, Stanford University, Stanford, CA, March 1990. Also available as Xerox PARC technical report SSL-89-101.

[6] D. R. Cutting and J. O. Pedersen. The TDB cookbook. Xerox internal memorandum.

[7] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of SIGIR'90*, September 1990. Also available as Xerox PARC technical report SSL-90-10.

[8] J. L. Fagan. Automatic phrase indexing for information retrieval. In *Proceedings of the Tenth Annual International ACMSIGIR Conference on Research and Development in Information Retrieval*, pages 91–101, June 1987.

[9] C. Faloutsos and S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4), October 1984.

[10] Edward A. Fox and Robert K. France. Architecture of an object-oriented expert system for composite document analysis, representation, and retrieval. Technical Report TR-86-10, Virginia Tech, Department of Computer Science, Blacksburg VA 24061, April 1986.

[11] D. Harman and G. Candela. A very fast prototype retrieval system using statistical ranking. *SIGIR Forum*, 23(3,4):100–110, Summer 1989.

[12] IBM. *STAIRS/VS: Reference Manual*, 1979.

[13] R. Kaplan and M. Kay. Phonological rules and finite state transducers. Unpublished manuscript, 1982.

[14] L. Karttunen, K. Koskenniemi, and R. Kaplan. A compiler for two-level phonological rules. Report CSLI-87-108, Center for the Study of Language and Information, 1987.

[15] D. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1968.

[16] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[17] J. M. Kupiec. Augmenting a hidden Markov model for phrase-dependent word tagging. In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 92–98, Cape Cod, MA, 1989. Morgan Kaufmann.

[18] T. Radecki. Fuzzy set theoretical approach to document retrieval. *Information Processing and Management*, 15(5):247–259, 1979.

[19] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 41(4):288–297, June 1990.

[20] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, November 1983.

[21] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[22] G. L. Steele, Jr. *Common Lisp, The Language.* Digital Press, second edition, 1990.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.