

IXE Library Architecture

Giuseppe Attardi, Antonio Cisternino
Ideare SpA

July 2001

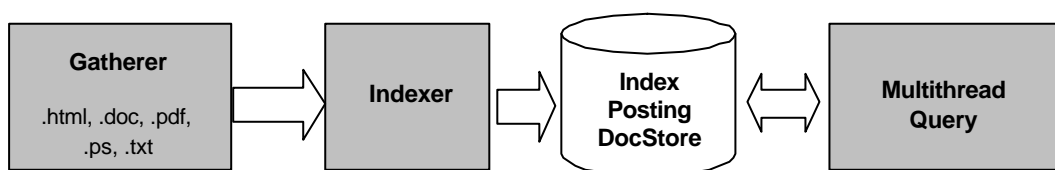
1. Architecture

IXE is a of C++ class library for indexing and searching large collections of documents. The library has been designed for efficiency and scalability. The library can handle multiple collections of documents with overall size of several Terabytes. Using the library several kinds of applications can be built, ranging from Web Search Engines, local disk indexing and retrieval, document management systems, specialized indexing and search of multimedia collection.

The library is provided with sample applications for indexing text documents in various formats (HTML, PDF, DOC) and the corresponding search server which can be used out of the box for implementing a Web Search Engine. Such sample applications are documented with separate **man** pages.

The overall architecture of an IXE application will consists in:

1. Gatherer: collects document for indexing
2. Indexer: builds the index for a collection of documents
3. Document store: repository of documents
4. Query server: processes queries to the index.



A simple gatherer is supplied with the library, but for a large scale search service, IXE can be interfaced to a more sophisticated spidering system.

The IXE library has been designed with a modular architecture, centered around a flexible relational database table, as described in Figure 1.

The Document Store contains document descriptions, including document attributes like name, title, abstract, date, size as well as document contents if desired. The store is capable of handling up to several Terabytes of material.



A library of C++ classes provides access to the functionalities of document gathering, extraction/analysis, indexing and search.

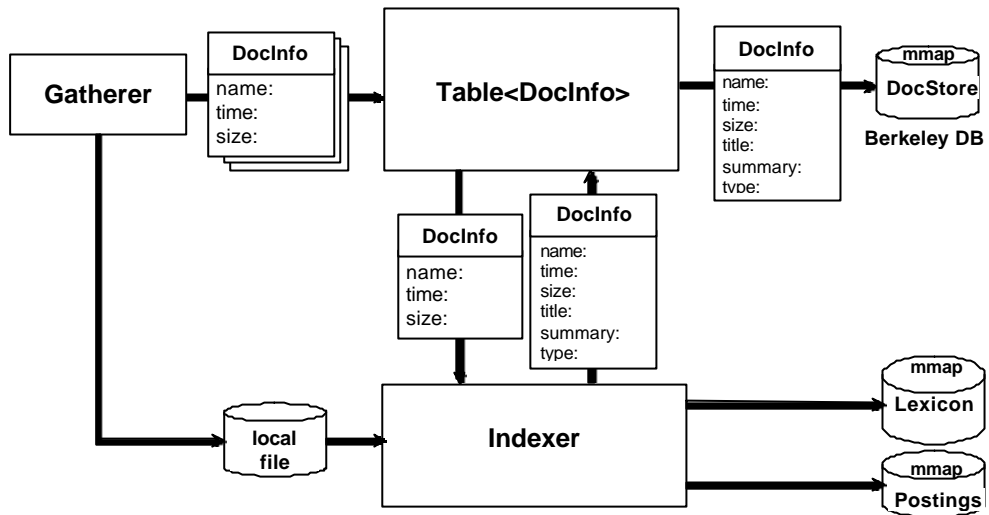


Figure 1. IXE Architecture (version 1.1)



2. Document Store

The document store is a relational database table, where documents are stored. Documents are described by `DocInfo` objects (i.e. instances of class `DocInfo` or its derivatives). A `DocInfo` contains the minimal set of attributes required to identify and retrieve a document from the store: i.e. a document ID, a size in bytes and a modification time. A comparison operator must be supplied for each kind of `DocInfo`, which is used to determine whether two documents are the same or the document has changed. For instance, if the size and modification time of the document has changed, it needs to be re-indexed. Specialized versions of `DocInfo` might include an MD5 code of the contents to determine equality.

`DocInfo` is an abstract class defined as follows:

```

class DocInfo {
    DocID      id;    // document ID, primary key
    size_t     size;  // in bytes
    time_t     time;  // modification time

    virtual bool operator ==(DocInfo const &other) = 0;

    META(DocInfo,
        (KEY(id, Field::autoincrement),
         FIELD(size),
         FIELD(time))
    );
};

```

The `META` construct allows specifying storage attributes for each field (member variable) of the class. In this example, `id` is specified as an *autoincrement* field, i.e. a *primary key*, whose value is automatically incremented when inserting an object with an `id = 0`; `size` and `time` are specified as ordinary fields.

`META` is a macro which exploits template metaprogramming for creating a metaclass for the class, as described later.

The document store can store object of any concrete class derived from `DocInfo`. For instance, the following class describes Web pages:

```

class PageInfo : public DocInfo {
    char const *   name;    // unique name: pathname or URL
    Text<255>      source;  // external text, contains source filename
    char const *   site;   // site (domain)
    char const *   title;
    char *         summary;
    char const *   type;   // MIME content-type
    bool          sourceAvailable_; // whether we store the source doc

    META(PageInfo,
        (METACLASS(DocInfo),
         VARKEY(name, 2047, Field::unique),
         VARKEY(source, 255, Field::external),
         VARFIELD(site, 2048),
         VARFIELD(title, 2048),
         VARFIELD(summary, 2048),
    );
};

```



```

        VARFIELD(type, 512))
    );
};

```

The example uses **unique** fields, which are equivalent to primary keys, i.e. they are keys whose value must be unique in the table column. They can be used to identify an object in alternative to the primary key, when this is not known. For instance, we can insert a **PageInfo** record in table with a 0 value ID: the table will use the name to determine whether the document is already present: if so it will replace the previous record, otherwise it will assign a new ID and insert it.

In practice it is inconvenient to replace a document record for a document containing full text fields, if the document has not changed, since such insertion would require updating the whole index. Therefore the table checks whether the document has changed, by testing equality through a virtual method, which can be specialized for each subclass of **DocInfo**.

Two **PageInfos** are considered to describe the same document if they have the same size, modification time and URL:

```

virtual bool operator ==(DocInfo const &other) {
    return (size == other.size) &&
           (time == other.time) &&
           !::strcmp(name, ((PageInfo&)other).name);
}

```

Each class of **DocInfos** can be printed to a stream each in its most appropriate way by specializing method **serialize**. For instance **PageInfo** can be printed in a standard way, as tab separated columns, by this method:

```

ostream& PageInfo::serialize(ostream& s) const
{
    return s << name << '\t'
           << site << '\t'
           << title << '\t'
           << setw(160) << summary << '\t' // limit to approx 3 lines
           << type;
}

```

In C++, creating or opening a table of documents entails just constructing an object of class **Table**, for instance:

```
Table<PageInfo> table("/ixe/web");
```

creates a table for storing **PageInfos** in table **/ixe/web**.



3. Indexer

Class **Indexer** performs full text indexing on several columns of a table of documents. Each full-text column has its own index, which is created or updated when a new document is inserted in a table. Class **Indexer** has the following public interface:

```
class Indexer
{
public:
  Indexer(char const *dirname, size_t num_columns, bool incremental);
  ~Indexer();
  size_t add(DocInfo *docInfo, char const* source, int col,
            Field::IndexType type);
  void replace(DocID docID, byte* object);
}
```

Whenever a document is inserted in the table, for each full-text column it contains, the indexer is called supplying the **DocInfo**, the column number, the text source and the index type. The index type can be either **Field::fulltext** or **Field::external**. In the former case, the string in **source** is indexed and stored in the document store, while in the latter case, **source** contains the pathname of the file with the document contents and just the pathname is stored.

An indexer receives a short description (class **DocInfo**) of a document to be indexed. From such information (for instance the filename extension) it creates and invokes an appropriate document reader (class **DocReader**), which extracts occurrences of each term from the document and calls method **hit()** on the indexer for the appropriate column (**ColumnIndex**). Moreover a reader may extract information from the document to be added in appropriate fields in **DocInfo**.

The generated index is made up of two parts:

Lexicon	Associates to each word its frequency in the collection as well as its.
Postings	Stores the postings for each word in each document: document ID, word colors, document frequency and positions of occurrences within document.

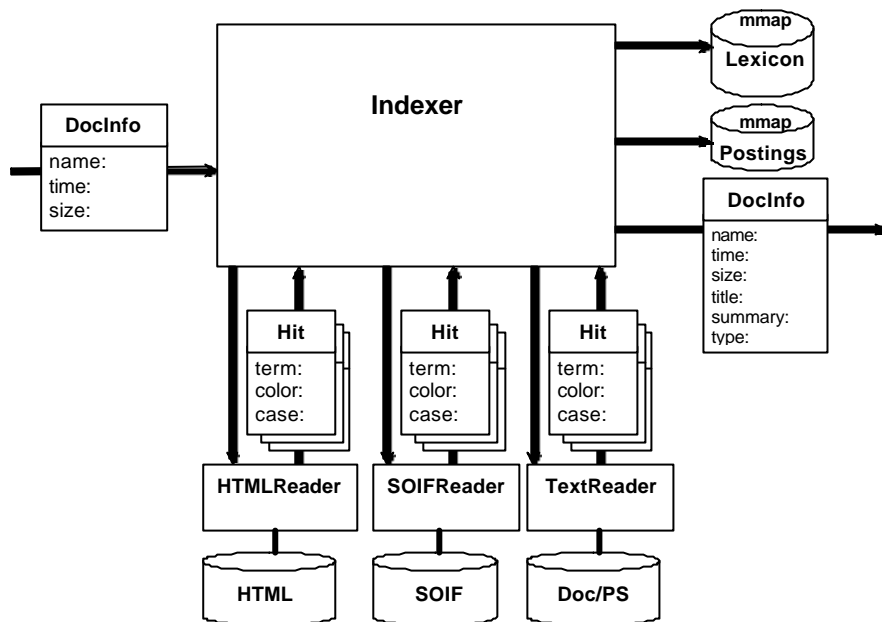


Figure 2. IXE Indexer (version 1.2)

3.1. Document Readers

The architecture is modular, and new document readers can be added as necessary to deal with other document formats. The task of a reader is to extract not just terms from documents, but full hit information, i.e. *case* (**upper**, **lower**, **capital**) and *coloring*, i.e. within which document structure or region (zone in Fulcrum SearchServer terminology) the term appears. For instance, the reader for HTML documents is capable of determining the tag or the element class within which a word appears.

Each reader is supplied with a specialized **DocInfo** structure, i.e. an instance of a class derived from **DocInfo**, which it can fill from information extracted from the document, e.g. its title and its summary, which it adds to its. For instance, the HTML reader uses a **PageInfo** structure, as defined above.

Such extended **DocInfos** are then stored in a Table implemented by means of a Sleepycat Berkeley Database. This DB is used when answering queries, providing summary information about selected documents.

Currently available readers include:

- GenericReader** reads text documents, and is capable of extracting text from Microsoft Office documents, and to skip encapsulated PostScript
- HTMLReader** parses HTML pages, extracts title, and summary and detects word colors, i.e. occurring within an HTML tag specified as color in the configuration file or within the CONTENT attribute of a metatag (in this case the color is the NAME



attribute of the metatag and META must be included among the specified tag colors).

The title consists in the words between the first <TITLE> ... </TITLE> pair. If no such title is present, the title is simply the file (not path) name.

The summary consists of the first 2048 bytes of the document, stripped of HTML tags.

Soi fReader reads files in SOIF format, produced by Harvest.

MemReader reads text from a string.

3.2. Incremental Indexing

IXE is capable of generating an index incrementally: new documents are added, old documents which appear changed, according to the equality operator in their **DocInfo** class, are replaced.

3.3. Customization

IXE indexing can be controlled either through command line options or through the configuration file **ixe.conf**.

In particular one can specify:

1. the association between file names and (MIME) types, used to select the appropriate document reader:

```
FileType HTML      *.asp *.htm* *.jsp *.php
FileType SOIF     *.soif
FileType text     *.txt *.text
```

2. which HTML elements to consider as colors, with associated weights:

```
Color 4 title
Color 3 h1
Color 2 h2 h3 h4 a
```

3. which elements to skip

```
ExcludeElement style script object
```

4. which files to include or exclude from gathering:

```
IncludeFile *.txt
ExcludeFile *.gif *.jpg
```

5. which converter to use for extracting text from documents:

```
FilterFile *.bz2 bzip2 -c %f > @E
FilterFile *.gz gunzip -c %f > @E
```



```
FilterFile *.Z      unzip -c %f > @E
FilterFile *.zip    gunzip -c %f > @E
FilterFile *.pdf    pdftotext %f @E.txt
```

6. whether to perform incremental indexing:

Incremental no



4. Search facilities

IXE supports Boolean search, phrase search, proximity search, prefix search, attribute search and color search.

4.1. Phrase search and proximity

Phrase search can be expressed by either “” or by infix characters, including “-“:

```
body matches “network computing”
```

```
body matches network-computing
```

Proximity search is expressed as follows:

```
body matches proximity 8 ((giuseppe attardi) & (associate professor))
```

4.2. Attribute search

Searching for words within a specified document attribute (column) can be expressed as:

```
site matches “it.unipi.*” and body matches network computing
```

The search will return all documents containing both words “network” and “computing” appearing in URLs from domain `uni pi . it`, including its subdomains.

4.3. Color search

It is possible for instance to search for words appearing within a certain HTML element:

```
body matches title = software AND linux
```

will search for documents which contain the word “software” in the title and the word “linux” anywhere. Also meta tags are used as colors:

```
body matches distribution=global academic-year
```

```
body matches author=giuseppe-attardi web-design
```

Which tags are to be indexed and whether to add meta tags as colors is controlled by entries in the configuration file (`ixe.conf`).



5. Search Server

IXE uses multithread architecture in its search server. The search server is activated with the **-b** switch:

```
search -b -i germany1 -i germany2
```

activates a search demon on two tables called **germany1** and **germany2**. A number of threads (specified by the configuration variable **ThreadsMin_Default** (5) or by the switch **-t**) is activated and used in round-robin. If all threads are busy, additional threads are activated, up to **ThreadsMax_Default** (or **-T**).

Threads that remain idle for a long period of time are deactivated.

To avoid that threads be blocked by a request submission that takes too long to arrive, there is a timeout on reading the input from socket, controlled by **SocketTimeout_Default** (10 seconds), or by **-o**.



6. Cursors

IXE provides a cursor interface to extract objects from a table of objects of class **T**:

```
Table: : Cursor<T> cursor(collection);
while (cursor.hasNext())
    cout << cursor.get();
```

Notice that method `cursor.get()` returns a genuine object of class **T**, whose methods can be directly invoked, without any need to reconstruct it or extract its individual fields through explicit code.

Several methods are available on **Cursor** objects, for performing:

1. *initialization*: constructor returns a cursor positioned before the first element;
2. *test*: `atEnd()`, `hasNext()`;
3. *access*: `get()`, returns the element pointed by the cursor, `next()`, moves the cursor and returns the next element.
4. *specific cursor operations*, depending on its type.

For instance, dumping the lexicon is done in the following way:

```
Lexicon: : Cursor cur = lexicon.cursor();
while (cur.next())
{
    word = cur.get();           // fetch the word
    cout << word;
    postList = cur.postlist(); // fetch the posting for the word
    cout << postList;
}
```

6.1. Query Cursors

Besides generic cursors, which scan a whole table or an inverted index, it is possible to use query cursors, which allow examining the results of queries. For example, consider the following class:

```
class VideoInfo {
    DocID id;
    char* title;
    char* format;

    META(DocInfo,
        (KEY(id, autoincrement),
         VARKEY(title),
         FIELD(format)));
};
```



The following code implements a search on a collection of video documents, described as objects of class `Video`:

```
Collection<Video> collection("/ixe/video");
Query query("title MATCHES Totti and format=wav");
QueryCursor<Video> cursor(collection, query);
while (cursor.hasNext())
    cout << docInfo(cursor.get());
```

The first line opens a `Collection` of `Video` objects, located in file `"/ixe/video"`. Then a query is created from an SQL-like expression. Search is performed by opening a `Cursor` on the collection for the results of the query. Each result is extracted by scanning the cursor.

A `QueryCursor` returns `QueryResult`'s, which describe individual results of a query. A `QueryResult` contains:

```
class QueryResult {
    CollectionID collection_;
    DocID      id;
    double     rank_;
    HitPosition position_;
}
```

The `rank_` is the information retrieval rank, computed by the cosine measure, and is used to sort the results by relevance to the query. Position information is useful for queries involving proximity.

So a typical loop for a search application would look like this:

```
vector<QueryResult> sorted;
QueryCursor<T> cursor(coll, query);
while (max_results-- && cursor.hasNext()) {
    tot++;
    if (i < sorted_size) {
        sorted.push_back(cursor.get());
        i++;
    } else {
        // remove smallest
        ...
        std::push_heap(sorted.begin(), sorted.end(), rank_greater());
    }
}
std::sort_heap(sorted.begin(), sorted.end(), rank_greater());
```

Since only `max_results` are required, the code maintains a heap of this size where results get sorted. The example uses a `rank_greater()` function object to sort results by information retrieval rank, but other applications can use different criteria. For instance, if an overall document rank is required, one can create an additional table:

```
class PageRank {
    DocID      docID;
    Double     rank;
}

Table<PageRank> PageRankTable("rank");
```



where items can be inserted and extracted as follows:

```
PageRankTable insert (pageRank);
Table<PageRank>: Cursor c(PageRankTable);
while (c.hasNext()) {... c->get() ...}
```

Random access to the elements is provided by:

```
PageRank = c->get((byte*)&docID);
```

6.2. Multiple Collection Search

For performing a search over a set of collections, one can do:

```
vector<Collection*> collections;
collections.push_back(collection1);
...
collections.push_back(collectionN);
QueryCursor cursor(query, collections);

QueryResult res = cursor.next();
```

A simple and effective way to perform efficient ranking of results on large collections is to create separate collections with documents of different overall importance, quality or freshness. For instance, one can put Web pages with high link popularity or referenced from authoritative sites in one collection and other documents in other collections. Since the search scans the collection in the order in which they are supplied in the command line, if a significant number of results are obtained from earlier collections, the search can just be cutoff at a fixed amount (e.g. Google stops at 40000). This has no detriment for queries looking for unusual keyword combinations, since all documents are searched if not enough answers are present in higher ranking collections.

Queries can be expressed either in a general SQL-like notation, or in a simplified notation suitable for interfacing directly to a search engine from a CGI script.

6.3. SQL Query Syntax

Queries are parsed by a bison parser (`Query/parser.yy`), using a flex lexical analyzer (`Query/parser.lex`). The parser produces a structure of type `Query`, subclass of `QueryExpr`, that represent the parse tree of the query condition.

The current query condition syntax is:

```
condition:      match
                | 'NOT' match
                | match relOp condition

match:          ident 'MATCHES' pattern
                | '(' condition ')'
                | comparison

pattern:        attr patternOp pattern
                | attr pattern
```



```

    | attr
attr:  color '=' primary
    | primary
color:  ident
ident:  WORD
primary: (' pattern ')
    | '!' primary
    | WORD
    | WORDSTAR
    | PHRASE
    | proximity
proximity: ' PROXIMITY' dist (' proxlist ')
dist:  LONG_NUM
proxlist: (' wordlist ') '&' proxlist
    | (' wordlist ')
wordlist:  WORD wordlist
    | WORD
relOp:  'AND'
    | 'OR'
patternOp: '&&'
    | '||'
comparison:  operand '=' operand
    | operand NE operand
    | operand LT operand
    | operand LE operand
    | operand GT operand
    | operand GE operand
    | operand LIKE operand
    | operand NOT LIKE operand
    | operand BETWEEN operand AND operand
    | operand NOT BETWEEN operand AND operand
operand:  factor
    | operand '+' factor
    | operand '|' factor
    | operand '-' factor
factor:  power
    | factor '*' power
    | factor '/' power
power:  term
    | term '^' power
term:  field
    | LONG_NUM
    | FLOAT_NUM
    | STRING
    
```



```
field: ident
      | ident '.' ident
```

6.4. Query Processing

For performing search, a query first gets compiled, then its cursor(s) get initialized.

```
QueryCursor<T>::QueryCursor(Collection<T>* collections,
                             Query& query) :
{
  current(QueryResult::min())
  {
    cursor = compile(collection, query, stop_words);
  }
}
```

Compiling the query entails generating a suitable structure used for query evaluation. Query evaluation is performed at each iteration step through the query cursor, by calls to virtual method

```
virtual QueryResult* QueryCursor<T>::next(QueryResult& min);
```

This method looks for the next result in the collection that satisfies the current sub-condition and is greater or equal to `min`. A `QueryResult` is a triple `<CollectionID, DocID, rank>`, ordered by `CollectionID` first and then by `DocID`. This form of iteration allows optimizing query processing by skipping useless results: for instance in a Boolean AND keyword query, one can look for the first `DocID` in the posting list of each word, select the largest one and check for its presence in the other posting lists.

Class `QueryCursor` has several subclasses, one for each kind of condition:

<code>QueryCursorAll</code>	boolean AND keywords
<code>QueryCursorSome</code>	boolean OR keywords
<code>QueryCursorAnd</code>	AND of several subconditions
<code>QueryCursorOr</code>	OR of several subconditions
<code>QueryCursorNot</code>	NOT subcondition
<code>QueryCursorIndex</code>	search a single indexed column
<code>QueryCursorFilterEq</code>	= condition on a single indexed column
<code>QueryCursorFilterRange</code>	>, >=, <, <= condition on a single indexed column
<code>QueryCursorJoin</code>	general SQL join on record row
<code>QueryCursorWord</code>	fulltext: single word in given color
<code>QueryCursorWordStar</code>	fulltext: prefix word in given color

Each class provides its own implementation of method `next()`, typically maintaining its own internal cursor(s), moving to the next candidate result and checking whether it satisfies the condition.

For instance class `QueryCursorWord` maintains an iterator to the posting list for the word. `QueryCursorWord::next(QueryResult* min)` advances to the next item through the `PostingList` iterator that is greater than `min`, and then checks that the query word appears in the required color before returning it.

Use of this operator effectively implements the so called Adaptive set intersection algorithm discussed in [Demaine 01]. By reordering the clauses in a conjunction according to the size of the result set, leads to the variant so-called Small Adaptive.



Class **QueryCursorIndex** handles a comparison expression on a single indexed field of a table. It uses a cursor on the secondary index of the table.

Class **QueryCursorJoin** handles comparison expressions on fields of a row, and uses a cursor on the primary key of the table.

More complex queries are made up as boolean combinations of the previous ones.

Compiling a query may result in simplifications, for instance, if a query word turns out to be a stop word, it is discarded.



7. Web Search Application

The program `searchWeb` performs queries on an index of Web pages, described by class `WebInfo`:

```

META(WebInfo,
  (SUPERCLASS(DocInfo),
  VARKEY(name, 2048, Field::unique),
  KEY(text, Field::fulltext),
  VARKEY(site, 255, Field::fulltext),
  VARFIELD(title, 2048),
  FIELD(MD5high),
  FIELD(MD5low),
  VARFIELD(description, 2048))
);

```

`searchWeb` supports queries with the following syntax:

```
search?q=query[&option]*
```

where

```

query:  expr+ [cache:docid:url]
expr:  [-]term | all | any
any:   word {OR word}+
term:  [where:][word | phrase] | site:domain
where: intitle | inurl
phrase: "word+" | word{'SEP' word}+
SEP:   [.-='\<>:;]
all:   [whereall:]word+
whereall:  allintitle | allinurl
domain: word{.word}*

option:
start=num      (first result, default: 0)
num=num        (number of results, default: 10)
hl=language   (interface language, default: en)
lr=lang_??[|lang_??]*
                 (search pages in these languages, default: none)
filter=0      (do not omit similar results, default: none)

```



8. File formats

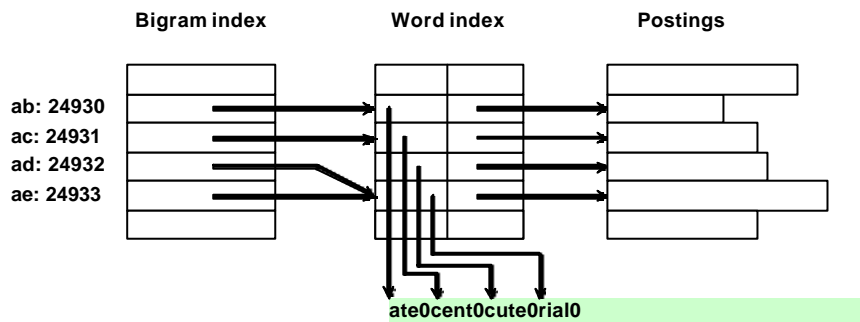
IXE uses three secondary storage structures:

1. Main index: contains the lexicon and the stop words and color indexes (.fti);
2. Postings: contains the postings for each word in each document (.pst);
3. Document Store: contains details and properties about each document (.bdb).

The index and postings files are accessed as memory mapped files, while the Document store is implemented as a Berkeley Data Base (public version from Sleepycat Software), which uses as well memory mapping as appropriate.

8.1. Lexicon

The lexicon is organized in a two level hierarchy, similar to a *suffix array with a supra-index*. The first level consists of a table of *bigrams*, corresponding to the first two letters of a word. An entry in this table contains the index of the first word in the full index starting with that bigram, as illustrated in the following picture:



Suppose the following words are present:

abate
 accent
 acute
 aeri al

The bigram index is used to narrow the search to a portion of the full index.

When searching for the word 'acute', first we extract 'ac', perform a look up in `bigram_index` and get the `begin_index` (= `bigram_index['ac']`), which refers to the first word starting with 'ac', i.e. 'accent'.



The next entry in `bigram_index` (`bigram_index['ac' + 1]`) contains the index (`end`) of the first word lexicographically after 'ad' (it may or may not start with 'ad'). In this case it is 'aerial', since there are no words starting with 'ad'.

A binary search is performed using the STL procedure `bwer_bound` with arguments the two iterators `begin` and `end` and so 'acute' is found.

The entries after the last one pointing to a word in the index are filled with the index one beyond the last one in `word_index`.

So in general `bigram_offset[bg]` contains, for each bigram `bg` (two letter prefix), the index of the first word lexicographically after `bg` (it may not start with `bg`).

8.2. Multiple Columns

An index may contain several columns are implemented by adding one prefix byte, representing the column number, to each indexed term. Therefore the full index structure is as follows:

8.3. Format of index file

```
FileHeader {
    FileFormat ff;
    Count num_docs;
    Count num_columns;
} fileHeader;
(columnIndex) 1
...
(columnIndex) num_columns
Count    num_stop_words;
off32_t  stop_word_offset[num_stop_words];
size_t   stop_word_size;
(stop-word index)
Count    num_colors;
off32_t  color_offset[num_colors];
size_t   color_size;
(color index)
```

where a column index is as follows:

```
Count    num_bigrams;          (0 if not present)
long     bigramible[num_bigrams];
Count    num_words;
LexEntry lexicon[num_words];
long     lex_size;
(lexicon)
```

The lexicon consists of a sequence of consecutive null-terminated words (stripped of the first two characters).



8.4. Postings File

The postings file contains just a sequence of consecutive doclists. Each *doclist* represents all the occurrences of a word in all documents and has the form:

`{DF}{posting}DF`

that is: the word document frequency (DF) followed by exactly DF postings, where a *posting* is:

`I [0xFF{C}...0xFF]{O}{L}{H}L`

that is: a docID (I) followed by zero or more colorID's (C) surrounded by 0xFF bytes followed by the number of occurrences in the document (O), followed by the length in byte of the hits (less O, to save space) and then by L hits.

A hit for the moment is just an ECD representing the position of the word in the document. Positions are represented as delta increments.

Note that color information is incomplete, since it does not express which occurrence of the term appeared with that color.

The integers are expressed as ECD (Epta Coded Digit, i.e. base 127), not raw binary. An ECD integer is represented as a series of digits in base 127, with all but the last digit having 0 in the sign bit.

8.5. Long Posting lists

In order to speed up queries involving long posting lists, a binary table is added in front of posting lists longer than **LongPostingThreshold** (8000). It contains pairs `<DocID, postingOffset>` for every thousands posting. This allows skipping to the posting list with closest thousands and to limit scanning sequentially up to 1000 Posting lists. The size of the block to skip should be close to the machine page size, to optimize **mmap** and disk access.

8.6. Color Entries

The color index is a list of all the HTML tag/meta names (in alphabetical order) selected for indexing. Each color is followed by its numeric ID that is simply a unique integer to identify it and its weight. Each entry is of the form:

`color0{I}{W}`

that is, a null-terminated color followed by the ID (I) and weight (W) in BCD format.



9. Basics of Information Retrieval

9.1. Documents and terms

In IR (Information Retrieval), the items we are trying to retrieve are called *documents*, and the documents are described by sets of *terms*. Usually a document is thought of as a piece of text and a term as a word or phrase which helps to describe the document, and which may indeed occur in the document, one or several times. So a document might be about dental care, and could be described by corresponding terms ‘tooth’, ‘teeth’ ‘toothbrush’ ‘decay’, ‘cavity’, ‘plaque’, ‘diet’ and so on.

More generally a document can be anything we want to retrieve, and a term any feature that helps describe the documents. So the documents could be a collection of fossils held in a big museum collection, and the terms could be morphological characteristics of the fossils. Or the documents could be tunes, and the terms could then be phrases of notes that occur in the tunes.

If a document D , is described by a term t , t is said to *index* D , and we can write,

$$t \rightarrow D$$

In fact an IR system consists of a set of documents, $D_1, D_2, D_3 \dots$, a set of terms $t_1, t_2, t_3 \dots$, and set of relationships,

$$t_i \rightarrow D_j$$

i.e. instances of terms indexing documents. A single instance of a particular term indexing a particular document is called a *posting*.

For a document D , there is a list of terms which index it. This is called the *term list* of D .

For a term t , there is a list of documents which it indexes. This is called the *posting list* of t . So each posting refers to a document in which the term appears, but additional information is often stored in the posting, e.g. the number of occurrences of the term in the document or their position.

A term t is said to index a document D , or $t \rightarrow D$. It will often be the case that D is made up of a list of words,

$$D = w_1, w_2, w_3 \dots w_k$$

and that many, if not all, of the terms which index D derive from these words. The relation between words and terms need not be a simple one. A single term ‘connect’ might derive from a number of words, ‘connect’, ‘connects’, ‘connection’, ‘connected’ and so on. A single word might give rise to more than one term. Nevertheless, if a term derives from words w_9, w_{38}, w_{97} and w_{221} in the indexing process, we can say that the term ‘occurs’ in D at positions 9, 38, 97 and 221, and so for each term a document may have a vector of positional information. These are the *within-document positions* of t , or the *wdp* information of t .

The *within-document frequency*, or *wdf*, of a term t in d , denoted as $f_{d,t}$, is the number of times the term appears in document d .



The *document collection frequency*, or *document frequency*, f_t is the number of documents in the collection where the term t appears.

The *length of a document* d , made up of k words, w_1 to w_k , is defined as $|d| = k$.

The *normalized document length*, or *ndl*, is k divided by the average length of the documents in a collection. So the average length document has *ndl* equal to 1, short documents are less than 1, long documents greater than 1.

9.2. Ranking

IXE computes the *document term weight* as follows:

$$w_{d,t} = 1 + \log(f_{d,t})$$

The logarithm is used to give diminishing returns as term frequencies increase. $f_{d,t}$ is adjusted to take into account also the color of the occurrences of term: e.g. if a word occurs in a title it counts as 4 occurrences, where 4 is the weight for title.

The term weight w_t is computed as:

$$w_t = \log(1 + N / f_t)$$

The cosine measure of similarity between a query Q and a document d is:

$$\text{cosine}(Q, d) = \frac{1}{W_d} \cdot \sum_{t \in Q} w_{d,t} \cdot w_t = \frac{1}{W_d} \cdot \sum_{t \in Q} (1 + \log(f_{d,t})) \cdot w_t$$

$$W_d = \sqrt{\sum_{t \in Q} f_{d,t}^2}$$

By approximating $W_d = \text{sqrt}(|d|)$, we have:

$$\text{cosine}(Q, d) = \frac{\sum_{t \in Q} (1 + \log(f_{d,t})) \cdot \log(1 + N / f_t)}{\sqrt{|d|}}$$

This is not much different from the following measure of similarity between a query Q and a document d :

$$M(Q, d) = \sum_{t \in Q} \frac{1 + \log(f_{d,t})}{|d|} \cdot \frac{1}{f_t}$$

An earlier version of IXE computed the *document term weight* as follows:

$$w_{d,t} = \frac{10 + \log(f_{d,t})}{|d|}$$



(dividing by $|d|$ introduces a *normalization* factor to discount the contribution of long documents) and stored the value

$$w_{d,t} \cdot wt = \frac{10000}{F_t} \cdot \frac{10 + \log(f_{d,t})}{|d|}$$

within the posting information of term t for document d in the index, where F_t is the total number of occurrences of term t in the whole collection and the 10000 factor ensures an integer value for the rank. Using these values, ranking a document with respect to a query Q requires just adding up such values for each $t \in Q$, but increased the size of the index of approximately 10%.

10. References

- [Demaine 01] [Erik D. Demaine](#), [Alejandro López-Ortiz](#), and [J. Ian Munro](#), “Experiments on Adaptive Set Intersections for Text Retrieval Systems,” in [Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments](#), Lecture Notes in Computer Science, Washington, DC, January 5-6, 2001.