

Design, Implementation and User's Guide to URSA, the UNICODE Retrieval System Architecture

Mark W. Davis and William C. Ogden
{madavis, ogden}@crl.nmsu.edu
Computing Research Lab
New Mexico State University
Las Cruces, NM 88005

1.0 Introduction

URSA¹, the UNICODE Retrieval System Architecture, is a high-performance text retrieval system that can index and retrieve UNICODE text. UNICODE is a 16-bit character encoding system that can represent most of the world's languages. As a result, URSA has the capacity to index and retrieve documents in all of the languages that can be represented in UNICODE. URSA also comes with tools for converting texts in 105 character sets into UNICODE (and back again). A quick overview of some of the other features of URSA shows how unique the URSA implementation is in comparison with the bulk of experimental retrieval engines. In URSA, for example, we have implemented the most comprehensive set of query and document weighting functions ever in an information retrieval system. The complete spectrum of weighting and ranking functions implemented in URSA represent the bulk of the weighting schemes developed in the past 40 years of text retrieval research, and includes the most recent successful document weighting schemes from Cornell and City University. Further, using a posting compression scheme that is both simple enough to allow rapid decompression and combination of posting data, yet which is specifically tuned to the kinds of data in postings, URSA indexes are only about 12%-25% of the size of the original texts. Finally, the URSA tools are robust enough to be used in industrial-grade applications and are based on a very simple object-oriented API. We have implemented an interactive retrieval engine to test approaches to document visualization using the URSA tools².

In this technical report, I overview the design, implementation and provide a sketch of the relevant APIs of the URSA libraries. Additionally, I present a guide to using the J24 tool set. The name "J24" comes from the July 24, 1998 deadline that I had set for finalizing the main components of URSA. The next section is a quick-start overview of how to use the

-
1. The URSA home page is at <http://crl.nmsu.edu/Research/Projects/tipster/ursa> and contains downloadable research papers and links to J24 and a cross-language text retrieval demo systems.
 2. The J24 interactive demo is available at: <http://crl.nmsu.edu/users/madavis/J24> or through the URSA home page at <http://crl.nmsu.edu/Research/Projects/tipster/ursa>.

J24 tool set to convert, index and retrieve documents in a wide variety of languages. In this document, I will use “J24” to refer to the tools and sample distribution, while “URSA” will refer to the larger project and the component libraries. After the quick start, I present a detailed discussion of the design of URSA that describes how the system stores and represents documents, tokens and postings, and the range of query weighting schemes understood by URSA. Next, I present a sketch of the query API, illustrating how to write custom applications to query and retrieve documents from URSA indexes. Finally, I provide detailed manual pages for each of the stand-alone J24 tools.

2.0 Quick-Start Guide to URSA and J24

2.1 Overview of J24 tools

In the sample J24 distribution, the *bin* subdirectory contains the following executables:

- **uconv**: converts texts from/to 105 character sets, including `ucs2` and `utf8` UNICODE representations.
- **j24_index**: indexes `ucs2` text.
- **j24_lookup**: retrieves postings and term information from URSA/J24 indexes.
- **j24_docstat**: retrieves document texts, summaries, weighting information and collection weighting data.
- **j24_query**: performs ranked retrieval of a UNICODE query, returning a ranked list of document numbers.
- **j24_trec**: performs ranked retrieval of a set of TREC-style queries, returning the ranked document identifiers lists in a format understood by the TREC evaluation tools.

The executables are distributed in pre-compiled form for Solaris 2.6. We can also provide source code or additional platform distributions to appropriate parties.

2.2 Quick-Start Overview

For this quick-start overview, we will:

1. Use **uconv** to convert texts from native codesets into `ucs2`.
2. Index the converted texts using **j24_index**.
3. Query and get ranked document lists back using **j24_query**.
4. Get document texts using data using **j24_docstat**.

2.2.1 Convert native codesets to `ucs2` using **uconv**

`ucs2` is the 16-bit representation of UNICODE in which every character is represented by a 16-bit value (0-ffff). Most native character sets for representing Western languages use 8-bit representations. ASCII, for example, is a subset of ISO 8859-1, which makes use of

the character range 128-255 to encode accented characters common to Western European languages. ISO 8859-1 is perhaps the most common language (and default) native encoding method used on computers today. UNICODE provides an alternative method for encoding documents, `utf8`, which mixes 8-bit and 16-bit representations together in the same data file. In essence, ASCII characters in `utf8` remain 8-bit characters, while all other characters have an extended, multi-byte encoding.

For this step in the quick-start overview, we will convert five documents from their native codesets into `ucs2` using **uconv**. **j24_index** can only index the `ucs2` form of UNICODE texts, although it uses `utf8` for its configuration files, to make it easier to modify the configuration files without special tools.

uconv requires external data files in order to convert different character sets. The program recognizes the environment variable `MUTTLIBDIR` that defines the location of the external data files. Set `MUTTLIBDIR` to the path of the `uconv_data` subdirectory:

```
% setenv MUTTLIBDIR /home/J24/uconv_data
```

Your path to the J24 distribution may differ from `/home/J24` in the these examples.

Convert the sample data files to `ucs2` (examples are shown executing from the J24 distribution directory):

```
% bin/uconv -s iso8859-1 -d ucs2 < data/english.iso8859-1 >
    data/english.ucs2
% bin/uconv -s koi8 -d ucs2 < data/russian.koi8 >
    data/russian.ucs2
% bin/uconv -s gb2312 -d ucs2 < data/chinese.gb2312 >
    data/chinese.ucs2
% bin/uconv -s jisx0212 -d ucs2 < data/japanese.jisx0212 >
    data/japanese.ucs2
% bin/uconv -s iso8859-1 -d ucs2 < data/french.iso8859-1 >
    data/french.ucs2
```

Note that the source character set is given by following `-s` and the destination character set is specified following the `-d`. To see the complete set of supported character sets for `uconv`:

```
% bin/uconv -h
```

If you load `data/english.ucs2` into a text editor, you will see that the conversion of the 8-bit characters to 16-bit characters makes the file appear strange. The same will occur using **xterm** or **kterm** to view the resulting Chinese and Japanese files.

2.2.2 Index UNICODE files using **j24_index**

The next step is to index the `ucs2` files. **j24_index** makes use of the URSA `ucs2` tokenization strategy to tokenize (gather words or characters from) the documents. The tokenization strategy is fairly sophisticated and can discover words in Western languages, Arabic and Eastern European dialects, with equal ability. For Chinese, character n-gram

sequences are indexed (1-grams by default), while for Japanese, sequences of Katakana and Hiragana are tokenized as words, while Kanji is subject to n-gram tokenization like Chinese. Similar rules apply to Korean, although separate switches are available for performing ngram tokenization of Hangul sequences. Note that although many n-gram sequences in Chinese may cross word boundaries and therefore be of little linguistic interest, there is a growing body of evidence that, for example, bigram tokenization of Chinese ideograms may perform better in information retrieval applications than more complex segmentation schemes. The URSA libraries nevertheless support plug-in tokenization and morphological analysis modules to provide for experimental examination of the impact of segmentation and morphological analysis, although the J24 toolkits only support Spanish, English and French suffix stemming (set via configuration file).

The structure of documents is fairly proscribed for J24 indexing using **j24_index**. Each document within a file must have a unique markers specifying the beginning and end of the document. Within the document there must be boundaries for document id start and end, for text start and end, and (optionally) for document summary start and end (this is also useful for title fields). The markers used for these categories are specified in a configuration file. The sample configuration file *data/common.cfg* contains standard LDC/TREC/TIPSTER-style markup for each of these boundary tokens, as well as boundary tokens for standard queries. The configuration file can also optionally specify a LANG variable. At present only ENGLISH, SPANISH and FRENCH are understood by the system and specify which version of fast, finite-state suffix-stemmers to apply to the data at index time. The RULES variable, if set to 1, will apply phrase indexing rules (again only for ENGLISH, SPANISH and FRENCH), indexing sequences of contiguous words that were not killed. Any ucs2 file containing newline-separated token patterns can also be specified using the KILLFILE variable. The WEIGHT variable indicates an octet query and document weighting scheme that is only relevant to **j24_query** and **j24_trec**. The WEIGHT octet specifies how the query terms and document terms are weighted both with respect to their counts in the query and document, as well as their counts in the collection as a whole. I provide more details on the weight octets later in this document.

To index the sample documents:

```
% bin/j24_index data/english.db data/english.cfg data/english.ucs2
% bin/j24_index data/french.db data/french.cfg data/french.ucs2
% bin/j24_index data/russian.db data/russian.cfg data/russian.ucs2
% bin/j24_index data/chinese.db data/chinese.cfg data/chinese.ucs2
% bin/j24_index data/japanese.db data/japanese.cfg
  data/japanese.ucs2
```

Although we have kept the ucs2 files for each language separate to permit language-specific indexing configuration, as well as file-specific document identifier configuration, it is possible to use only the single configuration file, *common.cfg*, to index the combined ucs2 files. Since ucs2 files, regardless of language, are all in the same character set, they can be concatenated together. *data/common.ucs2* contains the concatenated version of all the other ucs2 files in the data subdirectory. The configuration file can't specify language-specific processing, however, because the stemming algorithm would incorrectly attempt to

stem chinese bigrams with an English stemmer, for example. It does, however, specify all of the possible DOCID start and end values of the combined files.

To index the combined file:

```
% bin/j24_index data/common.db data/common.cfg data/common.ucs2
```

The indexes created for these small files are actually quite large in comparison with the original text files. This is because there is a large amount of overhead for creating these complex indexes. Luckily, the size of the index does not scale linearly with the size of the document set. For large data sets, the index is typically only about 25% of the original data size.

2.2.3 Query and get ranked document lists back using **j24_query**

After indexing the files, you perform queries using **j24_query**. The query and document ranking strategies are user configurable. A detailed discussion of the weighting “octets” is presented in the next section, “URSA/J24 Design Overview” on page 6. The default settings in the configuration files, WEIGHT=AB-AFD-BCA, is well-suited to short queries against medium-length documents. Alternate schemes are known to be better for longer, more complex queries.

The query for **j24_query** can be specified either in a file or on the command line. For command line queries, the query is assumed to be encoded in iso8859-1 and is automatically converted to ucs2. For queries in other languages, prepare the query in the foreign language, then convert it into ucs2 using **uconv** before running **j24_query**.

Sample query files have been provided for Japanese (*data/japanese_query.ucs2*), Russian (*data/russian_query.ucs2*) and Chinese (*data/chinese_query.ucs2*). To submit queries:

```
% bin/j24_query data/english.db data/english.cfg 0 100 "Dividends
for mining companies"

% bin/j24_query data/chinese.db data/chinese.cfg 0 100 -f
data/chinese_query.ucs2

% bin/j24_query data/russian.db data/russian.cfg 0 100 -f
data/russian_query.ucs2
```

Similar queries can be prepared for the French and Japanese files. The output format of **j24_query** is a ranked list of document weight and document number, one per line. The list is in descending order of query-document relevance. Note that the absolute ranges of the weights produced by **j24_query** will vary depending on the weighting scheme.

2.2.4 Get document texts using **j24_docstat**

The next step is to use the document numbers to retrieve specific features of the documents from the indexes. For example, using the Russian sample text and query, document 41 is the most highly ranked document in the set. Use the following command to extract document number 41:

```
% bin/j24_docstat data/russian.db -X -n 41 > data/russian-  
text.ucs2
```

This command puts the ucs2 text from document number 41 into data/russian-text.rus41. Note that the document identifiers begin with 1 in the documents, while the indexing system assigned numbers starting at 0 to the documents for record-keeping purposes.

You can then convert the document text back to koi8 using **uconv**:

```
% bin/uconv -s ucs2 -d koi8 <data/russian-text.ucs2 >data/russian-  
text.koi8
```

j24_docstat can also retrieve the document identifier, document summary line (if any), start and end positions of the document in the file and a variety of document weighting information.

3.0 URSA/J24 Design Overview

The design of the URSA libraries and J24 tools follows a rather generic plan for IR systems, but has enough special optimizations to make careful examination worthwhile. The indexing pipeline, for example, first extracts documents from text using a simplified SGML parser that is tuned to extract text regions, document ids and document summaries (or titles) from document files. Text regions are passed on to the UNICODE tokenization engine. The tokenization engine extracts “words” from the text while handling Chinese, Japanese and Korean (CJK) in special ways, like subdividing Hanzi, Kanji and Hanja sequences into n-grams. The words are inserted into an in-memory bucketed hash table that assigned words unique word numbers (wordno). During tokenization, variants resulting from stemming or other morphological operations are also accumulated in memory. Using English Porter stemming, for example, words ending in “s” will be reduced to their stems. The indexed form of the word is assigned a wordno, but the variant forms are recorded so that all observed variants can later be extracted. The variant information is written to a pair of files that are memory-mapped during the merge phase.

The document is assigned a document number (docno) and a buffer of wordno-to-docno mappings (postings) is maintained in memory. This buffer is periodically sorted by word number and written to a temporary file. The default posting limit is at 500,000 term occurrences. When 500,000 terms have been seen, the posting is written to a temporary file. This results in temporary files of around 2 Mb each. Several hundred of these temporary files can be created for large indexing operations. Because of the complex query and document weighting options, additional information is acquired during this stage. For example, the average individual and average IDF document weights are accumulated and computed during this phase of indexing. The total number of document terms, the number of unique document terms, the largest document term count and the term-document entropies and average entropy are all accumulated at this stage as well. Other information that is recorded at this stage includes byte positions for documents and summary information within files and document identifiers.

3.1 Merging and Finalizing

After indexing the entire document set, the resulting temporary files are merged together and the postings are compressed in the process. In order to perform merge operations, the system limitations on file descriptors must be adequate to open all temporary files simultaneously. The postings for each wordno are then extracted sequentially from each file and merged together. The new posting structure consists of a wordno being mapped to a vector containing docno/count pairs. Full positional information for terms is not recorded, although the tokenizer provides offset information and this may be added in the future to support “windowed” searches. The in-memory hash table of word-wordno associations is written to a memory-mapped external btree that maps words to wordnos and to byte position of the external posting structures. Since word distributions in documents, although not random, are not generally sorted either, a memory-mapped btree provides outstanding performance for rapidly getting wordno and posting position information.

3.2 Posting Compression for Small Indexes

The posting compression scheme uses a fairly simple integer packing scheme in which high-bits in each byte indicate a continuation of the packed integer. This is similar to the utf8 encoding scheme as well as other data encoding formats (Microsoft RIFF, MIDI, etc.). Since almost all documents contain fewer than 128 occurrences of a term, term counts within a document typically are represented by a single byte. Due to term redundancy, the overall posting, lexicon and support file size is approximately 25%. For example the iso8859-1 Financial Times of London collection is 584.2 Mb. The URSA index is 144.6 Mb, or 24.8% of the size of the original text. The index size is only trivially affected by encoding of the original text size, since encoding only impacts the btree for wordno lookup and the storage of document identifiers, so indexing the Financial Times database after converting it to ucs2 results in an index only 13% the size of the original text collection.

3.3 Efficiency Via Asymmetry

An interesting aspect of the URSA indexing and lookup procedures is their asymmetry; software objects for indexing bear little resemblance to software objects for lookup and retrieval. For example, an in-memory bucketed hash table appears to be very efficient for maintaining buffered postings. If start-up time for querying routines could be amortized to a single, initial cost—say by starting a server—then the same hash tables could be efficiently be reloaded for query purposes. For J24, however, it was important to have an extremely fast external representation that could convey the same information as the hash table. A memory-mapped btree serves this role. Again, the construction/query phases are asymmetrical, however. The btree is built from the hash table at the end of the indexing procedure as a large memory heap, then is written to disk. For lookup, the btree is memory mapped to minimize initial I/O costs. Lookup speeds on a Sun Ultra 1 are 25,238 words per second, including I/O times for reading the target words. The btree stores word number and the byte position of the posting for that word in the postings file.

3.4 The RMIT/University of Melbourne Weighting Octet

Queries and documents are weighted according to a user-specified octet of letter codes that correspond to weight assignments for queries and documents based both on internal counts as well as counts of terms in the collection as a whole. The scheme is identical to that presented in Zobel and Moffat’s 1998 paper, “Exploring the Similarity Space” in Communication of the ACM SIGIR, although the URSA scheme is more completely implemented than the specially modified versions of MG used in the Zobel and Moffat paper. The complete set of weighting components in the octet is shown in See Appendix “1” on page 20 and the accompanying explanation in See Appendix “2” on page 21. The J24 tools understand the octet via the WEIGHT variable in the configuration files. See Section 5.1.1 on page 15 for a more detailed description of configuration variables.

4.0 The URSA Query API

In this section, I will show how the query interface can be used in your own C and C++ programs by linking to the included libraries. The query interface consists of objects for submitting queries, getting ranked document sets back and retrieving information about documents, terms and postings. The tools **j24_query**, **j24_lookup** and **j24_docstat** are built around the three components of the query interface.

4.1 Object-Oriented Design

The query interface consists of four C structures, each of which contains both data and pointers to member functions. For consistency and performance reasons, it was decided not to use C++, but the implementation of the query interface closely resembles how a C++ class would be constructed. For example, here is the header file (*include/ursaFD-Lookup.h*) for the object used by **j24_docstat** to get information about documents:

```
#ifndef _URSA_FDLOOKUP_H_
#define _URSA_FDLOOKUP_H_

typedef struct _ursaFDLookup {

    char *path;
    double aveWd_A, aveWd_B, aveWd_C, aveWd_D, aveWd_E, aveWd_F,
        aveWd_G;
    double max_entropy;
    int totalDocuments;

    int dfm_len;
    unsigned char *dfm;

    int dfw_len;
    unsigned char *dfw;

    int dfs_len;
    unsigned char *dfs;
};
```

```

double (*getWd_B)(struct _ursaFDLookup *, int);
int     (*getUniqueTermCount)(struct _ursaFDLookup *, int);
int     (*getTotalTermCount)(struct _ursaFDLookup *, int);
int     (*getMaxTermCount)(struct _ursaFDLookup *, int);
int     (*getStart)(struct _ursaFDLookup *, int);
int     (*getEnd)(struct _ursaFDLookup *, int);
char*   (*getDocid)(struct _ursaFDLookup *, int);
char*   (*getXid)(struct _ursaFDLookup *, int);
string  (*getText)(struct _ursaFDLookup *, int);
int     (*getSummaryStart)(struct _ursaFDLookup *, int);
int     (*getSummaryEnd)(struct _ursaFDLookup *, int);
string  (*getSummary)(struct _ursaFDLookup *, int);

void    (*free)(struct _ursaFDLookup *);

} *ursaFDLookup;

ursaFDLookup map_ursaFDLookup(char *path);

#endif

```

The “object” has type `ursaFDLookup`. `ursaFDLookup` objects are instantiated by the `map_ursaFDLookup(char *path)` function, which takes a database path name and returns an `ursaFDLookup` object. The instantiated object has function pointers that can be used to call member functions. For example, the following piece of code creates and instantiates an `ursaFDLookup` object, then calls its member function to retrieve the document ID of the document with document number 23:

```

char *id;
ursaFDLookup u;
u = map_ursaFDLookup("mydatabase.db");
if(u){
    id = u->getDocid(u, 23);
    if (id) printf("%s\n", id);
    return;
}

```

Note the use of indirection to call the member function `getDocid`. Also note that the `ursaFDLookup` object, `u`, is the first argument to the function call. All function pointers in objects have this same property that they take the object as the first argument. Because structures in C have no “this” operator like C++ to access the internals of the data structure, the object must always be passed to the function.

4.2 Compilation and Makefile considerations

In order to compile programs that use the URSA/J24 API, you must add the `ursa.h` include file (in the `include` subdirectory) to your program’s header. `ursa.h` includes all other necessary header files for compilation. Further, you must add the `include` subdirectory to your compilation flags (`-I include` for gcc Makefiles in the J24 directory).

Three libraries are included in the *lib* subdirectory of the distribution, *libuio.a*, *libCTL.a* and *libursa.a*. These libraries must be available at link time (-L lib for gcc), and linked in using, for example, *-lursa -lCTL -luio* after your target compilation directive in your Makefile. Note also that Berkeley DB 1.85 header files are necessary to compile any program that makes use of this package.

4.3 Query API Definitions, Objects and Functions

4.3.1 Useful Definitions and Convenience Routines

The following type definitions (*include/Defs.h*) are important to consider when using the URSA Query API:

TABLE 1. Important URSA API Types

<i>URSA Type</i>	<i>Type</i>	<i>Description and notes</i>
string	unsigned short *	This represents a ucs2 string for the default libraries (compiled with UNICODE support). Strings can be accessed using a set of convenience functions defined in <i>include/Defs.h</i> that specify replacement functions for standard string manipulation routines in C.
character	unsigned short	This represents a ucs2 character.

The following convenience routines can be used on variables of type string:

TABLE 2. Convenience Routines for strings

<i>Function</i>	<i>Description</i>
int STRLEN(string)	Length of string in characters.
string STRDUP(string)	Returns a copy of the argument.
string STRCPY(string, string)	Copies the second argument into the first and returns the first.
string STRCAT(string, string)	Appends the second arg to the first, returning the first.
int STRCMP(string, string)	Returns a negative value, 0, or a positive value depending on whether the second argument is lexicographically less than, equal to or greater than the first argument.
int STRNCMP(string, string, int)	Like STRCMP, above, but only considers the strings up to the length of the third argument.
string NTS(char *, char *);	Converts a buffer (second arg) from its native codeset (first arg) to a ucs2 string and returns the string. The names of supported codesets for conversion can be determined using: uconv -h , described in Section 2.2.1 on page 3.
char *STN(char *, string);	Converts a string (ucs2 buffer) given by the second argument into a byte array in the encoding specified by the first argument. Returns the new byte buffer.

4.3.2 Query API Objects

The following objects constitute the URSA Query API:

TABLE 3. URSA Query API Objects

<i>Structure</i>	<i>Instantiation Function</i>	<i>Purpose</i>
ursaFDLookup	map_ursaFDLookup(char *path)	Lookup document text, get document identifiers, external identifiers, etc. <i>path</i> is path of database.
ursaFLookup	open_ursaFLookup(char *path)	Lookup postings, get word counts and other posting statistics. <i>path</i> is path of database.
ursaFQuery	new_ursaFQuery(ursaFLookup ul, ursaFDLookup ud, Option opt, string query, int useTrec, char *runid);	Perform queries, apply weighting functions, retrieve ranked document lists. <i>ul</i> and <i>ud</i> must have been previously opened. <i>opt</i> is the Option structure which contains configuration information. <i>useTrec</i> , if 1, parses query and formats output for TREC using <i>runid</i> . Set <i>runid</i> to NULL if not using TREC formatting.

while the following objects play important supporting roles to the primary objects:

TABLE 4. Supporting Objects for URSA Query API

<i>Structure</i>	<i>Instantiation</i>	<i>Purpose</i>
ursaFDocScore	new_ursaFDocScore(int size)	Object returned by submit function of ursaFQuery.
ursaFPost	ReadFPost(char *mem, int size)	Object returned by get function of ursaFLookup, contains all document occurrences of a term.
Option	load_Option(char *path)	Opens the configuration file specified by <i>path</i> , used by ursaFQuery.

4.3.3 Objects and their Functions

This section details the individual member functions accessible in each of the Query API objects as well as the ursaFDocScore and ursaFPost structures which are used to retrieve data from ranked queries and from posting databases. Note that some member functions in the include files are not described here because they represent “private” interfaces to the object that are only used within the object.

4.3.3.1 ursaFDLookup

```
int getUniqueTermCount(struct _ursaFDLookup *u, int n);
    returns the number of unique terms in document n.

int getTotalTermCount(struct _ursaFDLookup *u, int n);
    returns the number of terms seen in document n.

int getMaxTermCount(struct _ursaFDLookup *u, int n);
    returns the count of the term in document n with the highest
    count.

int getStart(struct _ursaFDLookup *u, int n);
    returns the start index (not byte) of document n in its file.

int getEnd(struct _ursaFDLookup *u, int n);
    returns the end index (not byte) of document n in its file.

char* getDocid(struct _ursaFDLookup *u, int n);
    returns the document identifier in iso8859-1 form for
    document n.

char* getXid(struct _ursaFDLookup *u, int n);
    returns the external id (file path) for document n.

string getText(struct _ursaFDLookup *u, int n);
    returns the text for document n. Returns NULL if document n
    doesn't exist.

int getSummaryStart(struct _ursaFDLookup *u, int n);
    returns the index (not byte) of the start of the summary text
    for document n in its file.

int getSummaryEnd(struct _ursaFDLookup *u, int n);
    returns the index (not byte) of the end of the summary text
    for document n in its file.

string getSummary(struct _ursaFDLookup *u, int n);
    returns the ucs2 summary for document n. Returns NULL if
    summaries not indexed or document n doesn't exist.

void free(struct _ursaFDLookup *u);
    frees the object.
```

4.3.3.2 ursaFLookup

```
ursaFPost get(struct _ursaFLookup *u, string s);
```

returns the posting structure for string *s*. NULL if the term doesn't exist.

```
string* getVariants(struct _ursaFLookup *u, string s, int *len);
```

returns an array of length *len* of strings that are the variant forms of *s*.

```
void free(struct _ursaFLookup *u);
```

frees the object.

4.3.3.3 ursaFQuery

```
void setMode(struct _ursaFQuery *u, char *octet);
```

sets the weighting function given in octet format: XX-XXX-XXX.

```
ursaFDocScore submit(struct _ursaFQuery *u);
```

submits the query to the system returning an object that describes the retrieval results.

```
void submitTrec(struct _ursaFQuery *);
```

submits the query as if it were a formatted collection of TREC-style queries.

```
void free(struct _ursaFQuery *);
```

frees the object.

4.3.3.4 ursaFDocScore

```
void free(struct _ursaFDocScore *u);
```

frees the object.

```
void print(struct _ursaFDocScore *u, FILE *f, int low, int high);
```

prints the score and document number, one per line, to file *f* for documents between *low* and *high*.

```
double nthScore(struct _ursaFDocScore *u, int n);
```

score of the ranked document *n*.

```
int nthDocno(struct _ursaFDocScore *u, int n);
```

document number of ranked document *n*;

```
string nthDocText(struct _ursaFDocScore *u, struct _ursaFDocLookup *ul, int n);
```

ucs2 text of document n.

```
string nthDocSummary(struct _ursaFDocScore *u, struct
    _ursaFDLookup *ul, int n);
```

ucs2 summary for document n.

5.0 The J24 Tools

In this section, I present the individual J24 tools and show how they can be used in shell scripts or using control programs written in, say, TCL/TK or PERL.

5.1 j24_index

SYNTAX:

```
j24_index [-vVh] [db path] [config] [file] ... [file]
```

-v makes the indexing process verbose, reporting every document processed.

-V makes the indexing process very verbose, reporting every observed word (mostly for checking configuration against document set).

-h prints this message and exits.

[db path] specifies the database root name

[config file] specifies the configuration file.

[file] specifies a text file.

j24_index indexes ucs2 files (compile-time switches can be used to compile a version for iso8859-1 exclusively, as well) according to the specifications in [config file]. The resulting index files are given a root name [db path]. The index is contained in 7 files as described in Table 5 on page 14.

TABLE 5. Files created by j24_index during indexing.

<i>Filename</i>	<i>Data description</i>
[db path]	Postings
[db path].t	btree of words to posting positions
[db path].v	variant forms for each index term
[db path].vm	map of [db path].v
[db path].dfs	document summary and text byte positions, document identifiers
[db path].dfw	document weight and average weight data
[db path].dfm	map of [db path].dfs

By default, **j24_index** reports as it processes each file and during the merge and finalization stages. the -v and -V switches can be used to increase the amount of reporting the system does during indexing. With -v set, each document processed within each file is also reported. With -V set, each token processed within each file, as well as the case and stemming transformations applied to that token, are reported.

During the indexing process, **j24_index** creates numerous temporary files with numeric extensions ([db path].0, [db path].1, ...). These files contain intermediate postings for the document collection and are merged together in the final step of the process. During the merge step all of the temporary files must be opened by the process to maximize performance. It may be necessary on large indexing jobs to increase the available file descriptor limit on your computer. If you are indexing a large text collection and receive an error similar to the following during the merge stage, you must increase the number of file descriptors available to the shell:

```
% 17:18:06 08/20/1998 [HOST=argos] [FATAL] Number of temporary
      files (124) exceeds available file descriptors (64).
```

Use

```
% limit
```

to determine the available system heap size and descriptors. To increase the limits, use the following for csh:

```
% unlimit descriptors
```

Remove any existing database files and re-run the **j24_index** process.

5.1.1 Configuration Files and Variables

Configuration files for URSA specify variables used for parsing documents, tokenizing text using language-specific techniques and weighting queries and documents. Configuration files are assumed to be in utf8 format. Note that in utf8 format, ASCII characters are encoded exactly like iso8859-1 characters, therefore you can use standard editors to specify separator tags in configuration files. Most SGML markup used in standard text collections is ASCII-encoded.

Table 6 on page 16 lists the variables understood by the J24 tools, acceptable values for the variables. “[val1],[val2]” means the configuration file can contain val1 OR val2. Ranges of acceptable values are indicated using brackets and a hyphen: [1-4] means the value can be between 1 and 4 (inclusive). Spaces and punctuation are significant in configuration files.

TABLE 6. J24 configuration variables, their values and meanings.

<i>Variable</i>	<i>Values</i>	<i>Meaning</i>
LANG	ENGLISH, SPANISH, FRENCH	Languages that specify which suffix stemming algorithm to apply to text.
RULES	[0-1]	Tokenize sequences of tokens between killed words as phrases in addition to as individual tokens.

TABLE 6. J24 configuration variables, their values and meanings.

<i>Variable</i>	<i>Values</i>	<i>Meaning</i>
CJKNGRAMS	[1-4]	Specifies the number of Hanzi (Chinese), Hanja (Korean) or Kanji (Japanese) that should be considered a token. Overlapping sequences are tokenized, so sequence ABCDEF subject to CJKNGRAMS=2 will yield tokens AB, BC, CD, DE, EF.
WEIGHT	[A,B,C,E,F][A-I]-[A-B][A-E][A-N]-[A-B][A-E]A	Weighting octet for specifying query and document weighting schemes. See Section 3.4 on page 8 for more information on the octet weighting scheme.
KILLFILE	path	Specifies a “kill file” that contains token sequences (one per line) that should be ignored by the tokenizer and not indexed or used in queries. Very common words that convey content-specific meaning are often used for kill words (also called “stop” words). The words in the file must be encoded in ucs2.
DOCSTART	val;val;...	Specifies tokens that signify the start of a document within a file. Semi-colons separate multiple acceptable start tokens and the tokens “\n” has special meaning as a literal newline.
DOCEND	val;val;...	Document end tokens.
DOCIDSTART	val;val;...	Document identifier start tokens.
DOCIDEND	val;val;...	Document identifier end tokens.
DOCTEXTSTART	val;val;...	Document text region start tokens.
DOCTEXTEND	val;val;...	Document text region end tokens.
DOCSUMMARYSTART	val;val;...	Document summary (or title data) region start tokens.
DOCSUMMARYEND	val;val;...	Document summary (or title data) region end tokens.
QUERYSTART	val;val;...	Query start tokens (used only by j24_trec).
QUERYEND	val;val;...	Query end tokens (used only by j24_trec).
QUERYIDSTART	val;val;...	Query ID start tokens (used only by j24_trec).
QUERYIDEND	val;val;...	Query ID end tokens (used only by j24_trec).
QUERYTEXTSTART	val;val;...	Query text start tokens (used only by j24_trec).
QUERYTEXTEND	val;val;...	Query text end tokens (used only by j24_trec).

5.2 j24_query

SYNTAX:

```
j24_query [db path] [config file] [low] [high] -f [query
file] | "[query]"
```

[db path] specifies the database root name
[config file] configuration file path.

[low] is rank index of first reported document
[high] is rank index of last reported document (-1 gets them all)
-f [queryfile] optional file that contains the query [query]
optional query.

j24_query queries [db path] using the query specified on the command line [query] or in the file [queryfile]. The query is tokenized based on the LANG variable, CJKNGRAMS setting and RULES setting in [config file], although all query/document start and end tags are ignored for parsing the query. **j24_query** returns a ranked list of document weights (a floating point value) and document number (an integer), one per line, in decreasing order of relevance. [low] and [high] specify the range of documents to report on. Only documents with relevance greater than 0 are reported under any circumstances. Queries given on the command line are assumed to be encoded in iso8859-1 and are converted to ucs2 prior to processing. Use the -f option to process ucs2 queries directly.

5.3 j24_docstat

SYNTAX:

```
j24_docstat [db path] -BCDEFGh -setXmSIdxTWU -n [docno]
```

[db path] is always required and specifies the database root name; must occur first.
-B..G gets the average weight of all documents using variants B..G.
-h prints this message and exits.
-n [docno] specifies the document number for other operations
-W gets the weight of [docno].
-U gets the unique terms of [docno].
-T gets the total indexed terms of [docno].
-s gets start index for [docno].
-I gets summary text (converts to iso8859-1) for [docno].
-S gets summary text (ucs2/iso8859-1 depending on compilation) for [docno].
-e gets end index for [docno].
-m gets the maximum term count in [docno].
-t gets text (converts to iso8859-1) for [docno].
-X gets text (usc2/iso8859-1 depending on compilation) for [docno].
-d gets DOCID for [docno].
-x gets external docid (path) for [docno].

j24_docstat retrieves information about a specific indexed document or about the collection of documents as a whole. The -B through -G options query the average document weights calculated for the collection as a whole and correspond to the RMIT/University of Melbourne document weighting codes (field 5: XX-XXD-XXX). These switches are mainly for diagnostic purposes. For the included tool set (compiled for UNICODE), the -X and -S options will produce ucs2 text). The -s and -e switches return where the document is located (in ucs2 characters) in the external file. If you are making use of the -s and -e fields to directly grab text from the file, remember that the byte positions will be twice

these values since ucs2 characters are 16 bits each. Use the `-x` switch to get the path to the external file containing the document.

5.4 j24_lookup

SYNTAX:

```
j24_lookup [db path] -mUT -e [term] -t [term] -V [term] -n [term] -S [term] -Q [term]
```

[db path] is always required and specifies the database root name; must occur first.

-S [term] searches for the term and prints documents and term counts in documents for that term.

-Q [term] (same as S, but doesn't report results)

-P [term] prints the term position for [term]

-V [term] prints the term variants for [term]

-m prints the maximum term-document co-occurrence, fm

-U prints the total unique words indexed in this collection

-T prints the total words indexed in this collection

-e prints the term entropy (RMIT calls this "noise", n_t).

-t prints the total number of this term's occurrences.

-h prints this message and exits.

j24_lookup retrieves information about words and postings. Of primary interest is the posting retrieval option, `-S`, that gets the complete postings for the supplied term, printing the document numbers and the counts of the term within the document, one per line. Note that unlike **j24_query** which will perform language-specific query processing depending on the configuration variables `LANG`, `CJKNGRAMS` and `RULES`, no query processing is done for the `-S`, `-Q`, `-C` and `-P` options. The term variants produced by `-V` are the list of stem variants produced by the English, Spanish and French stemmers. For the sample English text file, *data/english.iso8859-1*, after you convert it to ucs2 and index it using the English configuration file, *data/english.cfg*, **j24_lookup** will produce the following output for the stem "manag":

```
% j24_lookup ../data/english.db -V manag
Variant[0]=management
Variant[1]=manager
Variant[2]=managing
```

5.5 j24_trec

SYNTAX:

```
j24_trec [db path] [config file] [run id] [query file]
```

[db path] specifies the database root name

[config file] configuration file path.

[query file] is the query set.

j24_trec is used to run multiple queries contained in a file against a database for evaluation of standard queries from the National Institute of Standards and Technology (NIST)

in their Text Retrieval Evaluation Conference (TREC) series of evaluation summits. The QUERY... configuration variables detailed in Table 6 on page 16 specify how [query file] should be parsed to extract queries and query identifiers. The output of **j24_trec** is suitable for the evaluation programs available from NIST for evaluating TREC queries with known relevance judgments.

Appendix 1. URSA Implementation of the RMIT/University of Melbourne Octets: A weighting scheme is specified by an 8-character string XX-XXX-XXX. The first pair specifies the combining function and collection term weight strategies. The next triple specifies how terms in documents are weighted, and how document statistics normalize those weights. The final triple specifies the same information for queries.

Collection Functions				Document Weighting Functions						Query Weighting Functions					
Combining Functions $S_{q,d}$		Collection term weight w_t		Document-term weight $w_{d,t}$		Relative term-doc frequency $r_{d,t}$		Document weight W_d		Query term weight $w_{q,t}$		Relative term-query weight $r_{q,t}$		Query weight W_q	
A	$\sum_{t \in Y_{q,d}} (w_{q,t} \cdot w_{d,t})$	A	1	A	$r_{d,t}$	A	1	A	1	A	$r_{q,t}$	A	1	A	1
B	$\frac{\sum_{t \in Y_{q,d}} (w_{q,t} \cdot w_{d,t})}{W_q \cdot W_d}$	B	$\ln\left(1 + \frac{N}{f_t}\right)$	B	$w_t \cdot r_{d,t}$	B	$f_{d,t}$	B	$\sqrt{\sum_{t \in Y_d} w_{d,t}^2}$	B	$w_t \cdot r_{q,t}$	B	$f_{q,t}$		
C	$\sum_{t \in Y_{q,d}} (C + w_t)$	C	$\frac{1}{f_t}$			C	$1 + \ln(f_{d,t})$	C	$ Y_d $			C	$1 + \ln(f_{q,t})$		
E	$\sum_{t \in Y_{q,d}} \frac{w_{d,t}}{W_d}$	D	$\ln\left(1 + \frac{f_t^m}{f_t}\right)$			D	$\frac{f_{d,t}}{f_d^m}$	D	$\sqrt{ Y_d }$			D	$\frac{f_{q,t}}{f_q^m}$		
F	$2 \left[\frac{\sum_{t \in Y_{q,d}} (w_{q,t} \cdot w_{d,t})}{W_q^2 + W_d^2} \right]$	E	$\ln\left(\frac{N - f_t}{f_t}\right)$			E	$0.5 + 0.5 \frac{f_{d,t}}{f_d^m}$	E	$\log_2 Y_d $			E	$0.5 + 0.5 \frac{f_{q,t}}{f_q^m}$		
		F	s_t			F	$\frac{f_{d,t}}{f_{d,t} + W_d / \left(\frac{ave}{d \in \Delta} W_d\right)}$	F	f_d						
		G	s_t					G	$\sqrt{f_d}$						
		H	$\max_{t' \in T} (n_{t'}) - n_t$					H	1						
		I	$1 - \frac{n_t}{\log_2(N)}$					I	$0.3 + 0.7 \frac{\sqrt{\sum_{t \in Y_d} w_{d,t}^2}}{\left(\frac{ave}{d \in \Delta} \sqrt{\sum_{t \in Y_d} w_{d,t}^2}\right)}$						
								J	$0.3 + 0.7 \left(Y_d / \left(\frac{ave}{d \in \Delta} Y_d \right) \right)$						

Appendix 1. URSA Implementation of the RMIT/University of Melbourne Octets: A weighting scheme is specified by an 8-character string XX-XXX-XXX. The first pair specifies the combining function and collection term weight strategies. The next triple specifies how terms in documents are weighted, and how document statistics normalize those weights. The final triple specifies the same information for queries.

Collection Functions				Document Weighting Functions				Query Weighting Functions							
Combining Functions $S_{q,d}$		Collection term weight w_t		Document-term weight $w_{d,t}$		Relative term-doc frequency $r_{d,t}$		Document weight W_d		Query term weight $w_{q,t}$		Relative term-query weight $r_{q,t}$		Query weight W_q	
								K	$0.3 + 0.7 \left(\sqrt{ \Upsilon_d } / \sqrt{\text{ave}_{d \in \Delta} \Upsilon_d } \right)$						
								L	$0.3 + 0.7 \left(\log_2 \Upsilon_d / \text{ave}_{d \in \Delta} \log_2 \Upsilon_d \right)$						
								M	$0.3 + 0.7 \left(f_{d,t} / \text{ave}_{d \in \Delta} f_d \right)$						
								N	$0.3 + 0.7 \left(\sqrt{f_{d,t}} / \sqrt{\text{ave}_{d \in \Delta} f_d} \right)$						

Appendix 2. Key to the symbols used in Table 1 on page 20.

Symbol	Meaning	Symbol	Meaning
N	Number of documents in collection	f_d	
n	Total number of terms in collection	Υ	The set of distinct terms in the database.
$f_{d,t}$	Count of term t in document d.	Υ_d	The set of distinct terms in document d
F_t	Collection count of term t.	Υ_q	The set of distinct terms in query q
f_t	Number of documents containing term t.	$\Upsilon_{q,d} = \Upsilon_q \cap \Upsilon_d$	The distinct terms shared between the document d and query q .
$f_{q,t}$	Count of term t in query q.	$n_t = \sum \left(-\frac{f_{d,t}}{F_t} \log_2 \frac{f_{d,t}}{F_t} \right)$	“Noise”
f_m	The largest of f_t	$s_t = \log_2 (F_t - n_t)$	“Signal”
f_d^m	The largest $f_{d,t}$	Δ	The set of documents.