CPSC 67 Lab #2: Boolean Retrieval

Due Wednesday, Feb. 4 (11:59pm)

The goal of this lab is to perform boolean retrieval on the document collection you built in Lab #1. You will support single token queries, phrase queries, and queries involving the AND, OR, and NEAR operators.

To support this, you will need to use the SQL schema discussed in class which is posted on the wiki. Assuming you wrote Lab #1 to specifications, a Python program can be found on the wiki that will convert your directory structure and database from that lab into the directory structure and database that we will use in this lab. See the note on this program for warnings and caveats. Even if you choose not to use the conversion script, be sure to follow the new directory structure which requires putting all downloaded pages (previously separated into `raw/` and `searched/` directories) into only `raw/`. Finally, an API that makes use of this schema is available for download from the wiki. This API should cut down on errors in your code and avoid tedium in writing that portion of the assignment.

You will also download an additional 20 artists from a different genre and you maintain the mapping between artists and genres. See the wiki for the list of 20 additional artists.

1 Database Support

A significant portion of the lab involves converting your code from Lab #1 to use the new, expanded database representation. As we discussed in class, you need to make sure you understand the schema and how to use it before you can make headway on the assignment. Be sure you read through the schema and ask questions if you get stuck. You should make extensive use of MusicDB class, as discussed in class.

2 Boolean Queries

To support Boolean queries, you will first build an inverted index out of all the pages in stored in the SearchToCacheURL table. Then, you will use Boolean query techniques to support five types of queries:

- Single token queries: `alternative`
- Queries using AND: `classic` AND `rock`
- Queries using OR: `jazz` OR `fusion`
- Phrasal queries: `"distorted guitar"`
- Queries using NEAR: `funk` NEAR `soul`

To support NEAR queries and phrasal queries, you will need to create a positional inverted index. The textbook (Section 1.3 and 2.4) contains implementation-level details on how to process queries.

In theory, we could support combinations of the above queries, e.g. `"Depeche Mode" music`. The implementation of multi-word queries that involve phrasal and NEAR queries is more messy than it is informative; conversely, supporting multi-word AND and OR queries is actually quite easy. Therefore, we will support only single word queries or queries with two tokens such as those shown above

3 API

The API used in Lab #1 will change slightly to accommodate the extended database schema. Only changes to the previous API are mentioned in the comments. The Spider API will now be as follows:

```
class Spider(object):
    def __init__(self, cachedir, db, cookiejar=None):
        """
        Since we have folded the cached table and searched table into a
        single CacheURL table, there is no need to pass the tablename
        in as a parameter.  In addition, the db is no longer the name
        of the database, it is now an instance of the MusicDB class
        (whose definition can be found on the wiki).
        """

    def fetch(self, url, usecache=True, offline=False):
        """
        The parameters are the same, but while this previously returned
        (header, page), this will now return (cache_url_id, header, page)
        where cache_url_id is the id in the CacheURL table assigned to
        fetched page.
        """

    def cacheID(self, url):
        """
        This is unchanged.
        """

    def getDB(self):
        """
        New method returning a pointer to the MusicDB database.
        """
```

Your search engine, here Google, will have the following API:

```
class Google(object):
    def __init__(self, spider):
        """
        The constructor the same parameters as before.
        """

    def search(self, query, hits=10):
        """
        The same parameter set, but whereas this method used to return
        the links obtained by the search, it now returns a tuple of
        (search_id, links) where search_id is the id of the search as
        entered into the Search table; links is unchanged.
        """

    def artist_search(self, query, artist_id, hits=50):
        """
        A new method.  The parameter artist_id is the id of the artist
        as inserted into the Artist table.  Makes appropriate entries
        in the database and calls the search() method.
        """

    def fetch(self, url, search_id, rank):
        """
        A new method.  The Spider is used to fetch the url.  Then, the
        search_id and rank, along with the cache_url_id returned from the
        Spider fetch, are used to fill in the SearchToCacheURL table.
        """
```

The MusicDB API is quite extensive and is not included in this document. To see the full API (which you don't have to write, you just have to use), `import musicdb` into Python and run `help(musicdb.MusicDB)`.

Finally, you should create a class called `InvertedIndex` with the following API:

```
class InvertedIndex(object):
    def __init__(self):
        """
        Create an empty (positional) inverted index.
        """

    def insertDocument(self, wordlist, cache_url_id):
        """
        Insert a document into the inverted index.  The document is
        represented as a list of words.  The cache_url_id parameter is
        the id of the page in the CacheURL table.
        """

    def tokenQuery(self, token):
        """
        Returns a set of document ids matching the token; returns an
        empty set if the token is not found in any document in the
        collection.
        """

    def andQuery(self, token1, token2):
        """
        Returns a set of document ids matching two tokens; returns an
        empty set if pair of tokens are found in any document in the
        collection.
        """

    def orQuery(self, token1, token2):
        """
        Returns a set of document ids matching at least one of two
        tokens; returns an empty set if neither token is found in any
        document in the collection.
        """

    def phraseQuery(self, token1, token2):
        """
        Returns a set of document ids matching the phrase 'token1
        token2'; Returns an empty set if the phrase is not found in
        any document in the collection.
        """

    def nearQuery(self, token1, token2, nearness=3):
        """
        Returns a set of document ids containing token1 near token2
        where nearness is defined as +/- nearness; Returns an empty
        set if this not found in any document in the collection.
        """
```

4 Results

Once you have successfully downloaded 50 pages each for the 40 different artists, cached them, cleaned them, and made all appropriate entries into the database, you will perform the following (case insensitive) queries, then answer the question listed below the queries.

- `alternative`
- `classic`
- `rock`
- `acoustic` AND `guitar`
- `"acoustic guitar"`
- `"electric guitar"`
- `indie` OR `college`
- `seattle` NEAR `based` (defined as ± 3 words)

Now, for each query, answer the following question:

1. For each artist, how many pages that you have stored for this artist are returned by each of the above queries. Report the artist with the highest number of matching pages. For example, if Depeche Mode matched 32 pages for a particular query, and every other artist matched less than 32 pages for the same query, you would report Depeche Mode (32). Enter these values on the wiki. In the event of a tie, you can arbitrarily choose one artist, but you should report the number of items that are tied with the same score, e.g. Eric Clapton (2) [5 tied].

Report your results on the wiki as described in class.