Mini Project Report

Introduction

The mini-project consists of a search engine implementation on the Wikipedia data dump. By creating a suitable indexing structure, the user can search plain text or add field queries, and retrieve 10 relevant pages for the searched query.

Directory Structure

- **Dataset:** This folder consists of the data dumps for each mini project phase.
- **Index:** This folder is dynamically created during runtime and stores the index files, title files, and index metadata (further explained in the index structure section)
- **Docs:** This folder contains the queries to be loaded, along with any relevant project docs.
- **Src:** Contains the source code of the engine.
    - indexer.py: Control file called by index.sh to parse the data and merge the index files.
    - contentHandler.py: Parser file that contrails the sax parser control class. This class controls the reading of the data dump, parsing and extraction of relevant tokens, creation of inverted index, and initial storage into files.
    - merger.py: Contains the merger class, which inputs the temporary index files created by the parser, and merges tokens across files to a master index file. This file is then split into sub-files.
    - searcher.py: Contains the code for the entire search algorithm of the engine, including query loading, splitting, index loading, and score generation.
    - index_splitter.py: File containing auxiliary code to manipulate index files for testing/dev only.
    - utils.py: File containing common functions and global variables
- **index.sh:** Control script to create index
- **stats.txt:** Output file for index statistics

Index Creation

To create the index used in the search engine, firstly the XML sax parser is used to iterate through the data dump line by line. By using relevant XML tags, we are able to isolate pages, and regex filters are used to extract different sections from within the page content. To clean the tokens that are obtained, some common punctuation text and characters are removed. After this step, stopwords and case folding is done for the text, and it is stemmed appropriately.

The inverted index for this data is updated, with a dictionary and counter made for each different section of the page. This inverted index maps the token word to a string, that encodes the term frequency of the word across the page, as

```
# Format
<token>:<pageId1>t<title_freq>i<infobox_freq>b<body_freq>c<category_freq>r<ref_freq>l<extern_freq>|<pageId2>...

# Example
word:134t4c2b23|200t1r4

# This example shows that 'word' is in documents 134 and 200, with its frequency in each category recorded
```

Thus, with this indexing strategy, we are able to record the presence of the term in each document, pertaining to each category. By splitting with separator '|', we are able to obtain document frequency quickly as well.

This index is stored in temporary files while the parser is iterating. As the parser is loading a 90 GB file, the inverted index can't be stored in memory. Hence, after a fixed number of pages (50,000 in this implementation), the index is dumped to a temporary file. Concurrently, the title of each page corresponds to the page id, and is dumped at every 50,000 iterations as well. This can be quickly loaded during search, with the index id being the quotient of pageId / 50,000 and its position within the index being the remainder.

As the temporary indices will contain the same token spread across different files, it is important to merge these files. This is done iteratively in this case, with the number of merge iterations calculated as the nearest power of 2 higher than the file count (practically, 9 merge iterations were required). In each iteration, two files are merged together, which is repeated till one master index is created. However, this index file is too large to be loaded efficiently within the search code, and thus it is split into smaller chunks, with each chunk containing a part of the lexicographically-sorted index. To quickly access the correct file when searching, a secondary index is made (stored as word_manager), that keeps track of the last word in each file. As a result, we can binary search across this index and quickly find the correct index file to load the token from.

Index Stats

For the 90 GB dataset, the following stats were recorded:

```
Index Size: 22.9 GB

Index Files: 602

XML Pages Parsed: 22,286,847

Tokens while cleaning: 8,145,285,457

Inverted Index Tokens: 60,249,488

Index Generation Time: 26327.97 Seconds
# Here, parsing was 19246.13 seconds, approximately 43 seconds per 50,000 files
```

To get this, the parameters were:

```
Max Page Break (split index/title while parsing): 50,000

Index Chunk Value: 100,000
```

Code Optimizations & Limitations

To optimize the code for indexing in terms of both time and space complexity, it was necessary to split the index across chunks and merge them later. An alternative heuristic, wherein the index every 50,000 lines would be directly appended to the master index wasn't optimal in terms of runtime. The time spent on merging files allows the engine to be completely scalable in terms of data size.

To decrease runtime for each page while parsing, optimizations were done when tokenizing and stemming the page data. Using an optimal python list split and join method, along with pyStemmer (snowball stemmer) instead of porter stemmer, was able to reduce runtime per 50,000 iterations by almost 10 seconds. In addition to this, regex searches were more optimal as compared to character/string matching, and page data was continually operated and passed across sections, reducing redundancy of operations and length of string to be regex searched.

While merging index files, the number of merges required are pre-determined and balanced to discourage imbalanced file size merges. The merge operation, in addition, is done line-by-line, allowing the indices to be merged even for higher orders quickly.

For searching, the secondary index created allows us to load in the required index token from its appropriate file in O(logN) time, where N is the number of index files, thus preventing any redundant file accesses. The title of the search results is also loaded in O(1) time, as both the file and its position within the file is loaded instantly.

A key point for the search engine is that Unicode letters besides the English alphabet are being permitted as index tokens. This is mostly to allow the index to accommodate queries across languages, as non-English letters can also be stored and appropriately queried in this index. This is required for the bonus section of the project and allows the engine to be used for different characters as well. However, this does increase the index size which can be reduced to a much smaller amount. In addition, by allowing up to 15 characters per token, we are enabling a few irrelevant numeric values to be stored.

The major limitation of the engine is that the tokens don't store solely the relevant documents of the index. This increases search time for very common tokens, as each document id is individually parsed, and count extracted from it.