Information Retrieval Course
CSE 435/535 - Fall 2010
Project 2
Document Indexer in C++

Project Description In this project we will be indexing a collection of roughly 100,000 documents belonging to the news and wikipedia category. This collection has been introduced to you in project 1.

Due Date **Online submission through UBLearns' Digital Dropbox by 6^{th} November, 2010 11:59 P.M.**.

1 Indexing

1.1 Index creation and simulating writing to inverted barrels/shards

This project will include building a forward index and an inverted index for a larger document collection. The data structures in this project should support the Vector Space Model; this means that your indexing module should be capable of calculating tf×idf weights for all the terms in the collection.

For large scale engines, the index runs into GBs/TBs of documents, and hence their indexing process is both memory and disk based. Thus, the design of the Indexer should be such that it could easily be extended for querying such large data sets.

The idea is to distribute the forward index and the inverted index over a number of files instead of keeping it in one file. This can be implemented in the following manner:

TODO
File Dictionary and Forward indices:

- Change the datastructure of the file dictionary so that you are not holding the <fileID filename> as a key value pair in memory since filenames tend to be very long. Instead write **all** filenames as a continuous '$' separated string in some text file. The <fileID filename> as a key value pair in memory is thus changed to <fileID byte-offset-of-the-first-alphabet-of-the-filename-in-file>
- Store the classification label of the document in the forward index - 0 for "news" documents and 1 for "wiki".
  If the document is a wiki document:
  1. Add the document ID of the smaller "semwiki" document. This should indicate the actual filename of the semwiki document from where information must be read in project 3.
  2. Add the terms from the corresponding smaller "semwiki" document in the forward postings list of the wiki document
  3. This postings list will consist of term IDs and each term ID should know from which part (i.e. Infobox, Title, Link, etc.) of the semwiki document it came from and its position (either byte wise or token wise)

If the document is a "news" document, do not store any postings list at this point.

- Add entries for document-term normalization statistics in the forward index entries for all kinds of documents
- Flush the forward index to disk in binary format whenever you have processed 10,000 files. The forward index barrels/shards should be named as "index.for.x" where "x" is the shard number $\in \{000, 001, 002, ...\}$.
  →**NOTE**: since you are processing documents in sequential order, you should be able to maintain a range of document IDs that exist in a shard and hence figuring out which shard contains a particular document ID becomes easy

Inverted indices:
While all documents have been processed, do steps 1 and 2:

- Step 1: Keep accumulating the termID and docID pairs in memory using an efficient inverted index data structure until the total length of the postings lists reach 50,000. Then do the following:
  - Step 1a: Write this inverted index to the disk by splitting into separate binary files (barrels/shards) - elements of the index whose words start with alphabets "a-f" will be in one file, those starting with "g-k" will be in second file, and so on.
    →**NOTE**: In addition, meta information about each index file will be stored in another structure, which will always stay in the memory during indexing and retrieval. The metaindex can look like:
    a-f → index.inv.001
    g-k → index.inv.002
    l-p → index.inv.003 and so on. While iterating over the document collection, the shards of the inverted indices are thus created as:
    a-f → index.inv.001.y
    g-k → index.inv.002.y
    l-p → index.inv.003.y and so on. "y" is the block number i.e., for the first run of 50,000 postings entries, $y = 0$, for the second run, $y = 1$ and so on. Basically you have the "inverted index config file" file on disk that has information about this metaindex and you can tweak this file before indexing to control the number of partitions.
- Step 2: Reset the inverted index structure and go back to step 1.
- Step 3: **Optimization Step**:
  - Merge each block of records index.inv.x.y **for all** values of y and **for each** $x \in \{000, 001, 002, ...\}$ as follows:
    1. For each x create a binary file named "index.inv.x.pstngs".
    2. For each termID $n_{x.y}$ in $x.y$, retrieve the postings lists, merge the postings lists and write the final sorted postings list for $n_{x.y}$ in index.inv.x.pstngs. By doing this, the inverted index file "index.inv.x" will now contain <termID byte-offset-of-the-postings-list-in-index.inv.x.pstngs>
       →**NOTE**: The postings list can be sorted by any kind of consistent document ordering (as was discussed in class).
    3. Programatically delete the intermediate index files like "index.inv.001.y"

Both the forward and the inverted indices should be saved as binary files to save space. To view binary files one can use the software called TinyHexer[1] or any other binary file editor as can be found at `http://ehep.sourceforge.net/`. Please design your index structures in such a way so that you can easily add element level features in the keys or values without major changes in code.

Resources from Project 1:

- Stop words list and a Stemming Algorithm (Porter stemmer)

[1] http://www.softpedia.com/get/Others/Miscellaneous/tiny-hexer.shtml

- Cleaned news files from the TREC document collection and the Wiki markup files from Project 1
- Some code snippets from project 1 that you can plug and play

Data:
The entire data will be available at /projects/TREC/CSE535_Fall2010 folder on CSE machines.

2 Strict submission guidelines

- **Follow coding styles on using usernames in methods from project 1.**
- You are constrained to create not more than 100 barrels/shards. NOTE: you could also have barrels/shards like a→index.inv.001, b→index.inv.002 etc. However, the more barrels/shards you have, the more fragmented your filesystem will be. [Hint: Some query terms have a longer postings list than others]. You could use a count dictionary to obtain the statistics. Document this in your project report. **Your metafile will be tweaked based on your documented format during the day of the demo.**
- The command line interface to your program is as follows:
  $ > **dodo** [-i | −−index] <path_to_input_data_dir> <path_to_index_store_dir> <path_to_dict_store_dir> <inverted_index_config_filename>
  the "[-i | −−index]" is the switch that says the program should be in the indexing mode. Note that the switch string is either "-i" or "−−index" (lowercase) using standard unix command line conventions. The name of the executable must be **dodo**
- Your program should be cleaned of irrelevant outputs (couts) except the required ones in the final version that you submit
- The **name of your tarball submission** in Digital Dropbox should be <your_team_name_Project2> while the **actual filename** should be <your_team_name_Project2>.[tar | zip]

NOTE: If your program fails to compile or run, the only way we can grade your project partially is by looking at your project documentation

2.1 Strict submission checklist

- Must submit makefile (0 points for not being able to run make)
- Dictionary for file ids/offsets [20 points]
- Breaking up forward index; program not crashing; program deleting the intermediate *.y postings files [30 points]
- Total execution time [20 points even if bad, added from there. Total 30 points]
  Note:
  1. **The program must clearly write to console**: *"Indexing"* before the indexing begins and *"Optimizing"* before the optimization begins. The time for execution will involve the total running time of the executable. Some teams who are using boost are finding an increased time of 12 mins while parsing the wiki files. If you are using boost please send me an email so that I can adjust your executable's time.
- Forward index entries that contain SemWiki information: **Output to console** a sample forward index entry for **only one** wiki document. Do not output more than 5 entries of the forward postings list for such a wiki document. Your program should write to console *"Sample forward index entry for only one wiki document"* before the dump. You are free to output this at any point in time of your program execution. **Do not output more than once**. [10 points]
- Team work. [10 points] **Note:**

1. If you feel that somebody is not contributing enough and making your life difficult, please do email the TA so that it can be noted against the team's record.

3 Submission

Submit a tar or zip file named <your_team_name_Project2-final>.tar or <your_team_name_Project2-final>.zip using the Digital Dropbox in UBLearns. The contents of the compressed file is as follows:

−− <your_team_name_Project2-final>.tar
|+ All source and make files
|+ A pdf generated from Doxygen (Not required in project2 but will be needed in project3)
|+ A README text file similar to the one submitted for the first project. It should include the instructions to compile your code, the input arguments to be provided to your main program, etc.

Standard Message
Make sure all your programs at least compile and run on at at least one CSE machine: (e.g. metallica.cse.buffalo.edu [running Linux]) For evaluation, the TAs will download your most recent tar files (upto the submission date) with the name <your_team_name_Project2-final>.tar from UBLearns and run it. NOTE: There might not be a demo for project 2.

DO NOT TAR THE INDEX FILES AND SUBMIT THEM ALONG WITH THE SOURCE FILES. UBLEARNS CANNOT ACCEPT LARGE FILES. ALSO DO NOT EMAIL US YOUR TAR BALLED PROJECT - THEY WILL BE DELETED. WE ARE VERY STRICT ABOUT DEADLINES THIS TIME - LATE SUBMISSION BY 24 HOURS DEDUCTS 10 POINTS, BY TWO DAYS DEDUCTS 25 POINTS, BY THREE DAYS DEDUCTS 50 POINTS AND DONOT SUBMIT AFTER 3 DAYS