CS-GY 6193 **(Due: 10/14/21)**

Inverted Index Construction

Instructor: Torsten Suel *Name:* Cheng Chen, *Netid:* cc6858

1 Overview

The program can build an index for the document collection from TREC 2020 Deep Learning Track based basically on Merging Subindexes Algorithm. The collection is of size around `22G` with `3.2M(3,213,835)` records. This program will finally produce

(a) A page table, a structure that, given a docID, returns the URL of the page and the size (number of terms) of the page.

(b) A lexicon, a structure that, given a term, returns the start of the corresponding inverted list in the index, the end of the list, and the number of docs containing the term.

(c) The inverted index structure, which stores the actual list for each term. Each list is composed of chunks of 128 docIDs and term frequencies, along with metadata, i.e., arrays with last docIDs and sizes of chunks, at the beginning of the list.

2 Pipeline and Usage

The program consists of four sources files which should be compiled and run in four stages indicated by four different source `c++` files.

(a) Stage 1: *parse.cpp*

- This stage will produce `523` intermediate subindexes named from `index_n1` to `index_n523`. Each subindex is responsible for $3,213,835/523 = 6,145$ documents.
- The inverted lists are arranged alphabetically. The `docIDs` that contain the term are also increasingly sorted. The properties hold throughout all stages.
- This stage will also produce the **page table** named `page_table.bin`, assigning `docIDs` in the order the pages are parsed.

```
make parser
./parser 6145
```

(b) Stage 2: *phase*1*merge.cpp*

- This stage will merge the `523` intermediate subindexes into `21` larger intermediate subindexes named from `merge_n1` to `merge_n21`.

```
make merger
./phase1merger
```

(Only need to compile once for stage `2 & 3`.)

(c) Stage 3: *phase*2*merge.cpp*

- This stage will merge the `21` intermediate subindexes into the final inverted index named `merge_nfinal`, which will be converted to compressed binary format in the next stage.
- The total number of indexable terms is `41,085,728`.

```
(make merger)
./phase2merger
```

(d) Stage 4: *compress.cpp*

- This stage will scan the index file and produce the **compressed inverted index** structure named `index_nfinal.bin` in binary format using `var-byte` method, along with the **lexicon** structure named `lexicon`.

```
make compresser
./compresser
```

3 Details

This section explains the key functions/components/design decisions.

(a) Stage 1: *parse.cpp*

- I use `map<string, vector<uint32_t> > inverted_index` to organize the subindexes. This structure maps a token string to a vector of `docIDs` containing it. When writing to the index file, I use another `map<int, int> df` to parse the vector of `docIDs`, i.e., counting and mapping `docIDs` to its frequences. Subindexes in this phase are lines of

```
term docID,docID,docID,... freq,freq,freq,...
```

- I do not use `termID`.
- The `map` structure of `c++` is internally sorted. The lists are read alphabetically and the `docIDs` are read increasingly from the structure. This structure gets me rid of using Unix `sort`, though might also slow the program down.
- I only consider tokens that only consists of english letter and numbers with the help of `isalnum()`. I also limit the token length to 30 for convenience. However, actually the tokens get filtered by these limitations are not much.
- I use `strtok` to extract tokens from the web page content between `<TEXT>` and `<\TEXT>`. Using a dedicated library might be faster.
- I align the URLs to `1020` bytes which is larger than the maximum URL length. Together with the `4` byte page size, each page table entry consists of `1024` bytes. Therefore the total size for the page table is $1024 \times 3,213,835 = 3,290,967,040$.
- URL byte alignment might help query processing. Given a `docID`, I can multiply it by 1024, seek to this offset on file and read 1024 bytes to get url and page size. Then we might not need to load the whole page table into memory, though the time to access could be slower. Another disadvantage is the large page table has so many waste null bytes.

```
char page_url[NUM][1020];
uint32_t page_size[NUM];
```

(b) Stage 2: *phase1merge.cpp*

- From stage 1 I will get `523` subindexes, which is not efficient/possible to merge at once. I tried and found that I can only open at most `252 fstream` at the same time. And it is quite slow to merge this many files. Therefore I decide to merge in 2 passes: `523` $\rightarrow$ `21` $\rightarrow$ `1`.
- I use the `priority queue` structure provided by `c++` to function as min-heap to merge the lists. I construct the inverted list structure as follows.

```
/** structure in the priority queue **/
class elem {
   uint32_t fid;     /** 0,1,2,3, ..., 522 **/
   string token;     /** term **/
   string didlist;   /** docID list **/
   string freqlist; /** frequence list **/
public:
   /** Constructor omitted **/
   uint32_t getID() const {return fid;}
   string  getT() const {return token;}
   string  getDL() const {return didlist;}
   string  getFL() const {return freqlist;}
};
```

- I overload the comparison operator to order the inverted lists, keep them sorted. A list with smaller `token` or `fid` will have higher priority and be popped out first. (This is because the `docIDs` are assigned in the order the pages are parsed, subindexes with larger `fids` will contain larger `docIDs`.)

```
class comp {
public:
    int operator() (const elem* e1, const elem* e2) {
      if (e1->getT() > e2->getT())
         return true;
      else if (e1->getT() == e2->getT())
         return e1->getID() > e2->getID();
      return false;
    }
};
```

- If two inverted lists have the same `token` meaning they should be merged, just simply concatenate their `didlist` and `freqlist` since the first popped list contains smaller `docIDs`.
- The `fid` has another usage, i.e., a new inverted list will be pushed into the `priority queue` from the same `fid` as the popped list until this subindex is reading up.
- The intermediate subindex files are organized as blocks of three lines:

```
term
docID,docID,docID,...
freq,freq,freq,...
```

(c) Stage 3: *phase2merge.cpp*

- Same as stage 2 except for the parameters, e.g., file name, file number.
- The final inverted index contains `123,257,184` lines indicating `41,085,728` indexable terms/inverted lists.

(d) Stage 4: *compress.cpp*

- I implement the `var-byte` method as

```
vector<uint8_t> buffer;

void VBEncode(uint32_t num){
   while (num > 127) {
      buffer.push_back(num & 127);
      num >>= 7;
   }
   buffer.push_back(num + 128);
}
```

The buffer is needed to calculate the metadata, e.g., chunk sizes and last `docIDs`.

- I use `128 docIDs` as a chunk. Terms with less than `128 docIDs` will have one chunk.
- During traverse of the `didlist`, I **take differences** of `docIDs`, and then encode those smaller numbers. At the same time, I record the chunk size and last `docID` of each chunk.
- The index layout format is

```
METADATA 128docIDs 128freqs 128docIDs 128freqs ... METADATA ...
```

- I use `uint64_t` to record the start byte offset of the corresponding inverted list in the index and the end of the list in case the `lexicon` size becomes very large. The `lexicon` are lines of

```
term start_of_list end_of_list num_of_docs
```

4 Time/Space and Future Direction

This section explains how long it takes on the provided data set, and how large the resulting index files as well as the intermediate files are. I use `std::chrono library` to record the time.

(a) Stage 1: *parse.cpp*

- `3080.66s`
- `523` intermediate subindexes each around `25M`
- Page table of size `3.1G` (binary)

(b) Stage 2: *phase*1*merge.cpp*

- `830.20s`
- `21` intermediate subindexes each around `600M`

(c) Stage 3: *phase*2*merge.cpp*

- `472.60s`
- Uncompressed inverted index of size `12G`

(d) Stage 4: *compress.cpp*

- `665.76s`
- Final compressed inverted index of size `3.2G` (binary)
- Lexicon of size `1.4G`

The total amount of time used is $5049s \sim 84min$, which is not so fast. There are several possible directions to improve the speed.

- Use a more efficient library to parse `.trec` file instead of `getline` and `strtok`.
- Take advantage of `unordered_map` and Unix `sort` instead of `map`.
- Generate less intermediate(temp) subindexes.
- Carefully manage the available/used memory. I just try and error, e.g. `segmentation fault`, throughout the homework.