CS276 PA1 Report

by Jiawei Yao & Wei Wei

1 System Design

We followed the structure of the starter code and kept the predefined interface intact. To summarize:

- `Index` class implements the BSBI indexing algorithm
- `Query` class implements boolean conjunctive query processing algorithm
- `BaseIndex` defines the interface for writing/reading posting lists
- `BasicIndex` class writes uncompressed posting lists to/reads posting lists from disk
- `VBIndex` writes/reads posting lists with variable byte encoding for gaps
- `GammaIndex` writes/reads posting lists with gamma encoding for gaps

There are also some helper classes, e.g. `ListLengthComparator` which is used for sorting posting lists when processing the query.

Some key algorithm are explained in detail as follows.

1.1 Indexing Algorithm

Key steps in the indexing algorithm are:

1. Process blocks one by one:
    1. For each block, traverse each file and generate <Term ID, Doc ID> pairs
    2. After processing all files in the block, sort the <Term ID, Doc ID> pairs first by Term ID then by Doc ID and organize pairs with the same Term ID into a posting list.
    3. Create a file and store the posting lists with specified posting list writing algorithm
    4. Push the file into the merge queue
2. Merge blocks into a unified index:
    1. Pop two block files from the queue
    2. Merge the postings lists with the same Term ID (if possible) into a single one with standard merging algorithm
    3. Write merged posting lists into a merged file
    4. Push the merged file into the merge queue
    5. Proceed until the size of queue is 1

3. Write supplemental information (Term to Term ID mapping, Doc Name to Doc ID mapping, posting list positions) out to files

The indexing algorithm guarantees that:

1. Term ID and Doc ID are always greater than 0

2. When indexing the blocks, only one block is loaded into memory at a time

3. When merging two blocks, the indexes are loaded in in a stream fashion. That is, for each index to merge, only a posting list is loaded at a time.

1.2 Query Processing Algorithm

First, split the input string by whitespace. Then order the terms by postings list length to optimize query performance.

1.3 I/O

We use `java.nio` package for file I/O and mainly rely on `RandomAccessFile`, `FileChannel` and `ByteBuffer` classes. The latter two classes provide byte-level access to files, which we find useful as we need to record position in file for each posting list[1]. Though low-level, the `java.nio` package offers us finer control over disk access.

1.4 Variable Byte Encoding

A posting list is kept on disk in three parts: Term ID, #bytes, Doc ID gaps. We only compress the Doc ID gaps for two reasons:

1. The Term ID will always occupy 4 bytes and thus can be read deterministically when loading. So is for #bytes.

2. Compressing the Doc ID gaps already gives a huge space saving, and further compressing the Term ID and size will only provide trivial space saving but heavy loading overhead.

When loading a posting list, Term ID and #bytes are read in first. With #bytes, we can determine how many bytes to read in and then decode the Doc ID gaps.

1.5 γ Encoding

A posting list is kept on disk in two parts: #bits and IDs, where IDs include Term ID and Doc ID gaps. #bits is the number of bits after the IDs are γ-encoded. #bits is not encoded and is stored in 4 bytes on disk.

γ encoding is a bit more complicated than VB encoding. Writing a posting list involves:

1. γ-encode the IDs into a `BitSet`

2. Set #bits to the number of bits used in the `BitSet`

3. Convert the `BitSet` to a `ByteBuffer` (pad last byte with 0 if necessary)

1. Every time we need to write a posting list, we use the current position of `FileChannel` as the start position of the posting list in file.

4. Write #bits and the `ByteBuffer` to file

Similarly, when reading a posting list:

1. Read 4 bytes - it's #bits
2. Compute #bytes to read with #bits[2] and read into a `ByteBuffer`
3. Convert the `ByteBuffer` to a `BitSet`
4. γ-ecode the `BitSet` into integers up to #bits bits
5. the first integer is the Term ID and the following are the Doc ID gaps

2 Performance

We run the development set on a Retina 13 MacBook with normal load. The statistics are[3]:

Compression Method	No	VB	Gamma
Index Size	72M	35M	31M
Indexing Time	76s	100s	97s

Table 1. Index Sizes

The query processing time are[4]:

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Max. Memory
No Compression	2s	1s	2s	1s	2s	2s	2s	1s	202K
VB	2s	1s	2s	1s	2s	2s	2s	1s	202K
Gamma	1s	1s	2s	2s	2s	2s	2s	2s	202K

Table 2. Query Time

3 Discussion

a) In this PA we asked you to use each sub-directory as a block and build index for one block at a time. Can you discuss the tradeoff of different sizes of blocks? Is there a general strategy when we are working with limited memory but want to minimize indexing time?

Solution: Larger blocks would introduce larger memory usage but fewer rounds of merges. A extremely large block might not fit into the memory such that we cannot do in-memory sort. Similarly, smaller blocks would use less memory but lead to more rounds of merges. An too small block not only introduces unnecessary rounds of merges but fails to utilize memory efficiently.

A proper choice of block size would both fully utilize the memory and reduce the rounds of merge. So the block size can be chosen as:

$$\text{block size} = \text{total mem - auxiliary information size}$$

where auiliary information is the `termDict`, `docDict`, etc.

2. See `GammaIndex.numBytes(int bits)`.
3. The numbers in `indexsize.txt` and `indextime.txt` reported.
4. The numbers in `querytime.txt` and `query_memory_out` are reported.

b) Is there a part of your indexing program that limits its scalability to larger datasets? Describe all the other parts of the indexing process that you can optimize for indexing time/scalability and retrieval time.

Solution: There are several parts in our implementation which limits the scalability:

- sub-directory as a block: won't fit into memory if the subdirctory is too large
- global auxiliary information such as `termDict`: linearly grows with corpus size and will occupy too much memory, making not enough space for block (or even exceed the total memory size)
- loading the whole posting list into memory at once: if the term is too frequent, the posting list will be very large and won't fit into memory

Potential improvements include:

- use multi-way merge when merging blocks: will reduce disk I/O
- choose proper block size according to a)
- cache auxiliary information (e.g. `termDict`) to reduce memory usage
- load posting list in a buffered way instead of loading the whole list

c) Any more ideas of how to improve indexing or retrieval performance?

Solution: Ideas include:

- sort posting lists by size and load a posting list only when it's its turn to merge (the current implementation will load posting lists of all query terms)
- use skip list: reduce comparison when merging posting lists
- compress auxiliary information if it's too large and still keep them in memory (no need to do disk I/O)
- cache query results (good for frequent query)
- model indexing/retrieval into a distributed system (like what Google does)