Web Information Retrieval (67782)
Ex1: Index Structure Analysis

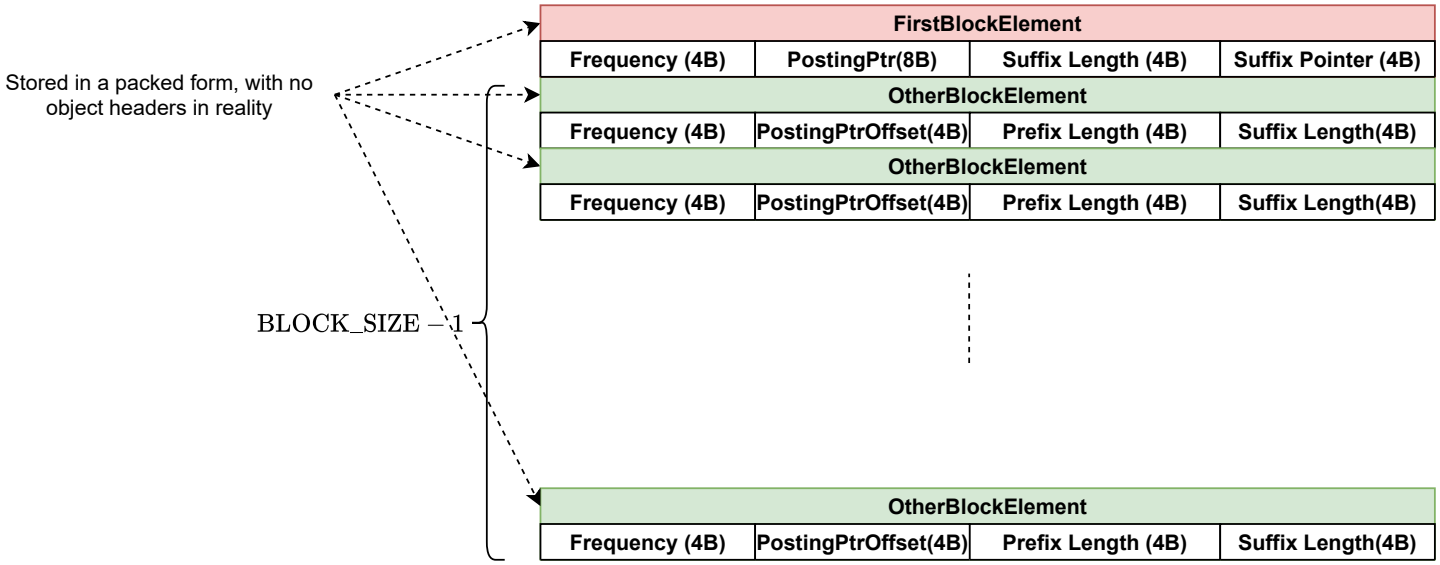
Submitted by: Daniel Kerbel (CSE: danielkerbel, ID: DELETED)

1 General Explanation and Diagram

1.1 Diagrams

Here are some diagrams which summarize the structure of the index. Read below for details.

Dictionary structure



Example for ["abc","abcd", "abkd", "aboo", "aboz"] with

$$\texttt{BLOCK\_SIZE} = 4$$

Terms.txt:

A B C D K D O O A B O Z

Suffix:	ABC	D	KD	OO	ABOZ
Suffix Pointer(FirstBlockElement)	0				8
Suffix Length	3	1	2	2	4
Prefix Length(OtherBlockElement)		3	2	2	

Inverted Index Example

Suppose we have a posting list with 3 entries:

[{ docId = 1, freq = 3}, { docId = 258, freq = 4}, { docId = 4,294,967,038, freq = 16,777,215 }]

Simplified view of dictionary:

Posting Pointers	Freq	More columns...
...	...	...
	3	...
	...	...

Group Varint encoding format

(between 5 and 17 bytes used for every 4 digits/every 2 posting list elements)

Length Byte of the form: $L_1 L_2 L_3 L_4$	L_1 bytes Number 1	L_2 bytes Number 2	L_3 bytes Number 3	L_4 bytes Number 4

Where L_i is 2 bits, representing length of integer(betwen 1 and 4 bytes)

Postings file:

...	...	...	...	...	...
Num (1B) 00010000	Num (1B) 1	Num (1B) 3	Num (2B) 257	Num (1B) 4	Num (1B) 11100000
Num (4B) 4,294,966,780	Num (3B) 16,777,215	Num (1B) 0	Num (1B) 0	...	...

Fixed Size Storage

1. Storage for reviews, ordered by docIDs
2. Storage for (docIdStart, docIdEnd), ordered by productIDs

1. Compact Review Storage

docID = index + 1

productID(10B)	helpfulness numerator (2B)	helpfulness denominator (2B)	score (1B)
Z123456789	5	7	3
B123456789	0	1	2
...	...	...	...

docID = 1

docID = 2

2. docID pair storage

Ordered by productIDs.
Indices have no direct meaning, but can apply binary search via productIDs

range of reviews with lowest(by lexicographic order) productID

range of products with highest productID

fromDocId(4B)	toDocId(4B)	productId(10B)
200	210	A123456789
40	45	A234567890
...	...	...
170	170	Z999999999

1.2 Dictionary

The dictionary uses (k-1)-in-k front-coding for compression, containing blocks of constant size, each block containing `BLOCK SIZE` terms encoded via front-coding. (I chose a block size of 4, for no particular reason)

Each element in the block contains the token document frequency(length of posting list) and a posting pointer(if this is the first element), or the gap from the posting pointer of the first element for any other element.

A full posting pointer is 8 bytes (As the posting file for the largest dataset, `all.txt`, even with compression, uses about 8 GB, which don't fit in an int), whereas an offset is 4 bytes. (In practice, probably a lot less, but this depends on the maximal posting list length)

In addition, the first block element contains a suffix pointer and suffix length (allowing de-referencing its term), and every other block element contains a prefix length and suffix length. [1]

Since other block elements do not include a suffix pointer, we need to de-reference every previous term in a block in order to determine the suffix pointer. (Formally, $\texttt{suffixPointer}_i = \texttt{suffixPointer}_{i-1} + \texttt{suffixLength}_{i-1}$)

This implies a space-time trade-off with regards to block size, a higher block size allows better compression of the terms, but higher dictionary search time.

There are a couple of optimizations that I did not yet implement due to time constraints,such as omitting the suffix length of the last block element, as it can probably be derived using the suffix pointer of the first element in the next block, but doing so just complicates the code and the benefits don't seem particulary big. Another option is to shrink the suffix/prefix lengths, maybe using 2 or even 1 byte - this would depend on the statistics of the dataset, I hypothesize that for a large dataset these lengths would tend to be smaller.

One important optimization that I did implement, is the use of a "packed" array (see class `PackedDictionaryElements`), instead of storing these blocks as plain Java objects, they are stored as bytes. This saves a lot of memory that would otherwise be wasted on object headers, at cost of higher running time when reading those bytes as objects when trying to retrieve an element from the dictionary.

In addition, two integers, total number of reviews and token collection frequency, which were calculated during parsing, are stored in a separate file, allowing $O(1)$ access.

1.3 Inverted Index

The inverted index consists of D posting lists, each identified by a posting list pointer and token document frequency(the length of the posting list)

Each posting list is a sequence of the form $(\texttt{docIdGap}_i, \texttt{frequency}_i)_i$ which are compressed using group varint encoding.

1.4 Compact Review Storage

A "compact review" consists of the following fields:

- Product ID (10 bytes)

[1] In class we store the entire token's length rather than the suffix length. Overall this doesn't matter because length = suffixLength + prefixLength

- Score (1 byte)
- Helpfulness Numerator(2 bytes)
- Helpfulness Denominator(2 bytes)
- Number of tokens (4 bytes)

The compact review store is a file containing these records, ordered by review IDs. Since each such review is of fixed size, it is possible to do $O(1)$ random access on the file to obtain a review with a given review ID.

1.5 Product ID to docID Pair Storage

In order to allow $O(\log D)$ implementation of `getProductReviews(String productId)`, there is a file containing D fixed size records of the form $(fromDocId, toDocId, productID)$ ordered by the product IDs. Each record represents the range of reviews for said product.

The fact the product ID stored on both the compact review storage as well as the pair storage might seem wasteful, but it reduces time spent on IO while doing binary search and allows better utilization of cache locality.

2 Main Memory Versus Disk

The token strings are loaded into memory, as well as the elements of the dictionary (in a packed form), and some statistics (token collection frequency and number of reviews)

The "compact reviews", "product ID to docID pairs" and posting lists are kept on disk. I use Java's buffered reader class when reading postings, which might load a few more kilobytes into memory, but they essentially take constant memory.

3 Theoretical Analysis of Size

N Number of reviews

M Total number of tokens (counting duplicates as many times as they appear)

D Number of different tokens (counting duplicates once)

L Average token length (counting each token once)

F Average token frequency, i.e., number of reviews containing a token

P Number of different products (namely $P = O(N)$)

R Average number of different reviews in which a token appears. (Namely, $R = O(N)$)

Dictionary $O(D \cdot L)$)

If we ignore front-coding or if all tokens don't share any suffixes, then the average space taken by the strings is $O(D \cdot L)$. Since `BLOCK_SIZE` $\ll D$ then it's clear that even with front-coding,

there's no asymptotic difference in the file size with (k-1)-in-k front coding in contrast to plain string concatenation.

As for the contents of the dictionary itself, we have D elements and each dictionary element takes a fixed amount of bytes, thus the dictionary contributes $O(D)$ memory, so in total we have a space complexity of $O(D \cdot L)$

Inverted Index $O(\frac{M}{F})$

On average, every F occurrences of some token will belong to the same document, thus they will be encoded as a single tuple within some posting list, giving us a total of $O(\frac{M}{F})$ tuples in the posting list.

Each tuple is of constant size, containing the docID gap and the document frequency, each using between 1 and 4 bytes(encoded via group-varint encoding), giving us a space complexity of $O(\frac{M}{F})$

Compact Review Storage $\emptyset(N)$

Mostly explained in a previous section - we have N records of fixed size

Product ID to docID Pair Storage $O(P)$

Mostly explained in a previous section - we have P records and each only uses 2 integers and 10 bytes(18 bytes per record)

4 Theoretical Analysis of Runtime

- `IndexReader(String dir)`: $O(D \cdot L)$

  The token strings(without repetitions of course) are held within memory The dictionary is loaded into memory:

  The token strings are held in a memory mapped file, which ideally(enough RAM) would be loaded in its entirety to disk, using $O(D \cdot L)$ space. The dictionary structure(essentially an array of fixed size records between 3 and 4 ints) is itself $O(D)$, therefore the total running time is $O(D \cdot L)$

- `getProductId(int reviewId)`: $O(1)$

  See compact storage, can access a review entry via random access based on docID.

- `getReviewScore(int reviewId)`: $O(1)$

  See compact storage, can access a review entry via random access based on docID.

- `getReviewHelpfulnessNumerator(int reviewId)`: $O(1)$

  See compact storage, can access a review entry via random access based on docID.

- `getReviewHelpfulnessDenominator(int reviewId)`: $O(1)$

  See compact storage, can access a review entry via random access based on docID.

- `getReviewLength(int reviewId)`: $O(1)$

  See compact storage, can access a review entry via random access based on docID. (The review length is computed when creating the compact storage during parse-time, by `SlowIndexWriter`)

- `getTokenFrequency(String token)`: $O(\log D)$

  Binary search in the dictionary takes $O(\log D)$, the token frequency is stored in the dictionary element so can be obtained in $O(1)$ once we found the position within the dictionary.

- `getTokenCollectionFrequency(String token)`: $O(\log(D) + R)$

  First, finding the posting list(and its length/token's length) takes $O(\log D)$

  As I defined before, the average token appears in $O(R)$ different reviews, which is the size of its posting list/document frequency, giving us a cost of $O(R)$ to iterate the posting list and sum up the frequencies to obtain the collection frequency.

- `getReviewsWithToken(String token)`: $O(\log D)$ if we don't iterate over the enumeration, otherwise $O(\log(D) + R)$

  Producing the enumeration(an iterator) only requires finding the posting pointer and document frequency of a given token, which like I explained above, is $O(\log D)$. Iterating over the entire enumeration would take $O(R)$ as we'd iterate over the posting list.

- `getNumberOfReviews()`: $O(1)$

  Stored in the `Dictionary\verb` object as a custom field, was calculated by `SlowIndexWriter` when parsing the input file.

- `getTokenSizeOfReviews()`: $O(1)$

  Same as above

- `getProductReviews(String productId)`: $O(\log P)$ if we don't iterate over the enumeration, otherwise $O(\log P + \frac{N}{P}))$

  To build the enumeration we simply need the lowest and highest docIDs of reviews for a product, which are stored in a single element within the docID pair storage, which is ordered by product IDs(see before), thus we can perform binary search over said file in $O(\log P)$

  If we include the iteration of the enumeration - splitting the N reviews over P sets of products(on average), means the average product has $\frac{N}{P}$ reviews, which is the size of the enumeration.