Chapter #5: Index Compression

* Why Compression ?
  - ↳ use less disk space.
  - ↳ keep more stuff in memory.
  - ↳ increase speed of transferring data from disk to memory.
    - ↳ speed(read compress+decompress) > speed (read decompress)
  - ↳ IR systems run faster on compressed posting lists.
  - ↳ We try to make the dictionary small enough to keep it main memory.
  - ↳ then reduce disk space for postings.

* Rule of 30 =>

If the search results returns more than 30 documents, its not necessary to review all of them, instead a sample of random 30 documents can be selected for review. If none of the samples were relevant, then its likely that remaining docs were irrelevant and the search was unsuccessful.

* lossless compression => all information is preserved.
* lossy compression => better compression ratios can be acheived by lossy which discards some information by stemming, lemmatization and stop word elimination.
* Heap's law => Heap's law estimates vocabulary size as a function of collection size.

$$M = kT^b \rightarrow b \approx 0.5$$

$30 \leq k \leq 100$ ↳ number of tokens in collection

The motivation for heap's law is that the simplest possible relationship b/w collection size and vocabulary size is linear log-log space.

- ↳ The dictionary size continues to increase with more documents in the collection.
- ↳ The size of dictionary is quite large with large collections.

CamScanner

Dated: _____

↳ Case folding & stemming reduce the growth rate of vocabs whereas including numbers & errors increases it.

★ Zipf's law ⇒ In a natural language, there are very few frequent terms and many rare terms.

"The ith most frequent term has frequency proportional to 1/i"

$cf_i \propto 1/i = a/i$ → normalizing constant

Collection frequency [the no. of occurrences of term t_i in the collection]

If the most frequent term occurs cf_1 times, then the second most frequent terms has half as many occurrences and the third most a third as many occurrences.

$$\log cf_i = \log a - \log i$$

Linear relationship b/w $\log cf_i$ and $\log i$.

↳ rank X frequency ~ constant

R	word	frequency	Rxf
10	he	877	8770
20	but	410	8200
30	be	294	8820

★ Dictionary Compression ⇒

↳ Written English – 4.5 characters/words

↳ Avg dictionary word ~ 8 characters

↳ Fixed length words for dictionary (20 bytes)

↳ short words dominate token counts but not type avg.

↳ Dictionary as a string ⇒

The simplest way is to sort the vocabulary lexicographically and store it in array of fixed-width entries.

↳ Allocate 20 bytes for the term itself. } expensive

↳ 4 bytes for its document frequency. }

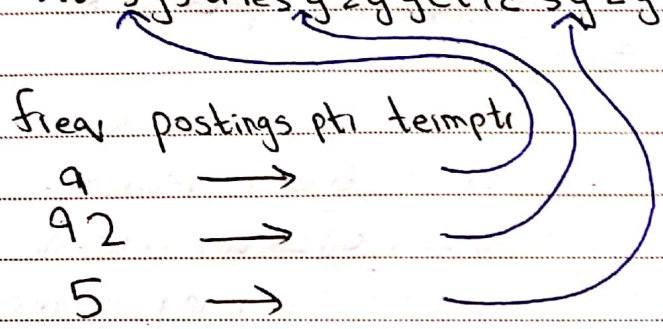
↳ 4 bytes for pointer to posting list.

We can overcome all the shortcomings by storing the

Dated: _____

Whole dictionary as a long string. This saves 60% of the space. In this scheme, we have 4 bytes for frequency, 4 bytes for postings pointer and the final 3 bytes for term ptr. we also have 8 bytes on avg. per term.

... systilesyzygeticsyzygialszaibelyites



To calculate dictionary size,

$$\text{Size} = M \times (4+4+3+8)$$

↳ number of terms

↳ Blocked storage ⇒ Group the terms into blocks of size k and keeping a term pointer to the start of the block. We store the length of the string as an additional byte at the beginning of the terms. We thus eliminate k-1 term pointers. For k=4, we save (k-1)x3 = 9 bytes

By increasing block size k, we get termptr size better compression but its a tradeoff with the block lookup speed.

(b) How to search queries?

↳ we first locate the term's block with binary search and its position within the list by linear search in the block.

Dated:

↳ Front coding => A data compression technique used to store and retrieve sorted strings efficiently. It works by storing only the difference b/w consecutive strings instead of storing each string in full.

Consider a block compressed (k=4)

8 automata 8 automate 9 automatic 10 automation

with front coding

8 automat * a 1 ◇ e 2 ◇ ic 3 ◇ ion

Common prefix | different | suffixes

end of prefix

replace * with this

★ Posting Compression =>

↳ The postings file is much larger than the dictionary by a factor of atleast 10.

↳ key desideratum => store each posting compactly.

↳ our goal is to use far fewer than 20 bits per docID.

↳ Variable Length Encoding =>

Since storing the document IDs in posting lists takes around 20 bytes, our target is to reduce the number of bytes so we decide to encode gaps in docIDs instead of the IDs themselves.

posting list => 283042 → 283043 → 283044

gaps => 1 → 1 → 1

we use VLE in this sense to have fewer bits for shorter gaps.

If the average gap for a term is G, we want to use ~$\log_2 G$ bits/gap entry.

Dated:

{Bytewise encoding}

↳ Variable Byte Encoding => Use an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is the continuation bit. It is set to 1 for the last byte of the encoded gap else 0.

To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts.

doc IDs	824		829	215406
gaps			5	214577
VB code	00000110	10111000	10000101	00001101 00001100 10110001

{Bitwise encoding} {scarce disk space}

↳ γ Codes =>

VB codes use an adaptive number of bytes depending on the size of the gap. Bit-level codes adapt the length of the code on a finer grained bit level {representing data with bits}

↳ unary code => The simplest bit-level code. The unary code of n is a string of n 1's followed by a 0. This is not very efficient and might exceed $\log_2 G$ bits $1 \rightarrow 10$ lower bound.

$9 \rightarrow 1111111110$

↳ γ encoding => A method within the factor of optimal. They implement VLE by splitting the representation of a Gap G into a pair of length & offset.

Offset is G in binary but with the leading 1 removed. (13 binary 1101) offset is 101.

Length encodes the length of the offset in unary code. 13 → Length of offset is 3 bits 00 1110

Therefore, γ code is 1110 101.

(1110: length, 101: offset)

Dated: _____

A γ-code is decoded by first reading the unary code till 0 that terminates it. Now, we know how long the offset is so we can prepend the chopped off 1 back to get the original number.

The length of offset is $\lfloor \log_2 G \rfloor$ bits and length of length is $\lfloor \log_2 G \rfloor + 1$ bits. γ codes are always of odd length $2\lfloor \log_2 G \rfloor + 1$ bits and they are within a factor of 2 assuming that the 2^n gaps b/w 1 and 2^n are equiprobable. In general, this is not the case.

↳ Entropy=>

The entropy of a set of integers can be measured by calculating avg. length of codes. The avg. length can be compared with the minimum possible code length.

If the avg. length of γ-code is closer to min. possible code length then the set of integers has low entropy, meaning that distribution of integers is relatively predictable or non-random. On the other hand, if it has high entropy, this means that the set of integers is relatively unpredictable or random. This can help in efficient index retrieval.

↳ Universal code=> A code like γ code with the property of being within a factor of optimal for an arbitrary distribution P.

↳ prefix free=> no γ code is prefix of another. So, there is always a unique decoding so we have increased efficiency.

↳ Parameter free=> γ-code is parameter free so it does not depend on changing gap distribution parameters.

All of this is Index Compression. The retrieval is finally highly efficient. Only 10-15% of total size of text in collection.