Efficient encodings for inverted index

Yiliang Xiong

April 19, 2018

Overview

Posting list

A posting list can be defined as a monotonically increasing sequence

$$0 \leq x_0, \leq x_1, ..., x_{n-1} \leq u$$

where u is the upper bound of *doc id*.

Posting list

A posting list can be defined as a monotonically increasing sequence

$$0 \leq x_0, \leq x_1, ..., x_{n-1} \leq u$$

where u is the upper bound of *doc id*.

The sequence can also be represented by the gaps between consecutive numbers:

$$\delta_0, \delta_1, ..., \delta_{n-1}$$

where $\delta_0 = 0$ and $\delta_i = x_i - x_{i-1}$.

Posting list

A posting list can be defined as a monotonically increasing sequence

$$0 \leq x_0, \leq x_1, ..., x_{n-1} \leq u$$

where u is the upper bound of *doc id*.

The sequence can also be represented by the gaps between consecutive numbers:

$$\delta_0, \delta_1, ..., \delta_{n-1}$$

where $\delta_0 = 0$ and $\delta_i = x_i - x_{i-1}$.

How to restore the original sequence?

Posting list

A posting list can be defined as a monotonically increasing sequence

$$0 \leq x_0, \leq x_1, ..., x_{n-1} \leq u$$

where u is the upper bound of *doc id*.

The sequence can also be represented by the gaps between consecutive numbers:

$$\delta_0, \delta_1, ..., \delta_{n-1}$$

where $\delta_0 = 0$ and $\delta_i = x_i - x_{i-1}$.

How to restore the original sequence?

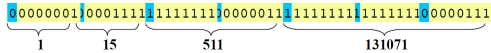
- Prefix sum $x_i = \sum_0^i \delta_i$
- High-performance algorithm exists:
  `https://en.wikipedia.org/wiki/Prefix_sum#Parallel_algorithm`

Variable-byte encoding

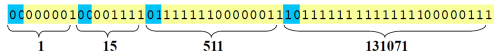
Google’s index format in early days

Byte-Aligned Variable-length Encodings

- Varint encoding:
  - 7 bits per byte with continuation bit
  - Con: Decoding requires lots of branches/shifts/masks



- Idea: Encode byte length as low 2 bits
  - Better: fewer branches, shifts, and masks
  - Con: Limited to 30-bit values, still some shifting to decode

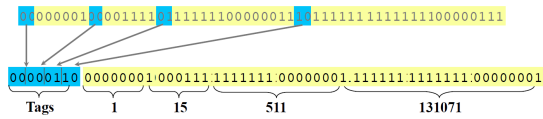


Stolen from Jeff Dean’s Challenges in Building Large-Scale Information Retrieval Systems.

Google's index format in early days (cont'd)

Group Varint Encoding

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix



- Decode: Load prefix byte and use value to lookup in 256-entry table:

```
...
00000110 ⟶ Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffff
...
```

- Much faster than alternatives:
  - 7-bit-per-byte varint: decode ~180M numbers/second
  - 30-bit Varint w/ 2-bit length: decode ~240M numbers/second
  - Group varint: decode ~400M numbers/second

Stolen from Jeff Dean's Challenges in Building Large-Scale Information Retrieval Systems.

More examples: Protobuf encoding

Each byte in a varint, except the last byte, has the most significant bit (msb) set. This indicates that there are further bytes to come.

So, for example, here is the number 1 - it's a single byte, so the msb is not set:

```
00000001
```

And here is 300 - this is a bit more complicated:

```
10101100 00000010
```

More details can be found at `https://developers.google.com/protocol-buffers/docs/encoding`

More examples: UTF-8

Bits at the beginning of the first byte determine the code pattern:

- `0_______` backwards compatibility with ASCII
- `11______` the first byte of a UTF-8 encoded character
- `10______` the remaining byte of a UTF-8 encoded character

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	`0xxxxxxx`			
2	11	U+0080	U+07FF	`110xxxxx`	`10xxxxxx`		
3	16	U+0800	U+FFFF	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	
4	21	U+10000	U+10FFFF	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

See history of UTF-8 at `https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt`

Simple9/Simple16 encoding

Simple9 (S9) encoding

The basic idea is to try to pack as many integers as possible into one 32-bit word.

To do this, Simple9 divides each word into 4 status bits and 28 data bits, where the data bits can be divided up in 9 different ways:

```
28 1-bit numbers
14 2-bit numbers
9 3-bit numbers (onebit unused)
7 4-bit numbers
5 5-bit numbers (three bits unused)
4 7-bit numbers
3 9-bit numbers (one bit unused)
2 14-bit numbers
1 28-bit number
```

Vo Ngoc Anh, Alistair Moffat, Index Compression using Fixed Binary Codewords (ADC 2004).

Simple16 (S16) encoding

Simple9 wastes bits in two ways,

- by having only 9 cases instead of the 16 that can be expressed with 4 status bits, and
- by having unused bits in several of these cases.

Simple16 (S16) encoding

Simple9 wastes bits in two ways,

- by having only 9 cases instead of the 16 that can be expressed with 4 status bits, and
- by having unused bits in several of these cases.

A new variation that avoids this:

```
5 5-bit numbers in Simple9, with 3 bits unused ==>
        3 6-bit numbers followed by 2 5-bit numbers
        2 5-bit numbers followed by 3 6-bit numbers
```

Jiangong Zhang, Xiaohui Long, Torsten Suel, Performance of Compressed Inverted List Caching in Search Engines (WWW 2008)

Frame-Of-Reference encoding

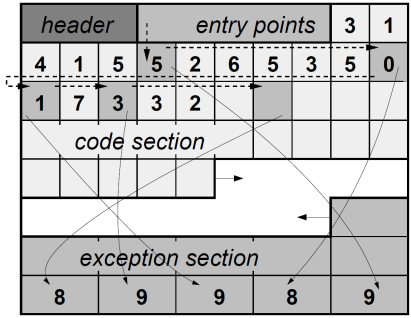
A naive approach to decompression

The naive way to implement any decompression scheme that distinguishes between coded and *exception* values, is to use a special code `MAXCODE` for exceptions:

```
/* NAIVE approach to decompression */
for(i = j = 0; i < n; i++) {
    if (code[i] < MAXCODE) {
        output[i] = DECODE(code[i]);
    } else {
        output[i] = exception[--j]);
    }
}
```

Patched Frame-Of-Reference (PForDelta)

An elegant and efficent way to handle exceptions:



- What if two consecutive exceptions have a distance of more than 2^b?

Marcin Zukowski, Sandor Heman, Niels Nes, Peter Boncz, Super-Scalar RAM-CPU Cache Compression (ICDE '06).

PForDelta decompression

Compared to the naive implementation:

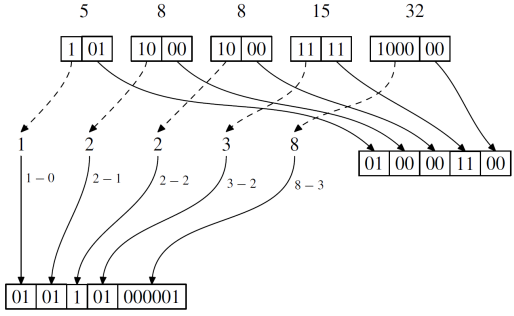
- no `if-else` statement
- reduce branch misprediction (better CPU pipelining)

```
int Decompress<ANY>(
    int n, int b, ANY *output, void *input, ANY *exception, int *next_exception) {
    int next, code[n], cur = *next_exception;
    UNPACK[b](code, input, n);  /* bit-unpack the values */
    for(int i = 0; i < n; i++) {                /* LOOP1: decode regardless */
        output[i] = DECODE(code[i]);
    }
    for(int i = 1; cur < n; i++, cur = next) {  /* LOOP2: patch it up */
        next = cur + code[cur] + 1;
        output[cur] = exception[-i];
    }
    *next_exception = cur - n;
    return i;
}
```

Elias-Fano encoding

Elias-Fano encoding

- Given n and u we have a monotonically increasing sequence $0 \leq x_0, \leq x_1, ..., x_{n-1} \leq u$
- Store the lower $l = log(u/n)$ bits explicitly
- Store the upper bits as a sequence of unary coded gaps (0^k1 represents k)
- Use at most $2 + log(u/n)$ bits per element



Peter Elias, Efficient Storage and Retrieval by Content and Address of Static Files (JACM 1974).

Sebastiano Vigna, Quasi-Succinct Indices (WSDM '13).

How to search

Looking up k-th element:

- Suppose you want to get the k-th element quickly
- Just scan the upper bits, one word at a time, doing population counting
- When you get to the right word, complete sequentially and pick the lower bits

How to search

Looking up k-th element:

- Suppose you want to get the k-th element quickly
- Just scan the upper bits, one word at a time, doing population counting
- When you get to the right word, complete sequentially and pick the lower bits

Searching for any element $x_i \geq q$:

- It's exactly the same: only, you count zeroes
- Zeroes tells you how much the upper bits are increasing, which is the important thing
- Just skip `q >> l` upper zeroes and complete sequentially

Advatanges

Almost optimal space usage

- $2 + log(u/n)$ bits per element

Advatanges

Almost optimal space usage

- $2 + log(u/n)$ bits per element

Independent of *doc id* distribution

- works for long and short posting list

Advatanges

Almost optimal space usage

- $2 + log(u/n)$ bits per element

Independent of *doc id* distribution

- works for long and short posting list

Reading sequentially requires very few logical operations

- only shift and bitwise operation

Advatanges

Almost optimal space usage

- $2 + log(u/n)$ bits per element

Independent of *doc id* distribution

- works for long and short posting list

Reading sequentially requires very few logical operations

- only shift and bitwise operation

Encoding integer sequence directly instead of $\delta_i = x_i - x_{i-1}$

- no need to calculate *prefix sum* after decoding

Thank you

Any questions?