Komprimering av data med rice coding & LZ4

Av Erik Orten og Eirik Tøndel

Rice coding

- Lossless/tapsfri (vi mister ikke informasjon på komprimeringen)
- Et bestemt tilfelle av Golomb coding
  - enkel implementering
  - bedre for datadistribusjoner som er eksponentielle

Rice coding algoritmen

1. Bestem kodingsregelen:
   a. velg parameter k, ikke-negativt heltall, som representerer dataene som skal komprimeres
2. Del rice-coding input tallet n inn i

Kvotient = n // 2^k (heltallsdivisjon)	Rest = n AND (2^k)-1 bit for bit Legg på k - countBits(rest) bits i starten

3. Kvotient q encodes med unær koding
   a. skriv q antall 1-ere etterfulgt av 1 0-er. Eksempel: q = 3, unær koding => 1110
4. Rest r encoding
   a. kod rest r med en binær-representasjon av bestemt lengde, med nøyaktig k bits
5. Kvotient og rest samlet sett: unær-kode sammenslått med rest-bitene av lengde k

Valg av parameter k

- Mål: finn et valg av k som gir kortest bitlengde gjennomsnittlig for dataene

1. Beregn gjennomsnittet av dataene
    a. avgNum(dataset) = 4
    b. k ≈ log2 (4) = 2
2. Evaluer bitforlengelsen av forskjellige k nær 2, f.eks. k=1 og k=3:
    a. velg en mengde “representative data” (tall) fra datasettet
    b. velg k som “omtrent” minimerer den totale bitlengden for de representative tallene

Dekoding av rice-komprimert data

1. Les 1-ere av kvotient q, helt til vi finner 0 bit. Antall 1ere = kvotienten q dekodet
2. Rest r: les de neste k bitene.
3. Kombiner kvotient q og rest r med formelen: tallet n = q * 2^k + r

Eksempel på rice-coding

1. Komprimer tallet 13
2. Vi har parameter k = 2
3. Kvotient q = 13 // 2^2 = 3
   = 1110

Kvotient = n // 2^k
(heltallsdivisjon)

4. Rest r = 13 AND (2^2 - 1)
   = 13 & 3 = 1101 AND 0011
   = 0001 = 1

Rest = n AND (2^k)-1
Legg på k - countBits(rest)

   a. legg på k - countBits (r) antall bits slik at lengden på rest blir lik k = 2 = len(01)
5. Kvotient og rest kombinert = 1110 01 (6 bits for å representere tallet 13!)

Dekoding:

1. Les antall 1ere i q: countUnary(1110) = 3 = kvotient q
2. Les k neste bits: convertToInt (01) = 1 = rest r
3. Finn tallet n -> n = q * 2^k + r
   = 3 * 2^2 + 1 = 3 * 4 + 1
   = 12 + 1 = 13

kvotient = countUnary(kvotient)
rest = convertToInt(rest)
n = q * 2^k + r

Når bør Rice coding brukes?

- Dataene er eksponensielt fordelt med mange lave verdier
- Lav varians i dataen
- Med et representativt valg av k, vil kvotientet ta lite plass
  - Tenk på hvordan vi lagrer kvotienten i unær representasjon
  - kvotient på 10, vil bruke 11 bits
  - kvotient på 2, vil bruke 3 bits

LZ4 egenskaper

-Mindre (dårligere) kompresjonsrate

-Meget rask kompresjon og dekompresjon

-Godt egnet for informasjonsinnhenting/”realtime”-formål

-Kan ta fordel av flere kjerner

-Dekompresjon kan nå ram-hastighet med flere kjerner

LZ4 algoritme

-Bit 0-3: Antall nye bytes, 4-7: Antall gamle bytes (kan kopieres fra dekodet buffer)

-1111-trenger ekstra byte for å representere

-11111111 i den ekstra byten betyr enda en byte trengs, vilkårlig lengde mulig

-For byte 4-7: legg til 4 på tallet, ettersom 4 er minimum (0000 = 4, 0001 = 5 etc)

-Nye bytes

-Offset for gamle bytes

LZ4 eksempel - OR deler sekvens med AND

1 block i "or" posting listen, 288 bits komprimert til 96 bits ("heldig"/best case eksempel)

nye,gamle	gamle cont	ny byte nr 1-4 (representerer 1)				ny byte nr 5-8 (representerer 2)				offset (alltid 2B)	
10001111	00001011	00000000	00000000	00000000	00000001	00000000	00000000	00000000	00000010	00000000	00101000

8 nye bytes

4+15+13 = 28 gamle bytes som kan refereres til

32 offset - hopp 32 bytes bak, referer til de (let i forrige block)

Gap encoded posting lists - numbers as uint32, so each PL is 9*32=288 bits									
And	1	1	1	1	1	1	1	1	1
Or	1	2	1	1	1	1	1	1	1
Somehow	1	4	1	16	12	30	3	60	29

LZ4 eksempel - OR deler sekvens med AND

1 block i "or" posting listen, 288 bits komprimert til 96 bits ("heldig"/best case eksempel)

nye,g	amle	gamle cont	ny byte nr 1-4 (representerer 1)				ny byte nr 5-8 (representerer 2)				offset (alltid 2B)	
1000	1111	00001011	00000000	00000000	00000000	00000001	00000000	00000000	00000000	0000001(	00000000	00101000

8 nye bytes

4+15+13 = 28 gamle bytes som k reres til

32 offset - hopp 32 bytes bak, referer til de (le plock)

Gap encoded posting lists - numbers as uint32 ach PL is 9*32=288 bits									
And	1	1	1	1	1	1	1	1	1
Or	1	2	1	1	1	1	1	1	1
Somehow	1	4	1	16	12	30	3	60	29

LZ4 konklusjon

-Fungerer ved å referere til tidligere like sekvenser

-Meget rask, egnet for realtime

-Grei kompresjon, brukte ⅓ av ukomprimert plass i best case eksempel