Differentiel kodning af data

Greenleaf

23. september 2005

Resumé

Der angives en samlet metode til at beskrive et dataset β ud fra et andet datasæt α ved et mål for differensen Δ og til senere at gendanne β givet α og den beskrevne differens Δ.

Metoden kræver ved beregning af Δ ikke adgang til α men kun til $H(\alpha)$ og β. Ved gendannelsen af β kræves kun α og Δ.

Indhold

1 Differentiel kodning

Differentiel kodning handler om at beskrive et datasæt β udfra et andet dataset α og de ændringer der skal foretages ved α for at give β, eller rettere de ændringer der *er* foretaget ved α.

Man kan se differentiel kodning som en funktion $\Delta = D(\alpha, \beta)$ hvor Δ er differensen eller beskrivelsen af ændringerne.

Tilsvarende kan man udføre den modsatte operation hvor $\beta = R(\alpha, \Delta)$.

Forudsat at Δ kan repræsenteres mere kompakt end β så er det en komprimeret måde hvorpå man kan lagre β givet at man også lagrer α.

1.1 Beregning af difference

Den nok mest kendte algoritme til at beregne Δ givet α og β er baseret på Longest Common Subsequence. Den finder den længste sekvens af α i β og omvendt.

α	Dette er en slags test
β	Det var så den test
LCS	Det en test

Hvis man opstiller de to tekste over hinanden, så kan man let se at den angivne LCS faktisk er længste fælles streng.

```
Dette er   en slags test
||| /      |||      ||||
Det var så en       test
```

Den ændring der skal kodes består derfor kun af det der ikke indgår i længste fælles delstreng. Der er fjernet "er ", indsat "var så "og der er fjernet "slags ".

Algoritmen kan implementeres med dynamisk programmering og kodes til at køre i $O(n \cdot m)$ og med et pladsforbrug på $O(n \cdot m)$ hvor n og m er længden af de to strenge. Man kan altså se at for lange strenge vil køretiden og pladsforbruget vokse betydeligt. For en fuldstændig LCS-beregning på to

strenge på hver 1MB vil man skulle bruge 1GB plads til udregningen og køretiden vil være væsentlig.

LCS vil kunne give en optimal løsning i enhver situation uanset hvilken form Δ antager. Den er bare upraktisk til større datamængder.

1.2 Blokkodning

Et praktisk anvendeligt alternativ til LCS kan være blokkodning af data. Istedet for at se α og β hver som *en* lang datablok kan man opdele den ene eller begge i blokke af en mindre størelse.

Hvis man opdeler α og β i blokke af eksempelvis hver 1024 elementer, så kan man til beregningerne i LCS nøjes med et pladsforbrug på 1024*1024=1048576. Om pladsen måles i bytes, words eller større enheder afhænger af hvor lange delstrenge man ønsker at søge efter. Delstrenge på 1 til 256 i længde kan kodes i bytes, mens længere strenge kræver tilsvarende mere plads. For blokke på 1024 vil en delstreng naturligvis aldrig overstige 1024 i længden, så et 16 bit word kan lagre den maksimale længde og det reele pladsforbrug bliver 2097152 bytes. Det er ikke uoverkommeligt.

Man skal dog holde sig for øje at hvis der indsættes eller fjernes meget data mellem de to datablokke, så er det ikke givet at blok n i α har nogen relation til blok n β. Man skal derfor for hver blok i α søge efter bedste match i alle blokke i β. Det gør løsningen mere tidskrævende men stadig $O(n \cdot m)$ da antallet af blokke vokser proportionalt med længden af data.

1.3 Problemer ved Longest Common Substring

Det kunne virke som om LCS er et naturligt valg, omend den taber noget ved ikke at kunne implementeres til hurtigt og med lille pladsforbrug at finde den største delstreng. Det er imidlertid ikke den metode der anvendes i praksis, da den har en meget stor defekt i backup-sammenhæng: den kræver hurtig "random access"adgang til både α og β mens den udføre sin beregning.

I en backupsituation har man tidligere lavet en kopi af α og gemt den på et fjernlager. Siden er α modificeret lokalt og er nu blevet til β. Man har dermed kun direkte adgang til β. Man kan hente α hjem fra fjernlageret og beregne Δ og derefter lagre Δ på fjernlager. Det vil koste tid og båndbredde at gøre det sådan. Desuden skal man huske at det ikke er givet at α ligger i rå form

på fjernlageret. Det er muligt at den også blev lagret i differensform, så den først skal gendannes fra dens oprindelsesdata, hvilket tager endnu mere tid og serverresourcer.

1.4 Blokkodning med hashing

Man kan se at i praksis kan LCS ikke bruges til backup på fjernlager, omend metoden er god til mange andre ting.

Istedet for at implementere sin differentielle kodning som $\Delta = D(\alpha, \beta)$ kunne man ændre metoden til at fungere i to skridt.

Hashing	$h1, h2 = H(\alpha)$
Beregning af differens	$\Delta = D(h1, h2, \beta)$

$h1$ og $h2$ er resultatet af henholdsvis en hurtig men svag og en stærk men langsom hashberegning[1] på α. $h1$ har en bitlængde på 32 pr. blok og $h2$ har en bitlængde på 128 pr. blok. Det er altså givet at hvis bloklængden for α er defineret så en blok fylder mere end 160 bit så vil størelsen af $h1 + h2$ være mindre end størelsen af α. I praksis vil det altid være gældende og $h1$ og $h2$ er en yderst kompakt repræsentation af α.

Hvis man istedet for, som tidligere foreslået, at hente α hjem fra fjernlageret vælger at hente $h1$ og $h2$, så vil man hurtigere være klar til at påbegynde kodningen.

Et eksempel Hvis man har en α på 100MB og denne opdeles i blokke af 100kB, så vil $h1$ fylde 4kB og $h2$ vil fylde 16kB. Det giver et pladsforbrug på 20kB for de to hash tilsammen. Det en et kompressionsforhold på 1/5120

Spørgsmålet er nu hvordan den nye differensfunktion skal kunne fungere uden adgang til α, da det er α der skal sammenligned med β

1.5 Differens beregnet fra hash

Hvis jeg har en blok $blok\beta$ på n elementer i et datasæt β og vil se om en tilsvarende blok $blok\alpha$ på n elementer kan findes i et andet datasæt α, så kan

[1] Den svage er eksempelvis en Rolling Adler 32 og den stærke er MD5

man udføre et væld af hurtige effektive strengsøgningsfunktioner og se om blokken kan findes i α.

Hvis jeg derimod ikke har α men istedet har to hashværdier af hver blok på n elementer i α lagret i $hashList\alpha$, så er det også muligt (og helt trivielt) at søge efter et match til $blok\beta$.

Man beregner en hash $hash\beta$ af $blok\beta$ og derefter søger man gennem $hashList\alpha$ til man finder et match. Hvis man har anvendt den stærke hash så er man i praksis sikker på at have fundet en tilsvarende blok hvis man finder samme hashværdi.

1.6 Håndtering af dataforskydning

Det er ikke givet at der findes en $block\alpha$ der opfylder at $H(block\alpha) = H(block\beta)$, da det kræver at blokkene er absolut identiske.

Et eksempel Hvis vi har to sæt data, der begge opdels i blokke af 4 elementer, så kan de se ud som herunder.

α	ABCD	EFGH	IJKL	MNOP
$H(\alpha)$	15	65	78	124
β	QABC	DEFG	HIJK	LMNO
$H(\beta)$	76	12	9	37

Man kan se at β kun er forskellig fra α ved at der er tilføjet et tegn i starten og fjernet et i slutningen. Alligevel er der nu ikke en eneste blok der matcher. Søger man efter parvist matchende blokke, så vil man intet opnå.

Det naturlige vil være at lade blokkene forskydes frem og tilbage i begge datasæt. Man vil da kunne finde optimale match. Det kan bare ikke gøres, da man er nødt til *en* gang for alle at beregne $hashList\alpha$ og lagre den på fjernlager. Den er derfor ikke fleksibel.

Man kan stadig flytte blokgrænserne i β og beregne hash af hver rejusteret blok i et forsøg på at finde bedste match.

Et eksempel Ser man på to datasæt hvor α er opdelt i blokke af 4 elementer, så kan man forsøge at finde matchende blokke for enhver position i β.

α	ABCD	EFGH	IJKL	MNOP
$H(alpha)$	15	65	78	124

β QABCDEFGHIJKLMNOP

Først beregnes en hash af fire elementer i β med offset 0.

H(QABC)=76. Der søges i $hashList\alpha$ efter 76 som ikke findes.

Derefter beregnes en hash af de fire elementer i β med offset 1. H(ABCD)=15. Der søges i $hashList\alpha$ efter 15 og der er et match med $block\alpha_0$

Man kan altså finde blockmatch mellem de to datasæt ved at tillade at blot det ene sæts blokke kan forskydes.

1.7 Gentagende hash af forskudte blokke

Det er illustreret hvordan forskydelige blokke kan løse problemet med manglende match mellem α og β, som opstår når der er indsat eller fjernet data.

Det introducerer et andet problem. I værste tilfælde skal man foretage en komplet hashberegning for hvert muligt block-offset i β. Det vil sige $hl = (m - n) * n$, hvor m er længden af β , n er bloklængden og hl er elementer der skal hashes. Det er for små blokke ikke uoverkommeligt, men arbejdet vil være voldsomt for store n og m som forekommer i praksis. Nærmere bestemt vil det være $O(n^2)$.

Det er på tide at introducere begrebet "rullende hash"som er stærkt anvendeligt i strengsøgning.

Jeg vil illustrere princippet med en meget simpel hashfunktion. Funktionen tolker en streng som et tal i radix$_{256}$[2], og selve hashværdien er strengens radix$_{256}$-repræsentation modulus 256. Strengen er en sekvens af bytes. I eksemplet består den af bogstaver og disses ASCII-værdi. Der beregnes en hash for blokke med længden 4.

s=DETTE ER EN TEST

Første blok er "DETT"hvis ASCII-koder er 68, 69, 84, 84. Skal denne streng tolkes som radix$_{256}$, så bliver det til $(68 * 256^3 + 69 * 256^2 + 84 * 256^1 + 84 * 256^0)\%256 = 84$

[2] $Radix_x$ betyder x-talsystemet. Tiltalsystemet er altså $Radix_{10}$ og tallet 12 skal tolkes som $1 * 10^1 + 2 * 10^0$

Dette var i sig selv simpelt nok, og ikke nogen forbedring frem for andre hashfunktioner. Arbejdet var $O(n)$ hvor n er strengens længde.

Den rullende egenskab, jeg vil beskrive, gør at nok er første hash $H(s)$ $O(n)$ men efterfølgende hash af $H(s+1)$ er $O(1)$, hvor s er strengen og hashen er for en blok af længden n.

Man kan se at det der sker ved at flytte en position til højre i strengen er at D glider ud af blokken til venstre og E kommer ind i blokken fra højre samtidigt med at E, T og T alle rykker en plads til venstre. Det vil sige at $H(ETTE)$ står i nær relation til $H(DETT)$. Nærmere bestemt gælder at $H(ETTE) = ((H(DETT) - D * 256^3) * 256 + E)\%256$ Det der sker er at $D * 256^3$ trækkes fra, da det var det bidrag D gav i hashberegningen før. Derefter ganger man 256 på ETT hvilket svarer til at flytte alle tegn/cifre en plads til venstre i $Radis_{256}$. Til slut lægges E til, da den er skiftet ind fra højre.

Med en rullende hash kan man altså beregne hash af første blok i endelig tid og derefter beregne hashen for blokken der er forskudt en plads langs strengen i konstant tid.

1.8 Hashens styrke

Et problem med en rullende hash er at den generelt ikke vil være stærk. Man kan altså forholdsvis ofte opleve at selvom a og b er forskellige så gælder at $H(a) = H(b)$ Det vil gælde for par af a og b med enhver hash når hashens bitlængde er mindre end bitlængden af henhodsvis a og b, men for en svag hash gælder det oftere.

Hvis hashen har en længde på 32 bit og data, som hashes, har en bitlængde på 800 bit, så er der $2^{32} = 4294967296$ mulige hashværdier mens der er $2^{800} = 6,67 * 10^{240}$ mulige dataværdier. Det er $1{,}55{*}10^{231}$ dataværdier for hver hashværdi. Det betyder at for alle datakombination vil der være overordentligt mange kollisioner... ialt. Man skal stadig holde sig for øje at for dataværdi er sansynligheden for at en tilfældig anden dataværdi har samme hash under 1/4294967296, som er en lav sansynlighed, men det er noget der vil ske i praksis med mange hashberegninger.

Det vil derfor ikke være sikkert at antage at man har et blockmatch bare fordi den rullende hash siger at det er tilfældet. Man bliver nødt til at supplere den hurtige, men svage, rullende hash med en stærk, langsom ikke-rullende hash som man kan stole på.

Den rullende hash skal altså fungere som et hurtigt filter der bortfiltrerer alle positioner hvor der med sikkerhed ikke er en blockmatch. Det kan den da det gælder for enhver fungerende hashfunktion at hvis $H(a) \neq H(b)$ så er $a \neq b$

Da den stærke hashfunktion skal være meget pålidelig, og man ikke kan gøre mere for at sikre sig mod falske blockmatches end at beregne den stærke hash, så skal den have en lang bitlængde, så sansynligheden for falske match er så ringe at sansynligheden for snart sagt enhver anden defekt i processen er større.

1.9 Bloklængdens betydning

Man kan altså hurtigt finde matchende blokke i α på ethvert offset i β ved at køre en rullende hash hen over β, men metoden vil ingen match finde hvis frekvensen af ændringerne er større end frekvensen af blokkene.

Hvis man har en bloklængde på 1024 elementer og der netop er en ændring for hver 1024elementer, så vil enhver blok på et vilkårligt offset i β indeholde en ændring og dermed kan den ikke genfindes i α.

Man vil ikke kunne repræsentere β mere kompakt som $\Delta = D(H(\alpha), \beta)$.

Hvis man derimod, givet samme ændringer som før, istedet havde en bloklængde på 1023 elementer, så ville man genfinde en blok, ikke finde næste blok men efter at flytte offset *en* position vil man igen finde en blok.

Det vil dog være en fejl at tolke det som at det er optimalt at have meget små blokke, da man skal huske at hver blok vil fylde den kombinerede bitlængde af den stærke og svage hash. Hashlisten for meget små blokke vil dermed fylde mere og tage længere tid at søge i. Yderligere gælder at man i det færdige resultat skal bruge en endelig plads til at lagre information om en blok og små blokke giver mange blokke der skal lagres. Senere foreslås en sekventiel runlength encoding som mindsker dette sidste problem.

Bloklængden er dermed af stor vigtighed og vælges den forkert for de givne data så kan man få meget ringe kompression.

2 Implementation af differentiel kodning

Foregående sektion beskriver hvad differentiel kodning er, og hvordan det kan udføres på en realistisk måde. Dette afsnit vil træffe konkrete valg og gennemgå i detaljer hvordan kodningen kan implementeres.

Herfter betragtes data som sekvenser af bytes og ikke som abstrakte strenge.

Metoden overordnet:

- Beregn $H(\alpha)$ som to blokke af henholdsvis stærk og svag hash.
- For hvert byteoffset i β beregnes den svage hash af en blok og denne søges i α's hashliste.
- Hvis hashværdien blev fundet, så beregn den stærke hash for samme blok og søg denne i α's hashliste.
- Hvis hashværdien igen blev fundet så noter bloknummeret fra α's hashliste i Δ og behandel den følgende blok i β på samme måde.
- Hvis blokken ikke blev fundet så noter første byte i blokken i β i Δ og behandel blokken en byte længere fremme i β på samme måde.

Sidst i sektion 6 gennemgår jeg et lille regneeksempel manuelt, så det er utvetydigt hvordan man udfører kodningen.

2.1 Valg af bloklængde

Det enkeltstående vigtigste valg ved kodningen er valget af bloklængderne. Det er i praksis umuligt at definere nogen optimal længde, men der er nogle ting man kan tage i betragtning ved et valg.

Jeg betegner herefter ændringer fra α til β som "støj"og ikke som ændringer.

- Store datamængder. Fyldige data repræsenteres generelt bedst med store blokke. Er data meget omfattende så er det upraktisk at kode mange små blokke, da det vil give store hashfiler og Δ også bliver større da den indholder information om mange små blokke istedet for få store. Store datamængder vil generelt også have en forholdsmæssigt mindre støjmængde, så der er større områder der er uændrede.

- Små datamængder. Er data små så vil en stor blok kunne give forholdsvis stort spild. Man vil i gennemsnit få et spild i slutningen af data, da længden af data og længden af blokke sjældent vil passe perfekt sammen. Med en datalængde på δ og en bloklængde på ω, så vil man skulle forvente at få et spild på $\frac{1}{2}\omega$. Det vil udgøre en uforholdsmæssigt stor del af data for små data, da $\lim_{\delta \to 0} \frac{1}{2}\omega/\delta = \infty$

- Er støjen lokal i start og slut, så er store blokke bedst, da den sammenhængde centrale del af data kan kodes som *en* stor blok.

- Er støjen spredt med høj frekvens gennem data, så vil store blokke være meget inefektive og man bør bruge blokke med en længde der giver dem en frekvens som er højere den gennemsnitlige støjfrekvens.

Det er desværre i praksis ikke muligt at lave den form for foranalyse, da det kræver adgang til både α og β og man kun har β og $H(\alpha)$. Hvis man vil have en optimal bloklængde, en længde der minimerer størelsen størelsen af Δ og $H(\alpha)$, så må man være i besidelse af en vis domæneviden.

Det er sædvanligt at støjen for et dokument er lokaliseret primært i slutningen. Samme gælder for eksempelvis en database hvortil der tilføjes data, så som en Outlook pst-fil. Støjen i en billedfil er derimod notorisk højfrekvens i en sådan grad at billeder ofte ikke kan kodes på en brugbar måde.

Det er en vurdering fra data til data hvilken blokstørelse der er bedst, men det vil generelt være et godt udgangspunkt at vælge en bloklængde på 5% af datalængden. Bemærk dog at hvis blokkene kodes med intervalkoder, så vil en promille være et udmærket udgangspunkt. Det er dog noget man bør eksperiementere med. Mere om intervalkodning i 4.5

2.2 Beregning af hashfil for α

Den konkrete beregning af hashværdierne for α kan udføres med Rolling Adler 32, herfter benævnt RA32, og MD5. Implementationen af RA32 beskrives mens beskrivelsen af MD5 overlades til andre kilder eller API'er.

α opdels i n blokke af m bytes, hvor m er bloklængden. Hvis datalængden ikke er et multiplum af m så paddes data med værdien 0 op til næste blokgrænse.

Har man data=abcdef og vil opdele det i blokke af 4 bytes, så udvider man data til abcdef00 så datalængden er delelig med bloklængden.

For hver blok n i data beregnes
MD5(data[n*m+0..n*m+m-1]) og RA32(data[n*m+0..n*m+m-1]) og resultaterne lagres i en datastruktur, som kunne være en array, under index n.

Disse hashværdier for de enkelte blokke skal lagres som to strukturer som er henholdsvis den stærke og den svage hash for α. Sammen med hver værdi skal lagres index for den blok i α der gav netop den værdi. Det er disse hashværdier og blokindex som skal bruges ved beregning af Δ. Værdierne skal lagres på disk og de skal bruges i hukommelsen når beregningen skal udføres.

I praksis får man mange blokke og man skal i værste tilfælde søge gennem alle blokke fra α for hvert byteoffset i β. Det kan give mange søgninger og tage lang tid. Man bør derfor overveje en hurtig struktur at søge I og jeg vil foreslå en hashtabel med eksempelvis 1024 positioner til lagring i hukommelsen.

For en α med længden 104857600B der opdeles i blokke af 512000B får man 205 blokke hvor den sidste blok består af 409600B data og 102400B er tilføjede nuller.

En hashtabel med 1024 positioner vil have mindre end en blok pr. position og man kan forvente at finde en blok i konstant tid.

2.3 Søgning i hashliste

Når man har beregnet en hash for en blok i β og skal se om man kan genfinde blokken i α så søger man en forekomst af hashværdien i α's hashliste.

Hvis hashlisten er implementeret som en hashtabel med 1024 positioner (buckets), så søger man en given hash ved at beregner $index = hash\%1024$. Derefter foretager man en lineær søgning i denne bucket i hashtabellen.

2.4 Kodning af Δ

Som tidligere beskrevet så repræsenterer Δ ændringerne fra α til β og man søger at finde størst mulige blokke fra α i β for at kunne beskrive β ved en henvisning til
alpha. Man kan dog i praksis ikke regne med at finde hele β ebskrevet ved blokke i α. Der vil være ændringer som skal skrives i deres rå form, og blokstørelsen kan gøre at selv data, som findes i begge udgaver, ikke kan

repræsentered ved en blokreference.

Man skal derfor opbygge Δ af blokreferencer og rå data.

Når man genfinder en blok fra β i α så skriver man dens blokindex til Δ og når man ikke kan finde et blockmatch så skriver man første byte i den umatchede blok til Δ

Et eksempel kunne være en Δ på blok0,blok1,rå:54,rå:2,blok4 hvor værdierne antyder at β har sine to første blokke identisk med α og at tredie blok er ændret fra hvad den var i α til værdierne 54 og 2. Den sidste blok er uændret.

Mere vedr. den faktiske kodning i **??**

3 Implementation af datagendannelse

Givet Δ og α, samt kendskab til den benyttede bloklængde l, er det trivielt at gendanne β.

Man læser Δ og når man møder et blockmatch med et index i, så skriver man til β værdierne $\alpha[l*i..l*i+l-1]$. Møder man ikke et blockmatch men en rå byte, så skrives denne direkte til beta.

Herunder et simpelt eksempel med bloklængden 4.

α ="DETTE_ER_EN_TEST"
Δ =Blok0, E, _, V, A, R, blok2,blok3

$\beta = DETT + E + \_ + V + A + R + \_EN\_ + TEST$

4 Filformatet

Hvis man ikke kan repræsentere blokindex og rå data på en kompakt måde, så vil Δ blive for stor. Det er derfor vigtigt at pakke data godt i en binært format.

Formatet, som beskrives herunder, er kompakt, men kan gøres mere kompakt hvis nødvendigt. Der er generelt ikke meget at hente og formatet vil blive væsentligt mere komplekst at arbejde med. Kun de rå data vil kunne pakkes markant bedre ved at udføre en særskildt kompression af dem, men samme resultat vil kunne opnås ved at komprimere hele Δ efter at den er dannet.

Δ består af en sekvens af koder som kan repræsentere forskellige former for data. Grundlæggende set er der blokke af dataog rå data som står for sig selv.

Første kode i Δ bør være en angivelse af bloklængden og da den kun forekommer en gang i hver Δ kan man gøre formatet simpelt ved blot at definere at der altid bruges fire byte til at angive bloklængderne.

Som alternativ til at lagre bloklængden i Δ kan man have en fast bloklængde som altid bruges i enhver Δ, men det er ineffektivt for forskellige datatyper med forskellig form for og grad af støj.

4.1 Blokrepræsentation

Et blockindex skal indeholde information offset samt blokkens længde. Hvis alle blokke har sammen længde, som hidtil antaget, så skal kun information om offset lagres.

Hvis der datalængden aldrig overstiger 256*bloklængden så kan blokkens offset lagres i en byte. Hvis datalængden aldrig overstiger 65536*bloklængden så kan offset lagres i to bytes. Tilsvarende gælder for andre forhold mellem data- og bloklængde. Man ser at repræsentationen af offset kan sætte en grænse for datalængden givet en vis bloklængde og blokrepræsentation. Man ser også at for små datalængder med så blokoffset kan det være en fordel at repræsentere offset med få bytes, hvor man for større datalængder kan være nødt til at enten repræsentere et offset ved flere bytes eller øge bloklængden.

Man kan også vælge at have to eller endda flere bloktyper i sin Δ. Man kan vælge en bloktype der lagres som en byte, en anden som lagres som to bytes, en tredie der lagres som tre bytes etc. Man skal blot skelne mellem disse og ens skelen mellem dem bør repræsenteres i en form som er mere kompakt end at bruge største bloktype i hver situation.

Et foreslag til en repræsentation i en konkret Δ er følgende.

Der er tre bloktyper

- Blokke på 1 byte der repræsenterer blockoffset fra 0 til $2^8 - 1$
- Blokke på 2 bytes der repræsenterer blockoffset fra 2^8 til $2^{16} + 2^8 - 1$
- Blokke på 3 bytes der repræsenterer blockoffset fra $2^{16} + 2^8$ til $2^{24} + 2^{16} + 2^8 - 1$

Det giver et maksimalt blockindex på 16843007. Med en bloklængde på 1024B kan man repræsentere datalængder på op til 17247240192B svarende til 16GB.

Man anvender til hver en tid den mindste repræsentation der er mulig. Ønsker man at lagre index 120 så bruger man kun en byte. Ønsker man at lagre index 340 så bruger man to byte og ønsker man at lagre index 134000 så bruger man tre byte.

4.2 Rådata-repræsentation

De data som falder udenfor et blockmatch skal kodes i en rå form. Hvis en enkelt byte ikke kan placeres sammen med andre bytes i en blok, så skal denne enkelte byte skrives for sig selv.

I praksis forekommer det sjældent at bytes, som falder udenfor blokkene, står alene. Sådanne "rå bytes"kommer oftest i grupper hvis størelse afhænger af typen af støj mellem α og β samt størelsen af datafilerne.

Det er ineffektivt at kode en sekvens af rå bytes som enkelte enheder. Istedet bør man kode dem som en gruppe hvis indehold og længde man skal repræsentere for at kunne foretage en senere dekodning.

Kodningen af den rå bloks længde er genstand for præcis samme overvejelser som gælder for kodningen af en almindelig bloks offset. Hvis man forventer at disse sekvenser bliver store, så skal man kode længden med mange bit. Hvis man ved at de bliver små, så kan man spare plads ved at kode længden med færre bits.

En typisk blok med rådata vil bestå af en længdeindikator fulgt af den angivne antal rå bytes.

Bemærk at det værste der kan ske er at alle rå bytes forekommer alene. Man kan generelt ikke med nogen repræsentation angive en længde på 1 samt en rå byte på mindre end ialt 9 bit, hvilket er mere end 8 bit som data reelt fylder.

Der er to bloktyper til repræsentation af rå data og de næsten identiske med de to mindste bloktyperne for matchede blokke.

- Blokke på 1 byte der repræsenterer datalængder fra 1 til 2^8
- Blokke på 2 bytes der repræsenterer blockoffset fra $2^8 + 1$ til $2^{16} + 2^8$

Bloktypen på tre bytes anvendes ikke, da den som oftest er overflødig. Det sker sjældent at man har mere end 64KB rå data med mindre man har valgt en meget stor bloklængde, eller har lavet markante ændringer i data.

4.3 Hvilken slags blok er dette?

Hvis man naivt nøjes med blot at skriver sine blokke ned i en datafil, så er det ikke senere muligt at skelne mellem blokkene. Man kan ikke kende en blok fra en anden og ved ikke hvordan data skal tolkes.

Den simple løsning er at sætte en byte foran enhver blok så denne byte kan fungere som id der kendetegner den enkelte bloktype. Det koster imidlertid en del plads og kan gøres bedre.

Der er umiddelbart fem bloktyper: match1, match2, match3, rå1 og rå2. Fem typer kan repræsenteres i tre bit. En byte rummer 8 bit, så den kan identificere to blokke med hver en id på tre bit. Det efterlader to tomme bit da 8-3-3=2 og disse er, i første omgang, spildte.

Man kan skrive en id-byte for hver to blokke, og kode blokkenes type i de laveste 6 bit. Havde man kun fire bloktyper istedet for fem, hvilket man kan vælge at have, så kunne man kode bloktypen for tre blokke i en enkelt byte og undgå bitspild.

Formatet bliver nu : xxaaabbb,bloka,blokb,xxaaabbb,bloka,blokb,...

4.4 Id skrives før data, men id kendes først efter data

Det komplicerer kodningen en smule at udskrive et blokid før selve blokken, da man skal kende typen af den følgende blok før man kan skrive id og selve blokkene ud.

I praksis løses problemet ved at lave en buffered skrivning hvor ens kode på et højere niveau blot skriver data ned til den bufferede skriver. Denne gemmer data til den er klar til at skrive en blok, hvorefter den skriver blokkens type og selve blokken.

Har man et blockmatch som man gerne vil skrive, så skriver man det blok til bufferen som gemmer informationen, men ikke skriver noget ud. Har man derefter et match mere så skrives dette også til bufferen som nu har to blokke klar til udskrivning. Baseret på de pågældende offsetværdier skrives

et dobbelt id ud efterfulgt af de to blokke.

Har man en rå byte som man vil skrive så skrives denne også til bufferen som gemmer den. Kommer der efterfølgende hundrede rå bytes mere, så gemmes disse også i bufferen og skrives ikke ud. Når der kommer et blockmatch som skrives til bufferen, så er to blokke klar og id skrives fulgt af blokkene.

En anden måde en rå blok kan afsluttes er ved at den når den maksimale størelse på 2^{16} bytes. Hvis det sker så er *denne* rå blok færdig.

4.5 Intervalkodning

Underbloktyper for matchende blokke er der noget som ikke blev beskrevet. Der er en naturlig sammenhæng mellem blokoffset for blokke der følger hinanden. Det forekommer oftest at hvis to blockmatch følger hinanden i Δ, så er offset øget med en fra første til anden blok.

Det er en naturlig følge af at hvis man genfinder blok x et sted så vil man sædvanligvis enten derefter finde rå data eller blok x+1 og samme gælder i næste skridt.

Man kan derfor have en sekvens blok5,blok6,blok7,blok8,blok9,... som fylder en hel blokrepræsentation hver gang. En mere fornuftig kodning ville bære at skrive blokkene som blok5-9. Det kan kodes som et element istedetfor som 5.

Denne tendens til at lave længere sekvenser har vidtrækkende konsekvenser. Hvis man kan repræsentere hundrede blokke i en sekvens på samme plads om man kan repræsentere en enkelt blok, så er det pludselig ikke længere et problem at bruge små bloklængder. De fremgik ellers tidligere at små bloklængder medførte mange blockmatch hvilket fyldte meget. Det gælder naturligvis stadig, men det et et langt mindre problem, og når man tager i betragtning at små blokke er mere tilbøjelige til at finde match, så overskygger fordelene i høj grad ulemperne.

Man kan altså indføre en ny bloktype som indeholder et begyndelsesoffset samt en længde af sekvensen. Man kan også fortsætte eksemplet fra tidligere og lave en bloktype for hver længde sekvens og offset.

Typekodningen af blokke brugte før to gange tre bit og spildte to bit for hver byte. Hvis man holder fast i at kode typen af to blokke i en byte, så kan man uden problemer udvide antallet af typekoder op til $2^4 = 16$

Der eksisterer allerede to typekoder for rå data og tre typekoder for matchede blokke. Det giver 11 ledige koder, hvilket er mere end nok.

Følgende bloktyper kan anvendes

- To byte, offset fra 0 til $2^8 - 1$ og længde fra 1 til 2^8
- Tre byte, offset fra 0 til $2^8 - 1$ og længde fra $256 + 1$ til $2^{16} + 2^8$
- Tre byte, offset fra 2^8 til $2^{16} + 2^8$ og længde fra 1 til 2^8
- Fire byte, offset fra 2^8 til $2^{16} + 2^8$ og længde fra $256 + 1$ til $2^{16} + 2^8$

Flere kan tilføjes, da der er id-koder nok. De overskydende koder kunne anvendes til yderligere kodning ved at lade en byte fungere som id for eksempelvis 2.5 blokke, men man vil vinde meget lidt og det vil gøre koden væsentligt mere kompleks.

At ikke bryde intervallet Man skal være forsigtig ved søgningen efter blockmatch når man er ved at opbygge et interval. Man kan eksempelvis have fundet et blockmatch for blok 20 og man søger nu videre og finder hefter et blockmatch for blok 5. Det bryder intervallet, da man skulle have haft blok 21 for at fortsætte. Det kan være ærgeligt og uafhjælpeligt, men det er muligt at intervallet faktisk fortsætter selvom man fandt et match på blok 5 og ikke blok 21. Det man skal holde sig for øje, er at blok 5 og blok 21 sagtens kan være helt identiske. Der er dermed reelt et match med dem begge og man kan frit vælge hvilket match man ønsker. Man kan tage det første og bryde intervallet eller man kan tage det andet match og fortsætte opbygningen af intervallet. Det er derfor fornuftigt, ved søgning efter blockmatch, at se om den næste ønskede blok faktisk kan bruges. Hvis derfor man har matchet med blok 20 før så søger man først efter et match på blok 21. Hvis dette fejler så søger man efter et vilkårligt match og afslutter intervallet.

Hvad er det optimale intervalstart? Da man kan komme ud for at et blockmatch reelt er et match med flere mulige identiske blokke, så kan man risikere at vælge en blok som ikke passer ind i et interval, selvom en anden identisk blok passer.

Et eksempel kan være et datasæt so dette
α=AAA—BBC—AAA—QER—RTY

β=AAA—QER—RTY

Man søger første blok AAA fra β i α og har to mulige match. Både blok 0 og blok 2 matcher. Hvis man vælger blok 0 så vil ens interval starte med 0 og man kan fortsætte intervallet hvis næste blok matcher med blok 1 i α. Det er imidlertid ikke tilfældet, da kun blok 3 matcher. intervallet er dermed brudt før det startede, og man må kode data som en enkelt blok. Havde man valgt blok 2 fra starten så havde intervallet fortsat med blok 3 og blok 4 og kunne kodes som et reelt interval.

Ovenstående "problem"kan kun løses ved, i sin buffer, altid at huske alle de mulige blokke og altid se hvilket interval der er længst og arbejde videre med dette. Det vil give bedre kompression, men det vil være væsentligt mere komplekst at kode.

Man bør derfor acceptere denne mulighed for suboptimalt resultat og blot sørgre for at ikke bryde et påbegyndt interval før det er nødvendigt.

5 Uændrede filer

Hvis der ingen ændring er fra α til β så vil det være tidskrævende og pladsmæssigt ineffektivt at kode β differentielt.

Hvis man har en bloklængde på δ så skal man forvente at kode i gennsmsnit $length(\beta) modulus \delta$ rp bytes, da de ikke passer med en blokgrænse i slutningen. Dette gælder for helt identiske filer.

En alternativ metode kunne være at kode en stærk hash for hele α som lagres sammen med den stærke og den svage hash for hver blok. Man kan dermed, før den differentielle kodning, beregne en stærk hash af hele β og sammenligne denne med α's stærke hash. Er de identiske så er filerne, med stor sansynlighed, også identiske og man kan lave en Δ som kun består af et symbol der skal tolkes som $\beta = \alpha$.

Dette symbol kan lagres som en enkelt type-byte.

6 Et regneeksempel

Eksemplet løb lidt løbsk da jeg ville illustrere flere pointer. Det er ikke noget man skal ønske at regne manuelt :-/

Dette eksempel behandler to små datablokke med få ændringer. Den optimerede kodeform, som beskrevet i afsnittet om filformatet, anvendes ikke, da den er irrelevant for eksemplet. Rå data kodes derfor som enkeltbytes med R foran og matchede blokke kodes ikke som intervaller men som enkeltblokke med B foran deres offset.

α=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24
β=0,1,2,3,4,5,7,5,9,10,11,12,13,14,15,16,17,18,19,20

En bloklængde på 4 vælges og den stærke og svage hash beregnes for hver blok i α. Som stærk hashfunktion benyttes 1 og som svag hashfunktion benyttes 2

summen af bytes i blokken og som svag hash bruges den stærke hash modulus 10. Disse to hashfunktioner er meget simple og skal blot illustrere. Den stærke hash har også den rullende egenskab, men det er tilfældigt og gælder ikke i praksis ved valg af eksempelvis MD5. Bemærk at den stærke hash har en bitlængde på 32 bit hvilket også gælde de data der hashes. Denne hash er altså sikret mod kollisioner. Udtrykket for at beregne den rullede hash for den svage hashfunktion er givet i 3

$$SH(a,b,c,d) = a * 256^3 + b * 256^2 + c * 256^1 + d * 256^0 \quad (1)$$

$$VH(a,b,c,d) = (a * 256^3 + b * 256^2 + c * 256^1 + d * 256^0)\%10 \quad (2)$$

$$VH(VH(a,b,c,d),e) = (VH(a,b,c,d) * 256 - (a * 256^4)\%10 + e)\%10 \quad (3)$$

Længden af α er ikke et multiplum af bloklængden, så hashen af den sidste blok bliver beregnet med tilføjede nuller. De beregnede hash er som følger.

Index	Data	Stærk	Svag
0	0,1,2,3	66051	1
1	4,5,6,7	67438087	7
2	8,9,10,11	134810123	3
3	12,13,14,15	202182159	9
4	16,17,18,19	269554195	5
5	20,21,22,23	336926231	1
6	24,0,0,0	16777216	4

Tabellen over hashværdierne i α viser at der ingen kollisioner er i de stærke hash, da det ikke eer muligt, mens der faktisk er en en enkelt kollision i den svage hash.

Kodningen af β Skridt for skridt gennemgår kodningen herunder.

1. Betragt blokken på offset 0 i β som er 0,1,2,3
2. Beregn den svage hash af data, $VH(0,1,2,3) = 1$
3. Søg i listen over α's svage hash hvor VH=1 findes for blok 0 og blok 5.
4. Beregn den stærke hash af data, SH(1,2,3,4)=66051.
5. Se om den stærke hash af blok 0 i α er 66051. Det er den, så der er et match.
6. Skriv til Δ B0, Δ er nu [B0].
7. Flyt søgevinduet i β en bloklængde frem.
8. Betragt blokken på offset 4 i β som er 4,5,7,5
9. Beregn den svage hash af data, $VH(4,5,7,5) = 1$
10. Søg i listen over α's svage hash hvor VH=1 findes for blok 0 og blok 5.
11. Beregn den stærke hash af data, SH(4,5,7,5)=67438341.
12. Se om den stærke hash af blok 0 i α er 67438341. Det er den ikke, så der er et intet match.
13. Se om den stærke hash af blok 5 i α er 67438341. Det er den ikke, så der er et intet match.

14. Skriv til Δ R4, Δ er nu [B0,R4].
15. Flyt søgevinduet i β en byte frem.
16. Betragt blokken på offset 5 i β som er 5,7,5,9
17. Beregn den svage hash af data, $VH(5,7,5,9) = 1$
18. Søg i listen over α's svage hash hvor VH=1 findes for blok 0 og blok 5.
19. Beregn den stærke hash af data, SH(5,7,5,9)=84346121.
20. Se om den stærke hash af blok 0 i α er 84346121. Det er den ikke, så der er et intet match.
21. Se om den stærke hash af blok 5 i α er 84346121. Det er den ikke, så der er et intet match.
22. Skriv til Δ R5, Δ er nu [B0,R4,R5].
23. Flyt søgevinduet i β en byte frem.
24. Betragt blokken på offset 6 i β som er 7,5,9,10
25. Beregn den svage hash af data, $VH(7,5,9,10) = 6$
26. Søg i listen over α's svage hash hvor VH=6 ikke findes og der intet match er.
27. Skriv til Δ R7, Δ er nu [B0,R4,R5,R7].
28. Flyt søgevinduet i β en byte frem.
29. Betragt blokken på offset 7 i β som er 5,9,10,11
30. Beregn den svage hash af data, $VH(5,9,10,11) = 5$
31. Søg i listen over α's svage hash hvor VH=5 findes for blok 4.
32. Beregn den stærke hash af data, SH(5,9,10,11)=84478475.
33. Se om den stærke hash af blok 4 i α er 84478475. Det er den ikke, så der er et intet match.
34. Skriv til Δ R5, Δ er nu [B0,R4,R5,R7,R5].
35. Flyt søgevinduet i β en byte frem.

36. Betragt blokken på offset 8 i β som er 9,10,11,12
37. Beregn den svage hash af data, $VH(9, 10, 11, 12) = 2$
38. Søg i listen over α's svage hash hvor VH=2 ikke findes og der er intet match.
39. Skriv til Δ R9, Δ er nu [B0,R4,R5,R7,R5,R9].
40. Flyt søgevinduet i β en byte frem.
41. Betragt blokken på offset 9 i β som er 10,11,12,13
42. Beregn den svage hash af data, $VH(10, 11, 12, 13) = 1$
43. Søg i listen over α's svage hash hvor VH=1 findes for blok 0 og blok 5.
44. Beregn den stærke hash af data, SH(10,11,12,13)=168496141.
45. Se om den stærke hash af blok 0 i α er 168496141. Det er den ikke, så der er et intet match.
46. Se om den stærke hash af blok 5 i α er 168496141. Det er den ikke, så der er et intet match.
47. Skriv til Δ R10, Δ er nu [B0,R4,R5,R7,R5,R9,R10].
48. Flyt søgevinduet i β en byte frem.
49. Betragt blokken på offset 10 i β som er 11,12,13,14
50. Beregn den svage hash af data, $VH(11, 12, 13, 14) = 0$
51. Søg i listen over α's svage hash hvor VH=1 ikke findes og der er intet match.
52. Skriv til Δ R11, Δ er nu [B0,R4,R5,R7,R5,R9,R10,R11].
53. Flyt søgevinduet i β en byte frem.
54. Betragt blokken på offset 11 i β som er 12,13,14,15
55. Beregn den svage hash af data, $VH(11, 12, 13, 14) = 9$
56. Søg i listen over α's svage hash hvor VH=9 findes for blok 3.
57. Beregn den stærke hash af data, SH(11,12,13,14)=202182159.

58. Se om den stærke hash af blok 3 i α er 202182159. Det er den så der er et match.

59. Skriv til Δ B3, Δ er nu [B0,R4,R5,R7,R5,R9,R10,R11,B3].

60. Flyt søgevinduet i β en bloklængde frem.

61. Betragt blokken på offset 15 i β som er 16,17,18,19

62. Beregn den svage hash af data, $VH(16, 17, 18, 19) = 5$

63. Søg i listen over α's svage hash hvor VH=5 findes for blok 4.

64. Beregn den stærke hash af data, SH(16,17,18,19)=269554195.

65. Se om den stærke hash af blok 4 i α er 269554195. Det er den så der er et match.

66. Skriv til Δ B4, Δ er nu [B0,R4,R5,R7,R5,R9,R10,R11,B3,B4].

67. Flyt søgevinduet i β en bloklængde frem.

68. Betragt blokken på offset 19 i β som er 20

69. Denne test af β er kun en byte stor og kan selvsagt ikke matche en blok, så den kodes direkte i rå form.

70. Skriv til Δ R20, Δ er nu [B0,R4,R5,R7,R5,R9,R10,R11,B3,B4,R20].

Efter denne længere smørre har man at Δ=B0,R4,R5,R7,R5,R9,R10,R11,B3,B4,R20

Da det er et tænkt eksempel, med meget små blok- og datalængder, er Δ ikke væsentligt mindre end β.

Det fremgår af eksemplet hvad problemet er med lange bloklængder, da man ser at efter de nye data der fulgte blok 0, da tog det lidt tid før systemet fandt ind i blokrækken igen. Havde bloklængden været større, så havde det taget endnu længere tid.

Man ser også hvordan α zero paddes for at kunne beregne hash for den sidste del selvom denne ikke passer med en blokgrænse.

Endelig ser man at den rest af β, som er mindre end en blok, må kodes i rå form, da man ikke kan finde en blok der matcher eftersom størelsen selvsagt ikke passer.

A Rolling Adler 32

Den svage rullende checksum er defineret således. X er data , s er startindex for data der skal hashes og l er længden af den datablok der skal hashes. For en blok på 5 byte som er 3,4,5,6,7 ,som skal hashes i blokke af tre bytes startende fra offset 1, vil X være 3,4,5,6,7, s er 1 og l er 3.

De sidste to udtryk viser hvordan hashen af data med længden l på position s+1 kan beregnes ud fra hashen af data på længden l på position s. Den rullende egenskab, som er baseret på samme princip som beskrevet i detaljer i 1.7

$$a(X, s, l) = (\sum_{i=s}^{s+l} X_i)\%2^{16} \tag{4}$$

$$b(X, s, l) = (\sum_{i=s}^{s+l} (s + l - i + 1)X_i)\%2^{16} \tag{5}$$

$$H(X, l) = a(X, l) + b(X, l) * 2^{16} \tag{6}$$

$$a(X, s + 1, l) = (a(X, s, l) - X_s + X_{s+l+1}\&2^{16} \tag{7}$$

$$b(X, s + 1, l) = (b(X, s, l) - (s + 1) * X_s + a(X, s + 1, l))\%2^{16} \tag{8}$$

B Kildekode

Foreløbigt fjernet.

B.1 Rolling Adler 32

En klasse der implementerer en svag checksum i c#

B.2 Differential coder

En klasse der implementerer en simpel differentiel koder.

B.3 Buffered Δ writer

En klasse der implementerer visse af de beskrevne metoder til differentiel kodning, men ikke alle, og ikke nødvendigvis på helt samme måde som beskrevet.

B.4 Hashcollection

En klasse der fungerer som en hashtabel med ekstra funktionalitet. Den kan eksempelvis forsøge at finde bestemte blokke med bestemte index når man er ved at opbygge et interval.