Compressie van XML-documenten

Kim Jenny RAMAEKERS

promotor :
Prof. dr. Frank NEVEN

Eindverhandeling voorgedragen tot het bekomen van de graad
Licentiaat in de Informatica,
afstudeervariant Databases

universiteit hasselt

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Abstract

Deze scriptie behandelt drie grote delen.

In het eerste hoofdstuk wordt er een overzicht gegeven van de belangrijkste compressietechnieken op tekstbestanden zoals onder andere de Huffman (zie Sectie 1.2) en de Lempel-Ziv-Welch codering (zie Sectie 1.4). De bekende tools (zie Sectie 1.8) die gebruik maken van deze compressie-algoritmes worden ook kort toegelicht.

In het tweede hoofdstuk zijn de algoritmes en de programma's die speciaal ontwikkeld zijn om XML-bestanden te comprimeren aan de beurt.

In het derde hoofdstuk worden de XML compressie-tools met elkaar vergeleken. Er wordt ook rekening gehouden met de uitvoerbare queries op bestanden die volledig of gedeeltelijk moeten gedecomprimeerd worden.

Woord vooraf

Net zoals elk willekeurig bestand kun je XML-documenten ook comprimeren. Aangezien een XML-document louter uit ASCII bestaat, kun je elke compressietechniek voor tekstbestanden gebruiken. Er bestaan echter tools, zoals XMill, die gebruik maken van aangepaste algoritmes toegespitst op XML. Er zijn ook technieken die XML-bomen als DAGs (directed acyclic graphs) opslaan door gemeenschappelijke subbomen te laten samenvallen.

De grote uitdaging is echter om queries op gecomprimeerde XML-documenten te kunnen uitvoeren zonder het XML-document volledig te decomprimeren. Uiteraard is dit sterk afhankelijk van de gebruikte compressietechniek en het soort queries die geëvalueerd worden.

Deze thesis bestaat erin een overzicht te geven van compressietechnieken voor XML en de algoritmes in de literatuur voor het evalueren van queries over gecomprimeerde XML-documenten met elkaar te vergelijken.

Hoofdstuk 1

Overzicht van compressie-technieken

XML-bestanden bestaan enkel uit ASCII-waarden, waardoor XML gehanteerd kan worden als een tekstbestand. Alle compressietechnieken voor tekstbestanden kunnen dus gebruikt worden op XML-documenten.

In dit hoofdstuk gaan we enkele belangrijke algoritmes en tools voor tekstcompressie bekijken.

1.1 Tekstcompressie

Tekstcompressie is een techniek die op computersystemen wordt toegepast om data te comprimeren, zodat deze bijvoorbeeld minder opslagcapasiteit inneemt of in een kortere tijdspanne verzonden kan worden. Bestanden kunnen immers vlug relatief omvangrijk worden en vermits geheugenruimte op elke machine een kostbaar goed is, wensen we een manier te vinden om zulke grote bestanden te reduceren en dit zonder verlies aan informatie.

Een vereiste is dat de compressor 'lossless' moet zijn. Dit wil zeggen dat de data zodanig moet samengepakt worden zodat een decompressor nog in staat is om de gecomprimeerde data weer in hun oorspronkelijke vorm te herstellen zonder dat daarbij enige mate van vervorming optreedt.

Een andere vorm van datacompressie is 'lossy' compressie waarbij er een geringe vervorming is tussen de gecomprimeerde en de originele data. Deze methode wordt bijna uitsluitend gebruikt voor informatie die te maken heeft met de menselijke gewaarwording: het menselijke oor en oog zijn niet perfect, maar ze zijn ongevoelig voor bepaalde vervormingen. Dit kan uitgebuit worden bij gecomprimeerde data waarbij een versterking van de compressie mogelijk is. Voorbeelden hiervan zijn JPEG voor elektronische beeldcom-

pressie en MP3 voor compressie van digitale geluidsbestanden.

In tekstbestanden komen sommige letters veel vaker voor dan andere. Een compressiemethode is daarom om unieke lettercoderingen van verschillende bitlengte te kiezen, waarbij de meest voorkomende letters de kortste codes krijgen toegekend. Bij de keuze van het compressie-algoritme is het dan ook interessant om de aard van het te comprimeren bestand te kennen.

Gewone tekst is met optimale technieken exact omkeerbaar te comprimeren tot circa 25% à 30% van de oorspronkelijke grootte. Vaak moet er een optimaal evenwicht gevonden worden tussen de mogelijke compressie-ratio, de compressie-tijd en de nodige opslagruimte, waarbij de compressie-snelheid kan reduceren.

Bij een grote hoeveelheid tekst kunnen lange woorden en zinsdelen vervangen worden door een kortere code. Als deze techniek wordt toegepast wordt de compressie beter naarmate er meer tekst is. Ook kan er gebruik gemaakt worden van een standaardbibliotheek bestaande uit woorden, waardoor enkel de speciale code nog ruimte inneemt.

We kunnen tevens een onderscheid maken tussen twee types van codering, namelijk individuele karakters (vb. Huffman en arithmetic -codering) of strings van karakters (vb. LZW).

We zullen in de volgende secties een kort overzicht geven van enkele belangrijke coderingen voor tekstbestanden en tools. Eerst lichten we het het begrip 'entropie' toe.

1.1.1 Entropie

Compressie van tekstbestanden bestaat uit de eliminatie van redundante informatie. Met het begrip entropie drukte de wiskundige Shannon in de jaren dertig uit hoe efficiënt een boodschap gecodeerd kon worden om nog als boodschap herkend te worden.

Een voorbeeld: de zin 'haal je de aardappelen wel op tijd van het vuur' zou je kunnen samendrukken tot 'háljedeárdapelenweloptydvhvür'. Deze tekst heeft een omvang van 72% van de originele tekst. De ingewikkeldheid van het programma dat deze samendrukking oplevert, of de maat voor ongeordendheid in een systeem, noemde Shannon de 'entropie'.

Hoge entropie komt overeen met een sterke wanorde en een geringe mate van voorspelbaarheid van gebeurtenissen. Voor lage entropie geldt het tegengestelde.

Voor een informatiebron met alfabet $S = s_1, s_2, \ldots, s_n$ is de entropie:

$$H(S) = \eta = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

p_i is de kans op voorkomen van s_i en $\log_2 \frac{1}{p_i}$ geeft de hoeveelheid informatie van s_i, zoals het aantal bits nodig om s_i te coderen.

Voorbeeld

Beschouw de volgende string:

$$rrr \sqcup ssss \sqcup ttttt \sqcup uuuuuu$$

Als we op deze string het Huffman algoritme toepassen, bekomen we Tabel 1.1. De entropie $\eta = 2.267$. Het minimum aantal bits om deze string te coderen is 48 bits. Als we het aantal bits nu effectief tellen, komen we aan 48 bits. Voor dit voorbeeld is het Huffman algoritme dus optimaal.

Symbool	**Probabiliteit**	**Codewoord**
r	3/21	100
s	4/21	00
t	5/21	01
u	6/21	11
$\sqcup$	3/21	101

Tabel 1.1: Huffmantabel

1.2 Huffmancodering

De Huffmancodering is genoemd naar David Huffman die de codering in 1952 voor het eerst beschreef. Het is een manier om gegevens die bestaan uit een rij van symbolen optimaal en zonder verlies aan data te comprimeren.

Het principe is eenvoudig: een gegeven reeks symbolen waarvan sommige meer kans hebben voor te komen dan andere, is het mogelijk om de volledige rij op een kortere manier te schrijven. Een kortere code kan toegekend worden aan de symbolen die meer voorkomen dan de andere.

In de meeste gevallen laat deze techniek toe om de grootte van een bestand te reduceren met gemiddeld 25 tot zelfs 50 à 60 procent. Hierbij wordt gebruikt gemaakt van het feit dat er bij grote bestanden een typisch groot onderscheid bestaat tussen de frequenties waarin de verschillende karakters voorkomen.

De computers coderen over het algemeen karakters gebruik makend van de standaard ASCII tabel, die een code met acht bits aan elk symbool toewijst. Zoals bijvoorbeeld de letter *a* een ASCII waarde van 97 heeft en als 01100001 binair gecodeerd wordt. De karakters die vaker voorkomen worden hetzelfde behandeld als de meer zeldzame karakters, zoals ü. Een bestand dat 100 karakters heeft, zal 800 bits vereisen. Deze waarde is onveranderbaar, ook al bestaat het bestand uit 100 unieke karakters of zijn het 100 voorkomens van hetzelfde karakter. De voordelen van het coderen volgens de ASCII waarden is dat de grenzen tussen karakters gemakkelijk worden bepaald en dat er een 8 bit code gebruikt wordt bij elk karakter. Deze is vast en universeel gekend.

Nochtans, in bijna om het even welke tekst, komen sommige karakters meer frequent voor dan anderen. Zou het dan ook niet meest logisch zijn om kortere bitcodes aan frequentere karakters toe te wijzen dan aan de mindere frequente? Dit idee is niet nieuw. Een eerder gezien voorbeeld van gegevenscompressie is de code van Morse die door Samuel Morse in de negentiende eeuw werd ontwikkeld. De letters, die door de telegraaf verzonden werden, werden gecodeerd met punten en streepjes. Morse merkte op dat bepaalde letters vaker dan andere voorkwamen. Om de gemiddelde tijd te verminderen, die werd vereist om een bericht te verzenden, wees hij kortere series toe aan letters die vaker voorkwamen. Zoals bijvoorbeeld *e* als (.) en *a* als (.-) en bij langere series van letters die minder vaak voorkwamen, zoals bij *q* als (- -.-) en bij *j* als (. - - -).

Dit idee, van het gebruik van kortere codes voor vaker voorkomende karakters, werd reeds in acht genomen door Claude Shannon en R.M. Fano in de jaren '50 bij het ontwikkelen van het Shannon-Fano compressie-algoritme (zie Sectie 1.3). Nochtans, D.A. Huffman publiceerde een document in 1952

dat het algoritme in kleine mate verbeterde.

Wanneer we de techniek van een variabele codelengte toepassen, kunnen we karakters met hoge graad van voorkomen een kortere code toekennen dan karakters die beduidend minder voorkomen. Op deze manier is het mogelijk om met een nieuwe code het totaal aantal bits van het bestand sterk te reduceren. Uiteraard is het ook onmiddellijk duidelijk dat, wanneer alle karakters ongeveer evenveel voorkomen, het principe van Huffman niet opgaat en het algoritme in dit geval niet geschikt is als compressiemethode.

We dienen dus een algemene methode te vinden om aan de verschillende karakters een nieuwe binaire code toe te kennen, rekening houdend met het feit dat veel voorkomende karakters een kortere code toegekend dienen te krijgen dan minder voorkomende karakters.

Een eerste bedenking die we hier kunnen maken is het feit dat bij karakters met verschillende codelengte intern geen onderscheid meer gemaakt wordt tussen de verschillende karakters. We zouden dus nood hebben aan een neutraal scheidingskarakter die het einde van een karaktercode aangeeft. Indien echter onze nieuwe code de prefix-eigenschap zou hebben, wat betekent dat geen enkel codewoord de prefix is van een ander codewoord, zou bovenstaand probleem onbestaande zijn.

Om dit te verwezenlijken, creëert de Huffman codering een binaire boom, de zogenaamde Huffmanboom. Elk karakter kan hiermee voorgesteld worden als een blad in de binaire boom. Nu kunnen we eveneens aan elk karakter in deze boom een unieke code toekennen. Start hiervoor bij de wortel van de boom en volg het pad naar het blad waar het te coderen karakter zich bevindt. Een binaire '0' staat voor een linkse afslag in dit pad, terwijl de '1' staat voor de rechtse afslag. Merk op dat de code die op deze manier voor elk karakter geconstrueerd wordt uniek is, vermits er nooit meerdere karakters in eenzelfde blad kunnen voorkomen en in elke binaire boom een uniek pad naar een bepaald blad bestaat.

Tevens is het een feit dat alle karakters in bladeren vervat zijn, wat met zich meebrengt dat geen enkele karaktercode prefix is van een andere code. De reden hiervoor is dat, volgens de boomstructuur van het XML-document, een blad nooit aanwezig kan zijn op een pad naar een ander blad. Op deze manier is voldaan aan de hierboven gedefinieerde prefix eigenschap zodat dubbelzinnigheid tijdens het decoderen onmogelijk is, ondanks het feit dat de karaktercodes kunnen verschillen in lengte.

1.2.1 Het algoritme van Huffman

De vraag die we ons stellen is op welke manier men de karakters moet schikken in een binaire boom zodat de nieuwe codering van elk karakter kan resulteren in een optimale compressie.

Veronderstel dat het aantal karakters dat we wensen te coderen gelijk is aan x. Volgende stappen dienen dan gevolgd te worden om de gewenste boom op te bouwen:

1. Zet alle symbolen in een lijst, gesorteerd in aflopende volgorde van frequentie.
2. Herhaal totdat de lijst nog één symbool bevat:
   (a) Neem twee symbolen uit de lijst met de laagste frequentie.
   (b) Maak een subboom aan met deze twee symbolen als bladeren en maak een nieuwe ouderknoop.
   (c) Geef de som van de frequenties van de kinderen aan de ouderknoop en voeg de bekomen waarde gesorteerd toe aan de lijst.
   (d) Verwijder de kinderen uit de lijst.
3. Ken een codewoord toe aan elk symbool, vertrekkend van de wortel naar het blad.

1.2.2 Voorbeeld

Gegeven de volgende string:

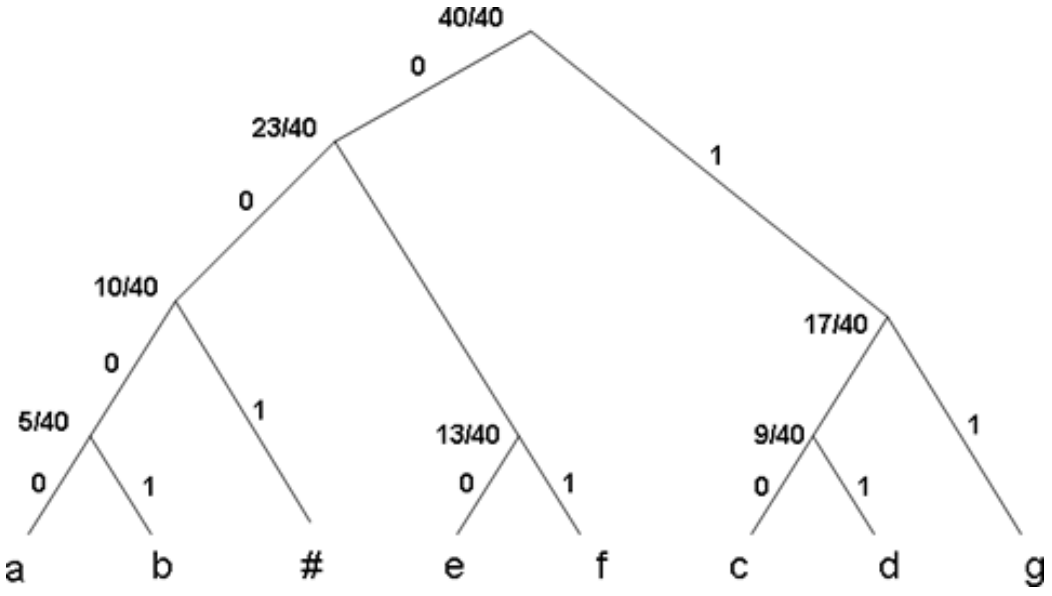
$$aa\#bbb\#cccc\#ddddd\#eeeeee\#fffffffggggggggg$$

Deze string telt 40 karakters. De e komt zes keer voor, wat een waarschijnlijkheid van 6/40 geeft, enz. .... Deze frequenties zet je in de Huffmantabel. Dit resulteert in Tabel 1.2.
Hierna passen we het algoritme van Huffman toe zoals deze boven gedefinieerd is. In de tabel zijn ook de codewoorden opgenomen die we kunnen samenstellen uit de codeerboom van Figuur 1.1.

Na het opbouwen van de Huffmanboom kunnen we starten met coderen. Dit gebeurt door de tekst een tweede maal te doorlopen en voor elk karakter de bijbehorende code af te leiden uit de opgestelde Huffmanboom. De code van een letter vind je door de nullen en enen te concateneren bij het doorlopen van het pad gaande van het blad naar de wortel.

Symbool	**Waarschijnlijkheid**	**Codering**
a	2/40	0000
b	3/40	0001
c	4/40	100
d	5/40	101
#	5/40	001
e	6/40	010
f	7/40	011
g	8/40	11

Tabel 1.2: Huffmantabel voor de voorbeeldstring



Figuur 1.1: Huffmanboom voor de voorbeeldstring

Een nadeel van de Huffmancodering is dat we het bestand twee keer moeten doorlopen: eenmaal om de frequenties van de karakters te tellen en eenmaal om de codering uit te voeren.

Een ander probleem bij het decoderen is dat de codewoorden bij de ontvanger gekend moeten zijn. Een mogelijkheid zou kunnen zijn deze mee te sturen vóór de data, wat resulteert in extra overhead bij het versturen. De codewoorden kunnen ook op voorhand gekend zijn, indien deze bijvoorbeeld gebaseerd zijn op statistische eigenschappen van de taal.

Een betere aanpak voor willekeurige data is echter dynamische of adaptieve Huffman codering. Men spreekt ook wel van het FGK-algoritme. Hieraan zijn de namen van Faller, Gallager en Knuth verbonden.

Bij gebruik van deze methode wordt de boom on-the-fly opgebouwd, dit zowel bij de zender als bij de ontvanger. Bij een eerste optreden van een karakter wordt dit doorgestuurd. Met andere woorden worden aan alle mogelijk voorkomende karakters gewicht één toegekend en wordt hierna de bijbehorende Huffmanboom opgesteld.

Na dit proces worden de karakters ingelezen. Bij herhaling van karakters wordt eerst het aantal aangepast, hierna wordt de boom geherstructureerd en vervolgens wordt de code aangepast.

1.3 Shannon-Fano codering

De Shannon-Fano codering techniek is de voorganger van de Huffman codering en gebruikt een gelijkaardig algoritme. De fundamentele principes zijn hetzelfde: de codes voor frequentere karakters zijn korter dan die voor minder frequente karakters, en het prefixprincipe is nog van toepassing. Nochtans, aangezien het coderen volgens Huffman een bottom-up methode is om de bitcodes voor elk symbool te bepalen, gebruiken Shannon en Fano een top-down methode.

Zoals eerder werd beschreven is de Shannon-Fano codering gebaseerd op codewoorden van variabele lengte. Dit betekent dat sommige bronboodschappen worden voorgesteld door kortere codewoorden dan andere bronboodschappen. Het criterium voor het schatten van de lengte van een codewoord is de probabiliteit van voorkomen van de bronboodschap. Hoe hoger de probabiliteit, hoe korter het codewoord. Hoewel Shannon-Fano codewoorden produceert van variabele lengte, zorgt deze methode er toch voor dat een serie van codewoorden uniek decodeerbaar is.

1.3.1 Algoritme

Het algoritme voor het leveren van de codewoorden is:

1. Sorteer de lijst van symbolen volgens zijn frequenties in dalende volgorde.

2. Verdeel de lijst van probabiliteiten in twee groepen, zodat beide groepen een (bijna) gelijke som hebben van de probabiliteiten.

3. De eerste helft van de lijst krijgt het binair cijfer '0' toegewezen en de andere helft krijgt cijfer '1' toegewezen. Dit betekent dat het codewoord voor de symbolen in de eerste helft allen met '0' zullen beginnen en het codewoord in de andere helft allen met '1'.

4. Pas recursief dezelfde procedure toe op beide helften. Blijf de groepen onderverdelen en voeg bits aan de codes toe totdat elk symbool een overeenkomstig codeblad op de boom heeft.

Merk op dat de lengte van elk codewoord i gelijk is aan $-\log_2 p_i$, met p_i de probabiliteit van voorkomen van i. Dit is van toepassing zolang het mogelijk is om de lijst in twee groepen te verdelen met gelijke probabiliteit. Als dit niet meer mogelijk is kunnen sommige codewoorden een lengte hebben van $-\log_2 p_i + 2$.

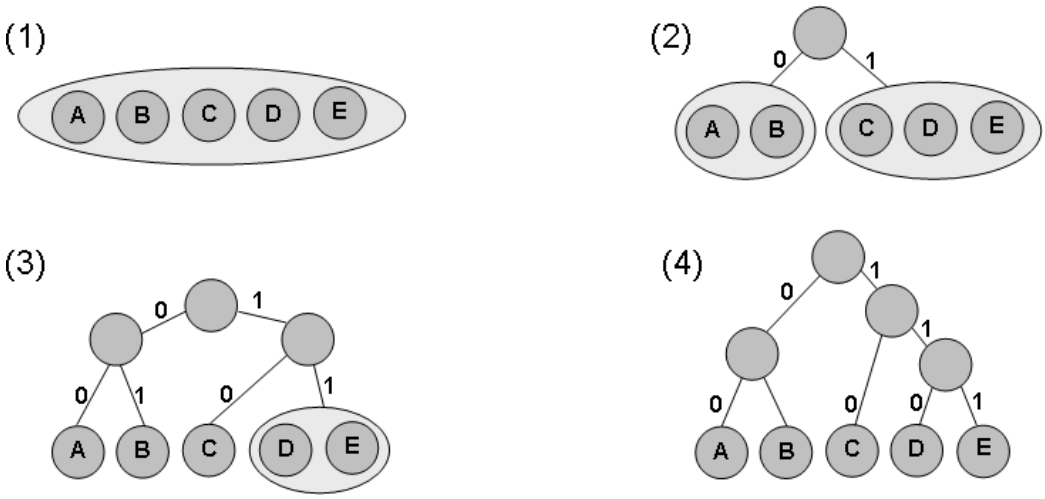
Het Shannon-Fano algoritme levert een gemiddelde lengte S van codewoorden die voldoen aan $H \leq S \leq H + 1$, met H de entropie gegeven door $\eta = H(S) = \sum_{i=1}^{n} -p_i \log_2 p_i$. Met andere woorden is er bij kleinere entropie een kleiner aantal bits nodig voor de codes.

1.3.2 Voorbeeld

We illustreren dit aan de hand van volgende symbolen: A, B, C, D en E samen met zijn frequenties (zie Tabel 1.3). De codewoorden zijn ook opgenomen in de tabel.

Symbool	**Frequentie**	**Codewoord**
A	15	00
B	7	01
C	6	10
D	6	110
E	5	111

Tabel 1.3: Tabel bij het Shannon-Fano voorbeeld



Figuur 1.2: Het algoritme van Shannon-Fano toegepast op het voorbeeld

De binaire boom, die we verkrijgen uit het toepassen van het algoritme van Shannon-Fano op ons voorbeeld, wordt weergegeven in Figuur 1.2.

Bij de eerste verdeling verkrijgen we twee delen waarbij de eerste helft bestaat uit de symbolen A en B met als frequentie 21, en de tweede helft

bestaat uit C, D en E met als frequentie 14. We hebben gekozen voor deze verdeling omdat hierbij de frequenties het dichts bij de helft (17.5) liggen.

Bij het recursief toepassen van het algoritme krijgen we een tweede verdeling. De eerste helft wordt verdeeld in twee delen, zodat de symbolen A en B elk een blad vormen en zo een codewoord toegekend krijgen. In de tweede helft wordt er gesplitst tussen C en D, zodat symbool C een blad vormt en een code krijgt. Bij de derde, en tevens laatste verdeling, worden symbolen D en E een blad en krijgen ook deze hun codewoord.

Het voordeel van de Shannon-Fano codering is dat de woordlengten meteen volgen uit de waarschijnlijkheden. Tegenwoordig worden echter slimmere coderingsmethoden gebruikt zoals de optimale Huffman codering.

1.4 Lempel-Ziv-Welch codering (LZW)

Het LZW algoritme is een exact omkeerbaar compressie-algoritme dat door de heren Abraham Lempel, Jacob Ziv en Terry Welch is ontwikkeld in 1977. Het principe is gebaseerd op het zoeken naar overeenkomsten tussen de karakters, zoals witruimtes en spaties, welke vervolgens omgezet worden in kortere codes.

Vele bestanden, vooral tekstbestanden, bevatten bepaalde strings die vaak voorkomen, zoals het lidwoord 'het'. Inclusief de spaties neemt de string vijf bytes of 40 bits in beslag om te coderen. Maar wat als we de volledige string toevoegen aan de lijst van enkele karakters na het laatste karakter, bijvoorbeeld code 256? Dan kan men telkens wanneer we de string 'het' tegenkomen, de code 256 doorsturen, in de plaats van 32, 104, 101, 116, 32. Dit zou slechts negen bits innemen (256 decimaal = 1 0000 0000 binair).

Het LZW-algoritme gebruikt een zogenaamd woordenboek om het bestand te comprimeren. In tegenstelling tot de Huffman-compressie, waar de tekens afzonderlijk worden gehercodeerd, gaat dit algoritme een lijst bijhouden, namelijk een woordenboek van strings. Het algoritme begint met een woordenboek van 256 tekens, namelijk de ASCII-tabel. Tijdens het comprimeren zal deze tabel aangroeien met alle strings die frequent in het bestand voorkomen. Zo ontstaat er echter een probleem: hoe kan men, bij meer dan 256 tekens, deze allemaal coderen in acht bits? Dit probleem wordt gemakkelijk opgelost door vanaf index 256 een code met negen bits te gebruiken. Vanaf index 512 tien bits, vanaf 1024 elf bits en zo verder met alle indices die een macht van twee zijn.

Bij het LZW-algoritme is het niet nodig een woordenboek toe te voegen aan het gecomprimeerde bestand. In elk systeem zit een ASCII-tabel verwerkt en dit is het enige dat nodig is om het gegeven bestand te kunnen decomprimeren. Het algoritme om te comprimeren wordt eigenlijk omgekeerd uitgevoerd. Er worden eerst twee bytes codes ingelezen. De eerste code komt uit de ASCII-tabel dat ons in staat stelt om onmiddellijk te decoderen.

1.4.1 Het LZW compressie-algoritme

Het algoritme voor het leveren van de codewoorden in pseude-code is:

```
s = next input character;
while not EOF
```

```
{
    c = next input character;
    if (s + c) exists in the dictionary
        s = s + c;
    else
    {
        output the code for s;
        add string s + c to the dictionary with
        new code;
        s = c;
    }
}
output the code for s;
```

Het algoritme leest karakter per karakter. Als de code in het woordenboek voorkomt, zal het algoritme het huidige karakter toevoegen aan de huidige string die in verwerking is en wacht het op een volgend karakter.

Als de huidige string niet vindbaar is in het woordenboek, voegt het algoritme de huidige string toe aan het woordenboek en geeft de code weer van de huidige string (zonder laatste karakter). Het algoritme begint hierna opnieuw met een nieuwe string die gelijk is aan het nieuwe karakter.

Echter is er een nadeel dat er niet gezocht wordt naar de optimale string waardoor de tabel snel groeit.

Het algoritme voor het decoderen van de codewoorden in pseudo-code is:

```
s = NULL;
while not EOF
{
    k = next input code;
    entry = dictionary entry for k;
    output entry;
    if (s != NULL)
        add string s+entry[0] to dictionary with
        new code;
    s = entry;
}
```

Een pluspunt is dat dit algoritme zijn eigen woordenboek bouwt dat precies gelijk is aan die van de compressor, zodat alleen de codes doorgestuurd moeten worden. De decoder ondervindt echter wel een probleem als hij een nieuwe code niet kent. Dit wordt opgelost door een nieuwe code aan te maken en toe te voegen aan het woordenboek.

1.4.2 Voorbeeld

Stel dat we de volgende string hebben:

yabbadabbadabbadoo

Tabel 1.4 toont de compressie van deze string.

Huidige string	**Input**	**Toevoegen**	**Nieuw woordenboek**
			alle enkele karakters
yabbadabbadabbadoo	*y*	*ya*	*ya*
abbadabbadabbadoo	*a*	*ab*	*ya ab*
bbadabbadabbadoo	*b*	*bb*	*ya ab bb*
badabbadabbadoo	*b*	*ba*	*ya ab bb ba*
adabbadabbadoo	*a*	*ad*	*ya ab bb ba ad*
dabbadabbadoo	*d*	*da*	*ya ab bb ba ad da*
abbadabbadoo	*ab*	*abb*	*ya ab bb ba ad da abb*
badabbadoo	*ba*	*bad*	*ya ab bb ba ad da abb bad*
dabbadoo	*da*	*dab*	*ya ab bb ba ad da abb bad dab*
bbadoo	*bb*	*bba*	*ya ab bb ba ad da abb bad dab bba*
adoo	*ad*	*ado*	*ya ab bb ba ad da abb bad dab bba ado*
oo	*o*	*oo*	*ya ab bb ba ad da abb bad dab bba ado oo*
o	*o*		

Tabel 1.4: Tabel voor het LZW-algoritme bij de voorbeeldstring

1.4.3 LZ77

Lempel en Ziv hadden in 1977 een eerdere variant, namelijk de 'adaptieve Lempel-Ziv' of kortweg 'LZ77', ontwikkeld. Hierbij wordt de invoerdata opeenvolgend gescand en de langste herkenbare invoerstring, die reeds in de woordenlijst bestaat, geparsed. De gekende string wordt vervolgens vervangen door zijn bijhorende code.

Elke geparste invoerstring, die uitgebreid wordt met een extra invoerkarakter, geeft een string weer die niet in de woordenlijst bestaat. Deze nieuwe

string wordt dan aan de woordenlijst toegevoegd en krijgt een unieke code toegewezen.

Op deze manier wordt de woordenlijst incrementeel opgebouwd gedurende het compressie-proces. Voor decompressie gebruikt de decoder dezelfde woordenlijst als de codeur en stelt deze op dezelfde manier samen als de codeur.

1.5 Run-length encoding (RLE)

De run-length encoding, afgekort tot RLE, is waarschijnlijk de eenvoudigste codering die zijn voordeel haalt uit de context. Het principe is dat elke stroom van gelijke bronboodschappen (dit noemt men een run) in een set van bronboodschappen wordt vervangen door een teller en één enkele bronboodschap.

Dit intuïtief principe werkt het best op data waarin runs van bronboodschappen worden opgemerkt. RLE wordt dan ook meestal gebruikt voor bestanden die een groot hoeveelheid van opeenvolgende gelijke bitpatronen bevatten.

Er zijn vele varianten van het RLE algoritme. Sommige kunnen bijvoorbeeld alleen tekens of volledige blokken met tekens herhalen, afhankelijk van de data die er gecompresseerd moet worden.

1.5.1 Het algoritme van RLE

Het basisidee is als volgt:

1. Zoek naar meer dan drie opeenvolgende voorkomens van een symbool.

2. Vervang deze reeks van dezelfde karakters door een scheidingskarakter dat weinig of niet voorkomt (bijvoorbeeld '*') in de te comprimeren string. Hierachter wordt het aantal voorkomens van dit symbool toegevoegd, gevolgd met het symbool zelf.

Op het eerste zicht mag het scheidingskarakter overbodig lijken, maar dit speciale karakter duidt aan dat het eerst volgende karakter het aantal herhalingen aanduidt van het hierop volgende karakter.

Het is belangrijk om op te merken dat deze methode slechts effectief is als er runs van vier of meer worden vervangen. Een run van twee zou leiden tot expansie van de boodschap en een run van drie leidt noch tot compressie noch tot expansie.

Er zijn speciale gevallen mogelijk. Als het scheidingskarakter al voorkomt, wordt dit vervangen door twee opeenvolgende karakters. Verder wordt het aantal voorkomens binair voorgesteld met 1 byte (8 bits). Het aantal voorkomens kan dan variëren tussen 0 en 255 ($= 2^8 - 1$). Als het symbool meer dan 255 keer repeteert wordt de reeks opgesplitst in verschillende reeksen.

1.5.2 Voorbeeld

Volgens het basisidee van RLE wordt de reeks

$$RTAAAASDEEEEE$$

vervangen door

$$RT*4ASD*5E$$

Stel dat we de volgende tekenreeks in het alfabet [a-z]* willen comprimeren:

$$dghakaaaaaaaaaaaaabbbbaaakhffff$$

Stel eveneens dat we ons alfabet uitbreiden met de tekens [0-9] om herhalingen aan te geven, dan zouden we de string als volgt kunnen comprimeren:

$$dghak14a4b3akh4f$$

In het bovenstaande voorbeeld wordt elke stroom van gelijke bronboodschappen vervangen door het aantal van de bronboodschap en de bronboodschap zelf. Merk op dat er geen vlagkarakter gebruikt wordt aangezien het niet tot het alfabet behoort.

1.6 Rekenkundige codering

Rekenkundige, of in het engels 'Arithmetic', codering lijkt zeer veel op de Huffman codering, maar werkt meestal iets efficiënter. De basis voor rekenkundige codering is gebaseerd op het maken van schattingen. Wanneer deze schatting juist is, is dit de meest effectieve manier van coderen. Wanneer de schatting echter verkeerd is, is er een mogelijkheid dat de gecomprimeerde data groter is dan het origineel. Dit is uiteraard niet de bedoeling.

Rekenkundige codering is een techniek dat de informatie van berichten in een reeks van berichten toelaat gecombineerd te worden om dezelfde bits te kunnen delen. Deze techniek laat toe om het totaal aantal bits die worden verzonden de som van de oorspronkelijke informatie van de individuele berichten asymptotisch te benaderen.

Laten we een voorbeeld nemen om deze techniek toe te lichten. Veronderstel dat er duizend berichten verzonden worden, elk met een waarschijnlijkheid van 0.999. Gebruik makend van de Huffman codering heeft elk bericht minstens één bit nodig waardoor 1000 bits verzonden worden. Anderzijds is de hoeveelheid informatie van elk oorspronkelijk bericht $\log_2 \frac{1}{p_i} = .00144$ bits. Dat maakt dat de som van de 1000 berichten slechts 1.4 bits bedraagt. Het blijkt dus dat de rekenkundige codering voor alle berichten slechts drie bits gebruikt. Dat is een factor van honderd minder dan die bij de Huffman codering. Natuurlijk is dit een extreem geval. Wanneer alle waarschijnlijkheden klein zijn, zal de aanwinst minder significant zijn. De rekenkundige codering is daarom het nuttigst wanneer er grote waarschijnlijkheden zijn in de kansverdeling.

Het globaal idee van de rekenkundige techniek is om elke mogelijke opeenvolging van n berichten voor te stellen door een afzonderlijke interval op een reële rechte tussen '0' en '1'. Voor een serie van berichten met waarschijnlijkheden $p_1,\ldots,\ p_n$, zal het algoritme de serie toewijzen aan een interval van grootte $\prod_{i=1}^{n} p_i$, beginnend met een interval van grootte '1' met telkens het interval verkleinend met een factor van p_i bij elk bericht i. Een bronboodschap met een hoge probabiliteit versmalt het interval minder dan een bronboodschap met een lagere probabiliteit, zodat bronboodschappen met hogere probabiliteit minder bits bijdragen tot het gecodeerd geheel.

1.6.1 Algoritme

Het algoritme voor het leveren van de codewoorden, in pseudo-code, is:

```
laag = 0;
```

```
hoog = 1;
zolang er bronboodschappen zijn
    lees een bronboodschap in;
    codeBereik = hoog - laag;
    hoog = laag + codeBereik * hoogBereik(bronboodschap);
    laag = laag + codeBereik * laagBereik(bronboodschap);

geef laag weer;
```

Merk ook op dat de grootte van het laatste interval gelijk is aan het product van de probabiliteiten van alle bronboodschappen die voorkomen in de te coderen tekst.

Het algoritme voor het decoderen van de codewoorden in pseudo-code is:

```
neem laag;
doe
    zoek de bronboodschap welke zijn interval laag
    omsluiten;
    geef die bronboodschap weer;
    bereik = bovengrens - ondergrens van het interval
    van bronboodschap;
    laag = laag - ondergrens van het interval van bron-
    boodschap;
    laag = laag / bereik;
totdat er geen bronboodschappen meer zijn;
```

Merk op dat hier expliciet een probleem ontstaat omdat we niet weten dat er nog bronboodschappen zijn om te decoderen. We kunnen op twee manieren beslissen hoe het decodeer-algoritme stopt:

- We kunnen een speciaal symbool EOF (End OF File) coderen dat slechts éénmaal in een reeks van bronboodschappen voorkomt.
- We kunnen ook de lengte van de set van bronboodschappen meegegeven met de gecodeerde tekst.

Elk van deze twee manieren zorgt ervoor dat er extra bits worden toegevoegd. Er zijn drie soorten rekenkundige compressies:

- *fixed model*: hierbij worden vaste probabiliteiten gebruikt voor alle sets van bronboodschappen, waardoor men over het algemeen een zeer lage compressiewaarde krijgt.

- *semi-adaptieve model*: hierbij wordt er een eerste maal door de set van bronboodschappen gelopen om de probabiliteiten te verzamelen. Een tweede keer voor het coderen van het geheel waarbij de tabel met de probabiliteiten meegegeven moet worden met het gecodeerde geheel.

- *adaptieve model*: In het adaptieve model wordt de probabiliteit van elke bronboodschap dynamisch geschat telkens er een nieuwe bronboodschap wordt ingelezen. Deze methode vereist dat er slechts één maal door de boodschappen gelopen wordt.

1.6.2 Voorbeeld

Stel dat we het volgend bericht willen coderen:

$$ONGEORDEND$$

We berekenen eerst de probabiliteit van voorkomens van de bronboodschappen. Hier kennen we aan elke bronboodschap een deel van het interval [0,1] toe. Tabel 1.5 geeft een overzicht van deze waarden.

Bronboodschap	**Probabiliteit**	**Subinterval**
O	2/10	0 - 0.2
N	2/10	0.2 - 0.4
G	1/10	0.4 - 0.5
E	2/10	0.5 - 0.7
R	1/10	0.7 - 0.8
D	2/10	0.8 - 1

Tabel 1.5: Tabel voor de Rekenkundige Codering van de voorbeeldstring

Vervolgens berekenen we met het gegeven algoritme de opeenvolgende waarden van 'laagBereik' en 'hoogBereik' (zie Tabel 1.6).
De uiteindelijke waarde voor 'laag' is 0.0581266304.

Merk op dat de grootte van het laatste interval, nl. $0.058126656 - 0.0581266304 = 2.56^{-8}$. Dit is precies gelijk aan het product van de probabiliteiten van de bronboodschappen in de tekst, nl. $(\frac{2}{10})^8 * (\frac{1}{10})^2 = 2.56^{-8}$.

Stel nu dat we het getal 0.0581266304 moeten decoderen naar een woord, wetende dat we enkel gebruik maken van de eerste tabel. We vinden dat het getal 0.0581266304 in het interval [0.0; 0.2] ligt. We vinden eveneens dat de eerste bronboodschap van het woord gelijk is aan de bronboodschap uit het interval [0.0; 0.2], nl. O. Nu berekenen we bereik: 0.2 - 0.0 = 0.2. We trekken de ondergrens van het interval [0.0; 0.2] af van het getal 0.0581266304 en delen

Bronboodschap	**laagBereik**	**hoogBereik**
	0.0	1.0
O	0.0	0.2
N	0.4	0.8
G	0.056	0.06
E	0.058	0.0588
O	0.058	0.05816
R	0.058112	0.058128
D	0.0581248	0.058128
E	0.0581264	0.05812704
N	0.058126528	0.058126656
D	0.0581266304	0.058126656

Tabel 1.6: 2de tabel voor de Rekenkundige Codering van de voorbeeldstring

dit getal door 0.2. Het getal dat we bekomen is 0.290633152. Hier herbegint het algoritme. Zo vinden we na een aantal stappen de oorspronkelijke tekst 'ONGEORDEND' weer.

1.7 Burrows-Wheeler Transformatie

De Burrows-Wheeler Transformatie wordt afgekort tot BWT en in het engels ook wel 'block sorting' genoemd. BWT werd in 1994 gepubliceerd door Michael Burrows en David Wheeler. BWT is een transformatie van een tekenreeks naar een vorm waarin deze gemakkelijker te comprimeren is.

Wanneer een string van karakters door BWT wordt omgezet, verandert er niks aan deze karakters. Het herschikt enkel de orde van de karakters. Als de originele string verscheidene substrings heeft die vaak voorkomen, zal de getransformeerde string verscheidene plaatsen hebben waar één enkel karakter herhaaldelijk opeenvolgend in een rij voorkomt. Deze transformatie is nuttig voor compressie, aangezien hier andere compressietechnieken beter op toegepast kunnen worden.

Om een goede compressie te bereiken, moet de tekenreeks minimaal enkele kilobytes omvatten, daar BWT volledige blokken tekst behandeld als één string.

1.7.1 Voorbeeld

De Burrows-Wheeler transformatie wordt uitgevoerd door alle rotaties van de tekst te sorteren waarbij de laatste kolom als output dient.

Bijvoorbeeld, de string '.BANANA.' wordt omgezet in 'BNN.AA.A'. De gevolgde stappen staan beschreven in Tabel 1.7.

Input string	**Alle rotaties**	**Gesorteerde rotaties**	**Output**
.BANANA.	.BANANA.	ANANA..B	BNN.AA.A
	..BANANA	ANA..BAN	
	A..BANAN	A..BANAN	
	NA..BANA	BANANA..	
	ANA..BAN	NANA..BA	
	NANA..BA	NA..BANA	
	ANANA..B	.BANANA.	
	BANANA..	..BANANA	

Tabel 1.7: Tabel voor het voorbeeld bij BWT

1.8 Tools

De bovenstaande compressie-algoritmes kunnen als een alleenstaande applicatie gebruikt worden om een tekst te coderen. Er kunnen meerdere technieken samengevoegd worden om een nog betere compressie te bekomen. We zullen in deze sectie enkele van deze programma's bekijken die het meest bekend zijn.

1.8.1 Zip

In de volksmond komt de term 'zippen' nog al eens voor. Hiermee wordt niet alleen het comprimeren van bestanden bedoeld, maar ook het samenvoegen van meerdere gecomprimeerde bestanden tot één archief. Zippen is ontstaan doordat veel mensen hetzelfde programma gebruikten om bestanden te verkleinen. Eén van de eerste veel gebruikte toepassingen was immers 'Pkzip', die door Phil Katz ontwikkeld werd in 1989. Deze tool produceert bestanden met de extensie .zip. ZIP-bestanden worden vooral gebruikt op DOS- en Windows-platformen.

De tool Pkzip maakt gebruik van het zogenaamde 'deflate' algoritme. Deze techniek definiëert een verliesloos gecomprimeerde gegevensformaat dat gebruik maakt van een combinatie van het LZW algoritme en de Huffman codering. De efficiëntie is vergelijkbaar met de betere compressiemethodes voor algemeen gebruik. De data kan, zelfs voor een willekeurig lange opeenvolgende stroom van inputgegevens, worden geproduceerd of worden verbruikt, gebruik makend van slechts een vooraf bepaalde begrensde hoeveelheid tussenopslag. De deflate techniek werd ontwikkeld als reactie op het bezit van patenten die LZW en andere compressie-algoritmen hadden.

Er bestaan tegenwoordig meerdere programma's om het ZIP-bestandsformaat te creëren, te wijzigen en te openen, zoals onder andere WinZip en WinRar.

WinZip is een commerciële zip-tool vooral bestemd voor Microsoft Windows gebruikers en is ontstaan in de vroege jaren '90 als shareware GUI voor Pkzip.

WinRar is een shareware programma om bestanden te comprimeren. Het maakt bestanden aan met de extensie .rar. De compressieverrichtingen van RAR zijn typisch langzamer dan dezelfde gecomprimeerde gegevens met ZIP en andere compressie-algoritmen, maar een betere compressie wordt in de meeste gevallen wel bereikt.

1.8.2 Gzip

In Unix en Unix-achtige besturingssystemen gebruiken we vooral het bestandsformaat *gzip* voor de compressie in combinatie met de meest bekende tool *tar* voor de archivering. De term gzip staat voor 'GNU zip' en maakt gebruik van de extensie .gz. Ook onder Microsoft Windows kan gzip uitgevoerd worden.

Gzip is, zoals het formaat ZIP, gebaseerd op het deflate algoritme. Echter mogen deze twee niet met elkaar verward worden aangezien deze niet compatibel met elkaar zijn. Gzip archiveert geen bestanden, maar comprimeert ze enkel. Dit is ook de reden waarom gzip vaak samen met een afzonderlijk hulpmiddel voor de archivering wordt gezien.

Om het gemakkelijker te maken om software te ontwikkelen die compressie gebruikt, werd de *zlib*-bibliotheek gecreëerd. Het ondersteunt het gzip-gegevensformaat en de deflate-techniek. De bibliotheek wordt wijd gebruikt wegens zijn kleine omvang, efficiëntie en veelzijdigheid. Zowel Gzip als Zlib werden geschreven door Jean-Loup Gailly en Teken Adler.

Een nog betere compressie bereikt de opvolger van Gzip, namelijk de tool Bzip2, maar deze is ook heel wat trager. Het programma Bzip2 comprimeert bestanden door gebruik te maken van het Burrows-Wheeler transformatie tekstcompressie algoritme en Huffman codering. De compressie is over het algemeen aanzienlijk beter dan dat wordt bereikt door de meer conventionele op LZW gebaseerde, compressors.

Hoofdstuk 2

Compressie van XML

XML wordt meer en meer een populaire manier voor het voorstellen, opslaan en uitwisselen van gegevens over het Internet. De hoeveelheid gegevens beschikbaar in XML groeit snel waardoor efficiënte transport en opslagtechnieken dan ook noodzakelijk geworden zijn. Een dergelijke techniek hiervoor is compressie. De conventionele compressoren, zoals de codering van Lempel-Ziv-Welch of Huffman, bereiken redelijke compressie. Nochtans hanteren deze algoritmes niet de specifieke syntaxis en de semantiek van XML en missen zo verscheidene kansen voor een betere compressie.

We gebruiken de termen 'verhouding' en 'factor' om de compressie aan te duiden:

- $compressie - verhouding = \frac{omvang(gecomprimeerd\ bestand)}{grootte(origineel\ bestand)}$
- $compressie - factor = (1 - \frac{omvang(gecomprimeerd\ bestand)}{grootte(origineel\ bestand)}) \times 100$

In dit hoofdstuk geven we een overzicht van enkele tools (zoals XMill in Sectie 2.3 en XPRESS in Sectie 2.5) en technieken (zoals de DAGs in Sectie 2.8).

2.1 Gebruikte databestanden

Voor de experimenten van de verschillende programma's in dit hoofdstuk, gaan we enkele XML-bestanden gebruiken.

We gaan zo een tiental testbestanden gebruiken die over een grote variëteit aan XML-gegevens beschikken. In Tabel 2.1 is een overzicht van de bestanden met enkele karakteristieken.
Het XML-bestand van Baseball [31] bevat de statistieken van alle spelers en teams uit de Major League 1998. DBLP [27] is een populaire bibiliografische

XML-Bestand	Grootte	Max Diepte	# Elem	# Attr
Baseball	639,8 KB	6	27 080	0
DBLP	127,7 MB	6	3 332 130	404 276
TCP-H	30,8 MB	3	1 022 976	1
Mondial	1,7 MB	5	22 423	47 423
Nasa	23,9 MB	8	476 646	56 317
Sigmod	467,2 KB	8	11 526	3 737
XMark	111,1 MB	4	1 666 315	621 490
SwissProt	109,5 MB	5	2 977 031	2 189 859
TreeBank	82,1 MB	36	2 437 666	1
UWM	2,2 MB	5	66729	6
WSU	1,6 MB	4	74557	0
PSD	683,6 MB	7	21 305 818	1 290 647
Weblog	30 MB	3	641 037	1
Shakespeare	7,3 MB	6	179 072	0

Tabel 2.1: XML-bestanden met enkele karakteristieken

databank die relatief regulair is. TCP-H [27] is de TCP-H benchmark databank omgezet in een XML-formaat. De geografische databank van de wereld wordt beschreven in Mondial [27]. Nasa [27] bevat allerhande gegevens over de astronomie en is ook redelijk diep genest. De XML-versie Sigmod [27] geeft een index van artikels weer uit de betreffende ACM Sigmod Records. Het bestand XMark is met de data-generatie tool 'xmlgen' van de XML Benchmark project XMark [28] aangemaakt. Dit bestand stelt een databank van veilingen voor. SwissProt [27] is een databank van DNA-strings met een veelzijdige beschrijving. De TreeBank [27] bestaat volledig uit Engelstalige zinnen en heeft een diep geneste structuur. UWM [27] en WSU [27] bevatten de gegevens van lessen aan de betreffende universiteiten. De Proteïne Sequentie Databank wordt in PSD [27] samengevat en is nogal diep genest. De Weblog [29] is een reeks van logbestanden van de Apache HTTP Server. In dit XML-bestand is er een minimale redundantie en een hoog niveau van integratie met andere databanken. Het XML-bestand Shakespeare [30] bevat de volledige verzameling scenes van Shakespeare en bevat grote stukken tekst.

De experimenten werden uitgevoerd op een Intel Pentium 4, 2.66 GHz CPU, 256 MB RAM met een dualboot-systeem Linux SuSE 10 en Windows XP.

2.2 Algemene compressie methoden op XML-data

In het vorige hoofdstuk hebben we een overzicht gezien van de belangrijkste compressietechnieken. Daar deze methoden bij het coderen van de tekstbestanden geen verlies van gegevens veroorzaken, kunnen deze technieken ook op gewone XML-bestanden uitgevoerd worden.

Aan de hand van enkele experimenten hebben we resulaten bekomen voor de compressie met Gzip [33], Bzip2 [34], Winzip [35] en Winrar [36]. Verder zijn de technieken van de rekenkundige en de LZW codering, de RLE encodering en het algoritme van Huffman uitgeprobeerd aan de hand van een algemene codeerprogramma [32].

2.2.1 Zip en Gzip

In Tabel 2.2 zijn de resultaten van de programma's Gzip, Bzip2, Winzip en Winrar opgenomen. Er is telkens geopteerd voor de best mogelijke compressie.

XML-Bestand	**Grootte**	**Gzip**	**Bzip2**	**Winzip**	**Winrar**
Baseball	639,8 KB	53,7 KB	26,6 KB	66,7 KB	31,9 KB
DBLP	127,7 MB	22,9 MB	15,2 MB	24,1 MB	12,5 MB
TCP-H	30,8 MB	2,6 MB	1,3 MB	2,98 MB	1,38 MB
Mondial	1,7 MB	156,1 KB	107,2 KB	176 KB	112 KB
Nasa	23,9 MB	3,6 MB	2,6 MB	3,69 MB	2,25 MB
Sigmod	467,2 KB	78,3 KB	47,5 KB	81,4 KB	43,6 KB
XMark	111,1 MB	36,2 MB	24,4 MB	37 MB	20,3 MB
SwissProt	109,5 MB	13,1 MB	8,3 MB	14 MB	7,48 MB
TreeBank	82,1 MB	29,9 MB	25,7 MB	32,4 MB	26,9 MB
UWM	2,2 MB	150,2 KB	91,7 KB	170 KB	105 KB
WSU	1,6 MB	89,2 KB	57,8 KB	105 KB	64,9 KB
PSD	683,6 MB	101,1 MB	73,2 MB	105 MB	71,3 MB

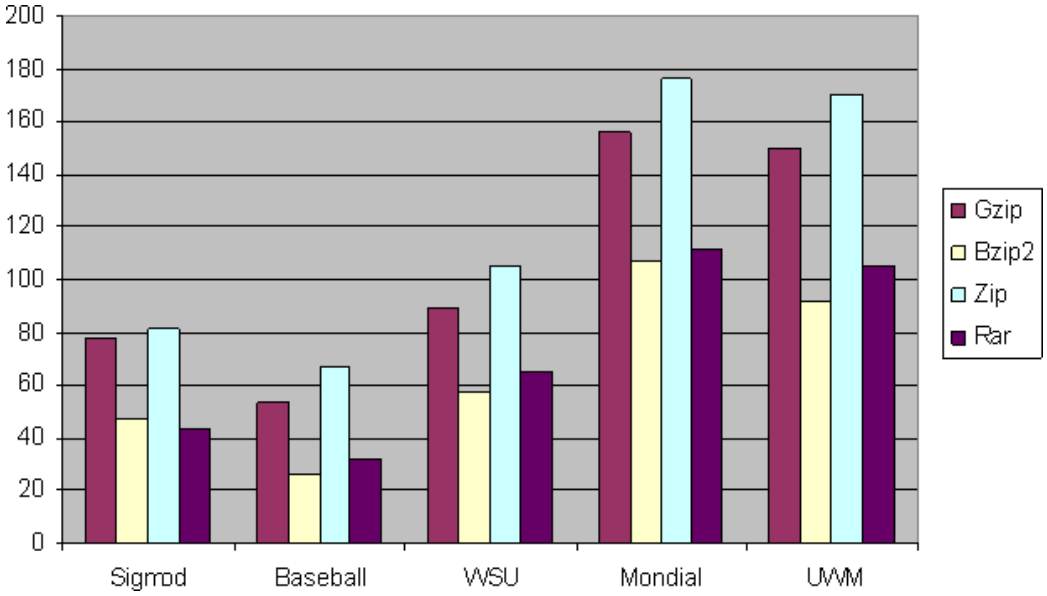
Tabel 2.2: Compressie met Gzip, Bzip2, Winzip en Winrar

In de Tabel 2.2 kunnen we zien dat Gzip een compressie-factor haalt tussen 63,6 % en 93,4 %. Voor Bzip2 is dit een factor tussen 68,7 % en 96,4 %. Bzip2 maakt gebruik van het Burrows-Wheeler Transformatie, wat een betere codering oplevert ten opzichte van zijn voorganger Gzip. De XML-bestanden

die grote stukken tekst bevatten, scoren ook minder goed zoals Treebank. Dit is te wijten aan het gebruik van de bibliotheek Zlib bij de compressie.

Als we kijken naar Winzip, dan zien we dat deze een compressie-factor heeft tussen 60,5 % en 92,2 %. Bij Winrar is dit tussen 67,2 % en 96,5 %. Winrar verricht dus duidelijk een betere compressie dan Winzip en Gzip. Algemeen bekeken zijn Bzip2 en Winrar gemiddeld even sterk en hangt de compressie van het bestand zelf af. Winzip is de grote verliezer en preseert bij elke bestand het minst goed.

In Figuur 2.1 en Figuur 2.2 zijn de resultaten van de compressie duidelijker zichtbaar.



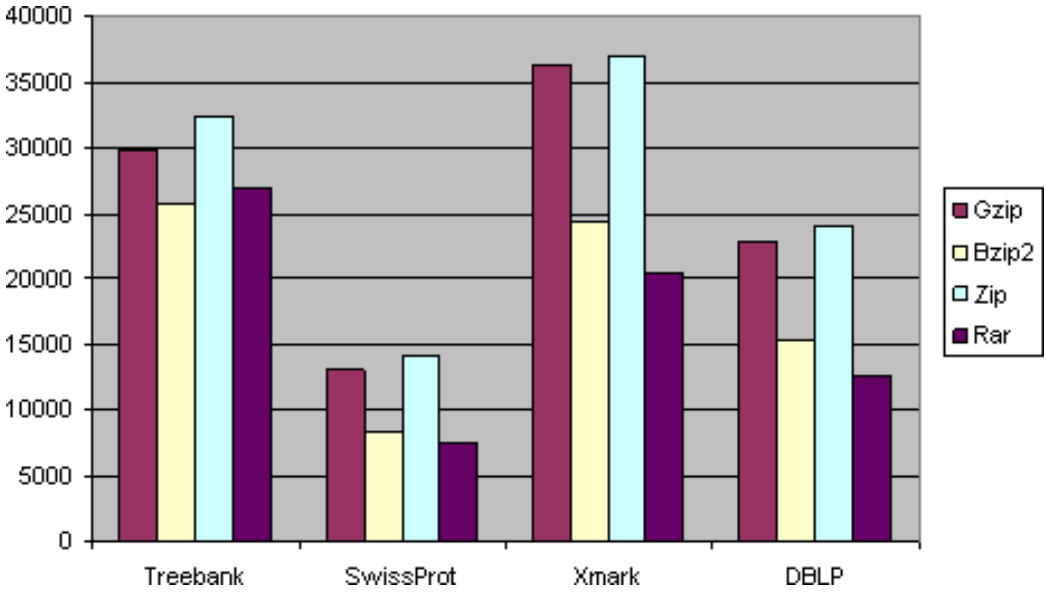
Figuur 2.1: Compressie van kleine bestanden

Als we de snelheden van de compressie gaan beschouwen, merken we op dat de snelheden nauw samenhangen met de grootte van het bestand. Bij Winzip en Gzip is er bijna geen verschil in de snelheid van compressie en zijn de tijden zeer klein. Zo ook zijn de snelheden van Bzip2 en Winrar zo goed als gelijk aan elkaar, maar liggen de tijden een stuk hoger dan bij Gzip en Winzip.

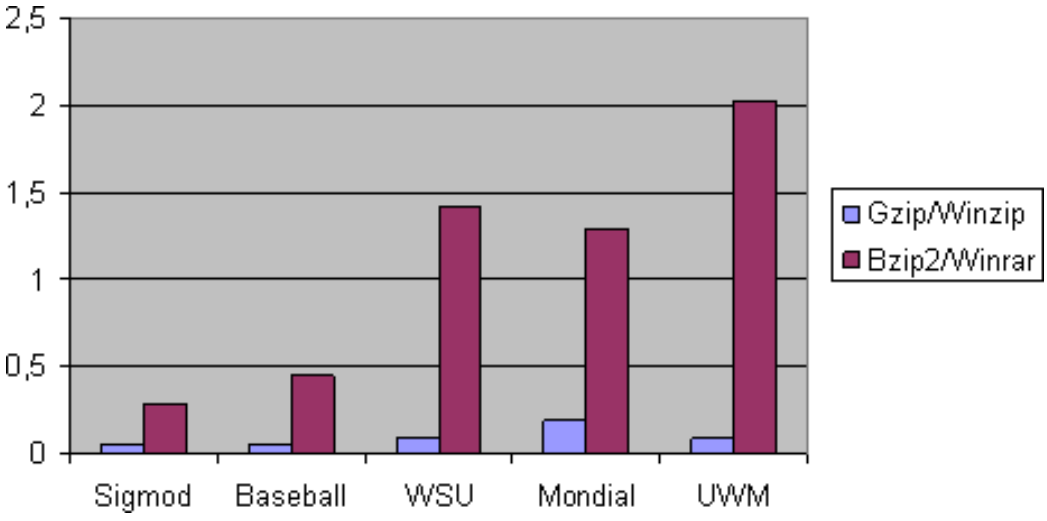
In Figuur 2.3 en Figuur 2.4 zijn de snelheden van de compressie duidelijker zichtbaar.

2.2.2 Entropie compressie

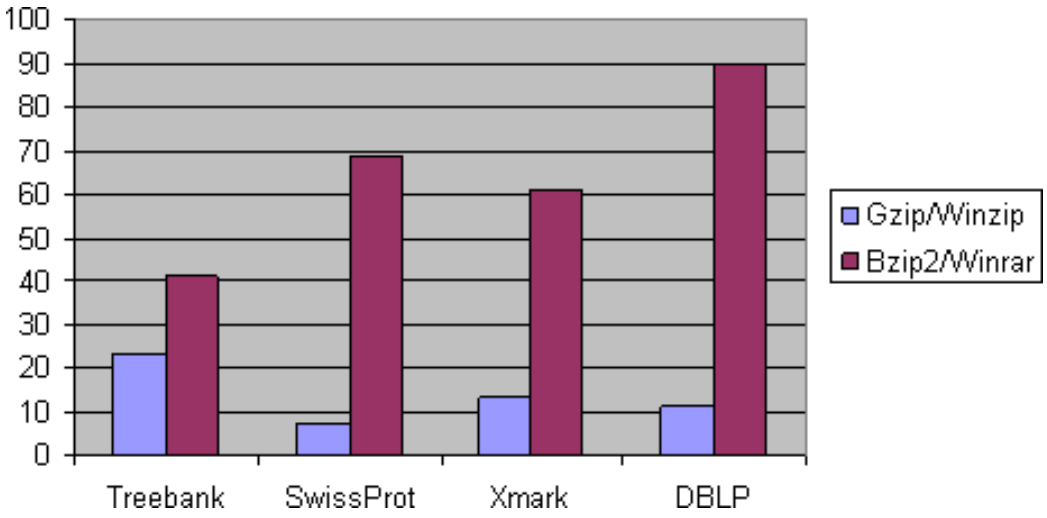
In Tabel 2.3 zijn de resultaten van de rekenkundige en de LZW codering, de RLE encodering en het algoritme van Huffman te vinden.



Figuur 2.2: Compressie van grote bestanden



Figuur 2.3: Compressie-snelheid van kleine bestanden



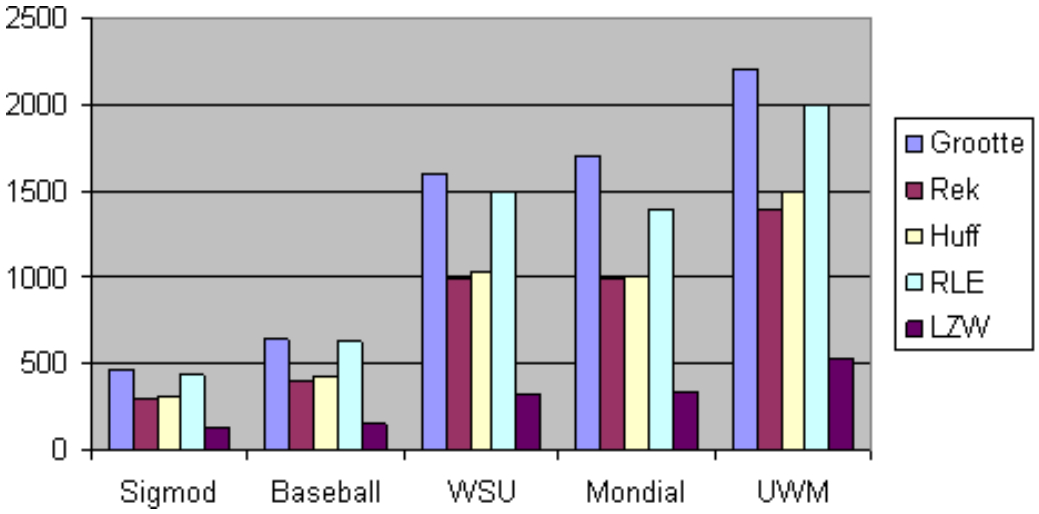
Figuur 2.4: Compressie-snelheid van grote bestanden

XML-Bestand	**Grootte**	**Rek.**	**Huff**	**RLE**	**LZW**
Baseball	639,8 KB	398,2 KB	417,4 KB	629,5 KB	150,1 KB
DBLP	127,7 MB	85,76 MB	93,94 MB	127,7 MB	40,4 MB
TCP-H	30,8 MB	19,8 MB	20,3 MB	30,8 MB	6,5 MB
Mondial	1,7 MB	993,2 KB	998 KB	1,4 MB	330,4 KB
Nasa	23,9 MB	14,9 MB	7,2 MB	22,3 MB	8,1 MB
Sigmod	467,2 KB	228,8 KB	310 KB	440,7 KB	126,3 KB
XMark	111,1 MB	68,89 MB	71,93 MB	111,1 MB	47,6 MB
SwissProt	109,5 MB	79 MB	84,36 MB	109,5 MB	36,4 MB
TreeBank	82,1 MB	54,8 MB	54,84 MB	69,6 MB	39 MB
UWM	2,2 MB	1,4 MB	1,5 MB	2 MB	523,3 KB
WSU	1,6 MB	984,7 KB	1019,4 KB	1,5 MB	319,5 KB
PSD	683,6 MB	479 MB	512,7 MB	683,6 MB	250,7 MB

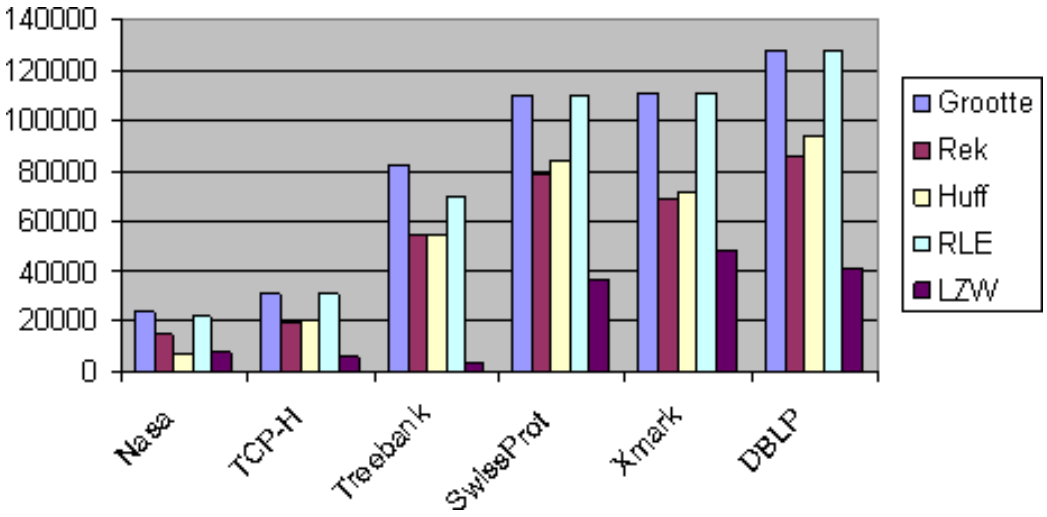
Tabel 2.3: Entropie compressie

Als we kijken naar Tabel 2.3 zien we dat de RLE encoding bijna geen verkleining van het XML-bestand oplevert. De rekenkundige codering presteert een beetje beter dan de Huffman, en ze zijn beide goed voor een gemiddelde compressie-factor van zo een 39 %. De LZW codering heeft de beste resultaten met een factor van gemiddeld 61 %. De snelheid van de compressie hangt van de grootte van het bestand af, en variëert van enkele seconden tot maximaal enkele minuten. In het algemeen gestelt is de snelheid zeer snel. Enkel voor LZW is de compressie snelheid wat trager, tot maximaal 15 minuten.

In Figuur 2.5 en Figuur 2.6 zijn de resultaten van de entropie compressie duidelijker zichtbaar.



Figuur 2.5: Entropie compressie van kleine bestanden



Figuur 2.6: Entropie compressie van grote bestanden

2.3 XMill

Het programma XMill is een compressor speciaal ontworpen voor XML-gegevens. XMill bevat en combineert andere reeds bestaande compressors om de heterogene XML-data te verkleinen. Deze tool gebruikt ook dezelfde bibliotheek als Gzip, namelijk Zlib.

De belangrijkste component van XMill is een 'cluster'-techniek die de gegevens in groepjes bundelt alvorens overeenkomstige compressie op de elementen toe te passen. XMill groepeert de strings met betrekking tot hun betekenis en benut de gelijkenissen ertussen zonder daarbij de informatie van schema's (zoals DTD en XML-schema) nodig te hebben. Deze methode is een veralgemening van de kolom-wijze compressie in relationele databases. Afhankelijk van het te comprimeren geheel kan de gebruiker kiezen tussen standaard cluster-technieken of door zelf de groepering te controleren met gebruikmaking van reguliere expressies. Daarom bereikt XMill ook veel betere compressie-verhoudingen dan conventionele compressoren zoals Gzip, waarbij XMill gemiddeld tweemaal de compressieratio van Gzip bereikt met gemiddeld dezelfde snelheid.

XML-bestanden zijn typisch veel groter dan andere gegevensformaten die dezelfde gegevens kunnen voorstellen. Eén van de meest intrigerende resultaten van XMill is dat de omzetting van de oorspronkelijk gegevensformaten in XML in feite de compressie zal verbeteren. Dit wil zeggen dat het sa-

mengeperste XML-bestand tot tweemaal kleiner is dan het samengeperste oorspronkelijke bestand. Deze compressieverbetering wordt bereikt bij ongeveer dezelfde compressiesnelheid.

Verder kan XMill nog uitgebreid worden met gespecialiseerde compressoren voor complexe gegevensstructuren zoals URL's, datums, beelden, of DNA-sequenties.

2.3.1 Voorbeeld

Weblog-bestanden worden vrijwel door alle webservers gebruikt voor het verkeer te analyseren. Deze bestanden bevatten informatie over de HTTP-aanvragen, zoals onder andere gegevens over het IP van de host, de URL en de grootte van het antwoordpakket.

Beschouw nu de volgende regel uit de Weblog data van een Apache webserver:

```
202.239.238.16|GET /a/b.html HTTP/1.0|text/html|200|
1997/10/01-00:00:02|-|4478|-|-|
http://www.abc-net.or.jp/|Mozilla/3.01 [ja] (Win95; I)
```

Hier zijn er elf waarden gescheiden door een '|'. De ontbrekende elementen zijn weergegeven door een '-'. Deze data is over een lange periode verzameld en kan dus veel ruimte in beslag nemen. Laten we voor dit voorbeeld veronderstellen dat dit een bestand 100.000 regels omvat. Het oorspronkelijke bestand Weblog.dat heeft een grootte van ongeveer 16 MB en gecomprimeerd met Gzip een grootte van 1.6 MB.

Om aan flexibiliteit te winnen, zetten we de gegevens om naar XML. We bekomen het volgend resultaat:

```
<apache:entry>
    <apache:host>202.229.245.18</apache:host>
    <apache:requestLine>GET /a/b.html HTTP/1.0</apache:requestLine>
    <apache:contentType>text/html</apache:contentType>
    <apache:statusCode>200</apache:statusCode>
    <apache:date>1997/10/01-00:00:02</apache:date>
    <apache:byteCount>4478</apache:byteCount>
    <apache:referer>http://www.abc-net.or.jp/</apache:referer>
    <apache:userAgent>Mozilla/3.01 [ja] (Win95; I)</apache:userAgent>
</apache:entry>
```

De grootte van het XML-bestand vergroot aanzienlijk, alsook het door GZip compromimeerde XML-bestand. Met de tool XMill wordt dit XML-bestand gecomprimeerd tot 0.72 MB. Dit is bijna de helft van de grootte van het oorspronkelijke (g)zipte bestandsformaat.

2.3.2 Onderliggende werking

De meest opvallende ontdekking bij XMill is dat bij het omzetten van verschillende data-formaten naar het XML-formaat, de grootte van het gecomprimeerde XML-bestand afneemt. Vele formaten worden tegenwoordig gebruikt voor onder andere biologische data of Weblogs. Deze worden telkens opgeslagen in een ASCII-bestand die ook meestal toepassing-afhankelijk zijn. Vertaald naar XML zal de omvang vergroten aangezien de XML-tags verbose zijn en dus herhaald moeten worden.

De XML data comprimeert behoorlijk met Gzip, maar is nog steeds groter dan de oorspronkelijke ge(g)zipte bestand. Met de tool XMill echter zal het gecomprimeerde bestand kleiner zijn dan het originele ge(g)zipte bestand, tot bijna de helft kleiner. De reden hiervoor is dat XML de aparte gegevens op deze manier bloot geeft zodat deze onafhankelijk zijn van de verschillende applicaties en een betere compressie oplevert. Op deze manier kan het oorspronkelijk bestand gecomprimeerd worden. Hierdoor zou er echter voor alle bestandsformaten een specifieke compressor ontworpen moeten worden. Door het converteren naar XML wordt er zo aan flexibiliteit en efficiëntie gewonnen.

De compressor XMill is gebaseerd op de volgende drie principes:

- *Afzonder de structuur van de gegevens*
  De structuur bestaat uit de XML-tags en de attributen die een boom vormen. De gegevens bestaan uit strings die de waarden in de structuur voorstellen. De structuur wordt afzonderlijk van de gegevens gecomprimeerd.

- *Groepeer de elementen naar verwante betekenis*
  De elementen worden in containers gegroepeerd en elke container wordt apart gecomprimeerd.

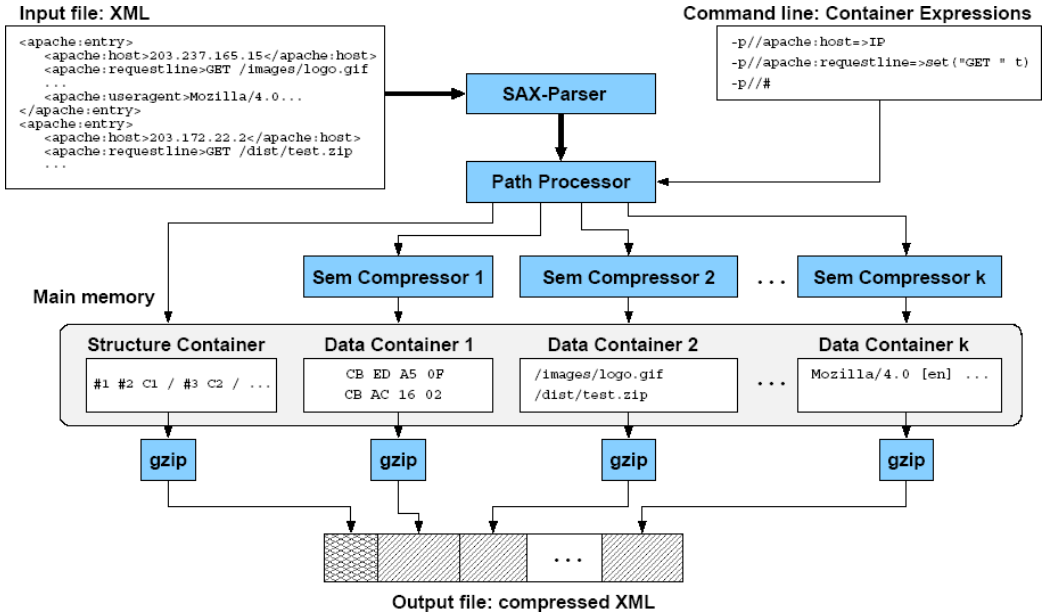
- *Pas semantische compressors toe op de verschillende containers*
  Elke container wordt verschillend gecomprimeerd naargelang zijn semantiek.

In de volgende secties worden deze principes nader toegelicht.

Een origineel component van XMill zijn de 'container-uitdrukkingen'. Dit is een beknopte taal om de elementen in containers in te delen en om de beste combinatie van semantische compressors te kiezen. Elke uitdrukking omschrijft één enkele container ofwel een verzameling van containers. De

juiste hoeveelheid hangt af van de aanwezige tags in de XML data. De keuze van de semantische compressor is omschreven door de samenvoeging van de kleinere compressors in een meer complexe vorm. Dit is zeer handig wanneer de XML data complexe datatypes bevat zoals integers gescheiden door komma's.

In Figuur 2.7 is de architectuur van de compressor XMill te zien, die gebaseerd is op de bovengenoemde principes.



Figuur 2.7: Architectuur van XMill

De SAX-parser [17] ontleedt het XML-bestand die dan de tokens (tags, attributen of waarden) naar de path-processor zendt. Elke token wordt aan een container toegewezen. De tags en attributen, die de structuur van het XML-bestand vormen, worden naar de structuur-container verzonden. De waarden van de elementen worden onderverdeeld in verschillende containers afhankelijk van de uitdrukkingen van de containers. Voordat een waarde in een container wordt geplaatst, is het mogelijk dat deze waarde reeds gecomprimeerd is door een bijkomende semantische compressor. De kern van de compressor XMill is de 'path-processor' die bepaald tot welke container de verschillende waarden horen. De gebruiker kan deze werking controleren door uitdrukkingen aan XMill mee te geven. Alle containers worden apart gecomprimeerd door middel van de tool Gzip en vervolgens in een uitvoerbestand opgeslaan.

Eigenlijk worden de containers in een main memory window gehouden die een vaste grootte, standaard 8 MB, hebben. Als deze window volledig gevuld is, worden alle containers gecomprimeerd en opgeslagen op schijf. Na deze bewerking gaat het compressieproces terug verder. Dit heeft als effect dat de invoergegevens gesplitst worden in vrijstaande gecomprimeerde blokken.

Na het laden en het uitpakken van de containers gaat de decompressor 'Xdemill' de structuur-container parsen. Hierna wordt de overeenkomstige semantische decompressor aangeroepen voor die bepaalde waarden en wordt uiteindelijk het uitvoerbestand aangemaakt.

Een optie in XMill is het behoud van spaties. Hierbij worden de lege ruimtes in container '1' geplaatst, aangezien container '0' de structuur bevat. Hierdoor wordt het gecomprimeerde document met zo een 4% vergroot.

Afzonder de structuur van de gegevens

De structuur van een XML-bestand bestaat uit tags en attributen die in XMill als volgt wordt beschouwd: de start-tags zijn gecodeerd aan de hand van een woordenboek. De eind-tags zijn vervangen door het token '/'. De waarden zijn vervangen door het nummer van hun container.

Als illustratie beschouwen we het volgende XML-bestand:

```
<Book>
    <Title lang="English">Transaction Processing</Title>
    <Author>Gray</Author>
    <Author>Reiter</Author>
</Book>
```

We beschouwen dat de waarde van het attribuut *@lang* in container 3 opgeslagen wordt, de *titles* in container 4 en de *authors* in container 5. Het woordenboek en de bekomen structuur zijn de volgende:

```
Book = #1, Title = #2, @lang = #3, Author = #4
Structure = #1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
```

In praktijk echter worden alle tokens als integers geëncodeerd, met tags en attributen als positieve waarden, dat de token '/' als '0' en de nummers van de containers als negatieve waarden. De bovenstaande structuur neemt 14 bytes in beslag.

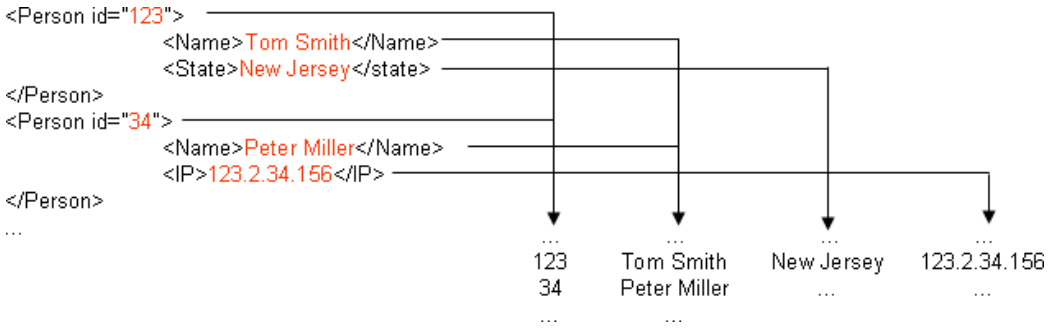
De encoding wordt vervolgens met de tool Gzip zeer goed gecomprimeerd. Hoe meer herhaling er in de structuur voorkomt, hoe efficiënter de compressie.

Veronderstel een lijst van boeken met volgende structuur:

```
#1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
#1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
...
```

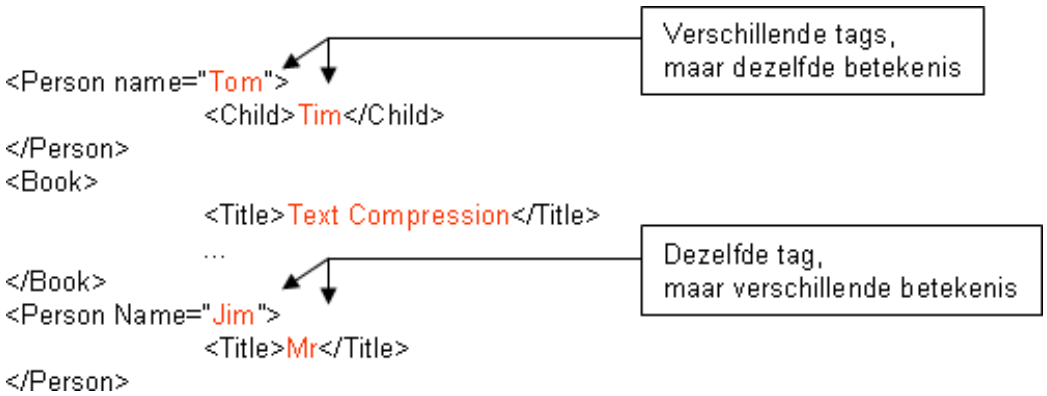
Groepeer de elementen naar verwante betekenis

Elke waarde wordt precies aan één container toegewezen. Deze onderverdeling kan gebaseerd zijn op de tag van de ouder, wat een redelijk goede benadering kan zijn. In Figuur 2.8 wordt deze werkwijze geïllustreerd.



Figuur 2.8: Groepering via ouder

Als de betekenis van de elementen van dezelfde tags nu echter verschillend zijn of als de waarden in verschillende tags dezelfde betekenis hebben, zijn er betere technieken voor groepering nodig. In Figuur 2.9 is hiervan een voorbeeld.



Figuur 2.9: Betere groepering vereist

De verfijnde methode om te groeperen maakt gebruik van het pad van een element en de uitdrukkingen ingevoerd door de gebruiker.

Een XML-document kan voorgesteld worden als een boom en zo is elk element bereikbaar via een pad vertrekkende vanuit de wortel van de boom. Beschouw nu het volgende voorbeeld:

```
<Doc>
    <Book>
        <Title language="English">Compression</Title>
        <Year>1995</Year>
    </Book>
    <Person>
        <Name>Tom</Name>
        <Title>Mr.</Title>
        <Child>Tim</Child>
        <Child>Karen</Child>
        <Child>Karen</Child>
    </Person>
</Doc>
```

In dit voorbeeld heeft $/Doc/Book/Title$ een verschillende betekenis als $/Doc/Person/Title$ en kunnen dus beter apart gecomprimeerd worden. Anderzijds hebben $Name$ en $Child$ dezelfde betekenis en kunnen zo in dezelfde container gestoken worden.

Door afgeleide expressies van XPath worden alle elementen die eraan voldoen in een bepaalde container gestoken. Beschouw de volgende uitdrukking:

```
e ::= label | * | # | e1/e2 | e1//e2 | (e1|e2) | (e)+
```

Uitgezonderd *(e)+* en # zijn het allemaal constructies in XPath. *label* kan ofwel een tag of een attribuut zijn, * geeft eender welke tag of attribuut weer, *e1/e2* is een concatenatie, *e1//e2* is een concatenatie met eender welke pad ertussen, en *(e1|e2)* is een OR-constructie. Hierbij komt nog de Kleenesluiting *(e)+*. De nieuwe constructie # lijkt op de *-constructie, maar creëert telkens een nieuwe container.

Een 'container-uitdrukking' is nu van de vorm $c ::= /e \mid //e$, waarbij $/e$ start bij de wortel van de boom en $//e$ een variabele diepte kan hebben in de boom. We noteren ook // in de plaats van //*.

Enkele voorbeelden: *//Name* maakt één container aan voor alle waarden waarvan het pad eindigt op *Name*. *//Person/Title* maakt een container voor alle *Person's Titles*. De constructie // plaatst alle waarden in één container.

//# maakt een reeks van containers aan met voor elke eindigende tag of attribuut een container. Een krachtige manier is voor elke tag in het XML-document een uitdrukking te maken zodat men een hele collectie bekomt zoals *//Title, //@language, //Name, ...*

De uitdrukking *//Person/#* maakt telkens een aparte container aan voor elke tag onder *Person*.

Om te weten te komen in welke container een waarde geplaatst moet worden, wordt er beroep gedaan op de functie *Match(c, p)* waar c een container-uitdrukking is en p als het pad horende bij de waarde. In Tabel 2.4 staat de definitie van *Match(c, p)*. Deze functie berekent een verzameling strings die de mogelijke toekenningen tot een container voorstellen. Er is een toekenning mogelijk als $Match(c, p) \neq \emptyset$.

Formeel geldt er als c_i (met i=1..n) de eerste container-uitdrukking is die getest wordt met pad p, dan wordt de waarde behorende bij pad p opgeslagen in de container die gekenmerkt is door $(i, Match(c, p))$.
Beschouw het volgende pad p = */Doc/Conf/Paper/Title*, dan geldt:
$Match(/Doc/Title, p) = \{\}$
$Match(/Doc//Title, p) = \{\epsilon\}$
$Match(//\#, p) = \{Title\}$
$Match(//\#/\#, p) = \{Paper/Title\}$
$Match(/(\#)+, p) = \{Doc/Conf/Paper/Title\}$
$Match(//\#//Title, p) = \{Doc, Conf, Paper\}$

$Match(/e, p)$	$= Match(e, p)$
$Match(//e, p)$	$= \bigcup_{q/p_1=p} Match(e, p_1)$
$Match(l, l)$	$= \{\epsilon\}$
$Match(l_1, l_2)$	$= \emptyset$ if $l_1 \neq l_2$
$Match(*, l)$	$= \{\epsilon\}$
$Match(\#, l)$	$= \{l\}$
$Match(e_1/e_2, p)$	$= \bigcup_{p_1/p_2=p} Match(e_1, p_1)/Match(e_2, p_2)$
$Match(e_1//e_2, p)$	$= \bigcup_{p_1/q/p_2=p} Match(e_1, p_1)/Match(e_2, p_2)$
$Match(e_1\|e_2, p)$	$= Match(e_1, p) \bigcup Match(e_2, p)$
$Match((e)+, p)$	$= \bigcup_{p_1/p_2/.../p_n=p} Match(e, p_1)/.../Match(e, p_n)$

Tabel 2.4: Definitie van de functie *Match(c, p)*

Pas semantische compressors toe op de verschillende containers

XML-documenten bevatten vaak vele verschillende data-formaten, zoals integers, datums en postcodes. De tool Gzip comprimeert deze gegevens echter

niet optimaal. Speciale compressors zijn nodig voor een betere compressie. XMill heeft hiervoor acht speciale compressors gelijst in Tabel 2.5.

Compressor	**Beschrijving**
t	standaard tekst compressor
i	integers compressor
di	delta compressor voor integers
e	woordenboek encoder
u	compressor voor positieve integers
u8	compressor voor positieve integers ≤ 255
rl	run-length encoder
"..."	constante compressor

Tabel 2.5: Semantische compressors van XMill

De standaard tekstcompressor *t* doet geen compressie, maar kopieert gewoon de string naar de container en laat de compressie aan Gzip over.

De compressor voor positieve integers *u* zijn binair gecodeerd, waarbij getallen kleiner dan 128 één byte gebruiken. Getallen kleiner dan 16384 gebruiken twee bytes waar de andere vier bytes gebruiken. De meest beduidende één of twee bits bepalen de lengte van de integer.

De compressor comprimeert alle integers *i* vergelijkbaar met compressor *u*. De compressor *u*8 slaat de getallen tussen 0 en 255 op in één byte. De delta compressor *di* slaat het verschil op tussen opeenvolgende getallen.

De RLE *rl* encodeert een reeks dezelfde waarden als een (waarde, aantal)-koppel. De woordenboek encoder *e* slaat enkel een positieve getal op en houdt de unieke waarden in een woordenboek bij. De constante compressor "..." slaat helemaal niets op, maar controleert enkel of de waarde de gegeven constante is.

De tool XMill onderscheidt drie soorten semantische compressors: atomic, gecombineerd en gedefinieerd door de gebruiker. We hebben zojuist hierboven de *atomic* compressors besproken. Dikwijls hebben waarden echter een structuur zoals een IP-adres, waarbij de atomic compressors niet meer optimaal zijn. We gaan gebruik maken van *gecombineerde* semantische compressors. XMill beschikt over drie soorten:

- *Sequentie compressor seq(s1 s2 ...)*
  Voorbeeld: *seq(u8 "." u8 "." u8 "." u8)* voor IP adressen van vier integers. Na elke compressor is een contante vereist.

- *Alternerende compressor or(s1 s2 ...)*

Voorbeeld: paginanummers zoals 145 - 199 of enkel 145 kan geschreven worden als *or(seq(u "−" u) u)*.

- *Repetitie compressor rep(d s)*
  Hier is d een begrenzer en s een andere semantische compressor. Voorbeeld: een reeks woorden gescheiden door komma's kan geschreven worden als *rep(", " e)*.

Voor meer ingewikkelde structuren kan de gebruiker zelf een compressor schrijven zoals voor DNA reeksen. Deze moeten dan aan XMill gelinkt worden.

De semantische compressors worden aan de opdrachtregel meegegeven in de vorm $C ::= c \mid c => s$, met c een container-uitdrukking en s de semantische compressor.

2.3.3 Gebruik

XMill is ontworpen om alle overeenkomstigheden in de gegevens van de semi-gestructureerde XML data zoveel mogelijk uit te buiten om de compressie te verbeteren. Deze regelmatigheid in de data kan door de gebruiker handmatig doorgegeven worden aan het programma in de vorm van hints. Dit is echter een mogelijke optie. Bij eventueel gebruik zal dit in geen enkel geval de invoerdata beperken. Dit past dan ook keurig in het plaatje van de typische semi-gestructureerde gegevens. Een toepassing kan direct gebruik maken van XMill met de standaard instellingen om de XML data te comprimeren, wat reeds een optimalisatie is op de compressors voor algemeen gebruik. Belangrijk is dat XMill nooit XML gegevens zal verwerpen, hoe onnauwkeurig de hints ook mogen zijn.

XMill is een opdrachtregel-tool dat gebruik maakt van volledige bestanden. Een gegeven bestand met extensie '.xml' wordt gecomprimeerd tot een bestand met extensie '.xmi'. Elk andere bestand zonder de extensie '.xml' wordt gecomprimeerd tot een '.xm'-bestand. Anderzijds wordt het oorspronkelijk bestand terug verkregen door de extensie '.xmi' te vervangen door '.xml' of door de extensie '.xm' te verwijderen.

Echter kan de gebruiker tevens de output schrijven naar de standaard output en optioneel lezen van de standaard input.

Container-uitdrukkingen kunnen meegegeven worden aan de opdrachtregel van XMill door de parameter p te gebruiken:

```
xmill -p c1 -p c2 ...  -p cn file.xml file.xmi
```

Voor elke waarde zal de path-processor testen of zijn pad overeenkomt met de in volgorde opgegeven uitdrukkingen: c1, c2, ... De uitdrukking *-p //#* is de standaard uitdrukking en wordt steeds achteraan de opdrachtregel toegevoegd zodat zeker elke waarde in een container zit.

Beschouw de volgende opdrachtregel:

```
xmill -p //Person/Tilte -p //Person/(Name|child) -p //#
file.xml file.xmi
```

Alle *Person's Titles* worden samengezet in een container. Alle *Person's Names* en *Childeren* komen samen in een container en alle overige waarden worden apart in een container gezet afhankelijk van hun eind-tag.

De container-uitdrukkingen kunnen ook met de semantische compressors (zie Sectie 2.3.2) worden uitgebreid. Het volgende voorbeeld illustreert het gebruik hiervan:

```
xmill -p //price=>i -p //state=>e -p //description// file.xml
file.xmi
```

Hier zijn de *prices* gecomprimeerd als integers en de *states* als waarden in een woordenboek. De waarden onder *discriptions* worden in een enkele container opgeslagen zonder een semantische compressor (standaard tekst compressor *t*). De overige gegevens worden gegroepeerd afhankelijk van hun tag. Standaard wordt automatisch aan het einde van *-p //#* toegevoegd zonder een semantische compressor.

2.3.4 Experimentele resultaten

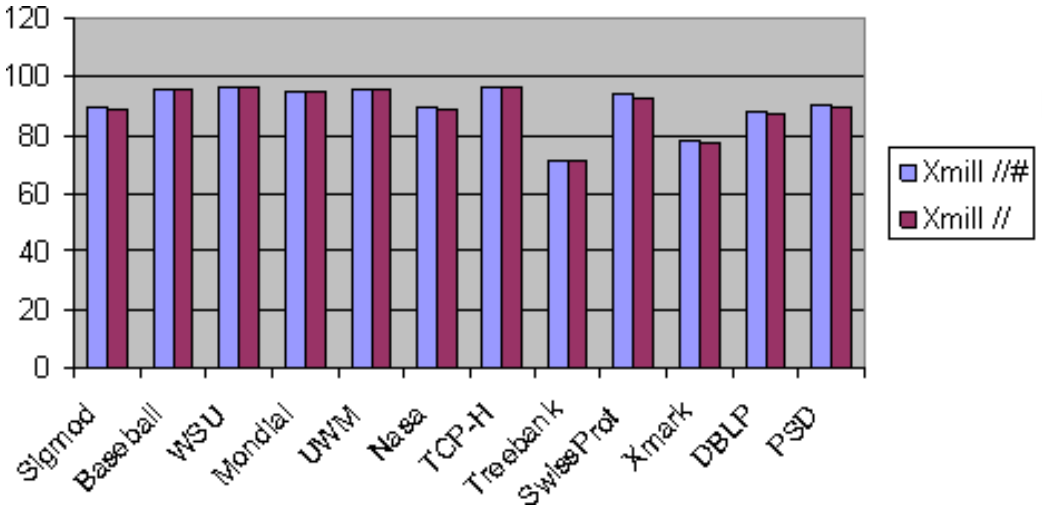
We hebben enkele experimenten uitgevoerd met de bestanden beschreven in Sectie 2.1. De versie XMill 0.9 [37] is gebruikt.
In Tabel 2.6 zijn de resultaten van de compressie te zien. Er zijn twee soorten instellingen gebruikt, namelijk groepering volgens de tag van de ouder (XMill //#), dit is tevens de standaard instelling van XMill, en geen groepering (XMill //).
In Figuur 2.10 zien we dat de compressie-factor van XMill //# een gemiddelde heeft van 90,17 % en bij XMill // is dit 89,58 %. XMill bereikt een betere compressie als er gewerkt wordt met container-uitdrukkingen die door de gebruiker gespecifieerd worden. De grootte is vaak tot 50% à 60% kleiner in tegensteling tot de data gecomprimeerd met Gzip, werkende met gemiddeld dezelfde snelheid. Bij bestanden met meer gegevens dan eigenlijke tekst

XML-Bestand	Grootte	XMill //#	Tijd //#	XMill //	Tijd //
Baseball	639,8 KB	26,3 KB	0m0s170	26,2 KB	0m0s148
DBLP	127,7 MB	15 MB	1m13s093	16,5 MB	1m8s392
TCP-H	30,8 MB	1 MB	0m21s973	1,2 MB	0m23s261
Mondial	1,7 MB	81 KB	0m0s567	84,3 KB	0m0s701
Nasa	23,9 MB	2,5 MB	0m10s453	2,6 MB	0m10s093
Sigmod	467,2 KB	49 KB	0m0s104	51,6 KB	0m0s120
XMark	111,1 MB	24,3 MB	0m49s475	25,4 MB	0m51s387
SwissProt	109,5 MB	6,4 MB	1m8s044	8,2 MB	0m48s331
TreeBank	82,1 MB	23,4 MB	0m19s257	23,5 MB	0m23s637
UWM	2,2 MB	89,3 KB	0m0s464	90,4 KB	0m0s508
WSU	1,6 MB	52,9 KB	0m0s388	58,7 KB	0m0s424
PSD	683,6 MB	65,6 MB	9m0s234	71,4 MB	5m57s522

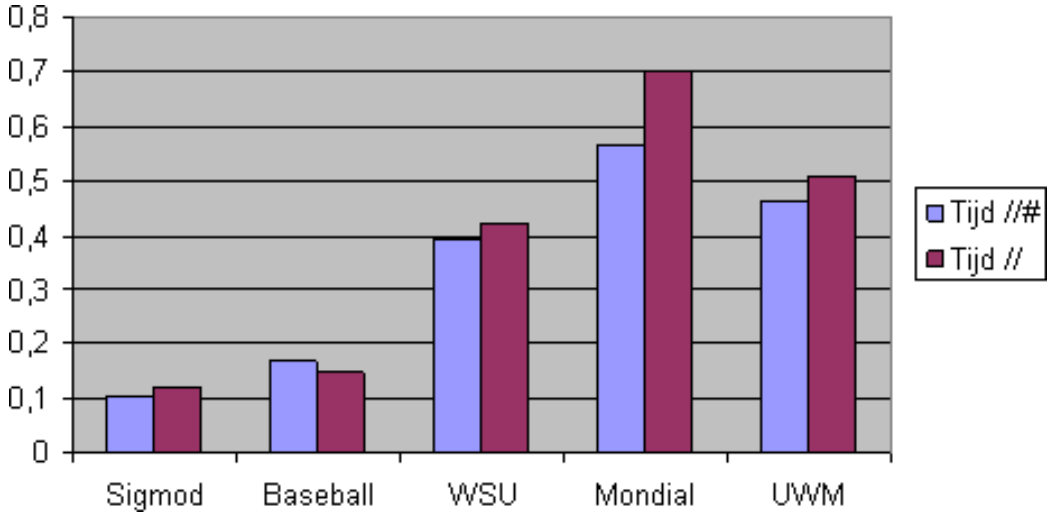
Tabel 2.6: Compressie met XMill

is de compressie ook optimaler. Bij diep geneste structuren is de compressie wat minder, zoals bij Treebank is de compressie-factor slechts 71 %.

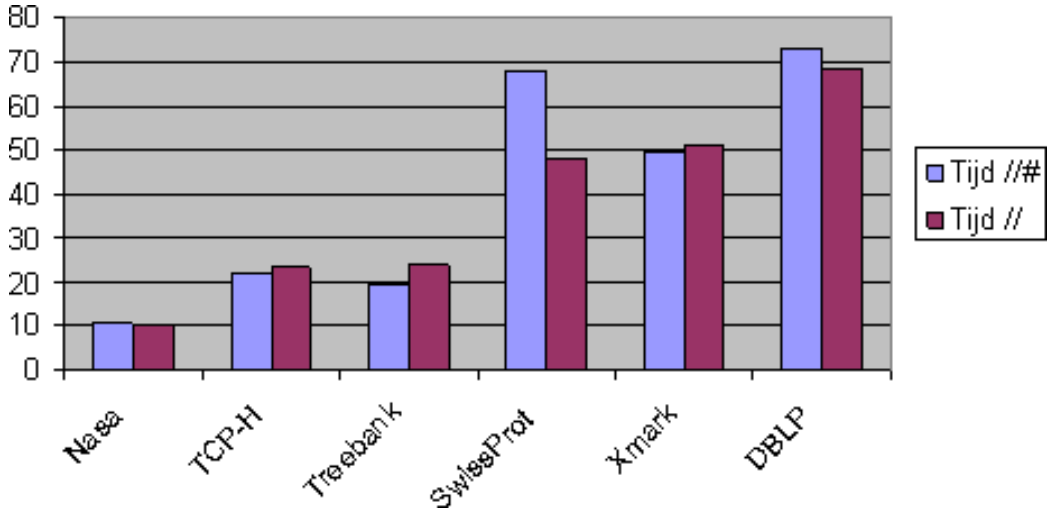
In Figuur 2.11 is de compressie-snelheid van de kleine bestanden weergegeven. Hier duurt de compressie nog geen seconde. Comprimeren met XMill zonder groepering, dus met XMill //, duurt een beetje langer dan groeperen via de ouder. In Figuur 2.12 is de compressie-snelheid van de grote bestanden weergegeven. Afhankelijk van de grootte van het bestand, dus ook het aantal elementen, duurt de compressie langer.



Figuur 2.10: Compressie-factor van XMill



Figuur 2.11: Compressie-snelheid bij kleine bestanden



Figuur 2.12: Compressie-snelheid bij grote bestanden

De compressor XMill heeft ook enkele beperkingen. Deze tool is niet ontworpen om te werken in combinatie met een query-processor. De data zal dus eerst gedecomprimeerd moeten worden vooraleer er queries op gesteld kunnen worden. Dit komt doordat XMill vooral ontworpen is voor data-archivering en data-uitwisseling.

Een andere beperking is dat enkel grote bestanden, meer dan 20 KB, profijt hebben te werken met XMill. Dit komt doordat XMill de gegevens teveel opsplitst in verschillende kleine containers waardoor de compressie niet efficiënt is.

2.4 XGrind

In tegenstelling tot de tool XMill behoudt XGrind de structuur van het oorspronkelijke document na compressie. Dit betekent dat het gecomprimeerd bestand geparsed kan worden, gebruikmakend van dezelfde technieken als voor het parsen van het origineel XML-document. De compressor XGrind bereikt efficiënt en gelijktijdig een redelijk goede compressie-ratio en een goede tijd voor het processen van queries.

2.4.1 Mogelijke bewerkingen

Er zijn drie grote bewerkingen die de compressor XGrind op het gecomprimeerd bestand kan uitvoeren:

- *Queries uitvoeren*
  De *exacte-* en $prefix$-match hebben geen decompressie nodig. De *bereik-* en *gedeeltelijke*-match hebben enkel directe decompressie nodig van elementen of attributen die in de querie voorkomen.
- *Updates*
  De updates kunnen meteen gecomprimeerd doorgegeven worden zonder enige decompressie.
- *Testen van validiteit*
  Een gecomprimeerd XML-bestand kan tegen een gecomprimeerde DTD gevalideerd worden.

2.4.2 Technieken voor compressie

XGrind gebruikt een contextvrije compressie waarbij de code, horende bij een string, niet afhankelijk is van de plaats van de string. Er zijn verschillende technieken voor het comprimeren van meta-data, opsommende attribuutwaarden en algemene waarden van de elementen en attributen. Deze worden verder kort toegelicht.

De gecomprimeerde output wordt, samen met de frequentie- en symbooltabellen, de 'Compressed Internal Representation' (CIR) genoemd en wordt aan de 'XML-Gen' doorgegeven. Tijdens de tweede doorloop van het document wordt de conversie meteen uitgevoerd.

Illustratie

Beschouw het volgende korte XML-bestand:

```
<STUDENT rollno = "604100418">
    <NAME>Pankat Tolani</NAME>
    <YEAR>2000</YEAR>
    <PROG>Master of Engineering</PROG>
    </DEPT name = "Computer_Science">
</STUDENT>
```

De bijhorende DTD is de volgende:

```
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT PROG (#PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer_Science | Physics | Chemistry
| ...)>
```

Gecodeerd met de verschillende compressors en rekening houdend met de DTD bekomen we het volgende:

```
T0 A0 nahuff(604100418)
    T1 nahuff(Pankaj Tolani) /
    T2 nahuff(2000) /
    T3 nahuff(Master of Engineering) /
    T4 A1 enum(Computer_Science) /
/
```

Hier staat *nahuff(s)* als output van de Huffman-compressor voor de string s, en *enum(s)* voor de output van de enumeratie-compressor voor de string s.

Tags

De meta-data of de tags worden gecomprimeerd op dezelfde manier als XMill.

Elke start-tag wordt gecodeerd door een 'T' gevolgd door een uniek element-ID. De eind-tags worden vervangen door een '/'. De namen van de attributen worden vergelijkbaar gecodeerd startend met een 'A'.

Opsommingen

In XML-documenten zijn opsommingen een vaak voorkomende constructie, zoals de steden van een land of de afdelingen in een bedrijf of de verzameling van postcodes. Deze waarden komen frequent voor als een enumeratie. In de DTD wordt duidelijk of de waarde al dan niet een waarde uit een opsomming komt. XGrind analyseert deze waarden samen met de DTD en codeert deze gebruikmakend van een simpele $\log_2 K$-codeerschema om zo de verschillende K waarden in die opsomming voor te stellen.

Elementen en attributen

Door gebruik te maken van een context-vrije compressor is het mogelijk de locatie van een willekeurige string te bepalen in een gecomprimeerd bestand, zonder enige decompressie. Dit is mogelijk doordat de querie, uitgedrukt als een pad-expressie, eerst gecomprimeerd wordt. Vervolgens wordt er gezocht naar de overeenkomende voorkomens in het gecomprimeerd document.

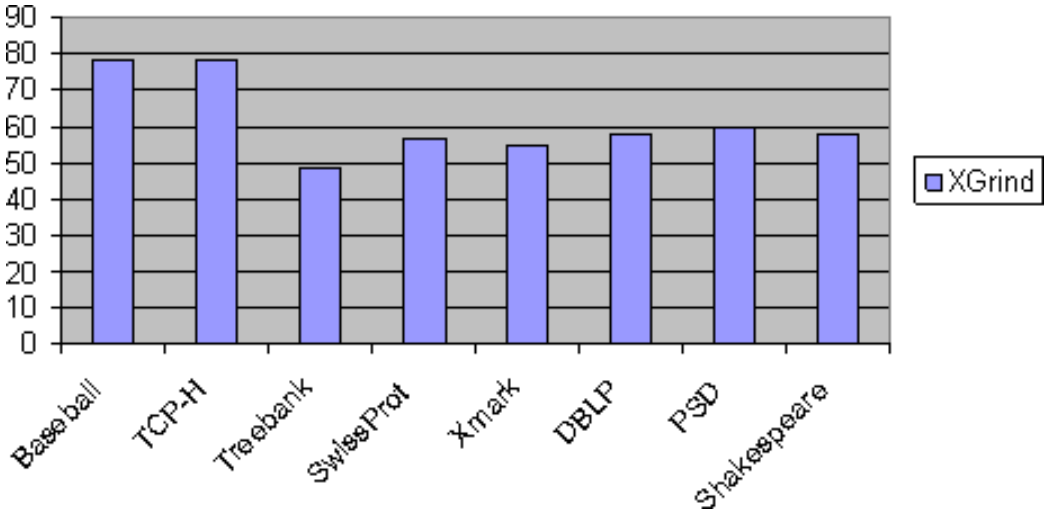
Voor deze compressor gebruikt XGrind de niet-adaptieve Huffman-codering. Voor elke waarde wordt er een aparte Huffmanboom berekend.

2.4.3 Experimentele beoordeling

De compressor XGrind heeft een goede prestatie in het algemeen. Het werkt eveneens goed als het XML-document vooral bestaat uit lange teksten en verschillende enumeraties in attributen.

In Figuur 2.13 is de compressie-factoren van XGrind te zien, met een gemiddelde van 61,5 %. De compressie-snelheid is redelijk lang, met een gemidddelde van 3 á 5 minuten.
Voor het uitvoeren van queries scoort zelfs XGrind beter op snelheid dan XMill, indien deze eerst het gecomprimeerde bestand decomprimeerd.

Een groot voordeel is het toelaten van indices op het gecomprimeerd document, dat XGrind voorziet door het behouden van semi-gesturctureerde XML-documenten in het gecomprimeerd formaat.



Figuur 2.13: Compressie-factor van XGrind

2.5 XPRESS

De tool XPRESS heeft als doel ruimte diskruimte en bandbreedte te besparen. XPRESS is ook een XML-compressor die direct en efficiënt queries kan evalueren op de gecomprimeerde data.

Net als XGrind behoudt het gecomprimeerde document de structuur van het oorspronkelijk bestand.

Een nieuwe codeermethode wordt in XPRESS gebruikt, namelijk de zogenaamde 'omgekeerde rekenkundige codering' (zie Sectie 2.5.1) genoemd of in het engels 'reverse arithmetic encoding'. Deze techniek berust op het coderen van de paden van de labels in de XML-gegevens waarop verschillende codeermethodes op uitgevoerd worden, afhankelijk van de types van die waarden.

2.5.1 Omgekeerde rekenkundige codering

In bestaande XML-compressors worden simpelweg elke tag voorgesteld als een uniek ID. Deze zijn nochtans onefficiënt om pad-uitdrukkingen op gecomprimeerde data uit te voeren.

Hiervoor wordt in XPRESS gebruik gemaakt van een nieuwe techniek, namelijk de 'omgekeerde rekenkundige codering'. Deze is geïnspireerd op de bestaande 'rekenkundige methode' die de paden van de labels codeert als een interval in [0.0, 1.0). Deze techniek zorgt er eveneens voor dat de pad-uitdrukkingen wel efficiënt op de gecomprimeerde gegevens kunnen uitgevoerd worden.

Als eerst stap wordt het hele interval [0.0, 1.0) opgedeeld in distincte subintervallen. Dit wil zeggen één interval voor elk aparte element. De grootte van het interval is afhankelijk van de frequentie van het element. In Tabel 2.7 wordt dit geïllustreerd.
Vervolgens gaat de methode het eenvoudige pad $P = p_1.....p_n$ van een element e coderen in een interval $[min_e, max_e)$ met behulp van volgende functie:

```
Function reverseArithmeticEncodeing(P = p_1.....p_n)
begin
    [min_e, max_e) := Interval_{p_n}
    if (n = 1) return [min_e, max_e)
    length := max_e − min_e
    [q_m in, q_m ax) := reverseArithmeticEncoding(p_1.....p_{n−1})
    min_e := min_e + length ∗ q_m in
    max_e := min_e + length ∗ q_m ax
    return [min_e, max_e)
end
```

Element	**Frequentie**	**Interval**$_T$
book	0.1	[0.0, 0.1)
author	0.1	[0.1, 0.2)
title	0.1	[0.2, 0.3)
section	0.3	[0.3, 0.6)
subsection	0.3	[0.6, 0.9)
subtitle	0.1	[0.9, 1.0)

Tabel 2.7: Tabel met de verschillende intervallen

Intuïtief herleidt deze functie het Interval$_{p_n}$, in verhouding tot het interval voor een eenvoudige pad $P' = p_1.....p_{n-1}$.

Voor $P = book.section.subsection$ en interval$_{p_n}$ = [0.6, 0.9), wordt dan $P' = book.section$ met subinterval = $[0.6 + 0.3 * 0.3 = 0.69, 0.6 + 0.3 * 0.33 = 0.699)$. In Tabel 2.8 wordt dit verduidelijkt.
De subinterval dat door de omgekeerde rekenkundige codering berekent wordt, voldoet aan de volgende eigenschap:

> Als pad P voorgesteld wordt als een interval I, dan bevatten alle intervallen voor de achtervoegsels van P het interval I.

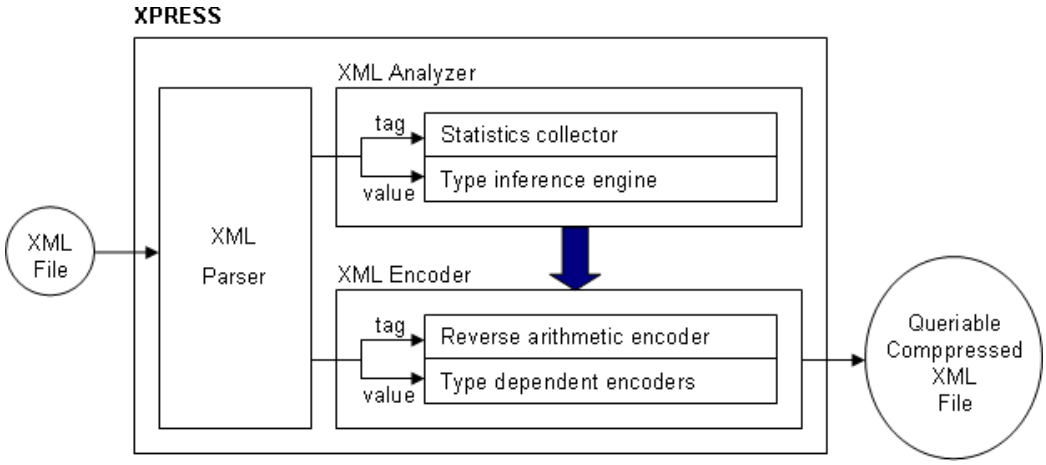
Element	Eenvoudig pad	Interval_T	Subinterval
book	book	[0.0, 0.1)	[0.0, 0.1)
section	book.section	[0.3, 0.6)	[0.3, 0.33)
subsection	book.section.subsection	[0.6, 0.9)	[0.69, 0.699)

Tabel 2.8: Tabel met subintervallen

Ter illustratie:
het interval voor *book.section.subsection* is [0.69, 0.699),
het interval voor *section.subsection* is [0.69, 0.78),
het interval voor *subsection* is [0.6, 0.9).
Dus [0.6, 0.9) $\supseteq$ [0.69, 0.78) $\supseteq$ [0.69, 0.699).
Stel nu dat de pad-uitdrukking van een querie als volgt is: *//section/subsection*, dan stelt deze uitdrukking interval [0.69, 0.78) voor. Dan zal de queryprocessor alle elementen selecteren die in dat interval liggen.

2.5.2 Werking

In Figuur 2.14 is de architectuur van de tool XPRESS te zien.



Figuur 2.14: Architectuur van XPRESS

De kern van XPRESS is de 'XML-Analyser' en de 'XML-Encoder'. Tijdens de voorbereiding van de eerste scan van de XML-data wordt de XML-Analyser aangeroepen. Deze verzamelt alle informatie die de XML-Encoder nodig heeft. De XML-Encoder genereert hierna het gecomprimeerde XML-document waarop queries toegepast kunnen worden.

De XML-Analyser bestaat uit twee delen, namelijk de 'statistics collector' en de 'type inference engine'. De 'statistics collector' berekent de frequen-

ties van de verschillende elementen die dan als input voor de omgekeerde rekenkundige codering dienen.

De 'type inference engine' leidt het type van de waarden van de verschillende elementen af en stelt vervolgens statistieken op voor de 'type dependent encoders' van de XML-Encoder.

XPRESS maakt gebruik van de volgende combinatie voor het efficiënt comprimeren en verkrijgen van de XML-data:

- *Omgekeerde rekenkundige codering*
  In Sectie 2.5.1 werd dit begrip verder uitgelegd.

- *Automatische afleiding van het type*
  Er is geen manuele tussenkomst nodig om de verschillende waarden te herkennen. Automatisch worden de beste compressors aan de verschillende waarden toegekend.

- *Verschillende coderingen voor verschillende types*
  Overeenkomstig met het type wordt op zijn waarde de beste codering toegepast.

- *Semi-adaptieve benadering*
  Het invoerdocument wordt tweemaal gescand. Een eerste keer door de XML-analyser om de statistieken te verzamelen en een tweede keer door de XML-encoder voor de uitvoering van de compressie.

- *Gelijkvormige compressie ('homomorf')*
  De structuur van het oorspronkelijk XML-bestand blijft in het gecomprimeerde document behouden.

Bij het automatisch afleiden, tijdens de voorbereiding van de eerste scan, worden de types van de waarden van de elementen door eenvoudige regels gededuceerd. De cijfers worden beschouwd als het type integer, getallen met een punt erin worden als het type float aanzien, strings waarvan het aantal verschillende waarden niet groter is dan 128, worden beschouwd als het enumeratie-type en de andere strings worden gewoon als strings aanzien.

XPRESS heeft zes coderingen voor mogelijke waarden, wetende dat de MSB (Most Significant Bit) van de codering steeds nul is. In Tabel 2.9 worden de zes verschillende coderingen toegelicht.
Op coderingen voor integers wordt eerst het getal in binaire vorm gezet waarop later de codering wordt toegepast.

Codering	Beschrijving
u8	integers waar $max - min < 2^7$
u16	integers waar $2^7 \leq max - min < 2^{15}$
u32	integers waar $2^{15} \leq max - min < 2^{31}$
f32	float
dict8	codering met een woordenboek voor enumeraties
huff	huffman codering voor algemene string

Tabel 2.9: De zes waarde-coderingen in XPRESS

Voor het coderen van de tags gebruikt XPRESS een benadering van de omgekeerde rekenkundige codering. De voorstelling van een pad is een interval in [1.0, 2.0). Bij het bekomen interval met de omgekeerde rekenkundige codering wordt gewoon 1.0 bijgeteld. Hierbij is ook steeds de MSB van de tweede byte één. Dit komt doordat de voorstelling van een float $S \times 1.M \times 2^{E-127}$ is en in binair vorm is de eerste bit een S, de acht volgende bits een E en de 23 bits erop een M.

De querie-processor kan dus door te kijken naar de MSB weten of het een waarde of een tag is. Om de start- en de eind-tags te onderscheiden is het interval $[1.0 + 0.0 = 0 \times 8000, 1.0 + 2^{-7} = 0 \times 8100)$ gereserveerd. Voor alle eindtags is er één byte 0×80 toegewezen aangezien de code voor het interval begint met 0×80. De codes voor de start-tags zijn altijd groter of gelijk aan 0×8100.

2.5.3 Beoordeling

Resultaten [4] tonen aan dat XPRESS uitstekende vooruitgang boekt in de uitvoering van queries op gecomprimeerde XML-documenten. Eveneens behaalt XPRESS een redelijk goede compressie-ratio door gebruik te maken van de omgekeerde rekenkundige codering, het afleiden van de types en het gebruik van de verschillende coderingen afhankelijk van het afgeleide type.

2.6 XQueC

De naam XQueC staat voor een XQuerie-processor en een Compressor. Waar de tools XGrind en XPRESS te kort schoten in het evalueren van queries, kan XQueC wel willekeurig complexe queries aan. XQueC is eigenlijk de eerste processor die complexe queries uit de hedendaagse wereld kan evalueren in een gecomprimeerd formaat.

Het XQueC systeem haalt zijn voordeel door gebruik te maken van het XMill principe. Hiermee wordt het afzonderlijk opslaan van gegevens en structuur bedoeld zodat er efficiënt queries kunnen gesteld worden op de gecomprimeerde data.

Het systeem hanteert ook een eenvoudig opslagmodel voor de gecomprimeerde gegevens dat geschikt is voor een reeks van toegangsmogelijkheden. Dit zorgt dat er veel alternatieven bestaan voor het evalueren van complexe queries. Er zijn hiervoor ook verschillende opslagmethoden mogelijk.

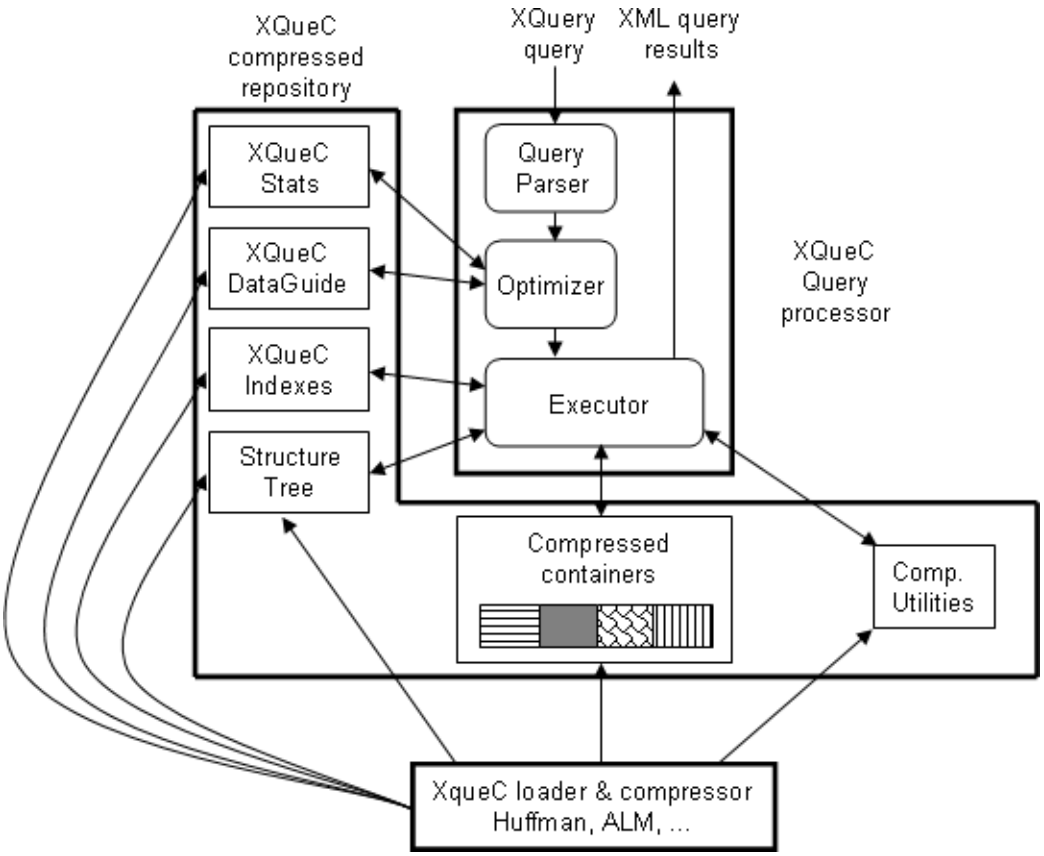
Ook hanteert het systeem een uitbreiding van het simpele algebra voor het evalueren van XML-queries om zodoende te kunnen genieten van compressie en decompressie. Deze algebra wordt zoveel mogelijk uitgebuit om zo verschillende methoden voor het evalueren van queries te kunnen genereren die de reguliere operators en compressors vrij met elkaar kunnen mengen.

2.6.1 Werking

De architectuur van XQueC is te zien in Figuur 2.15 en bevat de volgende onderdelen:

- De *loader en compressor* zetten de XML-bestanden om in een gecomprimeerd formaat, die hierna nog steeds queries kunnen evalueren.
- De *compressed repository* slaat de gecomprimeerde documenten op. Deze zorgt ook voor toegangsmogelijkheden tot de gecomprimeerde gegevens en geeft een reeks functies ter beschikking om bijvoorbeeld vergelijkingen uit te voeren op gecomprimeerde waarden.
- De *query processor* optimaliseert en evalueert queries geschreven in XQuery [18] op de gecomprimeerde gegevens. De volledige verzameling van operaties zorgt voor een efficiënte evaluatie op de gecomprimeerde opslagplaats.

Voor het comprimeren heeft XQueC zich gebaseerd op de mogelijkheid om via alternatieve evaluaties, inclusief 'top-down', 'bottom-up' en 'direct' (door



Figuur 2.15: Architectuur van XQueC

indices), tot een element te kunnen komen. Deze redenering volgt dus het principe van het scheiden van structuur en inhoud zoals dit in XMill van toepassing is. In XQueC krijgen de containers wel een dubbele werking, namelijk als gecomprimeerde opslag en als index om de toegang tot de structuur te ondersteunen.

Dit brengt echter twee gevolgen met zich mee in vergelijking met XMill. XQueC gebruikt een fijne textuur in de gecomprimeerde container. Hierbij wordt elk blad in het XML-document onafhankelijk van elkaar gecomprimeerd waardoor de opslag als een fijne textuur georganiseerd dient te worden. Dit in tegenstelling tot XMill waarbij de containers als een blok worden beschouwd. Eveneens wordt er in XQueC op waarden met dezelfde types dezelfde coderingen gebruikt. Als resultaat kunnen we profijt maken uit gelijkheden in de gegevens en is er tevens toegang tot elke gecomprimeerde waarde afzonderlijk.

Een tweede gevolg is dat we de orde van de waarden moeten behouden. Dit is nodig om vergelijkingen op de gecomprimeerde waarden te kunnen uitvoeren zonder deze eerst te moeten decomprimeren. Zo een algoritme wordt voorgesteds door *comp*-functie: voor alle x_1, x_2, $comp(x_1) < comp(x_2)$ als en slechts als $x_1 < x_2$.

Structuren van de opslag

In de opslagplaats worden er verschillende structuren gehanteerd (zie Figuur 2.16). Een overzicht van de verschillende structuren:

- *Woordenboek voor de elementen*
  Een woordenboek wordt gebruikt om de namen van de elementen en attributen te coderen. Alle verschillende namen worden omgezet in een string van bits met als lengte $\log_2(Naam)$.

- *Structuurboom*
  Om de volgorde van de knopen van het XML-document te behouden, krijgt elke knoop die geen waarde is (element of attribuut) een unieke ID. De structuurboom stelt nu een reeks van records voor met daarin zijn ID, de ID's van zijn kinderen en de ID's van zijn ouders. Ook wordt er voor een betere prestatie een zoekboom bijgehouden.

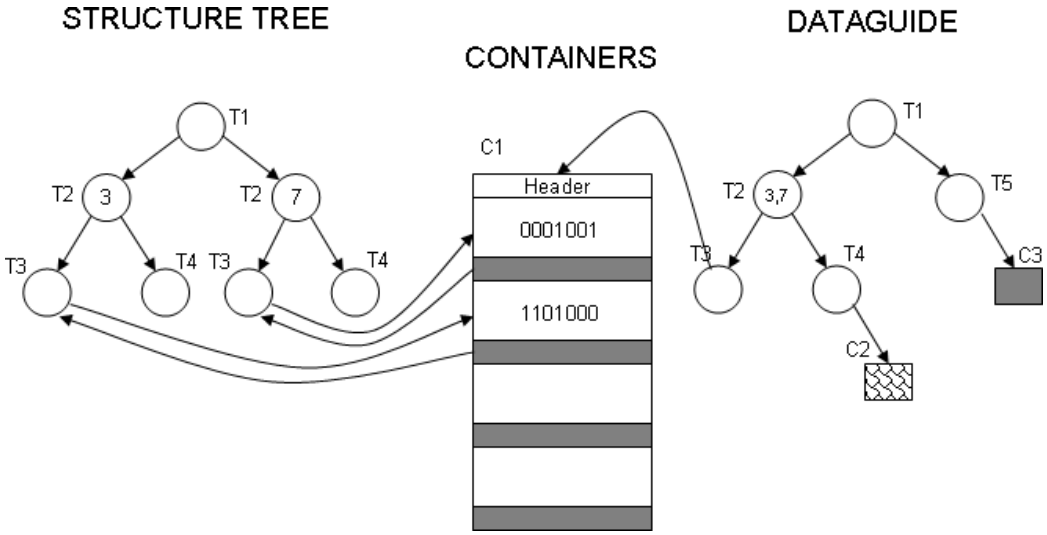
- *Containers voor de waarden*
  Alle waarden die voldoen aan dezelfde pad-uitdrukking, gezien van wortel tot blad, worden samen in gelijksoortige containers opgeslaan. We mogen dus waarden, gevonden via verschillende paden, opslaan in dezelfde container. Elke container is een reeks van records met daarin

de gecomprimeerde waarde en een pointer naar zijn ouder. Deze reeks behoudt ook zijn volgorde zoals in het oorspronkelijk document om hierna een binaire zoekactie te kunnen uitvoeren.

- *DataGuide*
  Een DataGuide [13] is een samenvatting van de structuur van alle mogelijke paden in het document.

- *Andere indices en statistieken*
  Wanneer een document geladen wordt, kunnen andere indices of statistieken aangemaakt worden.

- *Alternatieven voor de opslag*
  Er zijn vele verschillende opslagmethodes voor XML-bestanden, zolang deze maar het bestaan van containers en de toegang tot de elementen ervan toelaat.



Figuur 2.16: Structuur van de opslagplaats

De verschillende compressors die XQueC gebruikt, worden telkens aan één bepaalde type van waarden toegekend. Dit is om op deze waarden het gebruik van vergelijkingen toe te laten. Dus voor het type integer wordt een codering gekozen die voor alle waarden van het type integer van toepassing zal zijn.

Bij het coderen van strings wordt vooral gebruikt gemaakt van methodes die een woordenboek hanteren en die de volgorde respecteren. Het algoritme *ALM* is hiervan een voorbeeld. Deze gebruikt een woordenboek waarin verschillende strings dezelfde prefix mogen hebben en waarbij dezelfde strings

meerdere codes mogen bevatten. De niet-adaptieve Huffman-codering wordt ook gebruikt voor de codering van strings.

Voor het gebruik van queries is er de 'XQueC Query Processor' die een 'querie-parser', een 'optimizer' en een 'evaluator' bezit. De optimizer gebruikt een reeks van logische en fysieke operators. Er zijn drie soorten fysieke operators die XQueC gebruikt:

- *Operators voor compressie/decompressie*
  Om te weten welke compressor gebruikt wordt, wordt er een status-token toegevoegd aan het resultaat.

- *Operators voor datatoegang*
  Deze bibliotheek bevat operaties die toegang verlenen tot de structuurboom. Zo is er een operatie die de wortel van een document teruggeeft en is er een operatie die ouders en kinderen van een bepaalde lijst van ID's teruggeven. De functie $dg_{acc}(d, p)$ heeft als parameters een document d en een pad p die een lijst van pointers teruggeeft. Dit zijn de pointers naar de records van elementen of attributen die aan het pad p voldoen.
  Er zijn ook container-operatoren. Deze operatoren kunnen zorgen voor een container-scan of rekening houden met een gegeven pad die al dan niet voorzien zijn van een conditionele expressie. Er is ook een operator die de hele inhoud van een container teruggeeft.

- *Operators voor vergelijkingen*
  Deze bibliotheek bevat de $\bowtie$ (join) en de σ (selectie). Deze kunnen zowel op gecomprimeerde als op gewone data werken. Het is de taak van de 'Optimizer' om te beslissen welke operator te gebruiken en zo ervoor te zorgen dat de waarden die gebruikt worden ook dezelfde compressor-status hebben.

2.7 Delen van gemeenschappelijke subbomen

Talen om XML-bestanden te querieën zoals XPath maken hoofdzakelijk gebruik van de boom-structuur van het XML-document. Bij XMill (zie Sectie 2.3) komt deze onderscheiding tussen de structuur en de gegevens ook terug. De boom-structuur noemen we het 'skelet' van het document. Bij XMill wordt het skelet enkel gebruikt voor de compressie van de gegevens daar wij hier het skelet gaan aanwenden voor het efficiënt querieën van de gegevens.

De efficiëntie van het evalueren van queries hangt echter af of het skelet in het geheugen past. De skeletten van XML-documenten zijn doorgaans redelijk klein, maar voor grote XML-documenten met miljoenen knopen zijn de boom-skeletten te groot. De oplossing hiervoor is om de boom-skeletten te verkleinen, gebaseerd op het delen van gemeenschappelijke subbomen zodat pad-queries mogelijk worden op deze gecomprimeerde skeletten.

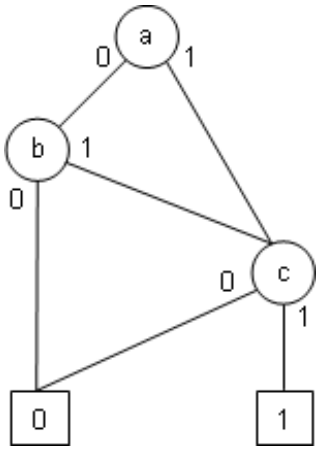
De techniek van het comprimeren van boom-skeletten door het delen van gemene subbomen kan gezien worden als een veralgemening van de compressie van Booleanse functies in 'Ordered Binary Decision Diagrams' (OBDDs) [10] of vrij vertaald 'Geordende Binaire Beslissingsdiagrammen'. Zo kunnen we de nodige algoritmes van de OBDDs gebruiken die dan de basis voor nieuwe algoritmes waardoor directe evaluatie kan gebeuren van de pad-queries op gecomprimeerde skeletten.

Een OBDD is een boom of een DAG met een root waarbij elke knoop een binaire beslissing voorstelt. Deze gegevensstructuur is dus een gerichte, acyclische graaf met een root, waarbij elke tussenliggende knoop een Booleaanse variabele voorstelt die elk nog twee kinderen hebben. De edge van een knoop naar zijn kind, stelt een toekenning van de variabele voor van '0' of '1'. De OBDDs volgen deze twee regels: Twee kinderen van dezelfde ouder mogen niet dezelfde varabele voorstellen. Er mogen geen isomorfe subbomen voorkomen aangezien deze samengenomen moeten worden. In Figuur 2.17 is de OBDD van de Booleaanse functie $F = (a + b)c$ voorgesteld.

2.7.1 Voorbeeld

Bekijken we hier een illustratie van een gecomprimeerde skelet van een XML-document. Het XML-document stelt een simpele gegevensbank voor:

```
<bib>
    <book>
        <title>Foundations of Databases</title>
        <author>Abiteboul</author>
```

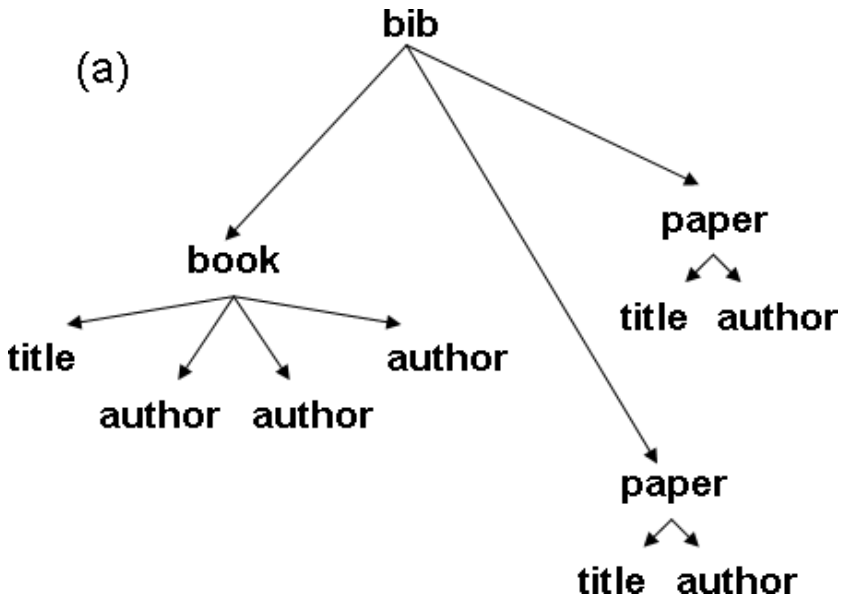


Figuur 2.17: OBDD van $F = (a + b)c$

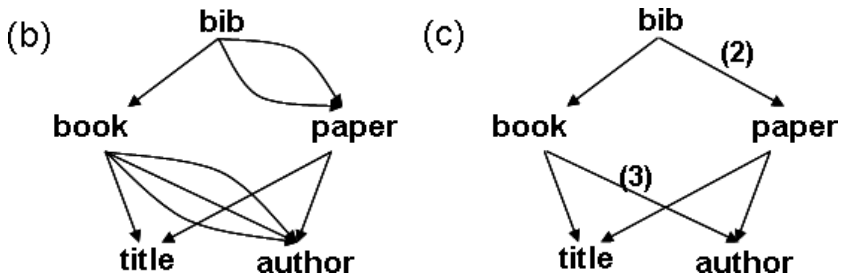
```
        <author>Hull</author>
        <author>Vianu</author>
    </book> <paper>
        <title>A Relational Model for Large Shared
        Data Banks</title>
        <author>Codd</author>
    </paper> <paper>
        <title>The Complexity of Relational Query
        Languages</title>
        <author>Vardi</author>
    </paper>
</bib>
```

In Figuur 2.18(a) is het originele skelet van het bovenstaande XML-document te zien. In Figuur 2.19 zijn de verkleinde versies van dit skelet (a) te zien. In skelet (b) is een gecomprimeerde skelet bekomen door de gemeenschappelijke subbomen te delen. Het is ook belangrijk om op te merken dat de volgorde van de uitgaande edges behouden moet worden. Met andere woorden kan de oorspronkelijke skelet (a) bekomen worden door skelet (b) te doorlopen via de 'depth-first' techniek.

Meer compressie zoals skelet (c) in Figuur 2.19, wordt bekomen door het vervangen van dezelfde egdes door één edge waaraan een teller wordt meegegeven.
De verkleinde voorstelling van het skelet toont de volgende interessante eigenschap van de meest gebruikte XML-documenten: XML-bestanden bezitten



Figuur 2.18: Een boom-skelet



Figuur 2.19: Twee gecomprimeerde skeletten

een regelmatige structuur waarin veel herhaling van subbomen voorkomen. Als we een voorbeeld van een bestand met extreme regelmaat zouden bekijken, als voorbeeld een XML-gecodeerde relationele tabel met R rijen en C kolommen, dan zou het skelet ervan een grootte $O(C \times R)$ hebben. Het verkleinde skelet zoals in Figuur 2.19 (b) heeft dan grootte $O(C + R)$. Het nog meer verkleinde skelet, door het samennemen van dezelfde edges zoals in Figuur 2.19 (c) heeft grootte $O(C + \log R)$.

Gecomprimeerde skeletten zijn eenvoudig te berekenen en maken het mogelijk om op een natuurlijke manier de gegevens te ondervragen. Elke knoop in het verkleinde skelet stelt een reeks van knopen in de oorspronkelijke boom voor. Dit wordt ook wel 'bisimilatie' genoemd. Het doel van een pad-querie is om een reeks knopen te selecteren uit de originele boom. Echter kan deze bekomen set voorgesteld worden door een subset van knopen uit een deels gedecomprimeerd skelet.

De verkleinde skeletten kunnen tot 10% groot zijn ten opzichte van de oorspronkelijke bomen. Dit geldt zelfs voor de XML-bestanden met miljoenen knopen. Ze passen dus volledig in het geheugen zodat er zonder problemen queries op uitgevoerd kunnen worden.

2.7.2 Meer detail

Voor het comprimeren van XML-bestanden met de techniek van het 'delen van de gemeenschappelijke subbomen' worden gegevens en structuur van het document uiteen gehaald en worden deze afzonderlijk verwerkt. De gewone gegevens worden opgeslagen en geïndexeerd via de gebruikelijke methoden. Terwijl de structuur van het XML-document als een boom of 'skelet' wordt opgeslaan, waarbij de knopen de namen van de elementen en van de attributen behouden. Het skelet is van groot belang voor het stellen van queries aangezien deze boom de structuur van het XML-document behoudt.

In deze sectie wordt toegelicht hoe een skelet tot een minimum wordt verkleind zodat deze volledig in het geheugen past en er efficiëntere queries op gesteld kunnen worden zonder volledige decompressie.

Definities en notaties

Om een XML-document voor te stellen, zowel in het origineel formaat als verkleind formaat, worden 'instanties' gebruikt. Zo een instantie I is van de vorm: $I = (V, \gamma, root, S_1, ..., S_n)$ waarbij V de set van knopen voorstelt. De

functie $\gamma : V \rightarrow V^*$ kent elke knoop aan zijn kinderen toe, *root* de wortel van de boom is en $S_1, ..., S_n$ subsetten van V zijn.

De setten van knopen worden gebruikt om namen van elementen en van attributen voor te stellen of om eigenschappen (vb. het vergelijken van strings, het resultaat van subqueries, ...) voor de gecomprimeerde bewerkingen en queries af te leiden. De setten krijgen daarom willekeurige namen waardoor ze gemakkelijk te onderscheiden zijn.

Ter illustratie geven we hier de instantie I horende bij het voorbeeld uit Sectie 2.7.1. In Figuur 2.20 is de voorstelling van deze instantie I te zien.

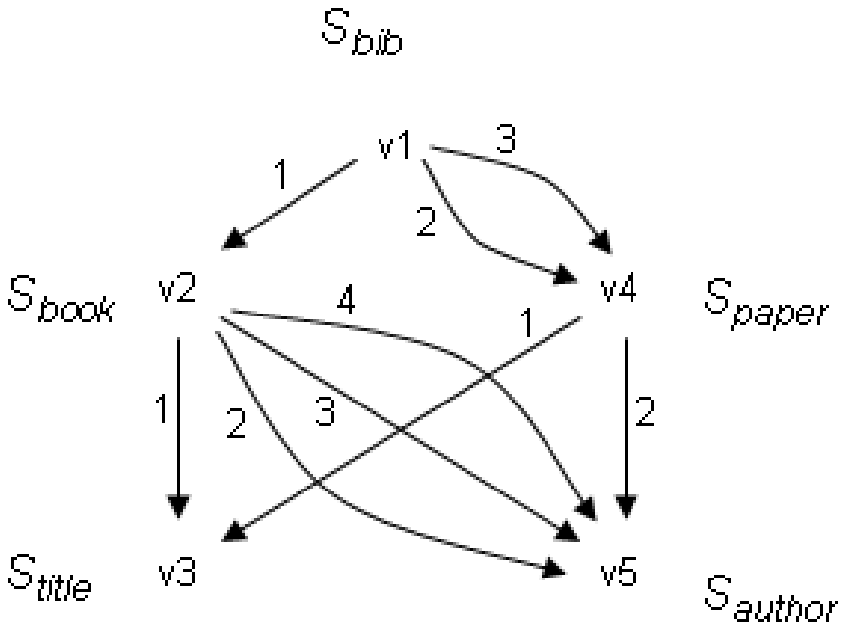
Instantie I:

- $V = v_1, v_2, v_3, v_4, v_5$
- $\gamma(v_1) = v_2 v_4 v_4$
- $\gamma(v_2) = v_3 v_5 v_5 v_5$
- $\gamma(v_3) = \oslash$
- $\gamma(v_4) = v_3 v_5$
- $\gamma(v_5) = \oslash$
- De setten zijn $S_{bib}, S_{book}, S_{title}, S_{paper}, S_{author}$

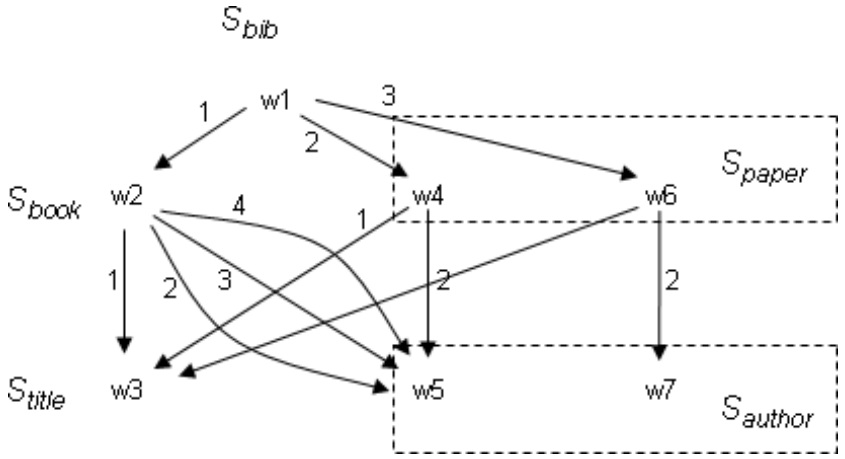
We noteren $v \xrightarrow{i} w$ als de knoop w voorkomt op positie i van $\gamma(v)$. Als de knopen v_0 en v_n voorkomen in een instantie I en er is ook een reeks van knopen $v_1, ..., v_{n-1}$ zodat $v_{j-1} \xrightarrow{ij} v_j$ voor $j = 1, ..., n$, dan wordt de getallenreeks $i_1, ..., i_n$ een 'edge-pad' genoemd van v_0 naar v_n.

Ook geldt er voor elke knoop $v \in V$ en voor elke deelset $S \subseteq V$, dat $\Pi(S) = \bigcup_{v \in S} \Pi(v)$ waarbij $\Pi(v) = \{P | P \text{ is een edge-pad van } root \text{ naar } v\}$. In Figuur 2.20 zijn deze edge-paden eenvoudig te herkennen daar er indices zijn boven de edges.

In Figuur 2.21 is een voorstelling van een andere instantie J te zien. Deze instantie J is equivalent met instantie I uit Figuur 2.20. De edge-paden voor elke set zijn hetzelfde binnen een Schema $\sigma = \{S_1, ..., S_n\}$. Dus twee Σ-instanties, I en J zijn equivalent als $\Pi(V_I) = \Pi(V_J)$ en $\Pi(S_I) = \Pi(S_J)$ voor $S \in \sigma$.
In het originele, volledig gedecomprimeerde, skelet is er telkens een uniek edge-pad van de wortel naar elke knoop wat de eigenlijke boomweergave van



Figuur 2.20: De voorstelling van de instantie I



Figuur 2.21: De voorstelling van een instantie J

het XML-document is. Voor elke instantie I is er nu exact één boom-instantie $T(I)$ die equivalent is aan I.

Elke knoop in het gecomprimeerde skelet stelt een set van knopen voor in de gedecomprimeerde boom. Gecomprimeerde en niet gecomprimeerde instanties zijn aan elkaar gerelateerd door de eigenschap van 'bisimilair'. Een bisimilaire relatie $\sim$ voor een σ-instantie is een equivalente relatie: $v \sim w$ (voor alle $v, w \in V$)

- voor alle i, als $v \xrightarrow{i} v'$ dan bestaat er een $w' \in V$ zodat $w \xrightarrow{i} w'$ en $v' \xrightarrow{i} w'$, en
- voor alle $S \in \sigma : v \in S \Leftrightarrow w \in S$

Voor een instantie I en de bisimilaire relatie $\sim$ op I, dan is $I/\sim$ de instantie verkregen als volgt: vervang de knopen in V_I door zijn equivalente klassen van knopen met betrekking tot $\sim$. In andere woorden plaats alle knopen $v, w \in W$ waarbij $v \sim w$ in een enkele knoop met naam "$v \sim w$".

Voor alle instanties I en de bisimilaire relatie $\sim$ gedefinieerd op I, is nu I equivalent met $I/\sim$. Voor elke instantie I bestaat er een bisimilaire relatie $\sim$ op de boom-instantie $T(I)$ zodat I isomorf is met $T(I)/\sim$. Dit gaat opS omdat de paden van de wortel tot de knopen in de boom-instantie en de andere instanties gelijk zijn.

Verder geldt er dat voor elke instantie I er exact één 'minimale' instantie $M(I)$ bestaat die equivalent is met I. Dus er is geen instantie die equivalent is met I met minder knopen dan $M(I)$. Als een boom-instantie $T(I)$ een XML-document voorstelt, dan stelt de $M(I)$ zijn gecomprimeerde instantie voor. Andere equivalente instanties zijn reeds gedeeltelijk gedecomprimeerd.

Er bestaat een algoritme die $M(I)$ berekent met een gegeven instantie I in lineaire tijd.

2.8 Samenhangende deelgraven

In vorige Sectie 2.7 hebben we gezien hoe we XML-documenten kunnen comprimeren door het delen van de gemeenschappelijke subbomen. Hierbij wordt een identieke herhalende deelboom voorgesteld door een pointer naar de eerst voorkomende deelboom. Op deze manier is de grootte van de minimale unieke DAG gemiddeld 1/10 ten opzichte van de grootte van de originele boom.

De methode die we verder gaan bespreken, is gebaseerd op gemeenschappelijke deelgraven waarbij herhalende patronen in de boom vervangen worden door een geschikte pointer. We gaan hiervoor de boom voorstellen door een context-vrije grammatica. Het resultaat is meer dan de helft kleiner dan de minimale DAG.

2.8.1 Context-vrije boom grammatica

We weten reeds dat de minimale DAG van een boom t:

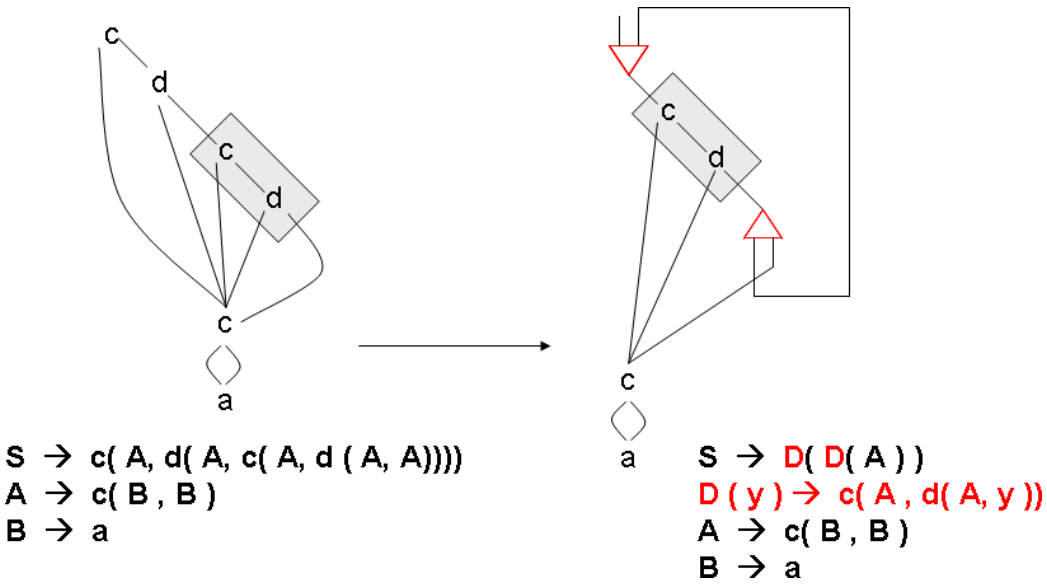
- uniek is
- berekend kan worden in lineaire tijd
- ten hoogste exponentieel kleiner dan t is
- equivalent is met de minimale reguliere boom grammatica voor t (zie Figuur 2.22 links)

Een veralgemening van het delen van identieke deelbomen is het delen van willekeurige patronen of samenhangende deelgraven in een boom (zie Figuur 2.22).

Voor een boom t is de minimale deelgraaf [11]:

- niet uniek
- berekenbaarheid is NP-compleet
- ten hoogste dubbel exponentieel kleiner dan t
- equivalent met de minimale context-vrije (cv) boom grammatica van t (zie Figuur 2.22 rechts).

Bij een cv boom grammatica kunnen de niet-terminalen zich in de boom bevinden en niet enkel aan de bladeren. Dit is in tegenstelling met de reguliere versie. De parameters $y_1, y_2, ...$ worden in de productie gebruikt om aan te geven waar de deelbomen van de niet-terminalen moeten geplakt worden.



Figuur 2.22: (links) De minimale DAG en de reguliere boom grammatica; (rechts) De minimale deelgraaf en de context-vrije boom grammatica

Het vinden van de minimale deelgraaf voor een gegeven boom is gelijk aan het zoeken naar de minimale cv boom grammatica die de boom produceert. Dit probleem is echter NP-compleet. Het zoeken naar een kleine cv boom grammatica kan nochtans in lineaire tijd gevonden worden. Hiervoor bekijken we het algoritme BPLEX (zie Sectie 2.8.2) die een kleine voorstelling teruggeeft van een gegeven boom.

2.8.2 BPLEX

Het algoritme BPLEX (= Bottum-up multiPLExing) maakt gebruik van 'Straight-Line' (SL) context-vrije boom grammatica's. Deze grammatica's garanderen dat ze maximaal één boom genereren. Meer formeel is een grammatica Straight-Line, als er een lineaire orde $A_1, ..., A_n$ van zijn niet-terminalen is zodat voor elke productie $A_k \rightarrow rhs$ alle niet-terminalen in rhs een index j hebben met $j > k$.

Voorbeeld:
$A_1 \rightarrow A_2(A_3, a, A_3)$
$A_2(y_1, y_2, y_3) \rightarrow A_5(y_1, y_2, A_6(y_3, y_3))$
Verder wordt een cv boom grammatica 'regulier' genoemd, als alle niet-terminalen rang 0 hebben. Deze wordt 'lineair' genoemd als voor elke productie $A(y_1, ..., y_k) \rightarrow t$ elke parameter y_i ten hoogste één keer voorkomt in t. We verkorten verder een SL lineaire cv boom grammatica tot SLT.

Het idee van BPLEX is om patronen, die elkaar niet overlappen, bottom-up te vinden die meermaals voorkomen in de input-grammatica en deze te vervangen door nieuwe niet-terminalen die de overeenkomstige patronen genereren.

Het algoritme BPLEX (zie Figuur 2.23) neemt een willekeurige SL reguliere boom grammatica als input en geeft een (kleinere) SLT grammatica als output terug. Er worden ook volgende drie parameters meegegeven: het maximale aantal K_N knopen en producties die onderzocht worden vanuit een gegeven knoop, de maximale grootte K_S van een nieuwe productie en de maximale rang K_R van een nieuwe productie.

Als $A_1, ..., A_l$ de niet-terminalen in SL-orde zijn van de SLT input-grammatica G dan verwijst $<_G^l$ naar de post-orde volgorde over de knopen van $rhs_G(A_1), ..., rhs_G(A_l)$. z geeft de huidige positie weer met betrekking tot deze orde. Tijdens elke stap berekent BPLEX een reeks van herhalende matches (RepM) door het vergelijken van de patronen op positie z met de rechter leden van de laatste K_N producties van G en dit met een index groter dan l. Een reeks van nieuwe matches (NewM) wordt berekent door het vinden van een paar niet overlappende voorkomens van patronen vanuit positie z en dit op de K_N meest recente bezochte knopen. Dit laatste gebruikt de techniek van de 'sliding window' uit het LZ77-compressie algoritme. Als er minstens één match gevonden is, voert BPLEX het delen ervan uit, wat de grootste reductie van de grammatica veroorzaakt. Bij het niet vinden van een match wordt er naar de volgende knoop gegaan en bij geen volgende knoop, wordt de huidige SLT grammatica teruggegeven.

```
Procedure BPLEX(G:grammar, K_N: int, K_S: int, K_R: int): grammar
begin
    A_l := last symbol in the SL ordening of G
    z := leftmost leaf of rhs_G(A_l)
    while true do
        repM := RepM(G,z,K_N)
        newM := NewM(G,z, K_N, K_S, K_R)
        if newM ≠ Ø or repM ≠ Ø then
            m := max(newM, repM)
            if m ∈ repM then
                G := G[m ← A], with rhs_G(A) = p_m      Vervang patroon door
            else                                        Non Terminal
                k := rank(p_m)
                A := fresh(G,k)
                G := add(G,A(y_1, ... , y_k) → p_m)     Vervang patroon door
                G := G[m, c_m ← A]                      Nieuw Non Terminal en
            fi                                          voeg nieuwe productie toe
        elseif ∃ w∈ z <^l_G w then z := next(<^l_G, z)
        else break
        fi
    od
    return G
end BPLEX
```

Figuur 2.23: Het algoritme BPLEX

Illustratie

Als verduidelijking van de werking van het algoritme BPLEX, passen we dit toe op de reguliere boom grammatica G uit Figuur 2.22 (links). In de laatste en tweede productie wordt er niets gedeeld. Vervolgens scant BPLEX de eerste productie. Wanneer de hoogste d wordt aangetroffen op adres (1, 2) (adres(u, v): met u de productie en v het pad naar de knoop in productie u), wordt ook de match van het patroon $d(A, y_1)$ gevonden, samen met zijn gezel van de laagste d. Deze heeft grootte '1' en wordt vervangen. Er wordt een nieuwe niet-terminaal D met rang 1 toegevoegd aan G: $D(y_1) \rightarrow d(A, y_1)$. De twee matches worden vervangen en de eerste productie S bekomt de productie $S \rightarrow c(A, D(c(A, D(A))))$. Het nieuwe patroon $rhs(D)$ heeft geen matches met de nieuw bekomen grammatica op adres $z = (1, 2)$ en er werden eveneens geen nieuwe paar matches gevonden. Bijgevolg wordt z veranderd in de startproductie S met adres $z = (1, \epsilon)$.

De $rhs(D)$ heeft geen match, maar het patroon $c(A, D(y_1))$ heeft een match op adres $(1, \epsilon)$ en op $(1, 2.1)$. Zodoende wordt een nieuwe niet-terminaal E van rang 1 toegevoegd met zijn productie $E(y_1) \rightarrow c(A, D(y_1))$. Beide matches worden vervangen door E. We bekomen nu het resulterende grammatica G':

$S \rightarrow E(E(A))$ $D(y_1) \rightarrow d(A, y_1)$
$A \rightarrow c(B, B)$ $E(y_1) \rightarrow c(A, D(y_1))$
$B \rightarrow a$

Zowel de bekomen grammatica G' als de cv boom grammatica in Figuur 2.22 (rechts) hebben grootte '7' waarbij de edges naar de parameters niet in rekening gebracht worden. Merk op dat het algoritme BPLEX het patroon $p = c(A, d(A, y_1))$ niet gevonden heeft zoals in Figuur 2.22 (rechts). Het kleinere patroon $d(A, y_1)$ wordt eerder vervangen voordat p volledig gescand kan worden.

Hoofdstuk 3

Query-evaluatie over gecomprimeerde XML-documenten

In dit hoofdstuk worden de verschillende XML-compressietools met elkaar vergeleken. De querie-evaluatie is hiervan een belangrijk onderdeel, daar een tool interessanter is als het gecomprimeerde XML-bestand niet of gedeeltelijk uitgepakt moet worden vooraleer er queries op afgevuurd kunnen worden.

3.1 XMill

De tool XMill is vooral ontworpen voor data-archivering en data-uitwisseling. XMill heeft een betere compressie dan Gzip, tot een factor 2. Echter kunnen op de gecomprimeerde XMill-bestanden geen queries gesteld worden. De relevante containers dienen eerst gedecomprimeerd te worden.

3.1.1 XMill vs. Gzip

In deze sectie worden de compressie-verhouding en compressie/decompressietijden van XMill ten opzichte van Gzip onder diverse instellingen bekeken.

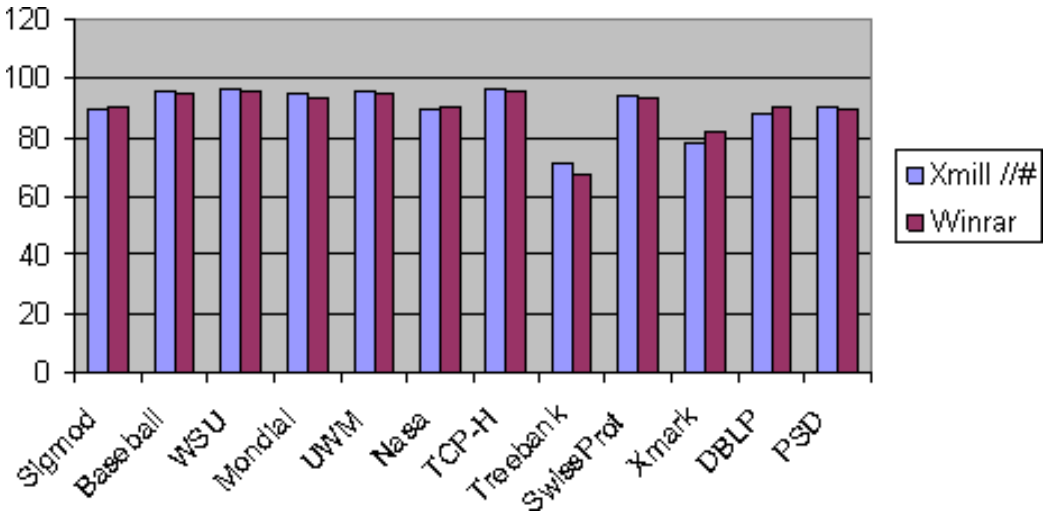
Voor documenten die groot in omvang zijn en minder tekstblokken bevatten, comprimeert XMill met de standaard instellingen tot 45% à 60% beter dan Gzip. Bij het gebruik van semantische compressors wordt de bestandsgrootte door middel van XMill verlaagd tot circa 35% à 47% in tegenstelling tot Gzip. Voor de bestanden met meer tekstblokken presteert iets beter dan Gzip.

Voor decomprimeren met beide tools, gebruikmakend van 'Xdemill' en 'Gun-

zip', is de snelheid vergelijkbaar. Merk op dat XMill snelheid verliest bij het samenvoegen van data in de verschillende containers.

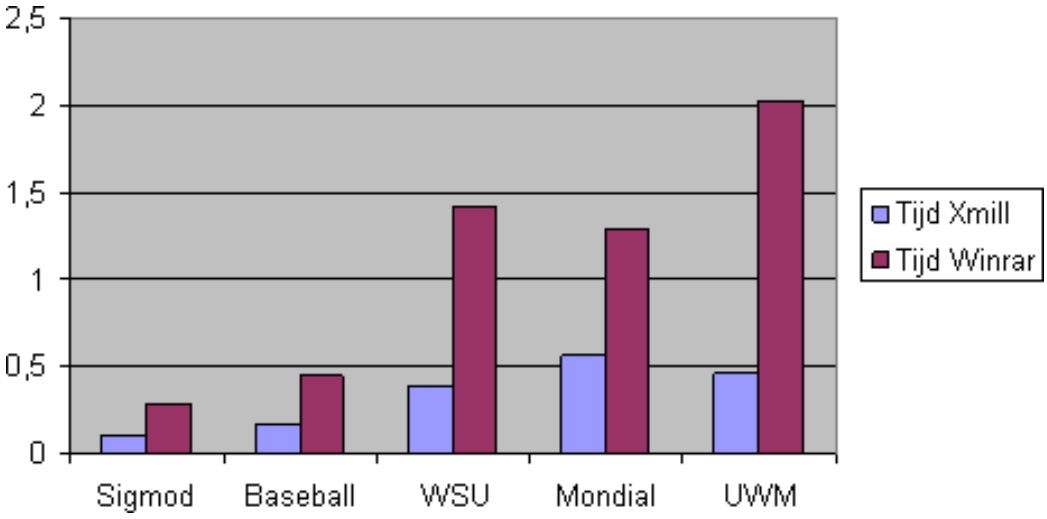
3.1.2 XMill vs. Winrar

In Figuur 3.1 zijn de compressie-factoren van XMill en Winrar zichtbaar. Het is duidelijk dat de factor van XMill een gemiddelde heeft van 90,17 % en van Winrar is deze gemiddeld 88,86 %. XMill heeft dus een kleine verbetering in de compressie, wat vooral te wijten is aan de grootte van het XML-bestand en de kleinere hoeveelheid tekstblokken.

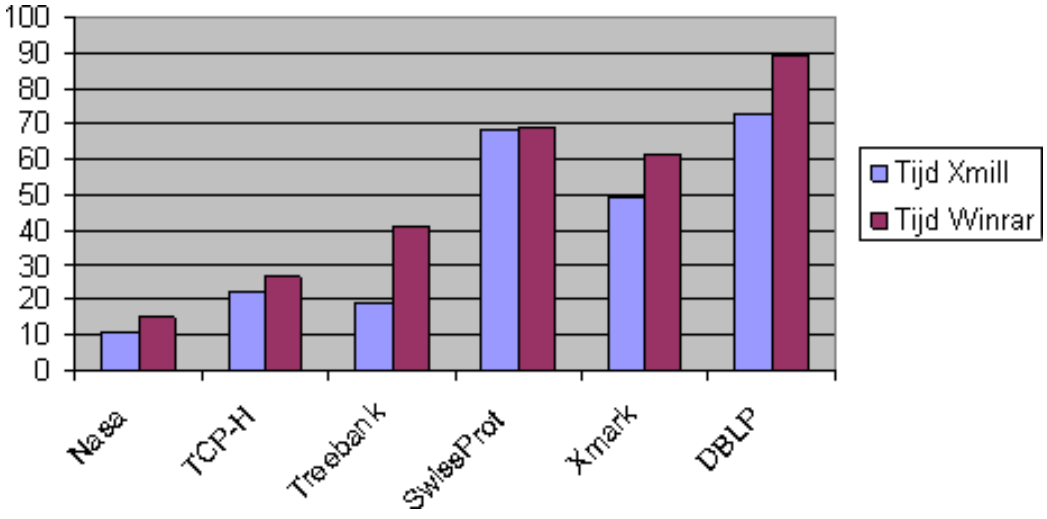


Figuur 3.1: Compressie-factor van XMill vs. Winrar

In Figuur 3.2 en in Figuur 3.3 zijn de tijden voor de compressie zichtbaar. Hier heeft duidelijk Winrar meer tijd nodig voor de eigenlijke compressie dan XMill.



Figuur 3.2: Compressie-snelheid van XMill vs. Winrar (kleine bestanden)



Figuur 3.3: Compressie-snelheid van XMill vs. Winrar (grote bestanden)

3.2 XGrind

De tool XGrind is de pionier in het stellen van queries op gecomprimeerde XML-bestanden. XGrind zondert de gegevens niet af van de structuur waardoor een gecomprimeerd XML-bestand steeds een XML-bestand blijft. De tags zijn wel gecodeerd met een woordenboek en de waarden zijn samengedrukt met het algoritme van Huffman. De structuur wordt echter niet gewijzigt in het gecomprimeerd document waardoor de orde behouden blijft en queries hierop mogelijk worden.

De querie-processor kan beschouwd worden als een uitgebreide SAX-parser die in staat is om 'exacte-match' (zie Sectie 3.2.1) en 'prefix-match' op gecomprimeerde waarden toe te passen. Ook kan er 'gedeeltelijk-match' en 'bereik-match' (zie Sectie 3.2.2) op gedecomprimeerde waarden verwerkt worden.

Echter kan XGrind sommige operaties op gecomprimeerde bestanden niet ondersteunen zoals ongelijkheidsselecties, joins, aggregaties, geneste queries of samengestelde operaties.

3.2.1 Exacte-match querie

Een voorbeeld van een exacte-match querie wordt hier aangehaald en uitgeschreven in XML-QL [12]:

```
CONSTRUCT <student rollno=$r>
    WHERE
        <student rollno=123456789>
            <name>$n</name>
            <year>$y</year>
            </dept name=$d>
        </student> IN "student.xml",
     CONSTRUCT <name>$n</name>
    </student>
```

Deze querie geeft de namen van de studenten terug waarvan de *roll*-nummer gelijk is aan 123456789.

3.2.2 Bereik querie

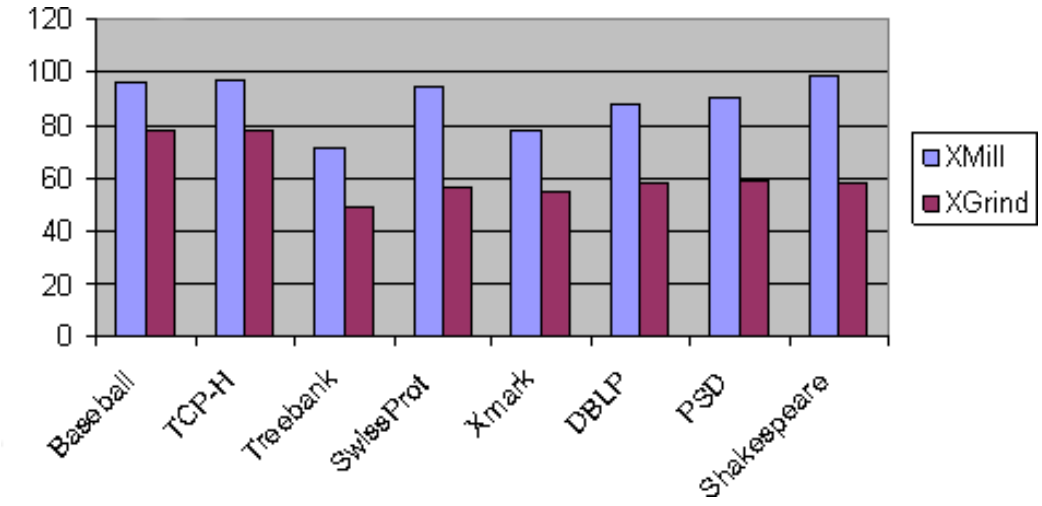
Hier wordt een eenvoudig voorbeeld gegeven van een bereik-querie die alle studenten teruggeeft waarvan de data ligt tussen 1998 en 2000.

```
CONSTRUCT <student rollno=$r>
    WHERE
        <student rollno=$r>
            <name>$n</name>
            <year>$y</year>
            </dept name=$d>
        </student> IN "student.xml",
        $y ≥ 1998 and $y ≤ 2000
    CONSTRUCT <name>$n</name>
</student>
```

3.2.3 XGrind vs. XMill

Daar de compressor XMill geen queries op het gecomprimeerd document kan uitvoeren, hebben we enkel de compressie-verhouding vergeleken.

In Figuur 3.4 zijn de compressie-factoren van XGrind en XMill te zien. XGrind heeft een lagere compressie-ratio dan XMill met een gemiddelde factor van 61,56% ten opzichte van XMill. In het slechtste geval is de factor 49% ten opzichte van XMill. Deze resultaten gelden voor een grote verscheidenheid van documenten. Als het XML-bestand veel enumeraties bevat dan verbetert de compressie-ratio.



Figuur 3.4: Compressie-factor van XGrind en XMill

De tijd van het compressie-proces van XGrind ligt steeds binnen de grens van tweemaal de tijd van XMill. Dit is te verklaren doordat XGrind het

bestand tweemaal moet doorlopen en XMill slechts éénmaal. Bij bestanden met veel tekst ligt de compressie-tijd van XGrind binnen anderhalf de tijd van XMill. Dit is verklaarbaar doordat XMill het document slechts één keer moet doorlopen en geen twee keer zoals bij XGrind.

De exacte-queries, toegepast in XGrind, zijn veel sneller dan de bestanden die XMill of Gzip eerst moet decomprimeren. We spreken hier over een gemiddelde snelheidstoename van 80% sneller. Dit resultaat betekent dat XGrind beter werkt dan XMill of Gzip. Zelfs als deze tools zouden beschikken over een algoritme die 'nul' tijd nodig heeft om exacte-match queries op ongecomprimeerde XML-documenten uit te kunnen voeren. Verder heeft XGrind minder ruimte nodig om een querie te voltrekken dan XMill of Gzip.

Voor bereik-queries waarbij een deel van het document gedecomprimeerd moet worden, is gemiddeld, tot 50%, sneller dan XMill.

3.3 XPRESS

Om queries op gecomprimeerde data te evalueren in XPRESS is er een querie-processor voor handen. Deze querie-processor verdeelt een lange pad-uitdrukking in kortere paden. Deze pad-uitdrukking wordt dus in een reeks intervallen omgezet, gebruikmakend van de 'benaderde omgekeerde rekenkundige codering'.

Door gebruik te maken van een reeks intervallen, test de querie-processor de elementen in het gecomprimeerde XML-bestand. Deze test controleert als de gecodeerde waarden al dan niet in een interval voorkomen.

De waarden in de condities van een exacte-match querie worden omgezet naar gecodeerde waarden. Vervolgens gaat de querie-processor de elementen zoeken die aan de uitdrukking en de waarde voldoen zonder enige decompressie.

Voor de bereik-match condities, worden de elementen die getallen zijn, gecodeerd door een numerieke codering. Echter voor elementen met tekst is er een gedeeltelijke decompressie nodig vermits de codering voor gewone tekst werd gebruikt (zoals 'huff' en 'dict8'). Deze behouden verder geen informatie over de volgorde van de waarden.

3.3.1 XPRESS vs. XMill

De tool XMill kan geen queries evalueren op de gecomprimeerde data, maar is vooral gespecialiseerd in het verkleinen van de XML-data. Een volledige decompressie is dus nodig vooraleer er queries op uitgevoerd kunnen worden.

Doordat XMill een woordenboek-codering gebruikt voor de structuur, de semantisch gerelateerde waarden groepeert in containers en vervolgens comprimeert gebruikmakend van Zlib, bereikt XMill de beste compressie-ratio met een gemiddelde van 92%. XPRESS daarentegen heeft een compressie-ratio van 73%. Dit is vooral te wijten aan het feit dat XPRESS de types afleidt om de meest geschikte methodes voor compressie te kunnen toepassen. Deze techniek werkt vooral goed als de waarden enumeraties, floats of integers zijn.

Ook de compressie-tijd is beter in XMill doordat deze Zlib gebruikt.

In XPRESS wordt gebruik gemaakt van de automatische afleiding van de types die geen menselijke tussenkomst meer nodig hebben om de waarden van de types manueel te interpreteren en wat wel in XMill nog nodig is.

3.3.2 XPRESS vs. XGrind

In XGrind is het mogelijk om queries op de gecomprimeerde gegevens uit te voeren, zo is de extra kosten om het hele bestand te decomprimeren verdwenen. Echter blijven de extra kosten voor de handhaving en de evaluatie van eenvoudige paden naar elk element bestaan zoals deze bestaan op het ongecomprimeerde XML-document. Door het gebruik van de omgekeerde rekenkundige codering in XPRESS worden deze extra kosten door de gedeeltelijke decompressie tot zijn minimum herleid.

De compressie-tijden in XGrind is langer dan in XPRESS. Dit komt doordat XGrind gebruik maakt van een XML-Schema (nl. DTD) om de compressors voor de waarden te bepalen en XPRESS gebruikt gekende coderingen via eenvoudige regels.

Evaluatie

Beschouwen we drie XML-documenten. De kenmerken zijn zichtbaar in Tabel 3.1. *Grootte* geeft de grootte van het XML-bestand in megabytes weer. *Diepte* is de lengte van het langste pad. *Tags* geeft het aantal verschillende elementen aan. *Getallen* geeft het aantal verschillende elementen aan die van het type integer of float zijn. *Enumeraties* staat voor het aantal verschillende elementen die enumeraties zijn.

Document	**Grootte**	**Diepte**	**Tags**	**Getallen**	**Enumeraties**
Baseball	17.06	6	46	19	5
Course	12.28	6	18	5	4
Shakespeare	15.3	5	21	0	0

Tabel 3.1: Kenmerken van drie XML-documenten

We gaan op deze documenten telkens vier types van queries uitvoeren. Deze zijn de volgende:

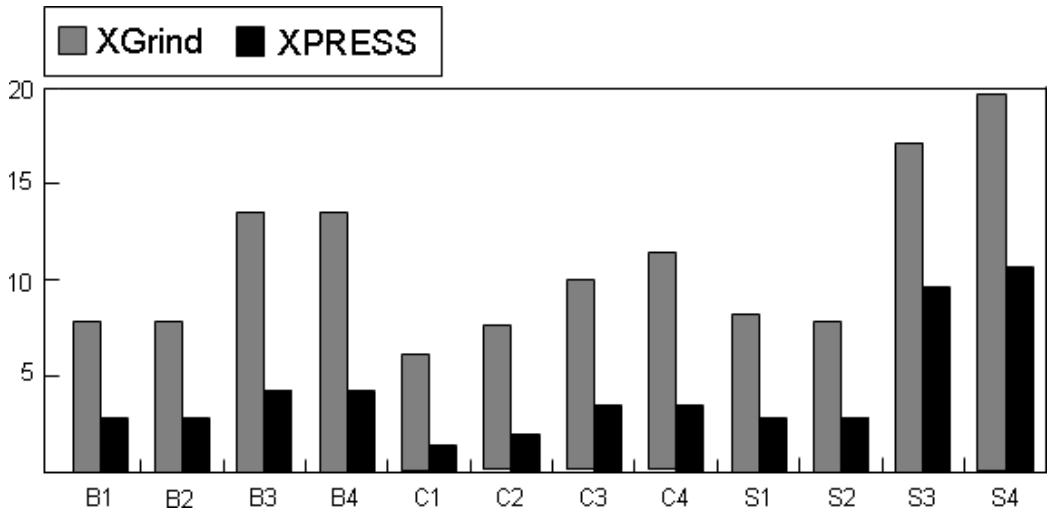
1. Eenvoudige pad-uitdrukking
2. Gedeeltelijke-match pad-uitdrukking
3. Ingewikkelde pad-uitdrukking
4. Bereik-match pad-uitdrukking

In Tabel 3.2 staan de vier verschillende queries voor elk document beschreven.

Naam	Querie
B1	/SEASON/LEAGUE/DIVISION/TEAM/PLAYER/GIVEN_NAME
B2	//TEAM/PLAYER/SURNAME
B3	/SEASON/LEAGUE//TEAM/TEAM_CITY
B4	/SEASON/LEAGUE//TEAM[TEAM_CITY $\geq$ Chicago and TEAM_CITY $\leq$ Toronto]
C1	/root/course/selection/session/place/building
C2	//session/time
C3	/root/course//session/time/start_time
C4	/root/course//session/time[start_time $\geq$ 800 and start_time $\leq$ 1200]
S1	/PLAY/ACT/SCENE/SPEECH/STAGEDIR
S2	//PGROUP/PERSONA
S3	/PLAY/ACT//SPEECH/SPEAKER
S4	/PLAY/ACT//SPEECH[SPEAKER $\geq$ CLEOPARTA and SPEAKER $\leq$ PHILO]

Tabel 3.2: Vier soorten queries

We gaan de kosten vergelijken van XPRESS en XGrind die de verschillende queries met zich meebrengen. In Figuur 3.5 is de tijdsduur van de verschillende queries op de drie documenten weergegeven.



Figuur 3.5: Querie evaluatie tijden: XGrind vs. XPRESS

De kost van de queries van het eerste type (B1, C1 en S1) tonen aan dat de benaderde omgekeerde rekenkundige codering in XPRESS geen achteruitgang van de efficiëntie doet voorkomen.

Daar de lengtes van de uitdrukkingen van het tweede type (B2, C2 en S2) kort en eenvoudig zijn, is de tijd voor de verwerking van deze queries

steeds lager dan bij de queries van type 3 en 4. Het verschil in type 2 tussen XGrind en XPRESS is minder opvallend dan bij type 3 en 4.

In XPRESS is het verschil overtreffend als we kijken naar de meer ingewikkelde queries, namelijk type 3 (B3, C3 en S3). Dit komt doordat de querie-processor in XPRESS de queries efficiënter evalueert door het gebruik van de omgekeerde rekenkundige codering.

Bij de bereik-queries is de kloof eveneens groot, want XPRESS houdt de extra kosten van een gedeeltelijke decompressie zo klein mogelijk door coderingen te gebruiken die de orde van de informatie behouden.

Gemiddeld is de prestatie voor het uitvoeren van queries in XPRESS circa 2.83 keer beter dan bij XGrind.

3.4 XQueC

XQueC is de eerste XQuery processor binnen de gecomprimeerde data die een goede wisselwerking bekomt tussen de compressie-verhouding, de querie-uitdrukkingen en de bevraagbaarheid. Dit is het resultaat van de fragmentatie en het opslagmodel voor de gecomprimeerde XML-bestanden.

Voor het efficiënt evalueren van de queries construeert XQueC een structuurboom van de XML-document en een overzicht ervan, de zogenaamde 'DataGuide', die alle verschillende paden voorstellen. Voor een goede toegang tot de gecomprimeerde waarden in de containers linkt XQueC elke individuele verkleinde waarde aan zijn overeenkomstige knoop in de structuurboom. De container wordt gelinkt aan zijn overeenkomstige pad in de DataGuide. Deze verfijnde structuur zorgt echter voor een toename in de opslagruimte wat eveneens een grotere compressie-ratio teweegbrengt.

3.4.1 XQueC vs. XMill

De techniek van de fragmentatie die XQueC gebruikt, is geleend van XMill en is gebaseerd op het scheiden van de structuur en de inhoud van het XML-document. XMill is een zeer efficiënte XML-compressor, maar is echter niet ontworpen voor het evalueren van queries op de gecomprimeerde data. XMill hecht belang aan het feit dat de waarden aan de bladeren van een XML-boom, die gevonden werden op hetzelfde pad, dikwijls gelijkaardige inhoud vertonen. Om die reden groepeert XMill deze waarden in één enkele container en kiest voor elke conainer een gepaste compressie-methode. Met als gevolg dat de toegang tot elke aparte knoop binnen de container ontzegd wordt. Een volledige container moet vervolgens geheel gedecomprimeerd worden vooraleer er queries op verwerkt kunnen worden.

Waar bij XMill een container waarden kan bevatten van verschillende paden, gaat de tool XQueC voor elke afzonderlijke pad de waarden in een container plaatsen. Vervolgens wordt elke waarde individueel gecomprimeerd wat zorgt voor een individuele toegang tot het blad en voor een doeltreffende verwerking van de queries.

3.4.2 XQueC vs. XGrind/XPRESS

Bij XGrind en XPRESS is het toegelaten om eenvoudige pad-uitdrukkingen te evalueren binnen het gecomprimeerd gebied. De efficiënte uitvoering van de queries is echter beperkt doordat de verwerking van de queries gebaseerd

is op een eenvoudige top-down mechanisme. Hierdoor kunnen heel wat expressies zoals joins en geneste queries niet voltrokken worden zonder dat er veel tijd gestoken wordt in het uitpakken van de tussenliggende resulaten.

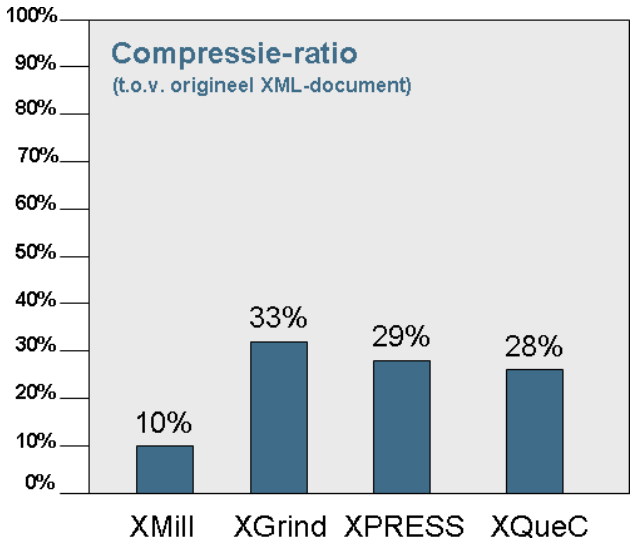
De tool XQueC is een verbetering ten opzichte van de tools XGrind en XPRESS, want XQueC kan willekeurig complexe queries op gecomprimeerde documenten evalueren. De tool XGrind kan beschouwd worden als een compressor die de structuur van het XML-document behoudt en een querie-processor voor basis-uitdrukkingen. Terwijl de tool XPRESS ook een compressor is, maar de querie-processor is al wat meer uitgebreider.
In Tabel 3.3 wordt een kort overzicht gegeven van de tools XGrind, XPRESS en XQueC.

Tool	**XGrind**	**XPRESS**	**XQueC**
Struct./Tekst compr.	Binair/Huff.	Rev. Arith. Enc./Huff.	meerdere
Homomorf	ja	ja	nee
Querie	=, prefix	=, prefix, num. bereik	=, <, prefix
Taal	subset XPath	subset XPath	XQuery
Evaluatie	top-down	top-down	meerdere

Tabel 3.3: XGrind vs. XPRESS vs. XQueC

XGrind en XPRESS comprimeren de XML-bestanden door de 'homomorf' codering waarbij het gecomprimeerd formaat nog steeds een XML-document is. Bij XGrind worden de tags binair omgezet door simpele getallen en de tekst wordt gecodeerd via de methode van Huffman. Bij XPRESS worden de paden voorgesteld door een interval van reële getallen in $[0.0, 0.1]$ met de 'Reverse Arithmetic Encoding'. Verder kan XGrind 'exacte' en 'prefix' match aan in het gecomprimeerd domein. XPRESS kan verder nog de 'range' queries met numerieke waarden evalueren. De evaluatie van de queries gebeurt top-down in de XML-structuur.

De gemiddelde compressie-factors van XMill, XGrind, XPRESS en XQueC worden met elkaar vergeleken en een overzicht van de resultaten [13] staan in Figuur 3.6 beschreven. XMill haalt het beste resultaat met een gemiddelde compressie-verhouding van 10% ten opzichte van het oorspronkelijke XML-bestand. Daarna volgt XQueC met een ratio van gemiddeld 28%. We merken op dat XPRESS en XQueC bijna even goed comprimeren. XGrind heeft de minst goede resultaten en verkleint de XML-bestanden gemiddeld 1/3 van het origineel bestand.



Figuur 3.6: Overzicht van de gemiddelde compressie-ratio's

3.5 DAGs

Om de queries te evalueren wordt er gebruik gemaakt van de taal Core XPath [14] dat een onderdeel van XPath is en de belangrijkste aspecten ervan bevat.

3.5.1 Core XPath Taal

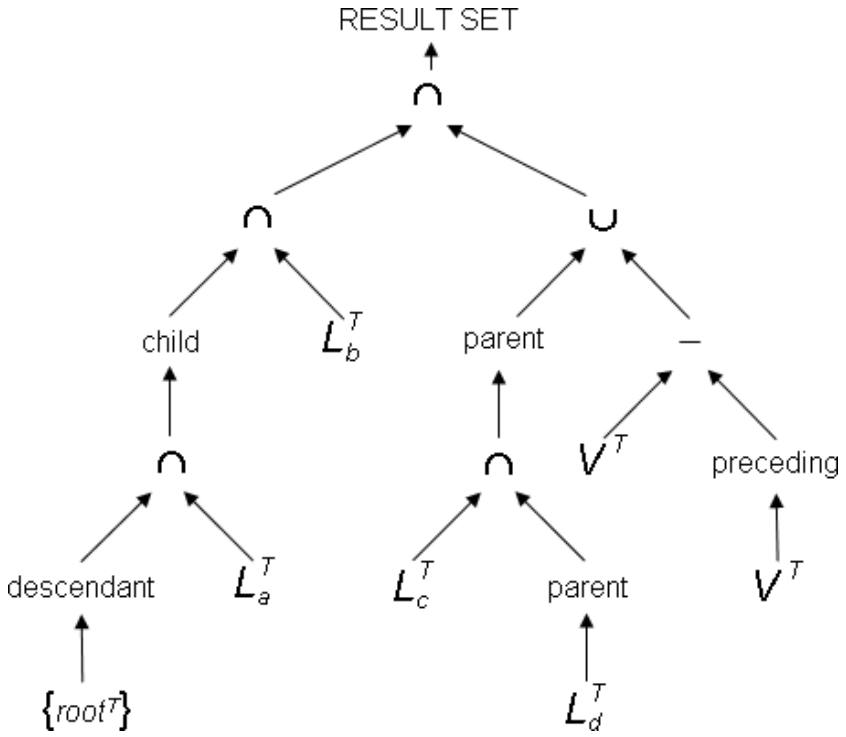
Elke Core XPath querie op een boom-instantie T kan geschreven worden als een algebraïsche uitdrukking door de volgende bewerkingen:

- De set van knopen S^T van de instantie T
- De binaire operaties op setten $\bigcup, \bigcap, -$: $2^{V^T} \times 2^{V^T} \rightarrow 2^{V^T}$
- De axis operaties χ (vb. zelf, ouder, descendant, ...)
- Een operatie $V|root(S) = \left\{V^T | root^T \in S\right\}$

Core Xpath herleidt het probleem van het evalueren van queries met binaire relaties naar het manipuleren van sets. Dit wordt bereikt door de paden in de voorwaarden om te keren zodat de set knopen onmiddellijk in de querie berekend kan worden en dit met betrekking tot de wortel van de querie-boom. Ter verduidelijking een voorbeeld van een Core XPath querie:

$/descendant :: a/child :: b[child :: c/child :: d\ or\ not(following :: *)]$.

Deze querie heeft een boom-instantie T met als schema $\sigma = (root^T, L_a^T, L_b^T, L_c^T, L_d^T)$. $L_x^T \subseteq V^T$ duidt de sets van alle knopen in T genaamd x aan.



Figuur 3.7: Een querie-boom

Deze querie wordt in de algebraïsche vorm geëvalueerd zoals in de querieboom in Figuur 3.7 te zien is.

Operaties op de gecomprimeerde instanties

Benedenwaartse axis operaties zoals afstammeling, descendent-or-self en kind kan op de gecomprimeerde instanties in lineaire tijd uitgevoerd worden.

Het idee is om de DAG van de beginnende instantie vanaf de wortel te doorlopen waarbij elke knoop v éénmaal bezocht wordt. Hierna kiezen we een nieuwe selectie uit v op basis van de selectie van zijn voorouders en splitten we v als er verschillende voorvaders van v verschillende selecties vereisen. We onthouden ook telkens welke knoop we reeds gekopiëerd hebben zodat we herhaling voorkomen.
In bepaalde gevallen (vb. bij nieuwe selecties) is er decompressie nodig om het resulterende skelet te kunnen bekomen. Het is echter de bedoeling om een volledige decompressie te voorkomen wanneer dit niet nodig is.

Binaire set operaties en opwaartse axis operaties (zelf, ouder, voorouder en voorouder-or-self) veroorzaken eveneens geen decompressie op de gecomprimeerde instanties. De opwaartse axis operaties worden op een recursieve manier berekent ten opzichte van de benedenwaarde axis operaties. Axis operaties als $following-sibling$ en $preceding-sibling$ zijn meer complex en kunnen mogelijk ontbindingen nodig hebben. Andere $following$ en $preceding$ axis operaties kunnen bekomen worden uit volgende bewerkingen:

- $following(S) = descendant-or-self(following-sibling(ancestor-or-self(S)))$
- $preceding(S) = descendant-or-self(preceding-sibling(ancestor-or-self(S)))$

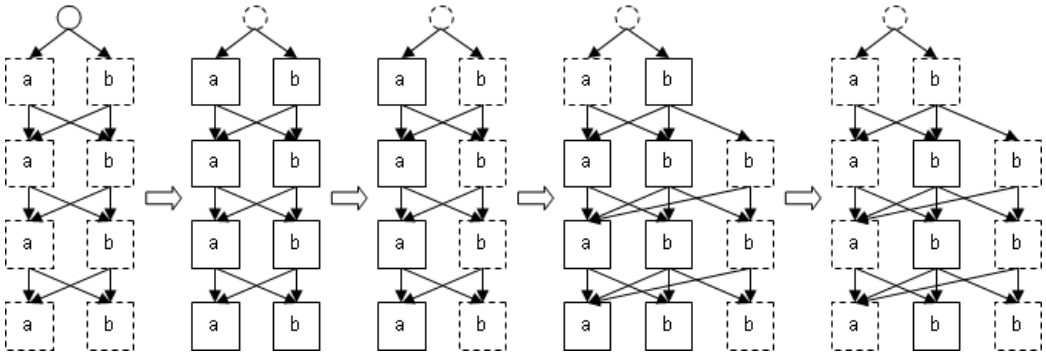
Evaluatie

Elke Core XPath querie kan eenvoudig geëvalueerd worden op een gecomprimeerde instantie $I = (V, \gamma, root, S_1, ..., S_n)$ waarbij:

- Een uitdrukking $S_k \circ S_m$ ($\circ$ een binaire set operator, $1 \leq k, m \leq n$) die evalueert tot een instantie $J = (V, \gamma, root, S_1, ..., S_n, (S_k \circ S_m))$
- Een uitdrukking $\chi(S_k^I)$ die evalueert tot een instantie J, waar de nieuwe selecties zijn toegevoegd (afhankelijk van de axis operatie) en waarbij mogelijk gedeeltelijke decompressie is uitgevoerd ten gevolge van de nieuwe selectie.

- Een uitdrukking $V|root(S_k^I)$ die evalueert tot een instantie J, waarbij de nieuwe selectie $v \in V^I|root \in S_k^I$ is toegevoegd.

Ter illustratie beschouwen we de volgende instantie $I = (V, \gamma, root, L_a, L_b)$ en de querie $/descendant :: a/child :: b$. We herschrijven deze querie in de algebraïsche vorm $child(descendant(root) \bigcap L_a) \bigcap L_b$.



Figuur 3.8: Evaluatie van querie $//a/b$

Eerst evalueren we $descendant(root) = D$ en we voegen deze selectie bij de instantie. We bekomen I^1 in Figuur 3.8 (b). Vervolgens nemen we de doorsnede van D met L_a en bekomen we $A = D \bigcap L_a$. De nieuwe instantie is $I^2 = (V, \gamma, root, L_a, L_b, D, A)$ in Figuur 3.8 (c). Hierna verkrijgen we $C = child(D)$ en de nieuwe instantie I^3 in Figuur 3.8 (d) is inclusief C. Tenslotte berekenen we $B = C \bigcap L_b$ en bekomen we de laatste instantie I^4 in Figuur 3.8 (e) wat ook het resultaat van de querie is.

In theorie kan de methode van het delen van gemeenschappelijke deelbomen leiden tot een exponentiële verkleining van de grootte van de instantie en zelfs tot tweemaal exponentieel comprimeren van de XML-boom. In het slechtste geval kunnen de gecomprimeerde bomen exponentieel gedecomprimeerd worden, zelfs op simpele queries. Verrassend genoeg is de decompressie enkel exponentieel in de grootte van de queries en niet in de gegevens zelf.

Voor een gegeven instantie I en een Core XPath querie Q op I, is de complexiteit van het evalueren van Q $O(2^{|Q|} \times |I|)$ bij het volledig decomprimeren van de instantie. Dit komt doordat het aantal edges en knopen in de instantie bij elke operatie kunnen verdubbelen in het slechtste geval. Als de instantie echter niet gedecomprimeerd wordt, is de complexiteit $O(|Q| \times |I|)$.

De gemiddelde compressie ligt tussen 1/10 en 1/15 van de oorspronkelijk grootte van het skelet van het originele document. Doordat grote delen van

de gegevens in het geheugen opgeslagen kunnen worden, kan de evaluatie van de queries aanzienlijk versneld worden.

We vergelijken de tools XMill, XGrind en XPRESS niet met DAGs, daar DAGs enkel de structuur van het XML-document comprimeert en niet de waarden erin.

3.5.2 DAGs vs. XQueC

We kunnen de compressie-verhoudingen van de tool XQueC niet vergelijken met de techniek van de DAGs daar bij de DAGs geen gebruik wordt gemaakt van een opslagplaats buiten het XML-document zelf. Verder worden bij de DAGs de eigenlijke waarden niet gecomprimeerd en de inhoud is toch wel een belangrijk aspect binnen het document.

Een mogelijke vergelijking is echter het aantal knopen Nin de samenvatting van de structuurboom, de zogenaamde 'DataGuide' die door XQueC aangemaakt wordt en die een overzicht geeft van alle top-down paden binnen het XML-bestand. Anderzijds hebben we het aantal knopen van de DAG $|DAG|$. In Tabel 3.4 zien we de resulaten van enkele XML-bestanden.

In het algemeen zijn het aantal paden N bij XQueC opvallend kleiner dan het aantal knopen in de minimale DAG. Dit is te verklaren doordat XQueC de equivalante knopen samen bundelt dat gebaseerd is op de paden. Op te merken is dat bij meer variabele en herhalende structuren in de gegevens dat leidt tot een redelijk grote DAG, maar toch een compacte structuurboom in XQueC heeft. Een toename van drie paden van N kan een verdubbeling van $|DAG|$ veroorzaken. Bij het XML-document 'Treebank' is de structuur van het XML-bestand bijna niet comprimeerbaar waarbij de DAG kleiner is dan het aantal paden bij XQueC.

XML-document	**Grootte**	N	$\|DAG\|$
Treebank	82 MB	338.738	319.654
XMark	111 MB	514	38.655
SwissProt	109 MB	117	38.936
Shakespeare	7 MB	58	1.121
DBLP	128 MB	125	326
NASA	24 MB	24	8.391

Tabel 3.4: N in XQueC vs. $|DAG|$ in DAGs

3.6 BPLEX

We hebben reeds gezien in Hoofdstuk 2 dat het algoritme BPLEX wordt gebruikt om een kleine pointer-voorstelling van een boom-structuur van een XML-document te bekomen. Deze boom-structuur kan als een 'binaire' boom (met rang) of 'zonder-rang' voorgesteld worden. Terwijl BPLEX (bijna) even goed presteert op beide voorstellingen is dit echter niet het geval voor de minimale DAG.

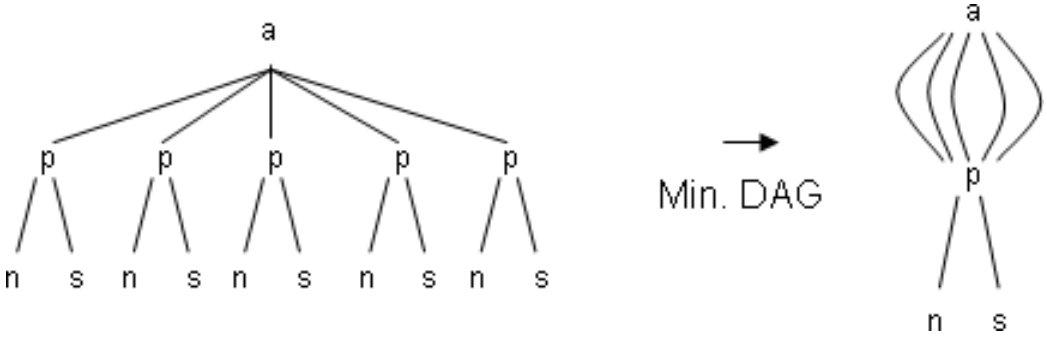
3.6.1 BPLEX vs. DAGs

Beschouw het volgende skelet van een XML-document:

```
<agenda>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
</agenda>
```

Bomen/grammatica's zonder rang

Een (geordende) zonder-rang boom-voorstelling van dit skelet is in Figuur 3.9 te zien. Deze boom heeft als wortel de knoop a (agenda) met eronder zijn vijf pointers dat elk verwijst naar de knoop p (person), die op zijn beurt elk twee pointers heeft naar een knoop n (name) en een knoop s (street). De grootte van deze zonder-rang boom is 15 edges (het aantal edges geeft de grootte weer).



Figuur 3.9: (Geordende) zonder-rang boom en zijn minimale DAG

Omgevormd tot de minimale DAG heeft deze boom zonder rang (zie Figuur 3.9) een grootte van 7.

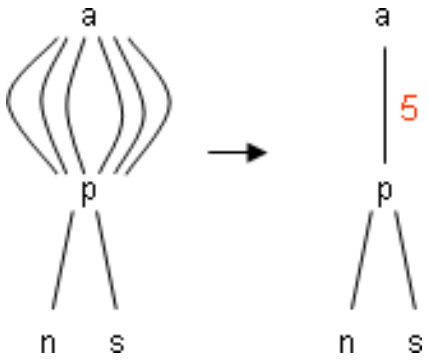
Een boom zonder rang heeft een unieke minimale DAG dat ook geschreven kan worden als een minimale reguliere boom-grammatica. Deze kan als input voor BPLEX dienen en heeft een grootte van 6 edges (zie verder bij 'Binaire bomen/grammatica's (met rang)' voor deze grammatica G').

We kunnen de grootte van de minimale DAG nog verder verkleinen door het gebruik van tellers die de samenvallende edges laten samenvloeien tot één edge met een teller erbij zoals in Figuur 3.10 te zien is. Hier zien we dat de DAG voor de boom zonder rang kan voorgesteld worden met enkel 3 edges en equivalent is met de volgende reguliere boom-grammatica met tellers:

$A \rightarrow agenda([5]P)$

$P \rightarrow person(name, street)$

De DAGs met tellers kunnen omgevormd worden tot een reguliere boom-grammatica van dezelfde grootte die een binaire voorstelling genereren van de oorspronkelijke boom zonder rang. In de grammatica kunnen ook tellers bij knopen voorkomen, maar deze worden uitgeplooid tot kettingen van knopen. BPLEX verandert de tellers niet.



Figuur 3.10: Minimale DAG en zijn minimale DAG met teller

Binaire bomen/grammatica's (met rang)

Het BPLEX-algoritme werkt op bomen (met rang). Zo een boom kan omgevormd worden tot een binaire boom (met rang) zonder het aantal edges te veranderen (zie Figuur 3.11 dat de binaire versie is van de geordende boom zonder rang uit Figuur 3.9). Vervolgens wordt de minimale DAG gezocht en wordt deze weergegeven als een reguliere boom-grammatica waar BPLEX

kan op losgelaten worden. De bijhorende reguliere boom-grammatica G heeft de volgende producties:

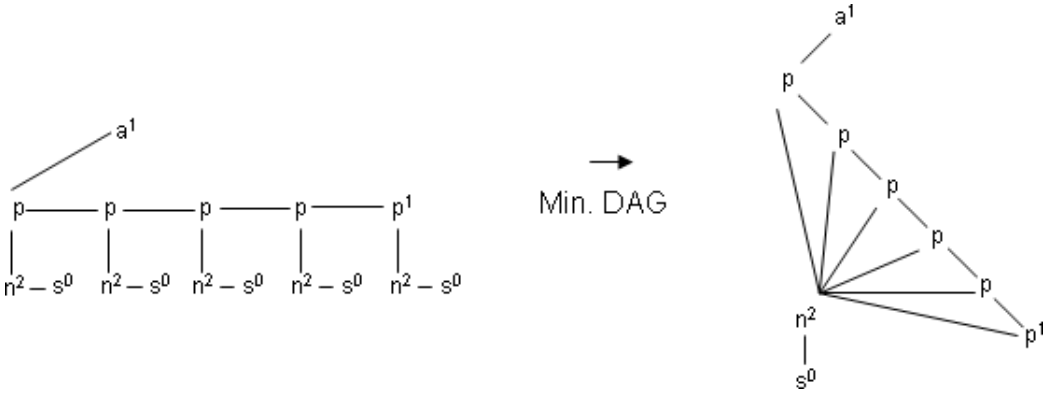
$S \rightarrow agenda^1(person(A, person(A, person(A, person(A, person^1(A))))))$
$A \rightarrow name^2(B)$
$B \rightarrow street^0$

De grootte van deze grammatica G is 11. Een blad (resp. de laatste broer) in de boom zonder rang heeft geen linker (resp. geen rechter) kind in de binaire boom-voorstelling. Dit wordt weergegeven door de superschrift 2 (resp. 1) en door een 0 voor een laatste blad.

Het algoritme BPLEX verandert G in de volgende output-grammatica G' en deze heeft grootte 6:

$S \rightarrow agenda^1(C(D(D(\epsilon))))\ D(y_1) \rightarrow C(C(y_1))$
$A \rightarrow name^2(B)\ C(y_1) \rightarrow person(A, y_1)$
$B \rightarrow street^0$

Hierbij zijn de voorkomens van het patroon $p = person(A, y_1)$ vervangen en voor het patroon $p' = person^1(A)$ wordt de ontbrekende parameter y_1 als 'leeg' (ϵ) gemarkeerd. Zoals in de D-productie beschreven is, genereert deze kopieën op een pad in de binaire boom dat een exponentiële afname in de grootte van de input-grammatica kan veroorzaken.



Figuur 3.11: Binaire boom met rang en zijn minimale DAG

Als we de binaire boom uit Figuur 3.11 nu omvormen tot zijn minimale DAG, heeft deze grootte 11.

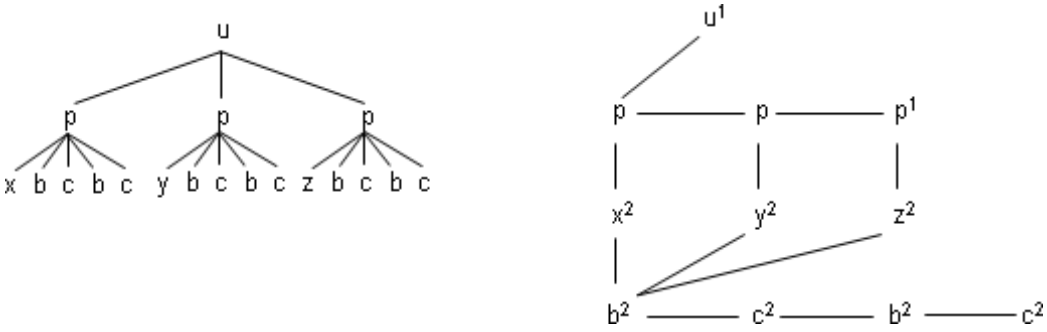
Beoordeling

We hebben gezien dat de grootte van minimale DAG van een boom zonder rang verschillend kan zijn ten opzichte van de minimale DAG van zijn binaire

voorstelling (met rang). In de meeste gevallen is echter de minimale DAG van de boom zonder rang kleiner dan die van de binaire boom. De reden hiervoor is dat de ketting van de rechter kind-edges in de binaire boom, broers van deelbomen worden in de boom zonder rang.

Dit is duidelijk te zien in de binaire boom in Figuur 3.11 waar zijn minimale DAG enkel één kopie heeft van de deelboom $name^2(street)$ en grootte 11 heeft. Bij de minimale DAG van zijn boom zonder rang in Figuur 3.9 is er enkel één kopie van de deelboom $person(name, street)$ en heeft slechts grootte 7.

Als voorbeeld van een binaire boom waarbij zijn minimale DAG kleiner is dan de bijhorende boom zonder rang, beschouwen we de boom T zonder rang uit Figuur 3.12. Zijn minimale DAG zonder rang heeft als grootte 18 edges, maar de minimale binaire DAG heeft maar grootte 12. Dit komt doordat de binaire versie slechts één kopie van de deelboom $b^2(c^2(b^2(c^0)))$ beschouwt.



Figuur 3.12: Boom T zonder rang (= min. DAG) en zijn min. binaire DAG

Het is dus duidelijk dat de compressie door het delen van gemeenschappelijke subbomen, of de minimale DAG, gevoelig is voor de rang. Dit kunnen we ook afleiden uit de tabel uit Figuur 3.13 waar er een duidelijk verschil is tussen de grootte van de minimale binaire DAG (met rang) en de minimale DAG zonder rang met teller (mDAG).

Experimentele resultaten

Het algoritme BPLEX (zie [15] voor de initiële implementatie) is echter niet gevoelig voor de rang en dit resultaat is ook in de tabel (Figuur 3.13) terug te vinden. Er werden grote invoer-parameters gebruikt: $K_N = 30000$ (grootte van de window), $K_S = 20$ (maximale grootte van het patroon) en $K_R = 10$ (maximale rang). De grootte van de output-grammatica ligt tussen de 0.1% en 21% van de originele boomstructuur of is gemiddeld de helft tot 75% kleiner dan bij de DAGs.

Verder is de prestatie van BPLEX te wijten aan de drie invoer-parameters. Vooral met kleine waarden voor K_S en K_R zijn de compressie-resultaten goed. Waarden waar $K_S > 10$ en $K_R > 5$ zorgen niet meer voor een toename van de compressie. Vooral de grootte van het window K_N heeft invloed op de compressie en is op zijn best als $K_N > 100$ en boven de 20000 heeft geen invloed meer.

Input bestand	Grootte van het skelet	Minimale Binaire DAG grootte		Minimale Dag met teller (zonder rang) grootte		BPLEX output grootte	
SwissProt (457,4 MB)	10.903.568	1.437.445	**12,2%**	1.100.648	**10,1%**	311.328	**2,9%**
DBLP (103,6 MB)	2.611.931	533.183	**20,4%**	222.754	**8,5%**	115.902	**4,4%**
Treebank (55,8 MB)	2.44.727	1.454.494	**59,4%**	1.301.688	**53,2%**	519.542	**21,2%**
1998statistics (657 KB)	28.306	2.403	**8,5%**	726	**2,6%**	410	**1,4%**
Catalog-02 (104 MB)	2.240.231	52.392	**2,3%**	32.267	**1,4%**	26.774	**1,2%**
Catalog-01 (11MB)	225.194	6.990	**3,1%**	8.503	**2,8%**	3.817	**1,7%**
Dictionary (104 MB)	2.731.764	681.130	**24,9%**	441.322	**16,2%**	160.329	**5,9%**
Dictionary (11M)	277.072	77.554	**28,0%**	46.993	**17,0%**	20.150	**7,3%**
JST_snp.chr1 (36M)	655.946	40.663	**6,2%**	25.047	**2,3%**	12.859	**1,8%**
JST_gene.chr1 (11M)	216.401	14.606	**6,7%**	5.658	**2,6%**	4.000	**1,8%**
NCBI_snp.chr1 (190M)	3.642.225	809.394	**22,2%**	15	**<0,1%**	59	**<0,1%**
NCBI_gene.chr1 (24M)	360.350	14.356	**4,0%**	11.767	**3,3%**	7.160	**2,0%**

Figuur 3.13: Resultaten van DAGs en BPLEX

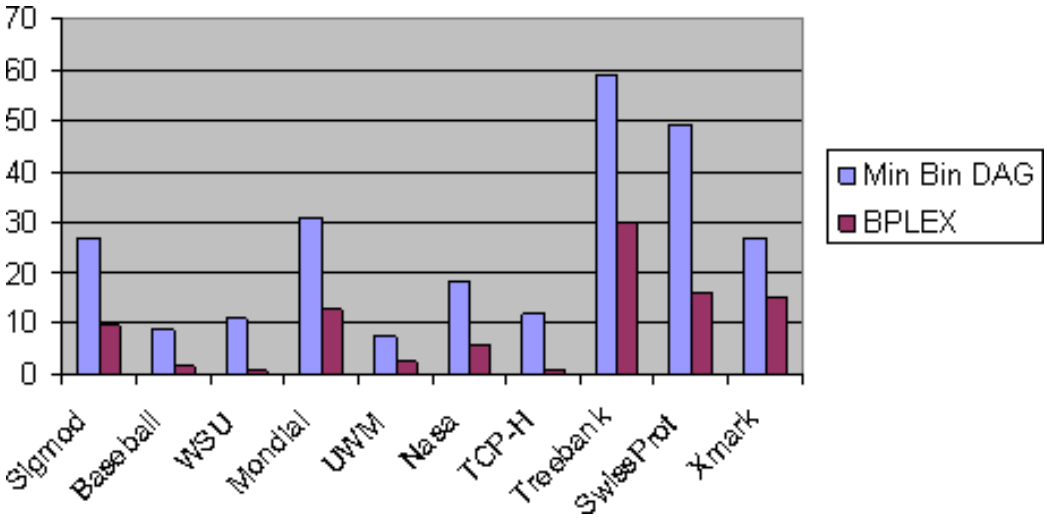
Gaan we nu de instellingen aanpassen voor BPLEX en we gebruiken voor de grootte van de window 100, de maximale grootte van het partoon 10 en de maximale rang ook 10. In Tabel 3.5 staan deze resultaten.

XML-Bestand	**Grootte**	**# Edges**	**Min. Bin. DAG**	**BPLEX grootte**
Baseball	639,8 KB	27 080	2 401 (8,9%)	507 (1,9%)
TCP-H	30,8 MB	1 022 976	120 365 (12%)	7 543 (0,74%)
Mondial	1,7 MB	22 423	6 896 (31%)	2 861 (13%)
Nasa	23,9 MB	476 646	86 202 (18%)	28 217 (5,9%)
Sigmod	467,2 KB	11 526	3 114 (27%)	1 106 (9,6%)
XMark	111,1 MB	1 666 315	447 459 (27%)	246 935 (15%)
SwissProt	109,5 MB	2 977 031	1 453 608 (49%)	466 570 (16%)
TreeBank	82,1 MB	2 437 666	1 447 621 (59%)	736 517 (30%)
UWM	2,2 MB	66729	4 983 (7,5%)	1 853 (2,8%)
WSU	1,6 MB	74557	7 865 (11%)	512 (0,69%)

Tabel 3.5: Extra resultaten van DAGs en BPLEX

Je ziet dus duidelijk dat zelfs bij andere waarden voor de meegegeven parameters BPLEX veel beter presteert dan de DAGs. In Figuur 3.14 staan deze

compressie-ratio's weergegeven. In Figuur 3.15 en in Figuur 3.16 staan de compressie-snelheden van BPLEX afgebeeld in seconden. Hier zie je duidelijk dat de tijd van het aantal edges afhangt.



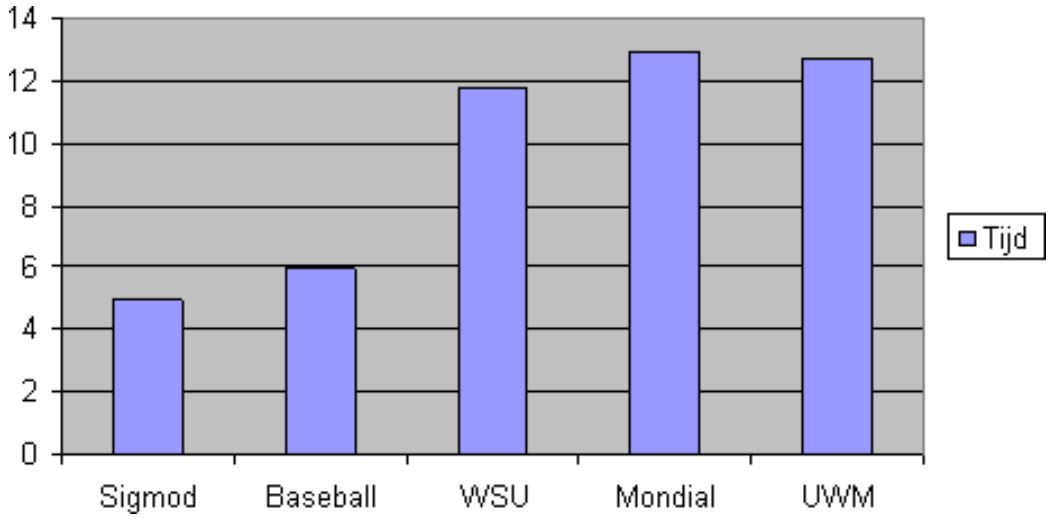
Figuur 3.14: Compressie-ratio's van DAGs en BPLEX

3.6.2 Evaluatie

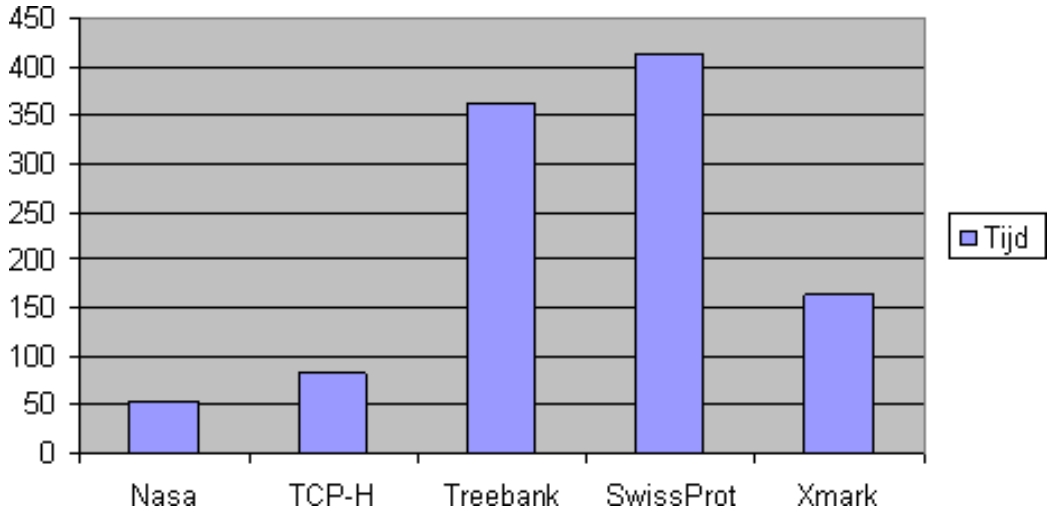
De SLT grammatica's zijn zeer efficiënt om XML-documenten voor te stellen. Deze grammatica's kunnen een grote XML-bestand voortstellen die volledig in het geheugen kunnen. Hoe worden deze XML-bomen nu verwerkt zonder de bijhorende grammatica te decomprimeren?

Telkens een knoop in de XML-boom gelezen wordt of een edge wordt doorlopen, kan dit verwezenlijkt worden binnen de grammatica door een 'overhead' of extra kost van maximaal de grootte van die grammatica G ($|G|$). In het algemeen kan er gezegd worden dat hoe meer compressie hoe langzamer de leesoperaties op de gecomprimeerde voorstelling. Echter zijn er uitzonderingen zodat er geen stijging van de extra kost is, zelfs een daling ervan is mogelijk. We overlopen enkele operaties [9]:

- *Het valideren van een XML-type* waarbij een XML-document voorgesteld wordt door een SL cv boom grammatica en deze moet de validiteit beoordelen van een XML-type. Voor het beschrijven van een XML-type wordt onder andere gebruikt DTDs en XML Schema's. Deze validatie gebeurt in polynomiale tijd en staat in verband met de grootte van G.



Figuur 3.15: Compressie-snelheden van BPLEX (kleine bestanden)



Figuur 3.16: Compressie-snelheden van BPLEX (grote bestanden)

- *Het testen van equivalentie* waarbij er onderzocht wordt of twee SL cv boom grammatica's dezelfde boom genereren zonder volledige decompressie. Dit kan gedaan worden in PSPACE of als de beide grammatica's lineair zijn zelfs in polynomiale tijd.

De evaluatie van queries in Core XPath in DAGs en BPLEX zijn equivalent met elkaar. De lineaire SL cv boom grammatica's in BPLEX is PSPACE-compleet [16] zoals bij de DAGs [7]. Dit betekent dat, ook al heeft de output van BPLEX betere compressie-verhoudingen dan de DAGs, de tijd voor het evalueren van de queries blijft hetzelfde voor BPLEX als voor DAGs.

Conclusies

XML voorziet flexibiliteit in het publiceren en uitwisselen van heterogene gegevens op het web. Desondanks is de taal in het algemeen verbose en daardoor zijn de XML-documenten meestal groter dan andere voorstellingen die dezelfde inhoud bevatten.

De grootte van de XML-bestanden kan echter een probleem worden voor toepassingen die met XML werken. Er is een aanzienlijke toename in de opslagruimte, het verwerken en het uitwisselen van de gegevens.

Het comprimeren van XML-documenten is de laatste jaren meer zijn opgang aan het maken doordat de redundante gegevensvoorstellingen in XML verkleind moeten worden voor een betere prestatie.

De algemene tekstcompressors (o.a. Zip en Gzip) en de codeermethodes (o.a. Huffman en LZW) zijn niet bekwaam genoeg om bruikbare gecomprimeerde XML-data te produceren. Daar XML-bestanden een bepaalde structuur met tags hebben, gaan de speciale XML-compressors dit benutten.

De compressor XMill is de meest bekende techniek en haalt de hoogste resultaten voor de compressie-verhouding. Dit wordt gerealiseerd door de structuur en de gegevens afzonderlijk te comprimeren en op te slaan.

Een groot nadeel voor het evalueren van queries op de verkleinde XML-bestanden met XMill is het volledig decomprimeren ervan. We hebben enkele technieken gezien die de XML-bestanden kunnen querieën zonder een volledige decompressie. Hierbij kunnen we een onderscheid maken tussen benaderingen die wel of niet de structuur van de XML-document verkleinen.

Beknopte voorstelling

De compressor XGrind is een parser die op SAX gebaseerd is. Voor de codering van tekst en getallen wordt er respectievelijk de Huffman en de Rekenkundige codering gebruikt. De tags of meta-data worden apart in een schema gecomprimeerd. Ook simpele queries zoals 'prefix' en 'exact-match' worden op de gecomprimeerde gegevens ondersteund.

XPRESS is de opvolger van XGrind, maar voor het coderen wordt de omgekeerde rekenkundige techniek toegepast voor de paden die in het interval $[0.0, 1.0)$ liggen. De getallen kunnen volgens vier verschillende manieren gecodeerd worden. Een bijkomende querie ten opzichte van XGrind kan uitgevoerd worden, namelijk de numerieke 'bereik'-querie.

De techniek XQueC verkleint de XML-bestanden met een goed resultaat en kan bovendien de volledige XQuery op een efficiënte wijze ondervragen. Dit komt doordat XQueC zijn voordeel haalt uit het feit dat er verschillende compressie-algoritmes gebruikt worden en de gegevens met dezelfde eigenschappen worden gegroepeerd. Voor de tekst-waarden wordt het 'ALM'-algoritme gebruikt die de volgorde behoudt.

In vergelijking met XGrind en XPRESS bereikt XQueC de beste evenwicht tussen de compressie-ratio, de goede bevraagbaarheid en de vele XQuery-uitdrukkingen. Voor compressie-tijd bereikt XMill het beste resultaat daar XMill slecht één keer het document moet doorlopen .

Pointer-voorstelling

Deze benaderingen van compressie maakt het mogelijk om grotere XML-documenten in het geheugen te houden waar ze dan efficiënt geëvalueerd worden. Het geeft ook de gelegenheid voor een aanzienlijke versnelling doordat er geen nood is aan een bijkomstige opslagplaats daar anders de bomen te groot zijn en niet volledig in het geheugen kunnen gehouden worden.

We merken op dat deze voorstellingen enkel de structuur verkleinen en niet de waarden van het XML-document.

Verder wordt tijd bespaart bij het evalueren van queries op de gecomprimeerde bomen, want overbodige verwerkingen worden vermeden. De compressie-verhoudingen hebben een goede resultaat. De evaluatie van queries is zeer efficiënt.

We hebben twee technieken gezien die gebruik maken van pointers, namelijk het delen van subbomen en het delen van subgraven. De techniek van het delen van subgraven is de opvolger van de DAGs, want elke DAG kan gezien worden als een *reguliere* boom grammatica die juist één boom genereert. Een SL cv boom grammatica is een *context-vrije* boom grammatica die juist één boom genereert. Waarbij de DAGs het toelaten om identieke deelbomen te delen, staat de SL cv boom grammatica het toe om identieke patronen of samenhangende deelgraven in de boom te delen.

Dit resulteert in betere compressie-verhoudingen bij de SL cv boom grammatica's dan bij de DAGs. Waar SL cv boom grammatica's leiden tot dubbele

exponentiële verhoudingen, staan de DAGs slechts een enkele exponentiële verhouding toe in de compressie.

We kunnen concluderen dat het algoritme BPLEX het meest doeltreffend is voor het comprimeren van de XML-structuur van een XML-document.

Bibliografie

[1] *XMILL: An Efficient Compressor for XML Data*, Hartmut Liefke, Dan Suciu, SIGMOD Conference 2000.

[2] *Efficient Query Evaluation over Compressed XML Data*, Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, Andrea Pugliese, EDBT 2004.

[3] *XGRIND : A Query-friendly XML Compressor*, Pankaj M. Tolani, Jayant R. Haritsa, ICDE 2002.

[4] *XPRESS: A Queriable Compression for XML Data*, Jun-Ki Min, Myung-Jae Park, Chin-Wan Chung, SIGMOD Conference 2003.

[5] *XQueC: Pushing Queries to Compressed XML Data*, Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, Andrea Pugliese, VLDB 2003.

[6] *Path Queries on Compressed XML*, Peter Buneman, Martin Grohe, Christoph Koch, VLDB 2003.

[7] *Query Evaluation on Compressed Trees (Extended Abstract)*, Markus Frick, Martin Grohe, Christoph Koch, LICS 2003.

[8] *XQzip: Querying Compressed XML Using Structural Indexing*, James Cheng, Wilfred Ng, EDBT 2004.

[9] *Efficient Memory Representation of XML Documents*, Giorgio Busatto, Markus Lohrey, Sebastian Maneth, DBPL'05.

[10] *Graph-Based Algorithms for Boolean Function Manipulation*, R. E. Bryant, 1986.

[11] *Tree Transducers and Tree Compressions*, S. Maneth, G. Busatto, 2004.

[12] *XML-QL: A Query Language for XML*, http://www.w3.org/TR/NOTE-xml-ql/.

[13] *XQueC: Embedding Compression Into XML Databases*, A. Arion, A. Bonifati, I. Manolescu, A. Pugliese, 2005.

[14] *Core XPath*, http://www.w3.org/TR/xpath.

[15] *BLPEX*, http://bplex.sourceforge.net/.

[16] *Tree Automata and XPath on Compressed Trees*, M. Lohrey, S. Maneth, 2005.

[17] *SAX-Parser*, http://www.saxproject.org/.

[18] *XQuery*, http://www.w3.org/TR/xquery/.

[19] *Introduction to Data Compression*, Guy E. Blelloch, 2001.

[20] *The Scientist and Engineer's Guide to Digital Signal Processing: Chapter 27 - Data Compression*, Steven W. Smith, 1997.

[21] *A block-sorting lossless data compression algorithm*, M. Burrows, D. J. Wheeler, 1994.

[22] *Dataguides: Enabling query formulation and optimization in semistructured databases*, R. Goldman, J. Widom, 1997.

[23] *Comparative Analysis of XML Compression Technologies*, Wilfred Ng, Yeung Wai Lam and James Cheng, 2005.

[24] *Wikipædia Britannica*, nl.wikipedia.org.

[25] *Data Compression*, http://www.stanford.edu/ udara/SOCO/index.htm.

[26] *Technologie van Multimediasystemen en -Software: Opslag en compressie van multimedia*, http://didactiek.edm.luc.ac.be/tms/.

[27] *XML Data Repository*, http://www.cs.washington.edu/research/xmldatasets/.

[28] *XMark An XML Benchmark Project*, http://monetdb.cwi.nl/xml/.

[29] *Apache HTTP Server - Log Files*, http://httpd.apache.org/docs/1.3/logs.

[30] *Shakespeare*, http://www.ibiblio.org/xml/examples/shakespeare/.

[31] *Baseball*, http://www.cafeconleche.org/examples/baseball/.

[32] *Entropy Compression Methods*, Arkadi Kagan, http://sourceforge.net/projects/compressions/.

[33] *Gzip*, http://www.gzip.org.

[34] *Bzip2*, http://www.bzip2.org.

[35] *Winzip*, http://www.winzip.com.

[36] *Winrar*, http://www.winrar.com.

[37] *XMILL*, http://www.cs.washington.edu/homes/suciu/XMILL.

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen en uw akkoord te verlenen.

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Compressie van XML-documenten
Richting: **Licentiaat in de informatica** Jaar: **2006**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Deze toekenning van het auteursrecht aan de Universiteit Hasselt houdt in dat ik/wij als auteur de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij kan reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

U bevestigt dat de eindverhandeling uw origineel werk is, en dat u het recht heeft om de rechten te verlenen die in deze overeenkomst worden beschreven. U verklaart tevens dat de eindverhandeling, naar uw weten, het auteursrecht van anderen niet overtreedt.

U verklaart tevens dat u voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen hebt verkregen zodat u deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal u als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze licentie

Ik ga akkoord,

Kim Jenny RAMAEKERS

Datum: