2012•2013
FACULTEIT WETENSCHAPPEN
master in de informatica: databases

Masterproef

Programmeertalen voor computerclusters

Promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Copromotor :
dr. Geert BEX

Tom Smets

Masterproef voorgedragen tot het bekomen van de graad van master in de informatica , afstudeerrichting databases

2012•2013
FACULTEIT WETENSCHAPPEN
master in de informatica: databases

Masterproef

Programmeertalen voor computerclusters

Promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Copromotor :
dr. Geert BEX

Tom Smets

Masterproef voorgedragen tot het bekomen van de graad van master in de informatica , afstudeerrichting databases

universiteit hasselt | Maastricht University

universiteit hasselt

Programmeertalen voor computerclusters

Auteur:
Tom Smets

Promotor:
Prof. dr. Jan Van den Bussche
Begeleider:
Geert Jan Bex

Masterproef voorgedragen tot het behalen van de graad van master in de informatica/ICT/kennistechnologie

2012-2013

Abstract

De laatste jaren zijn termen zoals 'Big Data' en 'Cloud Computing' veelvuldig gebruikt binnen de IT-wereld. Dit is niet verbazend aangezien de hoeveelheid data die bedrijven ter beschikking hebben steeds groter wordt, en ondertussen kan oplopen tot enkele petabytes of zelfs exabytes. Denk hierbij maar aan bedrijven zoals Facebook of Google. Het verwerken van zulke grote hoeveelheden data brengt uiteraard nieuwe uitdagingen met zich mee. Het spreekt voor zich dat data van deze grootte-orde niet meer door één enkele server verwerkt kan worden, en dat we beroep zullen moeten doen op een cluster van computers of servers. Het schrijven van gedistribueerde applicaties voor deze computer clusters is echter geen sinecure. Vooreerst kunnen typisch niet alle aspecten van een programma in parallel worden uitgevoerd, en zullen computers (ook wel 'compute nodes') dus onderling informatie moeten uitwisselen. Aangezien de snelheid waarmee data over het netwerk kan worden uitgewisseld aanzienlijk kleiner is dan bijvoorbeeld de snelheid waarmee van een harde schijf gelezen kan worden, is het belangrijk om de hoeveelheid communicatie zo klein mogelijk te houden. Verder bevinden compute nodes uit een cluster zich typisch in een andere 'state'. Hiermee bedoelen we dat ze zijn opgebouwd uit andere hardware, ze over andere inputdata beschikken of hiernaast nog andere processen aan het uitvoeren zijn. Het is zelfs mogelijk dat gedurende de uitvoer van een programma bepaalde compute nodes (al dan niet permanent) onbeschikbaar worden. Het is ondertussen wel duidelijk dat het schrijven van gedistribueerde applicaties heel wat nieuwe problemen met zich meebrengt, en er dus ook vraag is naar nieuwe technologiën of oplosingen. In deze thesis gaan we op zoek naar deze oplossingen. Hierbij stellen we ons de vraag: "Welke programmeerparadigma's bestaan er voor het schrijven van gedistribueerde applicaties, wat zijn hun voor- en nadelen, en voor welke situatie zijn ze het meest geschikt?" Het tevens van groot belang voor deze domeinstudie dat we de achterliggende werking van de beschikbare programmeermodellen goed doorgronden.

Voorwoord

In de eerste plaats ben ik vooral dank verschuldigd aan prof. dr. Van den Bussche om dit zeer boeiende en actuele onderwerp aan mij toe te kennen. Verder heeft hij mij ook steeds goed begeleid, waarbij ik toch voldoende ruimte kreeg om mijn eigen ideëen uit te werken. De literatuur die hij me aanraadde was bovendien van uitstekende kwaliteit, wat mijn onderzoek er enkel makkelijker op maakte. Tot slot is hij erg goed in het begrijpen (en uitleggen) van het grotere geheel, waardoor ik zelf beter het overzicht kon bewaren van de besproken literatuur en het einddoel makkelijker voor ogen kon houden.

Hiernaast ben ik ook dank verschuldigd aan mijn co-promotor dr. Geert Jan Bex. Zijn vergaande kennis, zowel praktisch al theoretisch, hebben me vaak geholpen wanneer bepaalde aspecten mij onduidelijk waren. Bovendien had hij vaak een andere aanpak in gedachten dan ik zelf had, zeker wat betreft de implementatie(s) van de case study. Dit heeft meermaals geleid tot interessante discussies en alternatieve scenario's, wat op hun beurt tot nieuwe conclusies leidde.

Inhoudsopgave

Inleiding

Het doel van deze thesis is een vergelijkende studie te maken over de beschikbare programmeermodellen voor computerclusters. Naast de voor- en nadelen en de praktische toepasbaarheid van deze modellen trachten we ook hun achterliggende werking steeds goed te doorgronden. Uiteraard gaat dit gepaard met een grondige literatuurstudie. Om tot betere inzichten te komen, en bovendien deze programmeerparadigma's in de praktijk te testen, zullen we ook eenzelfde case study implementeren in al deze modellen.

Concreet zullen de volgende drie programmeerparadigma's aan bod komen. Vooreerst bespreken we het Map/Reduce framework Hadoop, dat op korte tijd is uitgegroeid tot een ware hype. Zelfs chipfabrikant Intel heeft sindskort zijn eigen versie van het Hadoop framework gelanceerd.[7] Dit is het enige programmeermodel dat aan bod komt waarbij de gebruiker zelf weinig tot geen controle heeft over de (manier waarop) communicatie die plaatsvindt tussen de verschillende compute nodes.

Vervolgens bespreken we een message passing paradigma genaamd de Java Message Service. Zoals de naam al doet vermoeden biedt JMS een standaard aan voor de manier waarop verschillende applicaties of computers met elkaar communiceren. Een ander bekend voorbeeld van een message passing programmeermodel is de Message Passing Interface, dewelke in de praktijke de de facto standaard is voor het schrijven van gedistribueerde applicaties.

Tot slot beschouwen we nog de relatief nieuwe programmeertaal Bloom, ontwikkeld aan de universiteit van Californië. Hoewel deze taal nog vrij onbekend is, is ze toch gebaseerd op enkele zeer interessante ideëen, dewelke in Hoofdstuk 3 aan bod zullen komen.

Hoofdstuk 1

Hadoop

1.1 Inleiding

Hadoop is een framework ontworpen voor het ontwikkelen van applicaties voor computerclusters. Het is voornamelijk bedoeld voor grote clusters, bestaande uit enkele honderden tot duizenden nodes, die bovendien opgebouwd zijn uit commodity hardware. Het voordeel van dit soort computerclusters is dat ze makkelijk uitbreidbaar zijn, en bovendien goedkoper aan te kopen dan bijvoorbeeld paralelle database management systemen[18][20]. Apache Hadoop is opgebouwd uit twee delen: een implementatie van het Map/Reduce programmeermodel en een gedistribueerd file system genaamd HDFS.

Map/Reduce is een programmeermodel voor het verwerken en genereren van grote datasets. Zoals de naam doet vermoeden gebeurt de verwerking door een Map en een Reduce functie die geschreven wordt door de gebruiker. Vervolgens wordt de uitvoering van het programma automatisch geparalelliseert door het onderliggende framework. Bovendien is Hadoop ook resistent voor het falen van nodes: het werk wordt automatisch gepropageerd naar andere nodes. Het is duidelijk dat dit alles de programmeur veel werk bespaart. In ruil hiervoor wordt hij gedwongen zich te beperken tot het Map/Reduce programmeermodel.

Het gedistribueerde file system van Hadoop is ontworpen om grote hoeveelheden data op te slaan in een grote cluster. Belangrijk hierbij is dat er ook één of meerdere kopies van de data wordt opgeslagen. Een voordeel hiervan is dat de data niet verloren gaat of onbeschikbaar wordt indien een node al dan niet tijdelijk onberijkbaar wordt. Een ander belangrijk voordeel is dat dit de localiteit van data verhoogt. Hiermee bedoelen we dat berekeningen bij voorkeur uitgevoerd worden op de node die de data bevat, of indien dit niet mogelijk is er een node gekozen wordt die zo 'dicht' mogelijk bij de data ligt. Zo liggen twee nodes die op dezelfde switch zijn aangesloten dichter bij elkaar dan twee nodes waarbij er meerdere switches gepasseerd moeten worden.

1.2 Map/Reduce

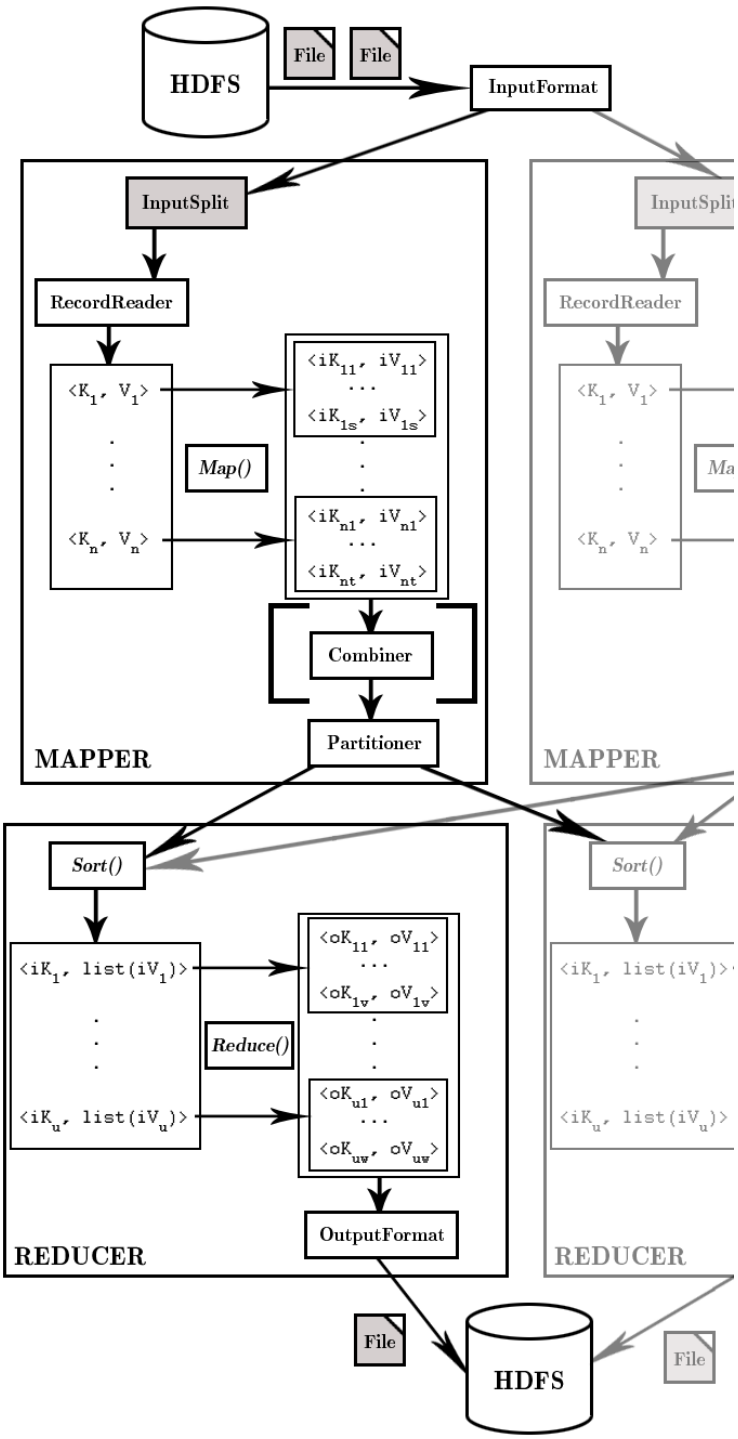
Het Map/Reduce model is niet nieuw. De functionaliteit was reeds bechikbaar in Lisp[22], alsook in andere functionele programmeertalen. Het komt er op neer dat op elk input record er een map operatie wordt toegepast om zo een set van tussenliggende key/value paren te bekomen. Vervolgens wordt een reduce operatie toegepast op alle values met dezelfde key om de data te combineren. De output is dus opnieuw een set van key/value paren. Het voordeel van dit programmeermodel is dat het gemakkelijk geparallelliseerd kan worden. In dit hoofdstuk zullen we eerst het programmeermodel in detail bespreken, om vervolgens de interne werking uitgebreid aan bod te laten komen. Verder bekijken de alternatieve implementaties van Map/Reduce genaamd Corona en YARN. Tot slot vermelden we ook het platform Pig dat programma's geschreven in de high-level programmeertaal Pig Latin omzet naar Map/Reduce programma's.

1.2.1 Programmeermodel

Zoals reeds vermeld bestaat een Map/Reduce programma uit twee functies die door de gebruiker worden gespecificeerd: een map functie die één voor één elk paar van een set input key/value paren omzet in een set tussenliggende key/value paren, en een reduce functie die de tussenliggende values met dezelfde key verwerkt en mogelijk omzet tot een set output key/value paren (typisch één of geen output paar per reduce operatie). Het Map/Reduce framework zorgt ervoor dat de tussenliggende key/value paren gegroepeerd worden per key.

1.2.1.1 Data flow

Om meteen een goed overzicht te verkrijgen van de werking van Map/Reduce overlopen we de dataflow met bijhorende types zoals getoond in Figuur 1.1. Hierbij zullen de gebruikte functies, klassen en structuren kort worden aangehaald, maar hun details worden pas later stap voor stap besproken.



Figuur 1.1: Schematische voorstelling van de data flow binnen het Map/Reduce framework.

Hoewel dit niet noodzakelijk is zijn de input bestanden voor een Map/Reduce job meestal opgeslagen in het gedistribueerde file systeem HDFS. Hoe deze bestanden worden opgesplitst wordt bepaald door de gebruikte *InputFormat* klasse. Deze klasse selecteert de bestanden die als input voor de Map/Reduce job gebruikt moeten worden en splitst deze vervolgens op in *InputSplits*. Een node uit de cluster die een InputSplit verwerkt noemen we een *Mapper*. Hoewel dit niet duidelijk is uit Figuur 1.1, kan één mapper meerdere input splits verwerken. Het geheel van verwerking van een input split door een mapper noemen we een *Map taak*. De *RecordReader* zet vervolgens een input split om in een set input key/value paren. Voor elk input key/value paar wordt vervolgens een *Map()* functie uitgevoerd. Deze zet één enkel input key/value paar om in een set van tussenliggende key/value paren. Het type van deze tussenliggende paren mag verschillen van de input key/value paren. Het resultaat van de Map functie op al de input key/value paren in een Mapper is een set van tussenliggende key/value paren. Met andere woorden, de resultaten van de Map functies worden als één geheel beschouwd. Dit is perfect mogelijk aangezien ze toch allemaal het zelfde output type hebben. De set van tussenliggende key/value paren kan vervolgens door een *Combiner* verwerkt worden, hoewel dit volledig optioneel is. Vaak zal een combiner functie erg gelijkaardig zijn aan de reducer, en soms zelfs identiek. Het spreekt voor zich dat de output van een combiner functie opnieuw een set tussenliggende key/value paren oplevert van hetzelfde type. Tot slot wordt deze set door een *Partitioner* verdeeld in een aantal partities. Hierbij komt het aantal partities overeen met het aantal Reducers. Dit zijn nodes op de cluster die reduce taken uitvoeren. De tussenliggende key/value paren worden zo verdeeld zodanig dat alle paren met dezelfde key bij eenzelfde Reducer terecht komen.

De Reducers krijgen dus een set van tussenliggende key/value paren als input. Deze worden gegroepeerd per key (wat een beetje ongelukkig de *Sort phase* genoemd wordt). We beschikken nu dus over een lijst van key/list(value) paren. Voor elk key/list(value) paar wordt er een reduce functie uitgevoerd. Het resultaat van één reduce functie is een lijst van output key/value paren. Het type van deze output paren mag opnieuw verschillen van tussenliggende key/value paren. Net zoals bij de Mapper worden de resultaten van de reduce functies beschouwd als een grote set van output paren. Hoe deze output naar het gedistribueerde file systeem wordt geschreven wordt bepaald door de gebruikte *OutputFormat* klasse.

1.2.1.2 Hello World

Het volgende voorbeeld is een beetje het "Hello World" programma van Map/-Reduce. Het doel is het aantal voorkomens per woord te vinden van een verzameling documenten. De map en reduce functie worden in pseudocode weergegeven.

```
1 map(Int key, String value) {
2     // key: byte offset
3     // value: lijn uit een bestand
4     for each word w in value
```

```
5           EmitIntermediate(w, ''1'');
6   }
7
8   reduce(String key, Iterator values) {
9       // key: een woord
10      // values: een lijst van counts
11      int result = 0;
12      for each v in values
13          result += ParseInt(v);
14      Emit(key, AsString(result));
15  }
```

De map functie output elk woord met een bijhorende counter van voorkomens, waarna de reduce functie het aantal voorkomens voor een bepaald woord optelt. Uiteraard moet ook de input voor de mapper gespecificeerd worden, wat in de volgende paragrafen besproken wordt.

1.2.1.3 Input formaten

Hadoop ondersteunt verschilllende types van input formaten, gaande van simpele tekst bestanden tot database input. Voor we hier dieper op ingaan tonen we de algemene vorm van de map en reduce functies in Hadoop:

```
1 map: (K1, V1) -> list(K2, V2)
2 reduce: (K2, list(V2)) -> list (K3, V3)
```

Het type van de input key/value paren van de map functie mogen verschillen van het type output key/value paren. Deze laatste moeten echter van hetzelfde type zijn als de input van de reduce functie. Het type output van de reducer mag echter weer verschillen met zijn input, hoewel dit vaak niet het geval is.

Input splits Hadoop splitst de input voor een Map/Reduce job in *input splits* of *splits*. Deze splits zijn van vaste grootte. Voor elke split wordt er een map taak gestart, die voor elk input record van de split de map functie uitvoert. Indien er veel splits zijn is de tijd om een split te verwerken klein ten opzichte van de verwerkingstijd van de volledige input. Dit leidt tot een betere load-balancing aangezien een snellere node meer splits kan verwerken dan een tragere node. Indien het aantal splits klein zou zijn, bijvoorbeeld gelijk aan het aantal beschikbare nodes, zouden we moeten wachten tot de tragere nodes klaar zijn met hun verwerking. In het extreme geval is het aantal splits zelfs kleiner dan het aantal beschikbare nodes, waardoor de cluster niet optimaal gebruikt wordt. Zelfs indien de hardware van verschillende nodes identiek is, is load balancing gewenst. Immers, er kunnen andere jobs draaiende zijn en bovendien kunnen er processen falen. Het is duidelijk dat de kwaliteit van load-balancing verhoogt naarmate de grootte van de splits afneemt. Men moet echter oppassen dat de split grootte niet te klein gekozen wordt, aangezien het managen van de splits alsook de creatie van map taken voor overhead zorgt. Per default is de

split grootte gelijk aan de grootte van een HDFS block, die standaard 64 MB bedraagt. De reden hiervoor wordt uitgelegd in paragraaf 1.3.1 Blocks.

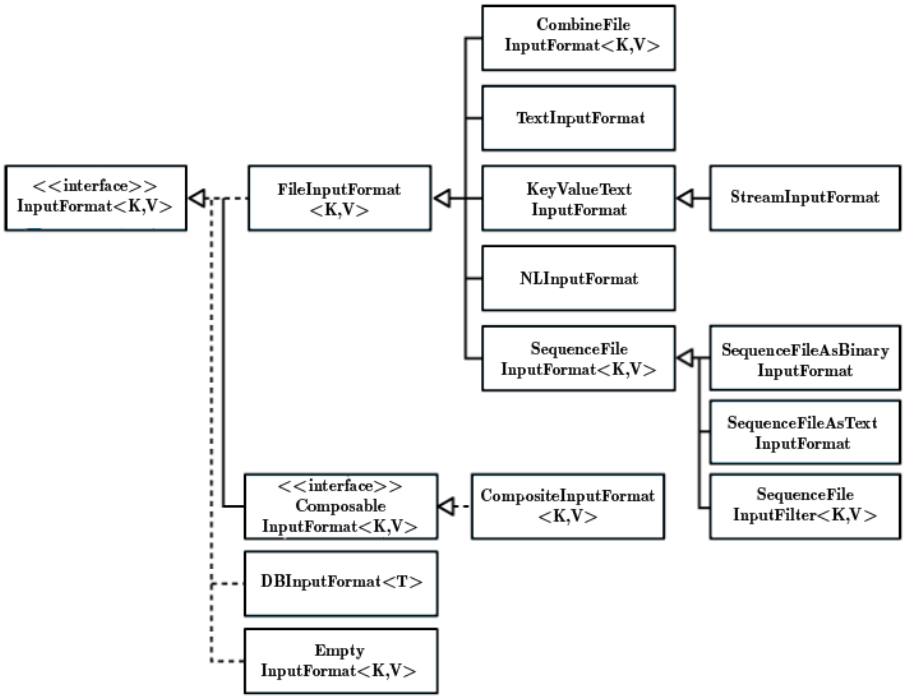
InputFormat Bij het schrijven van een Map/Reduce applicatie zal je je niet direct moeten bezig houden met input splits, aangezien deze aangemaakt worden door de *InputFormat* klasse. Een InputFormat instantiatie is verantwoordelijk voor het creëeren van splits en het verdelen ervan in records. Zijn interface is als volgt:

```
1 public abstract class InputFormat<K, V> {
2     public abstract List<InputSplit> getSplits(JobContext
          context) throws IOException, InterruptedException
         ;
3
4     public abstract RecordReader<K, V>
5     createRecordReader(InputSplit split,
         TaskAttemptContext context)
6      throws IOException, InterruptedException;
7 }
```

Zoals de naam doet vermoeden berekent de functie getSplits de splits voor een Map/Reduce job. De *jobtracker*, een soort van 'master node' die de job coördineert, gebruikt de locaties van deze input splits om map tasks toe te kennen aan worker nodes. Deze worker nodes of *tasktrackers* worden gekozen zodanig dat de data zich zo dicht mogelijk bij de node bevindt. Merk op dat de input splits niet noodzakelijk dezelfde grootte hebben. Dit is bijvoorbeeld het geval wanneer een input split gelijk is aan een file. De grootte van de splits is bepalend voor de volgorde waarin ze verwerkt worden. Zo worden de grootste splits eerst verwerkt in een poging de totale runtime van de job te minimaliseren. We beschouwen een klein voorbeeldje met twee worker nodes W_1, W_2 en vijf input splits I_1, I_2, I_3, I_4, I_5 dat dit verduidelijkt. Stel dat de grootte van de splits als volgt is:

$I_1 = 1\text{MB}$, $I_2 = 1\text{MB}$, $I_3 = 1\text{MB}$, $I_4 = 1\text{MB}$, $I_5 = 4\text{MB}$

Het is duidelijk dat de grootte van een split recht evenredig is met de verwerkingsduur ervan. Stel dat in dit geval de verwerking 1MB één seconde duurt, ongeacht de totale grootte van de split. Een naïeve uitvoering zou als volgt gaan. De jobtracker kent I_1 toe aan W_1 en I_2 aan W_2. Na één seconde zijn de splits verwerkt en wordt I_3 toegekend aan W_1 en I_4 aan W_2. Tot slot wordt na nog een seconde I_5 toegekend aan W_1. De totale uitvoeringstijd bedraagt dus $1 + 1 + 4 = 6$ seconden. In de praktijk zal de jobtracker echter de grootste splits eerst toekennen. Zo zal in het voorbeeld dus eerst I_5 worden toegekend aan W_1 en I_1 aan W_2. Na één seconde is W_2 klaar met de verwerking van I_1 en krijgt hij I_2 toegekend. Analoog voor I_3 en I_4. Na vier seconden is W_1 klaar met de verwerking van I_5 en W_2 met de verwerking van I_4. De uitvoeringstijd bedraagt in dit geval dus slechts vier in plaats van zes seconden.



Figuur 1.2: De klasse hiërarchie van InputFormat

Naast de getSplits functie bevat de InputFormat interface ook een createRecordReader functie. Een map taak roept deze functie aan met een bepaalde input split om een RecordReader te bekomen. Deze laatste is niet veel meer dan een iterator over records die gebruikt wordt door de map taak om key/value paren te genereren en door te geven aan de map functie.

FileInputFormat *FileInputFormat* is de basisklasse voor alle implementaties van InputFormat die bestanden gebruiken als hun bron voor input data. In deze klasse wordt bepaald welke bestanden tot de input van een Map/Reduce job behoren, en wordt er tevens de functionaliteit voor het genereren van splits voor de input files geïmplementeerd. Figuur 1.2 toont afgeleide klassen van FileInputFormat.

FileInputFormat input paths De input bestanden voor een job worden gespecificeerd door een verzameling van paden. Deze paden kunnen zowel een bestand, een folder, of een *glob* voorstellen. Merk op dat een folder enkel bestanden, en geen andere folders mag bevatten! Deze zullen immers als een bestand geïnterpreteerd worden, wat een foutmelding zal genereren. Indien je recursief toch alle bestanden in subfolders als input wilt gebruiken kan dit door de

mapred.input.dir.recursive property op *true* te zetten. *Globbing* is het gebruik maken van wildcard karakters om meerdere bestanden te specificeren met één uitdrukking. Hadoop ondersteunt dezelfde glob karakters als Unix bash, zoals getoond in Tabel 1.1. We illustreren dit met een klein voorbeeldje. Stel je hebt een *root* folder met daarin de folders *2008*, *2009*, *2010*. Dan is de glob */200?* een afkorting voor */2008 /2009*. Jammer genoeg zijn glob uitdrukkingen niet altijd krachtig genoeg. Zo is het meestal niet mogelijk om een specifieke file uit te sluiten. Hiervoor biedt Hadoop een oplossing met behulp van de *PathFilter* interface. Door deze te implementeren kan je bijvoorbeeld paden uitsluiten die aan een zekere reguliere expressie voldoen, zoals getoond wordt in onderstaand code fragment.

```
1 public class ExcludeRegex implements PathFilter {
2     private final String regex;
3     public ExcludeRegex(String regex) {
4         this.regex = regex;
5     }
6     public boolean accept(Path path) {
7         return !path.toString().matches(regex);
8     }
9 }
```

FileInputFormat heeft twee functies om input paden voor een job toe te voegen, namelijk de volgende:

```
1 public static void addInputPath(Job job, Path path)
2 public static void addInputPath(Job job, String
    commaSeparatedPaths)
```

Om bepaalde bestanden uit te sluiten kunnen we nu gebruik maken van de setInputPathFilter() functie van FileInputFormat:

```
1 public static void setInputPathFilter(Job job,
2     Class<? extends PathFilter> filter)
```

Stel nu dat we alle bestanden uit de folders 2008 en 2009 willen selecteren, behalve de bestanden met een png-extensie. Dit kunnen we bereiken met de volgende code:

```
1 fif.addInputPath(job, new Path("/200?"));
2 fif.setInputPathFilter(job,
3     new ExcludeRegex(("^.*/.*\.png$"));
```

Glob	Betekenis
*	één of meerdere willekeurige karakters
?	een willekeurig karakter
$[ab]$	een karakter uit de set $\{a, b\}$
$[\hat{}ab]$	een karakter dat niet in de set $\{a, b\}$ zit
$[a - b]$	een karakter uit de range $[a, b]$ zit, waarbij a lexicografisch voor b komt
$[\hat{}a - b]$	een karakter dat niet in de range $[a, b]$ zit
$\{a, b\}$	ofwel expressie a, ofwel expressie b
\c	een karakter c met c een metakarakter

Tabel 1.1: Beschikbare Glob karakters in Hadoop

FileInputFormat input splits We hebben reeds besproken hoe we een set van input bestanden kunnen specificeren. De volgende vraag is hoe FileInputFormat deze bestanden omzet in splits. Hiervoor is het belangrijk om te weten dat een HDFS block de grootste eenheid van data is die gegarandeerd op een enkele node opgeslagen is (zie paragraaf 1.3.1.1). Per default is de grootte van een split gelijk aan de grootte van een HDFS block, en zullen dus enkel bestanden groter dan een HDFS block gesplitst worden. Indien de gebruiker toch controle wil hebben over de splitgrootte, kan dit door gebruik te maken van de Hadoop properties uit Tabel 1.2. Door bijvoorbeeld mapred.min.split.size groter te kiezen dan een HDFS block, zullen splits groter zijn dan een block. Dit zal echter het aantal blocks dat niet lokaal is voor een map taak vergroten, aangezien een HDFS block de grootste eenheid data is die gegarandeerd op een node staat. Bijgevolg is het beter de block grootte aan te passen indien je applicatie een grotere minimum splitgrootte vereist. Formeel wordt de grootte van een split bepaald door de volgende formule:

```
1 max(minimumSize, min(maximumSize, blockSize))
```

Aangezien per default de blockSize kleiner is dan de maximumSize en groter dan de minimumSize klopt onze eerdere stelling dat de splitgrootte gelijk is aan de grootte van een HDFS block.

Naam	Type	Default waarde	Beschrijving
mapred.min.split.size	int	1	Kleinste geldige grootte voor een split in bytes
mapred.max.split.size	long	long.MAX_VALUE	Grootste geldige splitgrootte in bytes
dfs.block.size	long	64 MB	Grootte van een HDFS block in bytes

Tabel 1.2: Properties voor het instellen van de grootte van een input split.

Kleine bestanden Herinner je dat er voor elke input split een map taak wordt opgestart. Veel input splits zorgt dus voor extra overhead. Dit kan problematisch worden bij zeer veel kleine bestanden. Stel dat je bijvoorbeeld een bestand van 1 GB hebt dat in zestien splits van 64MB wordt verdeeld, en 10.000 bestanden van 100 KB. Deze 10.000 bestanden gebruiken elk een map taak, wat een enorme overhead met zich meebrengt. Hierdoor kan de uitvoeringstijd van deze job tien, tot honderden keren hoger liggen dan de taak met slechts een enkel input bestand. Dit probleem kan deels verholpen worden door gebruik te maken van de abstracte klasse *CombineFileInputFormat* (zie Figuur 1.2). In tegenstelling tot FileInputFormat, plaatst CombineFileInputFormat zoveel mogelijk bestanden in een split, zodanig dat elke map taak meer data moet verwerken. Hierbij wordt rekening gehouden met de locatie van de bestanden in de cluster. Met andere woorden, er zal eerst getracht worden de map taak uit te voeren op de node waar de data staat. Indien dit niet mogelijk is wordt de voorkeur gegeven aan een node die op hetzelfde rack geplaatst is. Het blijft echter een goed idee om veel kleine bestanden te vermijden aangezien dit een zware tol eist van de Namenode (zie 1.3.1 Namenode). Tot slot merken we nog op dat het gebruik van CombineFileInputFormat wat extra werk zal vergen aangezien er geen concrete klassen van geïmplementeerd zijn (in tegenstelling tot FileInputFormat die bijvoorbeeld door de klasse TextInputFormat geïmplementeerd wordt).

Het voorkomen van input splits Sommige applicaties vereisen dat bestanden niet gesplitst worden. Dit is bijvoorbeeld het geval wanneer je in een map taak wil nagaan of de records in een bestand op een bepaalde manier geordend zijn. Er zijn meerdere oplossingen voor dit probleem. De eerste is om de minimum split grootte zo in te stellen dat deze groter is dan het grootste bestand uit de input (door bijvoorbeeld de split grootte gelijk te stellen aan long.MAX_VALUE). Een tweede manier is door een subklasse te implementeren voor de subklasse van FileInputFormat die je gebruikt, en vervolgens de *isSplittable()* functie te overschrijven en *false* te returnen. Merk op dat je ook nog een RecordReader nodig hebt die de volledige inhoud van een file als een enkele waarde beschouwd. Deze dient de gebruiker zelf te implementeren.

Tekstuele input Hadoop bevat verschillende concrete implementies van de abstracte klasse InputFormat voor het verwerken van tekstuele input, waaronder *TextInputFormat*, *KeyValueTextInputFormat*, *NLineInputFormat* en *XML*.

TextInputFormat Bij het TextInputFormat komt een record overeen met een lijn uit de input, en de key met de offset (in bytes) van deze lijn ten opzichte van het begin van het bestand. De waarde is de inhoud van deze lijn, zonder de line terminators. We illustreren dit met een simpel voorbeeldje.

```
Dit is een
bijzonder simpel
voorbeeld.
```

Dit bestand zou worden omgezet in de volgende records:

```
(0, Dit is een)
(11, bijzonder simpel)
(27, voorbeeld.)
```

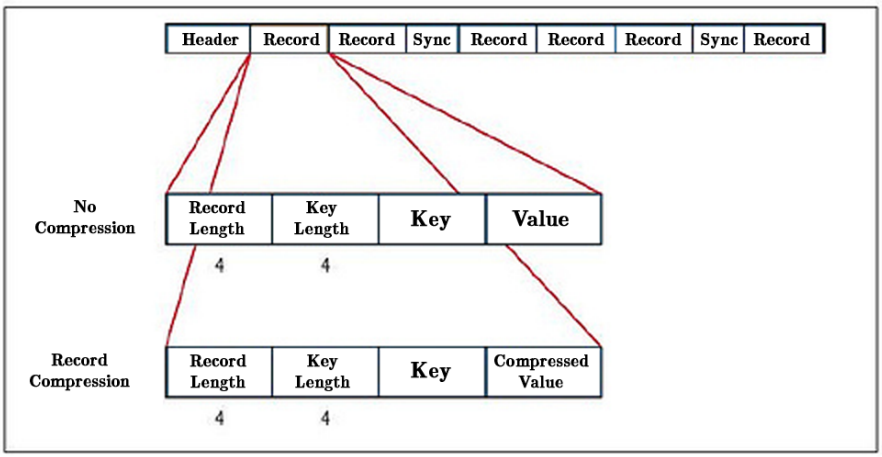
Je vraagt je misschien af waarom de key de offset in bytes is, en niet het lijn nummer. Herinner je dat een bestand wordt opgesplitst in input splits, dewelke onafhankelijk (en dus mogelijk op hetzelfde tijdstip) verwerkt worden. De grootte van de voorafgaande splits is gekend (door de Namenode), maar het aantal voorafgaande lijnen is dit niet. Een input split wordt immers gemaakt op basis van het aantal bytes, niet het aantal lijnen. De offset kan dus berekend worden door de grootte van de vorafgaande splits op te tellen. Lijn nummers bijhouden is niet mogelijk zonder de onafhankelijkheid van splits op te geven, wat uiteraard niet gewenst is. Bovendien is de combinatie van een bestandsnaam en een offset net zo goed een unieke identifier in het file systeem.

Merk op dat het einde van een lijn meestal niet mooi samenvalt met het einde van een HDFS block. Hierdoor zal een node die een input split lokaal verwerkt toch één of meerdere remote reads moeten doen. Deze overhead is echter zeer beperkt en niet significant voor de totale uitvoertijd van een Map/Reduce job.

KeyValueTextInputFormat Een andere concrete implementatie van InputFormat is KeyValuetextInputFormat, die in de praktijk vaak gebruikt wordt. Hierbij is elke lijn in een bestand een key/value paar, dewelke gescheiden worden door een delimiter. Deze delimiter kan worden ingesteld door de *mapreduce.input.keyvaluelinerecordreader.key.value.separator* property aan te passen. Standaard staat hij ingesteld op het tab-karakter. Het default output formaat van Hadoop is trouwens van deze vorm.

NLineInputFormat Verder is er ook nog de NLineInputFormat klasse. Zoals de naam al doet vermoeden zorgt deze klasse ervoor dat een map taak een vast aantal lijnen krijgt toegekend. Net zoals bij TextInputFormat is de byte offset van een lijn de key, en de waarde van deze lijn de value. Door de *mapreduce.input.lineinputofrmat.linespermap* property wordt bepaald hoeveel lijnen elke map taak als input krijgt (per default 1). Merk op dat een klein aantal input lijnen per map taak kan zorgen voor veel overhead. Het aantal map taken dat gestart moet worden is immers omgekeerd evenredig met het aantal lijnen per split. Een klein aantal lijnen per map taak is dus enkel een goed idee indien de verwerking van een lijn zeer reken-intensief is. Een goed voorbeeld hiervan is bijvoorbeeld als de lijn de parameters bevat voor een bepaalde simulatie uit te voeren.

XML Het laatste tekstuele input formaat dat we bespreken is XML. Uiteraard bestaat de optie om een XML document door één map taak te laten verwerken. Dit is niet gewenst indien het een erg groot bestand betreft. In dat geval splitsen we het XML bestand door pattern-matching van bepaalde XML



Figuur 1.3: De structuur van een SequenceFile.

tags. Hiervoor dien je gebruik te maken van de *StreamInputFormat* klasse, en stel je de *stream.recordreader.class* property in op *StreamXmlRecordReader*. Vervolgens dien je de reader nog te configureren om de patronen voor de begin- en eindtags van het XML element te herkennen.

Binaire input Hadoop biedt ook ondersteuning voor binaire formaten met de *SequenceFileInputFormat* klasse. De interne structuur van een *SequenceFile* wordt getoond in Figuur 1.3. Zoals je kan zien is dit door Hadoop gedefinieerde file formaat opgebouwd uit een header, gevolgd door een aantal records die op bepaalde plaatsen gescheiden worden door *sync markers*. De header bevat onder andere de namen van de key en value klassen, compressie details, en eventuele metadata meegegeven door de gebruiker. Een sync point wordt aangegeven door een sync marker en wordt gebruikt om vanuit een arbitrair punt in het bestand het beginpunt van een record te vinden. Ze zorgen voor minder dan 1 % opslag overhead en komen niet noodzakelijk tussen elk paar records voor. Indien bijvoorbeeld de record reader faalt om het volgende record op te halen (bijvoorbeeld omdat de huidige positie in het bestand niet het begin is van een record) kan de sync() functie aangeroepen worden om de eerstvolgende record grens te vinden. Het belangrijkste gebruik van de sync points is echter om vanuit de beginpositie van een split de beginpositie van een record te vinden (herinner je dat de input splits en record grenzen zelden overeen komen). Een record tot slot is opgebouwd uit de lengte van een record, de lengte van de key, gevolgd door de waarde van de key en de al dan niet gecomprimeerde waarde van de value.

Meerdere inputs Het is niet ondenkbaar dat de input voor een Map/Reduce programma in verschillende formaten is opgeslagen, of dat er verschillende representaties zijn voor de input. Zo kan hetzelfde soort data bijvoorbeeld in een binaire SequenceFile zijn als in een tekstueel bestand. Verder is het mogelijk dat de data in de bestanden op een verschillende manier wordt voorgesteld. Data formaten veranderen immers wel vaker na verloop van tijd, en je zou natuurlijk liefst je oude mapper voor die data blijven gebruiken in plaats van deze data om te vormen naar het nieuwe formaat. Hiervoor biedt de *MultipleInputs* klasse een oplossing. Deze stelt je immers in staat een InputFormat en Mapper te specificeren per input pad. Herinner je dat we de de addInputPath() functie van FileInputFormat gebruikten om de input paden aan te duiden. Deze wordt nu vervangen door:

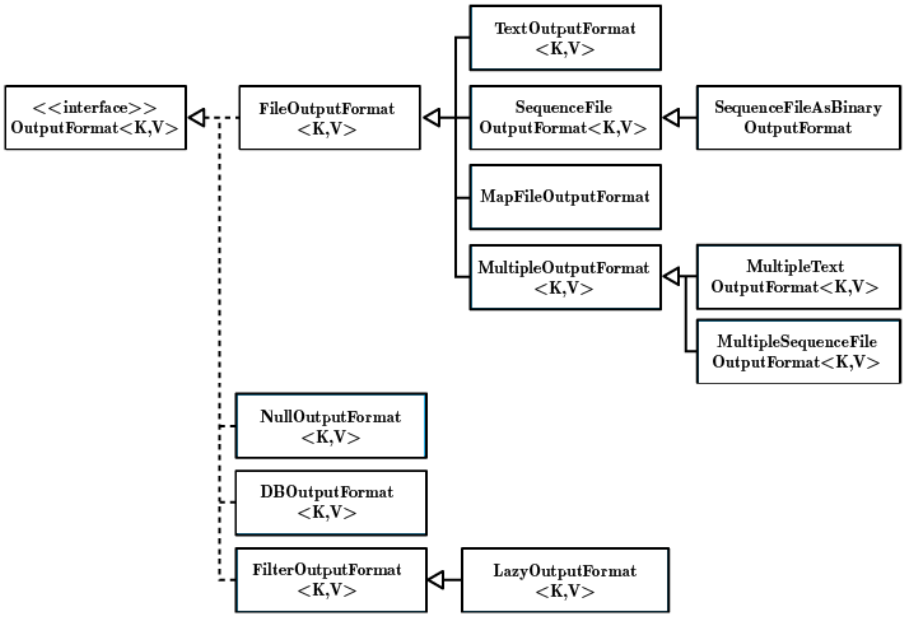
```
1 MultipleInputs.AddInputPath(Job job, Path path,
2 Class<? extends InputFormat> inputFormatClass,
3 Class<? extends Mapper> mapperClass)
```

Door deze functie meerdere malen aan te roepen kunnen we dus meerdere input representaties verwerken, elk met een andere mapper. Het spreekt voor zich dat alle mapper functies dezelfde output types moeten hebben. De reducers zijn immers niet op de hoogte zijn van de verschillende mappers gebruikt voor het genereren van de tussenliggende key/value paren.

Database input Als laatste input formaat vermelden we nog *DBInputFormat*. Door gebruik te maken van JDBC laat deze klasse toe om data in te lezen vanuit een relationele database. Merk op dat deze klasse vooral geschikt is voor het laden van relatief kleine datasets. Immers, indien teveel mappers data vanuit de database willen lezen kan deze overbelast raken (JDBC heeft standaard geen sharding mogelijkheden). Er zijn ook alternatieven beschikbaar voor het lezen van data vanuit een relationele database, zoals bijvoorbeeld Sqoop[3].

1.2.1.4 Output formaten

Hadoop biedt uiteraard ook een reeks output formaten aan die grotendeels overeenkomen met de hiervoor besproken input formaten. Figuur 1.4 toont de klasse hiërarchie van OutputFormat.



Figuur 1.4: Overzicht van de beschikbare output formaten in Hadoop.

Tekstuele output De *TextOutputFormat* klasse is de tegenhanger van KeyValueTextInputFormat, en schrijft dus key/value paren gescheiden door een delimiter naar een tekstbestand. Deze delimiter staat per default ingesteld op het tab-karakter en kan worden aangepast met de *mapreduce.output.textoutputformat.seperator* property. De klasse accepteert alle key/value types waarvoor de toString() functie voor gedefinieerd is. Deze klasse kan ook gebruikt worden om output te genereren die met TextInputFormat terug ingelezen zal worden. Herinner je dat bij TextInputFormat een lijn overeenkomt met de waarde van de value, en de offset in bytes de key is. Je genereert dit soort output door voor de value (of key) gebruik te maken van een NullWritable type.

Binaire output SequenceFileOutputFormat genereereert output die gelezen kan worden met de SequenceFileInputFormat klasse. Dit outputformaat is vooral geschikt voor het creëeren van output die nog door andere Map/Reduce taken als input zal worden gebruikt omwille van de compressie mogelijkheden. Voor meer informatie over compressie in Hadoop, zie paragraaf 1.3.7.

Meerdere outputs Per default genereert elke reducer één output bestand van de vorm *part-r-xxxxx* met *xxxxx* het partitienummer. De *MultipleOutputs* klasse laat je echter toe om de reducer (of mapper) meerdere bestanden te laten genereren met een bestandsnaam van het formaat *naam-r-xxxxx*. *xxxxx* is opnieuw het partitienummer, *naam* is een willekeurige string die door het pro-

gramma bepaalp wordt. Het partitienummer zorgt ervoor dat er geen bestanden gegenereert kunnen worden met dezelfde naam. De functie waarmee dit kan is de volgende:

```
1 multipleOutputs.write(KEYOUT key, VALUEOUT value, String
      path)
```

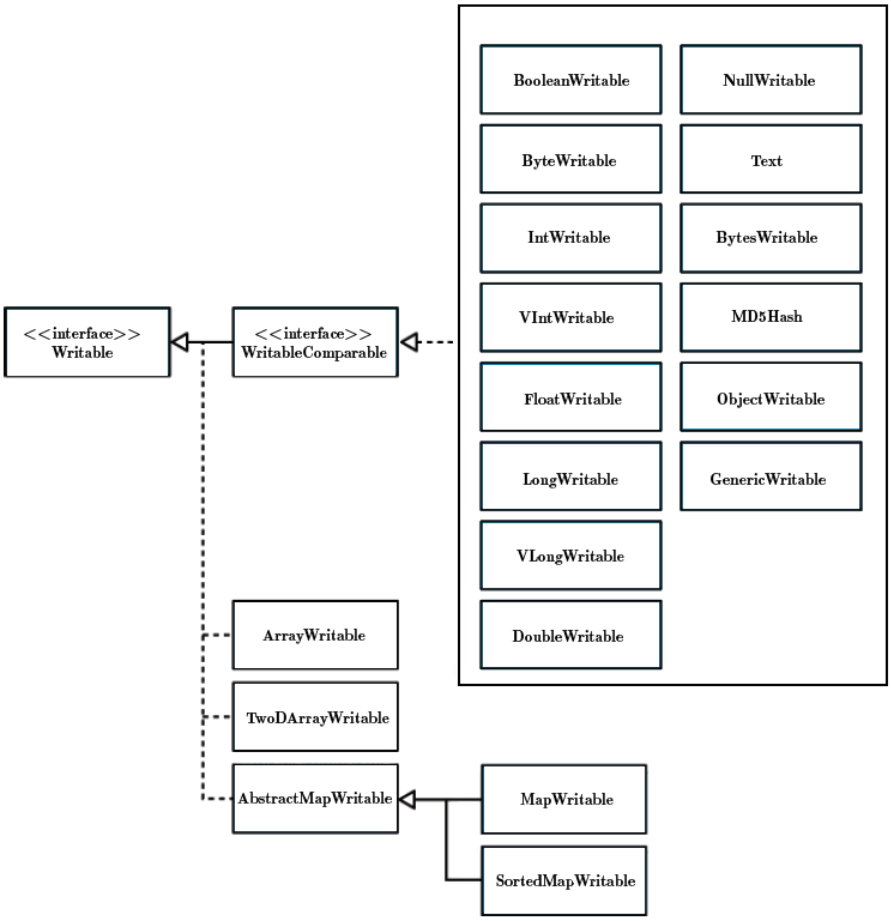
Hierbij zijn KEYOUT en VALUEOUT vanzelfsprekend de output types voor de key en value. Het output path kan trouwens het /-teken bevatten waardoor er ook naar subdirectories geschreven kan worden.

Vermijden van lege output bestanden Standaard maakt elke reducer een output bestand aan, ook al is deze leeg. Indien dit niet gewenst is kan je gebruik maken van de *LazyOutputFormat* klasse. Hierbij wordt er pas een output bestand aangemaakt van zodra het eerste output record gegenereerd is.

Database output Tot slot kan er ook nog database output gegeneerd worden met de *DBOutputFormat* klasse. Ook hier geldt de opmerking dat dit vooral geschikt is voor kleinere hoeveelheden data.

1.2.1.5 Serializatie

We bespreken in deze paragrafen het omzetten van objecten in Hadoop naar een byte stream (serializatie) voor netwerktransmissie of het opslaan naar harde schijf, alsook de deserializatie van een byte stream. Het is van belang dat een serializatie formaat compact is om zo goed mogelijk gebruik te maken van de bandbreedte in de cluster aangezien dit een schraarse resource is. Bovendien moet het serializeren en deserializeren ook zo weinig mogelijk tijd vergen. Omwille van deze redenen biedt Hadoop een compact en snel serialisatie formaat *Writables* aan. In Figuur 1.5 wordt een overzicht gegeven van de beschikbare Writable klassen.



Figuur 1.5: De Writables klasse hiërarchie.

De Writable interface is als volgt gedefinieerd:

```
1 public interface Writable {
2     void write(DataOutput out) throws IOException;
3     void readFields(DataInput in) throws IOException;
4 }
```

De write en readFields functies zorgen respectievelijk voor de serializatie en deserializatie van een object. Onderstaand voorbeeldje toont hoe je een nieuw IntWritable object kan aanmaken.

```
1 IntWritable writable = new IntWritable(67);
```

Aangezien de meeste implementaties van de *WritableComparable* interface een voor de hand liggende betekenis hebben gaan we ze niet individueel bespreken.

De mogelijkheid tot het vergelijken van types is trouwens cruciaal voor Map/Reduce, aangezien er keys vergeleken worden in de sorteer fase. Deze sorteer fase wordt geoptimaliseerd met behulp van de *RawComparable* interface, dewelke toelaat records van een byte stream te vergelijken zonder de objecten te deserializeren. Uiteraard bespraart dit de overhead die het creëeren van objecten met zich meebrengt. De RawComparable interface bevat enkel onderstaande *compare* functie:

```
1 public int compare(byte[] b1, int s1, int l1, byte[] b2,
    int s2, int l2);
```

Hierbij geven l1 en l2 de lengte, en s1 en s2 de startposities aan van de te vergelijken byte streams. Merk op dat Figuur 1.5 ook arrays, tweedimensionale arrays en maps ondersteund, hoewel deze geen implementatie zijn van de WritableComparable inteface. Ze moeten bijgevolg gedeserializeert worden alvorens ze gesorteerd kunnen worden.

De Writable interface implementeren De reden dat we bovenstaande interfaces expliciet vermelden is omdat het voor sommige applicaties nodig kan zijn om je eigen Writable klasse te implementeren. Indien je applicatie enkel gebruik maakt van een reeds geïmplementeerde klasse zal je normaal niet met deze functies in aanraking komen. Aangezien de Writable implementaties van Hadoop voor de meeste doeleinden voldoende zijn, en ze bovendien gefinetuned zijn om een zo goed mogelijke performance te garanderen, beperk je je (indien mogelijk) ook best tot deze klassen. Aangezien deze Writables aan de basis liggen van de data transmissie in Hadoop hebben ze immers een grote impact op performantie van een job. Indien je toch je eigen Writable klasse moet implementeren loont het dus de moeite om ook een RawComparator te implementeren, desondanks dat je je hier met details op byte level zal moeten bezig houden.

1.2.1.6 Vervolg Hello World

Nu we zowel de input als output formaten van Hadoop, alsook serialisatie besproken hebben, zijn we klaar om de pseudo-code van het "Hello World" voorbeeldje uit paragraaf 1.2.1.2 om te zetten in onderstaande broncode. Herinner je dat het doel is om het aantal voorkomens van woorden te tellen.

```
1 public class WordCount {
2     public static class Map extends MapReduceBase
        implements Mapper<LongWritable, Text, Text,
        IntWritable> {
3         private final static IntWritable one = new
            IntWritable(1);
4         private Text word = new Text();
5
```

```
6           public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output,
                Reporter reporter) throws IOException {
7               String line = value.toString();
8               StringTokenizer tokenizer = new
                    StringTokenizer(line);
9               while (tokenizer.hasMoreTokens()) {
10                  word.set(tokenizer.nextToken());
11                  output.collect(word, one);
12              }
13          }
14      }
15
16      public static class Reduce extends MapReduceBase
            implements Reducer<Text, IntWritable, Text,
            IntWritable> {
17          public void reduce(Text key, Iterator<
                IntWritable> values, OutputCollector<Text,
                IntWritable> output, Reporter reporter)
                throws IOException {
18              int sum = 0;
19              while (values.hasNext()) {
20                  sum += values.next().get();
21              }
22              output.collect(key, new IntWritable(sum));
23          }
24      }
25
26      public static void main(String[] args) throws
            Exception {
27          JobConf conf = new JobConf(WordCount.class);
28          conf.setJobName("wordcount");
29
30          conf.setOutputKeyClass(Text.class);
31          conf.setOutputValueClass(IntWritable.class);
32
33          conf.setMapperClass(Map.class);
34          //de combiner is optioneel
35          conf.setCombinerClass(Reduce.class);
36          conf.setReducerClass(Reduce.class);
37
38          conf.setInputFormat(TextInputFormat.class);
39          conf.setOutputFormat(TextOutputFormat.class);
40
41          FileInputFormat.setInputPaths(conf, new Path(args
                [0]));
```

```
42          FileOutputFormat.setOutputPath(conf, new Path(
                args[1]));
43
44          JobClient.runJob(conf);
45      }
46  }
```

Zoals je kan zien worden er in de klasse WordCount twee andere klassen geïmplementeerd: *Mapper* en *Reducer*. Ze bevatten respectievelijk de map en reduce functie. De mapper interface verwacht zowel de input key/value types, als de tussenliggende key/value types. De reducer krijgt op zijn beurt de types van de tussenliggende key/value paren alsook de output types. De map functie heeft de volgende specificatie:

```
1 map(K1 key, V1 value, OutputCollector<K2,V2> output,
      Reporter reporter)
```

De OutputCollector uit de map functie verzamelt de output keyaren, dewelke kunnen worden toegevoegd door zijn functie *collect(Object,Object)* aan te roepen.

Vooruitgang rapporteren De *Reporter* wordt gebruikt om vooruitgang te rapporteren, of om aan te geven dat het proces nog 'leeft'. Dit laatste is vooral nodig wanneer de verwerking van individuele key/value paren bijzonder veel tijd vergt. Hadoop zal immers annemen dat een taak ge-time-out is en het proces beëindigen indien er geen vooruitgang gerapporteerd wordt. Concreet kan dit door de *progress()* of de *setStatus(String status)* functie aan te roepen. Hadoop herkent ook automatisch vooruitgang in het lezen van een input record of het wegschrijven van output records. Een andere manier om dit probleem aan te pakken is door de *mapreduce.task.timeout* property een hogere waarde toe te kennen (of zelfs nul voor geen time-outs).

Job configuratie Bovenstaande uitleg is uiteraard ook van toepassing op de reduce functie, waardoor we enkel nog de job configuratie parameters dienen te bespreken. Dit gebeurt aan de hand van de JobConf interface, dewelke de Mapper, Reducer, InputFormat, OutputFormat klassen specificeert. Ook de paden van de input en output bestanden kunnen worden bepaald, eventueel met behulp van een Glob zoals in paragraaf 1.2.1.3.

Partitioner Verder kan ook de *partitioner* worden ingesteld met behulp van JobConf. De partitioner bepaalt welke reducer welke tussenliggende key/value paren zal verwerken. Herinner je dat het noodzakelijk is dat alle paren met dezelfde key bij dezelfde reducer terecht komen. Het is bovendien van belang dat elke mapper onafhankelijk zijn tussenliggende key/value paren kan partitioneren, zodanig dat er geen onderlinge communicatie nodig is. Het aantal partities is gelijk aan het aantal reducers, dat kan worden ingesteld door de

JobConf.setNumReduceTasks() methode (per default 1). De interface van de partitioner is als volgt:

```
1 public interface Partitioner<K, V> extends
      JobConfigurable {
2     int getPartition(K key, V value, int numPartitions);
3 }
```

Zoals je kan zien bevat deze één functie die geïmplementeerd moet worden, namelijk *getPartition()*. Deze functie bepaalt de partitie van een key/value paar aan de hand van het paar zelf en het aantal partities. De default partitioner is de *HashPartitioner*, dewelke de hashcode van de key module het aantal partities gebruikt om te bepalen tot welke partitie een key/value paar behoort. De HashPartitioner is een goede keuze indien je data uniform over de mogelijke waarden verdeeld is. Indien een bepaalde waarde echter (veel) vaker voorkomt dan andere kan dit ervoor zorgen dat een bepaalde reducer een veel grotere hoeveelheid data moet verwerken. Dit leidt tot een langere job uitvoeringstijd aangezien er op deze reducer gewacht zal moeten worden. In dat geval is het beter een eigen partitioner te implementeren die de data wel uniform verdeelt, en vervolgens deze in te stellen door middel van de *JobConf.setPartitionerClass()* methode.

Combiner Tot slot kan je met JobConf ook de *combiner* instellen. Een combiner zal er voor zorgen dat de map functie tussenliggende key/value paren niet meteen output, maar bijhoudt in lijsten. Voor elke tussenliggende key waarde is er een lijst. Van zodra een bepaald aantal key/value paren verwerkt is door de mapper zal de combiner alle lijsten verwerken, en de resulterende key/value paren outputten. Deze zullen op hun beurt door een reducer verwerkt worden, net alsof ze nooit door een combiner verwerkt zijn. Het voordeel van een combiner is dat er mogelijk minder bandbreedte wordt verbruikt op de cluster aangezien de combiner tussenliggende waarden aggregeert. In ons voorbeeld is de combiner dezelfde als de reducer. Deze zal dus het aantal voorkomens van een woord per mapper bepalen, en dit resultaat naar het file systeem wegschrijven. Dit zal hoogst waarschijnlijk voor minder disk read en writes worden, en dus mogelijk de totale uitvoeringstijd van de job verminderen. Merk op dat het instellen van een combiner niet altijd mogelijk is. Stel bijvoorbeeld dat we de gemiddelde temperatuur willen berekenen van de volgende data: (Maandag, 10) (Dinsdag, 14) Woensdag, 19) (Donderdag, 24) (Vrijdag, 29) Stel dat er twee map taken zijn, en dat de eerste de dagen maandag en dinsdag verwerkt. Het gemiddelde hiervan is 12°. De tweede mapper berekent het gemiddelde van de overige dagen, dit is 24°. Bijgevolg gaat het resultaat van de reduce functie 18° zijn. Dit resultaat is echter incorrect aangezien de gemiddelde temperatuur 19.2° is. Het is dus duidelijk dat de mogelijkheid om een combiner in te stellen applicatie-afhankelijk is.

1.2.1.7 De applicatie uitvoeren

Het enige dat ons nog rest is de job uit te voeren op een Hadoop installatie (dit kan zowel voor een lokale standalone, pseudo-gedistribueerde of volledig gedistribueerde installatie). We starten eerst de cluster op, moest dit nog niet gebeurd zijn:

```
1 $ HADOOP_HOME/bin/start-all.sh
```

Dit zal een Namenode, Datanode, Jobtracker en Tasktracker opstarten (zie paragraaf 1.3.1). We compileren nu WordCount.java en maken een jar aan:

```
1 $ javac -classpath HADOOP_CLASSPATH WordCount.java
2 $ jar -cvf wordcount.jar .
```

Vervolgens plaatsen we de inputbestanden, waarvan we veronderstellen dat ze in de map /localpath/input zitten, in HDFS:

```
1 $ bin/hadoop dfs -copyFromLocal /localpath/input.txt /
    hadooppath/input
```

Tot slot voeren we de applicatie uit:

```
1 $ bin/hadoop jar wordcount.jar wordcount /hadooppath/
    input /hadooppath/output
```

We kunnen de output bestanden ook weer uit het file systeem halen en lokaal opslaan als volgt:

```
1 $ bin/hadoop dfs -copyToLocal /hadooppath/output/part-r
    -00000
```

Hiermee eindigen we de introductie tot het Map/Reduce programmeermodel in Hadoop. In de volgende paragrafen gaan we dieper in op de interne werking ervan.

1.2.2 Interne werking

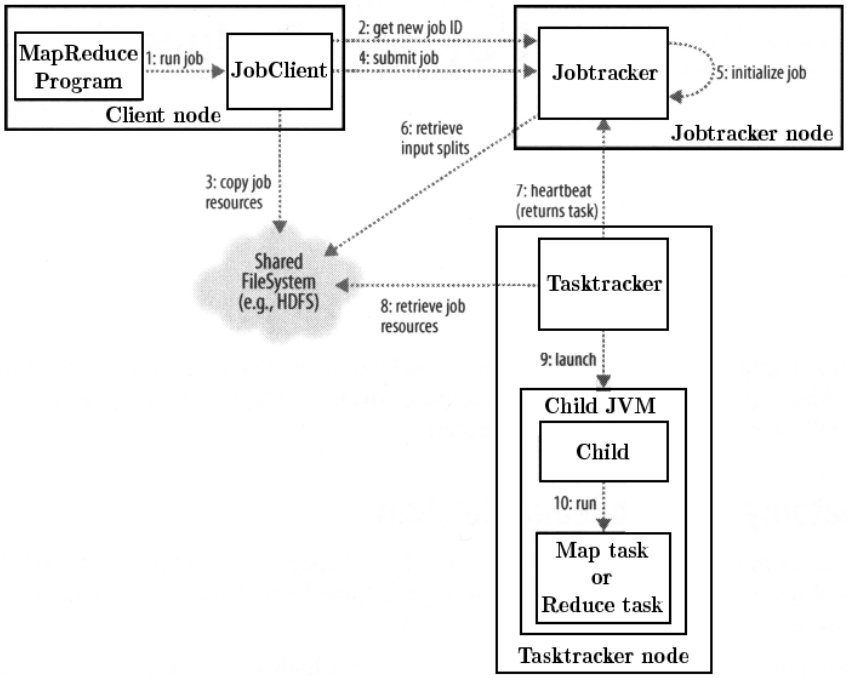
1.2.2.1 Architectuur

In essentie heeft het Hadoop Map/Reduce framework een master/slave architectuur. De master server wordt in Hadoop de *Jobtracker* genoemd, en de slave servers komen overeen met *Tasktrackers*. Er is één Tasktracker per node in de cluster. Een Tasktracker is eigenlijk een algemene naam voor de nodes die we Mapper of Reducer genoemd hebben in paragraaf 1.2.1. De Jobtracker houdt zoals de naam doet vermoeden een lijst van jobs bij die door de gebruiker ingediend werden. De Jobtracker zorgt tevens voor de toekenning van map- en reduce taken aan de Tasktrackers. De Tasktrackers voeren op hun beurt de taken uit die de Jobtracker hen toekent, en zorgen tevens voor de data transfer tussen de map en reduce fases. Figuur 1.6 toont hier een overzicht van. Merk op dat in Hadoop een node zowel een compute- als een Datanode is. Naast een Tasktracker zal een slave node dus ook nog een *Datanode* bevatten. De

Jobtracker, alsook de *Namenode*, worden meestal alleen op een node uitgevoerd omwille van performantie redenen (zie paragraaf 1.3.1).

1.2.2.2 Execution overview

Nu we bekend zijn met de structuur van Map/Reduce kunnen we een overzicht geven van de uitvoering van een Map/Reduce job. Hierbij maken we gebruik van Figuur 1.6.



Figuur 1.6: Overzicht van de architectuur in Hadoop Map/Reduce.

Job submission In de eerste stap voert de client de job uit door de submit() functie van Job aan te roepen (stap 1). Dit zorgt achterliggend voor heel wat handelingen:

1. Er wordt aan de jobtracker een nieuw job ID gevraagd (stap 2).
2. Er wordt nagegaan of de output directory gespecificeerd is en of deze niet reeds bestaat.
3. De input splits voor de job worden berekend.

4. De job JAR file, configuratie file en de berekende input splits worden opgeslagen in het file systeem van de Jobtracker.

5. Er woren replicas van de JAR file verspreid over de cluster (stap 3). Het aantal hiervan wordt bepaalt door de *mapred.submit.replication* property, die per default op 10 is ingesteld. Deze kopiën zijn nodig om snelle toegang tot deze JAR file te verzekeren wanneer de Tasktrackers map of reduce taken uitvoeren voor deze job.

6. De Jobtracker wordt verwittigd dat de job klaar is voor uitvoering (stap 4).

Nadat deze handelingen zijn uitgevoerd zal de client via de waitForCompletion() functie periodiek updates ontvangen over de vooruitgang van de job, en deze mogelijk naar de console outputten. Wanneer de job succesvol is uitgevoerd wordt dit op de console getoond samen met de *job counters*. Deze laatste worden bijgehouden door de jobtracker en houden job-level statistieken bij. Interessant om te weten is dat ze niet over het netwerk verstuurd worden, en dus niet voor extra bandbreedte zorgen. Enkele voorbeelden van job counters zijn het aantal gestarte map of reduce taken, het aantal gefaalde map of reduce taken, het aantal data- of rack-lokale map of reduce taken, etc.

Job initialisatie Zodra de client de Jobtracker heeft verwittigd dat de job klaar is voor uitvoering zal deze worden toegevoegd aan de queue van jobs. De scheduler zal de job vervolgens initialiseren (stap 5). Hierbij wordt er informatie zoals de status en de vooruitgang van de taken bijgehouden. Het betreft een vrij simpele First In, First Out scheduler, wat kan leiden tot lange wachttijden. Het voordeel is dat deze scheduler weinig overhead vergt, en er tevens geen starvation mogelijk is. Er zijn ook twee andere schedulers beschikbaar in Hadoop, namelijk de *fair scheduler* en de *capacity scheduler* (zie paragraaf 1.2.2.6). De scheduler zal ook de de lijst van taken opstellen waaruit de job bestaat. Hieronder vallen de map en reduce taken, maar ook een job setup en een job cleanup taak. Deze laatste zullen onder andere zorgen voor het aanmkaken van de output directory van de job, het reserveren van work space voor tijdelijke output, alsook het terug vrijgeven van deze workspace. Voor de lijst van map en reduce taken aan te maken dient de scheduler eerst de input splits die berekend zijn door de client op te halen uit het gedistribueerde file systeem (stap 6). Per input split wordt er één map taak gecreëerd. Het aantal reduce taken wordt bepaald door de mapred.reduce.tasks property in het Job object en kan worden ingesteld door de setNumReduceTaks() functie.

Taak toekenningen Nu de job is geïnitialiseerd is het tijd om de taken toe te kennen aan de Tasktrackers. Een tasktracker heeft een vast aantal map en reduce taken dat hij op een gegeven moment toegekend kan krijgen. Dit aantal verschilt van node tot node en is afhankelijk van het aantal cores en de hoeveelheid beschikbaar geheugen. Tasktrackers laten aan de jobtracker, via

hetzelfde kanaal waarmee heartbeat messages verstuurd worden, weten wanneer ze klaar zijn om een nieuwe taak uit te voeren. Bij het toekennen van een map taak houdt de jobtracker rekening met de netwerk locatie van de tasktracker, en wordt er een taak gekozen zodat de input split zo dicht mogelijk bij de tasktracker ligt. Hierbij gaat de voorkeur dus uit naar data-lokale taken, gevolgd door rack-lokale taken, en tot slot taken met off-rack data.

Taken uitvoeren Zodra een Tasktracker een taak heeft om uit te voeren moet hij de JAR-file van de job kopiëren indien hij hier nog niet over beschikt (stap 8). Indien het een map taak betreft wordt de input split omgezet in key/value paren, waarvoor eventueel remote reads voor worden gebruikt. Vervolgens start de tasktracker een Java Virtual Machine start voor elke map of reduce taak, om te vermijden dat eventuele bugs geen invloed hebben op de Tasktracker (stap 9 en 10).

Map taken Elke map of reduce taak stuurt ook periodiek berichten naar de Tasktracker om vooruitgang aan te geven. De uitvoer van een map functie wordt gesorteerd en naar lokale disk geschreven. Dit is omdat in HDFS één of meerdere kopiën van data worden bijgehouden, en dit uiteraard niet gewenst is voor tussenliggende key/value paren. Dit zou gewoon een verspilling zijn van bandbreedte en opslagplaats. De locaties van deze tussenliggende key/value paren worden doorgegeven naar de jobtracker (stap 7), dewelke ze op zijn beurt doorgeeft aan de reducers.

Reduce taken De reducers halen op hun beurt de data dan op via Remote Procedure Calls. Vervolgens sorteren ze de tussenliggende key/value paren en worden de paren met dezelfde key gegroepeerd. Dit is nodig omdat typisch veel verschillende keys gemapt worden op één reduce taak (door de partitioner). Voor elke unieke key wordt de set van overeenkomstige values doorgegeven aan de reduce functie. De output van de reduce functie wordt wel in HDFS opgeslagen. Hierbij staat een kopie op de huidige node opgeslagen, en de kopiën op off-rack nodes.

Vooruitgang Aangezien Map/Reduce jobs vaak lang kunnen duren is het belangrijk dat de gebruiker de vooruitgang van een job kan monitoren. Daarom heeft elke job en elke taak een *status*, die informatie kan bevatten zoals de staat van de job of taak (running, completed of failed), de waardes van de job counters, etc. Een map of reduce taak geeft tevens aan hoeveel werk reeds voltooid is. Voor een map taak is dit de hoeveelheid input key/value paren die reeds verwerkt zijn ten opzichte van de totale hoeveelheid input key/value paren. De vooruitgang van een reduce taak wordt opgedeeld in drie delen: het ophalen van de te verwerken tussenliggende key/value paren, het sorteren van deze paren, en het toepassen van de reduce functie op deze paren. Stel bijvoorbeeld dat de helft van de output key/value paren gegenereerd zijn, dan is de vooruitgang van deze reduce taak $\frac{5}{6}$ ($\frac{1}{3} + \frac{1}{3} + \frac{1}{2}$). Status updates van map

of reduce taken worden door de tasktracker doorgegeven aan de jobtracker door middel van heartbeats, die per default elke vijf seconden worden verstuurd. De jobtracker combineert deze updates dan om een globaal beeld te creëren. Een client kan tot slot al deze informatie opvragen met behulp van de *getStatus()* functie van Job.

Jobs afsluiten Wanneer de laatste taak van een job is uitgevoerd wordt zijn status naar succesvol veranderd, wordt er een bericht gestuurd naar de client dat de job voltooid is, en wordt teven de *waitForCompletion()* methode gereturned. Vervolgens worden alle job counters en statistieken uitgeprint. Tot slot gebeurt er nog een cleanup bij de job- en tasktrackers waarbij bijvoorbeeld tussenliggende key/value paren verwijderd worden.

1.2.2.3 Fault tolerance

Eén van de voornaamste kenmerken van Map/Reduce is dat het fouten zoals het crashen van processen of het falen van hardware (meestal) kan afhandelen zonder dat de volledige job moet worden heruitgevoerd. Er zijn verschillende niveaus van fouten te onderscheiden, namelijk het falen van een map of reduce taak, het falen van een tasktracker, en tot slot het falen van de Jobtracker. Deze laatste is de meest ernstige fout die kan optreden aangezien hier niet van hersteld kan worden, en dus de job volledig opnieuw zal moeten worden uitgevoerd. De kans dat dit gebeurt is relatief klein, maar toch is een single-point-of-failure zelden gewenst. Vandaar dat men deze situatie heeft trachten te verhelpen in Yarn (zie paragraaf 1.2.3.1).

Falen van een Tasktracker Het falen van een Tasktracker is niet zo ernstig en treedt op wanneer de Jobtracker gedurende tien minuten geen heartbeats meer heeft ontvangen. Dit interval kan aangepast worden met behulp van de *mapred.Tasktracker.expiry.interval* property. De Jobtracker zal vervolgens de Tasktracker verwijderen uit zijn lijst van beschikbare Tasktrackers. Alle map taken uitgevoerd door deze Tasktracker (voor niet voltooide jobs uiteraard) moeten worden heruitgevoerd. Immers, de door map taken gegenereerde tussenliggende key/value paren worden op de lokale disk bewaard, en zijn dus mogelijk niet beschikbaar voor reducers. Reduce taken moeten enkel opnieuw worden uitgevoerd indien ze nog niet voltooid waren. De output van reduce taken wordt immers in het gedistribueerde file systeem bewaard, waarbij er meerdere backups worden voorzien. De output is dus nog steeds beschikbaar indien de gefaalde Tasktracker onbereikbaar blijft.

Blacklisted Tasktrackers Het is ook mogelijk dat de Jobtracker een Tasktracker *blacklist*, en dus geen taken meer zal toekennen. Dit gebeurt wanneer een Tasktracker meer gefaalde map of reduce taken heeft dan een bepaalde threshold (bepaald door de *mapred.max.tracker.blacklists property*, en deze foutwaarde tevens (veel) hoger is dan de foutwaarde van de gemiddelde Tasktracker.

Dit kan bijvoorbeeld wijzen op hardware problemen bij een bepaalde Tasktracker. Na verloop van tijd neemt de foutwaarde automatisch af (met één per dag), en zal de Jobtracker uiteindelijk weer taken toekennen aan de Tasktracker.

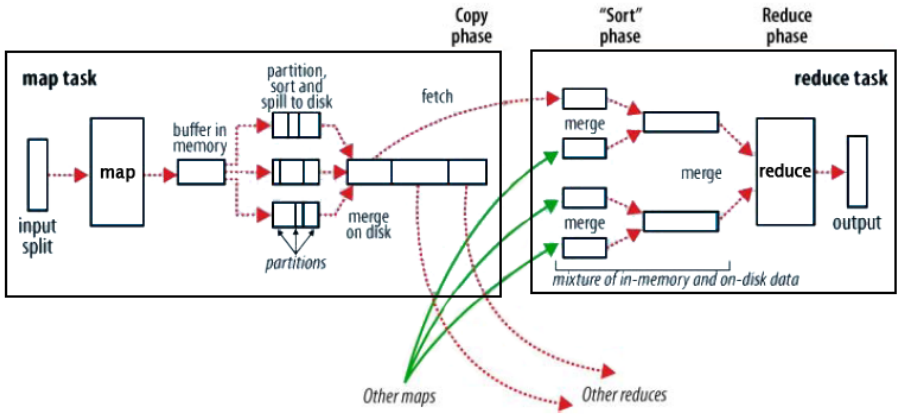
Falen van een map of reduce taak Tot slot beschouwen we nog het falen van een map of reduce taak. De meest voorkomende reden hiervoor is een bug in de user code. Wanneer dit gebeurt wordt er een error naar de log files geschreven en zal de Tasktracker de taak als failed beschouwen. De Jobtracker zal vervolgens de taak opnieuw toekennen aan een Tasktracker, waarbij er getracht wordt de Tasktracker waarop de taak gefaald is te vermijden. Indien na vier pogingen (instelbaar door de *mapred.map.max.attempts* en *mapred.reduce.max.attempts* properties) de taak nog steeds niet succesvol werd uitgevoerd stopt de Jobtracker met het toekenen van de taak, wat leidt tot het falen van de volledige job. Indien de gebruiker ondanks het falen van één of meerdere taken de job toch niet wil afbreken kan dit door gebruik te maken van de *mapred.max.map.failures.percent* en *mapred.max.reduce.failures.percent* properties. Deze geven respectievelijk voor map en reduce taken het getollereerde percentage gefaalde taken aan.

Map en reduce taken worden ook als gefaald beschouwd indien ze gedurende geruime tijd geen vooruitgang boeken (zie paragraaf 1.2.1.6 voor het rapporteren van vooruitgang). De threshold periode hiervoor wordt bepaald door de *mapred.task.timeout* property en is standaard ingesteld op tien minuten.

Naast vasthangen of crashen kunnen map en reduce taken ook beëindigd worden indien het een speculatieve uitvoering betreft (zie paragraaf 1.2.2.5), of omdat de Jobtracker de Tasktracker waarop de taak werd uitgevoerd als gefaald beschouwd.

1.2.2.4 Shuffle en sort

Met de *shuffle en sort* fase worden de stappen bedoelt die gebeuren om de output van de map functies bij de reducers te krijgen. Aangezien hier de enige echte communicatie stap gebeurt die in het Map/Reduce model plaatsvindt is het van groot belang deze fase in detail te bekijken. Hiervoor maken we gebruik van Figuur 1.7



Figuur 1.7: De shuffle en sort fase in Map/Reduce.

Bij de Mapper Zoals je kan zien in Figuur 1.7 wordt de output van de map functies niet simpelweg naar disk geschreven. Eerst worden de tussenliggende key/value paren gepartitioneerd (door de partitioner). Vervolgens worden de tussenliggende key/value paren binnen één partitie gesorteerd volgens key. Indien er een combiner functie gespecificeerd is wordt deze nu uitgevoerd. De output wordt bijgehouden in een circulaire memory buffer die per default 100MB groot is. Wanneer een bepaalde threshold bereikt is (per default 80%), zal de output naar disk beginnen *spillen*. Telkens de threshold van de buffer bereikt is zal er een nieuwe spill file worden aangemaakt, waardoor het mogelijk is dat na uitvoer van de map taak er verschillende spill files zijn. Wanneer er meer dan drie zulke spill files zijn zal de combiner (indien deze gespecificeerd is) nogmaals worden uitgevoerd. Deze worden allemaal gemerged tot één gepartitioneerde en gesorteerde output file.

Het is mogelijk om de output van de mapper te comprimeren alvorens ze naar disk te schrijven. Dit bespaart ruimte op de disk, zorgt voor snellere read en writes van en naar disk, en reduceert tevens de hoeveelheid bandbreedte die gebruikt wordt bij data transfers naar reducers. Per default staat compressie af, maar dit is gemakkelijk aan te passen met behulp van de *mapred.compress.map.output* en *mapred.map.output.compression.codec* properties. Uiteraard kost het comprimeren en decomprimeren van data tijd en CPU cycles.

Nadat een map taak succesvol is uitgevoerd zal zijn Tasktracker dit melden aan de Jobtracker via het heartbeat mechanisme. Reducers zullen op hun beurt aan de Jobtracker vragen op welke hosts zich map outputs bevinden. Dit blijven ze periodiek doen tot ze de locatie van alle mappers hebben ontvangen. Merk op dat mappers hun map output niet verwijderen van zodra de reducers deze data hebben opgehaald. Aangezien reducers kunnen falen wordt deze data bijgehouden tot de job volledig is uitgevoerd.

Bij de Reducer Zoals reeds vermeld vraagt een reducer aan de Jobtracker waar de reeds voltooide map outputs zich bevinden. Wanneer hij zulk een locatie doorkrijgt zal de reducer de output van zijn partitie kopiëren. Voor deze *copy* fase zijn er per default vijf threads beschikbaar, maar dit aantal kan worden aangepast met behulp van de *mapred.reduce.parallel.copies* property. Indien de map output klein genoeg is om in het geheugen van de reducer te passen zullen de outputs van verschillende mappers hier gemerged worden. Indien nodig worden ze naar disk geschreven en vervolgens gemerged met de andere map outputs. Het resultaat van deze *sort phase* (een meer toepasselijke naam zou de *merge phase* zijn) is één of meerdere gesorteerde bestanden. De inhoud van deze bestanden worden vervolgens in gesorteerde volgorde aan de reduce functie doorgegeven. De output van deze reduce functies wordt rechtstreeks naar het gedistribueerde file systeem geschreven.

Performantie Het is duidelijk dat de shuffle en sort fase er baat bij heeft zo veel mogelijk geheugen ter beschikking te hebben. Bij de mapper kan dit er voor zorgen dat het sorteren volledig in het geheugen kan gebeuren, en er maar één keer naar de disk geschreven moet worden. Aan de kant van de reducer kan dit ervoor zorgen dat de output van de verschillende mappers volledig in het geheugen gemerged kan worden. De map en reduce functies moeten echter ook genoeg geheugen krijgen om uitgevoerd te kunnen worden. Bijgevolg is het van belang om deze functies te programmeren zodanig dat ze zo weinig mogelijk geheugen gebruiken. Het geheugen dat gebruikt wordt door de mapper om de outputs te sorteren kan worden ingesteld door de *io.sort.mb* property, waarbij de grootte in megabytes wordt aangegeven. Bij de reducer wordt per default quasi al het geheugen gereserveerd voor de reduce functie. Dit kan worden aangepast door de *mapred.job.reduce.input.buffer.percent* property, die aangeeft welk percentage van het geheugen gebruikt wordt voor de sort fase van de reducer[5].

1.2.2.5 Speculative execution

Wanneer een job bestaat uit honderden of duizenden map of reduce taken is de kans reëel dat de uitvoertijd van sommige taken aanzienlijk langer is dan verwacht, wat kan leiden tot een langere uitvoertijd van de job. Wanneer een node uitzonderlijk veel tijd nodig heeft voor het uitvoeren van een map of reduce taak spreken we van een *straggler*. Dit kan bijvoorbeeld gebeuren door slecht werkende hardware, of het overbelasten van een node waardoor verschillende taken CPU tijd, geheugen en bandbreedte moeten delen. Dit is vooral problematisch wanneer een van de laatste taken wordt uitgevoerd op een straggler. Hadoop heeft een ingebouwd mechanisme voor het omgaan met dit type problemen. Wanneer de Jobtracker merkt dat de uitvoeringstijd van een taak hoger is dan verwacht zal deze een equivalente taak starten op een andere Tasktracker. Dit noemt men speculatieve uitvoering van taken. Wanneer één van de taken voltooid is zullen de andere taken beëindigd worden.

Hadoop zal taken niet enkel speculatief uitvoeren wanneer ze langer duren dan verwacht. Wanneer een job bijna volledig is uitgevoerd zullen de laatste taken ook meermaals worden opgestart. Dit zorgt slechts voor een kleine procentuele verhoging van de totale resources die de job gebruikt heeft, terwijl het de uitvoertijd met wel 44% kan verminderen[22].

Speculatieve uitvoering staat per default aan, maar kan worden uitgeschakeld door gebruik te maken van de *mapred.map.tasks.speculative.execution* en *mapred.reduce.tasks.speculative.execution* properties. Een mogelijke reden voor het uitschakelen van speculatieve uitvoering is een zeer druk gebruikte cluster. De totale throughput van de cluster kan immers verminderen wanneer we de uitvoeringstijd van één job willen verminderen door de uitvoering van redundante taken. Merk op dat redundante reduce taken dezelfde tussenliggende key/value paren verwerken, en dus zorgen voor een verhoogde netwerk trafiek in de cluster. Dit zou een mogelijke reden kunnen zijn om enkel de speculatieve uitvoering van reduce taken uit te schakelen.

1.2.2.6 Scheduling

Zoals reeds vermeld gebruikt Hadoop per default een First In, First Out scheduler. Via de *mapred.job.priority* property is het mogelijk een prioriteit toe te kennen aan jobs. De scheduler zal bij het kiezen van een job steeds deze met de hoogste prioriteitswaarde selecteren. Merk op dat bij deze methode een job met een erg hoge prioriteit nog steeds zal moeten wachten tot een lange job met een erg lage prioriteit voltooid is, indien deze job reeds gestart was. Uiteraard is het wenselijk om over meer controle te beschikken indien de cluster door meerdere gebruikers gedeeld wordt. Hadoop voorziet hiervoor nog twee andere schedulers: de *Fair Scheduler* en de *Capacity Scheduler*.

De Fair Scheduler Terwijl de FIFO scheduler vooral geschikt is voor single-user systemen, is de Fair Scheduler ontworpen voor multi-user systemen. Elke gebruiker heeft een pool van jobs. Per default heeft elke gebruiker evenveel resources ter beschikking, ongeacht het aantal jobs in zijn pool. Het is echter mogelijk om een bepaald gewicht aan een pool toe te kennen, waardoor deze meer of minder resources ter beschikking heeft dan andere gebruikers. Uiteraard is het zo dat wanneer een gebruiker geen jobs gesubmit heeft, er ook geen resources voor hem gereserveerd zijn. De Fair Scheduler is trouwens in staat om map of reduce taken te beëindigen van een gebruiker die meer resources gebruikt dan zijn fair share. Zo worden resources vrij gemaakt voor gebruikers die reeds gedurende een bepaalde tijd minder resources toegekend hebben gekregen dan het gewicht van hun pool hen toelaat.

De Capacity Scheduler Ook de Capacity Scheduler is geschikt voor multi-user systemen. Hierbij wordt de cluster verdeeld in een aantal queues, waarbij elke queue een bepaald deel van de capaciteit van de cluster krijgt toegewezen. Per queue wordt er een FIFO scheduler gebruikt. Ook hier is het mogelijk om aan jobs binnen een queue prioriteiten toe te kennen. Uiteraard kan een queue

resources “lenen” van een andere queue wanneer deze leeg is. Het verschil met de Fair Scheduler is dus vooral dat in een queue er een FIFO mechanisme gehanteerd wordt, terwijl jobs uit eenzelfde pool de resources van die pool delen.

1.2.3 Verschillende implementaties

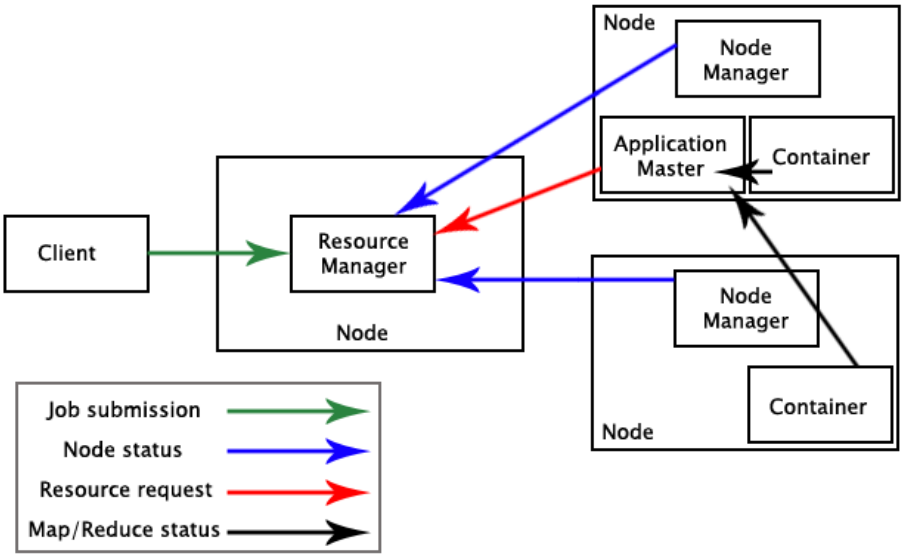
Naast de versie van Map/Reduce die we in de voorgaande paragrafen besproken hebben zijn er nog andere implementaties beschikbaar. We zullen er hiervan twee kort bespreken en vergelijken met de originele implementatie.

1.2.3.1 YARN

De eerste alternatieve implementatie die we gaan bekijken is Hadoop's Map/Reduce 2.0, ook wel YARN (Yet Another Resource Negotiator) genoemd[2]. Het probleem met Map/Reduce 1.x is dat de Jobtracker een bottleneck wordt bij zeer grote clusters (bestaande uit enkele duizenden nodes). De oplossing die YARN hiervoor hanteert is het opsplitsen van de verantwoordelijkheden van de Jobtracker. Immers, de Jobtracker kent taken toe aan Tasktrackers (scheduling), maar doet evenzeer aan het monitoren van de vooruitgang van taken en het eventueel herstarten of beëindigen van deze taken. In YARN worden deze verantwoordelijkheden verdeeld onder de *Resource Manager* en de *application master*. Voor elke Map/Reduce job is er een application master die draait gedurende de volledige uitvoertijd van de job. Deze onderhandelt met de Resource Manager voor cluster resources, die worden toegekend onder de vorm van een aantal *containers*. Elk van deze containers voert taken voor één bepaalde job uit, en heeft hiervoor een bepaalde hoeveelheid geheugen ter beschikking. Per node in de cluster is er ook een *node manager* die ervoor zorgt dat de containers niet meer dan de aan hen toegewezen hoeveelheid geheugen gebruiken. Een overzicht hiervan wordt getoond in Figuur 1.8.

Fault tolerance in YARN Herinner je dat Map/Reduce 1.x niet kan herstellen van het falen van de Jobtracker. Aangezien de functionaliteit van de Jobtracker in YARN verdeeld werd zijn er nu meer entiteiten die kunnen falen. Het falen van een map of reduce taak wordt analoog afgehandeld als in Map/Reduce 1.x. Het zelfde geldt voor het falen van een node manager (deze verschilt immers amper van een Tasktracker). Verder kan ook de Application Master falen. Indien dit gebeurt zal YARN de applicatie opnieuw starten, tot een bepaalde threshold bereikt is die bepaald wordt door de *yarn.resourcemanager.am.max-retries* property. De Resource Manager zal het falen van een Application Master ontdekken doordat er geen heartbeat messages meer verstuurd worden. Deze zal een nieuwe Application Master starten, en indien de property *yarn.app.mapreduce.am.job.recovery.enable* op *true* ingesteld staat zullen de reeds succesvol uitgevoerde taken niet opnieuw uitgevoerd worden.

Naast de Application Master kan ook de Resource Manager falen. Zonder de Resource Manager kunnen er geen nieuwe jobs starten, of containers worden gealloceerd. Dit is met andere woorden een zeer ernstige fout, en zal tot het gevolg



Figuur 1.8: Overzicht van de architectuur in YARN.

hebben dat alle niet voltooide jobs opnieuw moeten worden uitgevoerd, net zoals bij het falen van de Jobtracker. Hoewel dit nog niet volledig geïmplementeerd is zijn er wel plannen om een nieuwe instantie van een Resource Manager te laten opstarten via een checkpointing mechanisme, waardoor jobs niet volledig heruitgevoerd zullen moeten worden.

Voordelen van YARN Het is in ieder geval duidelijk dat het opsplitsen van de functionaliteit van de Jobtracker meerdere voordelen biedt. Zo kan er bijvoorbeeld van meerdere soorten fouten hersteld worden en wordt één bepaalde node (de Jobtracker uit Map/Reduce 1.x) minder belast. Bovendien biedt YARN ook meer algemene mogelijkheden dan Map/Reduce 1.x, waaronder bijvoorbeeld het uitvoeren van een MPI applicatie, of het uitvoeren van een gedistribueerde shell dat een script uitvoert op een set van nodes. Uiteraard vergemakkelijkt dit het managen en tevens gebruiken van de cluster. Een ander voordeel is dat YARN toelaat om meerdere versies van Hadoop op één cluster te draaien, wat niet mogelijk was met Map/Reduce 1.x. Een nadeel van YARN is dat Resource Manager een single point of failure blijft, alleszins tot het checkpointing mechanisme geïmplementeerd is.

1.2.3.2 Corona

Tot 2011 werd MapReduce van Hadoop gebruikt bij Facebook als basis voor hun infrastructuur. Tegenwoordig bevat hun grootste cluster meer dan 100 PB aan data, maken dagelijks 1000 werknemers gebruik van de clusters, waarbij er

allerlei soorten jobs worden uitgevoerd. Bovendien groeit hun data warehouse aan een sneltempo met een anderhalve petabyte per dag. Doordat de job tracker verantwoordelijk is voor zowel het managen van cluster resources, als het schedulen van alle jobs, wordt de job tracker een heuse bottleneck bij zeer grote clusters zoals deze van Facebook. Vandaar dat ze hun eigen scheduling framework ontworpen hebben, genaamd Corona[16]. Het doel van dit framework is betere schaalbaarheid, een verbeterd gebruik van de cluster en tevens de toekenning van resources op basis van de eigenlijke resource requirements van een taak, in plaats van het aantal map en reduce taken van een job. Hun manier van werken is analoog aan die in YARN, ze splitsen namelijk de taken van de job tracker op. De *Cluster Manager* houdt de nodes in de cluster bij en kent er resources aan toe. Per job wordt er een *Job Tracker* aangemaakt. Deze kan ofwel op de client worden uitgevoerd, ofwel op een node in de cluster. Door deze scheiding van verantwoordelijkheden kunnen er sneller scheduling beslissingen gemaakt worden, kunnen er meer jobs gemanaged worden en is de code van de Job Trackers minder complex. Zoals je al wel gemerkt hebt zijn er weinig verschillen tussen YARN en Corona. De reden dat Facebook niet gewoon YARN gebruikt heeft is dat dit niet compatibel is met hun aangepaste versie van HDFS, en het minder risicovol leek om een eigen scheduling framework te ontwerpen dan het oplossen van deze incompatibiliteiten.

1.3 HDFS

De voor- en nadelen van Hadoop zijn inherent verbonden met het achterliggende gedistribueerde filesysteem HDFS, en dus is het van belang dat we ook hier de algemene concepten van bespreken. Hierbij denken we onder meer aan de high availability van data in HDFS, alsook het omgaan met node failures zonder het verlies van data. Hoewel HDFS staat voor Hadoop Distributed Filesystem is dit niet het enige file systeem dat kan samenwerken met de Map/Reduce component van Hadoop. Andere opties zijn bijvoorbeeld het lokale file systeem, of Amazon S3.

HDFS is speciaal ontworpen voor de opslag van zeer grote bestanden, gaande van enkele honderden megabytes of gigabytes tot zelfs enkele terabytes.

Het feit dat HDFS ontworpen is voor een hoge throughput van data maakt dat het minder geschikt is voor applicaties die een lage latancy vereisen voor de toegang tot data. Verder is HDFS ook minder geschikt voor (zeer) grote hoeveelheden kleine bestanden. Er is immers één *Namenode* die alle metadata van het file systeem in zijn geheugen bijhoudt, waardoor het aantal files gelimiteerd wordt door de grootte van dit geheugen. Typisch treden er echter pas problemen op van zodra de hoeveelheid files van de orde van grootte van enkele miljoenen is.

1.3.1 Concepten

Alvorens we bespreken hoe HDFS data integriteit en high availability garandeert introduceren we eerst enkele concepten waaruit HDFS is opgebouwd.

1.3.1.1 Blocks

Net zoals een single disk file systeem worden bestanden in HDFS ook in *blocks* opgedeeld, dewelke als onafhankelijke eenheden worden opgeslagen. De default blockgrootte van HDFS is echter 64 MB, in tegenstelling tot de typisch enkele kilobytes van een single disk file system. Merk op dat een bestand in HDFS dat kleiner is dan een HDFS block de volledige grootte van een block zal innemen. Hier kan je een mouw aan passen door gebruik te maken van *archives*, maar dit creëert een read-only kopie van de data, wat het probleem niet bepaald oplost.

Er zijn verschillende redenen voor het invoeren van een block abstractie level in een gedistribueerd file systeem. Vooreerst laat dit bestanden toe dewelke groter zijn dan één enkele disk uit de cluster, aangezien blocks op verschillende nodes opgeslagen kunnen worden. Bovendien heeft een HDFS block een vaste grootte, in tegenstelling tot een bestand. Dit laat toe om te berekenen hoeveel blocks er op een bepaalde disk opgeslagen kunnen worden. Daarnaast is een block simpelweg een stukje data, en kan metadata elders opgeslagen worden. Aangezien gedistribueerde file systemen behoorlijk complexe systemen zijn, zijn deze vereenvoudigingen uiterst welkom.

Je vraagt je misschien af waarom een HDFS block veel groter is dan een block uit een meer typisch file systeem. Dit werd gedaan om de kost van disk seeks zo klein mogelijk te houden in vergelijking met de totale tijd om data van de disk te lezen. Indien een block groot is, is de transfer tijd immers significant groter dan het zoeken van het begin van een block (rotatie tijd). Bijgevolg leunt de totale transfer tijd dichter aan bij de disk transfer rate. Merk op dat meer traditionele filesystemen dit proberen te bereiken door blocks die vaak gelijktijdig nodig zijn achter elkaar te plaatsen. Merk verder ook op dat je HDFS blocks best niet al te groot maakt. Aangezien een Mapper typisch één HDFS block als input gebruikt, zal je job langer duren dan met een kleinere block grootte indien dit er voor zorgt dat het aantal Mappers kleiner is dan het aantal (beschikbare) nodes in de cluster.

De reden waarom een Mapper typisch één HDFS block als input gebruikt is als volgt. Een HDFS block is de grootste input die gegarandeerd op één node opgeslagen is. Indien een Mapper twee of meer HDFS blocks als input zou gebruiken zouden mogelijk sommige input splits over het netwerk verstuurd moeten worden, wat uiteraard niet gewenst is. Met andere woorden, met de grootte van een input split die gelijk is aan de grootte van een HDFS block is de kans op data locality het grootst. Hierdoor herkennen we nog een reden waarom HDFS blocks zo groot zijn. Doordat een Mapper één HDFS block als input gebruikt, is het aantal Mappers gerelateerd aan het aantal blocks. Het aanmaken van een Mapper en het toekennen van blocks aan Mappers brengt een zekere overhead met zich mee. Je hebt er dus baat bij om het aantal blocken

klein te houden. Uiteraard moet je hier ook niet te ver in gaan, aangezien er genoeg blocken moeten zijn om zo veel mogelijk nodes in de cluster te gebruiken.

Herinner je dat data integriteit en high availability garanderen belangrijke doelstellingen zijn van HDFS. Ook hierbij sluit het block concept nauw aan. Door blocks op meerdere nodes op te slaan (per default drie) kan men in de meeste gevallen herstellen van corrupte data of disk failures. Immers, van zodra een block op een bepaalde locatie onbereikbaar wordt kan de data van een andere locatie gelezen worden, en zo een nieuwe kopie worden aangemaakt. Dit mechanisme zorgt tevens voor de high availability van data. Een bijkomend voordeel is dat berekeningen vaker op de node kunnen worden uitgevoerd die de data bevat, of op een node die zich op hetzelfde rack bevindt. De replicatie van data heeft echter ook een keerzijde; write operaties zijn complexer en duurder om uit te voeren.

1.3.1.2 De Namenode en Datanodes

Zoals reeds vermeld werd is de *Namenode* een soort master node, en worden de worker nodes *Datanodes* genoemd. De Namenode is verantwoordelijk voor het beheren van de file structuur en de bijhorende metadata. Dit gebeurt aan de hand van twee files: de *namespace image* en de *edit log*, dewelke op lokale disk worden opgeslagen. De Namenode weet ook op welke Datanodes de blocks van een bepaalde file zijn opgeslagen, hoewel dit niet op lokale disk wordt opgeslagen. Deze mapping wordt geconstrueerd door het opvragen van de lijst van blocks die een Datanode bevat wanneer de cluster wordt opgestart. Nadien wordt deze lijst up-to-date gehouden doordat Datanodes de aanpassingen die ze maken doorsturen naar de Datanode.

Namespace image en edit log Wanneer een write operatie wordt uitgevoerd wordt dit eerst opgeslagen in de edit log. Elke versie van de edit log wordt geflushed (dus naar persistente storage geschreven) alvorens de write operatie bij een Datanode als succesvol wordt beschouwd. Dit zorgt er voor dat er geen verschillende versies van een edit log bestaan na het falen van de Datanode. Deze laatste houdt trouwens ook een versie van de file structuur en metadata in zijn geheugen, dewelke ook geupdate wordt bij een write operatie. Dit is nuttig bij het snel afhandelen van read operaties.

De reden waarom er een edit log gebruikt wordt om operaties zoals het aanmaken en verplaatsen van een bestand bij te houden is als volgt. De namespace image kan tot enkele gigabytes groot worden, en deze naar persistente storage (zoals de lokale harde schijf) schrijven kost te veel tijd. Vandaar dat de namespace image slechts periodiek geupdate wordt (door de secundaire Namenode).

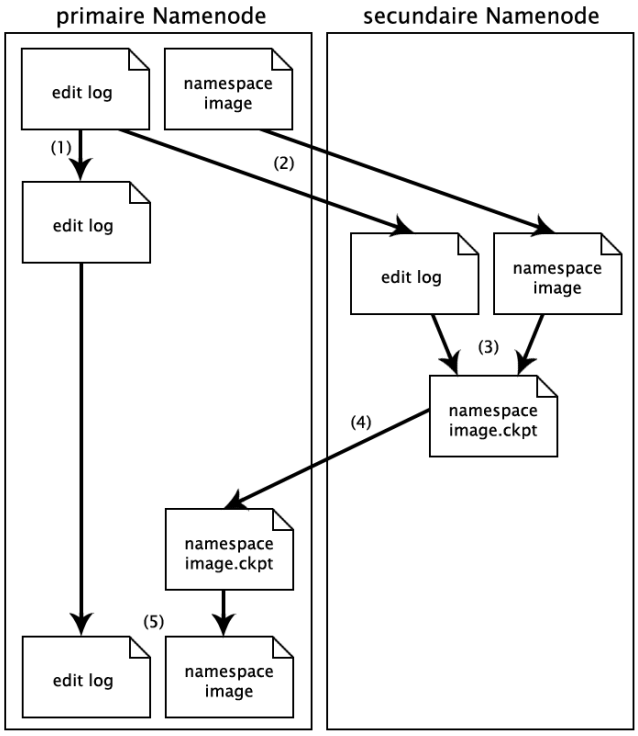
Een van de redenen voor het gebruik van een centrale node die de filestructuur van HDFS bijhoudt is nu ook duidelijk geworden. Indien er een Datanode faalt beschikt de Namenode over voldoende informatie om ervoor te zorgen dat data van de gefaalde Datanode gerepliceerd wordt naar andere Datanodes. Er zijn immers nog twee kopiën van de data beschikbaar, en de Namenode beschikt

over hun locatie. Op deze manier kan de replicatiefactor van data in HDFS terug hersteld worden.

Backups van de Namenode Het valt meteen op dat het erg belangrijk is om een backup van de Datanode te maken, aangezien het file systeem volledig onbruikbaar wordt indien de Datanode onbeschikbaar wordt. Dit kan op twee manieren gebeuren. Vooreerst kan de file structuur en de metadata zowel worden opgeslagen op de lokale disk als op een remote file systeem. Merk op dat updates naar beide opslaglocaties synchroon en atomisch gebeuren.

Secundaire Namenode De tweede manier om een backup van de file structuur en metadata bij te houden is door het gebruik van een *secundaire Namenode*. Zijn enige rol is het periodiek samenvoegen van de namespace image en de edit log. Aangezien dit behoorlijk CPU-intensief werk is gebeurt dit best niet op de primaire Namenode. De bekomen namespace image kan dan gebruikt worden indien de primaire Namenode onbeschikbaar wordt. Merk op dat de secundaire Namenode niet de volledige staat van de primaire Namenode bevat, waardoor het verlies van data zo goed als vast staat. Vandaar dat een secundaire Namenode best in combinatie met de eerst besproken backup techniek gebruikt wordt. Bij het falen van de primaire Namenode kan de secundaire Namenode de edit log van het remote file systeem gebruiken om een up-to-date file structuur te bekomen en de rol van primaire Namenode over te nemen.

Een bijkomend voordeel van het gebruik van een secundaire Namenode is trouwens dat deze er voor zorgt dat de grootte van de edit log binnen de perken blijft. Figuur 1.9 toont het checkpointing proces van de namespace image en de edit log dat hiervoor zorgt.



Figuur 1.9: Het checkpointing proces.

De stappen doorlopen worden tijdens het checkpointing proces zijn als volgt.

1. De primaire Namenode maakt een nieuwe edit log aan. Alle operarties die hierna plaats vinden worden naar de nieuwe edit log geschreven.
2. De primaire Namenode stuurt zijn namespace image en edit log door naar de secundaire Namenode.
3. De secundaire Namenode voegt de namespace image en edit log samen tot een nieuwe namespace image waarin alle operaties uit de edit log zijn doorgevoerd.
4. De secundaire Namenode bewaart de nieuwe namespace image op disk, en stuurt een kopie hiervan door naar de primaire Namenode.
5. De primaire Namenode ontvangt de nieuwe namespace image, en de oude namespace image en edit log worden vervangen door de nieuwe versies. (Indien van toepassing gebeurt dit zowel op de lokale disk van de primaire Namenode als op het remote file systeem.)

Het is duidelijk dat na dit checkpointing proces de edit log kleiner is, en het herstellen van het falen van de Namenode sneller kan gebeuren.

1.3.2 HDFS Federation

Aangezien de Namenode de file structuur en metedata in het geheugen bijhoudt om read requests snel af te handelen, stelt de grootte van het geheugen een limiet op het aantal files dat in HDFS geplaatst kunnen worden. Omdat het geheugen van één Namenode te klein kan zijn voor erg grote clusters is er sinds de 2.0 versie de mogelijkheid om meerdere Namenodes toe te voegen. Elk van deze Namenodes beheert dan een deel van het file systeem. Zo kan een Namenode bijvoorbeeld de files onder */user* beheren, en een andere die onder */system*. Merk op dat deze Namenodes niets te maken hebben met de secundaire Namenode die we zojuist besproken hebben. De file structuur en metadata die één Namenode bijhoudt noemen we een *namespace volume*, de mappings van de bijhorende blocks noemen we een *block pool*. Elk van de Datanodes communiceert met elk van de Namenodes en houdt blocks bij van meerdere block pools. Dit is logisch aangezien de input van een bepaalde Map/Reduce job zich vaak in één directory bevindt, en indien al deze blocks zich op één node zouden bevinden zou enkel deze node data lokale berekeningen kunnen uitvoeren. De namespace volumes zijn echter wel volledig onafhankelijk. Met andere woorden, de Namenodes communiceren niet onderling, en het falen van een bepaalde Namenode heeft geen invloed op de andere Namenodes. Tot slot merken we nog op dat een client een over een tabel met de mappings tussen Namenodes en bestandspaden moet beschikken om te interageren met een federated HDFS cluster.

1.3.3 High-availability

We hebben reeds besproken hoe we secundaire Namenodes kunnen gebruiken om te herstellen van het falen van een Namenode, en bovendien de ernst van dit falen verminderd door de introductie van HDFS federation. Een Namenode blijft echter een single point of failure in het opzicht dat wanneer deze onbeschikbaar wordt alle jobs met betrekking op zijn namespace volume geen read of write operaties meer kunnen uitvoeren. De Namenode is immers de enige die de mapping tussen bestanden en blocks bevat. Het starten van een nieuwe primaire Namenode kost tijd, aangezien deze eerst zijn namespace image in het geheugen moet laden, vervolgens het edit log moet uitvoeren, en tot slot van elke node uit de cluster zijn lijst met blocks moet ontvangen. Op een grote cluster kan dit proces 30 minuten of langer in beslag nemen.

Om deze situatie te verhelpen is er sinds de versie 2.0 een mechanisme ingevoerd dat HDFS *high-availability* genoemd wordt. Het komt er op neer dat er een tweede Namenode in standby staat om de taak van de primaire Namenode over te nemen zodra deze faalt. De tweede Namenode houdt tevens de file structuur en metadata in het geheugen, en update deze door gebruik te maken van de edit log op het remote file systeem. De Datanodes sturen hun block updates naar zowel de actieve als de standby Namenode, wat dus twee keer zoveel bandbreedte verbruikt. Indien de actieve Namenode zou falen kan de standby Datanode meteen zijn taken overnemen. Het enige dat nog dient te gebeuren is de clients hiervan op de hoogte brengen, wat gebeurd door de *failover control-*

ler. Deze zorgt er ook voor dat de gefaalde Namenode *gekilld* wordt. Het zou immers kunnen dat deze nog actief is, maar zeer traag of foutief opereert.

1.3.4 Read operatie

In deze paragrafen bekijken we in detail hoe een client (zoals een Map/Reduce job) een bestand inleest. De client contacteert eerst de Namenode om de locaties van de eerste paar blocks van de file te verkrijgen. Voor elk block krijgt hij de locaties van alle Datanodes die een kopie hebben van dit block. Deze locaties worden gesorteerd volgens hun *nabijheid* ten opzichte van de client. De meest nabije locatie is op de lokale disk van de client zelf, gevolgd door een node op hetzelfde rack van de cluster. Hierna volgen nodes op een verschillend rack in de cluster, en tot slot nog de nodes die zich in een ander data center bevinden. De client zal trachten de afstand tussen hemzelf en de data zo klein mogelijk te houden, en eerst de meest nabije Datanode vragen om het gewenste block. Op deze manier worden alle blocks één voor één, en in volgorde, over het netwerk verstuurd. Indien nodig zal de client de Namenode opnieuw contacteren voor de locaties van een volgende groep blocken uit het bestand.

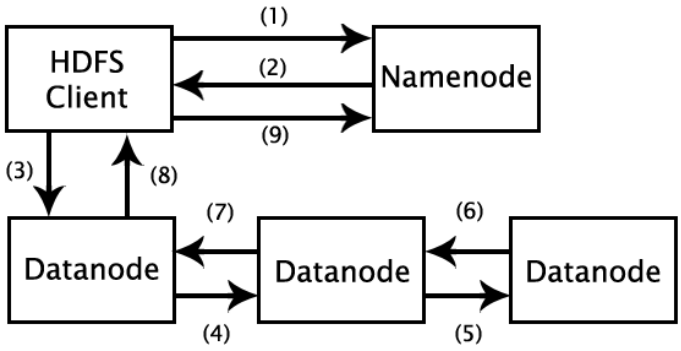
Wanneer de client merkt dat een Datanode onbereikbaar is of de checksum van een block foutief is zal hij dit block van de volgende Datanode uit de lijst trachten te verkrijgen. Bovendien onthoudt hij het adres van de falende Datanode zodat hij deze niet nodeloos tracht te contacteren bij het ophalen van een volgend block. Tot slot brengt de client nog de Namenode op de hoogte van dit falen.

Herinner je dat in een Map/Reduce job het de verantwoordelijkheid is van de Jobtracker om de locaties van de blocks te vragen aan de Namenode, en vervolgens op basis van deze locaties taken toe te kennen aan de Tasktrackers.

1.3.5 Write operatie

Naast de anatomie van een read operatie bekijken we ook de write operatie in detail. De stappen die plaatsvinden bij het schrijven van data naar een nieuwe file worden getoond in Figuur 1.10. Zoals je kan zien zal de client vooreerst de Namenode contacteren om een nieuw bestand aan te maken in zijn filesystem namespace (stap 1). Hierbij controlleert de Namenode of het bestand niet reeds bestaat, en of de client wel de juiste permissies heeft om dit bestand aan te maken. Indien deze voorwaarden voldaan zijn zal de client aan de Namenode vragen op welke Datanodes het eerste block uit het bestand opgeslagen moet worden. De Namenode selecteert hiervoor (per default) drie Datanodes en geeft hun locaties door aan de client (stap 2). Deze Datanodes vormen een pipeline waarbij ze telkens het block opslaan, en een ack packet sturen naar de vorige Datanode uit de pipeline (stappen 3 t.e.m. 8). De reden voor het vormen van een pipeline is duidelijk: dit voorkomt de vorming van een bottleneck bij de client. Dit proces herhaalt zich voor alle blocken uit het bestand. Merk op dat niet steeds dezelfde drie Datanodes geselecteerd zullen worden, aangezien dit nefast zou zijn voor de performantie van een Map/Reduce job. Je wilt immers

dat zo veel mogelijk taken worden uitgevoerd op de node waar de data zich bevindt, wat niet mogelijk is als alle data op slechts drie Datanodes geplaatst wordt. Tot slot geeft de client aan dat alle blocken van het bestand opgeslagen zijn (stap 9).



Figuur 1.10: De anatomie van een write operatie in HDFS.

Indien een Datanode faalt bij het schrijven van een block zal er ook geen ack packet van deze Datanode bij de client arriveren. Deze meldt het probleem aan de Namenode, die een nieuwe Datanode selecteert en in de pipeline plaatst.

1.3.5.1 Replica management

De enige vraag die ons nog rest is hoe de Namenode de Datanodes selecteert om replicas op te bewaren. Per default wordt de eerste replica op dezelfde node als de client geplaatst. (Indien de client geen node uit de cluster is wordt een willekeurige, weinig bezette node gekozen). De tweede replica wordt op een node uit een ander rack van de cluster geplaatst. De derde replica wordt tot slot op een node uit hetzelfde rack als de tweede node geplaatst. Deze strategie zorgt voor een goede betrouwbaarheid aangezien blocken op twee verschillende racks bewaard worden. Verder blijft het bandbreedte verbruik binnen de perken aangezien één replica lokaal geschreven wordt en er tevens maar één netwerk switch getraverseerd wordt. Tot slot is het ook positief voor de performantie van een read operatie om keuze te hebben uit twee verschillende racks.

1.3.6 Data integriteit

Voor elke I/O operatie naar of van disk, of over het netwerk bestaat de kans dat errors geïntroduceerd worden. Aangezien de volumes data waarmee Hadoop kan omgaan erg groot zijn, is de kans op dit soort fouten ook zeer reëel. Voor het opsporen van deze fouten gebeurd aan de hand van het alom bekende principe van checksums, wat hier niet verder besproken zal worden. Wat echter wel specifiek is voor HDFS zijn de replicas van blocks die het bijhoudt op andere

Datanodes. Deze kunnen gebruikt worden om een nieuwe replica van corrupte data te construeren. Wanneer een client een error in een block ontdekt zal deze dit melden aan de namdenode. De Namenode markeert vervolgens dit block en de bijhorende Datanode, en zal requests voor dit block naar één van de twee andere replicas verwijzen. Vervolgens wordt aan de hand van één van de resterende replicas een nieuwe replica aangemaakt op een Datanode verschillend van de Datanode met het corrupte block. Van zodra dit gebeurd is wordt het corrupte block verwijderd.

1.3.7 Compressie

Aangezien Hadoop met erg grote volumes van data werkt kan compressie een significante invloed hebben op de performantie. Compressie zorgt er immers voor dat data minder plaats zal innemen op de disk, maar ook minder bandbreedte zal verbruiken, wat uiteindelijk tot snellere job uitvoeringstijden leidt. Door middel van de *CompressionCodec* interface wordt er in Hadoop compressie ondersteund. Het is mogelijk om zelf een codec te implementeren, of gebruik te maken van een van de bestaande codecs (zie Tabel 1.3).

Compressie algoritme	bestandsextensie	splitting
DEFLATE	.deflate	nee
gzip	.gz	nee
bzip2	.bz2	ja
LZO	.lzo	nee
LZ4	.lz4	nee
Snappy	.snappy	nee

Tabel 1.3: Ondersteunde compressie codecs in Hadoop

Zoals je kan zien bevinden gzip en bzip2 zich onder de ondersteunde codecs. Beide laten een zeker level van controle toe over de time/space trade-off die ze maken: snellere compressie en decompressie leidt vaak tot een kleinere reductie van de data. Verder hebben verschillende compressie tools vaak ook erg verschillende karakteristieken. Zo is bzip2 trager dan gzip, maar is de compressiefactor ook groter. Verder is ook bzip2's decompressie snelheid groter dan zijn compressie snelheid. Het is duidelijk dat de keuze van de compressie tool veelal applicatie specifiek zal zijn.

1.3.7.1 Compressie in Map/Reduce

Er zijn verschillende fasen in een Map/Reduce job waar compressie aan te pas kan komen. Vooreerst kan de input voor een Map/Reduce job in gecompremeerde vorm opgeslagen zijn in HDFS. Map/Reduce zal automatisch de te gebruiken codec herkennen aan de hand van de file extenties. Wanneer de input

voor een Map/Reduce job gecomprimeerd is, is het van groot belang of het gebruikte compressie algoritme al dan niet *splitting* toelaat. Hiermee bedoelen we of het mogelijk is om vanaf een arbitrair punt in de gecomprimeerde stream te beginnen decomprimeren. Beschouw hiertoe het volgende voorbeeld. Stel er is een ongecrompirimeerd bestand van 1 GB opgeslagen in HDFS. Met een default block grootte van 64 MB zorgt dit ervoor dat dit bestand opgeslagen staat als 16 blocks (waarvan er replicas verspreid staan op de cluster). Een job met dit bestand als input zal dus 16 input splits aanmaken, waarbij elke input split door één mapper wordt verwerkt. Stel nu dat we over een gzip-gecomprimeerd bestand van 1 GB beschikken, dat opnieuw als 16 blocks opgeslagen werd in HDFS. In dit geval zal het niet mogelijk zijn om 16 input splits te creëren, aangezien er geen decompressie op een arbitrair punt in het bestand kan plaatsvinden. Bijgevolg moeten de 16 blocks gedecomprimeerd worden door één enkele Mapper, en zullen de meeste van de blocks niet op zijn lokale disk opgeslagen zijn. Dit kost dus zowel tijd als bandbreedte, en heeft een verminderde load balancing tot gevolg. In tegenstelling tot gzip laat bzip2 wel synchronisatiepunten toe tussen blocks, waardoor de blocks wel als onafhankelijke input splits verwerkt zouden kunnen worden.

Naast gecomprimeerde input bestanden kan uiteraard ook de output van een Map/Reduce job in gecomprimeerde vorm worden opgeslagen. Dit kan gespecificeerd worden door de *setCompressOutput* en *setOutputCompressorClass* functies van FileOutputFormat (in de main functie van de Map/Reduce klasse waar ook de job configuratie parameters worden ingesteld), zoals getoond wordt in onderstaand voorbeeld.

```
1 FileOutputFormat.setCompressOutput(job, true);
2 FileOutputFormat.setOutputCompressorClass(job, GzipCodec.
    class);
```

Tot slot bestaat er ook nog de mogelijkheid om de tussenliggende key/value paren te comprimeren. Deze worden naar disk geschreven en over het netwerk naar de reducers gestuurd, en dus kan compressie ook hier een tijdwinst opleveren. Hierbij maak je best gebruik van een snelle compressie tool, zoals LZO of Snappy. Dit kan door gebruik te maken van de *setCompressMapOutput* en *setMapOutputCompressorClass* functies van een *JobConf* object, zoals geïllustreerd wordt in onderstaande code.

```
1 conf.setCompressMapOutput(true);
2 conf.setMapOutputCompressorClass(GzipCodec.class);
```

1.3.8 Interactie met HDFS

We hebben reeds de interne werking van HDFS uitvoerig besproken in de vorige paragrafen. We sluiten de bespreking van HDFS af met een korte introductie tot interactie met HDFS via de command line interface. HDFS ondersteund al de typische operaties van een file systeem zoals het lezen van files, het aanmaken van directories, het verwijderen van data, etc. De belangrijkste operaties worden

getoond in Tabel 1.4. Je kan de volledige lijst van beschikbare operaties bekomen door het commando *hadoop fs -help* in te voeren.

Onderstaand voorbeeld toont nog kort hoe je een directory van het lokale file systeem naar HDFS kopiëert, en de inhoudt ervan oplijst.

```
1 hadoop fs -copyFromLocal home/tom/school/thesis/*.* hdfs
    ://localhost/user/tom/temp
2 hadoop fs -ls .
```

1.4 Pig

Eén van de grootste nadelen of moeilijkheden van Map/Reduce is dat het de gebruiker dwingt om het Map/Reduce programmeermodel te hanteren. Veel problemen kunnen moeilijk omgezet worden in dit programmeermodel, en vaak zijn er meerdere Map/Reduce jobs nodig om een bepaald probleem op te lossen. Dit geldt vooral voor programmeurs met weinig tot geen ervaring hiermee. Er kan geargumenteerd worden dat Map/Reduce vooral geschikt is voor ad-hoc analyse van grote data sets, wat ook zo is, maar zelfs binnen dit specifiek domein is het programmeer paradigma vaak te low-level en restrictief. Dit leidt op zijn beurt tot ingewikkelde code die moeilijk te onderhouden en hergebruiken is. Hiervoor tracht Pig een oplossing de bieden[20]. We zullen in deze paragrafen de voor- en nadelen van Pig, alsook de interne werking, bespreken en illustreren aan de hand van voorbeeld code. Het hoofddoel hierbij is inzien hoe Pig Map/Reduce kan aanvullen en verbeteren, niet het doorgronden van alle mogelijkheden en implementatie details van Pig.

1.4.1 Wat is Pig?

Pig bestaat uit twee delen: de taal *Pig Latin*, en het achterliggend framework dat Pig Latin programma's omzet in Map/Reduce jobs. Pig zorgt voor een level van abstractie bovenop Map/Reduce, door toevoeging van nieuwe datastructuren en (high-level) transformaties. Verder is het zo dat Pig Latin operaties de data flow van een programma beschrijven, wat ervoor zorgt dat de programmeur zich niet moet bezig houden met de manier van uitvoering binnen het Map/Reduce paradigma. Zo kan je bijvoorbeeld gemakkelijk een join operatie uitvoeren, wat niet bepaald triviaal is om te programmeren in Map/Reduce.

Om een beter beeld te krijgen van Pig beginnen we met een voorbeeldje. Onderstaande code toont hoe je het word count programma uit paragraaf 1.2.1.6 kan omzetten in een Pig Latin programma.

```
1 //De input wordt ingeladen en het enige veld in het
    record noemen we 'lijn'
2 input = LOAD 'input/wordcount.txt' as (lijn);
3 //TOKENIZE() splits de lijn op in woorden
```

```
4 //FLATTEN() neemt de bag van woorden gegeneerd door
    TOKENIZE() en produceert een record voor elk van de
    woorden.
5 words = FOREACH input GENERATE FLATTEN(TOKENIZE(line)) as
     word;
6 //Het resultaat wordt gegroepeerd per woord
7 wordGroups = GROUP words BY word;
8 //Voor elk woord wordt het aantal voorkomens geteld
9 wordCount = FOREACH wordGroups GENERATE COUNT(words) as
    count, group as word;
10 //Het resultaat wordt uitgeprint
11 DUMP wordCount;
```

Zoals je kan zien gelijken de transformaties uit dit voorbeeld op SQL operaties. Eigenlijk heeft het schrijven van een Pig Latin programma veel weg van het schrijven van een query uitvoeringsplan. Dit heeft als voordeel dat programmeurs makkelijker de manier waarop hun programma wordt uitgevoerd kunnen volgen en controleren dan bijvoorbeeld met SQL het geval is. Zeker voor doorwinterde systeem programmeurs is deze manier van werken handiger dan een taak in SQL programmeren, en vervolgens optimizer hints te geven om het meest geschikte query plan op te stellen. Langs de andere kant is Pig Latin toch high-level genoeg om je niet met elk detail van de uitvoering te moeten bezig houden zoals bijvoorbeeld in Map/Reduce.

1.4.2 Features

In de volgende paragrafen overlopen we de beschikbare features in Pig, alsook de motivatie erachter.

1.4.2.1 Beschrijving van de data flow

Pig Latin programma's beschrijven de data flow van een programma. Elk van de stappen van een Pig Latin programma specificeert één enkele high-level data transformatie. Dit is in tegenstelling tot SQL, waarbij een set van declaratieve constraints het eindresultaat bepaalt. Ervaren programmeurs geven vaak de voorkeur aan de aanpak van Pig, zeker wanneer het grote data sets betreft[20]. Merk op dat Pig Latin transformaties zoals GROUP en FILTER niet noodzakelijk in de volgorde waarin ze geschreven zijn worden uitgevoerd. Pig zal achterliggend het programma trachten te optimaliseren, en hierbij mogelijk de volgorde van statements aan te passen. Stel bijvoorbeeld dat we twee FILTER operaties willen uitvoeren. De ene betreft een User Defined Function die behoorlijk wat tijd vergt, de tweede gaat simpelweg na of een bepaald attribuut groter is dan een bepaalde waarde. Pig zal er voor zorgen dat de tweede filter eerst wordt uitgevoerd, aangezien dit de hoeveelheid data waar de tijdsrovende UDF op moet worden uitgevoerd verkleint. Indien je deze functionaliteit in een Map/Reduce programma geïmplementeerd zou hebben, zou de kans reëel geweest zijn dat je niet zo gemakkelijk deze twee filters zou kunnen omdraaien.

1.4.2.2 Snelle start en interoperabiliteit

Zoals je weet is het voor traditionele database systemen noodzakelijk om data eerst in te laden volgens een bepaald schema. Hiervoor zijn meerdere redenen. Vooreerst bieden database management systemen consistentie garanties voor transacties. Verder zijn ze in staat om efficiën tuples op te zoeken (via indexen), en eventueel aan te passen. Tot slot zijn de database schema's van belang voor het begrijpen van de data door bijvoorbeeld andere gebruikers. Pig is echter bedoelt voor read-only, en scan-centric workloads. Het is met andere woorden bedoelt voor de ad-hoc analyse van data in zijn geheel, of toch een groot deel ervan. Bijgevolg is er geen nood aan het garanderen van consistentie of geïndexeerde lookups. Bovendien is een veel voorkomende toepassing de analyse van tijdelijke datasets, waardoor het inladen van deze data in een DBMS onnodige overhead zou genereren. Dit zijn de redenen waarom schema's in Pig volledig optioneel zijn. Verder kan de input in eender werk formaat gespecificeerd zijn, zolang de gebruiker een geschikte functie voorziet om de data in te lezen. Hetzelfde geld voor de output in Pig. Dit zorgt voor een verhoogde interoperabiliteit tussen Pig en andere applicaties, die bijvoorbeeld data input data gegenereerd hebben, of de output data verder zullen verwerken.

1.4.2.3 Ondersteuning voor geneste data types

Data is meestal volgens de First Normal Form opgeslagen in een DBMS, wat inhoudt kolommen enkel atomische waarden bevat. Dit impliceert dat tabellen een 'platte' structuur hebben, zoals volgend voorbeeldje illustreert. De bedoeling is om per trefwoord de voorkomens in bepaalde bestanden weer te geven.

```
1 Tabel_trefwoorden: (trefwoord, bestand_id, positie)
```

Pig laat echter geneste data types toe, waardoor we gebruik zouden kunnen maken van map en bag structuren om de data als volgt voor te stellen. Per trefwoord houden we de volgende volgende data structuur bij[1]:

```
1 Map<bestand_id, Bag<posities>>
```

Dit heeft verschillende voordelen. Vooreerst is deze manier van data representatie natuurlijker dan genormalizeerde data. Verder neemt de data minder plaats in beslag. Immers, de eerste representatie zorgt ervoor dat de velden *trefwoord* en *bestand_id* net zo vaak opgeslagen worden als er voorkomens zijn in een bestand. Dit is uiteraard niet het geval bij de geneste representatie. Daarenboven is de input data voor Pig vaak in geneste vorm beschikbaar[20]. Bijgevolg zou het normalizeren van deze data voor onnodige overhead zorgen. Bovendien leidt het normalizeren van data vaak tot de creatie van meerdere tabellen, die bij het uitvoeren van berekeningen dan weer samengevoegd worden door middel van joins. Ook dit tracht Pig te vermijden. Tot slot vergemakkelijkt de ondersteuning van geneste datastructuren het schrijven van UDFs.

[1] Merk op dat we beter gebruik hadden gemaakt van een set, aangezien posities slechts één keer per bestand kunnen voorkomen. Pig bevat echter geen set type.

1.4.2.4 UDFs

Pig tracht zo veel mogelijk ondersteuning te bieden voor het schrijven van UDFs. Zo wordt geneste data toegelaten als input en output voor UDFs, en zijn alle operaties zoals gropuing, filtering en joining aanpasbaar. Stel dat we over een dataset beschikken die voldoet aan het volgende schema.

```
1 trefwoorden: [trefwoord: chararray, voorkomens: [
    bestand_id: chararray, posities: {positie: int}]]
```

Hierbij worden de [ ] gebruikt om een map structuur te modelleren, en de { } om een set voor te stellen. Onderstaand voorbeeld illustreert nu het gebruik van UDFs om geneste data als een veld van een tupel te genereren.

```
1 FOREACH trefwoorden GENERATE trefwoord, top3(voorkomens);
```

Hierbij is *top3()* een UDF die de drie bestanden terug geeft waarin het trefwoord het meest voorkomt. De output van deze operatie bevat dus een een veld met niet-atomaire waarden, wat in SQL niet mogelijk is.

1.4.2.5 Parallellisatie

De makers van Pig hebben ervoor gekozen om een relatief kleine set van types en operaties te ondersteunen die zich makkelijk lenen tot parallellisatie. Operaties die moeilijk te parallelliseren zijn, zoals gecorelleerde subqueries of inequality joins werden niet geïmplementeerd. We zullen de moeilijkheid van het parallelliseren van joins binnen Map/Reduce (en dus Pig) illusteren aan de hand van enkele voorbeelden.

Equality joins We bespreken eerst het geval van equality joins, dat dus ondersteund wordt door Pig. Stel dat we een tabel S en een tabel T hebben die we willen joinen volgens de conditie $S.A = T.A$. Dit valt gemakkelijk te implementeren door van het attribuut A een key te maken, en het volledige tupel als value te nemen. Dit zal er voor zorgen dat alle tupels met dezelfde waarde voor het join attribuut bij dezelfde reducer terecht komen. Deze kan dan alle tupels uit de relatie S joinen met de tupels uit relatie T (merk op dat de tussenliggende key/value paren dus ook informatie moeten bevatten die aangeeft uit welke relatie het tupel komt). Op deze manier is een equality join dus vrij simpel geïmplementeerd in Map/Reduce. Er zijn echter twee nadelen verbonden aan deze manier van werken. Vooreerst is het aantal reducers gelimiteerd door het aantal mogelijke waarden voor het attribuut A, aangezien alle tupels met dezelfde waarde voor A door dezelfde reducer verwerkt worden. Indien bovendien de data niet evenredig over de mogelijke waarden voor A verdeeld is, zullen sommige reducers veel meer tupels moeten verwerken, wat leidt tot een langere uitvoertijd voor de job.

Inequality joins Stel nu dat opnieuw over de relaties S en T beschikken, en we deze willen joinen volgende de conditie $S.A \leq T.A$. Elk tupel van S

moet nu gejoind worden met alle tupels van T die een gelijke of kleinere waarde hebben voor A. Dit soort join is duidelijk veel moeilijker te implementeren in Map/Reduce. Stel dat het attribuut A enkel niet-negatieve integer waarden kan aannemen, dan is een mogelijke aanpak als volgt. Voor elke tupel uit S genereren we een tussenliggend key/value paar met als key S.A. Voor elk tupel uit T genereren we voor alle waarden tussen 0 en T.A een tussenliggende key/value paar. Op die manier krijgt een reducer de tupels van uit S met één bepaalde waarde voor A, en alle tupels uit T die een grotere of gelijke A-waarde hebben. Het spreekt voor zich dat dit voor een enorme toename aan data kan zorgen, er aanzienlijk meer bandbreedte verbruikt zal worden, en dat deze aanpak bovendien niet werkt als ook negatieve integers toegelaten zijn. Uit dit voorbeeld blijkt duidelijk de reden waarom de makers van Pig geen inequality join operatie hebben geïmplementeerd: het past simpelweg niet goed binnen het Map/Reduce programmeermodel. Uiteraard kunnen inequality joins of gecorelleerde subqueries toch gebruikt worden door het schrijvenvan UDFs. Hierdoor is de gebruiker zich echter bewust over de (in?)efficiëntie van hun programma. Verder wordt er actief onderzoek gedaan naar het efficiënt(er) implementeren van zowel equality als inequality joins in Map/Reduce[17], waardoor deze situatie in de toekomst zou kunnen verbeteren.

1.4.2.6 Debugging

Applicaties worden zelden meteen foutloos geschreven, vaak zijn er enkele test runs nodig om alle bugs op te sporen en te verhelpen. Aangezien Pig bedoelt is voor erg grote data sets kan dit debugging proces vrij lang duren. Om deze situatie te verhelpen zullen gebruikers vaak een test data set creëren tijdens de ontwikkelingsfase van hun applicatie. Dit kost echter tijd, en indien men niet goed oplet is de dataset bovendien niet representatief voor alle mogelijke scenario's die zich kunnen voordoen. Om wille van deze redenen bechikt Pig over de mogelijkheid om automatisch een test data set te genereren die *Pig Pen* genoemd wordt. Dit laat de gebruiker toe om stap voor stap zijn programma uit te voeren, en het resultaat van elke operatie op de test data te inspecteren. Het samenstellen van de test data gebeut als volgt. Er wordt een klein aantal random samples genomen uit de echte data set. Vervolgens worden hieraan tupels toegevoegd om de data set zo representatief mogelijk te maken. Stel bijvoorbeeld dat een programma een join neemt tussen twee relaties. Dan zal Pig ervoor zorgen dat de test data set tupels bevat die aan de join voorwaarden voldoen. Wanneer Pig een representatieve data set gegenereerd heeft wordt er nog getracht zo veel mogelijk tupels uit deze data set te verwijderen, zonder dat de representatieve eigenschap ervan verloren gaat. Het is ondertussen wel duidelijk dat deze *Pig Pen* een zeer handige tool is en gebruikers heel wat tijd kan uitsparen.

1.4.3 Pig Latin

Alvorens we de omzetting van een Pig Latin programma naar één of meerdere Map/Reduce jobs bespreken, geven we een overzicht van de beschikbare data types en operaties in Pig.

1.4.3.1 Data model

Pig is opgebouwd uit de volgende vier data types:

- **Atoom:** Een atoom bevat een simpele atomische waarde zoals een string of een getal. Pig ondersteunt de volgende atomische types: *int, long, float, double, chararray, bytearray.*
- **Tupel:** Een tupel is een opeenvolging van *velden.* Een veld kan van eender welk datatype zijn (dus ook een bag of map).
- **Bag:** Een bag is een verzameling tupels waarbij sommige tupels meerdere keren kunnen voorkomen. Merk op dat het niet noodzakelijk is dat de tupels in een bag uit dezelfde types opgebouwd zijn. De volgende tupels kunnen in Pig dus perfect in dezelfde bag voorkomen:

```
1 ('squash', 'box'), (20), {('a'), ('b', 'c')}
```

  Het eerste tupel bevat twee velden van het type string, het tweede tupel bevat een integer veld, en het derde tupel stelt tot slot opnieuw een bag voor. De makers van Pig Latin gebruiken trouwens de term *relatie* om een (outer) bag aan te duiden.

- **Map:** Een map is essentie een verzameling van key/value paren. Hierbij moet de key (om efficiëntie redenen) een *chararray* zijn. Values mogen eender welk type aannemen, en net zoals bij een bag moeten niet alle values van hetzelfde type zijn. Deze flexibiliteit is vooral handig voor data sets wiens schema na een bepaalde tijd zou kunnen veranderen, door bijvoorbeeld de toevoeging van een extra veld.

1.4.3.2 Expressies

Een overzicht van de beschikbare soorten expressies in Pig Latin wordt gegeven in Tabel 1.5. Hierbij zullen we gebruik maken van onderstaand tupel t, met de velden $f1, f2, f3$.

```
1 t = ('test', {('Pig',1), ('Latin',2)}, ['age' -> 20])
```

Type expressie	Voorbeeld	Waarde voor t
Constante	'ja'	n.v.t.
Veld door naam	f1	'test'
Veld door positie	$0	'test'
Projectie	f2.$0	{('Pig'),('Latin')}
Map lookup	f3#'age'	20
Functie (of UDF) evaluatie	SUM(f2,$1)	1+2=3
Conditionele expressie	f3#'age' >18 ? 'y' : 'n'	'y'
Ontnesten	FLATTEN(f2)	('Pig', 1), ('Latin',2)

Tabel 1.5: De verschillende soorten expressies in Pig Latin.

De meeste expressies behoeven geen verdere uitleg, zeker aangezien het ook niet ons doel is om Pig Latin syntactisch volledig te doorgronden. Enkel de flatten operatie is misschien niet zo voor de hand liggend. Dit commando kan zowel tupels als bags ontnesten. Voor het geval van tupels zorgt flatten ervoor dat een veld van een tupel ontnest wordt, zoals getoond wordt in onderstaand voorbeeld.

```
1 //stel schema r1: (a, (b, c))
2 FOREACH r1 GENERATE $0, FLATTEN($1);
3 //schema van r1: (a,b,c)
```

Voor het geval van bags worden er nieuwe tupels gecreëerd uit de inhoud van een bag. Merk op dat dit ervoor kan zorgen dat er een cross product genomen wordt, zoals wordt aangetoond met volgend voorbeeld.

```
1 //stel schema r1: (a, {(b,c),(d,e)})
2 FOREACH r1 GENERATE $0, FLATTEN($1);
3 //r1 bevat tupels van de vorm (a,b,c) en (a,d,e)
```

1.4.3.3 Relationele operaties

We overlopen kort de beschikbare relationele operaties in Pig. Dit is van belang om in te zien hoe een Pig Latin programma kan omgezet worden naar één of meerdere Map/Reduce jobs.

- **LOAD:** De load operatie specificeert welke de input files zijn, hoe deze gedeserialiseerd moeten worden, en kan een schema specificeren waaraan de resulterende data moet voldoen. De syntax van het load commando is als volgt.

  ```
  1 Handle = LOAD 'data' [USING function] [AS schema];
  ```

  Hierbij kan data een bestandsnaam, een directory, of een glob zijn (zie paragraaf 1.2.1.3). Indien er geen deserialisatie functie wordt gespecificeerd zal de default deserializer gebruikt worden, dewelke een tab-seperated tekst bestand als input verwacht. Ook het specificeren van een schema is

volledig optioneel. Indien dit niet gebeurt kan je velden aanspreken door hun positie (dus bijvoorbeeld $0h voor het eerste veld). Een voorbeeld van een load commando is als volgt.

```
studenten = LOAD 'studenten.txt' AS (stud_nr: int,
    naam: chararray, voornaam: chararray);
```

Indien er geen schema, of geen types voorkomen in het schema zijn de velden van het default type bytearray. Belangrijk om op te merken is dat het load commando er niet voor zorgt dat data effectief wordt ingelezen, zoals bijvoorbeeld bij een DBMS het geval zou zijn. Het is zelfs zo dat er helemaal geen operaties worden uitgevoerd tot er een store commando aangeroepen wordt. Het load commando geeft dus enkel aan hoe input data gelezen moet worden, en niet wanneer.

- **STORE:** Het store commando dient van zelfsprekend voor het opslaan van data naar een bestand. De syntax is als volgt.

```
STORE alias INTO 'directory' [USING function];
```

Het specificeren van een serializatie functie is opnieuw optioneel, aangezien er per default een serializer gebruikt wordt die tab-seperated tekst bestanden aanmaakt. Merk op dat de output in een directory wordt opgeslagen, en niet in een bestand. Dit komt omdat Map/Reduce voor elke Reducer een nieuw output bestand aanmaakt. Indien je dus je output in één bestand wilt opslaan impliceert dit dus het gebruik van slechts één reducer. Deze situatie illustreert hoe het Map/Reduce paradigma vergaande invloed heeft op de beschikbare operaties in Pig en hun syntax, hoewel het vaak niet meteen duidelijk is.

- **FOREACH:** Om tupels uit een relatie één voor één te verwerken kun je gebruik maken van het foreach commando, dat de volgende syntax heeft.

```
Handle = FOREACH alias GENERATE expressie [,
    expressie ]
```

Hierbij stelt *alias* een bag voor, en *expressie* eender welke expressie (zie Tabel 1.5). Het spreekt voor zich dat elk tupel uit *alias* onafhankelijk van de andere tupels verwerkt zal worden, en deze operatie dus makkelijk te parallelliseren valt.

- **FILTER:** Het selecteren van tupels volgens bepaalde voorwaarden kan aan de hand van de filter operatie. Zijn syntax is als volgt.

```
alias = FILTER alias BY expression;
```

Zoals je ziet zijn weer alle soorten expressies uit Tabel 1.5 toegelaten (inclusief UDFs).

- **COGROUP:** Het groeperen van data in één of meerdere relaties kan door gebruik te maken van het cogroup commando met onderstaande syntax.

```
alias = COGROUP alias BY tuple_expression [, alias BY
    tuple_expression ] [PARALLEL n];
```

De cogroup operator groepeert tupels die dezelfde *group key* hebben, dewelke wordt bepaald door de tupel expressie(s). Een tupel expressie is simpelweg een opeenvolging van expressies gescheiden door komma's. Het resultaat van een cogroup operatie is een relatie die één tupel bevat voor elke unieke waarde van de group key. Het eerste veld van het tupel krijgt het alias *group* en is van hetzelfde type als de group key. Verder is er een veld voor elke originele relatie die betrokken is in de operatie. Dit veld krijgt de naam van de originele relatie en is van het type bag. Het bevat alle tupels uit de originele relatie die dezelfde group key hebben. We illusteren de cogroup operatie met enkelevoorbeelden. Stel dat de relatie A volgens het schema $(f1 : chararray, f2 : int, f3 : int)$ gedefinieerd is en de tupels $(a, 1, 2), (b, 3, 3), (c, 2, 1)$ bevat. Onderstaande cogroup operatie zal ervoor zorgen dat de bekomen relatie van het schema $(group : chararray, A : \{f1 : chararray, f2 : int, f3 : int\})$ is, en de tupels $(2, \{(a, 1, 2), (c, 2, 1)\}), (9, \{(b, 3, 3)\})$ bevat.

```
B = COGROUP A BY f2*f3;
```

We geven nog een voorbeeldje met twee relaties.

```
studenten = COGROUP studentHasselt BY naam,
    studentLeuven BY naam;
```

Stel dat beide relaties van het schema $(naam : chararray, voornaam : chararray)$ zijn, en dat ze respectievelijk de namen van studenten die studeren in Hasselt en in Leuven bevatten. Stel dat de relatie studentHasselt het tupel $(`smets', `tom')$ bevat en de relatie studentLeuven de tupels $(`smets', `tom'), (`smets', `debbie')$. De resulterende relatie bevat dan het tupel $(`smets', \{(`smets', `tom')\}, \{(`smets', `tom'), (`smets', `debbie')\})$.

- **GROUP:** Het group commando heeft exact dezelfde syntax als cogroup, en wordt enkel gebruikt om de leesbaarheid van programma's te verhogen. Zo wordt group gebruikt indien de operatie slechts betrekking heeft op één relatie, en cogroup indien er meerdere relaties gegroupeerd worden.

- **JOIN:** Het is de aandachtige lezer misschien al opgevallen, maar met de combinatie van een cogroup en een flatten operatie kun je joins modelleren in Pig Latin. Verder bouwend op ons voorbeeld met de studenten relaties kunnen we met behulp van onderstaande code dus een (equality) join realiseren.

```
studenten = COGROUP studentHasselt BY naam,
    studentLeuven BY naam;
studenten_join = FOREACH studenten GENERATE FLATTEN(
    studentHasselt), FLATTEN(studentLeuven);
```

Pig laat echter ook de volgende syntactische shortcut toe.

```
studenten_join = JOIN studentHasselt BY naam,
    studentLeuven BY naam;
```

- **UNION:** De union operatie geeft vanzelfsprekend de unie terug van twee relaties. Zijn syntax is als volgt.

```
alias = UNION alias, alias [, alias     ];
```

  Merk op dat de relaties niet van hetzelfde schema moeten zijn, hoewel de union operatie meestal wel in zo'n situatie gebruikt zal worden. Het is aan de gebruiker om te garanderen dat de relaties van hetzelfde schema zijn, of om ervoor te zorgen dat de rest van het progamma in staat is tupels met verschillende schema's te verwerken.

- **CROSS:** De syntax van de cross operatie (die het cross product van twee of meer bags genereert) wordt getoond in het onderstaande code fragmentje.

```
alias = CROSS alias, alias [, alias     ] [PARALLEL n
    ];
```

  Zoals je kan zien kan je optioneel ook de *parallel* optie specificeren. Deze is bepalend voor het aantal reducers, en is per default op één ingesteld. Zoals je weet is het optimaal aantal reducers erg applicatie (en cluster) specifiek.

- **ORDER:** Een relatie sorteren volgens één of meerdere velden kan als volgt.

```
alias = ORDER alias BY { * [ASC|DESC] | field_alias [
    ASC|DESC] [, field_alias [ASC|DESC]     ] } [
    PARALLEL n];
```

  Het lijkt misschien vreemd om een relatie te sorteren, aangezien een relatie een bag is, en dus per definitie ongeordend. In Pig is het zo dat wanneer je een relatie sorteert, deze ordening verloren gaat van zodra je er nieuwe operaties op uitvoert (zoals bijvoorbeeld een filter operatie).

- **DISTINCT:** Tot slot kan het distinct commande gebruikt worden voor het elimineren van duplicaten uit een bag.

```
alias = DISTINCT alias [PARALLEL n];
```

- **DIFFERENCE:** De difference operatie, die het verschil neemt van twee relaties A en B, is niet beschikbaar in Pig Latin. Waarom deze ontbreekt is niet echt duidelijk, aangezien deze operatie vrij makkelijk te implementeren valt binnen Map/Reduce. Je zou bijvoorbeeld als volgt te werk kunnen

gaan. Voeg aan beide relaties een 'tag' toe om aan te geven tot welke relatie ze behoren. Vervolgens zorg je ervoor dat tupels met dezelfde waarden voor elk veld bij dezelfde Reducer terecht komen (waarbij je het 'tag' veld buiten beschouwing laat). De Reducer weet nu hoeveel 'identieke' tupels er in relatie A en B zitten, en kan simpelweg het aantal tupels in A verminderen met het aantal tupels in B, en het resulterende aantal tupels outputten. Zoals je merkt is dit gemakkelijk en efficiënt te implementeren in Map/Reduce. Indien we echter enkel gebruik maken van de beschikbare Pig Latin operaties is het implementeren van een difference operatie heel wat complexer. Een voor de hand liggende methode zou gebruik maken van een filter of foreach operatie, gevolgd door een 'NOT EXISTS' (zoals een SQL-programmeur te werk zou gaan). In Pig Latin is er echter geen '(not) exists' of '(not) in' operatie beschikbaar; de filter en foreach operaties kunnen enkel gebruikt worden in combinatie met de expressies uit Tabel 1.5. Er zijn mogelijke oplossingen te bedenken voor het implementeren van een difference operatie in Pig Latin, maar ze zijn weinig voor de hand liggend, en minder efficiënt dan de voorgestelde implementatie in Map/Reduce. We concluderen dus dat een difference operatie best zou worden toegevoegd aan de lijst van Pig Latin commandos aangezien dit in veel scenario's van pas zou kunnen komen, en bovendien efficiënt geïmplementeerd kan worden.

1.4.4 Van Pig Latin naar Map/Reduce

We hebben reeds de motivatie achter Pig, alsook de taal Pig Latin besproken. Er rest ons nog de omzetting van een Pig Latin programma naar één of meerdere Map/Reduce jobs. Hierbij wordt er als volgt te werk gegaan. Pig gaat voor elk commando dat de gebruiker uitvoert na of het syntactisch correct is, en of de input bestanden en/of relaties geldit zijn etc. Vervolgens wordt er een *logisch plan* opgesteld voor elke relatie die gedefinieerd werd door de gebruiker. Stel bijvoorbeeld dat de relaties A en B reeds gedefinieerd zijn, en dat de gebruiker onderstaand commando invoert.

```
1 C = COGROUP A BY ... , B BY ... ;
```

Het logische plan voor C bestaat dan uit een cogroup commando met de logische plannen voor A en B als input. Herinner je dat een logisch plan pas wordt omgezet in een *fysiek plan* wanneer er een store commando wordt aangeroepen. Het spreekt voor zich dat dit soort lazy evaluation verschillende optimalisaties toe laat, zoals we later zullen aantonen. In de meest recente release versies van Pig is het zelfs zo dat niet elk store commando leidt tot de aanmaak van een fysiek plan. Dit noemt men 'multiquery execution' en bespreken we in de volgende paragraaf.

1.4.4.1 Multiquery execution

Door niet meteen elk store commando om te zetten in een fysiek plan en uit te voeren kunnen er nog extra optimalisaties worden doorgevoerd, zoals volgend voorbeeld aantoont.

```
1 A = LOAD 'input.txt';
2 B = FILTER A BY $0 == 'more';
3 C = FILTER A BY $0 == 'efficient';
4 STORE B INTO 'output_b';
5 STORE C INTO 'output_c';
```

In de vorige versies van Pig had dit stukje code geleid tot de aanmaak van twee Map/Reduce jobs. Zo'n job bestond uit het inlezen van de relatie A, en vervolgens een filter operatie uit te voeren. Deze filter kan gewoon plaatsvinden in de Map functie, waardoor de Reduce functie in essentie een no-op is. Het punt is dat het inlezen van de relatie van A twee keer moet gebeuren. Uiteraard zorgt ook de aanmaak en het managen van twee Map/Reduce jobs (in plaats van één job) voor extra overhead. Gelukkig werd deze situatie verholpen door de *multiquery execution*. Deze feature zorgt ervoor dat bovenstaand Pig Latin programma omgezet wordt in slechts één Map/Reduce job, waarbij de relatie A dus slechts één keer wordt ingelezen. Er worden simpelweg naar twee output directories geschreven, één directory voor de tupels van B, en eentje voor de tupels van C.

1.4.4.2 Van een logisch plan naar Map/Reduce

We hebben reeds besproken hoe een logisch plan is opgebouwd, en wanneer deze worden omgezet in een fysiek plan (en dus Map/Reduce jobs). De vraag is nu hoe de omzetting van een logisch naar een fysiek plan gebeurt. Door de keuze van de operaties die ondersteund worden in Pig Latin is deze omzetting relatief simpel.

- **COGROUP** De omzetting van een logisch naar een fysiek plan begint door van elke cogroup operatie een aparte Map/Reduce job te maken. De omzetting van een cogroup commando gebeurt als volgt. De map functie kent keys toe aan de tupels op basis van de BY clause(s), waardoor ze bij dezelfde reducer terecht komen, die voorlopig nog geen berekeningen uitvoert. Indien de cogroup operatie betrekking heeft op meerdere relaties zal er aan elk tupel een veld worden toegevoegd om aan te geven uit welke relatie het tupel komt. De reducer gebruikt deze informatie om de tupels in de juiste geneste bag te plaatsen (zie paragraaf 1.4.3.3 COGROUP).

- **FILTER & FOREACH** De filter en foreach operaties worden op hun beurt in de Mapper van de eerst volgende cogroup operatie geplaatst. Aangezien elke map functie één tupel als input krijgt kunnen hier gemakkelijk filter en foreach operaties worden uitgevoerd. De filter operatie zorgt er algemeen voor dat een (mogelijk aanzienlijk) deel van de tupels

verwijderd worden uit een relatie. Door deze in de map functie te plaatsen zullen er dus minder tupels verwerkt moeten worden door de reducers, en vindt er dus ook een kleinere data transfer plaats. Uiteraard verhoogt dit de efficiëntie. Merk op dat we de filter en foreach operaties net zo goed in de reducer hadden kunnen pushen. Dit is ook wat er gebeurt indien er nog filter of foreach operaties plaatsvinden na de laatste cogroup operatie. Herinner je dat Map/Reduce de specificatie van een combiner functie toelaat (zie paragraaf 1.2.1.6 Combiner). Bovendien kan de generate clause van een foreach commando algebraïsche functies zoals count, sum, max, etc. bevatten, dewelke zich perfect lenen tot preprocessing door een combiner alvorens de data naar de reducers gaat. Hier maakt Pig dan ook handig gebruik van om de datatransfer tussen mappers en reducers zo klein mogelijk te maken. Er is zelfs de mogelijkheid voorzien om aan te geven dat een UDF een algebraïsche functie is, waardoor ook hier een combiner voor gebruikt kan worden.

- **LOAD & STORE** Aangezien de input bestanden voor een Pig programma zich in HDFS bevinden, is de load operatie automatisch geparallelliseerd (op dezelfde manier als een normale Map/Reduce job). De store operatie schrijft op zijn beurt bestanden weg naar HDFS, wat opnieuw op dezelfde manier als een normale Map/Reduce job gebeurt (dus vrijwel automatisch).

- **DISTINCT** De distinct operatie kan simpelweg bekomen worden door de map functie een key te laten generen op basis van alle velden van een tupel. Zo komen identieke tupels bij eenzelfde reducer terecht. Net zoals voor algebraïsche functies maakt Pig hier opnieuw handig gebruik van de combiner om reeds identieke tupels te verwijderen, en zo data te reduceren alvorens ze naar de reducers verstuurd wordt.

- **ORDER** De order operatie wordt geïmplementeerd door middel van 2 verschillende Map/Reduce jobs. De eerste job neemt sample waarden (op een regelmatig interval) van de sorteer waarde. In de tweede Map/Reduce job worden in de map fase de tupels in ongeveer even grote intervallen verdeeld door gebruik te maken van de sample waarden. Zo kunnen de reducers op hun beurt de tupels in een bepaald interval sorteren om een globaal gesorteerde relatie te bekomen.

- **UNION** Een union operatie kan plaats vinden wanneer de data wordt ingelezen voor een bepaalde Map/Reduce job. Dit gaat door simpelweg de directories (of bestanden) waar de relaties uit de union operatie zich bevinden als input directories te gebruiken.

- **CROSS** De manier waarop de cross operatie geïmplementeerd wordt is minder eenvoudig (en de informatie hieromtrent is ook minder makkelijk terug te vinden). Het valt in ieder geval meteen op dat een cross operatie typisch veel data genereert. Het schrijven van deze data naar HDFS is kostbaar, waardoor je cross operaties best zo veel mogelijk tracht te

vermijden. Zonder in detail te treden kunnen we de lezer ook nog meegeven dat de cross operatie zorgt voor de duplicatie van tupels in de map fase (van één van de resulterende Map/Reduce jobs), wat zorgt voor meer bandbreedte verbruik tijdens de shuffle & sort fase.

1.4.4.3 Voor- en nadelen van Pig

We eindigen de bespreking van Pig met een overzicht van de voor- en nadelen ten opzichte van Hadoop Map/Reduce. Vooreerst is Pig duidelijk een higher level taal dan Map/Reduce, wat ervoor zorgt dat de programmeur zich met minder implementatie details zal moeten bezig houden. Bovendien leunt het redelijk dicht aan bij SQL, waardoor het programmeren ook intuïtiever verloopt. Dit zorgt bovendien voor leesbaardere programma's, die makkelijker te onderhouden en te delen zijn. Het kan in sommige gevallen ook handig zijn om meer controle te hebben over data dan door de beschikbare operaties in Pig mogelijk is. Gelukkig biedt de uitgebreide ondersteuning voor UDFs hier een oplossing voor, zeker aangezien deze ook efficiënt geïmplementeerd kunnen worden (denk bijvoorbeeld aan de algebraïsche UDFs die gebruik maken van de combiner functionaliteit). Tot slot zorgt de automatische generatie van test data door middel van de Pig Pen ervoor dat applicaties een kortere ontwikkelingsfase hebben, en clusters bovendien minder zwaar belast worden. Een mogelijk nadeel van Pig is dat applicaties mogelijk minder performant zijn dan wanneer ze meteen in Map/Reduce geschreven zouden worden. Dit is vooral het geval wanneer je een oplossing voor je probleem kan bekomen dat minder Map/Reduce jobs omvat, aangezien de output van een Map/Reduce job naar HDFS geschreven wordt. Langs de andere kant is het niet ondenkbaar dat de door Pig gegenereerde Map/Reduce code efficiënter is dan code die je zelf zou schrijven. Dit is zeker het geval naarmate Pig verder ontwikkeld wordt, en er gesofisticeerdere algoritmen worden gebruikt om de omzetting naar Map/Reduce jobs te maken. Tot slot stellen de makers van Pig dat Pig Latin programma's (in theorie) ook omgezet kunnen worden naar een ander framework dan Hadoop Map/Reduce [20]. Dit lijkt me echter zeer onwaarschijnlijk, aangezien de omzetting naar Hadoop Map/Reduce vergaande invloed heeft gehad op de beschikbare operaties in Pig en hun specifieke syntax. Een voorbeeldje hiervan is dat de store operatie een directory als parameter verwacht en geen bestand. Immers, een Map/Reduce job genereert een output bestand voor elke reducer. Bovendien is de omzetting van een logisch plan naar een fysiek plan alsook alle optimalisatie stappen volledig gekoppeld aan het Map/Reduce programmeer paradigma. Desalniettemin blijft Pig een heel aantal voordelen bieden waardoor het zeker een handige uitbreiding kan vormen voor een Hadoop cluster.

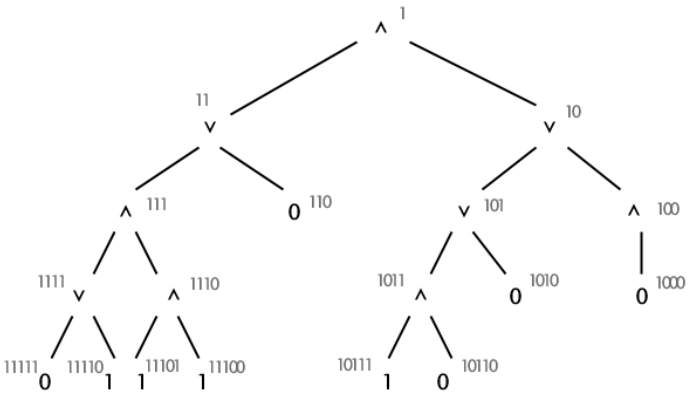
1.5 Hadoop in de praktijk

We hebben het Map/Reduce framework en het file systeem HDFS van Hadoop reeds uitgebreid besproken, en tevens de high level taal Pig Latin aangehaald

om aan mogelijke tekortkomingen van Hadoop tegemoet te komen. Om wat meer praktijkervaring op te doen, en mogelijk tot nieuwe conclusies te komen, zullen we een concrete case implementeren in Hadoop. Bovendien zullen we dezelfde case ook implementeren in de overige twee programmeerparadigma's, JMS en Boom.

1.5.1 Case study: gedistribueerde evaluatie van binaire expressiebomen

De case study die we gaan implementeren betreft de gedistribueerde evaluatie van een binaire expressieboom. Figuur 1.11 toont een voorbeeld van zo'n boom. Zoals je kan zien hebben alle bladeren van de boom het label '0' of '1'. De interne knopen zijn gelabeld met de AND-, OR- of NOT-operator. Het doel is om de waarheidswaarde te bekomen voor de wortel van de boom.



Figuur 1.11: Voorbeeld van een binaire expressieboom.

Merk op dat we een handige notatie hanteren voor het toekennen van *nodeID's* aan knopen. De wortel van de boom krijgt nodeID '1'. Vervolgens worden zijn kinderen gelabeld met zijn eigen nodeID gevolgd door een '1' of een '0' voor respectievelijk het linker- en rechterkind. Dit herhaalt zich voor de rest van de knopen in de boom. Op deze manier is informatie zoals de ouder van een knoop, de kinderen van een knoop, en de buren van een knoop afleidbaar enkel en alleen uit het nodeID. Dat we deze notatie kunnen gebruiken volgt uit het feit dat het een binaire boom betreft. Een ander belangrijk voordeel aan deze notatie is dat je enkel op basis van een nodeID kan weten op welke diepte deze knoop zich bevindt in de boom, wat erg handig zal blijken voor onze implementatie in Hadoop.

1.5.2 Omzetting naar het Map/Reduce programmeer paradigma

Aangezien Hadoop achterliggend voor de parallellisatie van een programma zorgt zal parallellisme niet de moeilijkheid vormen van de implementatie van onze case. De moeilijkheid ligt namelijk in de omzetting van een probleem naar een oplossing binnen het Map/Reduce framework. Zoals reeds werd aangehaald is dit programmeermodel vrij restrictief. Voor onze case betekent dit dat we meerdere Map/Reduce jobs nodig zullen hebben om het probleem op te lossen.

Het achterliggend idee ligt voor de hand: verdeel de boom in kleinere deelbomen, los deze (indien mogelijk) op, en reduceer zo de oorspronkelijke boom. Herhaal dit tot de volledige boom opgelost is. Concreet zullen we in de Mapper ervoor zorgen dat knopen uit dezelfde deelboom terecht komen bij eenzelfde Reducer (Er is geen garantie op welke knopen een Mapper als input krijgt). Vervolgens zal een Reducer nagaan of de deelboom oplosbaar is of niet. Indien dit niet zo is zal hij gewoon de deelboom terug wegschrijven als input voor de volgende Map/Reduce job. Indien hij de deelboom wel kon oplossen zal de resulterende output één record bevatten: het nodeID van de wortel van de deelboom met zijn bijhorende waarheidswaarde.

1.5.3 Implementatie

We hebben reeds een techniek bedacht om een binaire expressieboom te evalueren binnen het Map/Reduce programmeermodel. Onderstaand code fragment toont (in pseudo-code) hoe we dit algoritme kunnen implementeren binnen Hadoop.

```
1 map(Text nodeID, Text label) {
2
3 }
4
5 reduce(Text rootNodeID, Iterator nodes) {
6
7 }
```

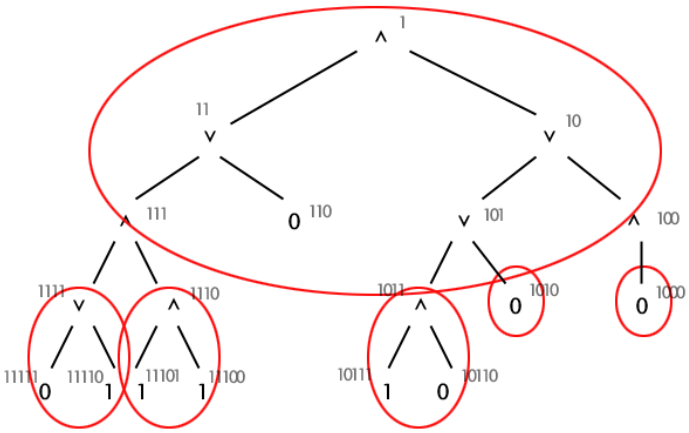
We leggen de code uit door het overlopen van de data flow. Zoals je kan zien staat de input opgeslagen in een tekst bestand, waarbij de keys en values gescheiden zijn door een tab-karakter. Het input bestand horende bij Figuur 1.11 zou er dus als volgt kunnen uitzien.

```
1 1         AND
2 11        OR
3 10        OR
4 111       AND
5 110       0
6 101       OR
7 100       NOT
8 1111      OR
```

```
1110    AND
1011    AND
1010    0
1000    0
11111   0
11110   1
11101   1
11100   1
10111   1
10110   0
```

Merk op dat de knopen in dit bestand in een willekeurige volgorde mogen voorkomen, en dat ze zelfs verspreid mogen zijn over verschillende input bestanden. Elke Mapper zal een deel van deze key/value paren als input krijgen. Stel nu dat we deelbomen van grootte drie willen verwerken, zoals geïllustreerd wordt door Figuur 1.12. Dan is het de bedoeling dat de Mappers ervoor zorgen dat de volgende intermediate key/value paren ontstaan:

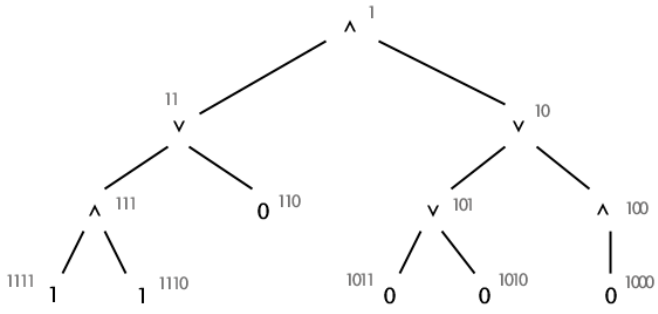
```
1       1       AND
1       11      OR
1       10      OR
1       111     AND
1       110     0
1       101     OR
1       100     NOT
1111    1111    OR
1111    11111   0
1111    11110   1
1110    1110    AND
1110    11101   1
1110    11100   1
1011    1011    AND
1011    10111   1
1011    10110   0
1010    1010    0
1000    1000    0
```



Figuur 1.12: De deelbomen die (onafhankelijk) door de Reducers verwerkt moeten worden.

Een reducer krijgt dus een deelboom zoals aangegeven in Figuur 1.12 te verwerken. Hoe weet een Mapper nu naar welke tussenliggende key een input key/value paar gemapt moet worden? De aandachtige lezer heeft opgemerkt dat de wortels van de bekomen deelbomen zich ofwel op diepte 0, ofwel op diepte 3 van de originele boom bevinden. Indien we dus een input key/value paar hebben waarvan de lengte van de nodeID deelbaar is door drie, wordt dit de wortel van een deelboom. Indien de lengte van de nodeID niet deelbaar is door drie doen we het volgende. We verwijderen de x laatste tekens van de nodeID, waarbij x de rest is bij deling door drie. Het resultaat is de tussenliggende key. Een analoge redenering geldt voor de verwerking van deelbomen met een andere grootte dan drie, en uiteraard is de grootte van een deelboom instelbaar bij de implementatie van ons programma.

We hebben reeds besproken hoe de input gespecificeerd wordt, en hoe de Mapper ervoor zorgt dat een Reducer een deelboom als input ter beschikking krijgt. In de Reducer gaan we nu als volgt te werk. Vooreerst dienen we na te gaan of we de deelboom kunnen oplossen of niet. Dit is vrij simpel: indien alle bladeren van de deelboom een waarde als label hebben (en dus geen operator), is de boom oplosbaar. In dat geval wordt de boom ook effectief opgelost binnen de Reducer. Het resultaat hiervan is één output key/value paar met als key de nodeID van de wortel, en als value de waarheidswaarde van de wortel na evaluatie van de deelboom. Stel nu dat de deelboom nog niet oplosbaar is. In dat geval zijn de tussenliggende key/value paren van deze Reducer eveneens de output key/value paren. Na één iteratie (of Map/Reduce job) ziet de boom er dus uit zoals getoond in Figuur 1.13.



Figuur 1.13: De gereduceerde expressieboom na één Map/Reduce iteratie.

In de volgende Map/Reduce job(s) zal de bekomen expressieboom verder gereduceerd worden tot er slechts één knoop overblijft. Het label van deze knoop geeft dan het resultaat van de evaluatie van de binaire expressieboom aan (In ons voorbeeld is dit '1').

1.5.4 Praktische overwegingen

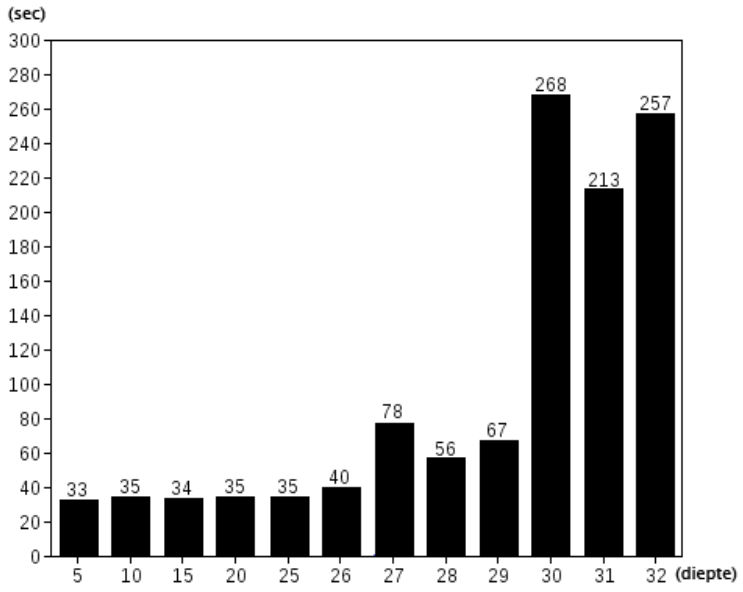
Zoals je gemerkt hebt bij het overlopen van ons voorbeeldje zijn er typisch meerdere Map/Reduce jobs nodig voor het evalueren van één enkele expressieboom. Het aantal jobs is ongeveer gelijk aan de diepte van de boom gedeeld door de grootte van de deelbomen die verwerkt worden. Uiteraard zorgt elke Map/Reduce job voor heel wat overhead. De input voor Mappers moet worden opgehaald, de tussenliggende key/value paren moeten bij de juiste Reducers terecht komen (in de *shuffle en sort* fase), en de output van de Reducers moet naar HDFS geschreven worden (herinner je dat dit ervoor zorgt dat de data in drievoud wordt opgeslagen). Deze grote overhead zou (in theorie) enorm terug te dringen zijn op de volgende manier. We gaan analoog te werk zoals voorheen, maar indien een Reducer een deelboom niet kan oplossen *wacht* deze op de oplossing(en) van andere Reducer(s), tot hij genoeg informatie heeft om zijn eigen deelboom op te lossen. Vervolgens communiceert hij zijn oplossing naar de andere Reducers. Op die manier zou er slechts één Map/Reduce job nodig zijn voor de evaluatie van een expressieboom, en zou er bijgevolg minder overhead zijn. Om nog maar te zwijgen van de bandbreedte die wordt uitgespaard, en het dubbel werk dat wordt uitgespaard. Immers, in onze implementatie wordt enkel de 'onderkant' van een boom aangepast. De rest van de boom wordt identiek weggeschreven naar HDFS, om in een volgende Map/Reduce job weer op een identieke manier verwerkt te worden. Helaas is communicatie tussen Reducers niet mogelijk in Map/Reduce, waardoor we opnieuw tot de conclusie komen die we al langer vermoedden. Het Map/Reduce framework bespaart de programmeur het werk om zelf voor de communicatie tussen nodes in de cluster te zorgen (m.a.w. het framework zorgt voor de parallellisatie), maar dit gaat ten koste

van efficiëntie en/of de bruikbaarheid van het framework. Ofwel beschikt men dus over een probleem dat mooi binnen het Map/Reduce programmeer paradigma past, en het dus naar alle waarschijnlijkheid sneller en efficiënter opgelost kan worden door gebruik te maken van Hadoop. Ofwel beschikt men over een probleem dat minder goed binnen dit paradigma past, en zal men aan efficiënte moeten inboeten indien men het toch binnen Hadoop wil implementeren.

Naast deze meer algemene conclusie over het gebruik van het Map/Reduce programmeermodel, heeft het implementeren van deze case ook geleid tot vaststellingen meer specifiek aan het gebruikte framework Hadoop. In onze implementatie maken we gebruik van tekstuele input bestanden. Het spreekt voor zich dat dit niet meteen tot de meest efficiënte representatievorm zal leiden. Jammer genoeg is het erg omslachtig om niet-tekstuele input bestanden te gebruiken voor een Map/Reduce job. Zoals we in paragraaf 1.2.1.3 reeds gezien hebben is er wel een binair formaat beschikbaar (SequenceFileInputFormat), maar dit formaat is specifiek aan Hadoop. Een Map/Reduce job kan dus wel output bestanden in dit formaat genereren voor de verwerking door eventueel volgende Map/Reduce jobs, maar het probleem blijft het input formaat voor de initiële Map/Reduce job. Een mogelijkheid zou zijn om een eigen binair formaat te creëren. Dit impliceert echter dat je de InputFormat klasse moet overloaden en tevens een RecordReader moet implementeren. Bovendien moet je er voor zorgen dat je bestanden verdeeld kunnen worden in input splits. Indien hierdoor de types van de key/value paren niet meer van een standaard Hadoop type (zie paragraaf 1.2.1.5) zijn, moet je ervoor zorgen dat deze nieuwe types de Writable klasse implementeren. Herinner je dat de Writable klassen van Hadoop trouwens vergeleken kunnen worden zonder eerst gedeserialiseerd te worden (in de shuffle en sort fase), wat een aanzienlijke snelheidswinst kan opleveren. De conclusie is dat het wel mogelijk is om met niet-tekstuele input bestanden te werken, maar dat dit voor veel extra werk voor de programmeur zou zorgen, zeker indien het programma zo efficiënt mogelijk moet werken. Aangezien het Map/Reduce programmeer paradigma zich vooral leent tot de ad-hoc analyse van grote hoeveelheden data (uit bijvoorbeeld log files) is deze 'beperkte' functionaliteit van Hadoop met betrekking tot input formaten wel verantwoord.

1.5.5 Testresultaten

We eindigen de bespreking van de evaluatie van binaire expressiebomen in Hadoop met enkele testresultaten. Het programma is ontwikkeld op een single-node distributie van Hadoop om de voor de hand liggende redenen dat dit makkelijker en sneller te debuggen is, en we op die manier niet onnodig een computer cluster gaan belasten. Hoewel het nagaan van de performantie van onze implementatie niet het hoofddoel is van deze case study tonen we in Figuur 1.14 toch enkele testresultaten.



Figuur 1.14: De uitvoeringstijd van het programma ten opzichte van de diepte van de geëvalueerde binaire expressieboom.

Zoals je kan zien geeft de x-as de diepte van de boom aan (een perfect binairee boom met diepte d heeft $2^{d+1} - 1$ knopen), terwijl de y-as de uitvoeringstijd weergeeft. De gebruikte computer bevatte een Intel i7-2670QM (4 cores, 2.2GHz) processor, 8GB RAM geheugen, en een harde schijf die opereert aan een snelheid van 7200 rpm. Op deze hardware zijn expressiebomen tot ongeveer een diepte van 32 in relatief korte tijd oplosbaar. Merk op dat een perfect binaire boom van deze diepte meer dan 4 miljard knopen bevat. Voor onze binaire expressie bomen is dit echter aanzienlijk minder, aangezien er één kans op drie is dat een interne knoop een NOT-operatie is, en dus maar één kind heeft. Tabel 1.5.5 toont de grootte van de gebruikte inputbestanden en het bijhorende aantal knopen per boom. Zoals je kan zien kan de NOT-operatie er dus voor zorgen dat een boom met een grotere diepte minder knopen bevat dan een boom met een kleinere diepte. Het is duidelijk dat des te hoger in de boom de NOT-operatie(s) zich bevinden, des te groter hun invloed op de totale grootte van de boom. Uit Figuur 1.14 bleek dat de uitvoertijd van de boom met diepte 30 groter was dan die van de bomen met diepte 31 of 32. Dit resultaat valt te verklaren door het groter aantal knopen in de boom met diepte 30, zoals je kan afleiden uit Tabel 1.5.5.

Diepte	Bestandsgrootte	Aantal knopen
5	1 KB	28
10	4 KB	296
15	93 KB	5,433
16	113 KB	6,218
17	66 KB	3,417
18	309 KB	15,421
19	639 KB	30,440
20	621 KB	28,255
21	2,522 KB	109,881
22	1,095 KB	45,743
23	2,042 KB	81,966
24	10,233 KB	394,965
25	13,681 KB	509,508
26	23,415 KB	841,205
27	24,693 KB	857,079
28	81,386 KB	2,732,442
29	117,974 KB	3,835,062
30	329,240 KB	10,373,359
31	224,517 KB	6,862,762
32	276,152 KB	8,196,448

Tabel 1.6: Overzicht van de gebruikte inputdata.

1.5.6 Hadoop op een computer cluster

We hebben ook getracht onze applicatie te testen op een computer cluster. Hierbij zijn we echter op een aantal problemen gestoten. Aangezien de gebruikte cluster voor een algemeen doelpubliek bestemd is, is er standaard geen Hadoop distributie geïnstalleerd op de compute nodes. Om een Map/Reduce job te kunnen uitvoeren moeten we dus eerst een Hadoop distributie installeren op de compute nodes die we wensen te gebruiken. Na het uitvoeren van onze applicatie moet deze Hadoop distributie uiteraard weer verwijderd worden. Dit vergt uiteraard wat extra programmeerwerk, en zorgt bovendien voor extra overhead, zeker aangezien de inputdata eerst op het gedistribueerde bestandssysteem HDFS geplaatst moet worden. Verder bleek ook de standaard Hadoop distributie niet geschikt voor high-end hardware. Zo worden threads zelden gesuspend op een compute node die 8 of 16 CPU cores ter beschikking heeft, in tegenstelling tot de 'commodity hardware' waarvoor Hadoop ontwikkeld werd. Deze aanname werd echter wel gemaakt door de ontwikkelaars van Hadoop. Zonder verder in detail te treden stellen we dat het niet mogelijk is om de standaard Hadoop distributie te installeren op high-end hardware zonder aanpassingen te maken aan de code van het framework. Omwille van deze beperking hebben we onze applicatie niet getest op een computer cluster. Bovendien zijn de conclusies die we uit het testen van onze applicatie op een computer cluster kunnen trekken beperkt. Zo

kunnen we bijvoorbeeld moeilijk uitspraken doen over de performantie van een programmeermodel gebaseerd op de uitvoertijden van één bepaalde applicatie.

Commando	**Operatie**
-ls *path*	Toont de inhoud van de *path* directory; inclusief bestandsnamen, gebruikersrechten, grootte, etc.
-lsr *path*	Identiek aan $-ls$ waarbij ook recursief alle subdirectories getoond worden
-mv *src dest*	Verplaatst het *src* bestand of directory naar de *dest* directory (binnen HDFS).
-cp *src dest*	Kopiëert het *src* bestand of directory naar de *dest* directory (binnen HDFS).
-rm *path*	Verwijdert het opgegeven bestand of de lege directory gespecificeerd door *path*.
-rmr *path*	Verwijdert het opgegeven bestand of directory, inclusief alle subdirectories, aangegevendoor *path*.
-copyFromLocal *localSrc dest*	Kopiëert een bestand of directory van het lokale file systeem naar HDFS.
-moveFromLocal *localSrc dest*	Verplaatst een bestand of directory van het lokale file systeem naar HDFS.
-cat *filename*	Toont de inhoud van *filename* op de standaard output.
-copyToLocal *src localDest*	Kopiëert de *src* file of directory naar de opgegeven locatie in het lokale file systeem.
-moveToLocal *src localDest*	Verplaatst de *src* file of directory naar de opgegeven locatie in het lokale file systeem.
-mkdir *path*	Creëert een nieuwe directory in HDFS, inclusief enige niet bestaande parent directories.
-stat [format] *path*	Toont informatie over *path*. Format kan de volgende parameters bevatten: bestandsgrootte in blocken (%b), bestandsnaam (%n), block grootte (%o) en aantal replica's (%r).
-setrep [-R] [-w] *rep path*	Past de replicatie factor aan voor het opgegeven *path*. -R past dit recursief toe. De -w optie geeft aan om te wachten tot het nieuwe replicatie level bereikt is.
-getMerge *src localDest*	Alle bestanden uit de *src* directory worden gemerged (dus gesorteerd), en in één enkel *localDest* bestand geplaatst in het lokale file systeem.

Tabel 1.4: De beschikbare commandos in HDFS.

Hoofdstuk 2

Java Message Service

2.1 Inleiding

Het tweede programmeerparadigma dat we bespreken is gebaseerd op het versturen van messages tussen verschillende processen, applicaties of computers voor onderlinge communicatie. Eén van de meest bekende standaarden behorende tot dit programmeermodel is de Message Passing Interface (MPI)[6]. Hiervan zijn er verschillende implementaties beschikbaar, waaronder bijvoorbeeld Intel MPI[8] en het open source Open MPI[12]. De interfaces van MPI zijn gespecifieerd in C, C++, en Fortran. Hoewel Java geen officiële MPI binding heeft, zijn er toch verscheidene pogingen geweest om beide te combineren.[19]. Er is echter ook een standaard voor het versturen van messages in Java genaamd de Java Message Service[23], die vergelijkbare functionaliteit aanbiedt als MPI. Beide worden ze geclasseerd als middleware, wat inhoudt dat ze zich als het ware tussen de applicatie en het netwerk bevinden, om op die manier netwerk activiteit vanuit de applicatie te vereenvoudigen. Verder is de kern van de functionaliteit die ze aanbieden hetzelfde: het aanmaken, verzenden, ontvangen, en lezen van messages. Door deze taken te abstraheren kunnen gedistribueerde applicaties betrouwbaarder, asynchroon, en *loosely coupled* ontworpen worden (zie paragraaf 2.2).

Van de Java Message Service zijn er, net zoals van MPI, verschillende implementaties beschikbaar. Hiertoe behoren onder meer Oracle JMS[10] en het open source ActiveMQ[1]. Deze laatste behoort tot de Apache Software Foundation, en is tevens de implementatie die we zullen gebruiken om de case study betreffende binaire expressiebomen te implementeren. Hoewel de voor- en nadelen van de message passing aanpak ten opzichte van het gebruik van Hadoop pas na deze case study uitgebreid besproken zullen worden, lijsten we nu toch al enkele duidelijke verschillen op. Vooreerst biedt Hadoop het voordeel dat het achterliggende framework zorgt voor de automatische parallellisatie van een applicatie. Het gedistribueerde file systeem HDFS zorgt voor replicatie van data, waardoor de beschikbaarheid van data verhoogt, er een grotere kans is op data-lokale

processen, en het systeem minder gevoelig is voor het onbeschikbaar worden van nodes. Deze voordelen zijn echter niet gratis: het Map/Reduce programmeermodel is vrij restrictief (ten opziche van het message passing paradigma). Hoewel dit probleem deels wordt aangepakt door de high-level programmeertaal Pig Latin, blijft het toepassingsgebied van Map/Reduce voornamelijk beperkt tot de analyse van grote (tekstuele) data sets. Dit is in tegenstelling tot het message passing programmeermodel, dat uiteraard veel ruimer toepasbaar is. Tot slot heeft de case study in Hadoop ons geleerd dat het gebruik van Hadoop op een computer cluster, die niet uitsluitend voor Hadoop bedoeld is, geen sinecure is. Naast het feit dat het framework geschreven is voor clusters opgebouwd uit commodity hardware, en niet voor high-end computers, onstaat er heel wat overhead bij het (tijdelijk) installeren van een Hadoop distributie voor de uitvoering van slechts één applicatie. Denk bijvoorbeeld maar aan de data die in HDFS moet worden ingeladen (in drievoud). Uiteraard kan geargumenteerd worden dat Hadoop niet ontwikkeld werd voor dit soort toepassingen, wat ook zo is. We willen hiermee enkel vaststellen dat dit het toepassingsgebied en het gebruik van Hadoop beperkt.

De Java Message Service heeft zoals reeds vermeld niet alleen een ruimer toepassingsgebied, het heeft bovendien ook geen last van de zojuist vermelde problemen specifiek aan het gebruik van een 'niet-dedicated' computer cluster. Uiteraard verliezen we ook enkele voordelen die Hadoop met zich meebracht. Automatische data replicatie, fault-tolerance, data-lokale processen, het automatisch toekennen van taken aan niet-actieve compute nodes, het zijn maar enkele van de functies die het framework verzorgde. Dit wil niet zeggen dat het niet mogelijk is deze functionaliteit te creëren met de Java Message Service, het zal echter heel wat programmeerwerk vergen. Het is natuurlijk ook zo dat bepaalde applicaties geen behoefte hebben aan al deze functionaliteit, zeker in combinatie met high-end hardware.

2.2 Voordelen van Messaging

We hebben in de inleiding een beknopte introductie gegeven tot de beschikbare message passing systemen, en reeds enkele voor- en nadelen van het Map/Reduce programmeermodel ten opzichte van het message passing model op een rijtje gezet. In de volgende paragrafen zullen we de voordelen van message passing op zich bekijken. Hierbij denken we iets algemener dan het gebruik van message passing voor het parallelliseren van applicaties.

2.2.1 Loose coupling

Een veelgebruikte toepassing van message passing is de communicatie en integratie van heterogene systemen, bijvoorbeeld in bedrijfscontext. In dit geval spreekt men ook wel van 'enterprise messaging'. Door gebruik te maken van messaging kunnen applicaties die in andere platformen geïmplementeerd werden toch eenvoudig met elkaar communiceren, zonder dat de applicaties of

modules *tightly coupled* zijn. Hiermee bedoelen we dat de applicaties weinig kennis moeten hebben van de manier waarop ze geïmplementeerd zijn, of de state waarin ze zich bevinden. Er zijn zelfs scenarios waarbij de verzender van een message niet weet welke applicaties een message kunnen ontvangen, welke hier een antwoord op zullen sturen, en of er zelfs een antwoord op verstuurd zal worden. In paragraaf 2.5 zullen we een dergelijke use case bespreken. Het belang van *loosely coupled* systemen wordt duidelijk wanneer we ze vergelijken met tightly coupled systemen. Zo zullen veranderingen in één bepaalde module een domino-effect hebben op andere modules waarmee gecommuniceerd wordt. Bovendien is het creëren van een nieuwe module mogelijk arbeidsintensiever bij tightly coupled systemen omwille van een verhoogde afhankelijkheid van andere modules. Dit houdt dus in dat je een diepgaandere kennis moet hebben van de reeds bestaande modules. Tot slot leiden tightly coupled systemen er ook toe dat een bepaalde module moeilijker te hergebruiken wordt.

Loosely coupled systemen zorgen ook voor een grotere flexibiliteit van systemen doordat ze typisch een hoger level van abstractie hebben. Deze flexibiliteit is belangrijk om snel te kunnen reageren op nieuwe software of hardware vereisten, of veranderingen in de business environment. Zo kan men bijvoorbeeld een bepaalde module of framework vervangen, of zelfs van softwareleverancier veranderen zonder dat dit tot vergaande aanpassingen leidt bij andere componenten en systemen.

2.2.2 Reduceren van bottlenecks

Bottlenecks treden op wanneer een bepaald systeem of proces de requests die hij ontvangt niet tijdig kan verwerken. Anders gezegd, de arrival rate van requests is hoger dan de service rate. Message passing kan hiervoor een oplossing bieden. In plaats van requests die één voor één synchroon worden afgehandeld door een server of bepaald systeem worden requests verstuurd naar een messaging service. Deze verdeelt de requests over meerdere servers of *message listeners*. Op die manier ontstaat er een betere load balancing en kunnen bottlenecks gereduceerd of zelfs geelimineerd worden.

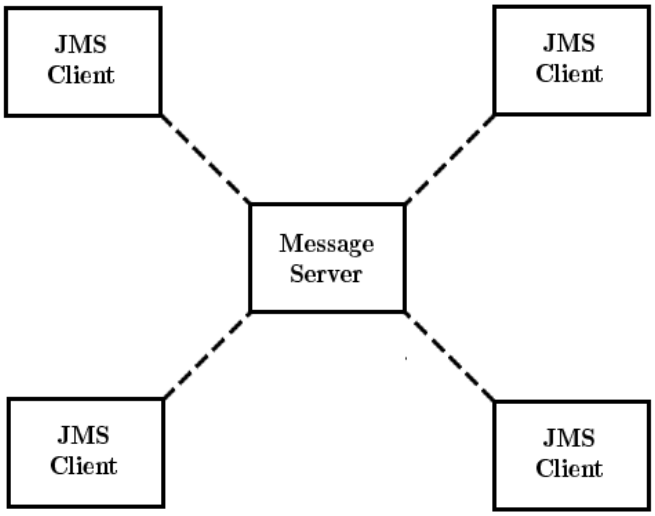
Het reduceren van bottlenecks zorgt er dus voor dat de throughput van een systeem verhoogd, en de response tijd mogelijk gereduceerd wordt. Dit leidt op zijn beurt tot een verhoogde schaalbaarheid van het systeem. Message passing op een asynchrone manier gebruiken maakt een systeem ook schaalbaarder. Immers, bij asynchrone communicatie blijven processen of applicaties niet 'wachten' op een antwoord op hun message maar gaan ze verder met een ander deel van hun berekeningen. Het spreekt voor zich dat applicaties op een asynchrone manier laten communiceren vaak meer moeite zal vereisen, of soms zelfs onmogelijk is. Desalniettemin faciliteert het gebruik van een message passing service asynchrone communicatie, wat dus tot de reeds vermelde voordelen kan leiden.

2.3 Architecturen en modellen voor message passing

In de volgende paragrafen zullen we een high-level overzicht geven van de mogelijke architecturen om message passing te verzorgen, alsook de verschillende messaging modellen of *messaging domeinen*.

2.3.1 Architecturen

Er zijn twee mogelijke manieren om message passing te verwezelijken: door gebruik te maken van een gecentraliseerde architectuur, of door gebruik te maken van een gedecentraliseerde architectuur. Een gecentraliseerde architectuur, ook wel een hub-and-spoke architectuur genoemd, maakt gebruik van een *message server* die verantwoordelijk is voor het afleveren van messages naar clients. Een message server wordt ook wel een *message router* of een *broker* genoemd, zoals bijvoorbeeld het geval is bij ActiveMQ. Figuur 2.1 illustreert deze structuur.

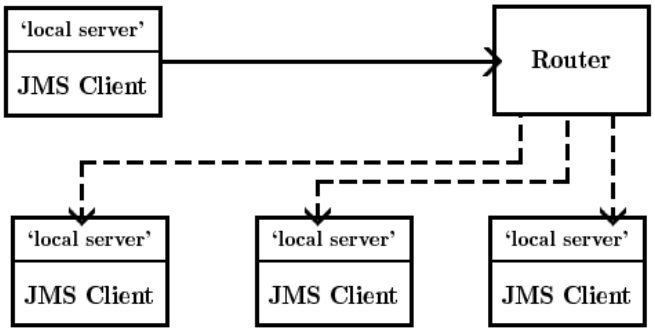


Figuur 2.1: Een hub-and-spoke of gecentraliseerde architectuur voor message passing.

Het valt meteen op dat de clients niet onderling in verbinding staan. Ze communiceren enkel met de message server, waardoor clients eenvoudig toegevoegd of verwijderd kunnen worden. In de praktijk kan het voorkomen dat één enkele message server niet voldoende capaciteit heeft om alle messages tijdig te verwerken, waardoor er een cluster van servers gebruikt die als logische eenheid gezien worden. Dit leidt tot een verhoogde schaalbaarheid, response time, en

een architectuur die minder vatbaar is voor het falen van een/de message server. Uiteraard zorgt dit ook weer voor toegevoegde complexiteit.

Bij gedecentraliseerde architecturen is er geen centrale server aanwezig; een deel van de functionaliteit die hierdoor verzorgd werd zit verwerkt in de clients. Voor het verspreiden van de messages wordt gebruik gemaakt van het IP multicast protocol. Dit laatste laat clients toe om bepaalde *multicast groups* te joinen, waarna het netwerk automatisch ervoor zorgt dat de messages naar alle leden van de groep verstuurd worden. Merk op dat dit niet impliceert dat messages gegarandeerd worden afgeleverd. Immers, multicasting maakt gebruik van UDP, aangezien het reliable protocol TCP eerder geschikt is voor point-to-point communicatie. Figuur 2.2 illustreert de gedecentraliseerde architectuur voor message passing. In de praktijk zal vaak voor een hub-and-spoke architectuur gekozen worden (mogelijk met meerdere brokers), zeker in bedrijfscontext.



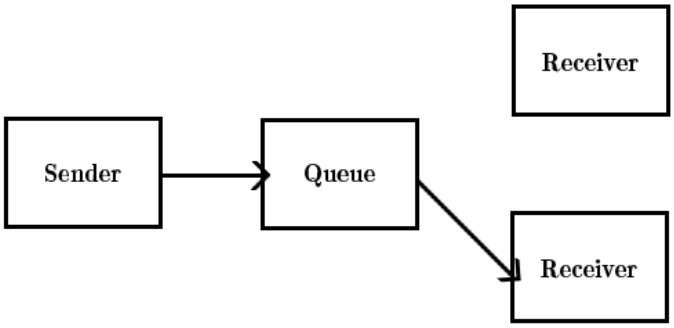
Figuur 2.2: Een gedecentraliseerde architectuur voor message passing gebaseerd op het IP multicast protocol.

2.3.2 Message passing modellen

2.3.2.1 Point-to-point model

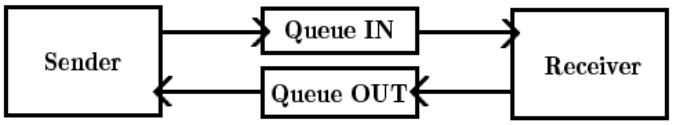
Naast de twee architecturen die we zojuist besproken hebben zijn er ook twee modellen voor message passing. Vooreerst is er het *point-to-point* of *p2p* model, dat bedoeld is voor het versturen van een message naar één ontvanger. De client die een message verstuurt noemen we een sender of *message producer*, de ontvangende client noemen we een receiver of *message consumer*. Messages kunnen zowel synchroon als asynchroon verstuurd worden via *queues*. Zoals je kan zien op Figuur 2.3 kunnen er meerdere receivers gekoppeld zijn aan een queue, maar zal de message slechts door één enkele receiver verwerkt worden. Het voordeel hiervan is dat er gemakkelijk aan load balancing gedaan kan worden. Zo zal een JMS implementatie erl voor zorgen dat de messages verdeeld worden onder de beschikbare receivers (hoewel er niet in de JMS specificatie staat hoe dit moet

gebeuren). Indien je een message naar één specifieke receiver wil versturen, kun je simpelweg gebruik maken van een queue waar enkel deze receiver aan gekoppeld is. Een ander typisch kenmerk van het point-to-point model is dat de messages afgeleverd worden in de volgorde waarin ze in de queue geplaatst zijn (hoewel je een message ook een bepaalde prioriteit kan geven om dit te omzeilen).



Figuur 2.3: Het point-to-point of p2p messaging domein.

Zoals reeds vermeld kunnen messages met het p2p model zowel synchroon als asynchroon verstuurd worden. In paragraaf 2.2.2 hebben we reeds aangehaald waarom asynchrone communicatie de voorkeur geniet. Een typisch voorbeeld van asynchrone communicatie is het versturen van een message naar een logging systeem. Bij synchrone communicatie verstuurt een producer een message naar één queue, en wacht deze vervolgens op een bericht in een andere (reply) queue, zoals getoond in Figuur 2.4.

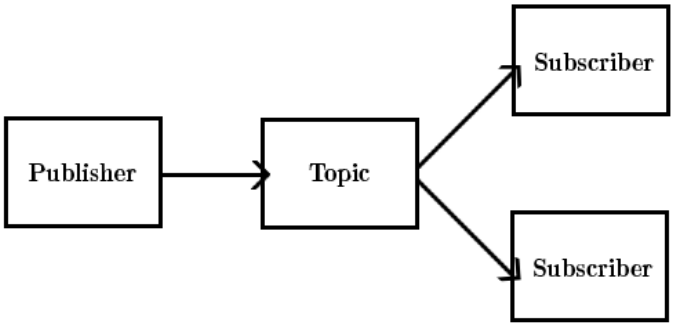


Figuur 2.4: Een overzicht van de werking van synchrone communicatie met het point-to-point messaging model.

2.3.2.2 Publish-and-subscribe model

De verstuurder van een message in het publish-and-subscribe model wordt een *publisher* genoemd, terwijl de ontvangers van messages *subscribers* zijn (zie Figuur 2.5). Het versturen van een message wordt ook het *broadcasten* van een

message genoemd aangezien alle subscribers een kopie van de message ontvangen. Het publish-and-subcribe of *pub/sub* model is een push-based model, wat inhoudt dat subscribers messages automatisch ontvangen zonder dat ze hiervoor moeten 'pollen' of een request uitvoeren. Het pub/sub model zorgt typisch voor minder coupling, aangezien publishers vaak niet weten hoeveel subscribers er zijn, wat deze met de message doen, en of ze er zelfs iets mee doen. Een voorbeeld hiervan is een applicatie die een message publiceert telkens er een foutmelding optreedt. Wat er verder met deze message gebeurt is voor de applicatie van geen belang.

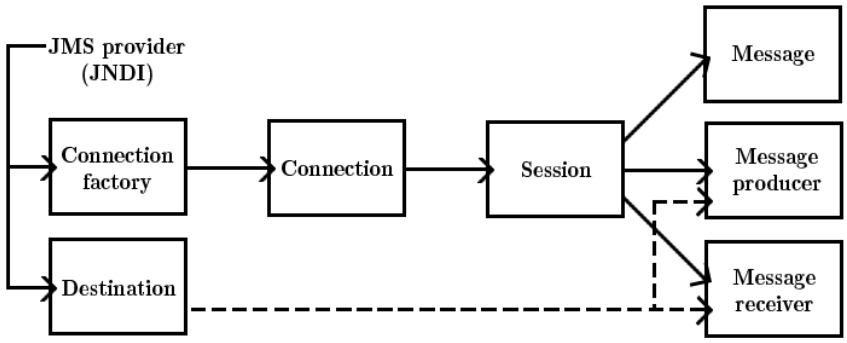


Figuur 2.5: Het publish-and-subscribe of pub/sub messaging domein.

Er zijn verschillende typen subscribers mogelijk. Zo zijn er bijvoorbeeld niet-duurzame subsribers, die enkel messages ontvangen wanneer ze 'actief' naar het topic luisteren. Daarnaast zijn er ook duurzame subscribers, die tevens messages ontvangen wanneer ze 'offline' zijn. De broker of message service dient hiervoor een queue bij te houden van messages die nog niet aan een bepaalde subscriber werden afgeleverd, waarover meer in paragraaf 2.8.1.2.

2.4 JMS API

In de volgende paragrafen zullen we de belangrijkste interfaces van de algemene API, de point-to-point API, alsook de publish-and-subscribe API bespreken. Zoals je kan zien in Figuur 2.6 zijn er zeven belangrijke interfaces beschikbaar in de algemene API.



Figuur 2.6: Een overzicht van de belangrijkste interfaces van de algemene JMS API.

2.4.1 ConnectionFactory

De *ConnectionFactory* is het basistype waarvan de *TopicConnectionFactory* en de *QueueConnectionFactory* van zijn afgeleid, dewelke gebruikt worden bij het pub/sub en p2p messaging model. De ConnectionFactory wordt zoals de naam doet vermoeden gebruikt om een *Connection* object aan te maken. Het bevat configuratie parameters voor een connectie die werden gedefinieerd door een administrator van het systeem. JMS clients kunnen dit soort objecten ('JMS administered objects') gebruiken door ze op te zoeken in een *JNDI namespace* (zie paragraaf 2.4.10). De interface van de ConnectionFactory is als volgt.

```
1 public interface ConnectionFactory {
2     public Connection createConnection() throws
          JMSException;
3     public Connection createConnection(String userName,
          String password) throws JMSException;
4 }
```

2.4.2 Destination

Net zoals de ConnectionFactory is een *Destination* een JMS administered object en kan het gebruikt worden door het op te zoeken in de JNDI namespace. Het wordt gebruikt om het adres van een message producer of receiver te specificeren, waarbij de syntax voor het adres afhangt van de gebruikte JMS provider. Merk op dat voor elke JMS provider geldt dat deze syntax 'verborgen' is voor de JMS clients; deze dienen enkel een JNDI lookup te doen. Bovendien is een destination geen specifieke applicatie is, maar eerder een 'virtueel kanaal' waarmee applicaties messages uitwisselen. Het is dus de basis interface voor de *Topic* en *Queue* interfaces uit respectievelijk het pub/sub en p2p messaging domein.

Het voordeel van deze virtuele kanalen is dat er minder coupling is tussen de applicaties die messages versturen en de applicaties die messages ontvangen.

2.4.3 Connection

Een *Connection* stelt een connectie voor van een JMS client met een JMS provider, en bevat de hiervoor vereiste resources (zoals bijvoorbeeld een TCP/IP socket). Bij het aanmaken van een Connection vindt (indien nodig) de authenticatie van een client plaats. Er wordt typisch maar één Connection aangemaakt per client voor het verzorgen van hun messaging. Uiteraard verziet de Connection interface methodes voor het aanmaken, starten, stoppen, en afsluiten van een connectie, zoals getoond in onderstaande specificatie.

```
1 public interface Connection {
2     public Session createSession(boolean transacted, int
          acknowledgeMode) throws JMSExcetpion;
3
4     public void start() throws JMSException;
5     public void stop() throws JMSException;
6     public void close() throws JMSException;
7 }
```

Merk op dat we enkel de belangrijkste methodes van de Connection interface hebben weergegeven. Zoals je kan zien in de specificatie kan een Connection object gebruikt worden om één of meerdere Session objecten aan te maken, dewelke allemaal gebruik maken van dezelfde socket. Aangezien het aanmaken van een Connection object dus tijd en resources vergt, is het vaak efficiënter om meerdere Session objecten te combineren met één Connection object. De start en stop methoden worden gebruikt voor het (tijdelijk) starten en stoppen van de aflevering van messages. De close methode sluit een connectie definitief af en zorgt ervoor dat de gebruikte resources weer worden vrijgegeven. Tot slot merken we nog op dat dit de basis interface is voor *TopicConnection* en *QueueConnection* uit het pub/sub en p2p domein.

2.4.4 Session

Een *Session* object wordt gebruikt voor het produceren en consumeren van messages. Een Session object is bovendien een factory voor het aanmaken van message producers en consumers, alsook messages op zich. Belangrijk om te weten is dat een Session object slechts vanuit één thread gemanipuleerd mag worden. Dit impliceert dat het versturen van messages, en het (asynchroon) ontvangen van messages typisch door twee aparte sessions geregeld wordt. Of anders gesteld, een message producer zal typisch gebruik maken van een session, en een message consumer van een andere session. Het is mogelijk om beide toch te combineren in één session, maar enkel indien de applicatie messages produceert in de *onMessage()* handler. Deze functie wordt door de JMS provider

aangeroepen wanneer een message consumer een nieuwe message dient te verwerken. Dit verklaart ook meteen de threading restrictie op een session: de JMS provider kan de onMessage() functie aanroepen wanneer de applicatie reeds een message aan het produceren is (of andere functionaliteit van een session aan het gebruiken is). Eén van de uitgangspunten van de JMS specificatie was zo weinig mogelijk impact te hebben op de achterliggende implementatie van een JMS provider. Men wou dus voorkomen dat een session in staat moest zijn om veilig meerdere threads af te handelen (aangezien enkele reeds bestaande messaging systemen dit niet ondersteunden). Onderstaand code-fragment toont een deel van de Session interface.

```
1  public interface Session extends java.lang.Runnable {
2  public TextMessage createTextMessage() throws
       JMSException;
3  public TextMessage createTextMessage(String text) throws
       JMSException;
4
5  public MessageConsumer createConsumer(Destination
       destination) throws JMSException;
6  public MessageProducer createProducer(Destination
       destination) throws JMSException;
7
8  public Queue createQueue(String queueName) throws
       JMSException;
9  public Topic createTopic(String topicName) throws
       JMSException;
10
11 public void close() throws JMSException;
12 }
```

Naast de *createTextMessage()* functies zijn er ook nog functies beschikbaar voor het aanmaken van andere soorten messages (zie Paragraaf 2.4.5). De *createConsumer* en *createProducer* functies worden uiteraard gebruikt voor het aanmaken van een consumer of producer voor een bepaalde destination. Tot slot worden *createQueue* en *createTopic* gebruikt om een queue of topic aan te maken, respectievelijk voor het p2p of pub/sub messaging domein.

2.4.5 Message

De Message interface is de basis voor alle beschikbare message types in JMS, met name: *Message*, *TextMessage*, *ObjectMessage*, *StreamMessage*, *BytesMessage* en *MapMessage*. Op het *Message* type na hebben alle soorten messages naast een *header* ook een *payload*. De header bevat velden voor de identificatie van de message, routing informatie, en de mogelijkheid tot het declareren van *Message properties* (zie Paragraaf 2.6.2). Het verschil tussen de beschikbare message types wordt vooral bepaald door het type payload dat ze bevatten. Aangezien de Message interface voornamelijk *get* en *set* functies bevat (en dit er behoorlijk

wat zijn) zullen we de specificatie hier niet weergeven. We geven echter wel beknopt een woordje uitleg bij de verschillende message types.

- **Message** Het basis message type heeft geen payload, en kan gebruikt worden als een event notification.
- **TextMessage** Dit message type bevat een String als payload en wordt bijgevolg gebruikt voor simpele tekstuele data te versturen.
- **ObjectMessage** Een *ObjectMessage* bevat een serializeerbaar Java object.
- **StreamMessage** Het *StreamMessage* type bevat een stream van Java types zoals int, double, boolean, etc. Met dit type kan je gemakkelijk een geordende reeks data omzetten in een byte stream en vice versa.
- **BytesMessage** Dit message type bevat een byte array als payload en kan bijvoorbeeld gebruikt worden indien JMS enkel gebruikt wordt voor de transfer van data waar de JMS client verder geen interesse in heeft.
- **MapMessage** Een *MapMessage* bevat een set van key/value paren als payload. De keys zijn steeds van het String type, terwijl de keys van eender welk primitief Java type mogen zijn.

2.4.6 MessageProducer

De *MessageProducer* is de basis interface voor de *TopicPublisher* (pub/sub) en *QueueSender* (p2p) interfaces, en wordt gebruikt voor het versturen van messages. Per default wordt de bestemming van deze messages bepaald door het Session object waarmee de MessageProducer werd aangemaakt, hoewel de bestemming ook per message bepaald kan worden. Hiernaast kan ook de prioriteit, delivery mode en time-to-live van messages worden ingesteld (zie Paragraaf 2.6). Dit kan zowel voor alle messages die verstuurd worden door een MessageProducer, of voor één bepaalde message. Naast de get- en set-functies voor de zojuist vernoemde opties bevat de MessageProducer interface nog de volgende functionaliteit:

```
1 public interface MessageProducer {
2     public void send(Destination destination, Message
          message, int deliveryMode, int priority, long
          timeToLive) throws JMSException;
3     public void close() throws JMSException;
4 }
```

Merk op dat enkel de Message parameter verplicht is bij het versturen van een message met behulp van de *send* functie. De *close()* methode wordt gebruikt om de MessageProducer af te sluiten en de gebruikte resources weer vrij te geven.

2.4.7 MessageConsumer

De *MessageConsumer* wordt gebruikt voor het ontvangen van messages, en is de basis interface voor *TTopicSubscriber* en *QueueReceiver* van respectievelijk het pub/sub en p2p messaging model. Net zoals bij de MessageProducer kan een MessageConsumer object worden aangemaakt door gebruik te maken van een Session object (waarbij er een Destination als parameter wordt meegegeven). Verder kan bij het aanmaken van een MessageConsumer ook een *MessageSelector* worden gespecificeerd. Hiermee zullen enkel messages die aan de MessageSelector voldoen worden afgeleverd bij de MessageConsumer (zie Paragraaf 2.7). Tot slot kan een client messages synchroon ontvangen, of ze asynchhroon laten afleveren door de MessageConsumer. In het eerste geval dient de client één van de *receive* functies aan te roepen. Hierbij beschikt men over de keuze tussen een message verwerken indien er een beschikbaar is (polling), of te wachten tot de volgende message arriveert. Indien men gebruik maakt van asynchrone messaging wordt er een *MessageListener* geregistreerd bij de MessageConsumer. Zodra de consumer een nieuwe message ontvangt wordt de *onMessage* functie van de MessageListener aangeroepen. We tonen nog kort een deel van de interface van de MessageConsumer.

```
1 public interface MessageConsumer {
2     public void close() throws JMSException;
3     public void setMessageListener(MessageListener
          listener) throws JMSException;
4     public Message receive() throws JMSException;
5     public Message receive(long timeout) throws
          JMSException;
6     public Message receiveNoWait() throws JMSException;
7 }
```

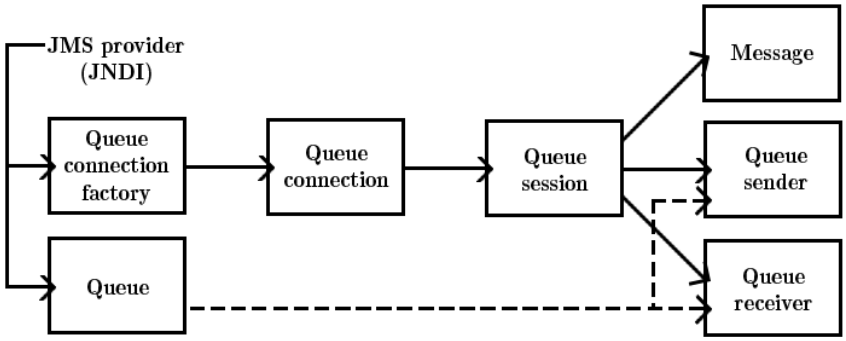
Zoals je kan zien zijn er drie verschillende *receive* functies beschikbaar. De *receive()* functie wacht net zo lang tot er een message beschikbaar is, de *receive(long timeout)* functie wacht tot er een message beschikbaar is of de timeout verlopen is. De *receiveNoWait()* functie tot slot ontvangt de volgende message indien er eentje beschikbaar is, en geeft anders *null* terug.

2.4.8 MessageListener

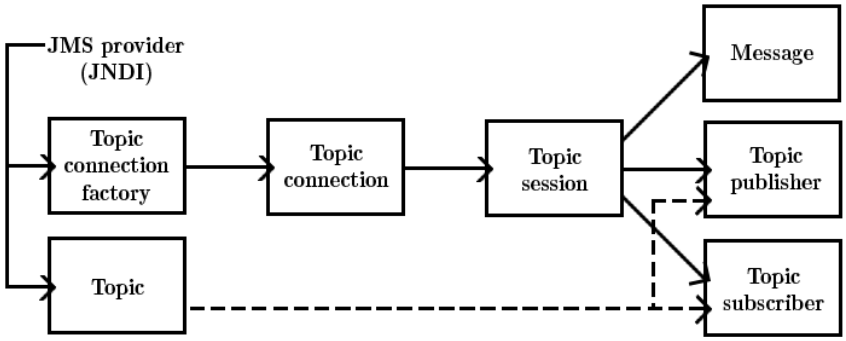
Zoals reeds vermeld wordt de *MessageListener* gebruikt voor het asynchroon ontvangen van messages van één of meerdere MessageConsumers. De Session die gebruikt werd bij het aanmaken van de MessageConsumer zorgt ervoor dat de messages één voor één afgeleverd worden aan de MessageListener. Merk op dat indien je de MessageListener zou registreren bij meerdere consumers, die bovendien door middel van verschillende Session objecten zijn aangemaakt, het niet meer gegarandeerd kan worden dat messages serieel worden afgeleverd. De MessageListener interface bevat enkel de *onMessage* functie, dewelke een Message als parameter heeft.

2.4.9 pub/sub & p2p API's

We hebben uitgebreid de belangrijkste interfaces uit algemene JMS API besproken, waarbij we telkens aanhaalden van welke interfaces uit het pub/sub en p2p messaging model afgeleid zijn van deze basis interfaces. We gaan de interfaces van het pub/sub en p2p model dan ook niet verder in detail beschrijven aangezien de achterliggende structuur dezelfde is als in de algemene JMS API. We tonen in Figuur 2.7 en Figuur 2.8 wel nog een overzicht van respectievelijk de point-to-point en publisher/subscriber JMS API's.



Figuur 2.7: Een overzicht van de belangrijkste interfaces van de point-to-point JMS API.



Figuur 2.8: Een overzicht van de belangrijkste interfaces van de pub/sub JMS API.

2.4.10 JNDI

Zoals reeds vermeld wordt de Java Naming and Directory Interface (JNDI) in JMS gebruikt voor het opzoeken van 'administered objects' zoals een Destination of ConnectionFactory (zie Paragraaf 2.4)[11]. JNDI is een java extensie die een API aanbiedt voor de toegang tot verschillende directory en naming services waaronder LDAP, NDS, CORBA Naming Service, etc[21][9]. Het is dus bedoeld voor het opzoeken van data en objecten aan de hand van een naam. Op deze manier kunnen ook gedistribueerde objecten of bestanden gelokaliseerd worden op het netwerk, zonder dat je steeds een IP adres, poort, etc. moet specificeren. Het bekendste voorbeeld van een naming service is waarschijnlijk het DNS of Domain Name System, dat internet domein namen omzet in IP adressen.

JNDI bewaart namen in een hiärchie. Typisch wordt de wortel van deze hiërarchie gebruikt om een *initial context* aan te maken. Alle lookups in JNDI gebeuren ten opzichte van een bepaalde context, waarbij de initial context is simpelweg het beginpunt is voor deze opzoekingen. Om een object op te zoeken dien je dus eerst een initial context te creëren, en vervolgens de *lookup* functie van deze context aan te roepen. We illustreren dit in onderstaand code-fragmentje.

```
1 Hashtable env = new Hashtable();
2 env.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.
    activemq.jndi.ActiveMQInitialContextFactory");
3 env.put(Context.PROVIDER_URL, "tcp://localhost:61616");
4 InitialContext ctx = new InitialContext();
5 TopicConnectionFactory conFactory = (
    TopicConnectionFactory)ctx.lookup(topicFactory);
```

Zoals je kan zien gebruiken we een Hashtable om de parameters van de inital context te specificeren. In dit voorbeeldje veronderstellen we dat er een ActiveQM (JMS server) applicatie actief is en gebruik maakt van poort 61616. Je zou deze parameters ook in *jndi.properties* bestand kunnen specificeren, en dit bestand meegeven in het classpath van je applicatie. De inhoudt van dit properties bestand zou er voor dit voorbeeld als volgt uitzien:

```
1 java.naming.factory.initial = org.apache.activemq.jndi.
    ActiveMQInitialContextFactory
2 java.naming.provider.url = tcp://localhost:61616
```

Een voordel van het gebruik van een jndi.properties bestand is dat wanneer meerdere applicaties van deze properties gebruik maken, je slechts op één plaats aanpassingen moet doorvoeren wanneer een property moet worden aangepast (bijvoorbeeld wanneer je je JMS server elders wilt hosten). Zonder een jndi.properties bestand zou je de code van al je applicaties moeten aanpassen en deze hercompileren.

2.5 Hello World

Nu we de belangrijkste interfaces van de JMS API hebben besproken is het tijd om deze kennis om te zetten in de praktijk. In dit deel bespreken we een eenvoudige chat applicatie waarbij we gebruik maken van een open source implementatie van JMS, namelijk ActiveMQ[1]. Hierbij zullen we gebruik maken van het publish and subscribe model aangezien dit zich perfect hiertoe leent. Immers, indien een client een bericht plaatst in een 'chat room' is het de bedoeling dat alle andere gebruikers dit bericht ontvangen (en niet slechts één gebruiker zoals bij het p2p messaging model). Een client is in dit geval zowel een publisher als een subscriber, aangezien hij zowel berichten kan plaatsen als berichten kan ontvangen. Het onderstaand code-fragment is afkomstig van de website van O'reilly [13] en toont hoe we zo'n chat applicatie kunnen implementeren.

```
1  public class Chat implements javax.jms.MessageListener{
2      private TopicSession pubSession;
3      private TopicPublisher publisher;
4      private TopicConnection connection;
5      private String username;
6
7      /* Constructor used to Initialize Chat */
8      public Chat(String topicFactory, String topicName,
           String username)
9          throws Exception {
10
11         // Obtain a JNDI connection using the jndi.
               properties file
12         InitialContext ctx = new InitialContext();
13
14         // Look up a JMS connection factory
15         TopicConnectionFactory conFactory =
16             (TopicConnectionFactory)ctx.lookup(
                   topicFactory);
17
18         // Create a JMS connection
19         TopicConnection connection = conFactory.
               createTopicConnection();
20
21         // Create two JMS session objects
22         TopicSession pubSession = connection.
               createTopicSession(
23             false, Session.AUTO_ACKNOWLEDGE);
24         TopicSession subSession = connection.
               createTopicSession(
25             false, Session.AUTO_ACKNOWLEDGE);
26
```

```
27          // Look up a JMS topic
28          Topic chatTopic = (Topic)ctx.lookup(topicName);
29
30          // Create a JMS publisher and subscriber
31          TopicPublisher publisher =
32              pubSession.createPublisher(chatTopic);
33          TopicSubscriber subscriber =
34              subSession.createSubscriber(chatTopic, null,
                    true);
35
36          // Set a JMS message listener
37          subscriber.setMessageListener(this);
38
39          // Intialize the Chat application variables
40          this.connection = connection;
41          this.pubSession = pubSession;
42          this.publisher = publisher;
43          this.username = username;
44
45          // Start the JMS connection; allows messages to
                be delivered
46          connection.start( );
47      }
48
49      /* Receive Messages From Topic Subscriber */
50      public void onMessage(Message message) {
51          try {
52              TextMessage textMessage = (TextMessage)
                    message;
53              String text = textMessage.getText( );
54              System.out.println(text);
55          } catch (JMSException jmse){ jmse.printStackTrace
                ( ); }
56      }
57
58      /* Create and Send Message Using Publisher */
59      protected void writeMessage(String text) throws
            JMSException {
60          TextMessage message = pubSession.
                createTextMessage( );
61          message.setText(username+": "+text);
62          publisher.publish(message);
63      }
64
65      /* Close the JMS Connection */
66      public void close( ) throws JMSException {
```

```
67          connection.close( );
68      }
69
70      /* Run the Chat Client */
71      public static void main(String [] args){
72          try{
73              if (args.length!=3)
74                  System.out.println("Factory, Topic, or
                        username missing");
75
76              // args[0]=topicFactory; args[1]=topicName;
                    args[2]=username
77              Chat chat = new Chat(args[0],args[1],args[2])
                    ;
78
79              // Read from command line
80              BufferedReader commandLine = new
81                java.io.BufferedReader(new
                      InputStreamReader(System.in));
82
83              // Loop until the word "exit" is typed
84              while(true){
85                  String s = commandLine.readLine( );
86                  if (s.equalsIgnoreCase("exit")){
87                      chat.close( ); // close down
                            connection
88                      System.exit(0);// exit program
89                  } else
90                      chat.writeMessage(s);
91              }
92          } catch (Exception e){ e.printStackTrace( ); }
93      }
94  }
```

2.5.1 Constructor van de Chat klasse

We beginnen met het bespreken van de constructor. Zoals je kan zien maken we gebruik van een jndi.properties bestand (aangezien er geen properties gespecificeerd worden bij het aanmaken van het InitialContext object), dat hieronder getoond wordt.

```
1 java.naming.factory.initial = org.apache.activemq.jndi.
      ActiveMQInitialContextFactory
2 java.naming.provider.url = tcp://localhost:61616
3 connectionFactoryNames = TopicCF
4 topic.topic1 = jms.topic1
```

De eerste twee regels hebben we reeds besproken in Paragraaf 2.4.10. De connectionFactory die we gebruiken is 'TopicCF', en wordt gebruikt voor het aanmaken van topicConnection objecten. Dit is (zoals reeds vermeld) een interface die afgeleid is van de Connection interface en gebruikt wordt in het pub/sub messaging model. De naam 'TopicCF' is vendor-specifiek. Vervolgens hebben we ook een Topic (afgeleid van een Destination object) gespecificeerd met de JNDI naam 'topic1'. Om dit Topic te kunnen gebruiken moeten we het ook registreren bij de JMS broker. Bij ActiveMQ gebeurd dit door de configuratie file *activemq.xml* aan te passen. Aan het xml element met de naam *broker* voegen we de volgende code toe:

```
1 <destinations>
2     <topic name="topic1" physicalName="jms.topic1" />
3 </destinations>
```

Merk op dat de xml elementen in *activemq.xml* in alfabetische volgorde moeten staan!

Nu we een TopicConnectionFactory object hebben aangemaakt kunnen we dit gebruiken om een TopicConnection te bekomen. Dit TopicConnection object gebruiken we op zijn beurt om twee Sessions aan te maken; eentje voor het versturen van messages een eentje voor het ontvangen van berichten van andere chat clients. In Paragraaf 2.4.4 hebben we reeds besproken waarom hier twee aparte Session objecten voor vereist zijn. Vervolgens maken we gebruik van de TopicSessions en het Topic object dat we hebben opgezocht via JNDI om een TopicPublisher en een TopicSubscriber object te creëren. Hierna registreren we het Chat object, dat is afgeleid van de MessageListener interface, als MessageListener bij de TopicSubscriber. Meer uitleg hieromtrent volgt bij de bespreking van de *onMessage* functie. Tot slot starten we de JMS connectie.

2.5.2 Messages ontvangen en versturen

Wanneer een TopicSubscriber een message ontvangt wordt de *onMessage* functie aangeroepen van het MessageListener object dat bij de subscriber geregistreerd staat. Merk op dat er per subscriber slechts één MessageListener geregistreerd mag worden. De *onMessage* functie in onze chat applicatie doet niets meer dan de payload van de message printen naar de standaard output.

Het versturen van een message gebeurd door gebruik te maken van het Publisher object in de *writeMessage* functie, die wordt aangeroepen door de *main* functie telkens de gebruiker een nieuwe lijn tekst invoert. Bij het aanmaken van een TextMessage moeten we enkel de payload specificeren; de headers met de nodige routing informatie worden automatisch ingevuld. De destination (dus het Topic) hebben we immers al meegegeven bij het aanmaken van de TopicSubscriber.

2.6 Anatomie van een message

In de komende paragrafen zullen we in detail de elementen bekijken waaruit een message is opgebouwd, namelijk de *headers*, de *message properties*, en de *payload*.

2.6.1 Headers

De message headers bevatten metadata zoals routing informatie, de time-to-live, de prioriteit van de message, etc. Er zijn twee soorten headers: headers die automatisch ingevuld worden door de JMS provider, en headers die door de programmeur ingesteld kunnen worden. Hoewel er voor elke header een get- en setfunctie beschikbaar is in de message interface, heeft het aanpassen van automatisch toegekende headers geen zin. De JMS provider zal deze aanpassingen immers negeren. Dat wil niet zeggen dat de gebruiker geen enkele controle heeft over deze headers: sommige waarden kunnen ingesteld worden door bijvoorbeeld gebruik te maken van de functionaliteit van een TopicPublisher. Het instellen van de prioriteit van een message is hier een voorbeeld van. We bespreken hieronder kort de belangrijkste automatisch toegekende headers:

- **JMSDestination** De JMSDestination header bevat de bestemming van een message, en komt overeen met een Queue of Topic. Herinner je dat bij het aanmaken van een message producer reeds een destination gespecificeerd wordt, dewelke dus de waarde bepaalt van deze header. Het opvragen van de JMSDestination met behulp van de *getJMSDestination* functie kan bijvoorbeeld nuttig zijn wanneer een client messages verwerkt van meerdere topics of queues. Concreet kan je dit doen door eenzelfde MessageListener te registreren bij meerdere message consumers.

- **JMSDeliveryMode** Er zijn twee soorten afleveringsmodi in JMS: persistent en niet-persistent, dewelke ingesteld kunnen worden met behulp van de *setDeliveryMode* functie van een MessageProducer. Stel dat een JMS server faalt wanneer ze bepaalde messages nog niet heeft verstuurd naar een consumer. Indien het persistente messages betreft zullen deze verstuurd worden wanneer de server terug beschikbaar wordt. Gaat het om niet-persistente messages zullen deze nooit verstuurd worden. Het spreekt voor zich dat persistente messages dus op een permanente opslagplaats (zoals een harde schijf) moeten worden opgeslaan alvorens de JMS server aan een message producer aangeeft dat hij de message goed ontvangen heeft. Dit brengt natuurlijk overhead met zich mee, die niet voorkomt bij het gebruik van niet-persistente messages. Meer informatie hierover vind je terug in Paragraaf 2.8.

- **JMSMessageID** Deze header bevat een String waarde die de message 'uniek' identificeert. Hoe 'uniek' deze is hangt af van de gebruikte JMS vendor. Ze kan bijvoorbeeld uniek zijn binnen één bepaalde JMS server, of uniek over alle servers heen.

- **JMSExpiration** De JMSExpiration header kan er voor zorgen dat een message niet meer afgeleverd wordt wanneer een bepaalde tijd verstreken is. Dit is nuttig voor messages die data bevatten die maar tijdelijk geldig zijn, zoals bijvoorbeeld de koers van een bepaald aandeel. Deze header kan worden ingesteld door de *setTimeToLive* functie van een message producer aan te roepen, waarbij de time-to-live in milliseconden wordt meegegeven. Een time-to-live van 0 zorgt ervoor dat een message nooit vervalt.

- **JMSRedelivered** Wanneer een consumer geen acknowledgement stuurt naar de JMS server bij het ontvangen van een message zal de server de *JMSRedelivered* header op *true* zetten. Meer informatie hierover wordt gegeven in Paragraaf 2.8.

- **JMSPriority** Een message producer kan een prioriteit toekennen aan een message met behulp van deze header. Er zijn 10 prioriteitslevels (gaande van 0 tot 9), waarbij de eerste vijf een *normale* prioriteit, en de laatste vijf een *verhoogde* prioriteit voorstellen. Een JMS server kan deze prioriteiten gebruiken om de volgorde waarin messages worden afgeleverd aan te passen.

Naast de automatisch ingevulde headers zijn er ook headers die door de programmeur ingesteld kunnen worden. De meest gebruikte hiervan is de *JMSReplyTo* header, die een destination (dus een JNDI naam) bevat. Deze kan gebruikt woren door een message producer om aan te geven waar een message consumer een reply kan geven op een message. Op deze manier zijn de producer en consumer nog meer loosely coupled bij een request/reply scenario. Onderstaand code-fragment illustreert dit.

```
1 //bij de producer
2 message.setJMSReplyTo(topic1);
3
4 //bij de consumer
5 Topic topic = (Topic) message.getJMSReplyTo();
6 ...
```

2.6.2 Message properties

Een message property is een key/value paar waarbij de key steeds een String is, en de value een String of Java primitive zoals een int, boolean, byte, etc. Ze kunnen door de gebruiker (of de JMS vendor) worden ingesteld om meer informatie toe te kennen aan een message. Van zodra een message verstuurd is worden de message properties read-only. Een consumer kan ze dus niet meer aanpassen door gebruik te maken van één van de set functies. Voor elk type dat de value van een message property kan aannemen is er een get en een set functie beschikbaar. Onderstaand code-fragmentje illustreert dit voor String properties.

```
1 public interface Message {
2 public String getStringProperty(String name) throws
      JMSException, MessageFormatException;
3 public void setStringProperty(String name, String value)
      throws JMSException, MessageNotWritableException;
4 ...
5 public Enumeration getPropertyNames() throws JMSException
      ;
```

Zoals je kan zien is er ook een *getPropertyNames* functie beschikbaar dewelke een lijst van de property namen teruggeeft. Deze kan dan gebruikt worden om de property values op te vragen. Indien je het type van een property niet kent kan je gebruik maken van de *getObjectProperty* functie om de value op te vragen.

2.7 Message filtering

Herinner je dat we tot nu toe gesteld hebben dat subscribers in het pub/sub messaging model alle messages van een bepaalde queue ontvangen. Verder stelden we ook dat een 'willekeurige' receiver in het p2p messaging model de eerst volgende message ontvangt ('willekeurig' aangezien er load balancing plaats kan vinden in het geval van meerdere receivers). Door gebruik te maken van message properties is het echter mogelijk om enkel messages te ontvangen die aan bepaalde voorwaarden voldoen. Dit proces noemen we *message filtering*.

We illustreren het nut hiervan met een klein voorbeeldje. Stel dat je een applicatie wilt ontwikkelen waarmee je de prijzen van de telecomoperatoren in Vlaanderen wilt opvolgen. Er moet een applicatie zijn voor klanten die de meest recente prijzen van de operatoren ontvangt, en indien deze voordeliger zijn dan de huidige deal van de klant krijgt deze hiervan een bericht. De klant zelf krijgt dus enkel een bericht indien er een betere deal is. De applicatie moet achterliggend nagaan welke deals hiervoor in aanmerking komen. Hiervoor moet hij dus alle berichten van alle operatoren ontvangen en verwerken. Ter verduidelijking: de klant is in dit geval een subscriber, de operatoren zijn publishers, en de destination is een topic.

Door gebruik te maken van message filtering kunnen we er echter voor zorgen dat de applicatie van de klant enkel die messages ontvangt die een betere deal bevatten. Onderstaand code-fragment illustreert hoe zo'n message filter of *message selector* eruit zou kunnen zien.

```
1 public class Klant implements javax.jms.MessageListener{
2     public Klant(String topiccf, String topicName) {
3         ...
4         topic = (Topic)ctx.lookup(topicName);
5         String filter = "currentPrice - newPrice > 0.0";
6         TopicSubscriber subscriber = session.
              createSubscriber(topic, filter, true);
7         ...
```

```
8       }
9       public void onMessage(Message message) {
10      //betere deal ontvangen, bericht tonen aan klant
11      ...
12  }
```

Zoals je kan zien is de message selector een string die als parameter wordt meegegeven tijdens het aanmaken van de subscriber. In de volgende paragraaf zullen we de syntax en sementiek van deze message selectors bespreken.

2.7.1 Message selectors

Message selectors kunnen gebruik maken van message headers en properties om een conditionele expressie te bekomen die bepaalt of een message al dan niet wordt afgeleverd aan een bepaalde consumer. Ze kunnen dus geen verwijzingen maken naar de message body. Verder zijn ze opgebouwd uit drie delen: identifiers, literals en vergelijkingsoperatoren. Hierbij komt een identifier overeen met ofwel een message header ofwel een message property (het zijn dus Strings). Merk op dat je geen headers kan gebruiken waarvan de value een object is, zoals bijvoorbeeld de JMSDestination aangezien deze een Destination object bevat. Een literal kan een String (tussen single quotes), een getal, of de booleaanse waarden *true* of *false* voorstellen. Tot slot worden de vergelijkingsoperatoren gebruikt om identifiers en literals te vergelijken in een booleaanse uitdrukking die evalueert naar *true* of *false*. De message wordt uiteraard enkel afgeleverd indien de message selector naar *true* evalueert. De beschikbare vergelijkingsoperatoren zijn de volgende:

- **Algebraïsche vergelijkingsoperatoren** Vooreerst zijn er de algebraïsche vergelijkingsoperatoren $=, >, >=, <, <=$, en $<>$ met de verwachte betekenis.
- **Meetkundige operatoren** Ook de meetkundige operatoren $+, -, *, /$ en de unaire min zijn beschikbaar.
- **LIKE operator** De LIKE operator wordt gebruikt bij het vergelijken vian Strings, waarbij er gebruik gemaakt kan worden van de wildcard operatoren _ en %. Hierbij staat de _ voor één willekeurig karakter, en het %-teken voor eender welke opeenvolging van karakters.
- **BETWEEN operator** De BETWEEN operator kan gebruikt worden om een bepaald interval uit te drukken. Merk op dat je de functionaliteit van deze operator ook kan bekomen door middel van de >=, <= en AND operatoren.
- **IN operator** Deze operator wordt gebruikt om aan te geven dat een identifier tot een bepaalde set moet behoren.
- **AND, OR, NOT operatoren** De functionaliteit van deze operatoren spreekt voor zich.

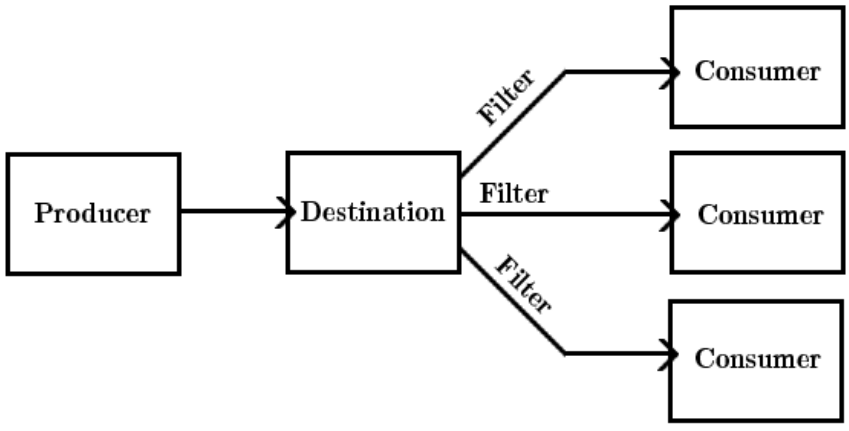
- **IS NULL operator** De IS NULL operator wordt gebruikt om na te gaan of een bepaalde header of message property bestaat.

2.7.2 Niet-afgeleverde messages

Indien een subscriber niet geïnteresseerd is in een message (door gebruik te maken van een message filter), wordt deze message simpelweg niet afgeleverd aan deze subscriber. In het p2p messaging model geldt dat een message niet zichtbaar is voor een receiver indien deze niet voldoet aan de message filter. In beide messaging modellen kan de situatie ontstaan waarbij geen enkele consumer geïnteresseerd is in een bepaalde message. Dit leidt ertoe dat een message in de queue of het topic zal bijgehouden worden door de JMS server, wat uiteraard niet gewenst is. Immers, de time-to-live van een message is per default van onbeperkte duur. Dit kan simpelweg opgelost worden door gebruik te maken van de *JMSExcpiration* header, of door gebruik te maken van een 'default' message consumer, die de messages verwerkt waarin geen enkele andere consumer in geïnteresseerd is. Deze laatste optie zorgt echter meer tightly couplad applicaties, aangezien dit vergaande kennis van alle andere message consumers vereist.

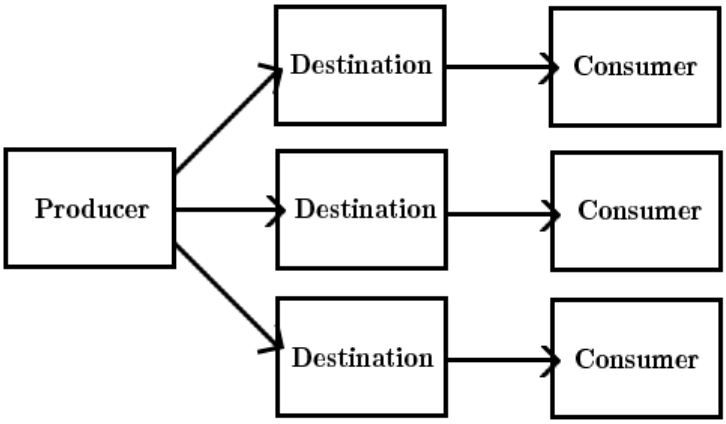
2.7.3 Design implicaties

Er zijn twee mogelijke manieren om messages te filteren, namelijk de *message filtering* aanpak en de *multiple destination* aanpak. Bij het message filtering model worden de messages naar één destination verstuurd, waarna de consumers enkel de messages ontvangen waarin ze geïnteresseerd zijn. Figuur 2.9 illustreert deze aanpak.



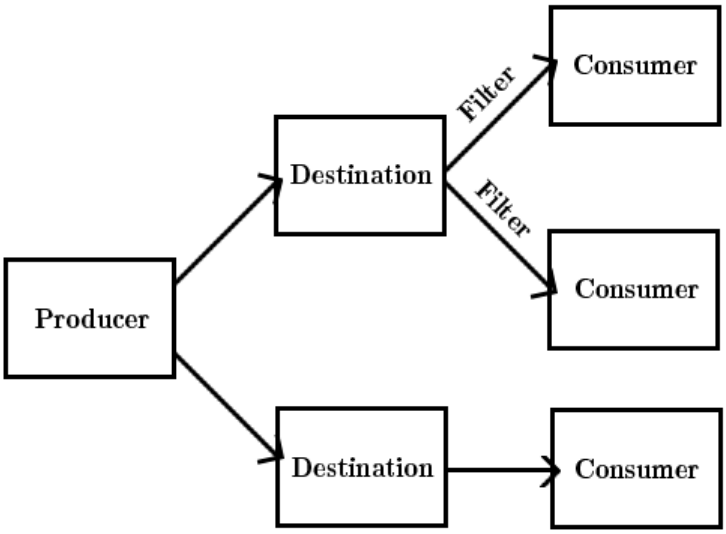
Figuur 2.9: Messages filteren gebruik makende van message selectors.

Bij het multiple desination model is het de message producer die (aan de hand van Java code) voor de filtering zorgt. In plaats van message selectors te gebruiken worden er meerdere destinations gebruikt, één per 'soort' message. Dit wordt getoond in Figuur 2.10.



Figuur 2.10: Messages filteren door gebruik te maken van meerdere Destinations.

Het verschil tussen beide modellen is waar de filtering plaats vindt; bij de message consumer of bij de producer. Bij het message filtering model beslissen de consumers zelf welke messages ze willen ontvangen. Dit zorgt voor minder coupling tussen producers en consumers, aangezien de message producer minder kennis nodig heeft over de consumers. Dit is in tegenstelling tot de multiple destination aanpak, aagezien een producer daar over voldoende kennis moet beschikken om te bepalen naar welke destination een message verstuurd dient te worden. Merk verder ook op dat het message filtering model makkelijker uitbreidbaar is. Indien er een nieuw type message verwerkt dient te worden kan je hier gewoon een nieuwe consumer voor aanmaken met de gepaste message selector. Bij het multiple destination model zou er nieuwe logica moeten worden ingebouwd in de producers om te bepalen naar welke destination dit soort messages moet gaan. Bovendien moet er een nieuwe destination worden aangemaakt waaraan nieuwe consumers gekoppeld worden die deze messages verwerken. Uiteraard kunnen beide aanpakken ook gecombineerd worden, zoals Figuur 2.11 illustreert.



Figuur 2.11: Een gecombineerde aanpak voor het filteren van messages.

Het is van belang dat de verscheidenheid aan messages die verwerkt worden door een bepaalde destination niet te groot, maar ook niet te klein is. Een te grote diversiteit zal immers voor onnodig veel message filtering zorgen bij message consumers, wat erop kan duiden dat de destination misschien beter gesplitst kan worden. Een te kleine diversiteit van messages per destination kan er dan weer voor zorgen dat message consumers bij meerdere destinations informatie moeten ophalen. In de praktijk leidt dit er toe dat er vaak een combinatie van beide modellen gebruikt zal worden.

2.8 Guaranteed messaging

Tot nu toe hebben we weinig aandacht besteed aan de fault tolerance van JMS. Er kunnen immers verschillende fouten optreden. Zo kan er bijvoorbeeld hardware of software falen bij de provider, of kan het netwerk onderbroken worden wanneer een client een message verstuurt naar de JMS server. Indien de *DeliveryMode* header van de message ingesteld is op *NON_PERSISTENT* betekent dit dat de message verloren zal gaan. Door gebruik te maken van de *PERSISTENT DeliveryMode* zal de message bewaard blijven tot de JMS provider van zijn fout herstelt of het netwerk niet langer onderbroken is. Hiervoor dient de JMS provider de message op te slaan op permanente storage alvorens er een acknowledgement naar de message producer verstuurd wordt. Aangezien dit voor extra overhead zorgt is het nuttig om na te gaan of je message we als persistent gemarkeerd moet worden. De delivery mode kan als parameter worden

meegegeven bij het aanmaken van een message producer, en eventueel worden aangepast door de *setDeliveryMode* functie van deze producer. Per default is deze ingesteld op persistent.

2.8.1 Message acknowledgements

Naast de twee soorten afleveringsmodi zijn er ook verschillende acknowledgement protocollen beschikbaar in JMS, waarvan AUTO_ACKNOWLEDGE en CLIENT_ACKNOWLEDGE en de belangrijkste zijnE. Ze kunnen worden ingesteld wanneer een Session object wordt aangemaakt. We bekijken acknowledgements vanuit het standpunt van de message producer, de JMS server en de consumer. Hierbij veronderstellen we dat we gebruik maken van de AUTO_ACKNOWLEDGE mode. Het verschil met de andere modi komt later aan bod.

2.8.1.1 De message producer

Wanneer een JMS server een message ontvangt van een message producer zal deze automatisch een acknowledgement versturen (mogelijk na het opslaan van de message op persistent storage). Van zodra de message producer deze acknowledgement ontvangt beschouwt deze de message als succesvol verstuurd. Het is nu de verantwoordelijkheid van de JMS server om ervoor te zorgen dat de message afgeleverd wordt bij de message consumer(s). Indien de message producer geen acknowledgement ontvangt van de JMS server treedt er een foutmelding op en wordt de message beschouwd als onafgeleverd.

2.8.1.2 De JMS server

Ook de JMS server ontvangt acknowledgements van de message consumers waar hij messages naar verstuurt. Indien het p2p messaging model gebruikt wordt zal de message verwijderd worden (uit het main memory en/of de persistente storage) van zodra één message receiver een acknowledgement verstuurd heeft. Bij het pub/sub messaging model moet de message in principe aan alle geïnteresseerde subscirbers worden afgeleverd. Indien een subscriber op het moment dat de message verstuurd wordt niet beschikbaar is (bijvoorbeeld door een onderbroken netwerkconnectie) zal deze de message in principe niet ontvangen. Het is echter ook mogelijk *durable subscribers* te gebruiken in JMS. Durable subscribers ontvangen ook alle berichten die gepubliceerd worden wanneer ze offline zijn, en kunnen aangemaakt worden met de *createDurableSubscriber* functie van een Session object. Indien durable subscribers gebruikt worden zal de JMS server de messages bewaren totdat de durable subsriber terug beschikbaar wordt. Merk op dat dit ervoor kan zorgen dat er behoorlijk veel messages bewaard moeten worden door de JMS server indien één of meerdere durable subscribers gedurende lange tijd onbeschikbaar zijn. Een mogelijke oplossing voor dit probleem is om de time-to-live van messages te beperken.

Tot slot nog deze opmerking in verband met het gebruik van durable subscribers in combinatie met niet-persistente messages. Stel er is minstens één durable subscriber offline. Dit betekent dat de JMS server de berichten voor deze subscriber op persistente storage moet bewaren, en versturen naar de subscriber zodra deze terug online komt. Aangezien er niet-persistente messages verstuurd worden is het echter mogelijk dat de JMS server faalt alvorens de message werd opgeslaan. Immers, de acknowledgement werd verstuurd van zodra de JMS server de message had ontvangen.

2.8.1.3 De message consumer

Van zodra een message consumer een message onvangt zal deze in AUTO_ACKNOWLEDGE mode meteen een acknkowledgement versturen naar de JMS server. Indien de JMS server om de een of andere reden deze acknowledgement niet ontvangt, zal deze de message als onafgeleverd beschouwen, en mogelijk opnieuw versturen. De situatie kan dus ontstaan waarbij een message consumer een bepaalde message meerdere malen ontvangt, en mogelijk dus ook meerdere malen zal verwerken. Om dit laatste te vermijden zal de JMS server bij het opnieuw versturen van een message de *JMSRedelivered* header op *true* zetten. Jammer genoeg is het wel aan de programmeur om de client applicatie van logica te voorzien om op te merken dat hij deze message reeds ontvangen had, en de opnieuw afgeleverde message niet te verwerken.

2.8.1.4 CLIENT_ACKNOWLEDGE mode

Zoals de naam doet vermoeden wordt deze afleveringmodus gebruikt wanneer de client zelf beslist wanneer er een acknowledgement verstuurd moet worden. De reden hiervoor is als volgt. Indien de AUTO_ACKNOWLEDGE mode gebruikt wordt, wordt de acknowledgement (vreemd genoeg) pas verstuurd nadat de *onMessage* functie van de MessageListener returned. Mogelijk bevat de onMessage functie heel wat rekenkundige bewerkingen, waardoor de JMS server onnodig lang moet wachten op een acknowledgement van de consumer. Gedurende deze tijd blijven er resources gereserveerd voor deze message bij de JMS server. Indien de onMessage functie bovendien een erg lange uitvoertijd heeft, kan de JMS server de message reeds opnieuw versturen. Dit kan vermeden worden door reeds een acknowledgement te versturen tijdens de uitvoering van de onMessage functie, zoals geïllustreerd wordt door onderstaand code-fragment.

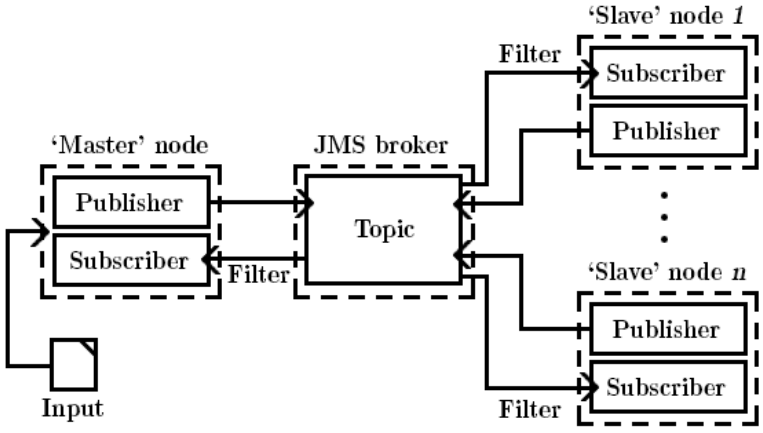
```
1 public void onMessage(Message message) {
2     ...
3     message.acknowledge();
4     ...
5 }
```

2.9 Case study

In de komende paragrafen bespreken we hoe we de gedistribueerde evaluatie van binaire expressiebomen kunnen implementeren met behulp van JMS, net zoals we voor Hadoop gedaan hebben. Meer informatie omtrent deze case study vind je terug in paragraaf 3.5.

2.9.1 Omzetting naar het JMS message passing programmeermodel

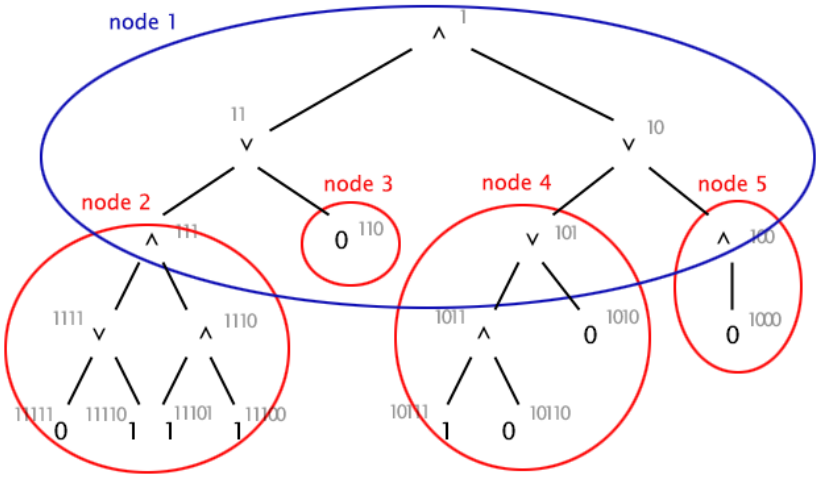
Herinner je dat de moeilijkheid van het implementeren van onze case study in Hadoop de omzetting was naar het Map/Reduce programmeermodel. Eens dit gebeurd was, zorgde Hadoop automatisch voor fault tolerance, load distribution, etc. Bij de Java Message Passing interface zullen we dit soort functionaliteit grootendeels zelf moeten voorzien. Het voordeel is echter dat we een minder restrictief programmeermodel moeten hanteren en 'overbodige' features van Hadoop achterwege kunnen laten. Zo is het falen van een node tijdens de uitvoering van ons programma op de computer cluster erg onwaarschijnlijk, aangezien deze is opgebouwd uit high-end hardware. Het belangrijkste probleem dat we dus moeten oplossen is hoe we de werklast gaan verdelen onder de beschikbare compute nodes, en hoe de onderlinge communicatie zal gebeuren. Figuur 2.12 geeft een overzicht van onze aanpak.



Figuur 2.12: De dataflow van de case study in JMS.

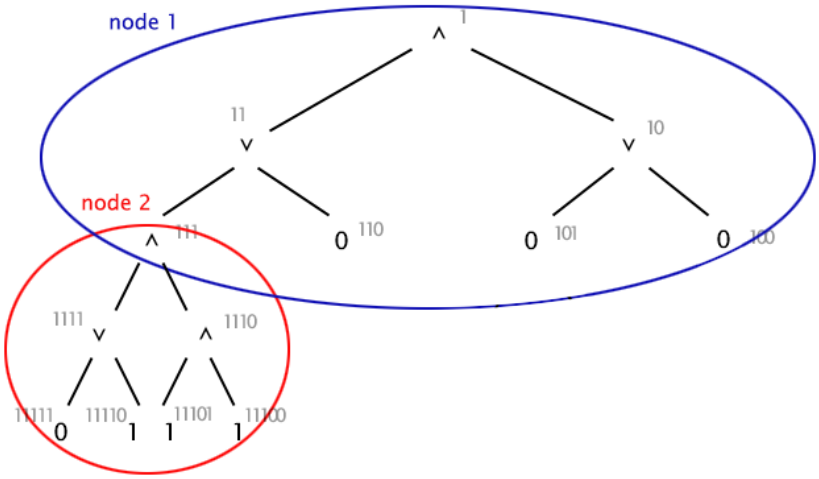
Zoals je kan zien maken we gebruik van één 'master' node die de input file(s) verwerkt. De publisher van deze master node stuurt berichten door naar de JMS broker met als inhoud de key en de value van één knoop uit de binaire expressieboom. Verder zijn er ook n 'slave' nodes, dewelke allemaal een deelboom

verwerken. Door gebruik te maken van message filtering (zie paragraaf 2.7) krijgt een bepaalde node enkel de key/value paren van zijn deelboom te zien. Zo wordt de data trafiek dus tot een minimum beperkt. Wanneer een slave node over voldoende informatie beschikt om zijn deelboom op te lossen zal deze een message versturen met de key en value van de root van zijn deelboom. Deze informatie wordt vervolgens gebruikt om een 'bovenliggende' deelboom op te lossen. We illustreren dit proces verder met het voorbeeld uit Figuur 2.13.



Figuur 2.13: Een voorbeeld van de evaluatie van een binaire expressieboom met behulp van JMS.

Zoals je kan zien in Figuur 2.13 trachten we een boom met diepte 5 op te lossen, waarbij we gebruik maken van 5 'slave' nodes. Het proces begint met de master node die de input data in afzonderlijke messages verstuurt naar de JMS broker (door messages te publiceren naar een bepaald topic). Vervolgens wordt de data door de JMS broker verdeelt onder de verschillende slave nodes. De message selectors zorgen ervoor dat de slave nodes enkel de messages die betrekking hebben op hun deelboom ontvangen. Vervolgens zal elke slave node trachten zijn deelboom op te lossen. Het is duidelijk dat in ons voorbeeld node 1 niet meteen in staat is om zijn deelboom op te lossen aangezien hiervoor informatie nodig is van nodes 2, 3, 4 en 5. Wanneer een van deze laatst genoemde nodes klaar is met de verwerking van zijn deelboom publiceert hij het resultaat naar het topic. Node 1 ontvangt deze messages, update zijn deelboom, en tracht deze opnieuw op te lossen. Dit proces herhaalt zich tot alle deelbomen verwerkt zijn, en dus de oplossing van de volledige boom gekend is. De message selector van de master node zorgt ervoor dat hij deze oplossing ontvangt. Figuur 2.14 geeft tot slot de situatie weer wanneer node 3, 4 en 5 hun deelboom reeds verwerkt hebben, maar node 2 nog niet.



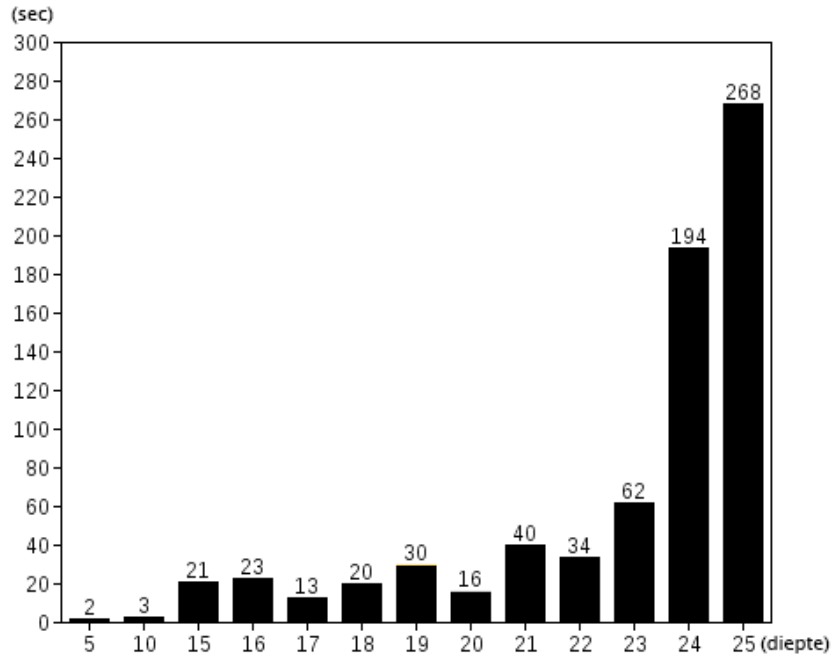
Figuur 2.14: De situatie nadat node 3, 4 en 5 hun deelboom verwerkt hebben.

2.9.2 Optimalisaties

Figuur 2.14 toont een interessante situatie waarbij nog niet alle deelbomen verwerkt zijn, maar node 1 toch al over genoeg informatie beschikt om de volledige boom op te lossen. Immers, de wortel van de boom bevat een AND operatie, en het rechterkind is reeds tot 0 gevalueerd. Bijgevolg zal de volledige boom tot 0 evalueren, ongeacht de waarde van het linkerkind van de wortel. Naast de AND operatie kan ook een dergelijke optimalisatie worden gemaakt bij de evaluatie van een NOT knoop. Indien één van de kinderen reeds naar 1 geëvalueerd werd zal het resultaat van deze knoop ook steeds 1 zijn. In een extreem scenario kan dit soort optimalisaties een enorme tijdwinst opleveren. Een voorbeeld hiervan is wanneer de wortel van een boom een OR operatie voorstelt, het rechterkind de waarde 1 bevat, en het linkerkind een deelboom van diepte 30 bevat.

Merk op dat node 1 in ons voorbeeld niets aan het doen is terwijl de overige slave nodes de oplossing van hun deelbomen berekenen. Het is dus perfect mogelijk om bij het binnenkomen van een nieuwe message te trachten de boom op te lossen. Bijgevolg zal de performantie zelden negatief beïnvloed kunnen worden door deze optimalisaties door te voeren. Het enige scenario waarbij de performantie kan dalen is wanneer node 1 zijn deelboom tracht op te lossen (en dit niet mogelijk blijkt), vlak voor de laatste message binnenkomt met de oplossing van een van de kinderen van node 1. De kans dat de optimalisaties tijdwinst opleveren is echter groter dan het laatst vernoemde scenario, waardoor we deze dus toch geïmplementeerd hebben.

2.9.3 Testresultaten



Figuur 2.15: De uitvoeringstijd van het programma ten opzichte van de diepte van de geëvalueerde expressieboom.

2.9.4 Alternatieve implementaties

Naast de zojuist besproken aanpak hebben we op nog twee alternatieve manieren de case study geïmplementeerd met behulp van JMS. Hiervoor zijn er verschillende redenen. Vooreerst heeft de reeds besproken aanpak de beperking dat er niet aan load balancing gedaan wordt. Het kan zijn dat sommige compute nodes hun data trager verwerken omdat ze bijvoorbeeld opgebouwd zijn uit andere hardware, andere processen uitvoeren die ook resources vergen, etc. Bovendien zijn onze binaire expressiebomen niet perfect binair omwille van de aanwezigheid van de NOT operatie. Het kan dus zijn dat een bepaalde deelboom vele malen kleiner is dan de overige deelbomen. In onze case study is het trouwens zo dat het oplossen van een deelboom vrij triviaal is, aangezien deze bestaat uit simpele AND, OR en NOT operaties. Er zijn voldoende andere scenario's denkbaar die ook op een boomstructuur (of algemener; een graafstructuur) gebaseerd zijn, maar waarbij de uit te voeren berekeningen veel complexer zijn, en bijgevolg de uitvoertijd voor het oplossen van een deelboom meer kan variëren. Dit alles kan er dus voor zorgen dat sommige compute nodes lange tijd *idle* zijn, en er op die manier resources onbenut blijven. Bovendien is de uitvoertijd (te) sterk afhankelijk is van de traagste node. Om deze redenen

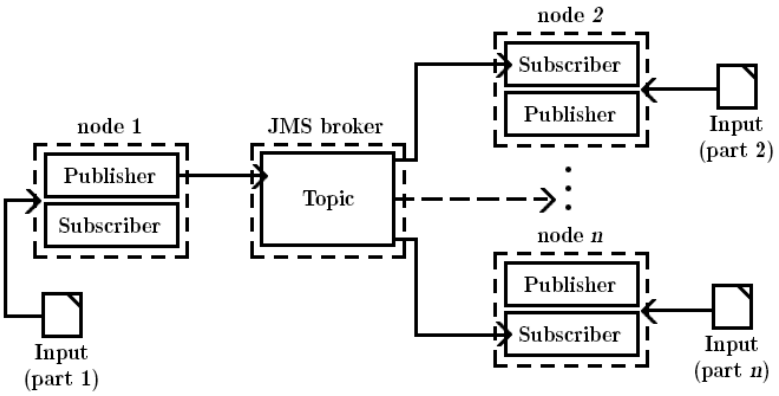
hebben we dus ook gezocht naar een aanpak die ook een zekere vorm van load balancing garandeert.

Verder wordt de input voor onze eerste implementatie door de zogenaamde master node verdeeld onder de beschikbare slave nodes. Er zijn echter scenario's te bedenken waarbij de input data reeds verspreid is. Een voorbeeld hiervan is wanneer er zowel input uit een database als uit log bestanden wordt gebruikt, of wanneer de data reeds verdeeld is over verschillende harde schijven (bijvoorbeeld wanneer er gebruik gemaakt wordt van opslagtechnieken zoals RAID[14]). Bijgevolg hebben we ook een applicatie geschreven waarbij verschillende worker nodes elk over hun eigen deel van de input data beschikken, en vervolgens de boom evalueren zonder tussenkomst van een master. We noemen dit het 'broadcast scenario', dat we in de onderstaande paragrafen zullen bespreken.

2.9.4.1 Broadcast met reeds verdeelde inputdata

In dit scenario veronderstellen we dat de inputdata reeds opgedeeld is, en dat elke compute node een deel van deze input data verwerkt. In de praktijk hebben we gebruik gemaakt van een programma dat het originele input bestand splitst in een aangegeven aantal nieuwe bestanden. Een compute node verwerkt zijn input data als volgt. Eerst tracht hij zo veel mogelijk deelbomen zelf op te lossen. Hierbij moet hij dus minstens de waarde van een knoop plus de waarde van zijn kind(eren) kennen. Merk op dat we enkel deelbomen kunnen oplossen waarvan de bladeren de waarde 0 of 1 hebben. Het heeft dus geen zin om in elke iteratie alle key/value paren waarover een compute node beschikt te overlopen, aangezien deze ook knopen met een AND, OR of NOT operatie bevatten. Concreet pakken we dit aan door een lijst bij te houden van alle knopen waarvan de waarde (0 of 1) reeds gekend is, en telkens er een deelboom wordt opgelost deze lijst aan te passen. Het spreekt voor zich dat dit een veel efficiëntere manier van werken is dan steeds de volledige lijst van key/value paren te overlopen. Bovendien kan het zijn dat de hoeveelheid data die een compute node moet verwerken te groot is om in main memory te bewaren. Door onze manier van werken is het echter voldoende om de lijst met opgeloste knopen in main memory bij te houden, en de overige data op disk op te slaan zonder dat er al te vaak disk reads moeten gebeuren.

Indien een compute node een deelboom tracht op te lossen zijn er drie mogelijke scenario's. Ofwel beschikt hij over zowel de ouder als de kinderen, en kan hij deze deelboom reduceren. Ofwel kent hij de waarde van een kind, maar is de ouder bij een andere compute node ondergebracht. In dat geval broadcast hij de waarde van het kind. Tot slot bestaat er nog de mogelijkheid dat een compute node over een ouder en één kind beschikt, maar het andere kind ontbreekt. In dat geval kan deze deelboom pas gereduceerd worden wanneer een andere compute node de waarde van het ontbrekende kind broadcast. Uiteraard worden er ondertussen zoveel mogelijk andere deelbomen opgelost. Figuur 2.16 geeft een overzicht van de communicatie die plaatsvindt tussen de verschillende compute nodes.



Figuur 2.16: Een overzicht van het broadcast scenario, waarbij wordt weergegeven hoe een key/value paar van node 1 verstuurd wordt naar de overige nodes. Slechts één van deze nodes zal het key/value paar kunnen gebruiken, terwijl alle nodes de message moeten inspecteren.

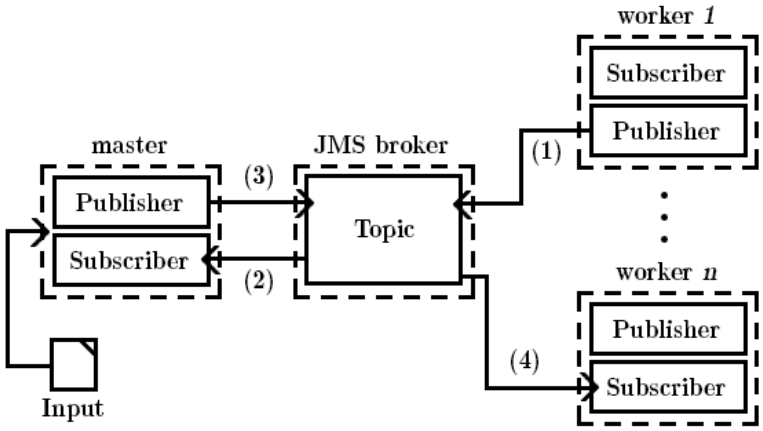
Uit Figuur 2.16 wordt meteen een nadeel van deze aanpak zichtbaar: aangezien niet geweten is bij welke compute node een bepaalde knoop zich bevindt, moeten ze elk alle messages inspecteren en nagaan of de message nut heeft voor hen of niet. We kunnen dus geen gebruik maken van de message filters zoals bij de eerste implementatie het geval was. Het master/slave scenario dat we hieronder bespreken gebruikt een centrale instantie (de master), die een mapping bijhoudt van keys en compute nodes, om op die manier de hoeveelheid messages te verminderen.

2.9.4.2 Master/slave scenario

Het master/slave scenario is ontwikkeld met het oog op load balancing en het vermijden van het broadcasten van messages. Hierbij zijn we als volgt te werk gegaan. De master krijgt een ongeordene file met key/value paren en de grootte van de een 'workload' als input. Met dit laatste bedoelen we het aantal key/value paren dat naar een worker verstuurd wordt wanneer deze vraagt om meer werk. Wanneer de master zo'n workload naar een worker verstuurd houdt deze een mapping bij van de keys die naar de worker verzonden zijn. Op die manier kan de master gericht key/value paren herverdelen wanneer nodig zonder gebruik te maken van een broadcast. We tonen hieronder het protocol dat gebruikt wordt door deze applicatie om de werking verder te verduidelijken.

- **REQUEST** Een REQUEST message wordt door een worker naar de master wanneer deze meer key/value paren wenst. Dit gebeurt wanneer een worker al zijn deelbomen heeft opgelost, of aan het wachten is op key/value paren die door andere workers verwerkt worden.

- **REPLY** Een REPLY is het antwoord dat de master verstuurt op een REQUEST message. Een REPLY message is van het type *mapmessage* en bevat een (instelbaar) aantal key/value paren.
- **NOMOVEKVPAIRS** Dit type message wordt door de master naar een worker verstuurd als antwoord op een REQUEST wanneer er alle key/value paren reeds verdeeld zijn.
- **DISTRIBUTE** Wanneer een worker een key/value paar verwerkt waarvan hij niet over de ouder beschikt, zal hij dit key/value paar naar de master versturen in een DISTRIBUTE message. De master stuurt deze message vervolgens door naar de worker die de ouder bevat. Merk op dat het mogelijk is dat geen enkele worker over de ouder beschikt, aangezien de key/value paren stap voor stap verdeeld worden (in willekeurige volgorde). In zo'n geval zal de master een HOLD message versturen naar de oorspronkelijke worker om aan te geven dat hij het key/value paar moet bijhouden, en het later opnieuw moet proberen. Merk op dat het mogelijk is dat de parent van dit key/value paar zal worden toegekend aan diezelfde worker.
- **HOLD** De werking van een HOLD message kwam reeds aan bod bij de bespreking van het DISTRIBUTE commando.
- **SOLUTION** Een SOLUTION message wordt door een worker verstuurd naar de master (en de overige workers) wanneer deze de waarde van de wortel van de boom heeft berekend.



Figuur 2.17: Een deel van de dataflow binnen het master/slave scenario. De stappen 1 t.e.m. 4 geven aan hoe het DISTRIBUTE commando wordt uitgevoerd.

Figuur 2.17 illustreert een deel van de dataflow binnen het master/slave scenario. Het is duidelijk dat de communicatie tussen twee compute nodes minder bandbreedte vereist dan bij het broadcast scenario aangezien messaging gerichter gebeurd. Hiervoor moet echter wel een 'omweg' gemaakt worden via de master, waardoor mogelijk de delay groter wordt.

Merk op dat we de master niet nodig zouden hebben voor het DISTRIBUTE indien we als volgt te werk zouden gaan. We kunnen een hashfunctie gebruiken voor het verdelen van de key/value paren door te hashen op de key. Op die manier kunnen workers rechtstreeks met elkaar communiceren zonder tussenkomst van de master, wat voor minder bandbreedteverbruik en een kleinere delay zorgt. Uiteraard dient de hashfunctie zo gekozen te worden zodat de keys zo random mogelijk verdeeld worden, en de workload bijgevolg evenredig verdeeld wordt onder de workers. Deze aanpak impliceert wel dat de workers kennis hebben van elkaar (er moet minstens geweten zijn hoeveel workers er zijn), en zorgt dus voor meer coupling tussen de applicaties. In het originele master/slave scenario is dit niet het geval; de master hoeft zelfs niet te weten hoeveel workers er zijn. Bovendien is het perfect mogelijk om tijdens de uitvoering van het programma extra workers toe te voegen, bijvoorbeeld wanneer er nieuwe compute nodes uit de cluster zijn vrijgekomen. Naast meer coupling brengt deze aanpak nog een ander nadeel met zich mee. Aangezien de key/value paren op basis van een hashfunctie verdeeld worden is op voorhand bepaald bij welke worker een key/value paar terecht zal komen. Met andere woorden, er kan niet meer aan load balancing gedaan worden zoals in het oorspronkelijke master/slave scenario. Tot slot merken we op dat deze aanpassing er toe zou leiden dat de applicatie erg gelijkaardig zou worden onze eerste implementatie, waarbij de master de boom 'gesorteerd' onderverdeelde. Bij de eerste implementatie zal het aantal messages dat verstuurd wordt echter aanzienlijk minder zijn, aangezien een worker slechts één deelboom moet oplossen (en dus ook maar 1 key/value paar moet distribueren). In de volgende paragraaf zetten we de voor- en nadelen van elke implementatie nog eens op een rijtje.

2.9.4.3 Voor- en nadelen van de verschillende implementaties

De meest eenvoudige applicatie is duidelijk het eerste scenario, aangezien het aantal messages hier zeer beperkt is (althans na het verdelen van de key/value paren in de setup fase). Er zijn echter ook een aantal beperkingen aan deze aanpak. Vooreerst is er geen load balancing, wat vooral bij deze aanpak een probleem kan zijn. Immers, de deelbomen kunnen behoorlijk verschillen in grootte door het al dan niet frequent voorkomen van de NOT operator. Verder is er, net zoals de overige twee scenario's, geen sprake van fault tolerance. De reden hiervoor is dat we gebruik maken van een computer cluster die opgebouwd is uit high-end hardware, en er dus minder kans is op falen. Het is gezien deze omstandigheden dus verstandiger om een programma simpelweg opnieuw uit te voeren indien een compute node onbeschikbaar wordt, dan om veel tijd te besteden aan het implementeren van fault tolerance mechanismen.

Stel dat we toch gebruik zouden maken van 'commodity hardware', waar-

door de kans op het falen van een node dus reëel wordt, en we een zekere vorm van fault tolerance willen inbouwen. Er zijn verschillende fouten die kunnen optreden. Een voorbeeld hiervan is dat de 'master', al dan niet permanent, onbeschikbaar wordt (niet toepasbaar op het broadcast scenario). Om hiervan te herstellen moet er een backup van de 'state' van de master worden bijgehouden. Dit kan bijvoorbeeld door het bijhouden van een log file die naar persistente storage wordt weggeschreven (op een andere node indien je wil herstellen van het permanent onbeschikbaar worden van de master), of door het bijhouden van een 'secundaire master'. Merk op dat deze termen en methoden zeer bekend klinken, aangezien Hadoop op dezelfde manier te werk gaat! Analoge voorbeelden kunnen gegeven worden voor het onbeschikbaar worden voor een compute node of een data node (dit laatste valt op te lossen met een vorm van data replicatie, zoals gebruikt in HDFS). Het punt dat we met dit voorbeeld willen maken is dan ook dat wanneer we meer functionaliteit wensen zoals fault tolerance, data replicatie, etc. we veel tijd kunnen besparen door gebruik te maken van een framework dat dit reeds voorziet. Het is immers perfect mogelijk om deze functionaliteit in te bouwen in een JMS framework, waarbij er dus geen Map/Reduce programmeermodel of HDFS file systeem gehanteerd wordt. Laat je echter niet misleiden: zo'n JMS framework is geen 'verbeterde' versie van Hadoop. Er gaan immers ook voordelen verloren zoals het automatisch parallellisme (in ruil voor een minder strict programmeermodel).

Naast fault tolerance en load balancing kunnen we ook de hoeveelheid messages die verstuurd worden bij de drie scenario's vergelijken. De eerste implementatie stuurt in de startup fase behoorlijk veel messages (wanneer de key/value paren verdeeld worden), maar nadien worden er nog amper messages verstuurd. Het aantal messages is met andere woorden recht evenredig met de grootte van de input data. Bij het master/slave scenario dient de input data ook verdeeld te worden onder de beschikbare compute nodes, waardoor er minstens even veel messages verstuurd worden als bij het eerste scenario. De key/value paren worden echter willekeurig verdeeld onder de workers, waardoor er na de startup fase nog key/value paren herverdeeld moeten worden. Afhankelijk deze verdeling kan deze aanpak tot aanzienlijk meer messaging leiden. Dit is de prijs die we betalen voor de load balancing. Het voordeel van het broadcast scenario is dat de input reeds verdeeld is onder de compute nodes. Nadien kan er echter aanzienlijk meer netwerktrafiek plaatsvinden, afhankelijk van de verdeling van de key/value paren. Bovendien leidt het toevoegen van compute nodes tot een stijging van het aantal messages, aangezien de kans dat een node over de parent van een key beschikt verkleint. Bovendien ontvangt elke nieuwe node ook elke message die verstuurd wordt.

We eindigen de vergelijking van de verschillende scenario's met de opmerking dat er bij de master/slave implementatie dynamisch workers toegevoegd kunnen worden. Tijdens de uitvoering van het programma kan de gebruiker dus extra worker instanties opstarten wanneer er nieuwe compute nodes vrijkomen op de cluster, om zo de uitvoertijd te verminderen.

2.9.5 Verglijking case study in JMS en Hadoop

De reden waarom we drie verschillende implementaties gemaakt hebben in JMS is om te illustreren dat dit programmeerparadigma veel meer vrijheid biedt dan bijvoorbeeld Map/Reduce. Zoals uit paragraaf 2.9.4.3 duidelijk blijkt heeft de programmeur de keuze hoe gesofisticeerd hij zijn programma wil maken, en of hij functionaliteit zoals load balancing en fault tolerance wil toevoegen. Het nadeel hiervan is uiteraard dat wanneer er veel functionaliteit vereist is, dit ook meer programmeerwerk vergt. Dit is in tegenstelling tot Hadoop, waarbij fault tolerance, load balancing, data replicatie,... 'gratis' zijn. 'Gratis', aangezien je je applicatie wel moet omzetten naar het Map/Reduce programmeermodel, wat mogelijk niet triviaal is en kan leiden tot een minder efficiënte aanpak. Eens deze omzetting gemaakt is, gebeurt de parallellisatie wel automatisch.

Naast een minder strikt programmeermodel biedt JMS nog een voordeel ten opzichte van Hadoop. De leercurve is namelijk een stuk minder steil. Het principe achter de Java Message Service, of achter een ander message passing model, is namelijk zeer simpel en intuïtief. Om echter efficiënte applicaties te ontwikkelen met Hadoop, en je job te fine-tunen, heb je behoorlijk wat kennis nodig van de achterliggende werking van het systeem.

Verder is JMS ook makkelijker te gebruiken in de praktijk. Het enige dat dient te gebeuren voor het uitvoeren van een JMS applicatie op een cluster is het draaien van een JMS broker op één bepaalde compute node. Dit staat in contrast met Hadoop dat gebruik maakt van zijn eigen file systeem (HDFS), dat op elke node (die je wenst te gebruiken) geïnstalleerd moet worden. Vervolgens moet de input data ingeladen worden in het file systeem, en de output data worden uitgelezen na het uitvoeren van de applicatie. Deze redenering gaat er van uit dat je geen dedicated Hadoop cluster gebruikt. Indien dit wel zo is, valt dit argument voor JMS natuurlijk weg.

Eens een gebruiker bekend is met het schrijven van Map/Reduce programma's en de werking van Hadoop, kunnen applicaties wel zeer snel ontwikkeld worden. Zeker indien het ad-hoc analyse van (tekstuele) data betreft. Bovendien zijn Hadoop programma's doorgaans opgebouwd uit minder regels code. Verder verzorgt het framework achterliggend ook reeds veel functionaliteit, waardoor de eigenlijke programmacode overzichterlijker is. Tot slot was het debuggen (naar mijn mening) sneller en gemakkelijker bij Hadoop, vooral in vergelijking met de JMS applicaties die veel messages versturen (waaronder het broadcast en het master/slave scenario).

Hoofdstuk 3

Bloom

3.1 Inleiding

De derde programmeertaal die we beschouwen is de declaratieve taal *Bloom*, en heeft (vrij vanzelfsprekend) als doel de ontwikkeling van gedistribueerde software te vergemakkelijken. Declaratieve programmeertalen drukken logica uit zonder hierbij de 'control flow' van een programma te beschrijven. Dit staat in contrast met imperatieve programmeertalen, die wel de volgorde van individuele expressies of instructies (of dus de control flow) bevatten. Declaratieve programmeertalen trachten dus eerder te beschrijven *wat* een programma precies moet bereiken, in plaats van *hoe* het dit moet doen. De reden waarom er voor een declaratieve taal werd gekozen wordt duidelijk in Paragraaf 3.2.

Het feit dat Bloom een declaratieve taal is doet vermoeden dat ze afgeleid is van een formele basis, wat ook zo is. Dit logisch model noemt *Dedalus*, en is een uitbreiding op *Datalog* waarbij er ook tijds- en ruimtecomponenten werden toegevoegd[24]. Voor onze doeleinden is het voldoende om te weten dat Dedalus een theoretisch model is ontwikkeld voor het makkelijker programmeren en analyseren van gedistribueerde applicaties, dat eigenlijk niet geschikt is als praktische programmeertaal.

Verder is er een prototype beschikbaar van Bloom dat geïmplementeerd werd als een domein-specifieke taal in *Ruby*[15], en de naam *Bud* draagt (wat staat voor *Bloom under development*). Aangezien Bloom voorlopig nog niet veel meer is dan een research project van de universiteit van Berkeley, is dit een zeer tactvolle keuze. Door de implementatie binnen Ruby zijn immers alle Ruby libraries, toolkits, etc. ook bruikbaar binnen Bud. De makers van Bud hebben verder ook plannen om Bloom beschikbaar te maken als domein-specifieke taal binnen andere, meer 'traditionele' programmeertalen[4].

Alvorens we de motivatie achter de ontwikkeling van Bloom, het programmeermodel, en het achterliggend theoretische model bespreken, halen we eerst de meest opmerkelijke eigenschappen van Bloom aan. Vooreerst moedigt de taal 'ongeordend' programmeren aan. De statements van een Bloom programma

kunnen met andere woorden in willekeurige volgorde uitgevoerd worden, in tegenstelling tot traditionele programmeertalen. Deze laatste zijn immers geëvolueerd uit het Von Neumann model, waarbij instructies in een welbepaalde volgorde worden uitgevoerd[25]. Computer clusters (of gedistribueerde systemen in het algemeen) zijn echter van nature uit ongeordend: verschillende compute nodes bevinden zich steeds in een verschillende 'state'. Hiermee bedoelen we dat ze uit andere hardware opgebouwd kunnen zijn, er andere processen (gelijktijdig) op kunnen draaien, hardware (al dan niet tijdelijk) kan falen, etc. Bovendien kan de 'afstand' tusen twee compute nodes erg verschillen naargelang de snelheid van de netwerkverbinding en de beschikbare bandbreedte. Volgens de makers van Bloom is dit één van de redenen waarom de ontwikkeling van gedistribueerde applicaties vaak zo moeizaam gaat: de programmeur moet de brug vormen tussen een geordend programmeermodel en een ongeordende omgeving.

Een andere kenmerkende feature van Bloom is dat het werkt op *collecties* of sets in plaats van de typische scalaire variabelen en structuren van een imperatieve programmeertaal. Andere voorbeelden van talen die inwerken op collecties zijn SQL, en zelfs Map/Reduce. De syntax die Bloom hanteert voor het uitvoeren van operaties op deze collecties heeft naast de eigenschappen die het erft van Ruby dan ook wat weg van SQL en Map/Reduce. Op die manier voelt de syntax van Bloom bekend aan voor programmeurs vertrouwd met deze talen. Eén van de redenen waarom er collecties gebruikt worden is omdat (een deel van) de operaties die hierop kunnen worden uitgevoerd makkelijk te parallelliseren zijn. Merk bovendien op dat collecties ongeordend zijn, wat opnieuw beter past bij de aard van gedistribueerde systemen.

Verder argumenteren de makers ook dat het gebruik van een high-level declaratieve taal leidt tot kortere en overzichtelijkere programmacode. Onze bevindingen hierover zullen we bespreken in de paragrafen over de implementatie van onze case study in Bloom (zie Paragraaf 3.5).

Tot slot bevat de compiler van Bloom ook een analyse techniek die in staat is om *points of order* te detecteren in programmacode. Dit zijn punten van niet-monoticiteit, waar met andere woorden mogelijk extra applicatie logica moet worden ingebouwd om consistentie te garanderen. In de komende paragrafen bespreken we de motivatie achter de ontwikkeling van Bloom, waarbij we vooral kijken naar het nut van de juist vermelde consistentie analyse techniek.

3.2 Motivatie

Eén van de nadelen die we hebben ondervonden bij de case study in JMS is dat indien je gebruik maakt van het message passing model, je zelf moet verifiëren of de consistentie van je applicatie blijft gelden indien de volgorde van messages verandert. Een eenvoudig voorbeeld waarbij de volgorde van de operaties invloed heeft op het resultaat van de uitvoering wordt gegeven in het onderstaand (pseudo-) codefragmentje. Hierbij veronderstellen we dat twee verschillende applicaties (of threads) de waarde van een bepaalde variabele A inlezen, en vervolgens verdubbelen.

```
1 //applicatie 1
2 READ(A);
3 A = 2*A;
4 WRITE(A);
5
6 //applicatie 2
7 READ(A);
8 A = 2*A;
9 WRITE(A);
```

Indien de instructies van beide applicaties in bovenstaande volgorde worden uitgevoerd zal de resulterende waarde van A vier keer zijn oorspronkelijke waarde bedragen. Aangezien we over gedistribueerde applicaties spreken kunnen de instructies echter ook in onderstaande volgorde worden uitgevoerd.

```
1 //applicatie 1
2 READ(A);
3 A = 2*A;
4 //applicatie 2
5 READ(A);
6 A = 2*A;
7 //applicatie 1
8 WRITE(A);
9 //applicatie 2
10 WRITE(A);
```

Na het uitvoeren van de instructies in deze volgorde zal het resultaat van A slechts twee keer zijn oorspronkelijke waarde bedragen. Het punt dat we hiermee willen aantonen is dat de volgorde van instructies of messages erg belangrijk kan zijn bij de uitvoering van gedistribueerde applicaties. Het is duidelijk dat het nagaan van de consistentie van deze applicties dus behoorlijk 'low level' is. Bovendien is dit proces applicatie-specifiek; sommige applicaties zullen gevoeliger zijn voor de volgorde van messages dan andere. Een imperatieve programmeertaal is typisch niet in staat om de programmeur ondersteuning te bieden bij het nagaan van de consistentie van zijn programma, omwille van de aard van imperatieve programmeertalen. Zoals reeds vermeld bevatten deze immers de control flow van een applicatie; ze zeggen ze dus hoe iets moet gebeuren, en niet wat et moet gebeuren. Een direct gevolg hiervan is dat het testen van de consistentie erg moeilijk is. Immers, het kan zijn dat inconsistentie enkel optreedt wanneer messages in een bepaalde volgorde arriveren. Het is duidelijk dat deze volgorde beïnvloedt kan worden door de scheduling van taken op één bepaalde compute node, tragere verbindingen tussen bepaalde nodes, het falen van bepaalde hardware componenten (een harde schijf, een router,...), etc. Bijgevolg is niet alleen het nagaan van de omstandigheden waarin inconsistentie optrad, maar ook het reconstrueren van deze omstandigheden (voor debugging doeleinden) erg moeilijk. De conclusie is duidelijk: het nagaan van consistentie en het debuggen van gedistribueerde applicaties is geen sinecure. Om vervolgens consistentie te garanderen ongeacht de volgorde van messaging is er vaak

ingewikkelde, applicatie-specifieke code nodig (waarin bijvoorbeeld een bepaald protocol wordt vastgelegd), die tot nog meer onderlinge communicatie tussen nodes leidt. Het resultaat hiervan zijn applicaties die moeilijk te begrijpen, testen, en hergebruiken zijn.

Om de zojuist vermelde problemen bij het ontwikkelen van gedistribueerde applicaties te vermijden hebben de makers van Bloom gekozen voor een declaratieve programmeertaal. Immers, bij zo'n taal kan de programmeur aangeven *wat* hij wil bereiken, en zal er achterliggend beslist worden *hoe* dit precies kan gebeuren. Bovendien zijn declaratieve talen typisch gebaseerd op een formele logische basis, waardoor er makkelijker een consistentie analyse gemaakt kan worden. De makers van Bloom stellen echter dat indien een declaratieve taal krachtig genoeg is, er nog steeds inconsistentie kan voorkomen[1]. Omwille van deze reden stellen ze ook een analyse techniek voor waarbij *points of order* in een programma geïdentificeerd kunnen worden. Dit zijn locaties in de programmacode waar er mogelijk extra coördinatie nodig is om consistentie in alle gevallen te verzekeren. Door de combinatie van een declaratieve programmeertaal en deze analyse tool zijn programmeurs in staat om makkelijker inconsistenties op te sporen, wat volgens de makers van Bloom de ontwikkeling van gedistribueerde applicaties vergemakkelijkt. Aangezien de consistentie analyse bovendien op een higher-level gebeurt dan traditionele read/write analyse, kan extra coördinatie in sommige gevallen vermeden worden.

Een opmerking die we bij bovenstaande analyse moeten maken is dat indien je gebruik maakt van Hadoop, je weinig tot geen tijd zal moeten besteden aan het nagaan van de consistentie van je programma. Immers, alle communicatie gebeurt achterliggend door het framework. Dit maakt het testen van Hadoop applicaties dus (deels) eenvoudiger ten opzichte van applicaties die gebruik maken van message passing. Door het simpele Map/Reduce programmeerparadigma is de resulterende code vaak ook overzichtelijker aangezien de belangrijkste bewerkingen zich bevinden in slechts twee functies: de map- en de reduce functie. De moeilijkheid bij Hadoop is echter, zoals reeds vermeld, het schrijven van je applicatie binnen het Map/Reduce programmeermodel (op een efficiënte manier). Verder heeft Hadoop ook een steilere leercurve (zie Paragraaf 2.9.5).

3.3 Programmeermodel

Zoals reeds vermeld wordt de state van een programma of applicatie in Bloom bepaald door een verzameling van ongeordende sets die we *collections* noemen. Een collection bestaat uit een aantal *facts*, vergelijkbaar met bijvoorbeeld de tupels bij een relationele database. De bewerkingen die worden uitgevoerd op collections worden bepaald door een ongeordende set van statements of 'regels'. Hierbij moeten we een heel belangrijke opmerking maken: een statement kan

[1] Merk op dat er ook gedistribueerde systemen bestaan die gebruik maken van een (krachtige) declaratieve taal waarbij er geen inconsistentie kan optreden. Een voorbeeld hiervan zijn parallelle databases. De makers van Bloom stellen echter dat gedistribueerde transactions te kostelijk zijn, en hebben daarom parallelle databases niet opgenomen in hun redenering.

enkel betrekking hebben op data die lokaal is voor een bepaalde node! Een ander belangrijk begrip in het programmeermodel van Bloom is een *timestep*. Elke 'ronde van evaluatie', waarin dus alle statements geëvalueerd worden, wordt een atomische timestep genoemd. Wanneer een statement geëvalueerd wordt kan dit er toe leiden dat er in de huidige of volgende timestep nieuwe facts worden afgeleid (op de lokale node). Een statement kan ook zorgen voor de toevoeging van nieuwe facts op een andere node. Dit gebeurt in een toekomstige, niet-deterministische timestep op die node. Afhankelijk van het soort collection dat gebruikt wordt kunnen facts gedurende één of meerdere timesteps bestaan.

3.3.1 Collections

Een collection heeft veel weg van een tabel in een relationele database. Ze bestaat uit een schema van (benoemde) kolommen, waarbij het mogelijk is dat een subset van deze kolommen een primary key vormt. Een kolom kan van eender welk type beschikbaar in Ruby zijn, en dus ook opnieuw een collection voorstellen. Onderstaand code fragmentje definiëert een collection van het type *table*, dat drie kolommen bevat. Hiervan is de combinatie 'naam' en 'voornaam' de primary key.

```
1 def state
2     table : studenten, ['naam', 'voornaam'],
3       ['studierichting', 'jaar']
4 end
```

Zoals je kan zien wordt niet aangegeven van welk type de kolommen zijn. Dit komt omdat Ruby geen type checking voorziet, en alles in Ruby een object is. Zelfs instanties van types zoals integers zijn objecten in Ruby. Verder merken we ook op dat Bloom collections binnen een 'state' methode gedeclareerd worden. Nu we meer vertrouwd zijn met het begrip collection kunnen we de verschillende types van collections bespreken. Dit wordt getoond in Tabel 3.3.1.

Type	Eigenschappen
table	Een collection waarvan de inhoud (facts) over meerdere timesteps blijft bestaan.
scratch	Een collection waarvan de inhoud slechts voor één timestep bestaat.
channel	Een subtype van een scratch collection dat gebruikt wordt voor de communicatie tussen verschillende nodes. Bevat een *location specifier* dewelke het netwerk adres van een node bevat. De communicatie gebeurt via het (unreliable) UDP protocol.
periodic	Een subtype van een scratch collection een key/value paar bevat. De key is een unieke identifier, terwijl de value een timestamp (de huidige system-clock) bevat. Bij het aanmaken van een periodic collection wordt er een parameter meegegeven die de periode waarmee het framework nieuwe facts genereert aangeeeft.
interface	Een subtype van een scratch collection die gebruikt wordt als interface tussen verschillende modules.

Tabel 3.1: Overzicht van de verschillende collections in Bloom.

We benadrukken nog kort het verband tussen timesteps en verschillende soorten collections. Een timestep bestaat, zoals reeds vermeld, uit het evalueren van de statements in een Bloom programma (wat in willekeurige volgorde kan gebeuren). Scratch collections (en subtypes hiervan) worden leeggemaakt na elke timestep, terwijl de inhoudt van tables blijft voortbestaan tot ze expliciet verwijderd worden. Een nieuwe timestep kan geïnitieerd worden door het voortbestaan van facts in een collection, of het arriveren van nieuwe facts. Dit laatste kan door gebruik te maken van een channel, dat facts ontvangt van andere nodes. Een andere mogelijkheid om een nieuwe timestep te initiëren is door gebruik te maken van de system clock met behulp van een periodic collection.

3.3.2 Operaties

Uiteraard zijn er ook verscheidene operaties beschikbaar in Bloom. Hun betekenis, alsook op welke collections ze toepasbaar zijn wordt getoond in Tabel 3.3.2. Hierbij gebruiken we de afkortingen 'lhs' en 'rhs' voor respectievelijk het linker- en het rechterdeel van het statement dat een bepaalde operator bevat.

Operator	Toepasbaar op	Betekenis
=	scratch	De inhoud van de lhs wordt bepaald door de rhs in de huidige timestep. Merk op dat de lhs niet in de lhs van een ander statement mag voorkomen! Immers, dit zou tot gevolg kunnen hebben dat de Bloom statements niet meer in willekeurige volgorde geëvalueerd kunnen worden.
<=	table, scratch	De inhoud van de rhs wordt toegevoegd aan de inhoud van de lhs in de huidige timestep.
< +	table, scratch	De inhoud van de rhs wordt toegevoegd aan de inhoud van de lhs in de volgende timestep
< −	table	Facts in de rhs worden verwijderd uit de lhs in de volgende timestep.
<∼	channel	Facts in de rhs worden toegevoegd aan de lhs in een niet-deterministische toekomstige timestep op een andere node.

Tabel 3.2: Overzicht van de verschillende operaties in Bloom.

Het gebruik van deze operaties wordt duidelijker in de volgende paragraaf, waarin statements in Bloom besproken worden.

3.3.3 Statements

Statements in Bloom bepalen de inhoud van collecties. Hun syntax is als volgt:

```
1 <collection variabele><operator><collection expressie>
```

Zowel de lhs als de rhs van een statement zijn instanties van de klasse 'BudCollection', dewelke allerlei methoden bevat voor het manipuleren van collections. Zo zijn er bijvoorbeeld map, reduce, group, en verschillende soorten join functies beschikbaar. Verder worden Bloom statements binnen method definities geplaatst die vooraf gegaan worden door het 'declare' keyword, zoals wordt geïllustreerd in onderstaand voorbeeldje.

```
1 def state
2     table : studenten, ['naam', 'voornaam'],
3         ['studierichting', 'jaar']
4     table : afgestudeerden, ['naam', 'voornaam'],
5         ['studierichting', 'jaar']
6 end
7
8 declare
```

```
9  def verwijder_oudstudenten
10     studenten <- afgestudeerden
11 end
```

In dit eenvoudig voorbeeldje worden simpelweg de facts uit de collection 'studenten' verwijderd die ook in de collection 'afgestudeerden' zitten.

3.3.4 Overzicht programmeermodel

Aangezien we redelijk wat nieuwe termen geïntroduceerd hebben in de voorbije paragrafen, vatten we nog even kort het programmeermodel van Bloom samen. De state van een Bloom programma wordt bepaald door een verzameling collections, dewelke ongeordende sets van facts (tupels) zijn. Bloom statements zijn expressies die inwerken op deze collections. Deze statements worden, in een willekeurige volgorde, geëvalueerd door het framework in wat men een timestep noemt. Communicatie tussen nodes gebeurd door het toevoegen van facts aan een channel, wat een bepaald type collection is. Het framework zorgt er achterliggend voor dat facts die toegevoegd worden aan zo'n channel bij de juiste node terecht komen (in een toekomstige, niet-deterministische timestep).

3.4 Hello World

Net zoals bij het Java Message Service hoofdstuk gaan we ook hier een chatapplicatie beschouwen om wat meer inzicht te verkrijgen in het programmeren met Bloom/Bud. Het doel is simpelweg het opzetten van een chat server waarop clients kunnen connecteren, berichten versturen en uiteraard berichten ontvangen. We hebben dus twee applicaties nodig: een chat server en een chat client. We beginnen met de bespreking van de chat server, waarvan de code hieronder getoond wordt.[2]

```
1  require 'rubygems'
2  require 'backports'
3  require 'bud'
4  DEFAULT_ADDR = "localhost:12345"
5
6  class ChatServer
7    include Bud
8
9    state do
10     table :nodelist
11     channel :connect, [:@addr, :client] => [:nick]
12     channel :mcast
13   end
14
```

[2]Merk op dat deze code sterk gebaseerd is op de code uit de 'get started' documentatie op `https://github.com/bloom-lang/bud/blob/master/docs/getstarted.md`

```
15   bloom :communicate do
16     nodelist <= connect { |c| [c.client, c.nick] }
17     mcast <~ (mcast * nodelist).pairs do |m,n|
18       [n.key, m.val]
19     end
20   end
21 end
22
23 ip, port = .split(":")
24 program = ChatServer.new(:ip => ip, :port => port.to_i)
25 program.run_fg
```

Zoals je kan zien bevinden de Bloom statements zich in een Ruby klasse genaamd ChatServer. Vooreerst is er het *state block*, dat de table 'nodelist' en de channels 'connect' en 'mcast' bevat. Merk op dat alle Bloom collections uit 3.3.1 gedefiniëerd moeten worden in dit state block. De nodelist collection bevat simpelweg een lijst van de geconnecteerde chat clients. Zoals je kan zien is er geen schema gespecificeerd bij de nodelist collection, waardoor het default schema voor een table gebruikt wordt dat een key afbeeldt op een value. Het default schema voor een channel beeldt een adres (een string waarde van de vorm ip:poort) af op een value.

Naast het state block is er ook nog het 'communicate' block dat twee Bloom statements bevat. De naam van zo'n *Bloom block* kan wilekeurig gekozen worden, en dient enkel om makkelijker het overzicht te bewaren in je programmacode. Zo kan je statements die verschillende aspecten van je applicatie verzorgen onderbrengen onder verschillende Bloom blocks. Het eerste statement in het block voegt simpelweg clients toe aan de nodelist wanneer deze connecteren. Het tweede statement zorgt ervoor dat messages die arriveren op het mcast channel gemulticast worden naar alle geconnecteerde nodes. Dit gebeurt door het carthesisch product te nemen van de mcast en nodelist collections, en vervolgens de resulterende 'tupels' één voor één te verwerken. We hadden dit statement ook kunnen schrijven zoals het eerste Bloom statement, zoals geïllustreerd wordt in onderstaand codefragmentje.

```
1    mcast <~ (mcast * nodelist).pairs { |m,n|  [n.key, m.
         val] }
```

Deze notatie is echter minder intuïtief, zeker voor gebruikers die niet bekend zijn met Ruby, waardoor we vooral de meer verbose notatie zullen hanteren. We vervolgen de bespreking van de chat applicatie met het overlopen van de code van de chat client, die hieronder getoond wordt.

```
1 require 'rubygems'
2 require 'backports'
3 require 'bud'
4 DEFAULT_ADDR = "localhost:12345"
5
6 class ChatClient
```

```
7   include Bud
8
9   def initialize(nick, server, opts={})
10    @nick = nick
11    @server = server
12    super opts
13  end
14
15  bootstrap do
16    connect <~ [[@server, ip_port, @nick]]
17  end
18
19  bloom do
20    mcast <~ stdio do |s|
21      [@server, [ip_port, @nick, s.line]]
22    end
23
24    stdio <~ mcast do |m|
25      [m.val[1]+ " says: " + m.val[2]]
26    end
27  end
28 end
29
30 program = ChatClient.new(ARGV[0], DEFAULT_ADDR, :stdin =>
       $stdin)
31 program.run_fg
```

Het valt meteen op dat de chat client, in tegenstelling tot de chat server, over een constructor beschikt (de 'initialize' functie). De aandachtige lezer heeft ondertussen opgemerkt dat, ondanks het ontbreken van een constructor bij de server, er toch twee parameters worden meegegeven bij het aanmaken van een Chatserver instantie. Deze parameters worden in een hash gestopt, en meegegeven aan de constructor van de Bud klasse, die impliciet wordt aangeroepen. Deze klasse, waar alle Bud programma's van afgeleid zijn, verzorgt onder andere de netwerk functionaliteit van Bloom. Zo kun je bijvoorbeeld het IP adres en de bijhorende poort meegeven die je applicatie dient te gebruiken. De constructor van de chat client roept expliciet de constructor van de Bud klasse aan met het 'super' commando. Hierbij wordt het 'opts' hash object meegegeven, dat in dit geval aangeeft dat de data die naar standaard input wordt geschreven in de ':stdin' collection gestopt moet worden. Dit is een voorgedefiniëerde scratch collection in Bud, en kan dus gebruikt worden zonder dat deze in een state block moet worden gedefiniëerd.

Naast een constructor beschikt de chat client ook over een *bootstrap block*. Dit block bevat Bloom statements die slechts één keer uitgevoerd worden, voor de uitvoer van de andere Bloom statements (dus voor de aanvang van de eerste timestep). De chat client maakt gebruik van dit bootstrap block om te connecte-

ren met de chat server. Aangezien we reeds enkele Bloom statements besproken hebben vergen de overige statements van de chat client geen verdere uitleg.

3.4.1 Ondervindingen

Zoals je wel hebt gemerkt is er vrij weinig Bud-code nodig voor het implementeren van een chat applicatie wat de makers dan ook als één van de voordelen van Bloom aanhalen. Bovendien zorgt dit voor overzichtelijke en makkelijk begrijpbare programmacode. Jammer genoeg ontbreekt er wel wat cruciale informatie in de 'get started' tutorial van de makers van Bloom. Vooreerst wordt er enkel gebruik gemaakt van de verkorte notatie voor het verwerken van facts uit een collection. Ervaren Ruby programmeurs zijn ongetwijfeld bekend met deze notatie, waarbij er een code block wordt meegegeven aan een functie. Voor zij die niet bekend zijn met Ruby was een woordje uitleg hierbij welkom geweest. Belangrijker is echter het ontbreken van duidelijke, en voldoende informatie omtrent de Bud superklasse. Aangezien deze klasse onder andere voor netwerk functionaliteit zorgt, een toch niet onbelangrijk gegeven bij gedistribueerde applicaties, had deze zeker meer 'in-depth' besproken mogen worden. Momenteel vind je hier enkel informatie over terug in de specificatie van Bud modules, klassen en functies.

Ook de manier waarop een Bud programma wordt gestart, uitgevoerd en gestopt komt amper aan bod in de tutorial (of het FAQ, of de beschikbare papers van Bloom), terwijl dit toch behoorlijk cruciale informatie is wanneer je zelf applicaties in Bud wil schrijven. Uit de specificatie van de Bud klasse blijkt dat er drie verschillende manieren zijn waarop een een Bud programma kan worden uitgevoerd. Vooreerst kan je na het aanmaken van een Bud instantie één of meerdere malen de *tick* functie aanroepen. Bij elke aanroep zal er één timestep worden uitgevoerd, wat uiteraard zeer handig is bij de ontwikkeling en het debuggen van applicaties. Hiernaast kun je een Bud programma ook voor 'onbepaalde tijd' laten uitvoeren door de *runf_fg* of *run_bg* functies aan te roepen. De *run_fg* functie zorgt ervoor dat het programma in een aparte thread in de voorgrond wordt uitgevoerd, waarbij de functie pas returned wanneer er een fout optreedt. De *run_bg* functie voert het programma uit in een thread op de achtergrond. Het is mogelijk om te interageren met de Bud instantie door gebruik de maken van de *sync_do* en *async_do* functies. Tot slot merken we op dat de *stop* functie moet worden aangeroepen (in Ruby) op de Bud instantie om de uitvoering te stoppen en gebruikte resources weer vrij te geven. Persoonlijk lijkt het mij een goede toevoeging om aan Bud de mogelijkheid toe te voegen om een stopconditie te definiëren (met behulp van Bloom statements in plaats van Ruby code).

3.5 Case study

We hebben reeds wat meer inzicht verkregen in Bud door de bespreking van de chat applicatie, en zullen nu verder gaan met de implementatie van onze case

study.

3.5.1 Niet-gedistribueerde implementatie

Bloom verschilt nogal van de vorige twee programmeerparadigma's aangezien het enkel werkt met sets (of collections), alle operaties (of statements) in elke timestep opnieuw geëvalueerd worden, en dit bovendien in een willekeurige volgorde kan gebeuren. Het spreekt voor zich dat dit een totaal andere manier van redeneren en programmeren vergt, waardoor zelfs het implementeren van een niet-gedistribueerde versie van onze caste study uitdagend kan zijn. We bespreken dan ook eerst de manier waarop we binaire expressiebomen kunnen evalueren in Bud door één enkele applicatie, zoals getoond in onderstaand codefragment.

```
1  class Trees
2    include Bud
3
4    state do
5      table :kvpairs, [:key] => [:value]
6      #bij een gebrek aan netwerk messages zorgt de
           periodic table voor het triggeren van nieuwe
           timesteps
7      periodic :keepgoing, 1
8    end
9
10   #constructor
11   def initialize(myself, filePath, opts={})
12     #leest de key/value paren in van file
13     ...
14   end
15
16   bootstrap do
17     #k/v paren in Bud collection plaatsen
18     kvpairs <= @kvarray
19     #dummy kvpair met key 0 wordt gebruikt voor de NOT
           operatie
20     kvpairs << ["0","0"]
21   end
22
23   bloom :solve do
24     #we maken gebruik van een temp collection (die de
           bladeren v/d boom bevat)
25     temp :myleafs <= kvpairs do |kv|
26       if kv.value=="1" or kv.value=="0"
27     [kv.key,kv.value]
28       end
```

```
29    end
30    #we zoeken naar de triplets die een parent en twee
          children bevatten en lossen die indien mogelijk op
31    #de NOT-operatie bevat slechts 1 kind, dit lossen we
          op door een join met het 'dummy' tupel met key 0
32    #indien er een deelboompje kon worden opgelost zorgt
         <+- ervoor dat eerst de oude parent verwijderd
33    #wordt alvorens de nieuwe parent geinsert wordt.
34    kvpairs <+- (kvpairs * myleafs * myleafs).combos do
          |p, l, r|
35      if l.key==p.key+"1" and p.key+"0"==r.key and p.
            value=="AND" and (l.value=="0" or r.value=="0")
36        [p.key, "0"]
37      elsif l.key==p.key+"1" and p.key+"0"==r.key and p.
            value=="AND"
38        [p.key, "1"]
39      elsif l.key==p.key+"1" and p.key+"0"==r.key and p.
            value=="OR" and (l.value=="1" or r.value=="1")
40        [p.key, "1"]
41      elsif l.key==p.key+"1" and p.key+"0"==r.key and p.
            value=="OR"
42        [p.key, "0"]
43      elsif l.key==p.key+"0" and "0"==r.key and p.value
            =="NOT" and l.value=="1"
44        [p.key, "0"]
45      elsif l.key==p.key+"0" and "0"==r.key and p.value
            =="NOT" and l.value=="0"
46        [p.key, "1"]
47      end
48    end
49    #verwijder de key/value paren waarvan er een
          opgeloste parent in de collection zit
50    kvpairs <- (kvpairs * kvpairs).combos do |p, c|
51      if p.key==c.key[0,c.key.length-1] and (p.value=="1"
            or p.value=="0")
52    [c.key, c.value]
53      end
54    end
55    # output de kvpairs
56    stdio <~ kvpairs { |kv| ["..." + kv.key + " " + kv.
          value] }
57    #output het resultaat van de boom (als het bestaat)
58    stdio <~ kvpairs do |kv|
59      if kv.key=="1" and (kv.value=="0" or kv.value=="1")
60    ["Boom succesvol gereduceerd; uitkomst: " + kv.value]
61      end
```

```
62      end
63    end
64
65    bloom :communicate do
66      ...
67    end
68  end
```

Zoals je kan zien maken we gebruik van twee collections: de 'kvpairs' table die vanzelfsprekend de key/value paren bevat, en de 'keepgoing' periodic collection. Deze laatste zorgt er, zoals reeds vermeld, voor dat nieuwe timesteps geïnitieerd worden. Dit gebeurt om de seconde, zoals aangegeven door de parameter 1. In de constructor van de Trees klasse worden de key/value paren ingelezen uit een bestand, en in een Ruby array geplaatst. Vervolgens worden ze in het bootstrap block in een Bud collection geplaatst. Hierbij wordt er ook een 'dummy tupel' toegevoegd dat we gebruiken voor de NOT-operatie, waarover later meer. Verder zijn er twee bloom blocks gedefiniëerd: 'solve' en 'communicate'. Aangezien deze laatste de functionaliteit bevat voor het gedistribueerd verwerken van binaire expressiebomen worden de bijhorende statements in dit code fragment niet getoond. Dit illustreert wel meteen dat functionaliteit makkelijk gescheiden kan worden in Bloom, en applicaties incrementeel kunnen worden ontwikkeld of uitgebreid.

Wanneer we het 'solve' block inspecteren valt meteen op dat we gebruik hebben gemaakt van een *temp* collection 'myleafs' om de bladeren van de boom in te verzamelen. Een temp collection is een soort scratch collection die in een Bloom block wordt gedefiniëerd. Het voordeel van zo'n temp collection is dat deze wordt ingevuld (of geëvalueerd) voor de overige statements. Op die manier heb je toch een zekere orde die aanwezig is in je programma (immers, Bloom statements worden in een willekeurige volgorde uitgevoerd). In de praktijk zal dit vaak erg nuttig blijken. De reden waarom we de temp collection nog niet eerder vermeld hebben is omdat deze ook niet voorkomt in papers over Bloom, de 'get started' tutorial, of de beschikbare voorbeelden. Deze informatie is terug te vinden op het 'Bud Cheat Sheet', dat trouwens nog allerlei andere nuttige informatie bevat, waaronder de *upsert* operatie. De syntax van deze operatie wordt getoond in onderstaand codefragmentje.

```
1 lhs <+- rhs
```

Voor elk fact dat geproduceerd wordt door de rhs zal de upsert operatie nagaan of er een fact is in de lhs met dezelfde key. Indien dit zo is wordt deze verwijderd alvorens het nieuwe fact wordt toegevoegd. Op het eerste zicht lijkt dit misschien een verkorte notatie voor twee Bloom statements: eentje waarbij er facts met een bepaalde key worden verwijderd, en een statement waarbij facts met een bepaalde key worden toegevoegd. Dit is echter niet het geval, aangezien statements in wilekeurige volgorde worden uitgevoerd. Het is dus perfect mogelijk dat eerst het statement dat nieuwe facts wil toevoegen eerst zal worden uitgevoerd. Dit zal leiden tot een foutmelding, aangezien de facts met dezelfde

key zich nog in de collection bevinden. Bijgevolg is deze upsert operatie in de praktijk vaak erg nuttig. Vandaar is het ook opmerkelijk dat ze niet duidelijker vermeld wordt door de makers van Bloom.

We hebben dus reeds de leafnodes verzamelt in de temp collection 'myleafs'. Door het carthesisch product te nemen van deze collection met zichzelf en de kvpairs table bekomen te tripelts van key/value paren. Per triplet gaan we na of het fact uit de kvpairs collection de ouder is van de twee facts uit myleafs. Als dat het geval is kunnen we de waarde van de ouder bepalen, het parent fact uit de kvpairs table verwijderen en vervangen door een fact met de opgeloste waarde voor die ouder. Een key/value paar dat de NOT-operatie bevat verdient hierbij speciale aandacht. Immers, deze heeft maar één kind, en kan dus geen 'normaal' triplet vormen. Herinner je dat we een dummy fact hebben toegevoegd met key 0 (dat dus nooit kan voorkomen in een boom). Wanneer een key/value paar met de NOT-operatie een triplet vormt met zijn (enig) kind en het dummy fact lossen we dit deelboompje op. Het resultaat van het volledige statement is dat de boom één niveau kleiner wordt.

Merk op dat we met het upsert statement de ouders wel verwerkt hebben, maar de kinderen zich nog steeds ongewijzigd in de kvpairs collection bevinden. Deze moeten uiteraard ook verwijderd worden, waarvoor we het carthesisch product nemen van de kvpairs collection met zichzelf. Hierbij gaan we per gegenereerd paar facts na of het om een ouder en zijn kind gaat, en of de ouders reeds is opgelost. Indien dit het geval is kunnen we het kind verwijderen uit de kvpairs table. De twee laatste statements uit het solve block printen de tussenresultaten en de uiteindelijke oplossing van de boom uit.

Het is duidelijk dat de gebruikte combinatie van Bud statements er voor zal zorgen dat dat boom in elke timestep verder gereduceerd zal worden, tot de kvpairs collection enkel nog de wortel van de boom bevat (en het dummy fact). Bovendien is de volgorde waarin de statements worden uitgevoerd irrelevant voor het eindresultaat.

3.5.2 Gedistribueerde implementatie

Uiteraard hebben we ook een gedistribueerde versie van ons programma gemaakt. Het mooie hieraan is dat het gewoon verder bouwt op de basis die we reeds hebben beschreven in de vorige paragrafen. Herinner je uit Paragraaf 2.9.4 dat we met behulp van JMS twee scenario's hadden geïmplementeerd: een master/slave scenario, en een broadcast scenario met reeds verdeelde input data. Beide manieren van werken zijn ook perfect toepasbaar op het programmeermodel van Bloom, dat ook gewoon op message passing gebaseerd is als communicatiemiddel tussen twee compute nodes. We hebben echter enkel het broadcast scenario geïmplementeerd in Bud, aangezien de voor- en nadelen van beide scenario's reeds aan bod gekomen zijn bij de implementaties in JMS. Opnieuw deze aanpakken vergelijken in Bud zou dan ook tot weinig nieuwe inzichten kunnen leiden. Onderstaand codefragment toont onze aanpak voor de gedistribueerde versie van onze applicatie.

```
class Trees
  include Bud

  state do
    ...
    #bevat een lijst van de overige compute nodes
    table :nodelist, [:node] => [:id]
    #channel waarop nodes kvpairs multicasten
    channel :distribute, [:@node, :key] => [:value]
    #bevat de kvpairs die gedistribueerd moeten worden
    table :lostnodes, [:key] => [:value]

  end

  #constructor
  def initialize(myself, filePath, opts={})
    #lees de key/value paren in van file
      ...
    #lees de ip+port combinaties in van de andere compute
        nodes
      ....
  end

  bootstrap do
    ...
    nodelist <= @nodearray
  end

  bloom :solve do
    ...
  end

  bloom :communicate do
    temp :leafnodes <= kvpairs do |kv|
      if (kv.value=="1" or kv.value=="0") and kv.key!="0"
          and kv.key!="1"
        [kv.key, kv.value]
      end
    end
    #'blijvers' zijn leafnodes waarvan we over de parent
       beschikken
    temp :blijvers <= (kvpairs * leafnodes).combos do |kv
        , l|
      if kv.key==l.key[0,l.key.length-1]
        [l.key,l.value]
      end
```

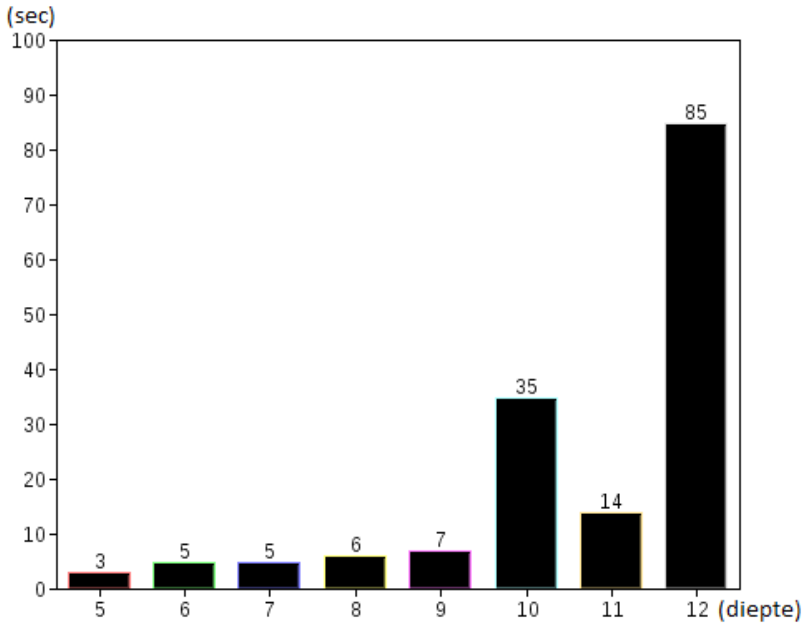
```
43      end
44      #de leafnodes die niet in 'blijvers' zitten gaan we
            multicasten
45      lostnodes <= leafnodes.notin(blijvers)
46      distribute <~ (nodelist * lostnodes).combos do |n, kv
            |
47        [n.node, kv.key, kv.value]
48      end
49      #verwijder de gedistribueerde nodes uit kvpairs
50      kvpairs <- lostnodes
51      #nieuwe nodes worden toegevoegd indien wij over de
            parent beschikken
52      temp :newnodes <= (distribute * kvpairs).combos do |d
            , kv|
53        if d.key[0,d.key.length-1]==kv.key and (kv.value
              !="0" or kv.value!="1")
54          [d.key,d.value]
55        end
56      end
57      kvpairs <= newnodes
58      #distribueer de nodes die wij ontvangen hebben en
            waar we de parent van hebben
59      #dit is een soort ACK message
60      distribute <~ (nodelist * newnodes).combos do |n, kv|
61        [n.node, kv[0], kv[1]]
62      end
63      #indien een andere node ons een kvpair heeft gestuurd
            : verwijderen uit lostnodes
64      #dit is dus een soort aankomende ACK message
65      lostnodes <- distribute { |kv| [kv.key,kv.value] }
66    end
67  end
```

Aangezien er reeds vrij veel commentaar is aangebracht in de code gaan we enkel de algemene aanpak overlopen. Vooreerst is het duidelijk dat een compute node over een lijst van de overige nodes dient te beschikken. Hiervoor maken we gebruik van een soort configuratiefile die een lijst IP adressen en bijhorende poorten bevat, waarvan de inhoud in het 'nodelist' channel geplaatst wordt. Verder is het state block ook uitgebreid met een channel waarop key/value paren gedistribueerd kunnen worden. Tot slot is er ook een table 'lostnodes' toegevoegd. Deze zal gedurende de uitvoering van het programma bladeren (dus key/value paren waarvan de waarde gelijk is aan 1 of 0) bevatten waarvan de ouder zich op een andere compute node bevindt. Dit zijn met andere woorden de key/value paren die gedistribueerd moeten worden. Merk op dat we een table gebruiken en geen scratch. Hiervoor zijn twee redenen. Vooreerst gebruikt Bloom achterliggend het unreliable protocol UDP. Het zou dus kunnen dat een

message verloren gaat. Dit zou tot gevolg hebben dat de boom niet geëvalueerd kan worden en het programma zal vastlopen. Door de te distribueren key/value paren in een table bij te houden en ze elke timestep opnieuw te versturen wordt dit voorkokmen. Wanneer een compute node een key/value paar ontvangt dat voor hem bestemd is zal hij een soort 'ACK message' versturen om aan te geven dat de knoop goed ontvangen is. De compute node die key/value paar heeft gemulticast zal op zijn beurt (bij het ontvangen van deze ACK message) de knoop verwijderen uit de lostnodes table. Er is echter nog een tweede reden voor het meermaals distribueren van key/value paren. Wanneer onze applicatie gestart wordt op meerdere compute nodes zullen er ongetwijfeld bepaalde nodes eerder beginnen aan het evalueren van hun statements, en mogelijk key/value paren multicasten voor de andere compute nodes geïnitialiseerd zijn. Het gevolg hiervan is dat bepaalde messages verloren gaan aangezien er nog niet op het aangegeven adres geluisterd wordt door de applicatie. Ook dit probleem wordt dus verholpen door de implementatie van ons 'reliable protocol'. Er kan natuurlijk ook geargumenteerd worden dat het veelvuldig multicasten van key/value paren overhead met zich meebrengt, de kans op verloren messages relatief klein is, en het laatst vermelde probleem ook op andere manieren verholpen kan worden. Zo zou je bijvoorbeeld een soort opstartfase kunnen implementeren waarbij de compute nodes onderling communiceren om aan te geven dat ze allemaal aan het luisteren zijn op hun adres en poort, en de evaluatie van de boom kan beginnen. Aangezien statements in Bloom in willekeurige volgorde en bovendien in elke timestep geëvalueerd worden is dit echter geen sinecure om te implementeren.

3.5.3 Testresultaten

We hebben onze gedistribueerde applicatie getest door zowel één, twee en drie instanties van het programma een expressieboom te laten evalueren op één enkele compute node. Hierbij gaf het scenario waarbij twee instanties samen een boom evalueren de beste resultaten. Deze resultaten worden getoond in Figuur 3.1.



Figuur 3.1: De uitvoeringstijd van de applicatie ten opzichte van de diepte van de geëvalueerde binaire expressieboom.

Zoals je kan zien is de implementatie in Bud aanzienlijk minder performant dan de implementaties in Hadoop of JMS. De verklaring hiervan vinden we in de manier waarop deelboompjes worden opgelost. Herinner je dat we hiervoor het carthesisch product nemen van alle key/value paren met twee keer de set van alle bladeren van de boom. Stel dat we over een boom van 1.000 knopen beschikken. Het carthesisch product heeft dan een grootte van (ongeveer) $1.000 * 500 * 500 = 250.000.000$ tupels. Voor elk van deze tupels moet worden nagegaan of het om een combinatie van een ouder met zijn twee kinderen gaat. Uiteraard is dit verre van performant. Het zou handiger zijn moesten we bijvoorbeeld één of meerdere joins kunnen gebruiken om tot dit resultaat te komen, aangezien dit ongetwijfeld performanter geïmplementeerd is in Bud. Helaas laat ons probleem dit niet toe aangezien we bijvoorbeeld niet kunnen joinen op de key waarde van nodes. Een mogelijkheid is om een extra veld toe te voegen aan de leafnodes collection dat de ouder van een key/value paar bevat. Op die manier kunnen we dit extra veld gebruiken om te joinen met de kvpairs collection. Afhankelijk van de manier waarop joins achterliggend worden berekend in Bud zou dit performanter kunnen zijn. Merk echter op dat het extra veld geen key is voor de leafnodes collection, en het dus onwaarschijnlijk is dat het opzoeken van een bepaalde waarde voor dit veld in constante of logaritmische tijd gebeurt. Zij die bekend zijn met relationele database management systemen krijgen denken nu ongetwijfeld aan het plaatsen van een index op dit extra

veld voor snelle opzoekingen toe te laten. Hierbij zijn echter twee problemen. Vooreerst wordt dit soort functionaliteit niet ondersteund door Bud. Bovendien wordt de leafnodes collection in elke timestep opnieuw berekend, wat impliceert dat ook de index in elke timestep opnieuw zou moeten worden opgebouwd. We concluderen dus dat we moeilijk een efficiëntere oplossing kunnen bekomen dan de aanpak met het carthesisch product.

3.6 Voor- en nadelen

Zoals je hebt gemerkt is het uiteindelijke programma behoorlijk eenvoudig en leesbaar. Jammer genoeg is het schrijven van een applicatie in Bud niet zo eenvoudig als het resultaat. Eén van de redenen hiervoor hebben we reeds aangehaald, namelijk de gebrekkige documentatie. Hiernaast is ook het werken met collections en de 'ongeordende' programmeerstijl een hele aanpassing voor de meeste programmeurs. Verder is het praktisch nut van deze ongeordende programmeerstijl soms twijfelachtig. Indien twee Bloom statements betrekking hebben op verschillende collections biedt dit uiteraard het voordeel dat ze in parallel door twee verschillende threads geëvalueerd kunnen worden. Jammer genoeg lijkt dit momenteel nog niet het geval te zijn in de huidige versie van Bud. Indien twee statements echter betrekking hebben op dezelfde collections kunnen ze echter niet meer in parallel worden uitgevoerd, of toch zeker niet in alle gevallen. In zo'n geval zou het handig zijn indien de programmeur kon aangeven dat deze twee statements steeds in een bepaalde volgorde moeten worden uitgevoerd, aangezien dit vaak tot eenvoudigere applicatie logica leidt. We merken trouwens op dat de makers van Bloom meermaals aanhalen dat de mogelijkheid bestaat om 'points of order' te definiëren en op die manier coördinatie logica in te bouwen in een Bud applicatie. Hoe dit in de praktijk gebeurt wordt echter nergens vermeld. Mogelijk wordt hiermee verwezen naar het gebruik van temp collections, die ingevuld worden voor de overige Bloom statements. Desalniettemin zou het uitermate handig zijn om meerdere Bloom statements te kunnen groeperen in een soort speciaal block dat steeds in dezelfde volgorde wordt uitgevoerd.

Verder maken we ook nog de opmerking dat programmeren in Bloom gelijkenissen vertoont met het programmeren in SQL, aangezien beide op een collections/relaties inwerken. Voor programmeurs met kennis van SQL kan dit de drempel naar Bloom verlagen. Bovendien kunnen de makers van Bloom inspiratie halen uit SQL (of database management systemen in het algemeen) om hun eigen taal uit te breiden en performanter te maken. Een voorbeeld hiervan is het ondersteunen van een index op een bepaald attribuut.

Een ander knelpunt van Bloom, dat opnieuw zijn oorpsrong vindt in de gebrekkige documentatie, is dat er weinig informatie beschikbaar is over de achterliggende implementatie van de beschikbare operaties. Op die manier is het voor de programmeur moeilijk om na te gaan hoe performant zijn programma is, waar de bottlenecks zich bevinden, en of/hoe dit eventueel verholpen kan worden.

We hebben ons in de voorbije paragrafen vooral negatief uitgelaten over Bloom. Zoals je gemerkt hebt ligt dit echter vooral aan het feit dat Bloom een vrij jong researchproject is, en er dus nog volop ruimte voor verbetering is. Het achterliggende idee is echter zeer goed gevonden: afstappen van programmeermodellen die gebaseerd zijn op de Von Neumann architectuur, aangezien dit niet aansluit bij de realiteit van gedistribueerd programmeren. Of er een commerciële toekomst is weggelegd voor een taal zoals Bloom is onzeker, maar het verder verkennen van hun denkpiste waarbij ongeordend gedistribueerd programmeren centraal staat is zeker een must.

Conclusie

We hebben in deze thesis drie verschillende programmeerparadigma's voor het schrijven van gedistribueerde applicaties uitvoerig besproken en getest. Vooreerst hebben we het Hadoop framework besproken, dat bestaat uit het Map/-Reduce programmeermodel en het gedistribueerde bestandssysteem HDFS. Vervolgens hebben we een hoofdstuk gewijd aan de Java Message Service, dat vergelijkbaar is met andere message passing paradigma's zoals de Message Passing Interface. Tot slot hebben we nog de declaratieve programmeertaal Bloom besproken, dewelke uitgaat van een ongeordende programmeerstijl. De communicatie tussen compute nodes wordt in Bloom eveneens verzorgt door message passing. In de komende paragrafen zullen we nog kort de voor- en nadelen van elke aanpak op een rijtje zetten en hieruit onze conclusies trekken.

3.7 Hadoop

Vooreerst hebben we gekeken naar het Map/Reduce programmeermodel en het bijhorende gedistribueerde bestandssysteem HDFS. Aangezien een goede kennis van de achterliggende werking van het framework een vereiste is voor het schrijven van performantie applicaties in Hadoop hebben we hier best veel tijd aan besteed. We kunnen dan ook stellen dat de leercurve voor Hadoop relatief steil is. Zeker als je er vanuit gaat dat de gebruiker zelf een Hadoop distributie wil installeren op een computer cluster. Ook de omzetting van een probleemstelling naar een algoritme voor het Map/Reduce programmeermodel vergt een zekere inspanning van een programmeur die weinig bekend is met Hadoop. Door het vrij rigide programmeermodel kan er bovendien een zekere overhead optreden bij de omzetting naar Map/Reduce, bijvoorbeeld omdat er meerdere Map/Reduce jobs nodig zijn voor het oplossen van een probleem. Immers, de resultaten van een Map/Reduce job die als input gebruikt worden voor de volgende job worden eerst (in drievoud) naar HDFS geschreven. Het moet echter wel gezegd worden dat Hadoop vooral ontwikkeld is voor de ad-hoc analyse van grote hoeveelheden data (bijvoorbeeld bestanden met logging informatie); een noemer waaronder onze case study moeilijk valt. Eens de zojuist vermelde hindernissen overkomen zijn biedt Hadoop echter heel wat voordelen. Zo zorgt het framework zelf voor de communicatie tussen compute nodes en is er functionaliteit zoals fault tolerance, data replicatie, load balancing, etc. voorzien. Indien je, zoals de

makers van Hadoop veronderstellen, gebruik maakt van commodity hardware zijn deze features zeker een must. Indien je echter over high-end hardware beschikt bestaat de kans dat je geen nood hebt aan al deze functionaliteit, en dat ze zelfs nefast is voor de uitvoertijd van je applicatie. Het immers spreekt voor zich dat deze functionaliteiten ook een zekere overhead met zich meebrengen. Een belangrijke eigenschap van HDFS is de automatische replicatie van data. Dit biedt niet alleen het voordeel dat data niet onbeschikbaar wordt of verloren gaat wanneer een compute node onbeschikbaar wordt; het zorgt er ook voor dat berekeningen zo dicht mogelijk bij de data kunnen worden uitgevoerd. In het optimale geval gebeuren de bewerkingen zelfs op dezelfde node als waar de data zich bevindt. Deze functionaliteit komt 'gratis' bij het gebruik van Hadoop. Hiermee bedoelen we dat je bij de overige twee programmeerparadigma's er zelf voor zal moeten zorgen dat je data bij je compute nodes geraakt. We concluderen dus dat het Hadoop framework zeer veel functionaliteit omvat die de programmeur niet zelf moet voorzien, mits hij in staat is om zijn probleem op te lossen met behulp van het Map/Reduce programmeermodel.

3.7.1 Pig

We hebben ook het framework Pig besproken, dat bestaat uit de programmeertaal Pig Latin en het achterliggend framework dat Pig Latin programma's omzet in Map/Reduce jobs. Pig zorgt voor een abstractielaag bovenop Map/Reduce door de toevoeging van nieuwe datastructuren en high-level operaties. Op die manier probeert het een oplossing te bieden voor het restrictieve en low-level programmeermodel van Map/Reduce. Net zoals SQL of Bloom werkt Pig op sets of collecties van data, en beschrijft het de dataflow van een programma. Dit zorgt voor leesbaarde programmacode, waardoor applicaties makkelijker onderhoudbaar en uitbreidbaar zijn. Bovendien biedt Pig de mogelijkheid om automatisch testdata te genereren die compact en toch zo volledig mogelijk is. We dienen echter wel de opmerking te maken dat applicaties die ontwikkeld worden met behulp van Pig mogelijk minder performant zijn dan wanneer ze in onmiddelijk in Map/Reduce geschreven worden. Dit hangt echter van het specifieke probleem af. Verder is het niet ondenkbaar dat de door Pig gegenereerde Map/Reduce code efficiënter bepaalde operaties implementeert dan wanneer je ze zelf zou schrijven (denk bijvoorbeeld aan join operaties). We concluderen dus dat Pig een aantal voordelen te bieden heeft ten opzichte van Map/Reduce, en dus een handige uitbreiding kan vormen voor een Hadoop cluster.

3.8 Java Message Service

Het tweede programmeermodel dat we besproken hebben steunt op message passing als communicatiemiddel tussen verschillende compute nodes. De Message Passing Interface is waarschijnlijk de meest bekende standaard behorende tot dit programmeermodel. We hebben echter gekozen voor de Java Message Service, die gelijkaardige functionaliteit aanbiedt. In de praktijk hebben we gebruik

gemaakt van het open source ActiveMQ dat deze JMS standaard implementeert.

Het belangrijkste verschil tussen JMS (of message passing in het algemeen) en Hadoop/Bloom is dat het programmeermodel veel vrijer is. Het toepassingsgebied is bijgevolg ook het grootst bij deze aanpak. Bovendien kan op deze manier vaak de efficiënste oplossing bekomen worden. Doordat het onderliggend principe van JMS erg eenvoudig is, is ook de tijd om vertrouwd te raken met het systeem erg klein, zeker in vergelijking met Hadoop.

Zoals je weet maakt JMS gebruik van een centrale instantie die een JMS broker genoemd wordt. Deze centrale instantie maakt het makkelijk om bijvoorbeeld functionaliteit zoals load balancing en fault tolerance in te bouwen. Het nadeel is echter dat de broker een single point of failure vormt. Dit valt te verhelpen door gebruik te maken van een cluster van JMS brokers, wat echter de complexiteit van het systeem verhoogt. Indien je nog meer functionaliteit wenst toe te voegen aan je applicatie, zoals bijvoorbeeld data replicatie, vergt dit wel een extra inspanning van de programmeur. Dit staat in tegenstelling tot het gebruik van Hadoop, waar al deze functionaliteit reeds omvat zit in het framework. Bovendien zorgt Hadoop automatisch voor de verdeling van de inputdata. Bij JMS dient hiervoor extra applicatielogica te worden ingebouwd. Verder passeren alle messages in JMS langs de broker, wat voor overhead kan zorgen. Dit is zeker het geval wanneer er grote hoeveelheden inputdata simpelweg verspreid moeten worden over de beschikbare compute nodes.

3.9 Bloom

Tot slot hebben we nog de decleratieve programmeertaal Bloom besproken, waarbij ongeordend programmeren centraal staat. Het vernieuwende idee achter Bloom is dat het afstapt van programmeertalen die gebaseerd zijn op de (geordende) Von Neumann architectuur, om zo beter aansluiting te vinden bij de realiteit van gedistribueerde applicaties. In de praktijk hebben we gebruik gemaakt van Bud, een implementatie van Bloom volledig ingebed in Ruby. Zoals we in Paragraaf 3.6 reeds hebben aangehaald zijn Bud applicaties makkelijk leesbaar, uitbreidbaar, opdeelbaar en typisch ook erg kort in vergelijking met gelijkaardige applicaties in Hadoop of JMS. Eén van de moeilijkheden bij Bloom volgt echter uit de ongeordende programmeerstijl, waardoor de programmeur voor alle statements moet nagaan of ze in willekeurige volgorde uitvoerbaar zijn. Daarnaast is Bloom ook een vrij jong researchproject, waardoor onder andere de beschikbare documentatie nog niet helemaal op punt staat. Verder is er ook ruimte voor uitbreidingen aan de taal, bijvoorbeeld om toch een volgorde af te dwingen voor bepaalde statements. Het heeft immers geen zin dat twee statements die op dezelfde collections inwerken in willekeurige volgorde kunnen worden uitgevoerd, aangezien ze toch niet (makkelijk) parallelliseerbaar zijn (zie opnieuw Paragraaf 3.6). Tot slot is er nog het groot verschil in performantie tussen onze applicatie in Bloom en de versie in JMS/Hadoop. Dit is deels te wijten aan het feit dat de beschikbare operaties in Bloom nog relatief beperkt zijn, en ze achterliggend mogelijk nog niet erg efficiënt geïmplementeerd zijn.

We concluderen dat Bloom een ‘work in progress’ is, en dat zeker het achterliggende idee van ongeordend programmeren de moeite waard is om verder te onderzoeken in de context van gedistribueerd programmeren.

Bibliografie

[1] Activemq. http://activemq.apache.org/.

[2] Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] Apache sqoop. http://sqoop.apache.org/.

[4] Bloom programming language. http://www.bloom-lang.net/faq/.

[5] Chapter 6: Tuning your mapreduce jobs. http://www.odbms.org/download/Pro%20Hadoop%20Ch.%206.pdf.

[6] Foldoc mpi. http://foldoc.org/mpi.

[7] Intel distribution for apache hadoop software. http://hadoop.intel.com/.

[8] Intel mpi. http://software.intel.com/en-us/intel-mpi-library.

[9] Introduction to the corba naming service.

[10] Java message service api overview. http://www.oracle.com/technetwork/java/overview-137943.html.

[11] The jndi tutorial. http://docs.oracle.com/javase/jndi/tutorial/index.html.

[12] Open mpi. http://www.open-mpi.org/.

[13] O'reilly jms examples. http://examples.oreilly.com/9780596522056/.

[14] Redundant array of independent disks. http://en.wikipedia.org/wiki/RAID.

[15] Ruby, a programmer's best friend. http://www.ruby-lang.org/en/.

[16] Under the hood: Scheduling mapreduce jobs more efficiently with corona. https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920.

[17] Mirek Riedewald Alper Okcan. Processing theta-joins using mapreduce. *SIGMOD*, 2011.

[18] Alexander Rasin Daniel Abadi David DeWitt Samuel Madden Michael Stonebraker Andrew Pavlo, Erik Paulson. A comparison of apapproach to large-scale data analysis. 2009.

[19] Byran Carpenter. mpijava. `http://www.hpjava.org/mpiJava.html`.

[20] Utkarsh Srivastava Ravi Kumar Andrew Tomkins Christopher Olston, Benjamin Reed. Pig latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

[21] Timothy A. Howes. The lightweight directory access protocol: X.500 lite. 1995.

[22] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. 2004.

[23] David A. Chappell Mark Richards, Richard Monson-Haefel. *Java Message Service*. O'REILLY, 2009.

[24] Neil Conway Joseph Hellerstein David Maier Russell Sears Peter Alvaro, William Marczak. Dedalus: Datalog in time and space. 2009.

[25] H. Norton Riley. The von neumann architecture of computer systems. 1987.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Programmeertalen voor computerclusters

Richting: **master in de informatica-databases**
Jaar: **2013**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik 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 mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Smets, Tom

Datum: **7/06/2013**