Parallel rekenen met Big Data technieken

Parallellisatie van Microsoft Windows applicaties
Use-case ALS

D.B.R. van Dam

26 nov. 15

ORTEC FINANCE

VU VRIJE UNIVERSITEIT AMSTERDAM

Parallel rekenen met Big Data technieken

Parallellisatie van Microsoft Windows applicaties

Use-case ALS

D.B.R. van Dam

26 nov. 15

Vrije Universiteit Amsterdam
Faculteit der Exacte Wetenschappen

Dr. Sandjai Bhulai

Dr. Rutger Hofman

Ortec Finance
Development/Research

Drs. Peter van't Hoff

Joris Cramwinckel MSc

Dankwoord

Mijn dank gaat uit naar mijn begeleiders Dr. Sandjai Bhulai en Dr. Rutger Hofman van de Vrije Universiteit Amsterdam voor hun steun en begeleiding bij het schrijven van dit verslag. Tevens wil ik voor de steun vanuit Ortec Finance Peter van ’t Hoff en Joris Cramwinckel bedanken. Tot slot wil ik mijn ouders bedanken voor de ondersteuning.

Abstract

This research project explores the possibility of using Big Data techniques for the parallelization of Microsoft windows executables. We focused on a use-case for ALS. Theoretical research has been done to determine the best suitable framework/technique. We chose Apache Spark to be the most suitable and continued our research with this technique. With a framework we wrote in Python we were able to import and process the structure of an ALS simulation. After mapping the individual components of ALS to an Apache Spark transformation, we were able to ensure that Spark could recognize and import the entire structure. Subsequently we started running tests with actual executables of an ALS mockup. In our single-core setup Apache Spark performed similar to the current situation, as expected. For our multi-core setup Apache Spark performed similar to a little bit better. The tests in our multi-machine setup using Microsoft Azure were successful when using non ALS tests. To complete these tests with ALS further research and programming is required.

Inhoudsopgave

Lijst met afbeeldingen

1 Introductie

Vanwege de constante vraag naar meer rekenkracht, wordt er continu gezocht naar meer mogelijkheden voor parallel rekenen. Er moeten steeds meer datahandelingen worden verwerkt of zware berekeningen worden uitgevoerd. En hierdoor blijft de vraag bestaan: Hoe kan het beter? Hoe kan het sneller?

Voor de dataverwerking, en dan vooral gericht op Big Data, zijn er meerdere open source frameworks beschikbaar. Deze technieken worden momenteel al op grote schaal gebruikt. Hierbij valt te denken aan bijvoorbeeld Hadoop en Spark. Voor het genereren van data of het maken van berekeningen met behulp van deze technieken is er nog niet heel veel onderzoek gedaan. In 2014 is er een begin gemaakt met het zetten van de stap naar een financiële toepassingen [1]. Hierbij is gebruik gemaakt van Scala om Monte Carlo simulaties uit te kunnen voeren.

De 'nieuwe' gebruikswijzen van deze Big Data technieken zijn vooral gericht op het werken in Linux. Naast het besturingssysteem Linux zijn er genoeg bedrijven die gebruik maken van Microsoft Windows. Voor dit platform is er alleen nog niet veel bekend over het gebruik van deze Big Data technieken.

In dit onderzoek streef ik ernaar om een bijdrage te leveren aan de kennis over de mogelijkheden met betrekking tot het parallelliseren van Microsoft Windows applicaties. Hierbij zijn de eventuele mogelijkheden gelimiteerd tot de huidige Big Data technieken.

Om deze doelstelling te realiseren zal er worden gekeken naar een use-case geleverd door Ortec Finance. Hierin bekijken we de mogelijkheden om een Microsoft Windows applicatie, gericht op financiële scenario simulaties, te parallelliseren. Deze mogelijkheden houden we dus beperkt tot Big Data gerelateerde oplossingen.

We zullen beginnen met een theoretische studie waarbij we de verschillende Big Data technieken bekijken. Er zal worden gekeken naar wat de mogelijkheden zijn van deze technieken en wat nodig is voor de gebruik van deze technieken. Aansluitend zullen deze technieken vergeleken worden om te kijken welke het beste aansluit bij onze use-case van Ortec Finance. Deze keuze is dan het begin voor de rest van het tweede deel van dit onderzoek.

In het tweede gedeelte zullen we verder ingaan op de stappen die gezet zijn om de mogelijkheden met de gekozen techniek te onderzoeken. We beschrijven onze werkwijze en resultaten. Aansluitend, zullen we eindigen met de conclusie en aandachtspunten van dit onderzoek.

2 Probleem Beschrijving / use-case ALS

Ortec Finance is een financiële organisatie die zich sinds 1985 richt op het meten en managen van risico en rendement. Het is een onafhankelijke specialist die, gebaseerd op bewezen methodologieën, consultancy diensten en bijbehorende systemen levert. Deze worden aangepast aan de wensen van hun klanten. Hierbij valt te denken aan verzekeraars, pensioenfondsen, vermogensbeheerders, woningcorporaties maar ook gemeenten.

Eén van de werkzaamheden van Ortec is het verrichten van "Asset and Liability Management" (ALM) studies. ALM wordt gebruikt om een balans te kunnen vinden tussen risico en rendement. Zo is er aan de ene kant de mogelijkheid om veel risico te nemen waardoor je verwachte rendement ook hoog kan zijn. Daarentegen kan het ook slecht gaan met deze gekozen strategie en omdat er veel risico genomen is kan het eindresultaat ook zeer negatief zijn. De andere kant is dat er weinig risico genomen wordt, waardoor de kans dat het fout gaat kleiner is, maar daardoor is het verwachtte rendement ook klein. Er moet dus een keuze gemaakt worden hoeveel risico er genomen moet worden tegenover hoeveel rendement behaald moet worden. Deze keuze kan gemaakt worden op grond van de resultaten van ALM studies. Banken, pensioenfondsen en verzekeringmaatschappijen zijn voorbeelden van bedrijven waar ALM wordt gebruikt.

De voornaamste techniek die Ortec Finance gebruikt voor het leveren van goede strategieën voor de ALM studies is gebaseerd op scenario analyses. Deze scenario analyses worden uitgevoerd met Monte Carlo simulaties. Tijdens de simulatie wordt een groot aantal economische scenario's (mogelijke toekomstige ontwikkelingen van de economie) gegenereerd. Hierin worden mogelijke ontwikkelingen van economische variabelen zoals rentes, inflatie, wisselkoersen, etc. op een samenhangende wijze bepaald. Hierna worden de effecten van deze scenario's op een model van een financiële instelling toegepast. De gebruiker kan vervolgens de effecten van verschillende beleidskeuzes met elkaar vergelijken en zo een afweging maken over het te voeren beleid. Voor het leveren van deze strategieën gebruikt Ortec het software pakket Asset & Liabilities Senariomodel (ALS). Dit is een Microsoft Windows applicatie ontwikkeld door ORTEC Finance voor het uitvoeren van de ALM studies.

Door veranderingen in regelgeving en grotere aandacht voor mogelijke risico's is de complexiteit van de modellen en het aantal scenario's in de afgelopen jaren alleen maar groter geworden. Simulaties en andere berekeningen kunnen in sommige gevallen wel een paar uur tot wellicht een dag duren. Deze lange rekentijd is één van de redenen die het onderzoek naar 'High performance computing', en in dit geval naar Distributed Computing, parallellisatie motiveren.

Het ALS product maakt momenteel gebruik van Alchemi[1], een oud niet langer onderhouden open source project, dat lokale parallellisatie mogelijk maakt via multicore programming. Het is gebaseerd op het master-worker paradigm en implementeert het concept van Grid threads. Het start momenteel meerdere processen op de machine van de gebruiker. Een nadeel van Alchemi is onder andere de zwakke beveiliging. De authenticatie is zwak want het werkt alleen met een wachtwoord. Verder worden de wachtwoorden als tekst opgeslagen in een database en de geparallelliseerde applicatie is kwetsbaar voor kwaadwillende gebruikers. Deze nadelen en omdat deze oudere techniek niet gemakkelijk de omschakeling kan maken naar multi-machine ontstond de vraag naar een alternatieve techniek.

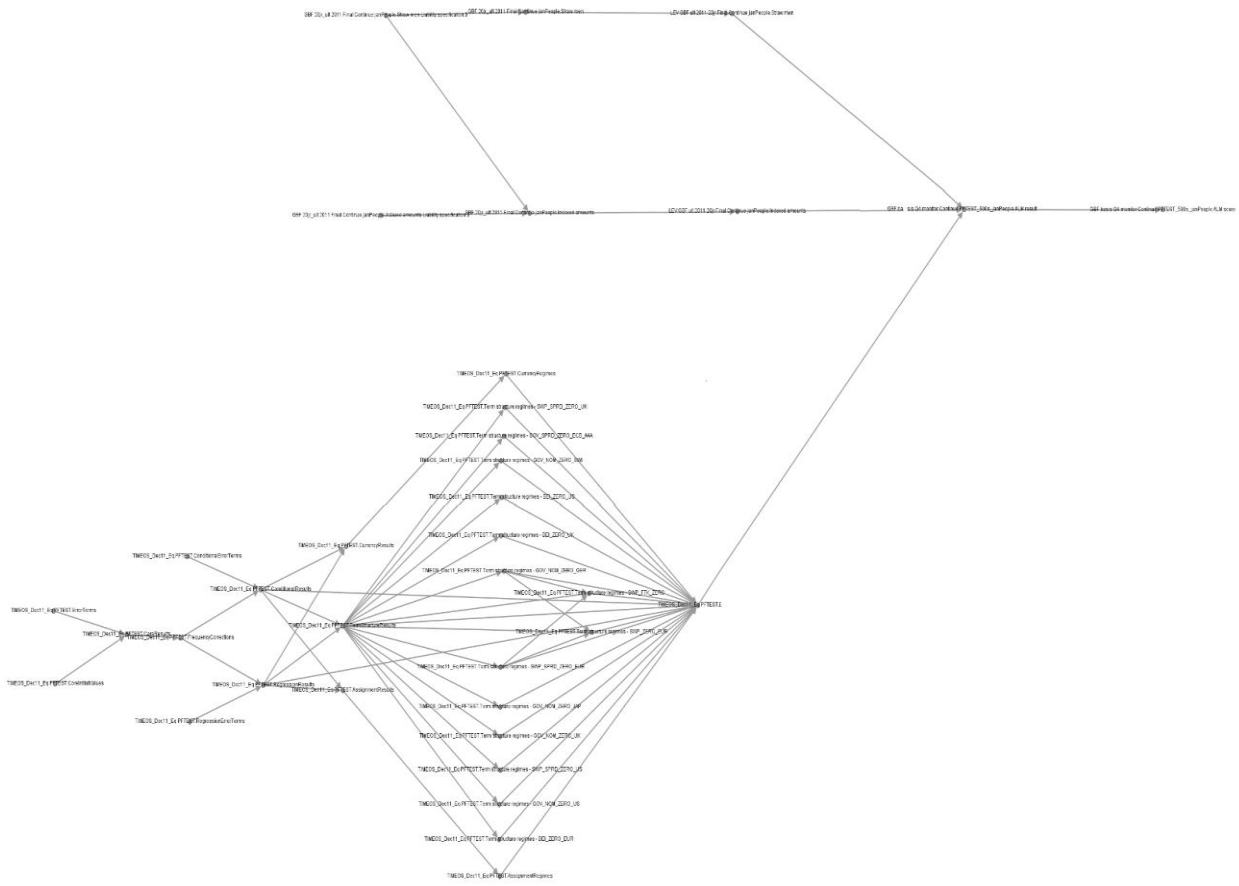
De doelstelling van deze use-case is een bijdrage te leveren aan de discussie over de manieren waarop de bestaande modellen kunnen worden verbeterd, 'verbeterd' in de zin van het mogelijk maken van kortere rekentijden. Daarbij zal worden ingegaan op de mogelijkheid van het gebruik van meerdere machines gelijktijdig en de toegankelijkheid van meerdere gebruikers. Naast deze eisen zal er ook rekening gehouden worden met de mogelijkheid om remote een taak te kunnen starten maar ook te stoppen. Taks Management is

[1] http://www.cloudbus.org/~alchemi/

ook een pré, hierbij moet gedacht worden aan het managen van een taak waaronder plannen, testen en bijvoorbeeld voortgang volgen. Verder zouden er meerdere verschillende versies van ALS naast elkaar moeten kunnen functioneren. Qua technieken zal het onderzoek zich richten op de huidige Big Data technieken.

Met betrekking tot de structuur van de berekening van het ALS product, kan nog worden opgemerkt dat dit is weer te geven als een graaf, een Directed Acyclic Graph (DAG) wel te verstaan. Deze graaf heeft geen cycli en alle edges zijn gericht, in andere woorden de knopen hebben directe ouders en/of kinderen. De knopen in de graaf staan voor de handelingen/stappen die gedaan moeten worden om een simulatie te voltooien. De afhankelijkheden van een handeling in de simulatie worden dus weergegeven in het aantal kinderen van deze knoop. Wat niet direct zichtbaar is in de knopen is het feit dat de knopen zelf ook weer uit meerdere taken kunnen bestaan. Dit geeft meer mogelijkheden in het opsplitsen van taken en is daarom een belangrijk feit voor het toepassen van de parallellisatie. We kunnen de taken ook opsplitsen en op deze manier een grotere graaf met meer knopen genereren.

Na het voltooien van de knopen vindt er bij ALS een datacommunicatie stap plaats. Hierin wordt de gegenereerde data weggeschreven naar een database. Doordat de “IO load” relatief klein is ten opzichte van de totale verwerkingstijd, zal dit geen grote rol spelen in deze use-case.



Figuur 1. Voorbeeld DAG structuur ALS

Een voorbeeld van de structuur van een graaf is weergegeven in Figuur 1. Hierin is duidelijk te zien dat er veel afhankelijkheden zitten in de ALS structuur. De graaf wordt gelezen van links naar rechts. Om rechts te

bereiken moeten de knopen links eerst worden afgewerkt. Dat er zoveel afhankelijkheden zijn geeft ons wel weer de mogelijkheid om hier te zoeken naar de mogelijkheden van de parallellisatie binnen ALS.

3 DAG scheduling probleem

Het parallellisatie probleem bij ALS is eigenlijk niet meer dan een scheduling probleem waarbij de verschillende knopen, en de bijbehorende taken, zo moeten worden ingepland dat de volgorde van deze knopen de afhankelijkheidsrelaties tussen de verschillende elementen van het systeem adequaat weergeeft. In dit hoofdstuk gaan we dan ook wat dieper in op de theoretische achtergrond van een (DAG) scheduling probleem. Op deze manier krijgen we een beter beeld hoe de taken van een DAG gepland kunnen worden. Op basis van deze theorie zal later ook een theoretische benadering worden gezocht van het de ALS graaf.

Scheduling problemen kunnen in het algemeen worden onderverdeeld in twee groepen: scheduling voor onafhankelijke taken of scheduling voor meerdere interactieve taken. Bij de eerste groep hebben we te maken met een standaard versie van het job shop scheduling probleem, waarbij de taken onafhankelijk van elkaar gepland kunnen worden op de processoren van het gedistribueerde cluster. Bij de tweede komt wat meer kijken, daar moet namelijk rekening worden gehouden met de interactie tussen verschillende taken. Hier moet bijvoorbeeld erop worden gelet dat eventuele afhankelijkheidsrelaties niet worden genegeerd.

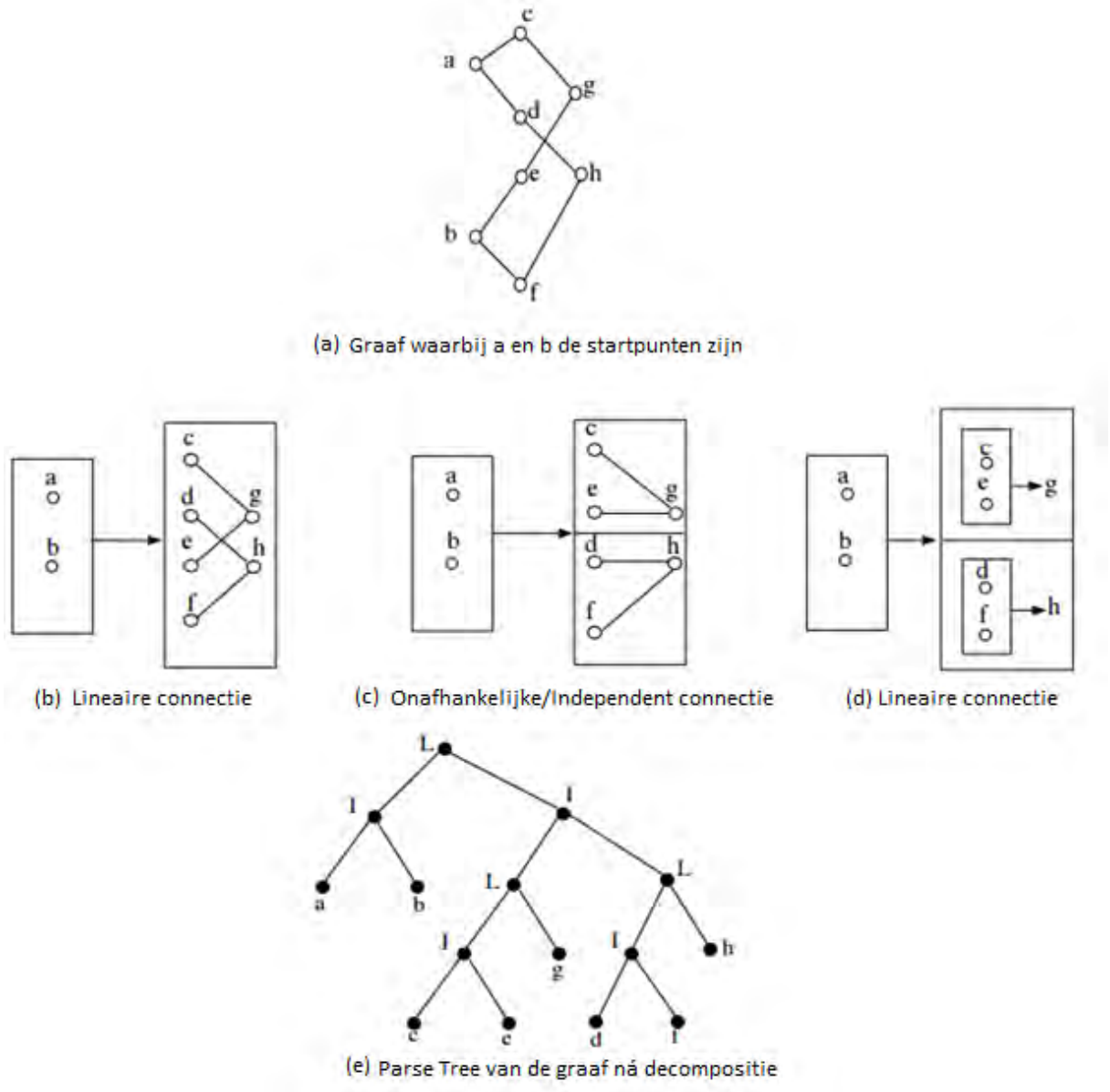
Bij het effectief plannen van taken is het doel om de totale runtime zo minimaal mogelijk te houden. Bovendien moet er rekening gehouden worden met de capaciteit van de middelen die beschikbaar zijn en met de eventuele eisen/eigenschappen van het probleem zelf. De structuur van de taken wordt veelal weergegeven in een graaf waar eventuele afhankelijkheden weer worden gegeven via edges.

Als we kijken naar ALS dan hebben we te maken met een sequentieel programma. Dat betekent een programma waarbij de knopen elkaar opvolgen. Voor ALS vormt er zich dan uiteindelijk een DAG. In zo'n geval wordt er vaak gesproken over een Program Dependence Graph (PDG). Hierbij representeren de nodes de taken met gewichten die de processing tijden weergeven [2]. In het geval van ALS zijn de gewichten/proces tijden niet van te voren bekend.

Het probleem van het parallelliseren van een PDG om minimale parallelle tijd te bereiken is een NP-hard probleem. Hierbij komen over het algemeen twee dingen kijken: het partitioneren en sequencing van de taken. In 1972 had Richard Karp [3] bewezen dat het Job sequencing probleem een NP-Compleet probleem is. Er zijn echter een paar simpele gevallen waarbij wel een oplossing is gevonden voor deze problemen in polynomiale tijd [4]. Hierbij kan gedacht worden aan gevallen waarbij de graaf een simpele Boom structuur heeft en de gewichten bij de taken overal het zelfde zijn.

Er zijn enorm veel verschillende heuristieken die het mogelijk maken om uiteindelijk een PDG wel te kunnen plannen over meerdere processoren en ook over meerdere pc's. McCreary [2] heeft in zijn paper meerdere technieken vergeleken. Daaruit kwam onder meer naar voren dat heuristieken gebaseerd op het kritieke pad, maar ook gebaseerd op de graaf decompositie constante acceptabele resultaten geven. Niettemin zijn de twee technieken gebaseerd op verschillende benaderingen. Bij beide methoden kan rekening gehouden worden met kosten voor de communicatie tussen de processoren. Het verschil zit in de aanpak van de parallellisatie. De kritische pad methode richt zich vooral op de sequentiële taken terwijl de graaf decompositie methode zich meer richt op de secties die onafhankelijk zijn en geparallelliseerd kunnen worden.

Graaf decompositie deelt de graaf eerst op in een sets van verschillende CLANs. Hierbij is een CLAN een subset van verschillende nodes, waar de elementen buiten de set dezelfde soort relatie hebben tot alle nodes in de set. Hierdoor zijn de uiteindelijke CLANS hiërarchisch in te delen in een parsetree waarbij de relatie tussen de verschillende taken wordt weergegeven. CLANs kunnen Lineair of Independent, primitief of triviaal zijn. Primitieve CLANs kunnen weer verder worden ontleed in lineaire of independent CLANs. De independent CLANs bestaan uit knopen die in parallel kunnen worden uitgevoerd en de lineaire CLANs bevatten afhankelijkheden in de uitvoer. Als laatste is elke node een triviale CLAN. Een voorbeeld van een graaf en decompositie is te zien in Figuur 2 [5].



(e) Parse Tree van de graaf ná decompositie

Figuur 2. Voorbeeld van een Graaf, de decompositie en de parsetree

De mogelijkheid voor parallellisatie wordt in de gedaante van een parsetree duidelijk weergegeven. De kinderen van de independent CLANs hebben geen sequentiële afhankelijkheden en kunnen daardoor worden parallel worden uitgevoerd. De boom moet depth-first worden benaderd zodat de lagere levels eerst worden uitgevoerd. Deze manier van uitvoeren geeft de mogelijkheid om gaandeweg structuur en planning aan te passen wanneer parallellisatie mogelijkheden zich voordoen [6].

Zodra een lineaire CLAN wordt bekeken moet de keuze worden gemaakt wat er gedaan gaat worden met de knopen. Ze kunnen allemaal worden geaggregeerd, ze kunnen allemaal worden geparallelliseerd of een combinatie van de twee. Na de keuze voor parallellisatie volgt de processor allocatie. Ditzelfde generieke proces zou ook kunnen worden toegepast op de DAG van ALS. In paragraaf 6.1 besteden we hier meer aandacht aan.

4 Methodes/Technieken

Er zijn meerdere technieken/methodes die van toepassing zijn als we denken aan parallellisatie. In dit hoofdstuk zullen enkele van deze technieken worden bekeken. Technieken zoals MPI[2] en frameworks van externen worden hierin niet meegenomen. Hoewel MPI een open standaard is en nog steeds één van de meest gebruikte libraries op het gebied van distributed computing, laten we het toch buiten beschouwing. We doen dit omdat het toch een wat oudere techniek is en het de flexibiliteit van eventuele aanpassingen daardoor ook erg verminderd. Frameworks van externe partijen worden in dit verslag ook overgeslagen, omdat de voorkeur van Ortec Finance uitgaat naar volledig eigen beheer van software en hardware. De Big-Data technieken die beschikbaar zijn voor parallellisatie zijn allemaal open source en bieden daarom volledige controle voor de gebruiker. Deze technieken zullen dan dus ook kort worden bekeken en vergeleken. Uiteindelijk zal er op basis van deze vergelijking een keuze worden gemaakt om met één van deze methodes door te gaan.

4.1 Apache Hadoop

Hadoop kan het best beschouwd worden in de context van 'big data': de voortdurende stroom van data die op allerlei manieren en in toenemende mate ter beschikking komt van gebruikers. Een van de problemen die zich hierbij voordoen is als volgt te formuleren: hoe sla je al die data op een efficiënte manier op en op welke wijze kun je al die data gebruiken voor analyse-doeleinden? De opslagcapaciteit van hard disks bijvoorbeeld is weliswaar zeer sterk toegenomen, maar dat geldt veel minder voor de snelheid waarmee de data kunnen worden gelezen. Een logische oplossing is om gebruik te maken van een cluster van machines.

"Hadoop" is de naam voor een open source platform dat kan worden gebruikt voor de opslag en analyse van grote hoeveelheden data. De Hadoop software is voornamelijk bedoeld voor 'distributed systems'; dat wil zeggen een cluster van computers, die samen opereren alsof het één gigantische computer betreft.

4.1.1 *Oorsprong*

Hadoop ontstond oorspronkelijk uit Apache Nutch, een open source web search engine. De oorsprong van Nutch vinden we in 2002, waar de ontwikkelaars er al snel achter kwamen dat de applicatie niet schaalbaar was met de biljoenen webpagina's op het internet. Op basis van een Google publicatie in 2003, waar in gesproken werd over Google 's Distributed Filesystem ofwel GFS, kon hiervoor een oplossing worden gevonden [7]. In 2004 werd een eigen versie van GFS geschreven t.w. Nutch Distributed Filesystem (NDFS).

In 2004 publiceerde Google een paper waarin ze MapReduce introduceerde. In 2005 werd Nutch helemaal aangepast op het gebruik van MapReduce en NDFS, waardoor het toepasbaar werd op meer dan alleen maar search applicaties. In februari 2006 splitste dit project zich dan ook af van Nutch en ging verder onder de naam Hadoop. De ontwikkelaar van Hadoop werkte op dat moment bij Yahoo! en hier kreeg hij een team en middelen toegewezen voor de verdere ontwikkeling van Hadoop. Hadoop werd een top level project bij Apache in januari 2008.

4.1.2 *Architectuur*

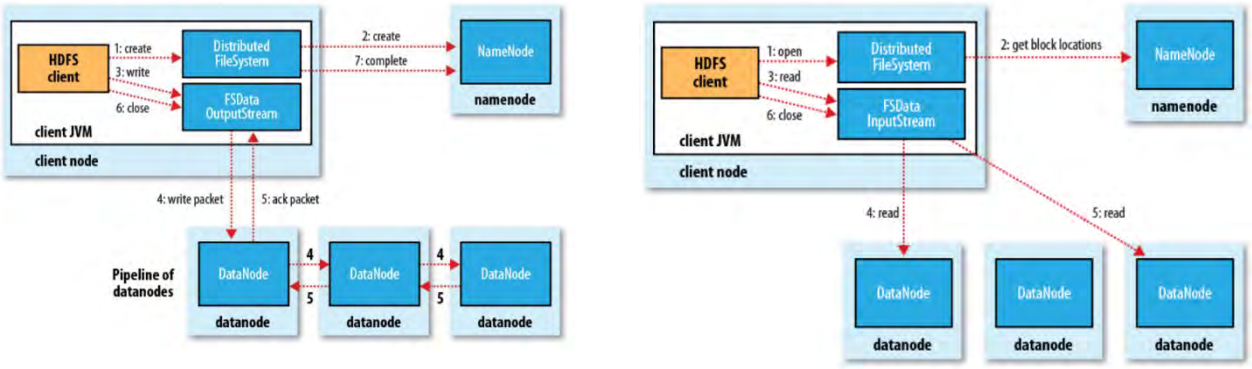
In deze paragraaf zal uitgelegd worden hoe Hadoop is opgebouwd. Er zal duidelijk worden hoe MapReduce werkt en wat voor rol YARN speelt.

4.1.2.1 **HDFS**

[2] http://www.mpi-forum.org/

Bij omvangrijke datasets is het noodzakelijk de opslag te distribueren over een cluster of netwerk van machines. Methoden die zorgen voor deze distributie worden normaliter aangeduid met de term: 'distributed filesystems'. Hadoop bevat ook een dergelijk systeem, namelijk Hadoop Distributed Filesystem, kortweg HDFS. Bijzondere aspecten van HDFS zijn onder andere dat:

1. HDFS zorgt voor de opslag van zeer grote bestanden, i.e. bestanden van honderden megabytes, gigabytes en nog grotere bestanden;
2. HDFS data verwerkt volgens een 'write-once, read-many-times' patroon, zie Figuur 3;
3. HDFS geschikt is voor een cluster van relatief eenvoudige machines.



Figuur 3. Read en Write proces van HDFS [7]

HDFS is gebaseerd op de volgende begrippen:

1. Blocks
2. Namenodes en Datanodes
3. Block Caching
4. HDFS Federation
5. HDFS High Availability

Blocks

Met een 'block' wordt in de context van HDFS verwezen naar het opdelen van omvangrijke bestanden in blokken die als onafhankelijke eenheden standaard 128 MB opslagruimte toebedeeld krijgen. 'Blocks' van bestanden kunnen eenvoudig opgeslagen worden op meerdere locaties. Dit zorgt ervoor dat bestanden die groter zijn dan het opslagmedia toch opgeslagen kunnen worden. Een tweede voordeel is dat het gebruik van 'blocks' in plaats van bestanden de opslag vereenvoudigt. Door omvangrijke bestanden te splitsen in delen met een vaste omvang wordt de beheersing vereenvoudigd. Het vaststellen van de hoeveelheid bestandsdelen die op één harde schijf kunnen worden opgeslagen is hier een voorbeeld van. Als derde voordeel kan worden genoemd dat blocks zich goed lenen voor replicatiedoeleinden, bijvoorbeeld om de risico's van beschadigde bestanden af te dekken.

'Namesnodes' en 'Datanodes'.

In het cluster van machines waarin HDFS wordt gebruikt, onderscheiden we twee functies die worden verricht door de computers in het netwerk. Het gaat om functies uitgevoerd in een 'master-worker' verband.
De 'namenode' is in dit verband de aanduiding van de computer die de rol speelt van 'master', dat wil zeggen de computer die verantwoordelijk is voor het beheren van de hiërarchische structuur van directories en bestanden. Tevens zorgt de 'master' voor het beheer van de bijbehorende metadata.
De namenode bevat ook informatie over de locatie van de verschillende 'blocks' van een bestand. Deze delen van een bestand zijn te vinden bij de 'workers' in het netwerk. Deze computers zijn dan de 'datanodes'. Deze 'werkpaarden' in het systeem zijn verantwoordelijk voor de opslag respectievelijk het opzoeken en zichtbaar maken van de data-blokken.

Block Caching

Hiermee wordt bedoeld dat in het geval van frequent gebruikte bestanden, de desbetreffende 'blocks' worden opgeslagen in het cachegeheugen van een 'datanode' om de leessnelheid te vergroten.

HDFS Federation

'HDFS Federation" staat voor de mogelijkheid die Hadoop biedt om een cluster van computers op een vereiste schaal te brengen. Dat wil hier zeggen, dat Hadoop de toevoeging van extra 'namenodes' faciliteert, die zich vervolgens toeleggen op het beheer van een selectie van bestanden, bijvoorbeeld user bestanden. Het voordeel hiervan is dat de onderlinge afstemming kan worden aangepast aan het niveau dat nodig is wanneer bij zeer grote clusters met heel veel bestanden, het beschikbare geheugen een knelpunt wordt.

HDFS High Availability

'HDFS High Availability' verwijst naar het gebruik, binnen Hadoop, van reserve 'namenodes' die de functie van actieve namenodes kunnen overnemen in geval van nood. Hierdoor kan het proces van verwerken van verzoeken van cliëntsystemen zonder veel vertraging kan doorgaan. Deze transitie staat onder leiding van een zogenaamde 'failover controller'.

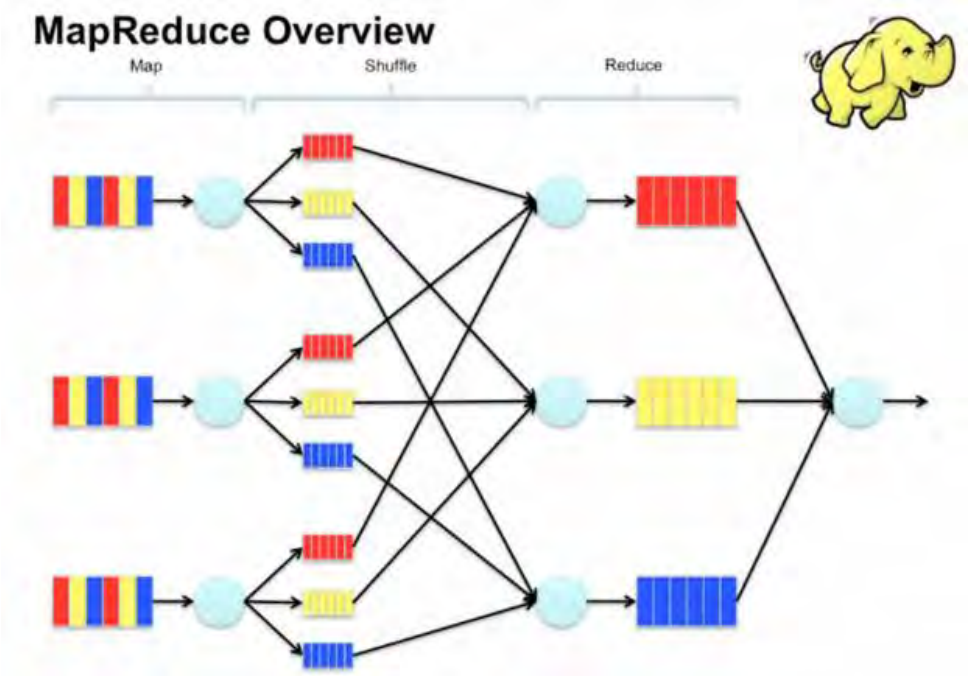
4.1.2.2 **MapReduce**

Een van de belangrijkste componenten van Hadoop is MapReduce. MapReduce is de software, bijvoorbeeld geschreven in Java of Python, die zich richt op het verwerken van grote data sets door zeer grote bestanden te splitsen en te verdelen over meerdere computers.

Voorbeeld: de verwerking van data, aangeleverd door weerstations op verschillende plekken op aarde, met het oog op de beantwoording van de vraag wat de hoogste temperatuur op aarde was in een bepaald jaar.

In theorie zou men de verwerkingstijd kunnen verbeteren door verschillende processen- bijvoorbeeld de data van verschillende jaarbestanden- te laten verrichten door verschillende parallel geschakelde computers. Dit is precies het punt waar Hadoop van nut kan zijn. Het idee achter Hadoop is om de verwerkingstijd van data te verbeteren door verschillende processen- bijvoorbeeld de data van verschillende jaarbestanden- te laten verrichten door een cluster van computers.

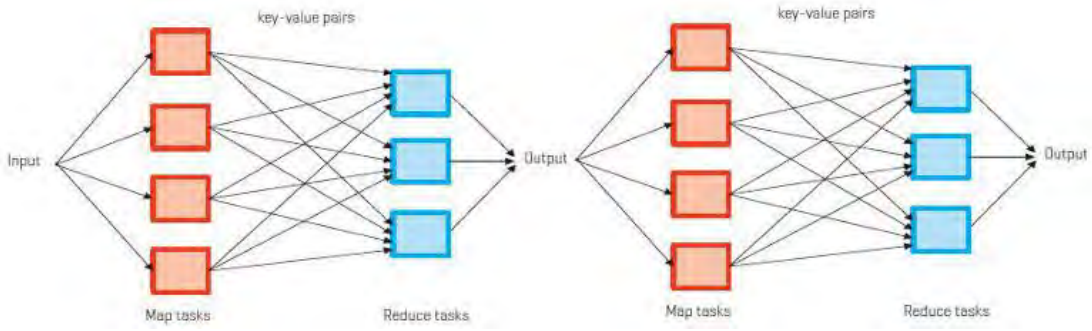
De applicatie MapReduce specificeert onder meer twee functies: de 'Map' functie en de 'Reduce' functie. De 'Map' functie is een instructie, die toegepast op de ruwe data, de data selecteert waarin we geïnteresseerd zijn, bijvoorbeeld de temperatuur en een jaartal. Bovendien zorgt het voor het filteren van onbruikbare records. Dit alles vormt dus de output van 'Map'. Hierna wordt de relevante data worden gesorteerd en gegroepeerd voor verdere bewerking. Deze stap zou ook gezien kunnen worden als 'Shuffle/Sort'. Deze output wordt vervolgens verstuurd naar de Reduce functie die de aangeleverde data doorloopt en uiteindelijk zorgt voor de finale output: bijvoorbeeld de hoogste temperatuur in een bepaald jaar. MapReduce bestaat dus eigenlijk uit drie dingen: Map, Shuffle/Sort en Reduce, zie Figuur 4.



Figuur 4. MapReduce schema

4.1.2.2.1 Chaining

Veel problemen zijn niet simpel op te lossen via één enkele MapReduce taak. Deze problemen zijn dan wel op te lossen via het concept MapReduce, namelijk door het toepassen van meerdere stappen in serie. Het werkt door middel van het linken van meerdere taken. Op deze manier wacht een latere taak op de voltooiing van eerdere taken waar die latere taak afhankelijk van is.



Figuur 5. Schema van twee MapReduce taken in een keten

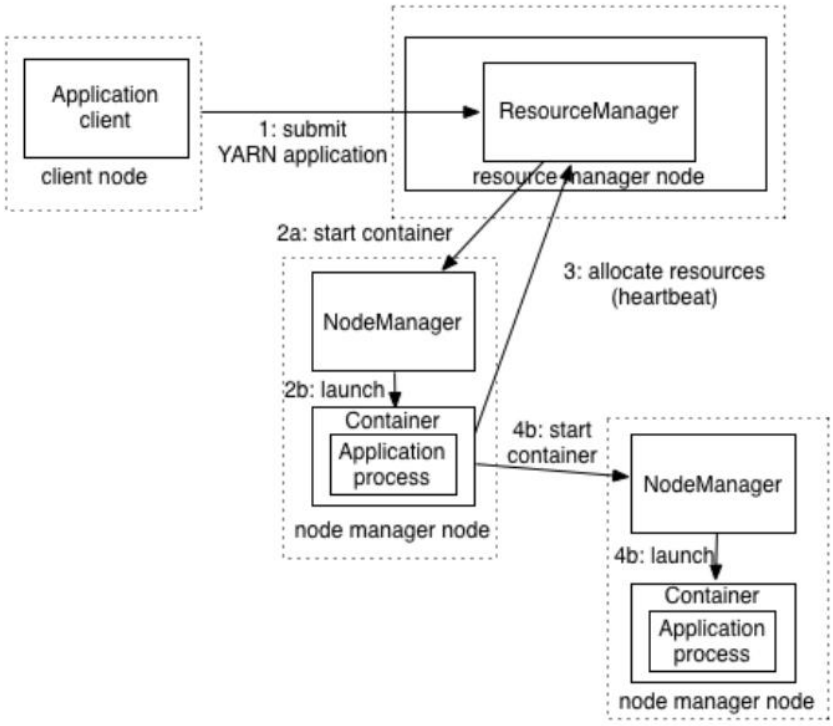
In een voorbeeld van een link met 2 taken, zie Figuur 5, wacht de tweede taak op voltooiing van de eerste taak. De eerste taak voert zijn taak uit en schrijft de output weg, die weer gebruikt wordt als input voor de tweede taak. Hierbij kan de keten aan taken zo groot worden als het nodig is voor de oplossing van het probleem.

4.1.2.3 **YARN**

YARN is de afkorting van “Yet Another Resource Negotiator” en dat geeft al enigszins aan waarnaar wordt verwezen.

De referent is de instantie binnen Hadoop die verantwoordelijk is voor het beheer van middelen als CPU en geheugen, die ter beschikking staan van het cluster van machines. YARN oefent deze functie uit door middel van een verzameling regels, protocollen en instrumenten voor de vervaardiging van software applicaties (API's). YARN bedient zich daarbij van de volgende twee op de achtergrond werkzame programma's 1) de 'resource manager' en 2) 'node managers'.

De 'resource manager' draagt zorgt voor de planning en de allocatie van de CPU en het geheugen van het cluster als geheel, terwijl 'node managers' zogenaamde 'containers' op gang brengen en monitoren. Containers zijn entiteiten die applicatie-specifieke processen uitvoeren, daarbij rekening houdend met de beschikbare middelen.



Figuur 6. De werking van YARN bij het starten van een applicatie

In Figuur 6 is te zien dat het verwerken van een YARN applicatie, bijvoorbeeld MapReduce, begint met het verzoek van een cliëntsysteem aan de 'resource manager' om een 'application master process' te starten. De 'resource manager' reageert hierop door een 'node manager' te zoeken. De desbetreffende 'node manager' zet de 'application manager' in beweging die bijvoorbeeld overgaat tot het uitvoeren van een eenvoudige berekening. Bij ingewikkeldere processen kan de 'application manager' vragen om de inschakeling van meerdere containers. Omdat ook hier de middelen schaars zijn, is hier een plannende entiteit nodig die de bestaande middelen verdeelt over de toepassingen.

YARN beschikt over verschillende planners en verschillende vormen van beleid om deze allocatie te organiseren. De plannende instanties zijn achtereenvolgens de First In First Out (FIFO) Scheduler, de Capacity Scheduler en de 'Fair Scheduler'. Deze laatste tracht aan alle actieve applicaties een gelijk deel van de middelen toe te kennen, de 'fair scheduling policy'.

4.2 Apache Tez

Tez is een applicatie framework gebouwd boven op Apache Hadoop YARN en heeft de mogelijkheid om complexe Directed Acyclic Graphs van taken te kunnen uitvoeren. Het generaliseert het MapReduce paradigma tot een krachtiger framework voor bewerkingen op data stromen in graven. Tez is als framework niet bedoeld voor end-users, maar het werkt als laag tussen end-user applicaties en Hadoop YARN.

4.2.1 *Oorsprong*

Tez ontstond in 2011. Het is gebaseerd op een publicatie over Dryad van Microsoft [8], waarbij de focus lag op het generaliseren van het MapReduce framework. Dryad was een onderzoeksproject van Microsoft voor het uitvoeren van parallelle data applicaties. Er waren enkele beta versies uitgebracht totdat in oktober 2011 werd besloten om te stoppen met dit onderzoek. De focus werd toen verschoven naar het Hadoop framework [9]. De ontwikkeling van YARN heeft mede geleid tot het ontstaan van Tez.

Tez is oorspronkelijk ontworpen door Hortonworks, maar op 24 februari 2013 is het overgegaan naar de Incubator van Apache waar het een top-level project werd op 16 juli 2014 [10]. Op het moment van schrijven is Tez bij versie 0.7.0.

4.2.2 *Benodigdheden*

Om met Tez te kunnen werken is er toegang nodig tot een werkend cluster waar YARN op is geïnstalleerd. Daarnaast moet er een duurzaam gedeeld bestandssysteem beschikbaar zijn en deze moet overweg kunnen met het Hadoop bestandssysteem interface. Voor development toepassingen is een volledig cluster niet noodzakelijk. Dan kunnen HDFS en YARN gewoon op een enkele machine draaien.

4.2.3 *Toepassingen*

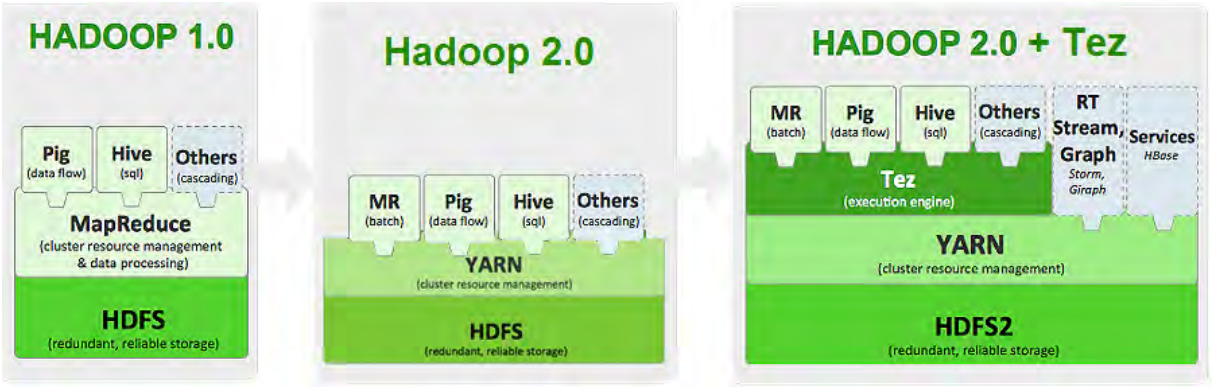
Enkele componenten van Tez zijn:

- Een omgeving die om kan gaan met traditionele MapReduce taken.
- Een omgeving die omgaat met DAG taken.
- Runtime planning.

4.2.4 *Architectuur*

Normale YARN applicaties kunnen worden geschreven op een wijze die een verbinding leggen tussen interactieve en batch taken. Dit wordt mogelijk gemaakt door het API en frameworks die geleverd worden door Apache Tez. Door de component library die Tez bevat, kunnen ontwikkelaars Hadoop applicaties schrijven die naadloos samengaan met YARN. Het is dan ook bovenop Apache Hadoop YARN gebouwd, zie Figuur 7.

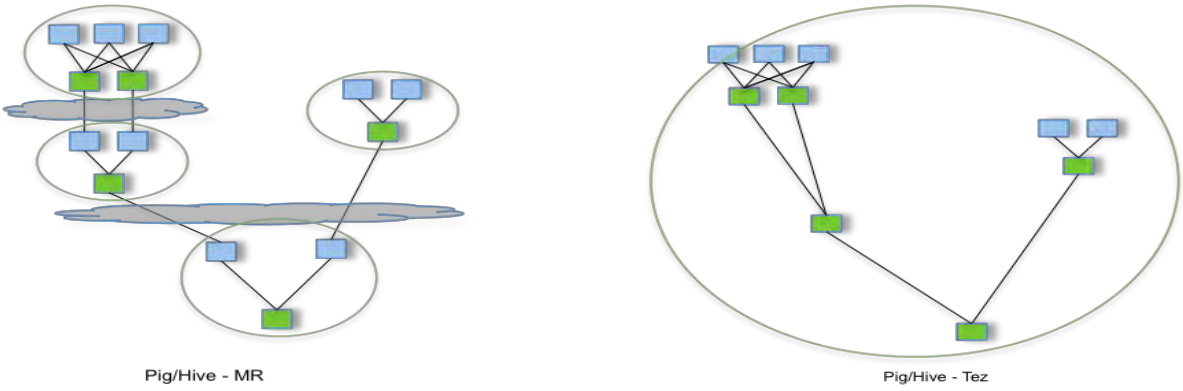
Doordat Tez naadloos samengaat met YARN, bezit het ook de voordelen hiervan. YARN levert het cluster management en de resource toewijzing service. Tez levert een nieuwe AppMaster, deze regelt en verwerkt een nieuwe taak definitie. De AppMaster communiceert met de ResourceManager van YARN voor het toewijzen van werker instanties van de uit te voeren taken. De NodeManager van YARN zorgt daarna voor het uitvoeren van de processen binnen deze instanties.



Figuur 7. Hadoop structuur 1.0 tot 2.0 + Tez (Een variant op [11])

Het originele doel van Tez was het maken van een executie engine die DAG executie stijlen ondersteunt voor Hive en Pig, data analyse platformen. Het gebruik van DAG executie ontwerpen voor gerelateerde query processen is al vaker onderzocht en de flexibiliteit van deze manier van executie maakt het zeer geschikt voor data processen zoals bijvoorbeeld iteratieve batch handelingen.

Een voorbeeld van het gebruik van de DAG-taak structuur in vergelijking tot de standaard MapReduce is gegeven in Figuur 8. Hier wordt gebruik gemaakt van het MRR patroon van Tez, Map Reduce. Hier heeft één Map taak meerdere Reduce taken. Hierdoor hoeven de data niet eerst worden weggeschreven naar het HDFS, maar kunnen gelijk worden doorgegeven aan het volgende proces. Dit elimineert de i/o naar disk en hiermee dus de wegschrijf tijden. De data kunnen nog wel worden weggeschreven, dit wordt dan gedaan op aangewezen checkpoints, dit heet check-pointing.



Figuur 8. Voorbeeld van een MapReduce executie plan vergeleken met het DAG executie plan van TEZ

De flexibiliteit die Tez biedt zorgt er wel voor dat er meer inspanning nodig is, in vergelijking met MapReduce, voor het goed gebruikt kan worden. Voor eindgebruikers is dit niet direct een probleem, omdat Tez geen end-user applicatie is zoals MapReduce. Het is ontworpen zodat ontwikkelaars end-user applicaties kunnen bouwen boven op Tez.

4.2.4.1 **Tez API**

Het idee van de DAG is dat het proces bij de root begint en zich omlaag werkt via de gerichte paden langs de knopen totdat het een eindpunt bereikt heeft. De graaf mag geen cycli bevatten omdat het fout tolerantie mechanisme dat Tez gebruikt gebaseerd is op het her uitvoeren van gefaalde taken. Bij cycli in een graaf is het lastig om bij de paden terug te lopen om te zoeken naar de laatste succesvolle knoop.

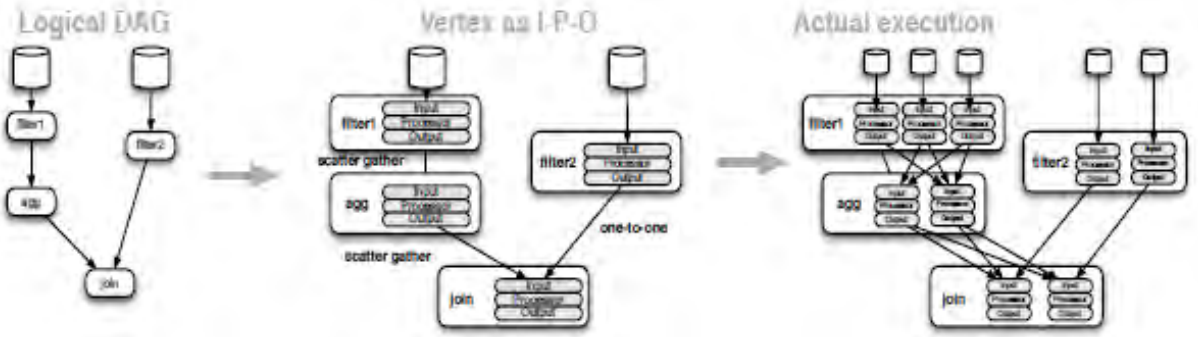
De API die Tez meelevert bevat drie componenten en is geschreven in Java [11]. De belangrijkste component is de DAG (Directed Acyclic Graph/ Acyclische gerichte graaf). Vervolgens wordt deze dan weer onderverdeeld in knopen en takken (nodes en edges). Via de knopen kan een zekere logica met betrekking tot de benodigde middelen en omgeving toegevoegd worden. Elke knoop staat tot een stap binnen de gehele DAG/taak. De takken omschrijven de connectie tussen twee knopen.

Aan de takken in de DAG moeten eigenschappen worden meegegeven zodat Tez de 'Logical DAG' bij het uitvoeren kan vertalen naar de losse fysieke set van taken (Figuur 9). Er zijn verschillende soorten eigenschappen die meegegeven kunnen worden.

Zo is er de data movement eigenschap. Deze bepaalt de datastroom tussen verschillende taken. Binnen de Data movement definiëren we drie verschillende types. Zo is er het type "One-to-One" dat ervoor zorgt dat data van de ene producerende taak naar een bepaalde consumerende taak moet gaan. De tweede optie is "Broadcast", hierbij gaan data van een producerende taak naar alle consumerende taken. En de laatste van de drie is *"Scatter-Gather"*. Hierbij verdelen de producerende taken alle taken op in shards die door de consumerende taken worden opgepakt. De i^{de} shard gaat naar de i^{de} consumerende taak.

Een tweede eigenschap betreft scheduling. Hierbij wordt gedefinieerd wanneer een consumerende taak wordt ingepland. Voor deze eigenschap hebben we de optie van "Sequential", waarbij een consumerende taak ingedeeld kan worden nadat een producerende taak afgerond is. Een andere optie is "concurrent" waarbij een consumerende taak tegelijkertijd met een producerende taak wordt ingedeeld.

De derde en laatste eigenschap die meegegeven kan worden heeft betrekking op de data source. Deze bepaalt hoelang de uitkomst van de taken bewaard blijft. De eerste mogelijkheid hiervoor is "Persisted". Met deze optie is de uitkomst beschikbaar wanneer de taak voltooid is, maar deze uitkomst kan later alsnog verloren gaan. Willen we dat de uitkomst ook achteraf beschikbaar blijft dan kiezen we voor de optie "Persisted-Reliable". Een andere optie is "Ephemeral", hierbij is de uitkomst alleen beschikbaar zolang de desbetreffende taak nog actief is.



Figuur 9. Omzetting van Logical DAG naar fysieke set van taken

4.3 Apache Spark

Apache Spark is een krachtige open source engine die gericht is op snelheid, gemak en geavanceerde analyses. Oorspronkelijk was Spark ontworpen voor interactie query's en iteratieve algoritmen. Dit waren twee grote en belangrijke zaken die niet goed worden aangepakt door batch frameworks zoals MapReduce. Spark blinkt uit in scenario's waarbij snelle prestaties vereist zijn, zoals iteratieve verwerkingen, interactief data opvragen, grootschalige batch berekeningen, streaming en graaf berekeningen.

4.3.1 *Oorsprong*

Spark begon als een onderzoeksproject op de UC Berkeley AMPLab in 2009 waar het vooral gefocust was op Big Data analyse. In 2010 werd het publiekelijk (open source) gemaakt onder een BSD licentie. Na deze stap groeide de ontwikkelaarsgemeenschap op Github flink en verhuisde het project naar Apache in 2013. Hier ontstond een groot netwerk dat mee werkte aan het project, bestaande uit meer dan 400 ontwikkelaars van 100 verschillende bedrijven. uiteindelijk groeide het uit tot een top-level project in 2014 [12].

Het doel was om een programmeermodel te ontwerpen dat een grotere hoeveelheid applicaties zou ondersteunen dan MapReduce en tegelijkertijd de fouttolerantie zou behouden. MapReduce is inefficiënt voor multi-pass applicaties die een low-latency nodig hadden en data moesten uitwisselen tussen meerdere parallelle operaties. Dit soort applicaties komen tamelijk vaak voor in analyses en bevatten onder andere:

- Iteratieve algoritmen, waaronder veel machine learning (machinaal leren) algoritmen en graaf algoritmen zoals PageRank.
- Interactief datamining, waarmee gebruikers data kunnen laden in het RAM geheugen. Tevens wordt bevorderd dat data worden gedeeld op een cluster wat gebruikers in staat gesteld data herhaaldelijk op te vragen.
- Streaming applicaties die het behoud van dezelfde geaggregeerde toestand op verschillende tijdstippen mogelijk maken.

Traditionele MapReduce en Directed Acyclic Graph (DAG) engines zijn sub-optimaal voor bovenstaande applicaties omdat ze gebaseerd zijn op een data stroom zonder cycli. Dit wil zeggen een applicatie een serie van verschillende taken moet uitvoeren, waarbij elke taak data leest van een stabiele opslag en het daarna weer terug schrijft naar het zelfde opslag medium. Hier zijn over het algemeen significante kosten verbonden aan het lezen en wegschrijven van de data [13].

Spark biedt Resilient Distributed Datasets (RDDs) die bovengenoemde applicaties efficiënt ondersteunen. Meer hierover is te vinden in paragraaf 4.3.5 op pagina 18.

4.3.2 *Benodigdheden*

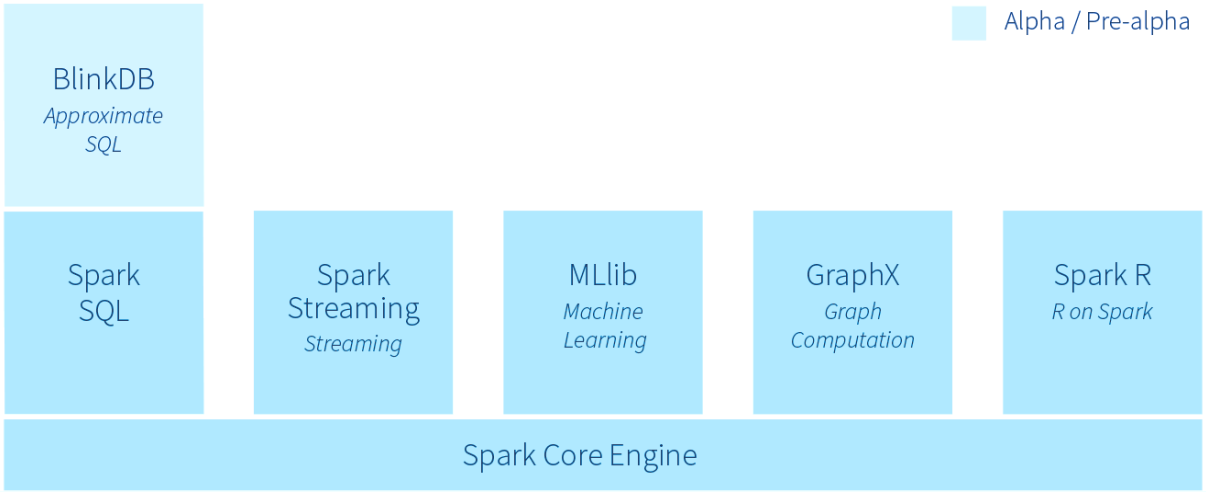
Spark kan standalone worden gebruikt. Hiervoor is alleen nodig dat Java op elke node is geïnstalleerd. Ook kan het gebruikt worden met een losse cluster manager zoals Mesos en YARN (Hadoop). In dit laatste geval is er ook een soort van gedeeld bestandssysteem nodig als Spark gebruikt wilt worden op een cluster.

Voor de volgende talen bestaat er een API:

- Python
- Scala
- Java
- R (vanaf 1.4.0 – juni '15)

4.3.3 *Toepassingen*

Spark heeft een ruim aanbod aan toepassingen. Hieronder worden de kern elementen één voor één kort toegelicht.



Figuur 10. Spark Ecosysteem [14]

Spark Core

Spark Core bevat de basis functionaliteiten van Spark. Hierbij kan worden gedacht aan componenten voor taak planning, geheugen management, error herstel en interactie met opslag systemen. De Spark Core bevat naast deze componenten ook de API die de RDD's definieert. Het bevat APIs voor het bouwen en manipuleren van de collecties van items die de RDD's representeren.

Spark SQL

Voor werken met gestructureerde data bestaat de package Spark SQL. Het maakt het mogelijk om de data te doorzoeken met behulp van SQL en tevens ook de Apache Hive variant van SQL (HQL). Spark SQL ondersteunt veel verschillende soorten databronnen waaronder Hive tabellen, Parquet en JSON.

Naast een SQL interface voor Spark, geeft Spark SQL ontwikkelaars ook de mogelijkheid om SQL query's te mixen met programmatische data manipulaties die ondersteund worden door de RDD's in Java, Python en Scala. Hierdoor kan SQL gecombineerd worden met meer complexere analyses.

Spark Streaming

Voor het verwerken van live data stromen heeft Spark de component Spark Streaming. Hierbij kan worden gedacht aan bijvoorbeeld logbestanden die gegenereerd worden door productie web servers. De API die Spark Streaming levert voor het manipuleren van de data stromen lijkt sterk op de Spark Core's RDD API. Hierdoor is

het voor ontwikkelaars gemakkelijk gemaakt om snel te kunnen leren van en te wisselen tussen applicaties die data bewerken vanuit geheugen, disk of die live binnenkomen.

MLlib

Voor Machine Learning (ML) heeft Spark de library MLib. Deze library omvat verschillende soorten machine learning algoritmen, zoals classificatie, regressie, clusteren en collaboratief filteren. Het ondersteunt tevens model evaluatie en data import. Deze technieken zijn zo ontworpen dat ze op een cluster geschaald kunnen worden.

GraphX

GraphX geeft Spark de functionaliteit voor het manipuleren van graven. Hierbij kan gedacht worden aan bijvoorbeeld een sociaal vrienden netwerk graaf. Tevens geeft GraphX de mogelijkheid voor graph-parallel computations. Doordat het de Spark RDD API gebruikt, kan GraphX gerichte graven met eigenschappen aan elke vertex en egde maken.

Cluster Managers

Spark is ontworpen om efficiënt te kunnen schalen van één tot meerdere duizenden nodes. Om dit te kunnen doen en tevens zijn flexibiliteit te behouden draait Spark op verschillende cluster managers. Enkele voorbeelden zijn Hadoop YARN, Apache Mesos en Sparks eigen Standalone Scheduler.

SparkR[3]:

SparkR is een nieuwe functionaliteit die sinds juni 2015 voor het eerst als officiële release werd vrijgeven. Het is een library voor de R Statistical taal die er voor zorgt dat R-gebruikers Spark functionaliteit geeft in de R shell.

Een voordeel van de grote open source community die Spark heeft, is dat er constant nieuwe innovaties zijn om Spark's horizon te verbreden, waarvan vele een oorsprong hebben in het UC Berkeley's AMPLab. Een voorbeeld van een alpha (on-going) project is BlinkDB. Het is een project dat zich richt op interactieve SQL query's die een afweging bieden tussen precisie en reactietijd.

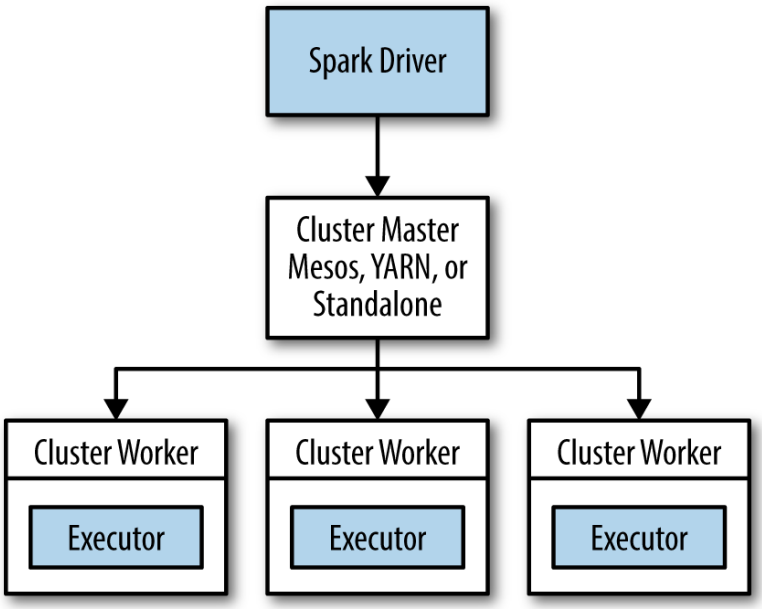
Spark bevat een geavanceerder Directed Acyclic Graph (DAG) engine, die cyclische data stromen ondersteunt. Bij een Spark job wordt er een DAG van taken gecreëerd die kan worden uitgevoerd op een cluster. Het verschil tussen MapReduce en Spark is dat, MapReduce een DAG aanmaakt met altijd twee taken, Map en Reduce, terwijl de DAG bij Spark een willekeurig aantal taken kan bevatten.

[3] Na het schrijven van deze sectie is versie 1.4.0 uitgebracht waarbij de eerste release zat voor SparkR.

4.3.4 *Cluster Management*

Zodra Spark op een cluster gebruikt gaat worden is een Cluster management engine noodzakelijk. Hiervoor kan onder andere Hadoop/YARN, Mesos en ook de Standalone Scheduler van Spark zelf gebruikt worden.

In Figuur 11 is de architectuur te zien van Spark in distributie modus. Hier is te zien dat Spark gebruik maakt van de master/slave structuur met één centrale coördinator, de driver, en vele werkers.



Figuur 11. De componenten van een gedistribueerde Spark applicatie

Spark komt met een script dat gebruikt kan worden om het programma in te voeren. Dit script is het spark-submit script. Dit script kan verbinding maken met verschillende cluster managers en kan beheren hoeveel middelen een applicatie krijgt. Bij Hadoop/YARN kan spark-submit de driver gebruiken in het cluster zelf, bijvoorbeeld op een worker node, terwijl bij andere een lokale machine nodig is.

4.3.5 *Resilient Distributed Dataset (RDD's)*

RDD's zijn gedistribueerde memory abstracties die er voor zorgen dat programmeurs berekeningen kunnen uitvoeren in geheugens op grote clusters op een error tolerante manier. Simpel gezegd is een RDD een onveranderlijke gedistribueerde collectie van voorwerpen. Elke RDD is weer opgesplitst in meerdere partities.

Ze bieden een beperkte vorm van gedeeld geheugen gebaseerd op coarse-grained transformaties in plaats van fine-grained. Door deterministische operaties op data in opslag of op andere RDD's kunnen ze worden aangemaakt. Doordat ze geen fijne schrijf operaties ondersteunen zijn ze minder geschikt voor bulk schrijf operaties. Voor applicaties waarbij asynchrone fine-grained updates worden gedaan naar gedeelde toestanden, zijn RDDs ook minder geschikt. [15]

Twee voordelen van RDDs zijn:

- Met bulk operaties op RDDs kunnen de taken gepland worden op basis van data locatie om prestaties te verbeteren.
- Als er niet genoeg geheugen is om de data op te slaan, dan kunnen partities worden weggeschreven op disk (persistentie). Hierbij wordt wel de aanname gedaan, dat de data gebruikt gaan worden voor scan-based operaties. [16]

4.3.5.1 Persistentie

Spark zal data naar disk schrijven zodra het niet meer in geheugen past. Door deze eigenschap kan elke hoeveelheid data worden gebruikt. Hetzelfde geldt voor de in de cache opgeslagen datasets die niet meer in het geheugen passen. Deze zullen worden weggeschreven naar disk of zullen terplekke opnieuw worden geconstrueerd, wat bepaald wordt door het RDD's opslag niveau, zie Tabel 1. Tevens kan het ook handig zijn om data met opzet te willen opslaan, hier kan vooral gedacht worden aan iteratieve algoritmen die vaak de data benaderen. Op deze manier kan voorkomen worden dat dezelfde RDD meerdere keren wordt aangemaakt.

Spark heeft meerdere verschillende levels voor persistentie. In Java en Scala worden de data standaard als geserialiseerd object opgeslagen in de JVM heap. In Python worden de data altijd geserialiseerd. Wanneer data naar disk worden geschreven, dan wordt het automatisch geserialiseerd. Als gewenst is dat de data worden gerepliceerd dan kan dat door _2 toe te voegen aan het einde van de verschillende opslag levels.

Level	Geheugen Gebruik	CPU Tijd	In Geheugen?	Op Disk?	Commentaar
Memory_Only	Hoog	Laag	Ja	Nee	
Memory_Only_Ser	Laag	Hoog	Ja	Nee	
Memory_And_Disk	Hoog	Gemiddeld	Gedeeltelijk	Gedeeltelijk	Schrijft naar disk als er teveel data zijn voor in het geheugen
Memory_And_Disk_Ser	Laag	Hoog	Gedeeltelijk	Gedeeltelijk	Schrijft naar disk als er teveel data zijn voor in het geheugen. Past Serialisatie toe op data in het geheugen.
Disk_Only	Laag	Hoog	Nee	Ja	

Tabel 1. Persistentie levels van org.apache.spark.storage.StorageLevel en pyspark.StorageLevel

4.4 Naiad

Naiad is een project dat het Microsoft research team startte in oktober 2012 [17]. Sinds eind 2013 is het project verhuisd naar Apache en bevindt het zich onder een Apache 2.0 open source license. Het is een systeem voor data-parallel dataflow berekeningen en het bevat onder andere libraries voor event processing, graph computation en Machine learning [18].

Het biedt ondersteuning voor het Microsoft .NET Framework, maar ook voor het cross platform open source .NET framework Mono. In april 2014 is er een release geweest waarbij ondersteuning voor YARN en Microsoft Azure HDInsight 3.0 werd geïntroduceerd. Het was daarmee één van de eerste end-to-end .NET oplossingen voor het implementeren van data analyses op Hadoop [19].

4.5 Vergelijking

In dit hoofdstuk gaan we in op de verschillen tussen Apache Spark, Apache Hadoop en Apache Tez. Naiad wordt hier weggelaten simpelweg vanwege het feit dat het een project is waar weinig tot geen voortgang in zit. De laatste release op het moment van schrijven is versie 0.5 van 17 oktober 2014.

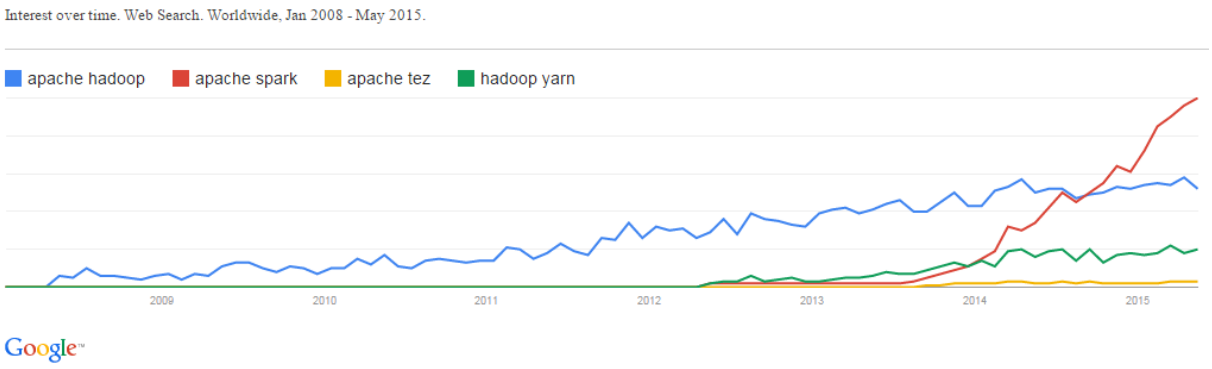
4.5.1 *Populariteit*

Om een indruk te krijgen van de populariteit van de verschillende technieken/methodes, maken we gebruik van Google trends. Deze service van Google geeft een indruk hoe vaak bepaalde termen gezocht en gebruikt worden. Google beschikt over een enorme hoeveelheid klantendata. Google verzamelt hun data uit zoek opdrachten via de Google zoekmachine, Gmail berichten, dingen die ingetypt worden in de Chrome URL balk en alle andere Google services. Daarnaast is Google nog steeds de meest populaire zoekmachine [20].

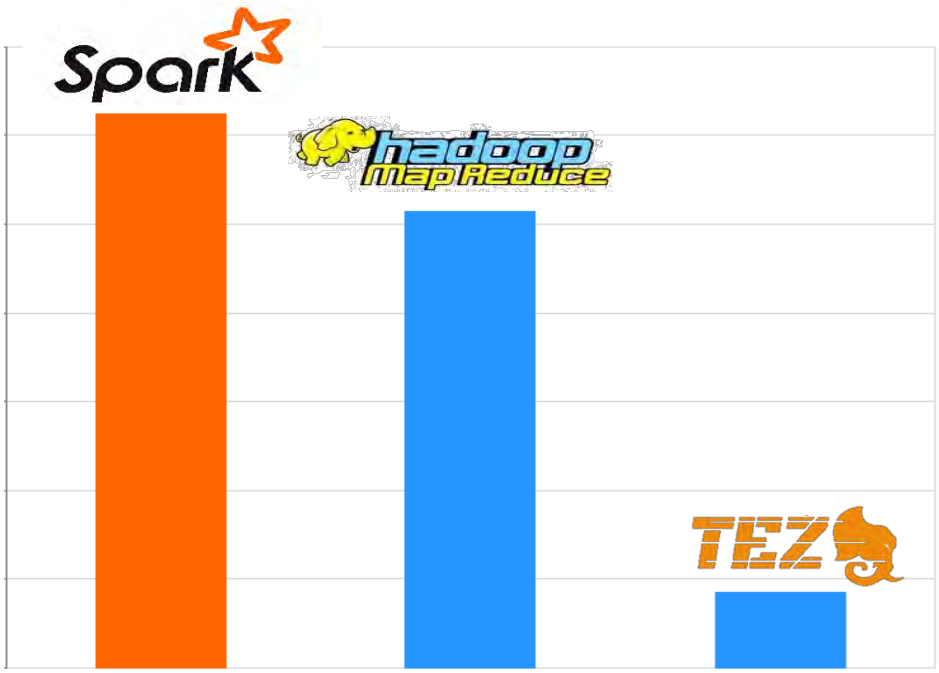
In Figuur 12 is een overzicht van de zoek trends te zien over de afgelopen drie jaar. Apache Hadoop Apache Spark en Apache Tez werden een top level project respectievelijk in 2008, begin 2014 en halverwege 2014. Voor Spark is dit goed te zien aan de trendlijn. De lijn toont een sterk stijgende lijn eind 2013 begin 2014.

Hadoop is in de periode vanaf 2008 tot halverwege 2012 de enige in zijn soort en heeft een redelijke constante toename in populariteit. Vanaf midden 2012 begonnen Spark en Tez zich te laten zien via Google trends. In de periode van 2012 tot begin 2014, waar Spark en Tez officieel geïntroduceerd werden, blijft de stijging in populariteit van Hadoop redelijk constant. In de periode dat Spark een top level project werd is een er een iets grotere stijging te zien in de populariteit maar, deze neemt weer af en daarna blijft de populariteit weer constant.

Spark is vanaf het moment van release enorm gegroeid en had tegen het einde van 2014 Hadoop ingehaald in de populariteit van Google trends. De populariteit is nog steeds stijgend. Tez is in vergelijking met Spark en Hadoop flink achtergebleven qua populariteit. Dezelfde verhoudingen zijn te zien als we kijken naar het aantal contribuanten van de open source projecten Spark, Hadoop MapReduce en Tez, zie Figuur 13.



Figuur 12. Google Trends vanaf 2008. Verhouding ten opzichte van de hoogst scorende zoekterm op.



Figuur 13. Verhouding contribuanten Spark, Hadoop MapReduce en Tez[4]

4.5.2 *Functionaliteit*

Apache Hadoop MapReduce is de oudste van de 3 bovengenoemde technieken en is daarmee ook zo goed als vervangen door de andere twee. Tez is meer een extensie op MapReduce, maar biedt naast de oude opties ook meer mogelijkheden. Spark is een volstrekt ander platform dat veel meer opties behalve MapReduce biedt.

Apache Spark en Tez hebben vergelijkbare eigenschappen en mogelijkheden. Op papier bezitten ze allebei in-memory mogelijkheden, functioneren ze allebei boven op Hadoop YARN en bezitten ze beide een DAG engine. In sommige gevallen wordt Spark als een meer volwassen versie van Tez gezien plus meer. Het bevat namelijk net zoals TEZ de DAG functionaliteit, maar bied daarnaast meer mogelijkheden.

Eén van de grootste verschillen is dat Spark, behalve dat het samen met YARN kan werken ook als Standalone kan functioneren. Tez kan daarentegen niet los functioneren en is afhankelijk van YARN.

Een belangrijk punt van Tez is dat het gebaseerd is op MapReduce van Hadoop. Hierdoor kunnen oudere MapReduce applicaties redelijk probleemloos worden overgenomen door Tez, maar het blijft wel het manco houden dat MapReduce ook heeft. Het kan niet worden toegepast op problemen die niet direct op te lossen zijn met het MapReduce concept. Spark daarentegen biedt naast het MapReduce batch processing platform enorm veel mogelijkheden, zie paragraaf 4.3.3.

Spark is in de korte tijd dat het nu meedraait enorm gegroeid en wordt zelfs al genoemd als vervanger van het ‘oude’ MapReduce van Hadoop. Deze flinke groei in populariteit is zeker één van de redenen waarom de voorkeur naar Spark zou kunnen gaan. Verder is de mogelijkheid voor het uitvoeren van taken die niet per definitie map-reduce zijn een erg belangrijk voordeel van Spark. De taken in ALS zijn namelijk niet altijd als MapReduce te schrijven. Door de eerder genoemde argumenten en het feit dat Spark ook gekarakteriseerd wordt voor de omgang met DAGs, wordt er voor de rest van dit onderzoek gekozen om met Spark verder te gaan.

[4] Kansas City Big Data Users Group July 2014

5 Apache Spark

In de voorgaande vergelijking kwam naar voren dat Apache Spark de logische keuze is om verder mee te gaan in dit onderzoek. In dit hoofdstuk besteden we daarom ook meer aandacht aan deze keuze. Er zal iets dieper worden ingegaan op de vraag hoe de DAG structuur tot stand komt en ook zal er iets meer informatie gegeven worden over scheduling met Spark.

5.1 Scheduling

Spark applicaties bevatten een Task Scheduler, dat wil zeggen een programma dat ervoor zorgt dat door Spark gespecificeerde operaties of taken ('jobs' zoals het opslaan van data) worden uitgevoerd op het moment dat een vooraf ingesteld signaal daartoe de aanleiding vormt. In het bijzonder kunnen meerdere parallelle taken tegelijkertijd worden uitgevoerd. Deze parallelle taken staan onder controle van dat deel van de code die op autonome wijze kan worden uitgevoerd, dat wil zeggen onafhankelijk van het hoofdprogramma.

The Scheduler van Spark is 'thread-safe'. De uitdrukking 'thread of execution' staat voor de kleinste reeks van instructies die door een Scheduler worden beheerd, terwijl 'thread-safe' verwijst naar de wijze waarop gedeelde datastructuren worden behandeld, namelijk op een manier die de veiligheid van de desbetreffende operaties garandeert.

Normaliter brengt de Scheduler van Spark opdrachten op een FIFO wijze ten uitvoer. Dat komt er in het kort op neer dat de eerste taak of operatie als eerste in aanmerking komt voor de toewijzing van de beschikbare resources als geheugen. Wanneer een specifieke taak wordt opgedeeld in stadia waarin specifieke taken moeten worden uitgevoerd, kan pas worden begonnen met de tweede hoofdtaak, wanneer alle stadia van de eerste hoofdtaak zijn voorzien van de benodigde middelen. Met de invoering van Spark 0.8 is echter een alternatieve werkwijze mogelijk geworden: 'fair sharing'. Dit komt erop neer dat aan alle bestaande 'jobs' of handelingen van Spark een ongeveer gelijk deel van de benodigde middelen wordt toegekend.

Het voordeel hiervan is dat bij de uitvoering van jobs geen rekening hoeft te worden gehouden met de tijdsduur in die zin dat kortdurende taken niet hoeven te wachten totdat langdurende zijn voltooid. Een tweede voordeel van 'fair scheduling' is dat de verschillende jobs kunnen worden georganiseerd in zogenaamde 'pools', die op verschillende wijze kunnen worden ingepland voor de uitvoering. Het is bijvoorbeeld mogelijk een 'high-priority pool' te construeren die plaats biedt aan belangrijke taken en operaties. Deze aanpak is geïnspireerd door de Fair Scheduler van Hadoop. Een ander aspect van de pools dat niet onbelicht mag blijven, is dat iedere pool standaard een gelijk deel van de resources krijgt toebedeeld. Binnen een pool geldt echter de FIFO volgorde.

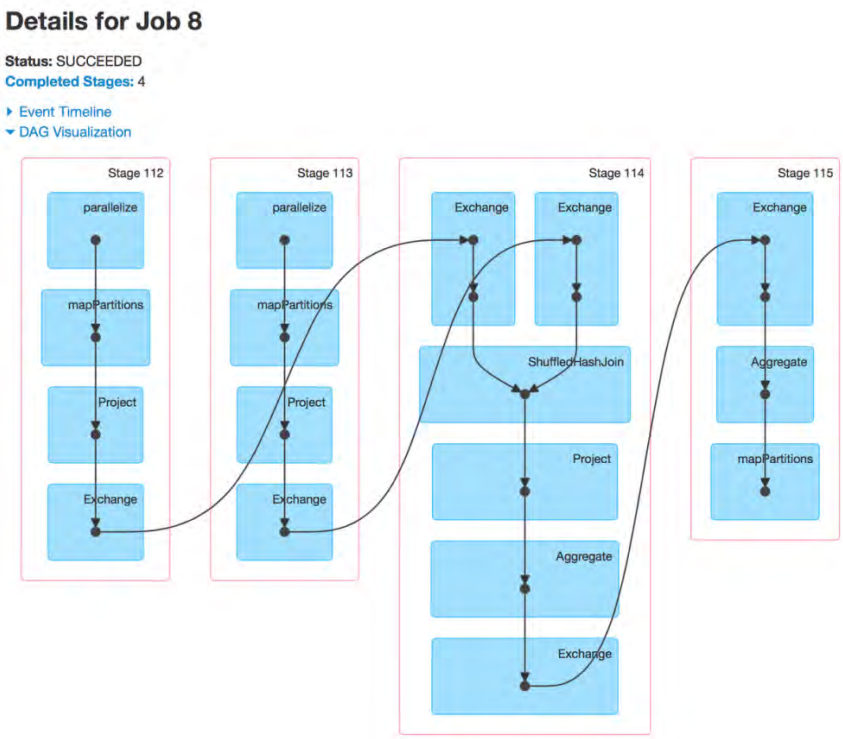
Specifieke eigenschappen van de pools kunnen worden aangepast. Het betreft de volgende eigenschappen: [21]

- Scheduling Mode: FIFO of FAIR modus voor de toebedeling van de resources
- Weight: daarmee kan het relatieve beslag van een pool op de beschikbare resources worden bepaald. 'Relatief' betekent hier het beslag door een pool in verhouding tot het beslag op de resources door de andere pools. Standaard hebben alle pools een gewicht van 1, maar dat is zoals gezegd aan te passen. Vooral als aan bepaalde pools een hoge mate van prioriteit moeten worden toegekend, kan dit tot uiting worden gebracht door aan de pool een zwaarder gewicht toe te kennen (bijvoorbeeld 1000).
- MinShare: tevens is het mogelijk aan iedere pool een minimum aandeel in de beschikbare resources toe te kennen. De Fair Scheduler zal dan alles in het werk stellen om eerst deze minimum aandelen te realiseren alvorens de rest van de resources worden toegekend op basis van het toegekende gewicht.

5.1.1 *DAG structuur*

Simpel gezegd maakt Spark van elke job (acties op de RDDs) een DAG en deze DAG wordt dan doorgegeven aan de DAG scheduler. En kort overzicht van de stappen die Spark zet is als volgt.

De DAG scheduler is een stage georiënteerde scheduler waarbij de DAG wordt opgedeeld in verschillende stages. Deze stages worden bepaald door middel van de operaties op de input data. Grofweg bestaan er twee soorten transformaties die kunnen worden uitgevoerd op de RDDs. Narrow en Wide transformaties. Hierbij zorgen de wide transformaties, e.g. reduceByKey, voor de afbakening van de stages. De narrow transformaties, e.g. map of filter, vereisen namelijk niet, in tegenstelling tot de wide transformaties, dat de data opnieuw geshuffeld hoeven te worden. Hierdoor kunnen meerdere taken gemakkelijker achter elkaar uit gevoerd worden, zie Figuur 14. [22]



Figuur 14. Voorbeeld van stages in een SparkJob

De stages worden daarna weer doorgespeeld aan de Task Scheduler. Deze start de taken in de stages via de cluster manager. De task manager heeft zelf geen informatie over de eventuele afhankelijkheden tussen de stages. Het aantal taken dat daadwerkelijk gestart wordt hangt dan ook weer af van het aantal partities in de input data én van het aantal beschikbare slaves/cores.

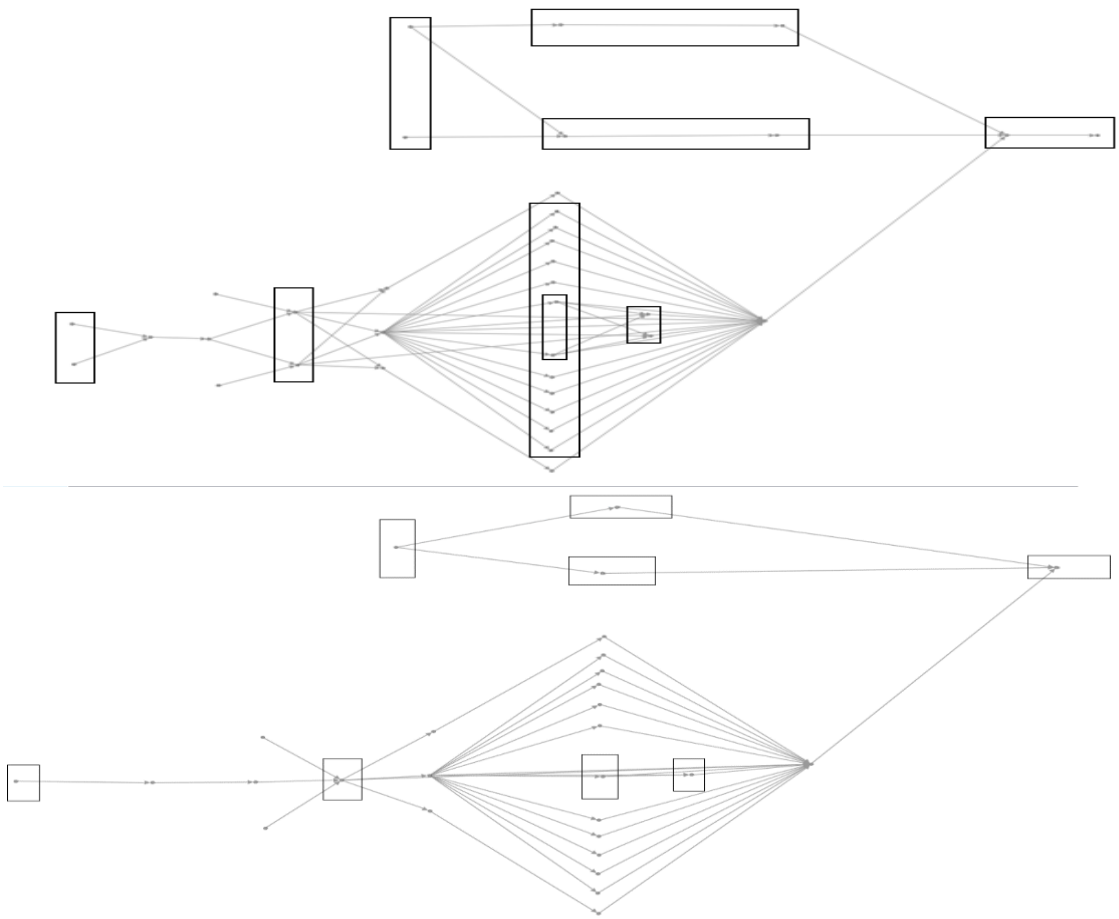
6 Werkwijze

In dit hoofdstuk wordt dieper ingegaan op de wijze waarop het DAG scheduling probleem is aangepakt. We geven hier eerst een korte beschrijving van een theoretische aanpak en daarna gaan we in op de werkwijzen en stappen die we gezet hebben met Spark.

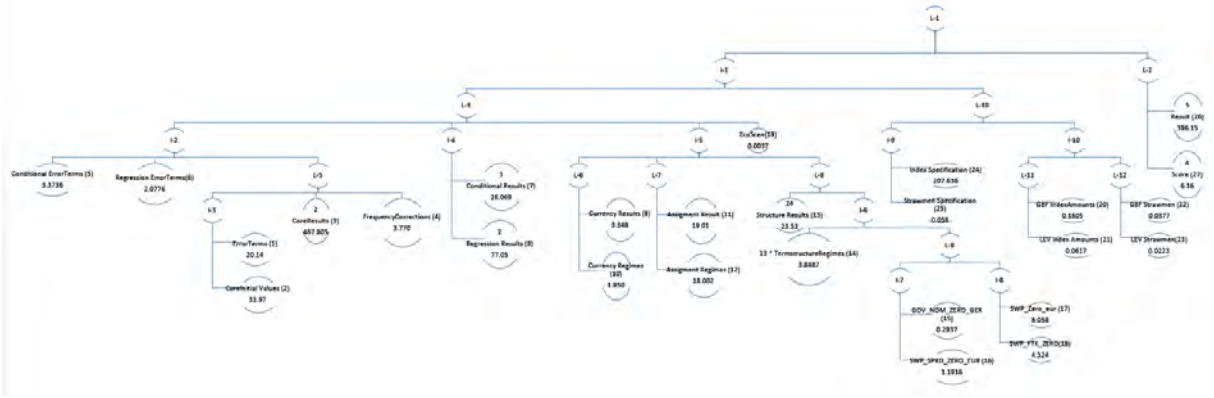
6.1 Theoretisch

In ons geval gaan wij uit van geen tot verwaarloosbare kosten voor de communicatie tussen processoren. Met andere woorden, er zijn geen kosten verbonden aan de keuze voor de parallellisatie van taken. Op grond van deze aanname, en omdat de methode meer gericht is op het zoeken naar sub-secties die te parallelliseren zijn, hebben we gekozen om voor het theoretische model de graaf decompositie methode te gaan gebruiken. Voor ALS hebben we gebruikt gemaakt van de performance test die een jaarlijkse simulatie uitvoert met een horizon van 15 jaar.

De eerste stap voor het algoritme was het herkennen en indelen van de CLANs. In Figuur 15 wordt de eerste iteratie weergegeven waarbij er zes onafhankelijke en drie lineaire CLANs worden geïdentificeerd. Na het voltooien van de decompositie krijgt men de parsetree zoals weergegeven in Figuur 16.

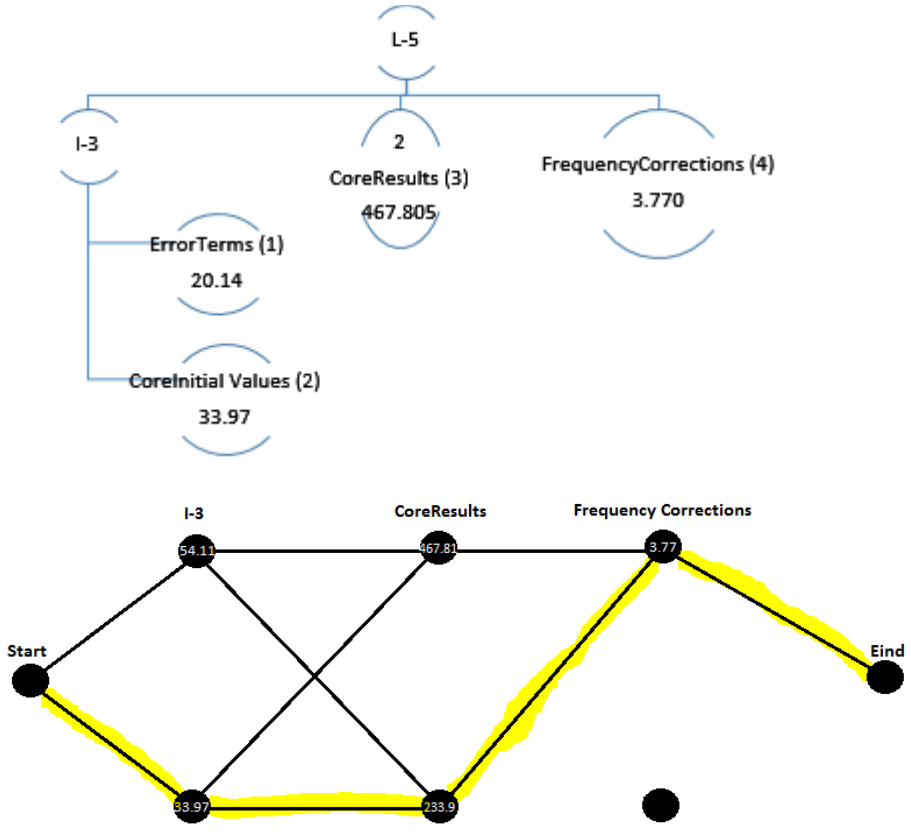


Figuur 15. Eerste iteratie in het identificeren van CLANs.



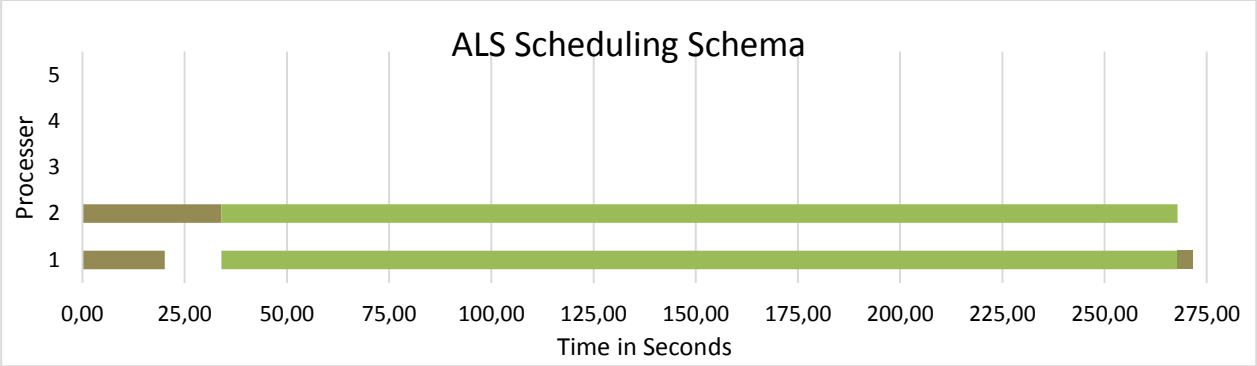
Figuur 16. Parsetree na decompositie van de ALS graaf (zie appendix voor grote versie)

Na het opstellen van de parsetree kan worden gekeken naar het plannen van de taken. Er wordt, zoals eerder vermeld, gebruik gemaakt van de depth-first strategie. Beginnend boven aan de boom volgt men de weg omlaag totdat de diepste Lineaire CLAN zijn gevonden en daar begint dan de scheduling stap. In ons geval is dat L-5. Wegens onze eerdere aanname, in paragraaf 6.1, over de verwaarloosbare communicatie kosten is de parallellisatie optie altijd de kortste weg. Figuur 17 toont deze CLAN plus de beslissingsgraaf. De verschillende lagen in de graaf representeren de mogelijkheden van het aggregeren of het parallelliseren van de taken. Hierbij is de eerste en bovenste laag de keuze voor het aggregeren en dus sequentieel uitvoeren terwijl de tweede en onderste laag staat voor de keuze van parallellisatie.



Figuur 17. Lineair CLAN en de beslissingsgraaf

Zodra de keuze voor aggregeren of parallelliseren is gemaakt volgt de stap van processor allocatie. In bovenstaande eerste iteratie krijgen we de indeling volgens Figuur 18. Na het verwerken van deze CLAN werkt men de overige lineaire CLANs van onder naar boven af tot men bij de bovenste L-1 CLAN is aangekomen.



Figuur 18. Indeling na eerste iteratie ALS graaf

6.2 Praktisch

In dit tweede stuk zal dieper worden ingegaan op hoe we te werk zijn gegaan met Spark als framework. Er wordt beschreven welke stappen we genomen hebben en we leggen uit waarom we deze stappen hebben gezet.

6.2.1 *Introductie*

Omdat Spark niet een mogelijkheid heeft om direct via C# te communiceren, worden we geconfronteerd met de uitdaging om een 'omweg' te vinden om toch een executable in C# te parallelliseren. De mogelijke talen die gebruikt konden worden waren Java, Python en Scala. Wegens ervaring in Python is er gekozen om aan de slag te gaan met Python.

Als IDE hebben we gekozen om te werken met Microsoft Visual Studio Professional 2013. Om dit te laten werken met Apache Spark moesten we een kleine aanpassing tot stand brengen. We moesten er voor zorgen dat de IDE Pyspark herkent en dat de connectie met Spark gemaakt kan worden bij het uitvoeren in debug mode. Hiervoor moesten we enkele paden toevoegen in de "Search Paths" van Visual Studio.

6.2.2 *Installatie Spark*

Zoals vermeld in paragraaf 4.3.2 zijn er meerdere manieren om Spark te installeren. Voor dit project hebben we ervoor gekozen het als standalone te gaan gebruiken van versie 1.4. Op deze manier is het niet afhankelijk van andere tools of frameworks en kunnen we alsnog de parallellisatie testen die we willen.

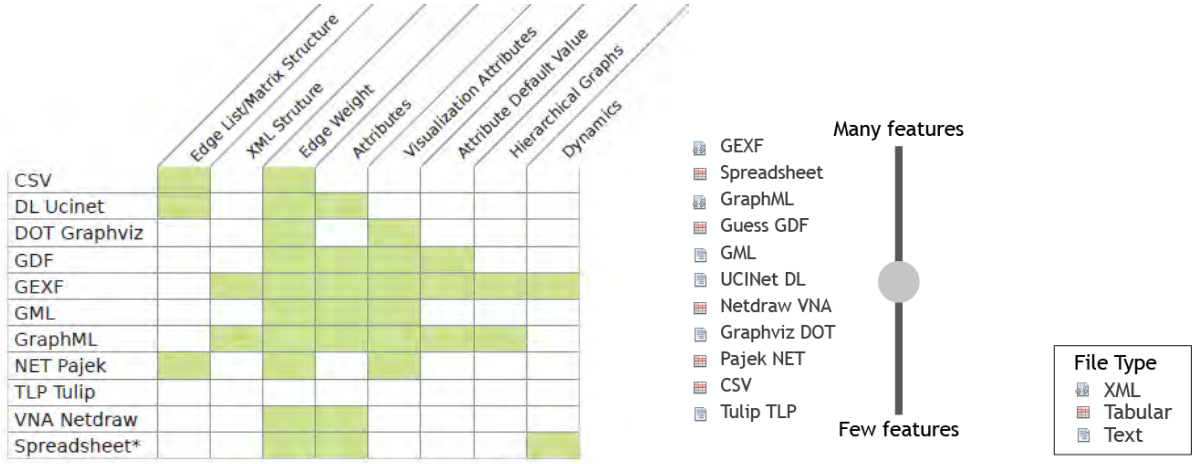
Om Spark daadwerkelijk te installeren waren er een paar dingen noodzakelijk. Ten eerste moet een versie van Spark ter beschikking staan. Ten tweede is een up-to-date versie van Java noodzakelijk en ten slotte, omdat we het gaan gebruiken op Microsoft Windows, hebben we Cygwin nodig. Cygwin is een collectie van GNU en open source tools die een functionaliteit bieden, vergelijkbaar met Linux, voor Windows [23]. Cygwin is noodzakelijk voor het, indien gewenst, bouwen van Spark, maar ook voor het activeren van de Master en slave nodes. Het werkt namelijk met bash commando's die via Windows eigenlijk niet uitgevoerd kunnen worden. Cygwin biedt hier de oplossing voor. Van de Apache Spark website kunnen ook pre-build versies worden gedownload van Spark. Dit scheelt een hoop werk met het zelf compileren van de source code.

6.2.3 *Export/Import DAG structuur*

Een andere belangrijke stap was het herkenbaar maken van de DAG structuur voor Spark. Zodra de structuur bekend zou zijn, kon het inplannen van de taken geregeld worden door Spark. In de tussentijd moesten we nog wel uitvinden op welke wijze we de executable konden laten functioneren binnen Python.

Om te beginnen moesten we dus de DAG structuur van ALS zien te splitsen van het programma zelf. Er was wel bekend hoe de structuur was opgebouwd, maar er was geen optie om dit visueel weer te geven of om dit ergens op te slaan. Om dit te kunnen regelen hebben we de code van het ALS programma aangepast. We hebben een optie toegevoegd die aan en uit gezet kan worden.

Deze aanpassing is gedaan in C#. Wanneer er een simulatie gekozen wordt zal er worden gekeken welke stappen er allemaal gezet moeten worden. Deze stappen worden vervolgens uitgevoerd en per stap wordt er dan weer gekeken wat de afhankelijkheidsrelaties zijn. Aan de hand van deze informatie kunnen we nu een DAG structuur exporteren. Er is gekozen om dit naar XML te doen volgens het GEXF format. Dit is een taal die gebruikt kan worden voor het beschrijven van complexe netwerk structuren. Het is afkomstig van een Gephi project en voor het eerst in gebruik genomen in 2007 [24]. De keuze is gebaseerd op het feit dat dit formaat de meeste mogelijkheden biedt in vergelijking met alternatieven, zie Figuur 19.



Figuur 19. Vergelijking Graph formaten [25].

Nu we de DAG structuur hebben was de volgende stap om er voor te zorgen dat Spark de structuur zou kunnen herkennen en inlezen. Zoals beschreven staat in paragraaf 5.1.1 worden, binnen Spark, de knopen van een DAG gekarakteriseerd door middel van transformaties op de RDD. Er moest dus een manier worden gevonden om de geëxporteerde DAG structuur te vertalen in transformaties voor Spark.

Omdat we eerder gekozen hadden om Spark te gaan gebruiken via Python, zal er eerst een vertaling moeten komen van de DAG in XML naar Python. Hiervoor hebben we gebruik gemaakt van NetworkX [26], een Python module bedoelt voor het maken, manipuleren en bestuderen van complexe netwerken. We hebben gekozen voor deze module vanwege de simpele werking en creatie van DAG structuren.

Voor het importeren van de XML hebben we een Node class, een DAG class en een ReadXML Class gemaakt. Er is gekozen om een aparte DAG class te maken, in plaats van de standaard van NetworkX. Dit is gedaan omdat we op deze manier alsnog de DAG objecten kunnen gebruiken van NetworkX, maar ook nog zelf functies kunnen toevoegen. De XML Import class importeert een XML bestand en gaat deze structuur af om vervolgens een DAG object te vullen met de knopen en dependencies.

Nu de DAG structuur van een ALS simulatie geëxporteerd en geïmporteerd kan worden is de volgende stap om de structuur over te brengen naar Spark. Zoals eerder vermeld herkent Spark de verschillende knopen door middel van transformaties op de data. In andere woorden, er moest dus een mapping komen van de knopen in het DAG object naar een transformatie met Spark.

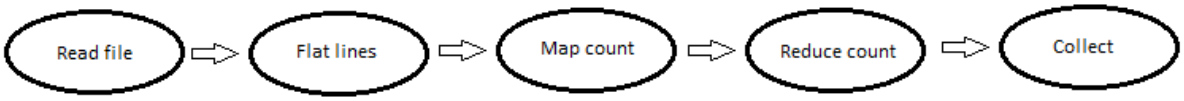
6.2.4 *Multi-core*

Voor ontwikkelaar/test doeleinden biedt Spark de mogelijkheid om een lokale versie te gebruiken. Op deze manier is het niet noodzakelijk om een Master op te starten. Tevens wordt bij het uitvoeren van de geschreven code een standalone versie opgestart. Op deze manier maken we gebruik van Spark met maar één master/knoop en zijn de testen dus Multi core.

6.2.4.1 **Wordcount**

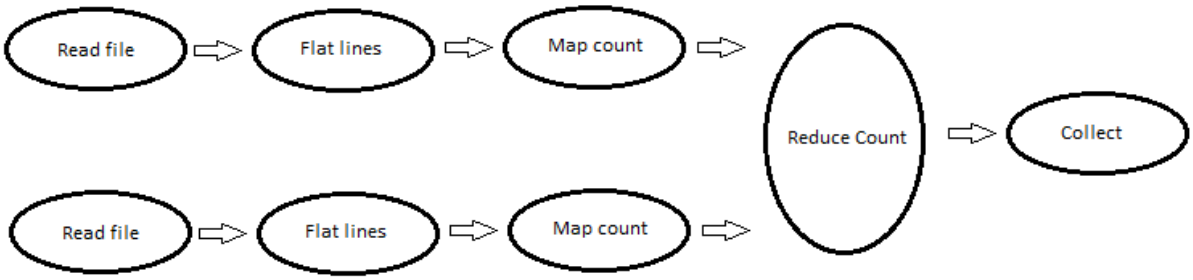
Om te controleren of het inlezen van de DAG en het uitvoeren in de juiste volgorde en op correcte wijze verliep, zijn er simpele tests uitgevoerd. De DAG structuur van de XML werd ingelezen en weggeschreven naar een DAG object. Een zelfgeschreven functie liep deze DAG af en zocht naar de eerste knopen die uitgevoerd konden worden, knopen zonder afhankelijkheden. Deze knopen hadden namen, corresponderend met transformatie functies die we hadden aangemaakt. Aan de hand van deze namen werd gecontroleerd of de juiste knopen werden geretourneerd.

Als test hadden we kozen voor een Wordcount programma. Deze bestond uit vijf achter elkaar opvolgende transformaties. Elke knoop had dus één kind en één ouder, zie Figuur 20. Deze test hebben we eerst uitgevoerd met de Wordcount functies in main class om de functionaliteit te testen. Daarna, om het generiek te houden, hebben we de functies verplaatst naar een losse Function class. Hierdoor werd de DAG uitgelezen in de main class en werd hier functies aangeroepen uit een andere class.



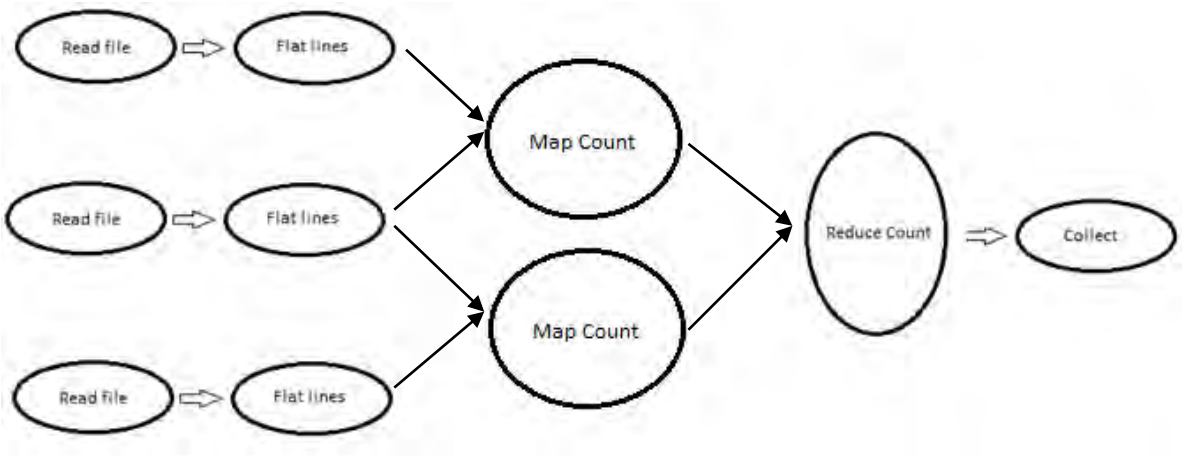
Figuur 20. Wordcount voorbeeld

Om wat uitgebreider te testen is het Wordcount voorbeeld uitgebreid zoals weergegeven in Figuur 21. Er is een tweede lijn met transformaties toegevoegd. Hierdoor ontstaat er een knoop met twee kinderen. Deze knoop, 'Reduce count', heeft hierdoor twee dependenties en kan dus niet verder gaan met de collect transformatie totdat beide kinderen afgerond zijn.



Figuur 21. Wordcount voorbeeld, meerdere kinderen

Een andere variant die getest werd is het scenario waarbij bepaalde knopen niet alleen meerdere kinderen hebben, maar ook dat sommige knopen meerdere ouders hebben. Wederom is het wordcount voorbeeld hiervoor gebruikt, zie Figuur 22.



Figuur 22. Wordcount voorbeeld, meerdere kinderen en ouders

Om het proces automatischer te laten verlopen is ervoor gekozen om aan de hand van het DAG object, een Function Classs te maken die per knoop in de DAG een transformatiefunctie genereert. Na het inlezen van de DAG, behorend bij een simulatie, zal er worden gekeken of de Functions class voor deze simulatie al bestaat. Indien het nog niet bestaat zal deze worden aangemaakt aan de hand van de XML. Door deze keuze hoeft er maar één keer een nieuw Function-object te worden aangemaakt voor vergelijkbare simulaties.

6.2.4.2 **ALS DAG**

Nadat we de testen hadden afgerond met het wordcount voorbeeld was het een geschikt moment om een meer ingewikkelde DAG structuur te gaan gebruiken. De vervolgstap bestond uit het testen met een ALS DAG. Er is voor gekozen om met een mock-up versie van ALS te werken in plaats van het echte programma. Het uitvoeren van de losse onderdelen van een ALS simulatie is niet het probleem, het hoofddoel is het kunnen draaien van een C# executable op Spark voor Microsoft Windows. Om deze reden hebben we er voor gekozen om een mock-up versie te gebruiken van ALS die de DAG structuur simuleert en die voor de vergelijking met de huidige situatie de looptijden overneemt van ALS zoals het momenteel draait.

Het mock-up programma is een klein programma geschreven in C# dat aan de hand van de DAG die door ALS geëxporteerd is de runtijd simuleert van de oorspronkelijke knoop. Evenwel zonder de echte handelingen uit te voeren. De transformaties die Spark de DAG structuur geven moeten nu de executable aanroepen én meegeven welke stap het van de ALS moet nabootsen.

Om de mock-up nog wat realistischer te maken is deze nog verder uitgebreid. In ALS wordt bij elke stap data ingelezen en weggeschreven. In het mock-up programma hebben we de optie toegevoegd voor de communicatie met een SQL database. Op deze manier wordt de communicatie met de database ook meegenomen in de vergelijking. Naast het rekenwerk in ALS is de communicatie met de SQL database een belangrijk onderdeel van de stappen.

Om een goede vergelijking te maken tussen Spark en het huidige systeem hebben we enkele vergelijkingen gemaakt. Ten eerste hebben we ALS uitgevoerd op een enkele core en daarna via Alchemi multicore. Ten slotte hebben we de gegenereerde XML DAGs met de looptijden van ALS in ons Python programma ingeladen. Deze

werden vervolgens verwerkt tot een Spark job. Door middel van deze testen werd gekeken of het Python programma overweg kon met de meer ingewikkelde DAG structuren. Tevens kon er gelijk gekeken worden hoe Spark functioneert in vergelijking met de huidige situatie.

6.2.5 *Multi-Machine*

Nadat we een werkende versie hadden van de ALS graaf via Spark multi-core processing, hebben we geprobeerd om deze versie naar multi-machine te verplaatsen. Voor deze testen moesten we eerst een cluster tot onze beschikking hebben met minimaal twee knopen. Hiervoor hadden we twee opties; we konden zelf ons cluster opzetten of we konden in de cloud een cluster huren. Voor Ortec Finance was het goede beheer binnen het cluster één van de eisen. Bij eigen clusters ligt het volledige beheer bij onszelf en om deze reden hebben we eerst een eigen cluster geprobeerd op te zetten.

6.2.5.1 **Virtuele Machines**

Voor een eigen cluster hadden we minimaal twee machines nodig waar we Spark op konden installeren. Binnen Ortec Finance waren meerdere virtuele machines beschikbaar om een cluster op te zetten. Deze machines waren al eerder gebruikt voor andere cluster doeleinden en konden dus prima als cluster functioneren voor dit project. Zoals al eerder vermeld in paragraaf 6.2.2 maakten we gebruik van een pre-build versie van Spark 1.4. Op deze manier hoefden we geen tijd te besteden aan het compileren van de code.

Met deze virtuele machines en een gecompileerde versie van Spark tot onze beschikking, was de volgende stap om de Master te activeren. De Master wordt geactiveerd door een bash script uit te voeren, een sh bestand. Microsoft Windows heeft echter zelf geen ingebouwde ondersteuning voor sh bestanden. Om dit bash script uit te voren moest er gebruik worden gemaakt van een hulpprogramma van derden, in ons geval gebruikten wij Cygwin.

Hier liepen we echter tegen een probleem aan. Sinds updates van Cygwin en Spark kwam het voor dat het starten van de launch scripts, de sh bestanden, niet functioneerde. Er is een gebrek aan ondersteuning voor de launch lib. Helaas was het op het moment dat we hier tegen aan liepen niet meer mogelijk een oudere versie van Cygwin terug te halen. Er kon alleen één versie terug worden gedownload, we hadden echter drie eerdere versies nodig. Om dit probleem op te lossen hebben we alternatieven geprobeerd voor Cygwin, waaronder Gow (Gnu On Windows)[5] een bekend en populair alternatief. Helaas gaf dit hetzelfde probleem.

6.2.5.2 **Cluster in de Cloud**

Nadat we zelf geprobeerd hadden een cluster op virtuele machines op te zetten en we moesten concluderen dat deze poging mislukte, zijn we overgestapt op een cluster in de Cloud. Onze conclusie was dat het uitvoeren van testen op een reeds bestaand cluster meer voordelen bood dan het laten functioneren van een eigen cluster.

Voor een Cloud oplossing zijn er meerder partijen die Spark clusters aanbieden. Voor ons was de eis van Microsoft Windows OS een beperkende factor. De meest bekende optie, Amazone EC2[6], bied helaas geen mogelijkheid voor Spark op Microsoft Windows. Google Cloud Dataproc[7] heeft wel een makkelijkere interface in vergelijking met Amazone EC2, maar heeft ook geen mogelijkheid om Spark op Microsoft Windows te gebruiken.

[5] https://github.com/bmatzelle/gow/wiki
[6] https://spark.apache.org/docs/latest/ec2-scripts.html
[7] https://cloud.google.com/dataproc/

De uiteindelijk gekozen optie was een voor de hand liggende, uitgaande van onze OS eis. We moesten Microsoft Windows OS gebruiken en daarom keken we naar de Cloud mogelijkheden van Microsoft. Microsoft biedt een Apache Spark mogelijkheid voor Azure HDInsight[8]. Azure HDInsight is een service, aangeboden door Microsoft in samenwerking met Hortonworks, die oorspronkelijk Apache Hadoop clusters aanbiedt. Sinds juli 2015 biedt Microsoft een preview versie van Apache Spark voor publiek gebruik. De Spark preview versie biedt een optie voor remote desktop benadering waardoor je direct verbinding kan maken met de Master van het cluster. Via de master kunnen vervolgens de Spark jobs worden gestart.

6.2.5.2.1 Aanpassingen

Om er voor te zorgen dat we ons programma konden gaan testen op het cluster moesten er eerst nog wat voorbereidend werk worden verricht. Het programma moest worden losgekoppeld van Visual Studio. Tot nu toe maakte het nog verbinding met Spark via Visual Studio, maar dit moet direct via de Master gaan. Verder moeten de geschreven classes van ons programma worden gekopieerd naar de Master en op de goede locatie worden gezet.

Voor het loskoppelen was een kleine aanpassing van de Main.py code voldoende. De verbinding met de Master werd uit de code gehaald en de input voor het programma moest voortaan via een argument worden meegegeven in plaats van een keuze menu in Visual Studio.

Om er voor te zorgen dat de installatie en het gebruik van ons Python programma gemakkelijk kan verlopen hebben we de geschreven Python classes in een package gestopt. In de package zitten de ReadXML classes, de DAG class, de Node class en ook de Function classes. Door de package met de 'develop' optie te installeren kunnen er nog aanpassingen worden gedaan in de source code die gelijk worden meegenomen in het systeem.

6.2.5.2.2 Wordcount

Nadat we de nodige aanpassingen hadden gemaakt was het tijd om te gaan testen. Om consequent te blijven in onze tests zijn we begonnen met het Wordcount voorbeeld zoals in Figuur 22. Op deze manier konden we testen of ons Framework de juiste structuren ook doorgeeft op een cluster. De enige extra aanpassingen die we moesten doen, om het voorbeeld werkend te krijgen, was het txt bestand in het HDFS te zetten. Hiervoor maakten wij gebruik van het "hadoop fs –copyFromLocal" commando. Op deze manier konden we een lokale txt naar het file systeem kopiëren. Zodra de file in het HDFS stond, konden we het Wordcount voorbeeld zonder problemen uitvoeren.

6.2.5.2.3 ALS DAG

Nadat we ons op basis van het Wordcount voorbeeld ervan verzekerd hadden dat ons Python programma goed functioneerde, was de volgende stap het testen met een ALS DAG. Hier liepen we alleen al snel tegen een probleem aan. Het programma werd opgestart, naar Spark gestuurd en ook als voltooid weergegeven, alleen gebeurde dit alles in minder dan een minuut tijd. Dit gaf ons de indruk dat er iets niet goed werd uitgevoerd, de code was veel te snel, uitgaande dat de kortste taak in de DAG al bijna 3 minuten moest duren. Het bleek al snel dat Spark de ALS DAG structuur wel herkent, maar op een of andere manier de functies niet uitvoerde en steeds oversloeg.

Na het beter bekijken van de code en het uitvoeren van enkele tests werd het al iets duidelijker. Het probleem leek zich voor te doen bij het aanroepen van de zelfgeschreven functie met argumenten. En meer

[8] https://azure.microsoft.com/nl-nl/services/hdinsight/apache-spark/

specifiek, het leek stuk te lopen op de argumenten. Zodra we een test run deden zonder argumenten, werd de functie gewoon herkend.

De Map functie in Spark, die we mede gebruiken om de taken af te handelen, accepteert zelf alleen een functienaam als argument, maar geen functienaam plus argumenten. Ons programma werkt door het aanroepen van een zelf geschreven functie mét parameters. De parameters werken als indicatie welke stap in de ALS executable moeten worden uitgevoerd. Om dus toch een functie met parameters mee te geven aan de Map functie van Spark, hebben we gebruik gemaakt van “partial”.

Partial is een onderdeel van de Python module functools[9]. Dit is een module voor hogere-orde functies. Met andere woorden, voor functies die zelf functies als input of output hebben. Met behulp van partial kunnen we aan hand van een naam en parameters een nieuw functie object retourneren. Dit object kon dan weer worden meegegeven aan de Map functie, de geretourneerde functie heeft zelf namelijk geen invoer vereiste.

Tijdens het lokaal testen ging dit zonder problemen, maar op het cluster lijkt deze aanpak met partial voor problemen te zorgen. Het lijkt erop dat het cluster niet overweg kan gaan met closures. Wegens de beperkte tijd in dit onderzoek én dat het enkel nog een praktische aanpassing betreft zijn we niet verder gegaan met het volledig testen van de ALS graaf. In hoofdstuk 8 Conclusie/Discussie zal kort een voorstel worden gedaan hoe dit laatste probleem verholpen zou kunnen worden.

[9] https://docs.python.org/2/library/functools.html

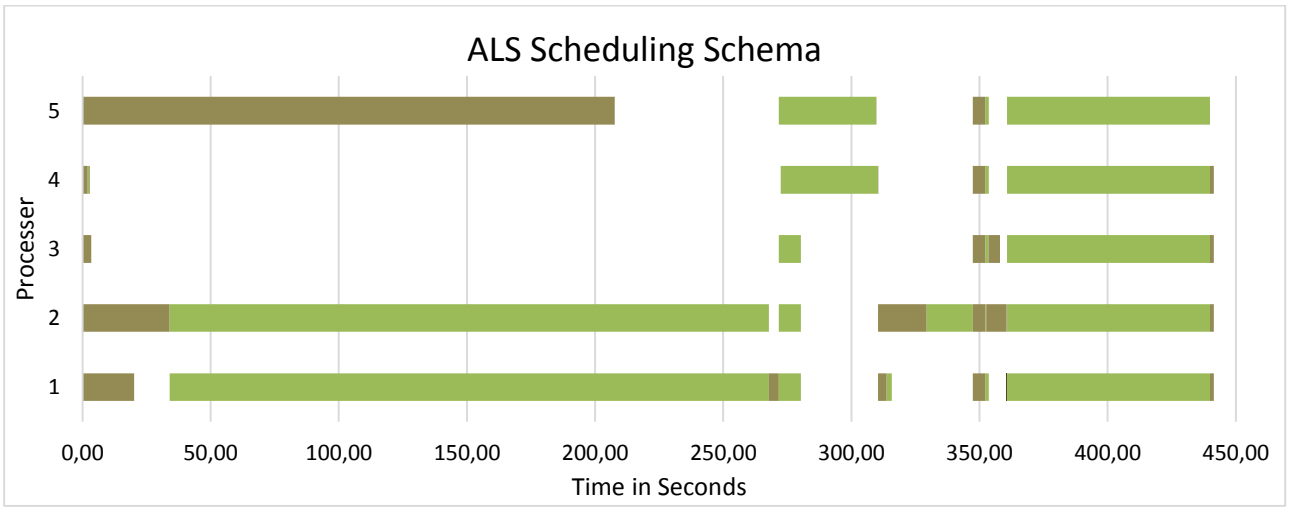
7 Resultaten

In paragraaf 6.1 hebben we aan de hand van de CLANS methode, handmatig, een optimum bepaald voor een ALS DAG scheduling probleem. Hierbij hadden we de communicatietijd tussen processoren als verwaarloosbaar aangenomen. In Tabel 2 is in tekst een indeling weergegeven van het uiteindelijke schema van de ALS DAG. Dit schema is een voorbeeld hoe Ortec deze specifieke DAG zou moeten uitvoeren willen ze de runtime van hun programma minimaliseren.

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
Error Terms	Core Inital Values	Conditional ErrorTerms	Regression ErrorTerms	Index amounts Specification
CoreResults	CoreResults		Strawmen Specification	
Frequency Corrections			GBF INdexAmounts	GBF Strawmen
Conditional Results	Conditional Results	Conditional Results	Regression Results	Regression Results
Currency Results	Assigment Results		LEV Index Amounts	LEV Strawmen
Currency Regimes	Assigment Regimes			
Structure Results	Structure Results	Structure Results	Structure Results	Structure Results
Termstructure Regimes	GOV_NOM_ZERO_GER	SWP_SPRD_ZERO_EUR	Termstructure Regimes	Termstructure Regimes
	SWP_ZERO_EUR	SQP_FTK_ZERO		
EcoScen				
Result	Result	Result	Result	Result
Score	Score	Score	Score	

Tabel 2. Indeling van de ALS knopen voor 5 processoren

In Figuur 23 is hetzelfde schema weergegeven als in bovenstaande tabel. Echter ligt hier de nadruk op de start –en eindtijd van de taken, en niet op welke taak wanneer wordt uitgevoerd. De wisseling van kleur, bruin en groen, geeft aan wanneer er van taak wordt gewisseld op een processor. Er is hier goed te zien dat de totale tijd van de DAG neerkomt op ongeveer 440 seconden wat 6 minuten 20 seconden is.

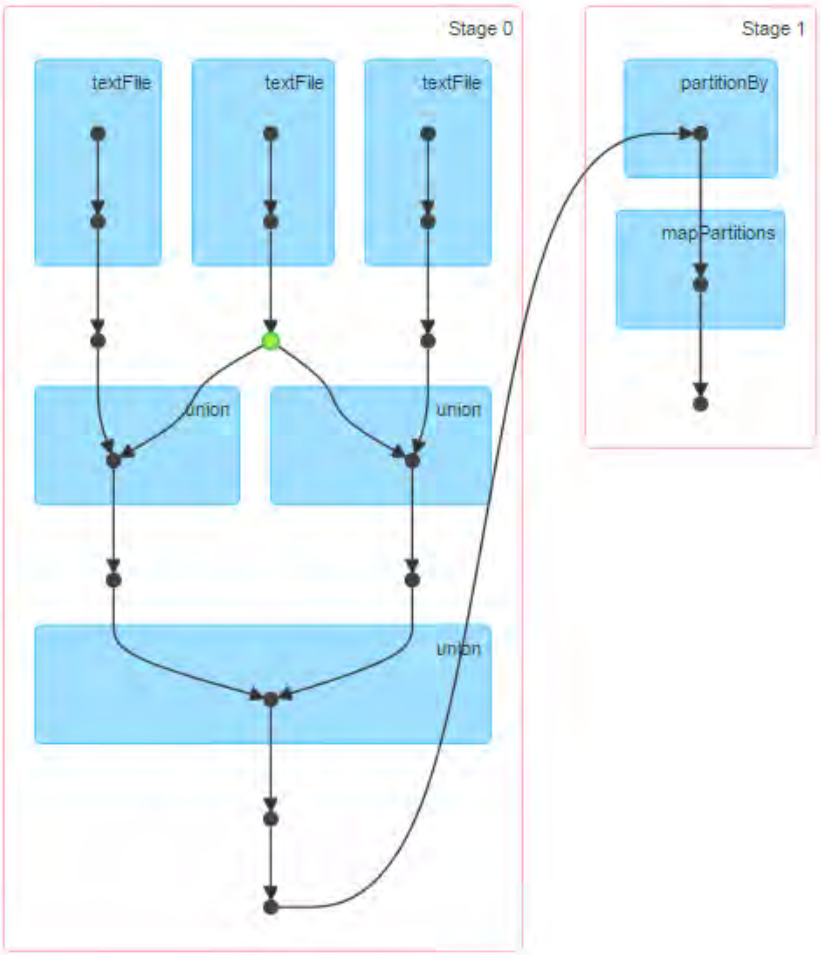


Figuur 23. Visuele indeling van de ALS taken voor 5 processoren start -en eindtijden

Een mooie nieuwe functie die vanaf Spark 1.4.0 beschikbaar is, is de DAG visualisatie optie. Zodra een taak naar Spark wordt gestuurd en de DAG structuur is herkend, geeft het Spark dashboard de optie voor DAG Visualisatie, zie Figuur 24.

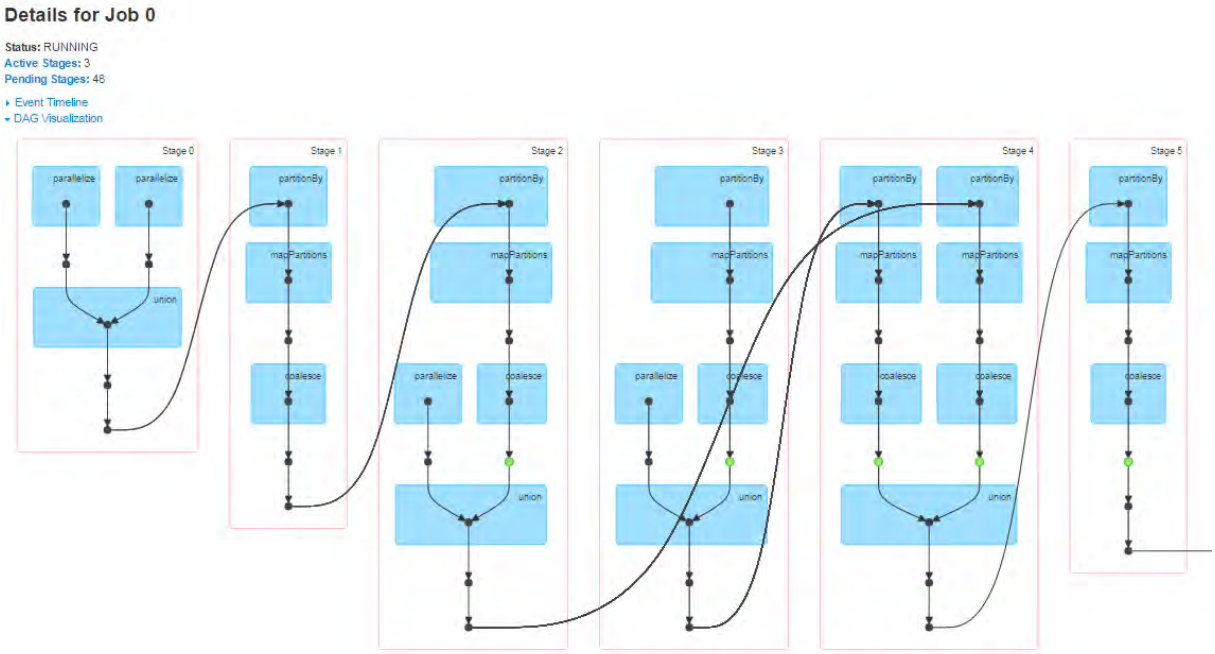
Figuur 24. Deel van het Spark Dashboard met de optie DAG Visualisatie.

De DAG visualisatie met Spark van het uiteindelijke Wordcount voorbeeld, zoals besproken in paragraaf 6.2.4.1., is te zien in Figuur 25. Hier is goed te zien dat de oorspronkelijke DAG, zie Figuur 22, volledig wordt herkend in Spark.



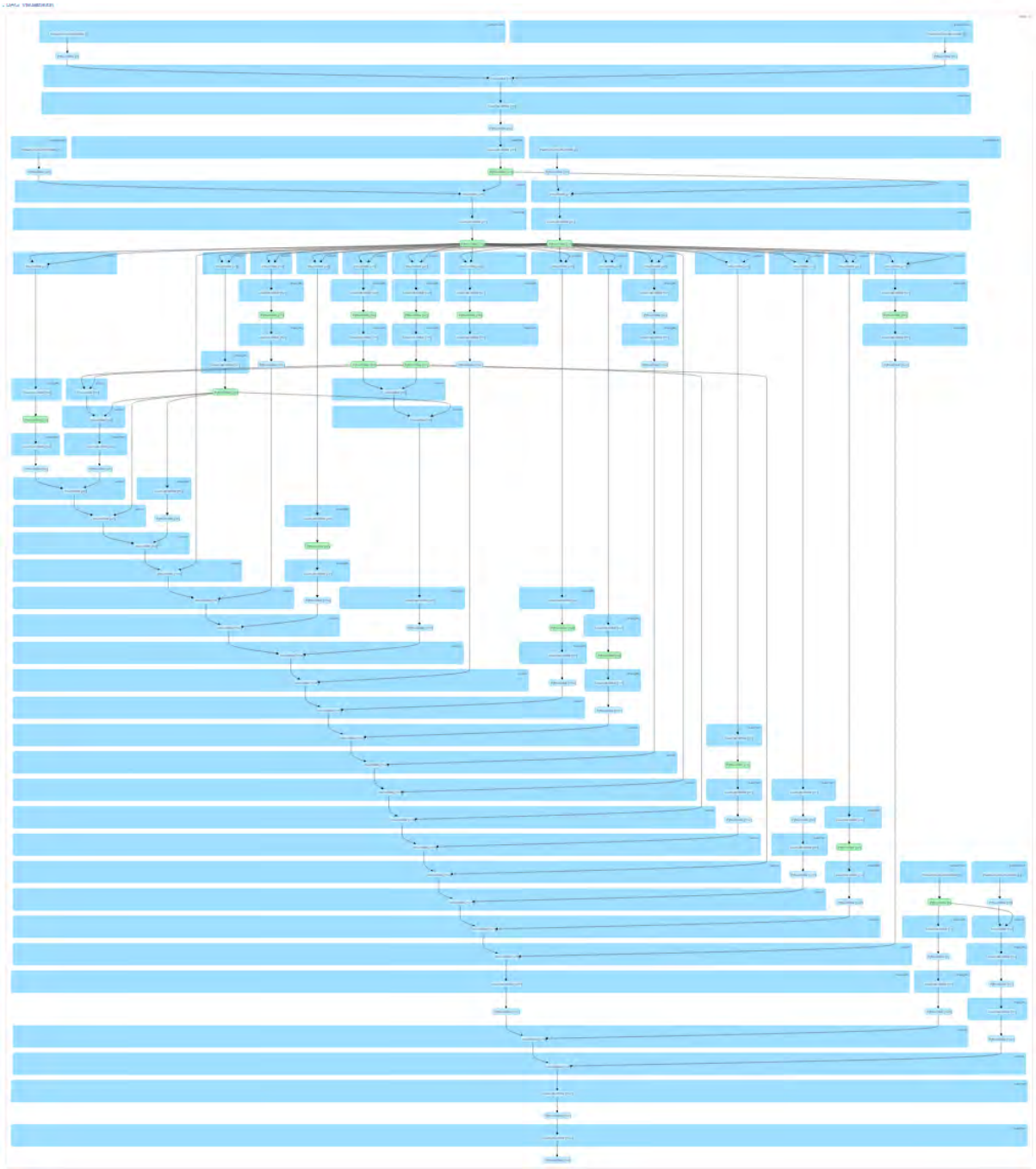
Figuur 25. Spark DAG Visualisatie Wordcount voorbeeld

Ook voor ALS is het mogelijk om een visualisatie van de structuur met Spark te krijgen. Echter hebben we hier een kleine aanpassing gemaakt. Aangezien de ALS DAG erg veel knopen heeft en in het bijzonder knopen met meerdere kinderen, was de DAG in 51 stages opgebouwd. In Figuur 26 is een gedeelte te zien van deze structuur met de vele stages. Door dit hoge aantal stages was het niet mogelijk om alle stages naast elkaar te krijgen voor een volledig visueel overzicht.



Figuur 26. Gedeeltelijke DAG visualisatie van ALS met Spark

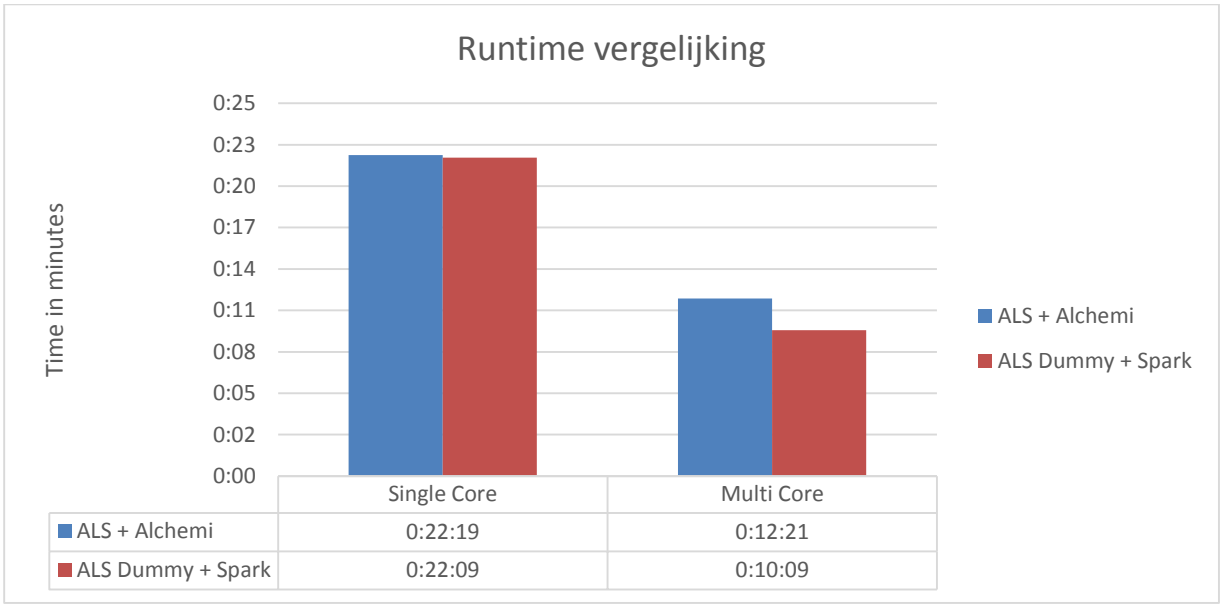
Door een kleine aanpassing in onze code is het uiteindelijk toch gelukt om een volledig overzicht te creëren. We hebben de code zo aangepast dat er nog maar één stage word herkend door Spark. Op deze manier zal bij de DAG visualisatie alles onder elkaar worden gezet. De verschillende knopen zijn klein, maar een overzicht is te vinden in Figuur 27. De structuur van de oorspronkelijke ALS DAG zoals in Figuur 1 is niet direct te herkennen, maar na wat grondiger bestuderen komt deze ook overeen met de oorspronkelijke structuur.



Figuur 27. Spark DAG Visualisatie ALS

`

Om een beter beeld te krijgen hoe Spark werkt ten opzichte van het huidige systeem, is er een vergelijking gemaakt met de runtijden. In Figuur 28 hebben we de runtijden van ALS met Alchemi single core en multi-core gezet tegenover de runtijd van de ALS mockup en Spark. In de single core vergelijking is goed te zien dat beide systemen vergelijkbaar presteren. Dit was ook te verwachten. In de multicore vergelijking is wel een duidelijker verschil te zien. Hier valt op dat met Spark er een lagere runtijd wordt behaald ten opzichte van Alchemi. Dit verschil zou verklaard kunnen worden door de manier van scheduling. Zoals eerder al benoemd doet Alchemi dit random en Spark gebruikt standaard de FIFO modes. De verschillende manieren van het inplannen van de taken kan dus zorgen voor een snellere afhandeling van de totale job.



Figuur 28. Runtime vergelijking Alchemi en Spark

8 Conclusie/Discussie

Het kunnen gebruiken van de Big Data technieken voor het parallelliseren van Microsoft Windows applicaties zou een grote stap voorwaarts zijn. Het zou mogelijkheden bieden om op een relatief makkelijke manier programma's in een cluster te laten uitvoeren. Tevens zou er geen noodzaak meer zijn om producten van derden af te nemen, omdat deze open source Big Data technieken alle toepassingen bieden die mogelijk zouden zijn.

Via dit onderzoek werd er gestreefd een antwoord te vinden op onze onderzoeksvraag/onderzoeksdoel. Zoals vermeld is in de introductie is de vraag of er mogelijkheden zijn voor het parallelliseren van Microsoft Windows applicaties met behulp van Big Data technieken. In dit hoofdstuk zullen we dan ook kort antwoord geven op deze vraag. Tevens zal er een aanbeveling worden gedaan over eventuele aanpassingen voor een vervolg op dit onderzoek.

8.1 Bevindingen

Met onze Wordcount en ALS testen op multi-core met Spark hebben we laten zien dat ons Python framework goed functioneert. Met behulp van ons framework is het mogelijk om structuren van executables in te lezen en door te geven aan Spark. Deze is in staat om de juiste structuur te herkennen en de verschillende taken uit te voeren. Het is gelukt met Spark een Microsoft Windows applicatie multi-core te kunnen uitvoeren. Hierbij is de prestatie enkele percentages beter in vergelijking met de huidige techniek die nu gebruikt wordt, Alchemi.

De multi-machine testen van het Python framework met behulp van het Wordcount voorbeeld verliepen zonder problemen. Dit geeft aan dat het framework, zonder executable, goed functioneert op het cluster. Als we echter mét executable gaan testen lopen we tegen problemen aan. De aanpak via Partial zorgt er voor dat er closures ontstaan en hier lijkt het cluster van Spark niet goed mee te werken. Het zou hier om een praktische aanpassing gaan om er voor te zorgen dat we geen gebruik meer maken van closures. Aangezien het Python framework goed functioneert met voorbeelden waar geen closures worden gebruikt, kunnen we met enige zekerheid vaststellen dat ook de multi-machine testen met ALS mogelijk moeten zijn met Spark.

8.2 Aanbevelingen

De eerste aanbeveling die we doen betreft het wegwerken van de closures in de Python code. De optie waaraan gedacht kan worden is het volledig elimineren van de Partial functie. Momenteel wordt deze dus gebruikt om een functie object terug te geven waarbij argumenten voorkomen die corresponderen met de stappen in de DAG. Om de Partial functie te kunnen vermijden is het dus noodzakelijk om op een andere manier deze argumenten mee te kunnen geven aan de functies.

De functies krijgen, naast de door ons gegeven parameters, ook de RDD meegegeven. De RDD wordt momenteel alleen gebruikt om te kunnen bepalen hoeveel taken een bepaalde knoop heeft. Aan de hand van deze hoeveelheid zal de knoop worden opgedeeld. Om onze eigen parameters te vermijden kunnen we proberen gebruik te maken van deze RDDs. Er zou geprobeerd kunnen worden om de informatie die nodig is voor de knopen in de RDDs te stoppen. Hierbij zou gebruik gemaakt kunnen worden van een lijst met tuples. Dit zou naar schatting in ongeveer een week getest kunnen worden.

8.3 Beperkingen

Het is ons opgevallen dat zodra we grotere DAG voorbeelden gingen draaien de opstart tijd van de Spark job erg lang kon duren. We hebben dit helaas alleen kunnen testen met de Local Mode van Spark. Hoe het

schaalt als Spark gebruikt zou worden in het cluster, StandAlone of Yarn, is dus niet duidelijk. Het geeft echter wel een indicatie dat als de DAGs groter en ingewikkelder zouden worden dit wellicht problemen zou kunnen opleveren.

Ondanks dat de Big Data technieken vooral bedoeld zijn voor de datahandelingen lijkt het dus zeker mogelijk om deze technieken ook te gebruiken voor andere doeleinden. Er is duidelijk geworden dat met behulp van een framework structuren van executables kunnen worden ingelezen en herkend door Spark, die op zijn beurt deze taken in de juiste volgorde kan afwerken. Er zal nog verder wat testwerk gedaan moeten worden, maar het lijkt niet onmogelijk.

9 Bibliografie

[1] S. Ryza, „Estimating Financial Risk with Apache Spark - Cloudera Engineering Blog," Cloudera, 14 07 2014. [Online]. Available: http://blog.cloudera.com/blog/2014/07/estimating-financial-risk-with-apache-spark/. [Geopend 07 2015].

[2] C. L. McCreary, A. A. Khan, J. Thompson en M. E. McArdle, *A Comparison of Heuristics for Scheduling,* Auburn - Alabama: Department of Computer Science and Engineering - Auburn University, 1994.

[3] R. M. Karp, „Reducibility Among Combinatorial Problems," *Proceedings of a symposium on the Complexity of Computer Computations,* pp. 85-103, 1972.

[4] A. Forti, *DAG scheduling for grid computing systems,* Udine: University Of Udine - Department of Mathematics and Computer Science, 2006.

[5] C. McCreary, „Partitioning for Parallelization Using Graph Parsing," *Technical Report CSE-TR-91-17 - Auburn University,* 1991.

[6] C. L. McCreary, „Partitioning and Scheduling Using Graph Decomposition," in *ACM symposium on theory of computing*, 1993.

[7] T. White, Hadoop: The Definitive Guide - Fourth Edition, O'Reilly Media, Inc. , 2015.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrel, „Dryad: Distributed Data-Parallel Programs from Sequential," Microsoft, 2007.

[9] Henschen, Doug, „Microsoft Ditches Dryad, Focuses On Hadoop," InformationWeek, 17 November 2011. [Online]. Available: http://www.informationweek.com/software/information-management/microsoft-ditches-dryad-focuses-on-hadoop/d/d-id/1101390?. [Geopend 01 June 2015].

[10] The Apache Software Foundation, „Tez Project Incubation Status," Apache, 16 July 2014. [Online]. Available: https://incubator.apache.org/projects/tez.html. [Geopend 01 June 2015].

[11] Shenoy, Roopesh, „What is Apache Tez?," 25 April 2014. [Online]. Available: http://www.infoq.com/articles/apache-tez-saha-murthy. [Geopend 1 June 2015].

[12] Sally, „The Apache Software Foundation Announces Apache™ Spark™ as a Top-Level Project: The Apache Software Foundation Blog," 27 02 2014. [Online]. Available: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50. [Geopend 08 06 2015].

[13] „Research | Apache Spark," The Apache Software Foundation, [Online]. Available: https://spark.apache.org/research.html. [Geopend 08 06 2015].

[14] Databricks, „About Spark | Databricks," [Online]. Available: https://databricks.com/spark/about. [Geopend 15 06 2015].

[15] H. Karau, A. Konwinski, P. Wendell en M. Zaharia, Learning Spark, Sebastopol, California: O'Reilly Media, Inc., 2015.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker en I. Stoica, „Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," *University of California, Berkeley,* 2012.

[17] Microsoft Research, „Naiad Help - Table of Content," Microsoft Research, [Online]. Available: https://microsoftresearch.github.io/Naiad/. [Geopend 01 07 2015].

[18] Inside Microsoft Research, „Naiad: Incremental, Iterative Computation for Big Data - Inside Microsoft Research blog," Microsoft Corporation, 10 05 2012. [Online]. Available:

http://blogs.technet.com/b/inside_microsoft_research/archive/2012/05/09/naiad-incremental-iterative-computation-for-big-data.aspx. [Geopend 01 07 2015].

[19] D. Murray, „Naiad on YARN and Azure HDInsight - Big Data at SVC," Microsoft Research at Silicon Valley's Big Data blog, 22 04 2014. [Online]. Available: https://bigdataatsvc.wordpress.com/2014/04/22/naiad-on-yarn-and-azure-hdinsight/. [Geopend 01 07 2015].

[20] eBizMBA Inc., „Top 15 Popular Search Engines | October 2015," eBiz, [Online]. Available: http://www.ebizmba.com/articles/search-engines. [Geopend 10 2015].

[21] „Job Scheduling - Spark 1.4.1 Documentation," [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html. [Geopend 06 08 2015].

[22] A. Or (DataBricks), „Understanding your Spark application through visualization," 22 6 2015. [Online]. Available: https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html. [Geopend 17 8 2015].

[23] Red Hat, Inc., „Cygwin," [Online]. Available: https://www.cygwin.com/.

[24] G. community, „GEXF File Format," [Online]. Available: http://gexf.net/format/.

[25] Gephi.org, „Supported Graph Formats," [Online]. Available: https://gephi.github.io/users/supported-graph-formats/. [Geopend 10 2015].

[26] NetworkX developer team, „Overview - NetworkX," GitHub, [Online]. Available: https://networkx.github.io/. [Geopend 07 2015].

[27] B. Saha, „Data Processing API in Apache Tez," Hortonworks, 17 September 2013. [Online]. Available: http://hortonworks.com/blog/expressing-data-processing-in-apache-tez/. [Geopend 01 June 2015].

[28] D. A. Heger, „A Brief Introduction to Apache Tez," DHTechnologies & Data Nubes.

[29] Apache Community, „Tez Design," Apache, [Online]. Available: https://issues.apache.org/jira/secure/attachment/12588887/Tez%20Design%20v1.1.pdf. [Geopend 01 June 2015].

[30] B. Saha, „Apache Tez: A New Chapter in Hadoop Data Processing," 10 September 2013. [Online]. Available: http://hortonworks.com/blog/apache-tez-a-new-chapter-in-hadoop-data-processing/. [Geopend 03 June 2015].

[31] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy en C. Curino, „Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," in *ACM SIGMOD International Conference on Management of Data*, Melbourne, Victoria, Australia, 2015.

[32] YAHOO! Developer Network, „Hadoop Tutorial - YDN," [Online]. Available: https://developer.yahoo.com/hadoop/tutorial/module4.html#chaining. [Geopend 22 06 2015].

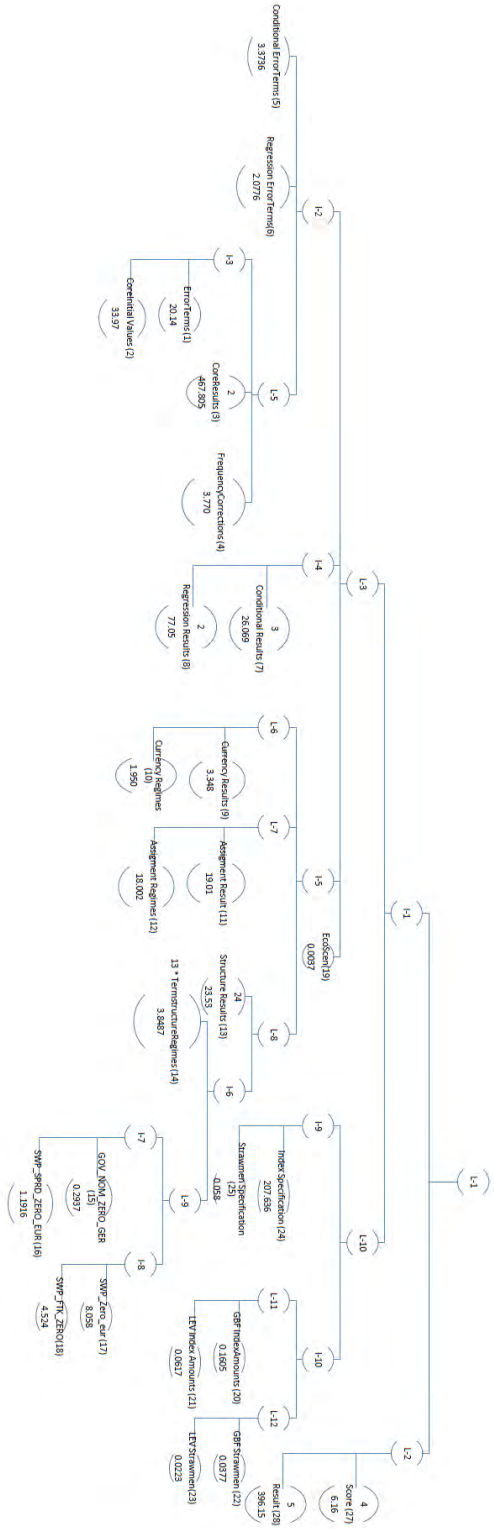
[33] „Spark: Lightning-fast cluster computing," [Online]. Available: https://spark.apache.org/. [Geopend 30 06 2015].

[34] „Apache Tez / Welcome to Apache Tez," The Apache Software Foundation, [Online]. Available: https://tez.apache.org/. [Geopend 30 06 2015].

[35] Aaron Davidson (Databricks), „A Deeper Understanding of Spark Internals," 17 07 2014. [Online]. Available: https://www.youtube.com/watch?v=dmL0N3qfSc8.

[36] S. Ghemawat, H. Gobioff en S.-T. Leung, „The Google File System," Google, 2003.

10 Appendix


Conditional ErrorTerms (5)
3.3736
Regression ErrorTerms(6)
2.0776
I-2
I-3
Corelnitial Values (2)
33.97
ErrorTerms (1)
20.14
CoreResults (3)
-67.805
2
I-5
FrequencyCorrections (4)
3.770
I-4
I-3
Regression Results (8)
77.05
2
Conditional Results (7)
26.089
3
I-6
I-7
Currency Regimes (10)
1.950
Currency Results (9)
3.348
Assignment Regimes (12)
18.002
Assignment Result (11)
19.01
I-5
I-8
EcoScen(19)
0.0097
13 * TermStructureRegimes (14)
3.8487
Structure Results (13)
23.53
24
I-6
I-1
I-9
I-7
SWP_3PRD_ZERO_EUR (16)
1.1916
GOV_NOM_ZERO_GER (15)
0.2997
I-9
Strawmen Specification (25)
-4.058
Index Specification (24)
207.686
I-10
I-8
SWP_FTK_ZERO(18)
4.524
SWP_Zero_eur (17)
8.058
I-11
GBP IndexAmounts (20)
0.1605
I-10
IDX Index Amounts (21)
0.0617
I-12
I-2
IDX Strawmen(23)
0.0223
GBP Strawmen (22)
0.0377
Result (28)
396.15
5
Score (27)
6.16
4

Ortec Finance bv
Boompjes 40
3011 XB Rotterdam
The Netherlands
Tel. +31 (0)10 700 50 00
Fax +31 (0)10 700 50 01

Ortec Finance bv
Barajasweg 10
1043 CP Amsterdam
The Netherlands
Tel. +31 (0)20 700 97 00
Fax +31 (0)20 700 97 01

Ortec Finance Ltd
23 Austin Friars
London EC2N 2QP
United Kingdom
Tel. +44 (0)20 3178 3913
Fax +44 (0)20 3178 6164

Ortec Finance AG
Poststrasse 4
8808 Pfäffikon SZ
Switzerland
Tel. +41 (0)55 410 38 38
Fax +41 (0)55 410 80 36

www.ortec-finance.com

ORTEC
FINANCE