Samenvatting Big Data Processing

Dylan Cluyse

2 januari 2023

Inhoudsopgave

Hoofdstuk 1

Big Data Processing

1.1 Inleiding

1.1.1 Breed beeld

Een gedistribueerd bestandssysteem is een manier om gegevens op te slaan en te beheren die is verspreid over meerdere computers in een cluster. In tegenstelling tot een traditioneel bestandssysteem, waarbij de gegevens op één centrale server worden opgeslagen, worden de gegevens in een gedistribueerd bestandssysteem opgeslagen op verschillende computers in het cluster. Dit maakt het mogelijk om grote hoeveelheden gegevens op te slaan en te verwerken zonder te worden beperkt door de beperkte hoeveelheid opslagruimte op één server. Het kan ook helpen om de prestaties te verbeteren door het gebruik van parallellisatie, waardoor meerdere computers tegelijkertijd kunnen werken aan het verwerken van de gegevens.

- Verschillende componenten op een netwerk die met elkaar communiceren.
- Een systeem met het doel om data beschikbaar te maken. Data dat later kan worden gelezen of geschreven.
- De mate van beschikbaarheid doet er niet toe.

1.1.2 De nood aan schaalbaarheid

In een gedistribueerd bestandssysteem is er een verschil tussen horizontale en verticale schaalbaarheid. Horizontale schaalbaarheid betekent dat het systeem kan worden uitgebreid door het toevoegen van meer computers aan het cluster, wat kan helpen om meer gegevens te verwerken en om de prestaties te verbeteren. Verticale schaalbaarheid betekent dat het systeem kan worden uitgebreid door het toevoegen van meer hardware aan een enkele computer, zoals extra geheugen of een snellere processor. Dit kan ook helpen om de prestaties te verbeteren, maar het is beperkter dan horizontale schaalbaarheid omdat het alleen mogelijk is binnen de beperkingen van een enkele computer.

Moore's Law bevestigt deze theorie. Deze wet zegt dat het aantal transistoren per achttien maanden verdubbelt. Met andere woorden is er een snelle nood aan nieuwe hardwarematerialen. Daarnaast is er ook nood aan nieuwe hardwarematerialen, zo is er nood aan fouttolerantie in een databankomgeving.

Het verkeer bij een databankserver gebeurt online, wat betekent dat de latency een rol speelt voor de gebruiker. De client moet een systeem kiezen die zo dicht mogelijk bij de client ligt. Online gameservers zijn een goede casus waarbij de systemen opgedeeld zijn per regio.

1.1.3 Kenmerken

Een gedistribueerd systeem kan herkend worden op basis van vier kenmerken:

- Geen gedeeld geheugen. Iedere verwerkingseenheid heeft een eigen geheugen.
- Onderling worden er berichten naar elkaar verstuurd.
- Componenenten zijn niet bewust van wat de andere componenten nu aan het doen zijn. Daarom sturen ze onderling berichten naar elkaar.
- Fouttolerant

1.2 Soorten systemen

We spreken van twee verschillende soorten systemen: parallele en gedistribueerde systemen.

- Bij een parallel systeem worden meerdere processen tegelijkertijd uitgevoerd. Deze verwerking gebeurt op verschillende verwerkingseenheden met een gedeeld geheugen. Het is makkelijker te ontwikkelen, maar met de kost van géén redundantie.
- Een gedistribueerd systeem bevat verschillende verwerkingseenheden met elk een eigen geheugen. De andere componenten zijn onbewust van wat de andere onderdelen aan het doen zijn. Er wordt onderling berichten met elkaar verstuurd. Dit noemt ook een shared-nothing architecture: Er wordt niets onderling gedeeld. De enige manier van communicatie is door middel van boodschappen.

1.2.1 CAP-theorem

Deze theorie wijst aan dat je, in een gedistribueerd databanksysteem, de afweging moet maken tussen drie factoren:

- Consistentie wijst aaan dat alle nodes dezelfde gegevens hebben.
- Beschikbaarheid wijst aan dat het systeem beschikbaar blijft voor alle gebruikers, zelfs al is er een crash of een paar nodes die niet beschikbaar zijn.
- Partitioneringstolerantie wijst aan dat het systeem blijft functioneren als er problemen zijn met de communicatie tussen de nodes.

Het is onmogelijk om alle drie deze eigenschappen tegelijkertijd te garanderen. Je moet een afweging maken of compromis sluiten.

Deze stelling komt terug uit databanken, maar hier is het anders. De consistentie van een relationele databank overstijgt de consistentie van een gedistribueerde databank.

1.2.2 Struikelblokken

Als we werken met een gedistribueerd systeem, dan zijn er vier horden waarmee we rekening moeten houden: * Split-brain scenario: de ene helft denkt het ene en de andere helft denkt het andere. Bijvoorbeeld: Het ene systeem denkt dat een bestand verwijderd is, terwijl het andere systeem denkt dat het nog bestaat. * Consistency en structuur raken snel verloren. * Testen wordt moeilijker. * De oorzaak van traagheid achterhalen wordt complexer: zowel hardware als software kunnen een rol spelen.

1.2.3 Fabels over gedistribueerde systemen

- ”Er is geen latency”. Latency is wel aanwezig. Enkel is de deze sterk minder naargelang de locatie van het systeem. het duurt een tijd vooraleer een bericht op een systeem aankomt.
- ”De bandbreedte is oneindig”. De bandbreedte op zowel de client als het distribueerd systeem is beperkt.

- ”Het netwerk is veilig”. Toegang tot het netwerk blijft iets waar je rekening mee moet houden.

- ”De netwerktopologie blijft hetzelfde”. Computers en hardware kan worden toegevoegd. Zo verandert alles binnen een netwerk op een dynamische manier.

- ”De transportkost van data is nul”. Data transporteren van begin- naar eindpunt vergt een inspanning qua energie en rekenkracht.

- ”Het netwerk is homogeen”. Alle onderdelen binnen een netwerk kunnen variëren van eigenschappen. Sommige delen van het netwerk kunnen snel zijn, sommige delen zijn traag.

1.3 Problemen met gedistribueerde netwerken

Bij een gedistribueerd systeem zijn er vier algemene problemen:

- Partial failures

- Niet-betrouwbare netwerken

- Niet-betrouwbare tijdsindicaties

- Onderlinge onzekerheid

1.3.1 Partial Failure

Er moet rekening worden gehouden met twee zaken:

- Allereerst is er de kans dat een systeem kan wegvallen. Sommige onderdelen van een netwerk kunnen werken, terwijl andere onderdelen down zijn of niet meer in gebruik. Hoe meer computers, hoe groter de kans dat één systeem (heel even) wegvalt.

- Als tweede punt kunnen andere systemen niet zien wanneer een systeem wegvalt. Het probleem wordt pas opgemerkt wanneer er geen antwoord is. De oorzaak kennen we niet. Dit kan liggen aan: overbelasting, defunct, te traag vergeleken met andere systemen, etc.

1.3.2 Niet-betrouwbare netwerken

Bij een asynchrone verbinding moeten we rekening houden met tijdsaannames. Boodschappen worden verstuurd zonder tijdsaannames. Het maakt de systemen niet uit hoe lang ze moeten uitvoeren of wanneer het bericht zal arriveren. De oorzaak van een netwerkfout is hier niet voor de hand liggend. Het is moeilijk om de oorzaak te achterhalen, want er zijn drie mogelijke problemen: een probleem tussen zender en ontvanger, de ontvanger kan niets ontvangen of de ontvanger kan niets versturen.

Het testen van een verbinding gebeurt met *pinging*. Om te vermijden dat een systeem iedere twee seconden een test uitvoert, werken we met *exponential back-off*. De tijd waarop een systeem wacht op een antwoord moet exponentieel vergroten. Begin met twee seconden wachten, daarna vijf seconden, daarna tien seconden, ... tot maximaal vijf minuten. Te snel berichten sturen moet vermeden worden, want zo wordt het netwerk belast en dan verergert de situatie. Eenmaal de capaciteit van de wachtrij wordt behaald, dan zal de vertraging (in seconden) exponentieel verhogen. IRL-voorbeeld: files.

1.3.3 Niet-betrouwbare tijdsindicaties

Er wordt het onderscheid tussen real-time en monotonische tijd gemaakt.

- Real-time tijd zijn klokken die gesynchroniseerd worden met het gebruik van een gecentraliseerde server.
- Monotonische klokken zijn klokken die op een vast moment starten en enkel vooruit gaan. Er is geen synchronisatie. Leap-seconden: een minuut is niet altijd 60 seconden. Soms kan dit 59 of 61 seconden zijn.

Causality is achterhalen wanneer een event werd uitgevoerd. Concensus is wanneer alle knopen (of nodes) met elkaar overeenkomen bij een beslissing.

1.3.4 Onderlinge onzekerheid

Nodes in een gedistribueerd systeem kan enkel veronderstellingen maken. De informatie dat een node bijhoudt verandert regelmatig. Voorbeelden hiervan zijn: klokken die desynchroniseren of nodes die niets terugsturen terwijl ze een update uitvoeren.

Split-brain

Split-brain is een concept rond inconsistente data-opslag. De ene helft van het systeem denkt dat iets juist is, terwijl de andere helft van het systeem denkt dat iets anders juist is. Bijvoorbeeld deel A denkt dat systeem 1 de baas is, terwijl deel B denkt dat systeem 2 de baas is.

Tweegeneralenprobleem

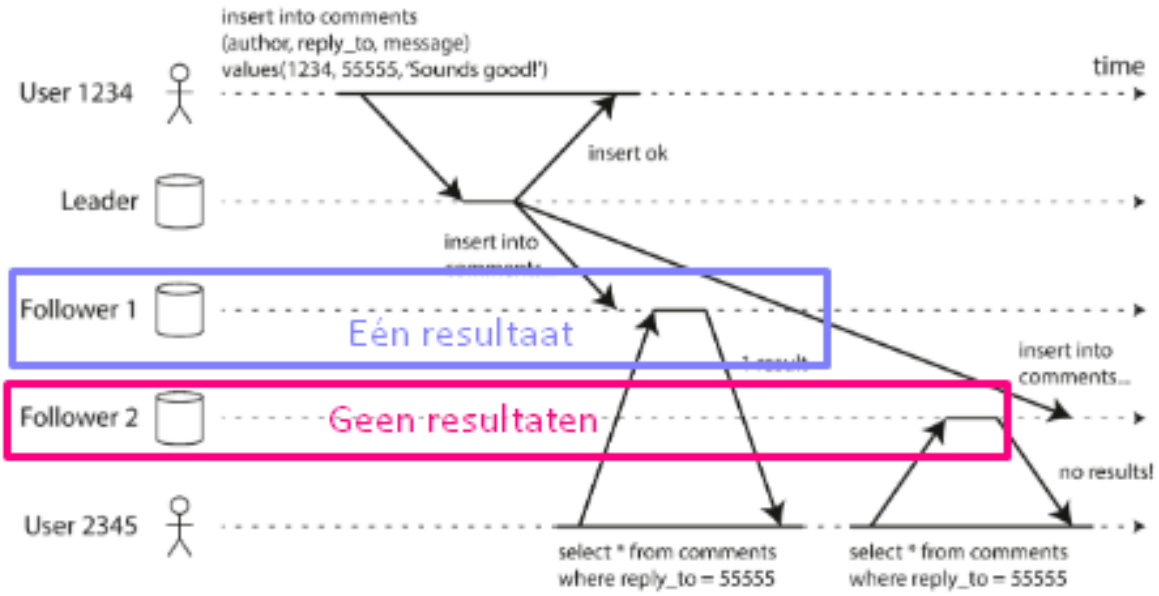
Het tweegeneralenprobleem bouwt verder op het split-brain concept. Dit weergeeft een scenario waarin beide partijen enkel winnen als ze samenwerken. Elk ander scenario leidt tot verlies. Beide partijen weten niet of de andere partij iets wilt ondernemen. Ze moeten het eerst vragen. De boodschap kan mogelijks niet tot de generaal komen. Dit probleem kunnen we toepassen binnen een online webshop. Er zijn vier gevallen:

- Als een online shop het pakket niet verstuurd en de klant betaalt geen geld, dan is er geen probleem.
- Als een online shop het pakket wél verstuurd en de klant betaalt geen geld, dan is de shop hier nadelig.
- Als een online shop het pakket niet verstuurd en de klant betaalt wél, dan is de klant hier nadelig.
- Als een online shop het pakket wél verstuurd en de klant betaalt wél, dan is iedereen *happy*.

1.4 Replication en partitioning

1.4.1 Partitionering

Bij partitionering zal je een groot bestand onderverdelen over meerdere knopen. Op deze manier hoef je niet alles op één plek op te slaan. Het nadeel hiervan is dat je geen toegang hebt tot het volledige bestand als één van de nodes niet bereikbaar is. Elk stuk data behoort tot precies één partitie. Bij Mongo en ElasticSearch noemt één partitie een *shard*.



Figuur 1.1: Het verschil tussen synchrone en asynchrone replication. Bij synchroon wacht je niet op bevestiging. Synchrone replicatie gaat direct door naar de target storage. Asynchrone replication wacht op bevestiging van de volgers op de source.

1.4.2 Replication

Replicatie is het maken en onderhouden van verschillende kopieën op meerdere knooppunten. Hiermee wordt redundantie aangeboden. Als de data op node A niet beschikbaar is, dan worden gebruikers doorverwezen naar node B. Dit concept komt vaak voor bij geografisch gespreide netwerken. Bijvoorbeeld een knooppunt in Oceanië, Azië, Europa, etc. Er zijn hier drie verschillende leader-volger technieken:

1. Bij single-leader doen alle clients wat de leider zegt. Alle writes komen vanuit één leider binnen één partitie. De boodschap van de clients kan gedateerd zijn. Als gevolg kunnen acties uitgevoerd worden die niet meer van toepassing zijn.
2. Bij leaderless replication versturen de clients elke write naar verschillende nodes. De clients lezen parallel. Zij zorgen ervoor dat de data OK blijft. Elk verstuurt boodschappen door naar de nabije clients. Achterhaalde data kan worden tegengegaan door te werken met timestamps.
3. Multi-leader replication bouwt verder op single-leader replication. Meerdere nodes worden in verschillende datacenters geplaatst.

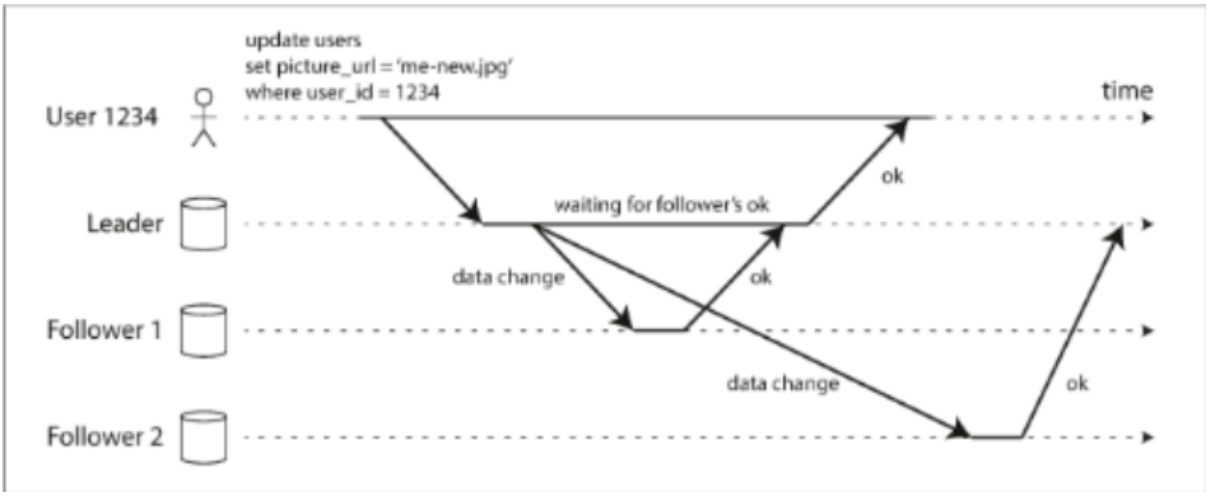
Replication-lag

Leaders zijn niet statisch. Deze kunnen veranderen door clients te promoveren tot leader. Als een leader verandert, dan is er de kans op replication-lag. Dit is wanneer een volger wordt gepromoveerd tot leader.

Replication kan synchroon of asynchroon verlopen. Synchroon is wanneer je wacht op de antwoorden van de volgers. Je bent hiermee zeker dat de data niet zal verloren gaan, maar ten gevolge zal het systeem trager zijn. Asynchroon is wanneer er niet wordt gewacht op de volgers. Alle overplaatsingen zullen vlot verlopen als er geen wissels gebeuren bij de leaders. Hoe groter de replication lag, hoe groter de kans op dataverlies.

Replicatiefouten

Fouten bij synchrone replicatie zijn minder voorkomend, maar de techniek kan gedwarsboomd worden. Zo heb je nog steeds een probleem wanneer een write-operation niet kan



Figuur 1.2: Voorbeeld van een fout bij synchrone replicatie.

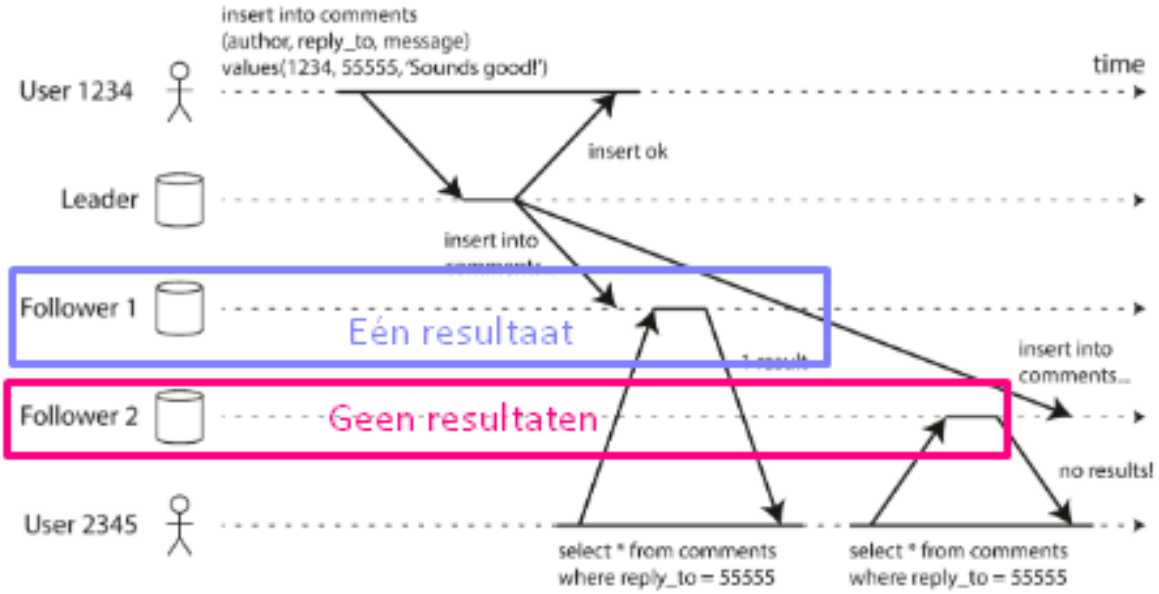
worden afgewerkt als één van de volgers niet online is. Je wacht tot de bevestiging van de volgers. De twee vaak voorkomende fouten bij asynchrone replicatie zijn monotonic read en read-after-write.

- Read-after-write (RAW) duidt, zoals de naam het aangeeft, op een fout bij het lezen. De client heeft in dit geval een comment geplaatst en vervolgens wordt er een bevestiging gegeven aan de client. Daarna wilt dezelfde gebruiker dezelfde post inlezen, maar die is nog niet opgeslaan door een andere volger. Je wilt dezelfde output krijgen als je hetzelfde in de databank schrijft. Bij RAW is het belangrijk om met een timestamp te werken. Zorg dat je bij een schrijfoperatie alles tot aan een punt moet laten voldoen aan de timestamp. Zo ja, haal de gegevens op en geef ze aan de client. Zo niet, wacht of kijk naar een andere volger.
- Een gelijkaardig, maar nog steeds verschillend probleem, is monotonic read. De gebruiker leest een post of comment, maar na een refresh is deze comment opeens niet beschikbaar of niet-bestaand. De volger loopt hier achter op de andere volgers. De klok bij de ene volger loopt voor op de andere. Het verschil hier is dat de gebruiker de tekst niet heeft geschreven, wat wel het geval is bij RAW. Dit lossen we op door de gebruiker altijd van dezelfde replica te laten lezen. Hieronder leest de gebruiker eerst van de volger mét het resultaat. Daarna probeert de gebruiker dit opnieuw, maar bij een volger die achterloopt.

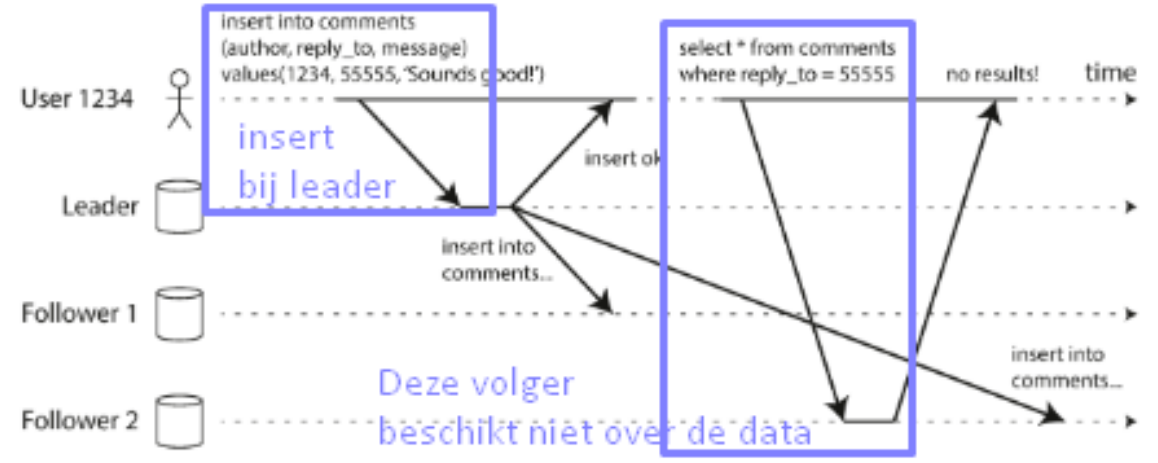
1.4.3 Replicatie en partitionering combineren

We kunnen niet zomaar data in stukken snijden. We moetne hotspots vermijden. Een hotspot is een plaats waar de verdeling geen goede verhouding heeft voor iedere node. Hiervoor hebben we twee oplossingen: key-value partitionering en hash partitionering.

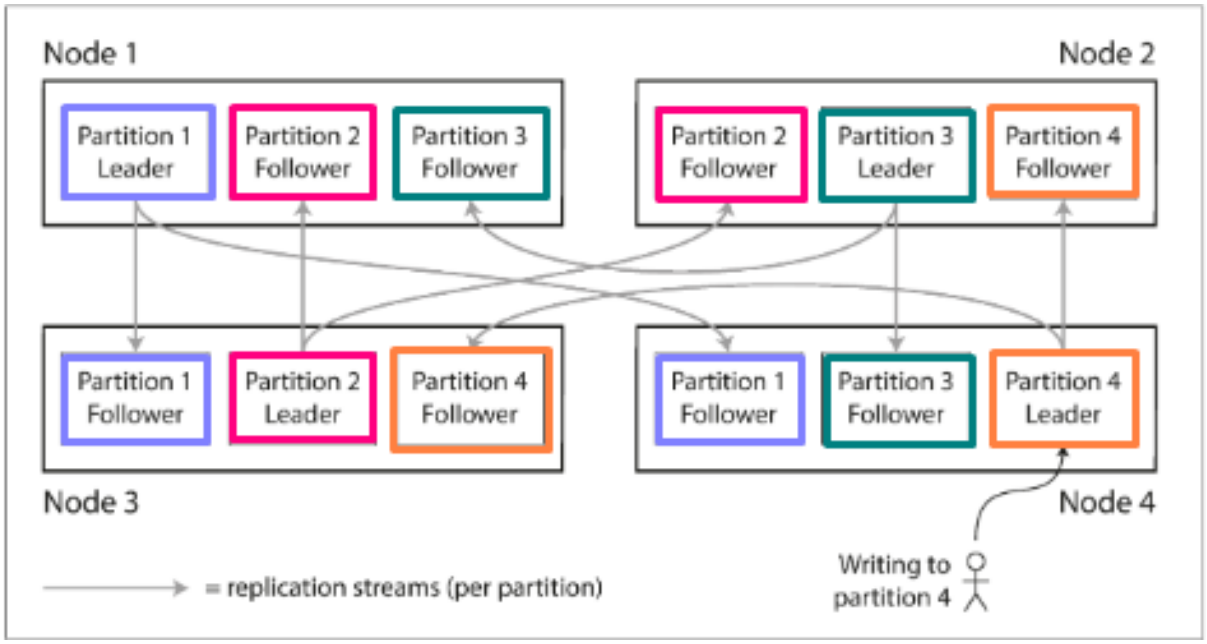
- Key-value partitionering is wanneer je een zo eerlijk en even mogelijke verdeling maakt over alle nodes. Alle sleutels binnen een node behoren tot een range. Je sorteert alle sleutels. Bijvoorbeeld alles van A t.e.m. E. Afhankelijk van de context wordt vaak voorkomende data binnen dezelfde partitie opgeslaan. Dit zorgt voor meer verkeer op partitie A-D vergeleken met X-Z. De ene partitie zal een hotspot worden, maar de andere zal geen verkeer krijgen.
- Hash partioning lost dit probleem merendeels op, maar het is niet de meest efficiënte implementatie. Hier verlies je sortering. De compromise hier is dat je wél een even verdeling zal krijgen. De hash houdt rekening met beschikbare plaats. De kans dat een partitie niet gebruikt zal worden is kleiner.



Figuur 1.3: Monotonic read



Figuur 1.4: Read-after-write probleem.



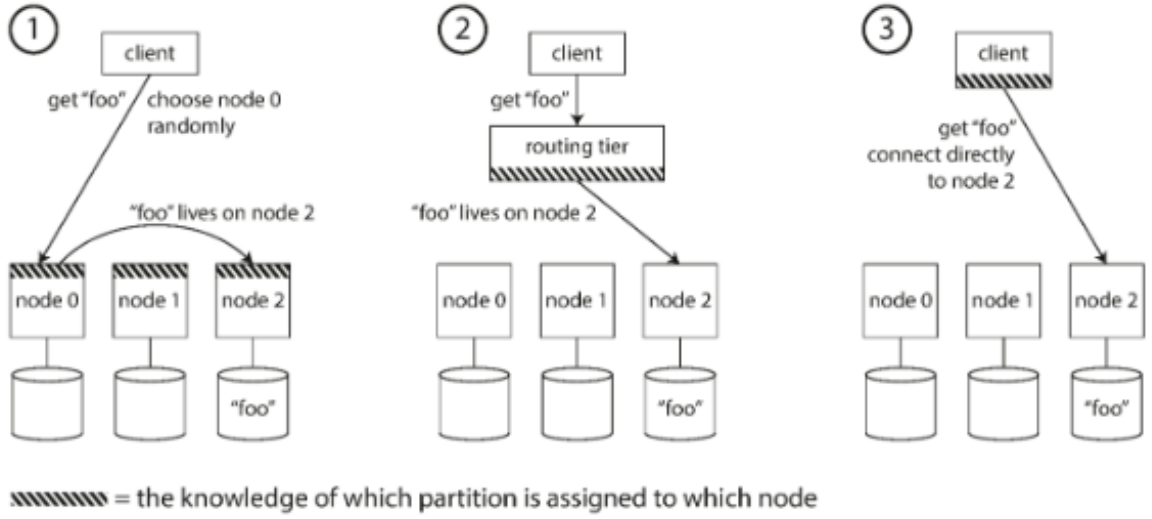
Figuur 1.5: Vier partities: elke leader heeft twee volgers. In de volgende foto zijn er vier nodes. Elke node heeft drie onderdelen. Over de vier nodes zijn er vier verschillende partities verdeeld. De replica's of volgers kan je achterhalen aan de hand van de stream. De leider van partitie 1 in node 1. De replica's zijn in Node 3 en in Node 4. De leider van partitie 2 is in node 3. De replica's zijn in Node 1 en in Node 2.

1.5 Request routing

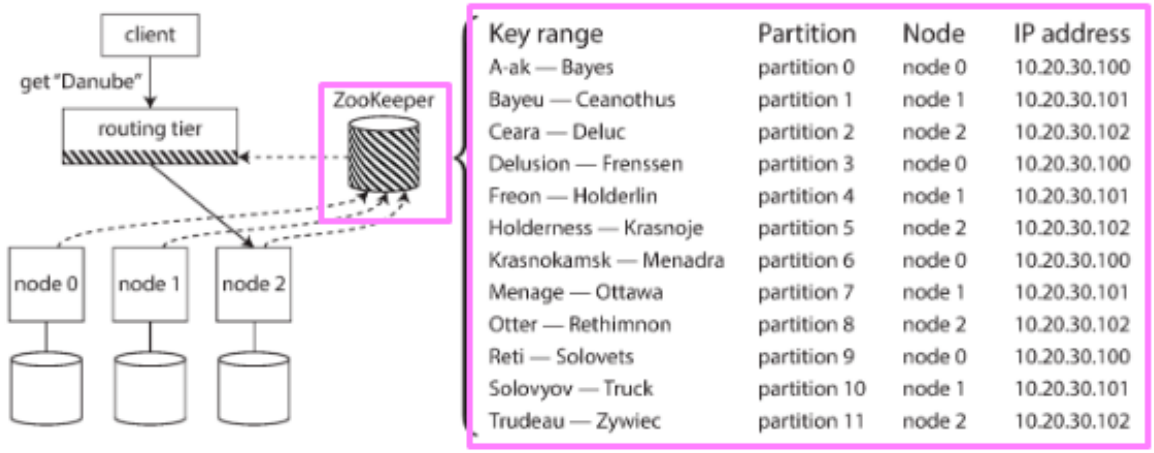
De plaats van data achterhalen kan op drie manieren:

1. Iedere node bevat metadata. De client kan een willekeurige knoop contacteren. De knoop weet waar het te zoeken woord is. Hieronder geeft knoop 0 mee dat het te zoeken woord op knoop 2 is.
2. Er is een laag tussen de client en de knopen. De routing-tier bevat metadata.
3. De client heeft directe toegang tot de metadata. Dit wordt het minste gebruikt.

De locatie van metadata onderhouden gebeurt met de coordination service. Dit zorgt voor het onderhoud en de mapping van de metadata. Het is de routing tier (RT) tussen de client en de knopen. De RT is direct verbonden met zowel de knopen, alsook met de Zookeeper. Als er iets verandert in de data van een node, dan meoten de nodes dit laten weten aan de Zookeeper.



Figuur 1.6: De drie mogelijkheden om de plaats van data te achterhalen.



Figuur 1.7: Zookeeper.

Hoofdstuk 2

Hadoop Filesystem

2.1 Inleiding

Het maakt gebruik van parallellisatie om de gegevens te verdelen over meerdere computers in een cluster, waardoor het verwerkingsproces sneller wordt. Dit kan worden gebruikt voor het analyseren van gegevens, zoals het ontdekken van trends en patroonherkenning. Hadoop is één van de eerste frameworks voor Big Data Processing. Het is een relatief oud project met *clunky* technieken.

Het is ontworpen met de gedachten om clusters te kunnen draaien op normale hardware. Als je cluster bestaat uit honderden computers, dan is de kans groot dat er één zal breken. Het basisidee van Hadoop is om dit soort fouten af te handelen en zodat het systeem blijft werken zoals voordien.

2.1.1 Hadoop stack

Een pure hadoop-stack bestaat uit vier onderdelen:

- Hadoop common: de gedeelde bibliotheken die door de andere modules worden gebruikt. Je ziet dit gedeelte niet. Het is de meeste onderste laag.
- HDFS is een filesystem. Dit zorgt voor de distributed file storage. Je merkt de delay amper.
- MapReduce is het processing-gedeelte. MapReduce laat je toe om parallel grote datasets te verwerken.
- YARN voorkomt dat één element in je cluster alle resources opeet. Dit element zorgt voor request-afhandeling van resourcevragen.

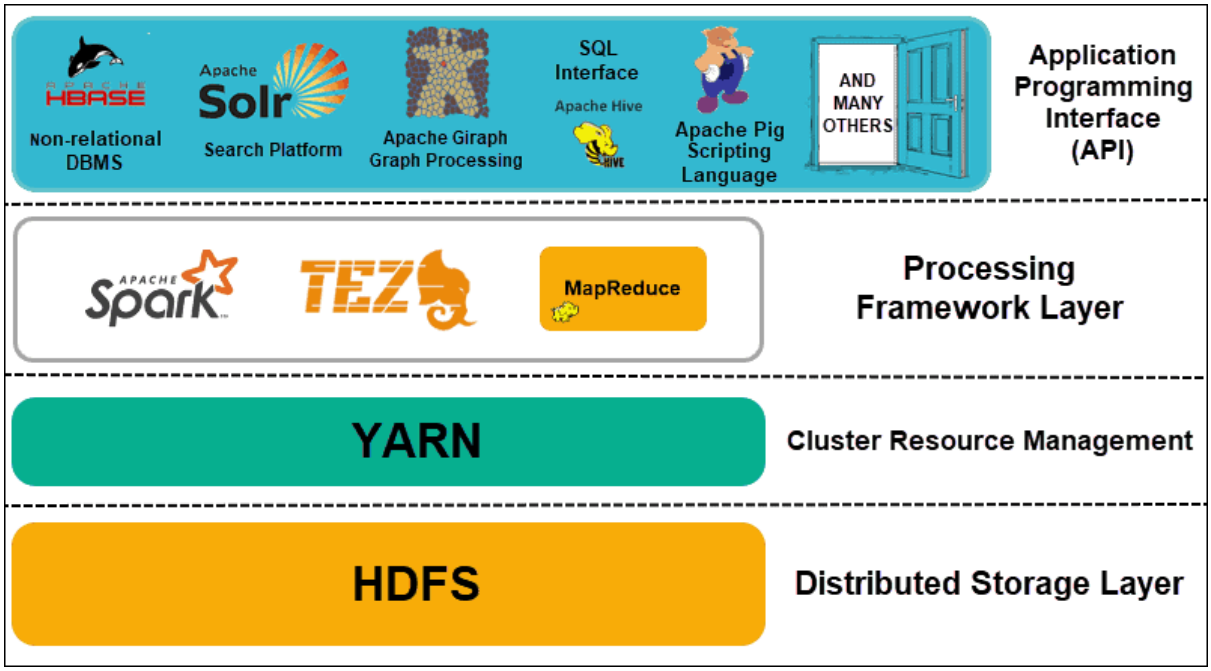
2.2 Hadoop filesystem

Algemeen

Een filesystem laat toe om data op te slaan en weer op te halen. Er zijn drie soorten data: files, mappen en metadata. Metadata is de info over de mappen of bestanden, zoals file length en permissies. Een filesystem zorgt ervoor dat de data toegankelijk is. De consistentie in filesystems wordt bewaard door middel van een logsysteem.

Designprincipes

De data wordt opgeslaan in een cluster van gewone machines. De focus van HDFS ligt op riante hoeveelheden data op te slaan. Er wordt gewerkt met veel groepen van machines. Als er één machine zou kapot gaan, dan neemt een andere machine over. Er is weinig vertraging. Write-once-read-many-times (WORM) betekent dat gegevens één keer worden



Figuur 2.1: Het Hadoop ecosysteem. Onderaan staat Hadoop FileSystem. Daarboven is er YARN, wat instaat voor het resourcemanagement. Boven het filesystem en de resourcemanager staan er de processing services, waaronder Spark en MapReduce. API's komen in deze samenvatting niet aan bod.

geschreven en vervolgens vele keren worden gelezen, maar niet wordt gewijzigd of verwijderd. HDFS ondersteunt deze techniek om zo een hoge fouttolerantie te kunnen bieden en doorvoer te kunnen bieden. Er zijn twee algemene redenen waarom er geen bestanden worden gewijzigd of verwijderd.

- HDFS is bedoeld om grote hoeveelheden data op te slaan. Verwijderen en wijzigen vergt kostbare energie. Daarnaast is HDFS fouttolerant, want het systeem moet blijven werken zelfs al vallen er nodes uit. Als een bestand wordt verwijderd, dan beïnvloedt dit de integriteit van de data.
- HDFS is bedoeld om een historiek uit te bouwen. De data wordt ingezet in data-analyse, waar nauwkeurigheid een rol speelt, dus er mogen geen wijzigingen aan de data worden aangebracht.

Het principe is 'write-once, read-many-times'. Een bestand wordt eenmalig gemaakt. De doorvoer van het systeem is hier belangrijker. De hoeveelheid verwerkte data per tijdseenheid. HDFS op een klassiek filesysteem. Eens de machine bezig is kan je een grote doorvoer hebben. Er is sprake van **append-only fashion**. De gegevens kunnen enkel achteraan worden toegevoegd en niet tussenin of helemaal vooraan. Deze werkwijze helpt om de prestaties en betrouwbaarheid van HDFS te verbeteren, omdat het aantal wijzigingen aan gegevens beperkt.

Anti-patterns

Er zijn enkele situaties waarbij HDFS *overkill* of ongeschikt is als oplossing:

- Wanneer snelheid een grote rol speelt. De latency is hier niet minimaal.
- Wanneer het merendeel van de data uit kleine bestanden bestaat. De nadruk bij HDFS ligt op grote bestanden. De NameNode houdt de directorystructuur bij, waardoor veel kleine bestanden zal leiden tot veel verschillende takken.

- Wanneer je meerdere schrijvers op eenzelfde moment wilt hebben. Als één client schrijft en de andere wilt lezen, dan kan er een conflict ontstaan bij de integriteit en het delen van een bestand.
- Wanneer je gegevens

* Append-only fashion: Als je inhoud vooraan of in het midden wilt toevoegen.

2.3 Hadoop-onderdelen

HDFS heeft twee componenten:

- De NameNode.
- De DataNode.

Kort samengevat is de namenode de node dat het bestandssysteem beheert en bijhoudt waar alle bestanden zijn opgeslagen. De datanode slaat de bestanden op en zorgt ervoor dat gegevens beschikbaar zijn voor lezen en schrijven.

2.3.1 NameNode

Een NameNode (NN) is de centrale coördinator. weet uit welke blokken de data bestaat en houdt bij op welke datanodes de blokken staan. Locaties worden niet persistent bijgehouden. Onderling weten ze dit door middel van *heartbeats*. Als je een file wilt hernoemen, dan lukt dat ook. Bij een cluster heb je altijd één NN.

De NN heeft de volgende rollen:

- Het beheerde de filesystem namespace.
- Het koppelt de datablokken aan de DataNodes.
- Het handelt de filesystemverzoeken, zoals het openen/sluiten en hernoemen van bestanden of mappen.
- Het gidst de client naar de best passende DataNode.

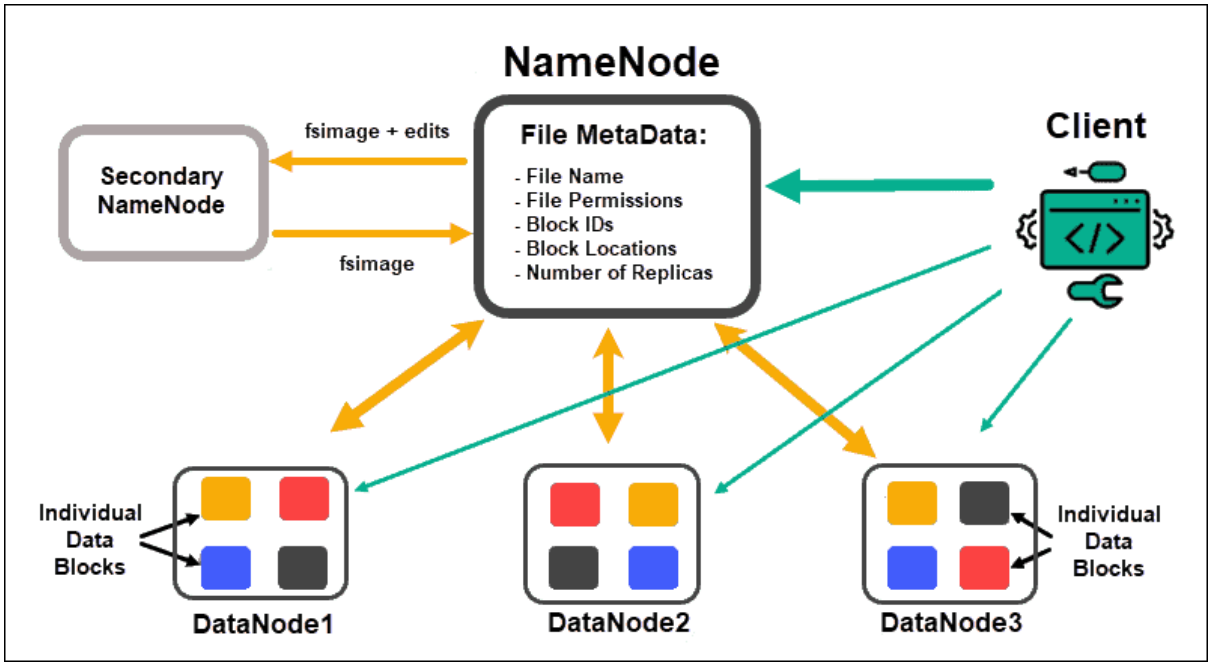
Opslag voor metadata

De NameNode slaat metadata over het filesystem op in twee bestanden: de namespace image en de edit log. De *namespace image* bevat informatie over de structuur van het filesystem (metadata), waaronder de directoryboom en de bestanden en mappen die het bevat. De metadata wordt in het RAM bijgehouden. Daarmee is het snel, maar vluchtig. Terwijl het systeem loopt wordt het opgebouwd, maar de informatie is niet persistent.

De *edit log* bevat een opname van alle wijzigingen in het filesystem, zoals het maken of verwijderen van bestanden en mappen. Dit is een append-log. Telkens als er iets verandert, dan wordt er informatie toegevoegd. Als de NN opnieuw zou opstarten, dan wordt de image en edit log gelezen. Daarna worden de veranderingen van de edit log toegepast op de namespace image om zo een nieuwe namespace image te maken. Daarna wordt een nieuwe edit log gestart.

Werking NameNode

De NameNode communiceert met de DataNodes om bij te houden op welke DataNodes welke datablokken worden opgeslagen. Daarnaast zorgt de NN ervoor dat de data beschikbaar en consistent is over een cluster. Wanneer een client data in HDFS wil raadplegen, stuurt hij een verzoek naar de NameNode, die vervolgens bepaalt op welke DataNode de gevraagde datablok staat en de client naar die DataNode stuurt.



Figuur 2.2: De verschillende componenten van HDFS, waaronder DataNodes en NameNodes.

2.3.2 DataNode

De DataNodes (DN) stellen de servers voor die de werkelijke datablokken opslaan. Ze zijn verantwoordelijk voor het afhandelen van lees- en schrijfverzoeken voor het maken/verwijderen en hermaken van datablokken. Deze acties gebeuren volgens de instructies van de NN. Alle instructies worden gegeven met een *block report*, wat een periodieke rapportering is van de NN.

Een DN heeft de volgende functies:

- Direct luisteren naar instructies van de NN.
- Lees- en schrijfverzoeken afhandelen.
- Het maken, verwijderen of repliceren van datablokken.

Een DN is het werkpaard van een HDFS-cluster, want ze beheren en slaan de data op die in een cluster kan worden teruggevonden. In het werkveld zijn er riante hoeveelheden DN's aanwezig. Alles wat de client leest is afkomstig van een DN. Elk blok heeft een replicatiefactor. De replicatiefactor wijst op hoeveel systemen het bestand beschikbaar moet staan. Een HDFS heeft veel DataNodes.

Werking DataNode

Wanneer een client data in HDFS wil raadplegen, stuurt hij een verzoek naar de NameNode, die vervolgens bepaalt op welke DataNode de gevraagde datablok staat en de client naar die DataNode stuurt. De DataNode haalt vervolgens de gevraagde datablok op en stuurt deze terug naar de client.

2.4 Single-point-of-failure

Als één NN uitvalt, of als de NS Image uitvalt, dan zullen de datablokken niet meer toegankelijk zijn. Alle data zal wél blijven bestaan op de DN. Alle blokken hebben random verwijzingen. Als client ben je er niet van bewust waarvan de data komt en waartoe die gaat.

2.4.1 Bescherming NameNode

Er zijn twee manieren om het NN te beschermen:

- De meest voor de hand liggende manier is om regelmatig een backup te nemen. Hadoop kan worden geconfigureerd om metadata op meerdere filesystems op een synchrone en atomaire manier te schrijven.
- De tweede optie is een secondary NN. Zolang je een systeem niet herstart wordt je edit log langer en groter. Dit is niet ideaal, want als het neemt zowel plaats in alsook zal het opstarten van een NS image langer duren. In latere versie van Hadoop hebben ze een secondary NN toegevoegd. Een secondary NN is een proces dat op een andere computer loopt en dat de veranderingen van de edit log verwerkt in de huidige namespace.

2.5 HDFS Blokken

Datablokken hebben niets te maken met "blokkenöp een traditionele harde schijf. De datablokken op een Hadoop FS verwijzen naar de stukken data waarin een bestand wordt verdeeld voor opslag. De standaardgrootte van een bestand is 128MB, dus relatief groot. Door een bestand in blokgroottet te verdelen, kan het in HDFS worden opgeslagen, zelfs als het groter is dan elke enkele schijf in een cluster. Het laatste blok van een bestand kan kleiner zijn dan de standaardblokgrootte als het bestand niet een exact veelvoud is van de blokgrootte.

Preventie

Om dataverlies te voorkomen, wordt voor elk bestand in HDFS een replicatiefactor ingesteld. Deze waard steld het aantal replicants voor. Een replicant is een 'dubbel' van een bestand. HDFS zal *proberen* om het aantal replicanten te laten overeenkomen met de opgestelde replicatiefactor. Verschillende bestanden kunnen afwijkende replicatiefactoren hebben. De factor hangt af van de gewenste redundantie en bescherming.

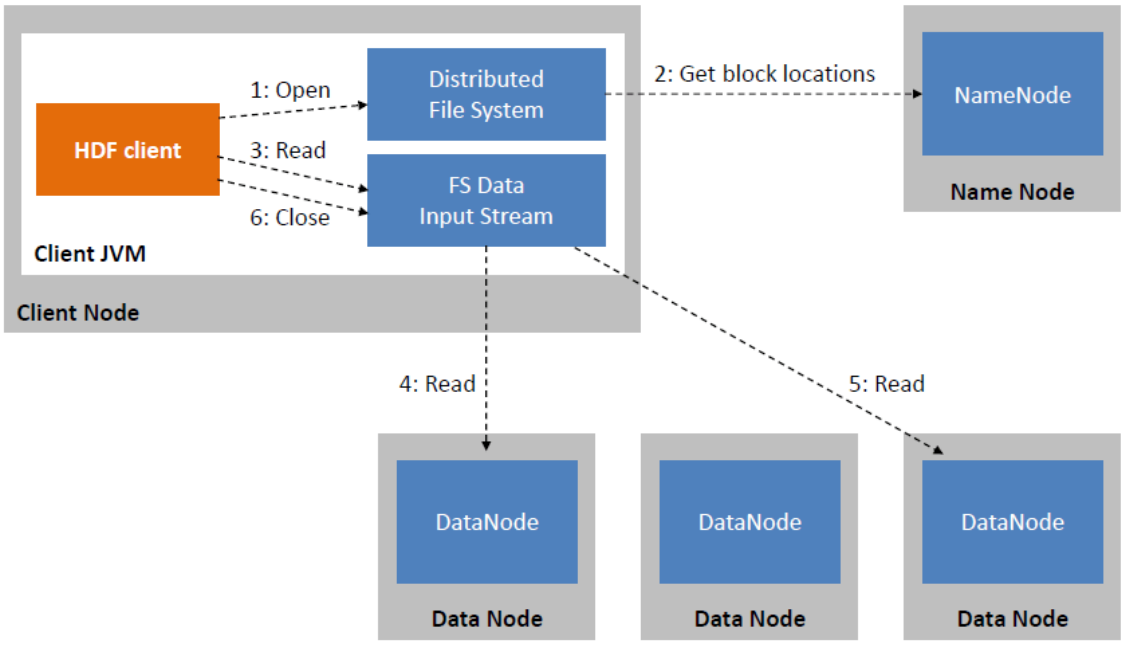
2.6 Anatomy

2.6.1 Anatomy of a File Read

Het proces om een bestand uit HDFS te lezen, omvat de volgende stappen:

1. De Client maakt een oproep naar een instantie van het HDFS.
2. De instantie maakt gebruik van Remote Procedure Calls (RPC) naar de NN om de lcoaties van de eerste paar blokken in het bestand te achterhalen. Voor elke blok geeft de NN de adressen van de DN's die een kopie hebben van dat blok. Die adressen zijn gesorteerd op afstand van de DN tot de client. De HDFS geeft een *DataInputStream*-object terug naar de client. Dit object bevat een *DFSInputStream*-object, die de Input/Output tussen de DN en de NN beheert.
3. De client roept een leesoperatie op de stream. Het *DFSInputStream*-object verbindt automatisch met de DN voor de volgede blok. Dit gebeurt transparant met de client, wie denkt dat het een continue stream aan het lezen is.
4. Eenmaal de client klaar is met lezen, dan stuurt de client een een *close* op de *FSDataInputStream*.

De client contacteert de DN direct om data op te halen. De NN gidst de client richting de meest optimale verbinding met een datablok. De NN geeft géén data, maar het verleent enkel richting door de locaties van de block requests terug te geven. Deze locaties staan in het RAM-geheugen, daarom gebeurt dit proces heel snel.



Figuur 2.3: File read schema opgehaald van Javatpoint (2022)

Rollen

- De *DistributedFileSystem* is een Java-klasse die een interface voorziet om met het HDFS te kunnen interrageren. Met dit interface kan een client toegang tot een bestand krijgen of een bestand manipuleren. De DFS communiceert met de NN om acties uit te voeren. De DFS gebruikt RPC's om verzoeken naar de NN te versturen en te ontvangen. De meest gebruikte functies zijn:
  - 'open' om een bestaand bestand te openen: een inputstream wordt teruggegeven.
  - 'create' om een nieuw bestand aan te maken: een outputstream wordt teruggegeven om data in het bestand te schrijven
  - 'delete' om een bestand te verwijderen
  - 'mkdirs' om een nieuwe directory te maken
  - 'listStatus' om een lijst van bestanden en directory statussen, van een gespecifeerd pad, terug te krijgen
- De DFSInputStream is een interne klasse dat gebruikt wordt om data uit een bestand te lezen. Het verbindt met het passende DN, daarnaast streamt het de data terug naar de client.
- De DataInputStream is een klasse dat een interface voorziet om datatypes uit te lezen. Het wordt gebruikt om data van eender welke inputstream te lezen, inclusief een DFSInputStream.
- Een FSDataInputStream is een klasse die de DFSInputStream inkapselt en extra functionaliteit aanbiedt. Het wordt gebruikt om eender welke data te lezen van een bestand in HDFS.

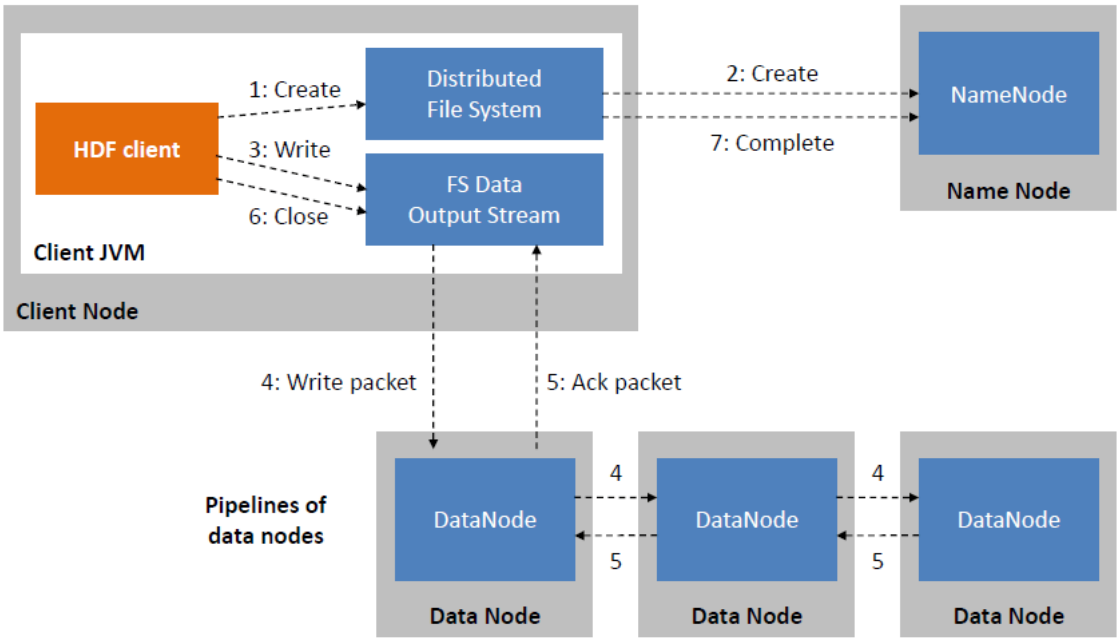
2.6.2 Anatomy of a File Write

Het proces om een nieuw bestand in HDFS te maken, omvat de volgende stappen:

1. De client roept een *create* of *delete* aan op de *DistributedFileSystem*.
2. Het HDFS maakt een RPC-oproep om een nieuw bestand in de namespace van het filesystem te maken, zonder blokken die eraan gekoppeld zijn. De NN voert controles uit, waaronder kijken of het bestand bestaat of kijken naar de machtigingen van de client. Als alle controles slagen, dan maakt de NN een record van het nieuwe bestand. Zo niet, dan faalt de *file creation* en dan krijgt de client een *IO-exception* te zien.
3. Het HDFS geeft een *FSDataOutputStream* terug aan de client. Dit object bestaat uit een *DFSOutputStream*, die de communicatie met de DN's en de NN beheerst.
4. Als de client gegevens schrijft, dan wordt de *DFSOutputStream* in pakketten gesplitst. Deze pakketten worden naar een *data queue* geschreven. Deze wachtrij wordt doorgestuurd naar een *DataStreamer*. Dit object vraagt om beurt aan de NN om nieuwe blokken te alloceren, dit met een lijst van geschikte DN's om de replicaties op te slaan.
5. De lijst van DN's vormt een pipeline.
6. Het *DataStreamer*-object streamt de pakketten naar de eerste DN in de pipeline, die elk pakket opslaat en doorstuurt naar de tweede DN in de pipeline. De tweede DN herhaalt dit, de derde DN herhaalt dit, enzovoort.
7. De *DFSOutputStream* onderhoudt een *acknowledgement queue*. Dit terwijl pakketten worden bevestigd door de DN's. De pakketten worden van de *ack queue* verwijderd als ze door alle DN's in de pipeline zijn bevestigd.
8. Eenmaal de client klaar is met het schrijven van gegevens, roept de client *close* aan op de stream. Dit spoelt alle overblijvende pakketten naar de DN-pipeline en wacht op bevestigingen voordat hij contact opneemt met de NN. De NN weet al uit welke blokken het bestand bestaat, dus deze node moet enkel wachten tot de blokken gerepliceerd zijn voordat de node succesvol terugkomt.

Rollen

- De *DistributedFileSystem* is een Java-klasse die een interface voorziet om met het HDFS te kunnen interrageren. Met dit interface kan een client toegang tot een bestand krijgen of een bestand manipuleren. De DFS communiceert met de NN om acties uit te voeren. De DFS gebruikt RPC's om verzoeken naar de NN te versturen en te ontvangen. De meest gebruikte functies zijn:
  - 'open' om een bestaand bestand te openen: een inputstream wordt teruggegeven.
  - 'create' om een nieuw bestand aan te maken: een outputstream wordt teruggegeven om data in het bestand te schrijven
  - 'delete' om een bestand te verwijderen
  - 'mkdirs' om een nieuwe directory te maken
  - 'listStatus' om een lijst van bestanden en directory statussen, van een gespecifeerd pad, terug te krijgen
- De *DataStreamer* is verantwoordelijk voor het streamen van data vanuit de client naar de verschillende DN's in een cluster. Het wordt gebruikt wanneer een gebruiker wilt schrijven naar een bestand in een HDFS. Dit object vraagt de blokallocaties aan de NN, zo wordt er een lijst bijgehouden van geschikte DN's.
- Een *DFSOutputStream* is een interne klassie die gebruikt wordt om data te schrijven naar een bestand. Het splitst de data in pakketten, streamt de pakketten naar de DN's en verzekert dat de data op een betrouwbare manier wordt afgehandeld.
- De *FSOutputStream* is een publieke lasse dat een interface voorziet voor clients om data uit te schrijven naar een bestand in een HDFS. Het inkapselt de *DFSOutputStream* en synchroniseert de data stream.



Figuur 2.4: File Write schema opgehaald van Javatpoint (2022)

Foutafhandeling

Bij het falen van een DN, terwijl er data naartoe wordt geschreven, dan worden er enkele acties afgehandeld. De client weet wat er achter de schermen gebeurt bij zo een afhandeling. Sommige blokken kunnen *under-replicated* zijn, maar de NN zal dit opmerken. De blokken zullen asynchroon worden gerepliceerd in de cluster. Dit proces herhaalt zich tot de replicatiefactor is behaald.

2.7 Command-Line

De CLI voor HDFS biedt een reeks commando's aan om te communiceren met een HDFS-cluster. Deze taal heeft gelijkenissen met de Bash-taal. Om een lijst te krijgen met beschikbare commando's en opties, kan je het commando "hadoop fs -help"gebruiken.

Kopiëren en ophalen

```
// Lokaal --> HDFS
hadoop fs --copyFromLocal input/docs.txt hdfs://localhost/user/dylan/docs.txt

// Alles in een Hadoop map bekijken
hadoop fs -ls .

// HDFS naar Lokaal
hadoop fs --copyToLocal hdfs://localhost/user/dylan/docs.txt output/docs.txt
```

Nieuwe elementen aanmaken

```
// Bestand aanmaken
hadoop fs -touch docs.txt

// Directory aanmaken
```

```
5 hadoop fs -mkdir input
```

2.8 MapReduce

Hadoop MapReduce is een programma dat wordt gebruikt voor het verwerken van grote hoeveelheden gegevens in een distributiefile-systeem. Het maakt gebruik van parallellisatie om de gegevens te verdelen over meerdere computers in een cluster, waardoor het verwerkingsproces sneller wordt. Dit kan worden gebruikt voor het analyseren van gegevens, zoals het ontdekken van trends en patroonherkenning. Een MapReduce bestaat uit drie fasen:

1. Mapping
2. Shuffle
3. Reduce

Voordelen van MapReduce

MapReduce vereist geen kennis van parallelisatie, data distributie en fouttolerantie binnen het programmeren. Dit gebeurt automatisch over een riant aantal machines gespreid. Alle details rond partitionering van de input-data, scheduling en foutafhandeling wordt al afgehandeld. Bij MapReduce kan de programmeur focussen op de logica van de applicatie en het is zo makkelijker om parallele programma's te schrijven die schaalbaar, efficiënt en fouttolerant zijn. Bij MapReduce is *re-execution* het belangrijkste mechanisme. Als een taak faalt, dan zal het framework automatisch de taak opnieuw proberen op een nieuwe machine.

2.8.1 Data Flow

Het framework zal iedere split aan een *map task* toekennen. Deze task

1. Bij een MapReduce job wordt de inputdata over verschillende *chunks* of *inputsplits* heen verdeeld. De *map job* zal het deel lijn per lijn bekijken. Iedere mapper zal een deel van het bestand te zien krijgen.
2. Het framework zal iedere split aan een *map task* toekennen. De *map task* verwerkt de data en genereert een set van key-value paren.
3. Het framework verzamelt en groepeert de waarden met dezelfde key. Deze groeperingen worden doorgegeven aan de *reduce task*.
4. De *reduce task* verwerkt de keys met hun waarden. De output is een verzameling van key-value paren. Deze output wordt naar een outputbestand in HDFS geschreven.

2.8.2 Mapping

De Mapper in Hadoop MapReduce is een programma dat wordt gebruikt om gegevens te verdelen over meerdere computers in een cluster, zodat ze parallel kunnen worden verwerkt. De Mapper leest de gegevens in en verdeelt ze in kleinere stukjes, die vervolgens naar de verschillende computers in het cluster worden gestuurd om te worden verwerkt. Dit maakt het mogelijk om grote hoeveelheden gegevens snel te verwerken en te analyseren. De Mapper is het eerste onderdeel van het MapReduce-proces en zorgt ervoor dat de gegevens op een gestructureerde manier worden verwerkt.

Voorbeeld luchttemperatuur

De mapfunctie zal ieder inputrecord verwerken. Ieder record bestaat uit lijnen tekst. Bij het verwerken worden de jaar- en luchttemperatuurvelden uit het bestand gehaald. Het filter recors met ontbrekende of verdachte temperaturen. De mapfunctie zal key-value paren voor ieder record gaan genereren. De key is het jaar en de value is de luchttemperatuur. De mapfunctie geeft de key-value paren aan de shuffle & sort fase.

2.8.3 Shuffling

De Shuffle-fase in Hadoop MapReduce is een belangrijk onderdeel van het MapReduce-proces waarbij de gegevens worden verzameld en gerangschikt op basis van de sleutels die aan de gegevens zijn toegekend. De Mapper verdeelt de gegevens in kleinere stukjes en stuurt deze naar de verschillende computers in het cluster, waar ze worden verwerkt. De Reducer verzamelt vervolgens de verwerkte gegevens van alle computers in het cluster en sorteert ze op basis van de sleutels, zodat de gegevens kunnen worden verwerkt en geanalyseerd. De Shuffle-fase is dus een cruciale stap in het MapReduce-proces omdat het ervoor zorgt dat de gegevens op een gestructureerde manier worden verwerkt en geanalyseerd.

2.8.4 Reduce

In een MapReduce job zal de input van de reducefunctie uit key-value paren bestaan. Die werden aangemaakt door de mapfunctie en doorgegeven aan de shuffle & sort fase. De reducefuntie verwerkt iedere key met de toebehorende waarden. Uiteindelijk is de laatste stap in de MapReduce om een output te maken. De reducer wordt bijvoorbeeld gebruikt om totale aantallen te berekenen of gemiddelden te berekenen voor een bepaalde groep gegevens. Het is een belangrijk onderdeel van het MapReduce-proces omdat het ervoor zorgt dat de gegevens op een gestructureerde manier worden verwerkt en geanalyseerd.

Voorbeeld luchttemperatuur

De reducefunctie zal, van de shuffle & sort fase, een verzameling van key-value paren ontvangen. De key is hier het jaar, en de value is de luchttemperatuur. De reducefunctie itereert door de temperatuurwaarden van ieder jaar en het neemt de maximale waarde. Dit hoort tot de finale uitvoer. De finale uitvoer bestaat uit een set key-value paren, waarbij de key het jaar is en de waarde de grootste luchttemperatuur die werd opgenomen in dat jaar.

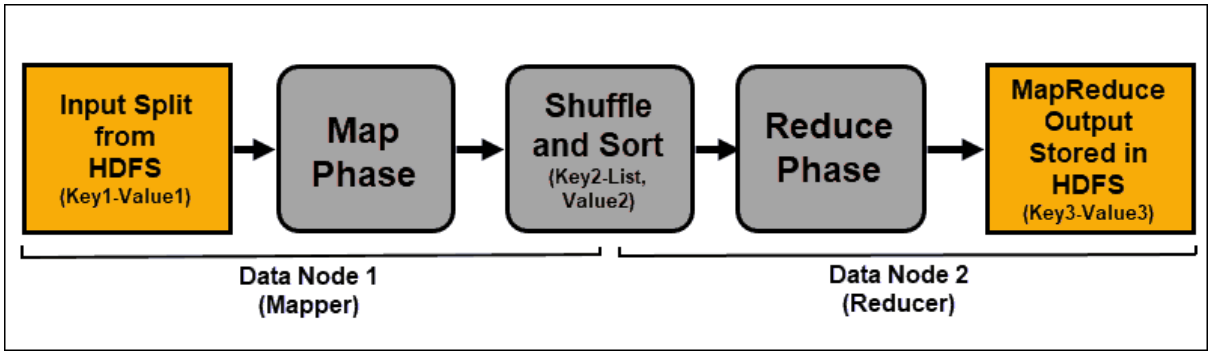
2.9 Java MapReduce

Java werd gekozen omdat dit de meest prevalente taal is voor MapReduce. Een MapReduce applicatie bestaat uit drie onderdelen:

1. Het schrijven van de mapfunctie. De mapfunctie zal iedere input record verwerken en maakt hierop een verzameling van key-value paren. De mapper-klasse wordt ge*extend* en enkel de *map()*-methode wordt overerfd. De *map()*-methode vraagt een sleutel als inputwaarde op. Dit is regelmatig de lijn waarop tekst voorkomt. Op basis van deze invoer maakt het een verzameling van key-value paren.

2. Het schrijven van de reducefunctie. De reducefunctie verwerkt iedere key met de toebehorende waarden. Je moet de Reducer-klasse *extenden* en de *reduce()*-methode overerven. De input bij deze methode is een sleutel en een set waarden dat de verzameling van output key-value paren gaat maken.

3. Boilerplate-code schrijven. Dit is de code die specifieert hoe de MapReduce job moet draaien. Iedere fase, inclusief extra-reduce fasen, moeten aan bod komen. Alles van job configuratie, input en outputpaden instellen en *job running* moet in de boilerplate code terug te vinden zijn.



Figuur 2.5: Een vereenvoudigde versie van de job-flow bij MapReduce.

4. In sommige gevallen kan je ook een combinerfunctie schrijven. Deze gelijkt sterk op een reducefunctie en wordt op de output van de *map task* toegepast. De combinerfunctie wordt gebruikt om het aantal data, dat tussen de mapper en de reducer bevindt, aan te passen. Zo wordt het proces efficiënter behandeld. Voor een combiner moet je de Reducer-klasse *extenden* en de *reduce()*-methode overerven, net zoals bij de reducefunctie.

WordCount-oefening

```
1  public class WordCount {
2
3    public static class TokenizerMapper
4    extends Mapper<Object, Text, Text, IntWritable>{
5
6      private final static IntWritable one = new IntWritable(1);
7      private Text word = new Text();
8
9      public void map(Object key, Text value, Context context
10     ) throws IOException, InterruptedException {
11       StringTokenizer itr = new StringTokenizer(value.toString());
12       while (itr.hasMoreTokens()) {
13         word.set(itr.nextToken());
14         context.write(word, one);
15       }
16     }
17   }
18
19   public static class IntSumReducer
20   extends Reducer<Text,IntWritable,Text,IntWritable> {
21     private IntWritable result = new IntWritable();
22
23     public void reduce(Text key, Iterable<IntWritable> values,
24     Context context
25     ) throws IOException, InterruptedException {
26       int sum = 0;
27       for (IntWritable val : values) {
28         sum += val.get();
29       }
30       result.set(sum);
31       context.write(key, result);
32     }
33   }
34
35   public static void main(String[] args) throws Exception {
36     Configuration conf = new Configuration();
37     Job job = Job.getInstance(conf, "word count");
38     job.setJarByClass(WordCount.class);
39     job.setMapperClass(TokenizerMapper.class);
40     job.setCombinerClass(IntSumReducer.class);
41     job.setReducerClass(IntSumReducer.class);
42     job.setOutputKeyClass(Text.class);
```

```
43        job.setOutputValueClass(IntWritable.class);
44        FileInputFormat.addInputPath(job, new Path(args[0]));
45        FileOutputFormat.setOutputPath(job, new Path(args[1]));
46        System.exit(job.waitForCompletion(true) ? 0 : 1);
47    }
48 }
```

2.9.1 Mapper-klasse

De mapperklasse is een generieke klasse dat vier typeparameters heeft. De mapfunctie zal bij ieder input record een verzameling key-value paren aanmaken. Hier zal de key het regelnummer zijn. De value zal de toebehorende tekst op de regel zijn.

- De input key type is een Object.
- De input value type is een Text.
- De output key type is een Text. Dit is de volledige tekst dat op een lijn terug te vinden is. representeert het woord waarop er geteld word.
- De output value type is een IntWritable. Dit is het aantal voorkomens van het toebehorende woord.

2.9.2 Reduce-klasse

De reduceklasse verwerkt iedere key en de toebehorende waardeN. De outputkey zal hier het woord zijn, en de outputvalue zal hier het aantal voorkomens van een woord in een bestand zijn.

2.9.3 Main-methode

De main-methode zal een Job-object aanmaken. Dit object geeft aan hoe de job moet worden gedraaid. De *setJarByClass*-methode specifieert het JAR-bestand dat de klassen bevat in de job. Hadoop gebruikt deze informatie om het meest passende JAR-bestand terug te vinden. Vervolgens worden de input- en outputpaden toegelicht.

Inputpaden

Het inputpad wordt bepaald door *addInputPath* van het object *FileInputFormat*. De input kan hier één bestand, maar ook meerdere bestanden in een directory zijn. Globbing is ook mogelijk.

Outputpaden

Het outputpad wordt bepaald door *addOutputPath* van het object *FileOutputFormat*. De outputfile of directory mag, voor het uitvoeren van de job, niet bestaan! Deze voorzorg wordt genomen om *data loss* te voorkomen.

Afwerking

De *waitForCompletion*-methode wacht tot de job is afgerond. Bij het al dan niet succesvol afwerken zal de job een boolean teruggeven.

2.9.4 Development Environment

POM-file

Alle projecten worden van het formaat 'Maven' zijn. Dit wordt vlot in Eclipse aangemaakt. De *dependencies* moeten de nodige MapReduce en Hadoop plugins bevatten. Alle dependencies kan de ontwikkelaar terugvinden in het *pom.xml* bestand. Dit bestand is het configuratiebestand van Maven.

JAR genereren

Om te kijken of alle dependencies en properties correct zijn ingesteld, moet de ontwikkelaar een JAR maken. Dit door te rechtermuisklikken op de POM-file, vervolgens moet een nieuwe build worden aangemaakt. In de uitvoer krijgt de ontwikkelaar een *build failure* of een *build success*.

JAR in een Hadoop NameNode plaatsen.

Het JAR-bestand wordt in de vagrantmap geplaatst. Er wordt verondersteld dat de Hadoop-container aan het draaien is. De JAR-file wordt naar een NN-cluster doorgestuurd met de volgende commando's.

```
1  // op de vagrantmachine
2  docker cp target/hadoop-example-1.0-SNAPSHOT.jar namenode:/
3  docker exec -it namenode bash
4  ls -l hadoop-example-1.0-SNAPSHOT.jar
5
6  // in de hadoop-shell
7  hadoop fs -mkdir input/ncdc
8  hadoop fs -copyFromLocal 190? input/ncdc
9  hadoop fs -ls input/ncdc
10
11 // het uitvoeren van de JAR in de hadoop-shell
12 hadoop jar hadoop-example-1.0-SNAPSHOT.jar be.hogent.dit.tin.MaxTemperature input/ncdc output/
       ncdc
13
14 // resultaat bekijken
15 hadoop fs -cat output/ncdc/part-r-00000
```

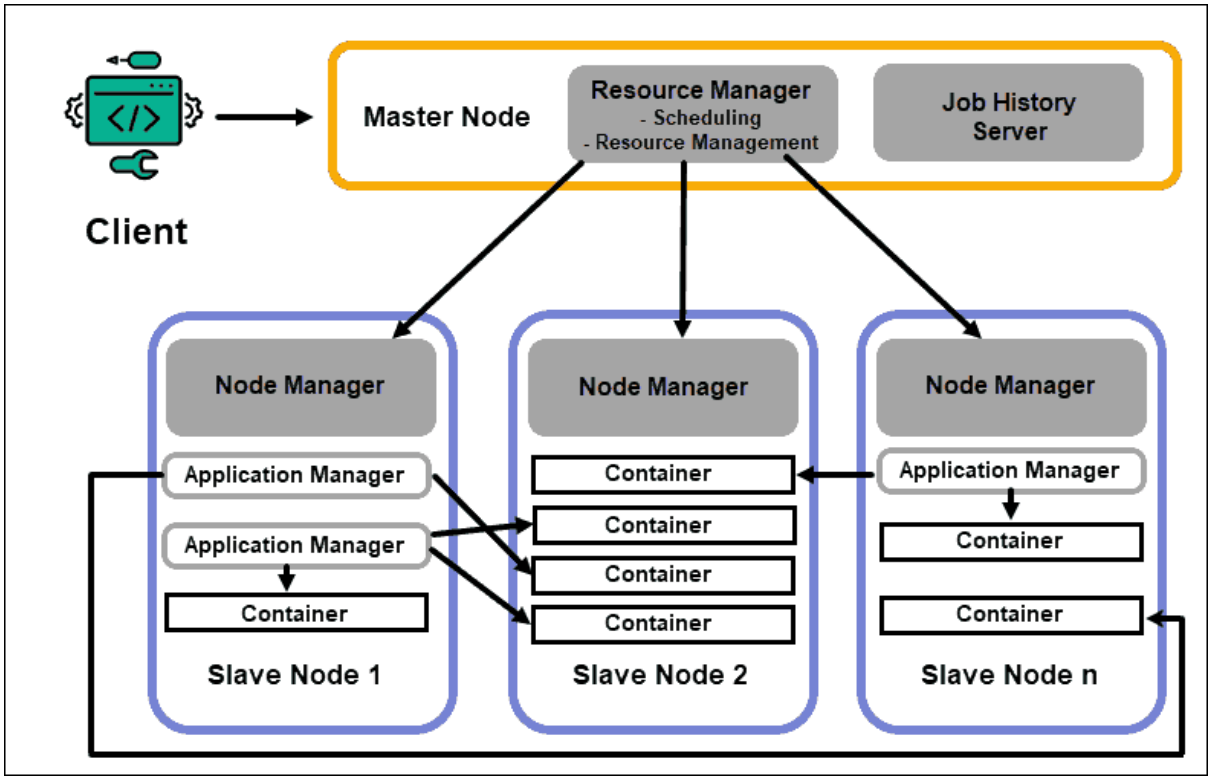
2.10 YARN

2.10.1 Algemeen

YARN is een resourcemanager voor Hadoop-clusters. Deze daemon onderhoudt het toewijzen van CPU, geheugen en middelen aan verschillende toepassingen die op de cluster draaien. YARN biedt API's om mideelen op een cluster te verzoeken te gebruiken, maar deze API's worden niet door de gebruiker gebriukt. In plaats daarvan schrijven gebruikers naar *higher-level* API's die door *distributed computing frameworks* worden aangeboden. Voorbeelden hiervan zijn: MapReduce, Spark en Flink. Deze high-level API's verbergen de details van resourcemanagers. Zo kunnen programmeurs focussen op de logica.

Extra ondersteuning

YARN biedt ook ondersteuning voor het inplannen en uitvoeren van toepassingen op de cluster, alsook het monitoren van de status van die toepassingen. Gebruikers kunnen zo een groot aantal toepassingen uitvoeren op deen Hadoop-cluster, waaronder *batch processing*, *stream processing*, *machine learning* en *interactive SQL*.



Figuur 2.6: YARN architectuur met daemon services.

2.10.2 Core Services

YARN biedt *core services* aan van twee types *long-term daemon processes*:

- Er is slechts één resourcemanager (RM) per Hadoop-cluster. Deze is verantwoordelijk voor het beheren van middelengebruik per cluster. De RM ontvangt *resource requests* van *application masters* en bepaalt vervolgens welke nodes op de cluster beschikbare middelen hebben om deze verzoeken te vervullen.

- Er is één actieve nodemanager (NM) op elke node van een Hadoop-cluster. De NM is verantwoordelijk voor het monitoren en opstarten van de containers op een node. Een container is een *ligth-weight execution environment* die een toepassingsspecifiek proces uitvoert met een beperkt aantal middelen, zoals geheugen en CPU. Containers in YARN zijn niet gerelateerd aan de containers dat Docker heeft.

De RM en NM werken samen om middelen toe te wijzen en applicaties uit te voeren op een Hadoop-cluster. Gebruikers kunnen hun toepassingen aan YARN voorstellen via high-level API's.

2.10.3 YARN Application run

1. Een client neemt contact op met de RM en vraagt om een *application-master-process* op te starten.

2. De RM zoekt een NM die de *application-master* in een contianer kan starten. Alles daarna is afhankelijk van de toepassing. Bij een berekening moet er een waarde naar de client worden teruggestuurd. Bij MapReduce moet er een gedistribueerde bewerking worden uitgevoerd, want alle woorden zijn verspreid over verschillende containers. YARN zelf biedt géén manier om onderdelen van een toepassing met elkaar te laten communiceren.

2.10.4 Resource requests

In een YARN resourcemanager wordt een resource request verstuurd om een specifieke taak mogelijk te maken. Deze request bevat een gepast aantal bewerkingsmiddelen, zoals CPU en geheugen. Deze requests bevatten *locality constraints*. Dit geeft aan waartoe de middelen moeten beschikbaar worden gesteld. Als een container een HDFS blok moet verwerken, dan moet de resource request toelichten dat een container op één van de nodes, waar er een replica is, moet worden geplaatst. Zo wordt de data doorvoer geminimaliseerd én de snelheid van de applicatie wordt verbeterd.

2.10.5 Scheduling

De scheduler bepaalt of de middelen beschikbaar zijn en, indien mogelijk, worden de middelen toegekend aan de toepassing. Als een cluster druk is, met andere woorden zijn de middelen niet direct beschikbaar, dan zal de scheduler de *resource requests* tijdelijk *on-hold* plaatsen. Nadien worden de middelen aan de applicatie toegekend.

Verschillende schedulers

Er zijn verschillende schedulers in YARN:

- FIFO, of *first-in, first-out* zal de middelen aan applicaties toekennen in de volgorde waarin de scheduler de request kreeg. *First come, first served*.
- *Capacity scheduler* laat administrators toe om specifieke middelen te reserveren voor gepaste types toepassingen.
- *Fair scheduler* zal de resources op een eerlijke manier proberen te verdelen over alle applicaties heen. Hier zijn er verschillende factoren zoals de noden aan middelen en voorafgaand gebruik voor iedere applicatie.

Hoofdstuk 3

Spark

Vooraf

Voor de komst waren er een aantal programmeermodellen gericht op filesystemclusters. De meeste waren gespecialiseerd, waaronder MapReduce, Storm, Impala en Pregel. Spark werd ontworpen om sneller en flexibeler te zijn dan de vooraf genoemde modellen. Sindsdien is het een populaire keze geworden voor een heleboel dataverwerkingstaken, waaronder *batch processing*, streamverwerking, *machine learning* en gegevensanalyse. Het biedt een aantal voordelen t.o.v. andere clustercomputersystemen, zoals:

- Flexibele en expressieve API
- Ondersteuning voor een breed aantal gegevensbronnen en formaten
- Mogelijk om te draaien in een groot aantal deployment-omgevingen, zoals on-premises clusters, cloudgebaseerde clusters en zelfs op één enkele machine.

Algemene voordelen

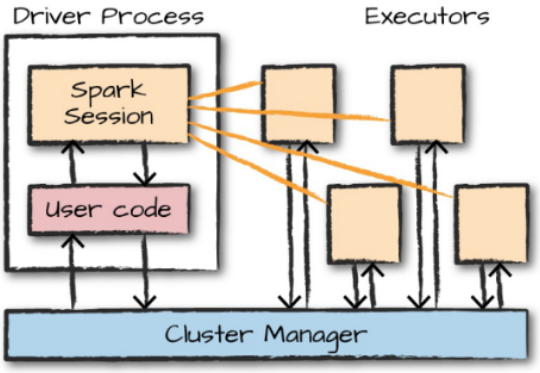
Spark is generiek en algemeen. Een analoge uitleg is om Spark een smartphone te noemen, terwijl andere modellen eerder een GPS, mobiele telefoon of digitale camera zijn. Spark is een *jack-of-all-trades*. daarom brengt het drie voordelen met zich mee:

- Toepassingen zijn makkelijker te ontwikkelen, want ze maken gebruik van een gecentraliseerde API.
- Het is efficiënter om verwerkingstaken te combineren. Voor Spark moeten pipelines vaak de gegevens wegschrijven naar opslag om ze door te geven aan een andere engine. Spark kan diverse functies uitvoeren op dezelfde gegevens, vaak in het geheugen.
- Spark maakt nieuwe toepassingen mogelijk, bijvoorbeeld interactieve queries op het streamen van ML.

3.1 Spark Application Architecture

Er zijn drie belangrijke rollen:

- Bij een Spark-applicatie is de *driverprocess* hetgeen wat de applicatiecode doet draaien. Dit proces onderhoudt de informatie van de Spark-applicatie, beantwoordt de input van de gebruiker en verdeeld het werk over de *executors*.
- Een executorprocess gaat het werk uitvoeren dat hen werd toegekend. De staat van de actie (geslaagd/niet geslaagd) wordt gerapporteerd aan het driverproces.
- De *clustermanager* houdt logs bij van de beschikbare resources en welke resources er aan de executors, indien mogelijk, kunnen worden gegeven. Spark ondersteunt verschillende clustermanagers, waaronder YARN, Mesos en Kubernetes.



Figuur 3.1: Opbouw en structuur van een Spark-applicatie

Partitionering

Parallelle verwerking met Spark kan enkel gebeuren wanneer het systeem in verschillende clusters is opgebouwd. Een partitie is in dit geval een verzameling van rijen die op een fysieke machine in een cluster is opgeslaan. De partities van een DataFrame tonen hoe de data fysiek is verdeeld over de cluster heen. Het aantal partities is gebonden op het niveau van parallelisme in een Spark applicatie. Als je één partitie hebt, dan ga je een parallelisme van één hebben, zelfs al heb je meerdere executors. Omgekeerd ook, want als je meerdere partities hebt met één executor, dan zal je nog steeds een parallelisme van één hebben. De afweging maken tussen aantal partities en aantal executors speelt hier een rol op het vlak van snelheid.

3.2 Spark in Java

3.2.1 Dataframe

Een dataframe is een gedistribueerde, in-memory tabel met benoemde kolommen en een schema. Het schema duidt elke specifiek datatype aan. Een dataframe is *immutable*, dus eenmaal aangemaakt kan het object niet meer worden gewijzigd. Dataframes kunnen worden aangemaakt uit verschillende bronnen, zoals Hadoop InputFormats of CSV-bestanden. Verdere transformaties, zoals aggregaties en filteren, kunnen zeker toegepast worden op eender welk type formaat. In Java-code verwijzen we naar het object 'Dataset<>'. Tussen de diamandtekens plaatsen we het object 'Row'.

3.2.2 Transformaties

Er zijn verschillende types transformaties die op een dataframe kunnen worden uitgevoerd. Deze transformaties verlopen *lazy evaluated*. Het onderscheid tussen *wide* en *narrow* transformaties wordt hier gemaakt:

- Een narrow transformatie is wanneer het resultaat voor een partitie kan worden berekend met enkel de data van die partitie. Iedere inputpartitie contribueert tot minstens één outputpartite. Deze transformatie kan ook gebeuren aan de hand van een pipeline, zo hoeft er geen data naar de disk te worden geschreven. Narrow transformaties zijn snel en efficiënt.
- Een wide transformatie heeft nood aan meerdere outputpartities. Sommige inputpartities zullen contribueren tot elk meerdere outputpartities. Hier is er nood aan een *shuffle*, waarbij er een proces wordt toegevoegd. In dit proces worden de partities, over een cluster heen, onderling met elkaar uitgewisseld. Een shuffle vereist dat er resultaten naar de *disk* wordt geschreven. Aangezien dit extra schrijfoperaties zijn, is het aangeraden om het aantal wide-transformaties zo minimaal mogelijk te houden.

Lazy Evaluation

Spark maakt gebruik van Lazy evaluation, wat wilt zeggen dat een transformatie niet zal resulteren in een nieuwe dataframe. De return-waarde is wél een nieuw object dat het resultaat van de bewerking bijhoudt. De berekening is nog niet uitgevoerd. Bij het triggeren van een actie kijkt Spark naar de graph van alle transformaties die werden opgeroepen. Net zoals bij SQL wordt er een *query execution plan* aangemaakt. Dit proces noemt *query optimization*. Spark kan mogelijks meerdere filters en filters gaan samensteken, of het kan andere soorten optimizatie gaan toepassen om zo min mogelijk data te moeten verwerken.

3.2.3 Acties

Transformaties in Spark bouwen een logische transformatieplan op. Die geeft aan hoe de inputdata moet worden getransformeerd om de gewenste output te krijgen. Het transformatieplan wordt niet uitgevoerd tot de actie wordt getriggerd. Een actie geeft Spark de instructie om een resultaat of verzameling van transformaties te berekenen. Sommige voorbeelden van acties zijn:

- Sommige rijen van een dataframe tonen.
- Het aantal rijen van een dataframe optellen.
- De dataframe opslaan als een native object, zoals een Java lijst.
- Een dataframe schrijven naar een externe *data sink*.

Eenmaal de actie is getriggerd, dan zal Spark het transformatieplan uitvoeren en de resultaat van de actie berekenen.

3.2.4 Physical plan

Spark's Physical Plan wijst aan hoe de Spark-applicatie zal uitgevoerd worden op een cluster van machines. Dit omvat details zoals:

- Welke transformaties moeten er worden uitgevoerd?
- Hoe moet de data gepartitioneerd of geshuffled worden?
- Hoe moeten de resultaten van de transformaties worden opgeslaan of geretourneerd?

Het belang van een physical plan

Het *physical plan* is belangrijk, want zo achterhalen ontwikkelaars hoe zij hun middelengebruik kunnen optimaliseren en het aantal data, dat tussen machines heen moet worden geschreven, kan geminimaliseerd worden. Verdere oplossingen zijn:

- Debugging en troubleshooten.
- Performance optimization
- Resource management
- Capaciteitsplanning

3.3 Spark Libraries

Dit is puur informatief. Spark bevat meerdere libraries om data-analyse of data-berekeningen op uit te voeren.

- SQL en dataframe
- Spark Streaming
- MLLib
- GraphX

3.3.1 Integratie met opslagsystemen

Spark is ontworpen om met verschillende externe systemen te werken. Het is *storage-system agnostic*, wat wilt zeggen dat het met een breed gamma aan storage systemen kan werken. Enkele voorbeelden van Spark-geïntegreerde storage systemen zijn:

- HDFS is de populairste keuze voor Big Data, zowel gestructureerde als non-gestructureerde data.
- Key-value stores
- Relationele databanken

3.3.2 Resilient Distributed Dataset

Voor Apache 2.0 werd er gewerkt met een Resilient Distributed Dataset of RDD object. Dit is een verzameling van objecten die parallel gemanipuleerd kunnen worden. Het is een fouttolerante werkwijze. RDD's zijn ook immutable en ze worden aangemaakt wanneer er wordt gelezen van een databron. RDD's zijn meer in de achtergrond verdwenen bij de nieuwere versies, maar ze blijven toegankelijk. RDD's komen in deze cursus niet meer aan bod.

3.4 Spark-applicaties schrijven

3.4.1 Een dataframe met nieuwe waarden aanmaken.

Stap voor stap wordt de volgende code uitgelegd:

1. Eerst wordt er een spark-object aangemaakt in de main-methode. Zonder deze kan de applicatie niet werken.
2. Er wordt een nieuwe lege lijst aangemaakt. We houden een Java-lijst bij van "Row's", ofwel de inhoud voor de dataframe.
3. We maken vijf nieuwe rijen aan. In iedere rij houden we twee waarden bij: tekst en een getal. Met de RowFactory wordt er een Row-object aangemaakt, om dan vervolgens aan de Java-lijst toe te voegen.
4. De komende twee stappen zijn optioneel, maar aan te raden. De ambitie is om een schema op te stellen. Het schema is een weergave van alle datatypes die in de dataframe aan bod komen. Zonder een schema moet Spark zelf uitvissen wat de datatypes zijn, en zelfs dan is er géén garantie dat het de juiste datatype is. In dit geval zijn er twee velden: naam in de vorm van een String en leeftijd in de vorm van een Integer. Deze lijst houdt de StructFields of schemavelden bij die in een schema moeten voorkomen. Dit is het schema nog niet!
5. Op basis van de lijst met datatypes wordt er een schema gemaakt. Het schema is van het type StructType.
6. Het dataframe wordt aangemaakt. Hiervoor komt het Spark-object van pas. Samen met de lijst van rijen als eerste parameter én het schema wordt het dataframe aangemaakt.
7. Vervolgens wordt de data gegroepeerd op basis van naam. Zo zijn er twee Brooke's in het dataframe. In het resultaat zal er maar één Brooke te vinden zijn, maar van de twee leeftijden wordt het gemiddelde genomen.
8. Het dataframe wordt in de terminal weergegeven. Alle kolommen, dus naam en leeftijd, worden weergegeven.
9. Optioneel, maar wel aangeraden om het Spark-object te sluiten.

```java
public  static void main(String[] args){

  SparkSession spark = SparkSession.builder()
                  .appName("applicatiePersonenMetLeeftijd")
                  .master("local[*]")
                  .getOrCreate();

  List<Row> inMemory = new ArrayList<>();

  inMemory.add(RowFactory.create("Brooke",20));
  inMemory.add(RowFactory.create("Brooke",25));
  inMemory.add(RowFactory.create("Denny",31));
  inMemor.add(RowFactory.create("Jules",30));
  inMemory.add(RowFactory.create("TD",35));

  List<StructField> fields = Arrays.asList(
    DataTypes.createStructField("name",DataTypes.StringType, true),
    DataTypes.createStructField("age",DataTypes.IntegerType, true);
  )

  StructType schema = DataTypes.createStructType(fields);

  Dataset<Row> df = spark.createDataFrame(inMemory, schema);

  Dataset<Row> result = df.groupby("name").avg("age");

  result.show();

  spark.close();
}
```

3.4.2 Een bestaande dataframe lezen

Een dataframe wordt vaak gelezen uit een externe bron, zoals JSON, CSV, Parquet of Kafka output. Het Spark-object bezit een methode *read()*. Deze methode geeft een DataFrameReader-object terug, wat nodig is om een bron in te lezen. Veronderstel dat we hier met dezelfde dataset te maken hebben, maar enkel al vooraf opgeslaan in een csv-bestand.

1. Spark-object aanmaken.
2. Schema maken door een lijst van StructFields te maken én die dan te gieten in een StructType-object.
3. Het CSV-bestand wordt ingelezen. Net zoals het aanmaken, gebeurt het inlezen ook met het Spark-object. In het CSV-bestand is er een header, vandaar dat er eerst de optie 'header:true' wordt meegeven. Vervolgens wordt het schema meegegeven én uiteindelijk het pad waar de CSV kan worden teruggevonden.
4. Net zoals daarnet wordt er gegroepeerd op naam. Per naam wordt de gemiddelde leeftijd teruggegeven.
5. Het dataframe wordt in de terminal getoond.

```java
public static void main(String[] args){

  SparkSession spark = SparkSession.builder()
                  .appName("applicatiePersonenMetLeeftijd")
                  .master("local[*]")
                  .getOrCreate();

  List<StructField> fields = Arrays.asList(
      DataTypes.createStructField("name",DataTypes.StringType, true),
      DataTypes.createStructField("age",DataTypes.IntegerType, true);
  )

```

```
13    StructType schema = DataTypes.createStructType(fields);
14
15    Dataset<Row> df = spark.read()
16              .option("header", true)
17              .schema(schema)
18              .csv("src/main/resources/PersonenMetLeeftijd.csv");
19
20    Dataset<Row> result = df.groupby("name").avg("age");
21
22    result.show();
23
24    spark.close();
25  }
```

Schema overerven

In de vorige twee berekeningen werd er een schema zelf, door de ontwikkelaar, aangemaakt. Dit is vrijblijvend, want Spark kan ook een schema overerven. Hieronder is er een alternatieve methode op stap twee van het vorige voorbeeld. Deze methode is minder efficiënt, want Spark moet zelf achterhalen wat de meest geschikte datatypes zijn. Daarnaast gaat Spark lijn per lijn door de volledige dataset heen. Wel kan je de workload minimaliseren door het schema op te bouwen o.b.v. een deel van de volledige dataset. Dit is een deelse oplossing, want er kunnen nog steeds problemen voorkomen bij de gekozen datatypes. Daarom is het aangeraden om zelf een schema op te bouwen.

```
1  Dataset<Row> df  = spark.read()
2              .option("header", true)
3              .option("inferSchema", true)
4              .option("samplingRation", 0.0001)
5              .csv("src/main/resources/PersonenMetLeeftijd.csv)
```

3.4.3 Een dataframe uitschrijven naar een nieuw bestand.

In de volgende code wordt er verondersteld dat er een Spark-object is aangemaakt, een schema is opgesteld én een dataset al is ingelezen. De dataset werd opgeslaan als DataFrame onder de naam 'result'.

1. Het pad wordt tweemaal gebruikt, dus dat wordt opgeslaan onder een variabele. Het pad kan zowel relatief als absoluut zijn. In dit voorbeeld is het pad absoluut.
2. Er zijn drie opties:
   (a) De inhoud van de dataframe wordt uitgeschreven naar een parquet-bestand. Dit bestand is een must bij een column-oriented databankstructuur.
   (b) Vervolgens wordt de mode meegegeven. Bij het uitschrijven wordt er een map gegenereerd. In die map kan het parquet-bestand worden teruggevonden.
   (c) Als laatste moet er een outputpad worden meegegeven.
3. Puur uit controle wordt het net aangemaakte bestand ingelezen door Spark. De outputpad, waar het parquet-bestand werd opgeslaan, dient nu als inputpad.
4. Het Spark-object wordt gesloten.

```
1  String path = "file:///C:/tmp/personen";
2
3  df.write()
4     .format("parquet")
5     .mode("overwrite")
6     .save(path);
7
```

```
8  Dataset<Row> dfParquet = spark.read().parquet(path);
9
10 spark.close();
```

3.5 Dataframe-kolommen inlezen, toevoegen en verwijderen.

Kolommen toevoegen

Een dataframe bestaat uit kolommen. Sommige functies vragen de naam van een kolom op als String, maar andere functies vereisen dat je een object van de klasse 'Column' meegeeft. In het volgende voorbeeld wordt er een extra kolom aangemaakt. De kolom is een berekening op de leeftijdskolom.

1. Bovenaan komen de imports. De klasse 'functions' is met een kleine letter geschreven, daarmee een uitzondering op de regel om alle klassen met een hoofdletter te laten starten.
2. Er worden twee kolommen aan de dataset toegevoegd. Een expressie of where-clausule wordt meegegeven met de *expr(...)*-functie.

```
1 import org.apache.spark.sql.Column;
2 import static org.apache.spark.sql.functions.col;
3
4 df = df.withColumn("ageNextYear2", expr("age + 1"))
5       .withColumn("over30", expr("age > 30"));
```

Kolommen selecteren

Tot nu toe werden alle kolommen van de dataframes bij de keuze betrokken. Spark laat de ontwikkelaar toe om kolommen, net zoals bij SQL, te kiezen. Hieronder wordt er een voorbeeld gegeven van de verschillende kolommen. Er wordt verondersteld dat er al een spark-object is aangemaakt én dat het pad in een String 'inputpath' wordt bijgehouden.

1. Er wordt een nieuwe dataset opgehaald. De dataset is opgeslaan in een CSV-bestand. Het bevat vijf kolommen: id, score, naam, jaartal en kwartaal. Enkel bij dit voorbeeld wordt het schema overgeërfd.
2. De dataset wordt in de terminal getoond. Alle vijf kolommen worden weergegeven.
3. Met de *select*-functie worden enkel de score, jaartal en kwartaalkolommen opgehaald. Een dataframe is *immutable*, dus het object moet worden overschreven.
4. De dataset bestaat nu uit drie kolommen. 'Id' en 'naam' zijn niet meer terug te vinden in de dataset.

```
1  Dataset<Row> df = spark.read()
2              .option("header",true)
3              .option("inferSchema",true)
4              .option("sampleRation", 0.001)
5              .csv(inputpath);
6
7  df.show();
8
9  df = df.select("score","jaartal","kwartaal");
10
11 df.show();
```

Kolommen verwijderen

Een kolom kan op een passieve wijze worden weggelaten door de kolom niet in een *select* te betrekken. In sommige gevallen, bijvoorbeeld bij een groot aantal kolommen, is het handiger om één specifieke kolom te verwijderen, in plaats van een select uit te voeren. De gouden regel is dat dataframes immutable zijn, dus hier moet de nieuwe versie van de dataframe opnieuw wordt toegekend.

```
df = df.drop("id","naam");
```

Kolommen hernoemen

Kolomnamen kunnen worden aangepast. Bij het toevoegen wordt er vaak een standaardnaam toegekend aan een kolom, bijvoorbeeld *count* of *sum*. Hieronder wordt de kolom met de naam *count* aangepast naar *aantalRijen*. Na een berekening kan je ook de *alias*-functie gebruiken.

```
df = df.withColumnRenamed("count","aantalRijen");
df = df.count().alias("aantalRijen");
```

3.5.1 Dataframes filteren

Voorbeelden van de studentendataset

1. Hou een dataset bij met alle rijen die een score hebben van 'A+'.
2. Hou een dataset bij met alle rijen waarvan de score 'B' is én de score werd behaald in het jaar 2010 of 2011.
3. Neem de dataset van nummer 3 en behoud enkel de unieke rijen.

```
Dataset<Row> dfAplus = df.select()
             .where(col("grade")
             .equalTo("A+");

Dataset<Row> dfB1011 = df.select()
             .where(col("grade").equalTo("B")
                     .and(col("year").isin(2010,2011)));

Dataset<Row> dfUniekB1011 = dfB1011.distinct();
```

3.5.2 Aggregaties uitvoeren op dataframes

Voorbeelden van de studenten-dataset

```
Dataset<Row> dfCountPerJaar = df.select("year")
       .groupby("year")
       .count()
       .orderBy(desc("count"));

Dataset<Row> dfStatisticsPerJaarEnOnderwerp = df.select("year","subject","score")
       .groupby("year","subject")
       .agg(max("score"), min("score"), avg("score"))

```

Voorbeelden van de ViewingFigures-oefening

```
Dataset<Row> chaptersPerCourse = chaptersDF
                    .drop("chapterId")
                    .groupBy("courseId").count()
                    .withColumnRenamed("count", "chapters");
```

3.5.3 User Defined Functions

User Defined Functions of UDF's worden ingezet wanneer een berekening de ingebouwde functies van Spark overstijgt. De berekening maken is mogelijk, maar de logica is te complex en vereist daarmee een aparte aanpak. Bij een UDF zijn er drie stappen:

1. UDF schrijven.
2. UDF registreren.
3. UDF oproepen.

```
UDF1<Double, Integer> score = new UDF1<Double, Integer>() {
  public Integer call(Double percent) throws Exception {
    if (percent > 0.9) {
      return 10;
    } else if (percent > 0.5) {
      return 6;
    } else if (percent > 0.25) {
      return 2;
    } else {
      return 0;
    }
  }
};

spark.udf().register("berekenScore", score, DataTypes.IntegerType);


Dataset<Row> result = percentageDF
  .withColumn("score",  call_udf("berekenScore",col("percentage")))
  .drop("percentage")
  .groupBy("courseId").agg(sum("score").as("total"))
  .join(titleDF, "courseId")
  .orderBy(desc("total"));
```

Hoofdstuk 4

Spark Streaming

4.1 Stream Processing

Structured Spark batch processing

Voordien gebeurde alles bij Spark in één keer. Dit wordt ook batch processing genoemd en omvat de volgende drie kenmerken:

- De data is bounded. De analogie wordt gemaakt met de logistiek bij de Colruyt. Wanneer de winkel dichtgaat, dan wordt er een verslag gemaakt van alles wat die dag werd verkocht. Bij een CSV of XML-bestand is de laatste rij gekend.
- Er wordt géén data toegevoegd. Het lezen van data gebeurt in één batch. Werken met batches is geen ideale methode voor real-time dataverwerking.
- Alle data wordt in een tabel bijgehouden.

4.1.1 Spark Stream Processing

Data wordt continu aan het systeem toegevoegd en moet in real-time worden verwerkt. Bijvoorbeeld bij een live dashboard. De status van de data is hier belangrijk. De data is niet-gebonden of *unbounded*, wat wilt zeggen dat er continu rijen achteraan worden toegevoegd. Het probleem is dat er géén oneindige tabel in het geheugen is. Een stream bestaat uit drie kenmerken:

- Het is *unbounded*, al denkt het systeem niet zo. Er is géén vast einde van de inputdata gekend en de stroom vloeit op een niet-gekende manier. Piekperioden, rustige momenten en alles tussenin zijn mogelijke scenario's.
- Er wordt telkens data achteraan de inputdata toegevoegd. Iedere nieuwe record in een datastream wordt behandeld als een nieuwe rij. De rij wordt telkens aan de inputtabel achteraan toegevoegd. Het real-time aspect klinkt als een radicale verandering voor het programmeren, maar alle code wordt geschreven zoals ze voor een batch processing job werd geschreven. Deze job moet telkens getriggerd worden om data te verkrijgen. Pas bij de trigger kijkt Spark voor nieuwe data en die data wordt achteraan de inputdata toegevoegd.
- Alle data wordt in een tabel bijgehouden.

Data ophalen met sockets

Spark zal gemiste input vergeten, want er wordt gewerkt met sockets. Een socket leest enkel de binnenkomende info. Data die voordien werd gestuurd, zal niet worden gelezen. Historiek wordt niet bijgehouden. Als de deur toe is, dan zal niemand staan wachten tot de deur opengaat.

Output sink

Dit is de bestemming waarnaar de data wordt geschreven. Dit kan een databank, bestand of systeem zijn dat data kan verwerken of manipuleren. Bij stream processing is dit het systeem dat de resultaten van real-time verwerkingsoperaties zal bijhouden. Een output sink dient als werkpaard om de gestreamede data persistent bij te houden.

4.1.2 Outputmodes

Er zijn drie outputmodes: append, update en complete. Deze instelling geeft aan hoe de data tijdelijk moet worden bijgehouden.

- Appendmode zal enkel de nieuwe rijen als resultaat zien. De historiek of bestaande rijen worden niet getoond of aangepast. Dit is de standaardmodus. Met enkel de nieuwe rijen is er weinig mogelijk. Zo is WordCount niet odnersteudn, want het aantal voorkomens wordt niet lang bijgehouden én de vorige geschreven data wordt niet betrokken. Dit is de meest efficiënte outputmode.
- Updatemode zal zowel nieuwe als oude rijen teruggeven. Van de oude rijen worden enkel aanpassingen gemaakt. Als een woord bij de vorige trigger werd gegeven, en nu zijn er vijf voorkomens meer, dan wordt dat aantal opgeteld.
  - Sommige outputsinks ondersteunen deze modus niet, want de bestanden hebben geen interne structuur. Eenmaal een bestand geschreven is, dan is het moeilijk om dit aan te passen. Bijvoorbeeld als de vorige rij vijf kolommen had, en nu wordt er een rij toegevoegd met zes kolommen, dan zal dit resulteren in een fout.
- Completemode zal het volledige resultaat uitschrijven. De output zal stelselmatig groeien en de job zal in de tussentijd niet draaien. Dit is een *go-to* mode voor aggregatieberekeningen, maar wordt met voorkeur gekozen voor kleine datasets. WordCount werkt wel bij deze modus. Deze mode is de meest flexibele outputmode.

Voorbeeld stock prices

Stel dat er een continue flow van stock prices is. Bij appendmode worden enkel de binnenkomende shares getoond. Bij updatemode worden enkel de veranderingen, dus nieuwe en aangepaste shares, weergegeven. Bij completemode worden alle shares na iedere trigger uitgeprint.

4.2 Code

Een streamingquery opbouwen vereist vijf stappen:

1. Inputsources bepalen
2. Data omzetten
3. Outputsink en outputmode bepalen
4. Processingsdetails bepalen
5. Query starten

4.2.1 Inputbron bepalen

Nu wordt er gewerkt met een *readStream* methode. Bij batch-processing werd er gewerkt met een gewone *read*-methode. De keuzes bij stream-processing lopen in lijn met batch-processing.

- De format zal in de meeste gevallen een socket zijn, wat voordien een CSV, parquet of tekstbestand was.

- Bij de opties worden de netwerkinstellingen en Kafka-informatie meegegeven. Het poort van de socket, gegevens over de broker, etc.
- Load start het leesproces.

```
Dataset<Row> df = spark.readStream()
                .format("socket")
                .option("host", "localhost")
                .option("port", 9999)
                .load();
```

4.2.2 Data omzetten

Je hebt enkel de huidige data nodig. Data kunnen zich in twee statussen bevinden: stateful en stateless. De opdeling loopt in lijn met *wide* en *narrow* berekeningen.

- Bij stateful kan iedere rij op zich worden verwerkt. Met enkel de huidige informatie is het systeem voldoende. Bijvoorbeeld mapping volgens select of filters.
- Bij stateless is de informatie van voordien nodig. Stateless komt bij een aggregatie goed van pas. De meeste combinaties worden ondersteund, tenzij de combinaties te moeilijk zijn om op een incrementele manier te berekenen.

```
Dataset<Row> words = lines.select(
  explode(split(col("value"), "\\s")).as("word")
);

Dataset<Row> counts = words.groupBy("word").count();
```

4.2.3 Outputsink -en modus bepalen

Er wordt bepaald hoe de geschreven outputdata moet worden geschreven. De output-mode en de outputlocatie moeten in de code te zien zijn. De code hieronder bouwt verder op de de counts-dataset die net werd opgesteld:

- In de code hieronder wordt er gewerkt met een complete outputmode, want er wordt een aggregatie op de dataset gemaakt.
- Alle output wordt naar de console geschreven. Andere mogelijkheden zijn: 'memory', 'kafka', 'rate', 'parquet' of 'csv'.

```
DataStreamWriter<Row> writer = counts.writeStream()
        .format("console")
        .outputMode(OutputMode.Complete());
```

4.2.4 Processingdetails bepalen

De laatste stap voor het starten van de query is het specifiëren hoe de data moet worden verwerkt. Er zijn vier mogelijkheden:

- De standaardoptie is het behandelen in microbatches. Deze worden kort na elkaar uitgevoerd. Wel moet de onderlinge tijd worden meegegeven.
- Werken met een vast triggerinterval is soortgelijk aan Linux cronjobs. Op een vooraf gespecifieerd moment wordt de query uitgevoerd.

- Alle data wordt op één moment verwerkt, daarna stopt de applicatie. Alle nieuwe data wordt in één batch verwerkt. Dit is handig wanneer er controle over een externe scheduler moet worden genomen.

- Continuous is een experimentele modus. Hierbij de data zo snel als mogelijk continu verwerkt, dus niet meer in micro-batches. Enkel een klein aantal dataframebewerkingen ondersteunen deze modus. De latency is hier minimaal.

Code

Dit is een aanvulling op de code van stap 3. Onder format en outputmode wordt er een trigger meegegeven. In dit geval gebeurt de verwerking iedere seconde.

```
DataStreamWriter<Row> writer = counts.writeStream()
        .format("console")
        .outputMode(OutputMode.Complete())
        .trigger(Trigger.ProcessingTime(1, TimeUnit.SECONDS));
```

4.2.5 Query starten

De query wordt gestart met een start-methode op de DataStreamWriter. Dit geeft een StreamingQuery terug, wat de actieve query voorstelt. Een StreamingQuery kan dienen voor verder beheer van de query.

```
StreamingQuery streamingQuery = writer.start();
try{
  streamingQuery.awaitTermination();
} except {
  e.printStackTrace();
};
```

4.3 Sources & Sinks

4.3.1 Bestanden lezen

Bestanden in een directory kunnen worden ingezet als input voor een stream. Wel moeten er drie zaken in het achterhoofd worden gehouden:

- Alle bestanden moeten in eenzelfde formaat zijn geschreven. Alles is ofwel een CSV-formaat, ofwel een tekstformaat, ofwel een specifiek ander formaat.

- Bestanden moeten atomisch beschikbaar zijn. Tijdens het lezen mogen de bestanden niet worden bewerkt of verwijderd.

- Bestanden worden genomen op basis van de timestamp, maar het verwerken daarentegen is niet vooraf bepaald. Alle bestanden worden parallel gelezen. Er is geen garantie welk bestand er eerst wordt bekeken.

- Naar een bestand schrijven gebeurt in append-only mode. Het is makkelijker om bestanden aan een bestaande directory toe te voegen, dan de inhoud van bestaande bestanden aan te passen.

4.3.2 Kafka-inhoud lezen

Net zoals bij bestanden kan de inhoud van een Kafka topic worden gezien als een stream. Het schema bij Kafka Dataframes zal altijd hetzelfde zijn. Parsen zal wel altijd noodzakelijk zijn.

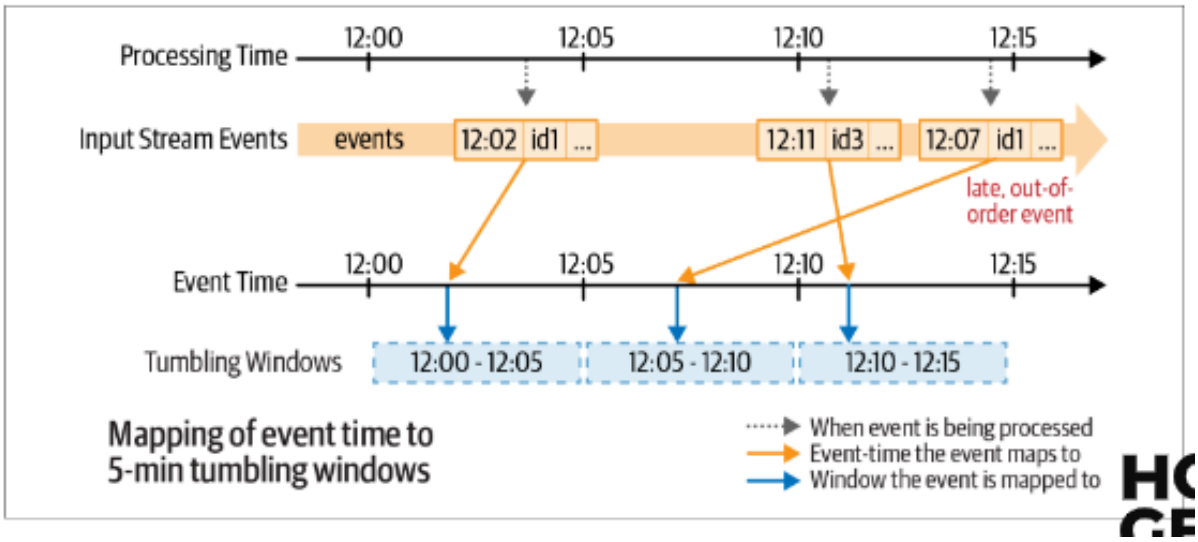
- De key-value wordt als binary teruggegeven, maar dit is in binary.
- De topic wordt als een String teruggegeven.
- De partitie waar de record in terug kan worden gevonden.
- De line offsset value van een record.
- De timestamp van de record en het type van de timestamp.

Naar Kafka schrijven.

Het is mogelijk om in alle drie outputmodes naar Kafka uit te schrijven. Ter aanvulling is het ook mogelijk om van een topic naar een topic te schrijven.Completemode wordt wel niet aangeraden. Bij het uitschrijven naar Kafka moet er met het volgende schema worden gewerkt:

- De key is optioneel. Dit in de vorm van een String of een binaire waarde.
- De value is verplicht.
- de topic is verplicht wanneer er geen topic werd meegegeven als optie. Als dit werd meegegeven in de applicatie, dan wordt deze kolom in het schema genegeerd.

```
SparkSession spark = SparkSession.builder();

final String topic = args[0];

Dataset<Row> messages = spark.readStream()
        .format("kafka")
        .option("kafka.bootstrap.servers"), BOOTSTRAP_SERVERS)
        .option("subscribe", topic)
        .load();

StreamingQuery query = null;

try {
  query = messages.writeStream()
     .format("console")
     .outputMode(OutputMode.append())
     .option("checkpointLocation", CHECKPOINT_LOCATION)
     .start();
} catch {
  e.printStackTrace();
}

try {
  query.awaitTermination();
} catch (StreamingQueryException e) {
  e.printStackTrace();
}

spark.close();
```



Figuur 4.1: Tumbling event time. Learning Spark (2022)

4.4 Aggregations with Event-Time windows

Gebruikelijk worden aggregaties niet over een volledige stream gedaan, maar in kleine stukken over een tijdspanne. Bijvoorbeeld in een casus waar sensoren data iedere minuut doorsturen. Hier ligt de nadruk eerder op wanneer de data werd aangemaakt en niet op wanneer de data werd verwerkt door Spark. Iedere timestamp wordt naar een window gemapt. Zo wordt er een groupby op het interval uitgevoerd.

Tumbling windows

Tumbling windows zijn windows van een vaste grootte en afwisselend. Ieder event wordt aan één window toegekend, en de windows overlappen elkaar niet.

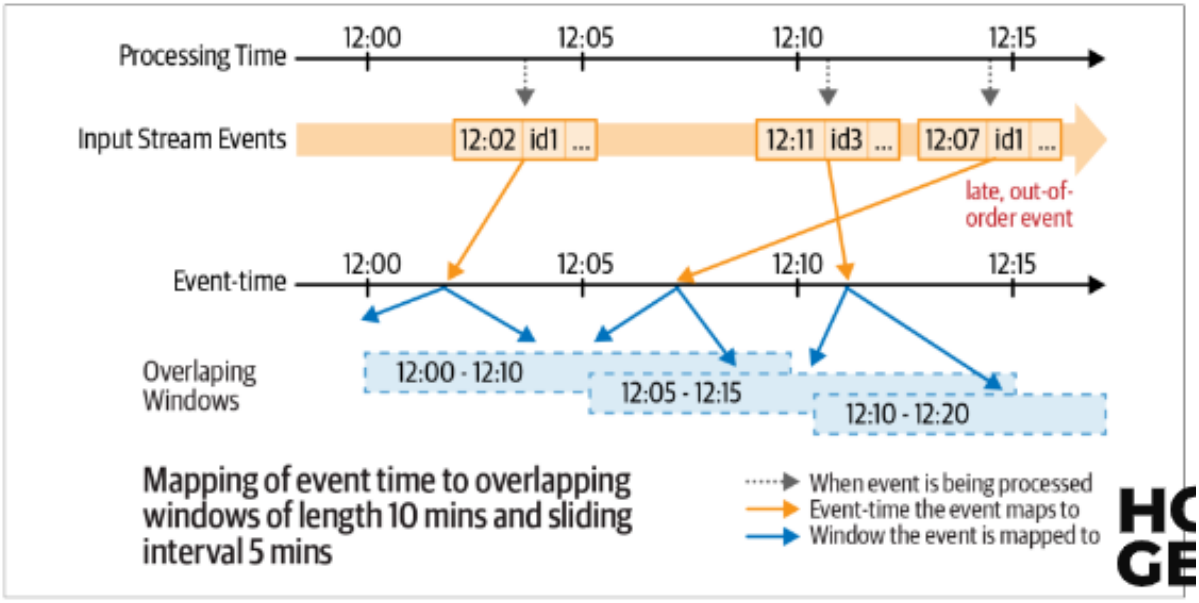
Overlapping windows

Overlapping windows zijn time windows die elkaar onderling overlappen. Een event kan tot meerdere overlappende windows behoren.

WordCount met time interval

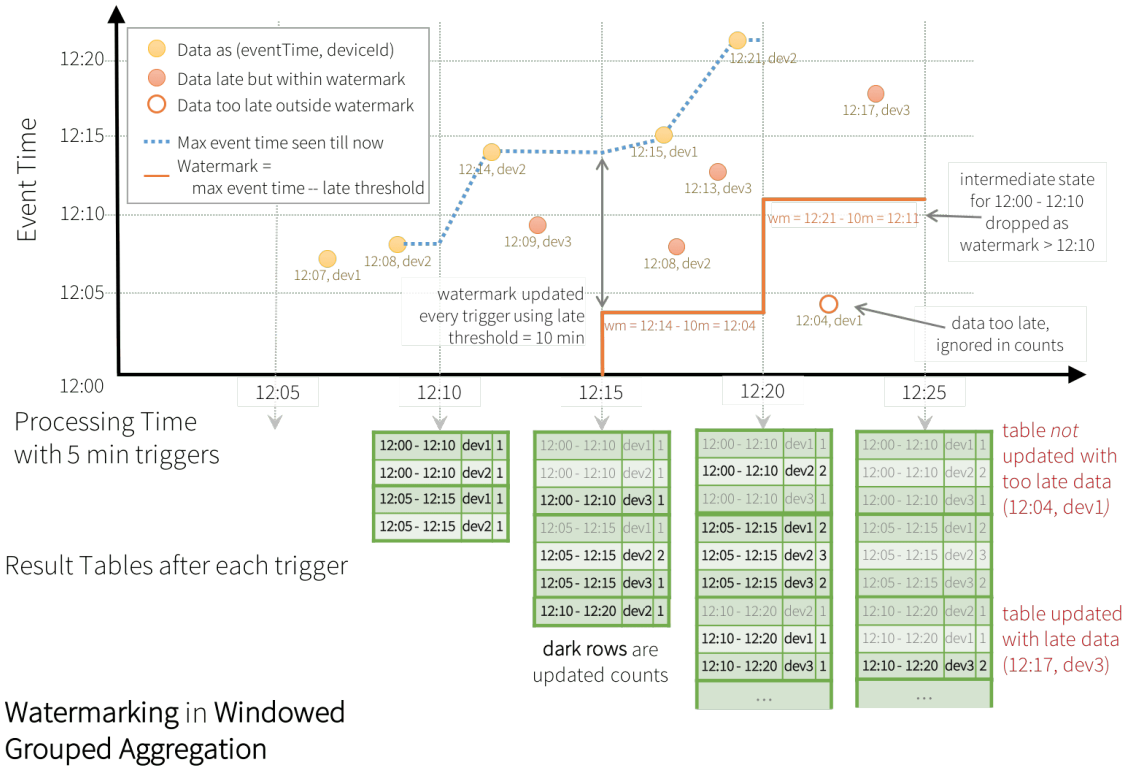
1. Allereerst wordt er een spark-object aangemaakt.
2. Vervolgens wordt de dataset als stream-object ingelezen. Er wordt gelezen vanuit een socket met de gewoonlijke opties. Wel wordt er een extra kolom toegevoegd, namelijk een timestamp. We veronderstellen voor de volgende stap dat het dataframe minstens twee kolommen heeft: value en timestamp.
3. (a) We veronderstellen dat de value-kolom een doorlopende tekst is van alle woorden. Dit is niet nodig, dus de explode-functie wordt gebruikt om alle woorden uit de tekst te halen. Dit wordt opgeslaan in een nieuwe kolom.

   (b) Op basis van de timestamp en het woord wordt er gegroepeerd. Hier moet er binnen een window worden gewerkt. De window-functie wordt aangesproken op de timestamp-kolom met als tijdspanne 10 seconden en sliding interval van 5 seconden.
4. Uiteindelijk wordt count als aggregatie-functie gebruikt. De order-by voorziet een chronologische volgorde.



Figuur 4.2: Overlapping time windows. Learning Spark (2022)

```
SparkSession spark = SparkSession.builder()

Dataset<Row> lines = spark.readStream()
                .format("socket")
                .option("host","localhost")
                .option("port",9999)
                .option("includeTimestamp",true)
                .load();

Dataset<Row> output = lines.withColumn("word",explode(split(col("value"),"\\s")))
                .select("word","timestamp")
                .groupby(window(col("timestamp"),"10 seconds", "5 seconds"), col("word"))
                .count()
                .orderBy(col("window"), col("count"));

StreamingQuery query = null;

try{
  query = output.writeStream()
            .format("console")
            .option("truncate",false)
            .option("numRows", 100)
            .outputMode(OutputMode.Complete())
            .trigger(Trigger.ProcessingTime(5,TimeUnit.SECONDS))
            .start()
} catch (TimeoutException e){
  e.printStackTrace();
}

try {
  query.awaitTermination();
} catch (StreamingQueryException e) {
  e.printStackTrace();
}
```



Figuur 4.3: De twee events komen aan tussen 12u20 en 12u25. Het watermark wordt gebruikt om het verschil tussen *late* en *too-late* aan te duiden. Opgehaald uit Databricks (2022)

4.4.1 Watermarking

Tijdens het streamen wordt de staat van de data bijgehouden. De staat vergroot zolang het systeem data zal streamen. De kans op *unbounded* data vergroot. De oplossing hiervoor is *watermarking*. Een watermark is een event-time threshold. De event-time is het moment wanneer iets bij de bron werd aangemaakt Enkele kenmerken van watermarking:

- De watermark kijkt naar de grootste of meest recente event-time. Deze threshold definieert welke events te laat in een stream terechtkomen. Een watermark delay is de tijd dat het systeem wacht op late data, voordat het systeem de data verwerpt.
- Enkel de relevante data wordt behouden. Het systeem verwerpt achterhaalde data.
- Enkel de laatste timestamp wordt geüpdate. Een watermark stelt hier een bezemwagen voor. Bijvoorbeeld iedere event met een event-time voor 12u10 wordt niet meegeteld, want de range ligt tussen 12u10 en 12u20.

4.4.2 Streaming joins

Er zijn twee mogelijkheden:

1. Een stream joinen met een statische dataframe.
2. Twee streams met elkaar joinen. Dit kan enkel met een outer join.

Stream-static joins

Data uit een gestreamede dataframe wordt gecombineerd met data uit een statische dataframe, bijvoorbeeld een dataframe opgebouwd uit een CSV. De code om dit mogelijk te maken, lijkt zeer sterk op de code voor een join bij een gewone dataframe.

- Dit is een stateless-operatie, daarom is er geen watermarking nodig.
- Een stream houdt geen state van de dataframe bij.
- Dit kan volgens een left-outer, inner of right outer join. Andere outer join opties, zoals een full outer join, zijn niet mogelijk. Het probleem is te wijten aan het incrementeel aflopen van de data.
- Iedere n-aantal seconden wordt de volledige statische dataframe gelezen. Caching zorgt ervoor dat de dataframe één maal moet worden gelezen.
- Aanpassingen aan de statische file worden tijdelijk genegeerd, vooral omdat caching wordt gebruikt. Pas bij het herstarten van de streamingquery worden de aanpassingen doorgevoerd. Dit hangt af van het gekozen bestandstype van de statische dataframe.

```
1 Dataset<Row> staticDF = spark.read() ...;
2 Dataset<Row> streamingDF = spark.readStream() ... .;
3 streamingDF.join(staticDF, "type");
4 streamingDF.join(staticDF, "type", "left_outer");
```

4.4.3 Stream-Stream joins

Data uit een stream wordt gecombineerd met data uit een andere stream.

- Beide bronnen veranderen snel van inhoud.
- Bijvoorbeeld een stream van iedere aankoop en een stream met info van iedere klant. De twee streams kunnen met elkaar worden ge*joined*.
- De grootste uitdagingen zijn:
  - De incompleetheid van de datastream. De kans dat er matches zijn tussen de inputs van de twee datastreams is kleiner.
  - Events in twee streams kunnen in eender welke volgorde komen. Mogelijk met vertraging wat het moeilijker maakt om matches terug te vinden.
- Om deze changes aan te pakken, wordt er gewerkt met *buffering*.

```
1 Dataset<Row> purchases = spark.readStream(). ...;
2 Dataset<Row> customers = spark.readStream(). ...;
3 Dataset<Row> matched = purchases.join(customers","adId);
```

Buffering

Bij buffering wordt de inputdata van beide streams tijdelijk bijgehouden. Zo wordt er gekeken naar matches, zelfs al is de data in een andere volgorde binnengekomen. Structured streaming verloopt zo effectiever, en je hebt een beter zicht op de data.

4.5 Keeping the state bounded

Het probleem bij de stream-stream joins is dat de engine een unbounded aantal streaming state kan bijhouden. Uit het voorbeeld van de aankopen zal niet iedere klant worden gekoppeld aan een aankoop en vice versa. Rijen zullen niet worden gebruikt en zo een lange periode in de buffer blijven.

State beperken

Twee zaken moeten zijn gekend:

- Wat is de maximale tijdsrange tussen de twee events?
  - In het voorbeeld van de aankopen: Hoe lang duurt het vooraleer een klant een aankoop uitvoert?
- Wat is de maximale tijsdperiode waarin een event tussen de bron en de processing engine kan worden vertraagd?
  - In het voorbeeld van de aankopen: Mogelijks kan er vertraging optreden bij het aankopen. De verbinding speelt hier een belangrijke rol.

State clean-up

De *delay limits* en *event-time constraints* kunnen omgezet worden in dataframebewerkingen. Bij deze bewerkingen worden watermarks en *time range conditions* gebruikt. Er zijn twee stappen:

1. Stel een watermark delay op voor beide inputstreams. Zo weet de engine hoe lang de vertraging van de input kan zijn.
2. Stel een *event-time constraint* op over de twee inputs heen. Zo weet de engine welke rijen verplicht zijn. M.a.w. de rijen die wél voldoen aan de *time-constraint*. Een *time-constraint* kan op twee manieren worden opgesteld:
   - Time range join conditions zoals *between righttime and (righttime + ïnterval 1 hour")*
   - Join op event-time windows zoals *"lefttimewindow = righttimewindow"*.

```
Dataset<Row> purchases = spark.readStream() ...
Dataset<Row> customers = spark.readStream() ...

Dataset<Row> purchasesWWatermark = purchases.withWatermark("purchasesTime","2 hours");
Dataset<Row> customersWithWatermark = customers.withWatermark("customersTime", "3 hours");

impressionsWithwatermark.join(
  clicksWithWatermark,
  expr(
    "custID = purchasesCustId AND " +
    "customersTime >= purchasesTime AND " +
    "customersTime <= purchasesTime + interval 1 hour"
  )
);
```

Outer joins with watermarking

De outer-type join specifiëren gebeurt helemaal onderaan. Wel zijn er enkele opmerkingen over outer joins:

- De watermark delay en event-time constraints zijn verplicht. De engine moet weten wanneer een event niet gaat matchen, anders leidt dit tot een null-resultaat.

- De *outer* null-resultaten worden met een delay aangemaakt. De engine doet dit om zeker te zijn dat er mogelijke matches niet worden verworpen.

```
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr(
    "custID = purchasesCustId AND " +
    "customersTime >= purchasesTime AND " +
    "customersTime <= purchasesTime + interval 1 hour",
    "leftOuter"
  )

);
```

Hoofdstuk 5

Kafka

Probleemstellingen

Veronderstel dat er een systeem is met n-aantal bronsystemen en m-aantal doelsystemen. Om de twee met elkaar te linken, zou ieder bronsysteem moeten verbonden zijn met ieder doelsysteem. Dit komt gelijk op hoogstens een *n x m* aantal berekeningen. Elke connectie neemt resources van het systeem. Daarnaast zijn er drie zaken waarmee er rekening mee moet worden gehouden.

- Protocol geeft aan hoe de data moet worden getransporteerd.
- Het formaat of de manier hoe de data is *geparsed*.
- Dataschema en de evolutie van de data: hoe is de data gevormd en hoe kan het veranderen. Wat is de locatie van de data?

Introductie Kafka

Kafka is platform dat streaming tussen bron- en doelsystemen ondersteund. Tussen de systemen is er een *broker*. Met Kafka als tussenpersoon verandert dit naar een *n + m*-aantal berekeningen.

Werking

1. Wanneer een bronsysteem data aanmaakt, dan wordt er data verstuurd naar Kafka. Events die de website heeft gestuurd wordt naar Kafka gestuurd. Een doelsysteem weet niet van wie de data komt.
2. Kafka slaat het bericht op in een topic. Een topic is een logische categorie waarin de message is opgeslaan.
3. Het doelsysteem kan dan de message van de topic opgebruiken, verwerken en nodige acties nemen.

Voordelen Kafka

- Het is een volwassen stuk open-source software.
- Het is een gedistribueerd systeem. De architectuur is fouttolerant gebouwd.
- De horizontale schaalbaarheid is goed. Op honderden brokers blijft het systeem even efficiënt werken.
- De *latency* is minimaal.

Use cases

De twee belangrijkste use cases om te weten:

- Kafka wordt ingezet bij het verzamelen van metrieken en de bijhorende analyse. Die data wordt verwerkt in een *recommendation*-systeem, bijvoorbeeld bij Netflix, YouTube of LinkedIn.
- De voornaamste use case is stream processing, waar er een oneindige stroom van data wordt beheerd.

5.1 Concepten van Kafka

Topic

Eén datastroom van boodschappen die met elkaar te maken hebben. De structuur is vergelijkbaar met een tabel. Er is geen beperking, buiten hardware-limitaties, op het aantal topics dat in een Kafka-systeem kan worden aangemaakt.

Partitie

Een topic is opgedeeld in een aantal partities. het doel van meerdere partities is om de *workload* en doorvoer evenwaardig te verdelen. Het aantal partities staat vast wanneer een topic wordt aangemaakt.

- De volgorde ligt definitief vast. De volgorde wordt bepaald door de offset. Het is vergelijkbaar met een oude typmachine. Eenmaal een lijn is geschreven, dan blijft dit zo. De volgorde waarin een partitie wordt geschreven, is dezelfde volgorde waarin een lezer de partitie zal lezen.
- Dit is een *append-only* logfile. Als iets geschreven is in de partitie, dan zal dit achteraan zijn. Daarnaast kan een boodschap niet gewijzigd of verwijderd worden. Als je een foute boodschap stuurt, dan moet je een tweede boodschap sturen om de fout recht te zetten.
- Een broker kijkt nooit naar de data. Ze plakken er enkel metadata aan, maar ze bewerken geen inhoud.
- Er is geen verhouding tussen nummers van verschillende partities. De volgorde en offset is enkel zinvol binnen dezelfde partitie. Offset 3 bij partitie 0 heeft geen invloed of correlatie met offset 3 van partitie 1.

Voorbeeldcasus

1. Iedere vrachtwagen in een vloot rapporteert de locatie van de GPS naar Kafka. Iedere vrachtwagen zal telkens, iedere 20 seconden, naar dezelfde topic schrijven. Het bericht omvat de huidige coördinaten.
2. De data van dezelfde truck moet in dezelde partitie worden opgeslaan.
3. Bij verschillende partities is er geen garantie dat een vrachtwagen telkens naar dezelfde partitie zal schrijven.

Kenmerken

Kafka bezit vier belangrijke kenmerken:

- De offset is enkel geldig binnen een gekozen topic partitie.

- De volgorde is enkel binnen een partitie gegarandeerd. Als je als consumer een topic leest, dan gebeurt dit hoogstwaarschijnlijk niet in dezelfde volgorde als hoe ze werden gepost.
- Data kan opstapelen. Als er niets wordt toegevoegd of gelezen, dan zal Kafka alles ouder dan een week verwijderen. Deze periode kan worden veranderd. Een tweede mogelijkheid is om een capaciteitslimiet in te stellen, bijvoorbeeld wanneer de partitie te groot is.
- Data is immutable. Eenmaal geschreven, kan de data niet meer worden aangepast.
- Als er geen key wordt meegegeven, dan kiest Kafka zelf een partitie om de data naar uit te schrijven. Deze keuze gebeurt willekeurig om de partities even groot te houden. Op de key wordt er een hash berekend.

5.2 Rollen

5.2.1 Broker

Iedere Kafka-cluster bestaat uit minstens één *broker*.

- Een *broker* is een server met een eigen ID en topic-partitions.
- Er is geen specifieke master-broker. Er wordt verbonden met een willekeurige broker die de metadata van de cluster bijhoudt. Verbinden met één broker staat gelijk aan verbinden met de hele cluster.
- Bij enkel één broker heb je geen fouttolerantie. Pas vanaf twee brokers biedt je systeem fouttolerantie aan. Per standaard zijn er drie brokers.
- Het toekennen van een partitie aan een broker gebeurt o.b.v. een algoritme.

Data partitioneren over verschillende brokers

Er zijn drie brokers en twee topics. Topic A heeft drie partities en topic B heeft twee partities. Kafka probeert altijd om de partities van één topic op verschillende brokers te zetten. Verschillende topics hoeven niet hetzelfde aantal partities te hebben. Als een broker uitvalt, dan zal replicatie hier de situatie redden.

Replicatie

Als er één broker uitvalt, dan is de data beschikbaar op een andere broker. Hou rekening met de hoeveelheid data dat er op een systeem wordt bijgehouden. De replicatiefactor wijst aan hoeveel keer de data moet worden bijgehouden. Data die van enorm belang zijn, krijgt best een hoge replicatiefactor toegekend, terwijl eerder dev-gerelateerde zaken zoals logging kan op een replicatiefactor van één worden gehouden.

5.2.2 Data synchroon houden met een *partition leader*

Er wordt het onderscheid gemaakt tussen leaders en followers:

Leaders

Een partition leader wordt op één broker tegelijkertijd toegekend. Er kan niet meer dan één broker leider zijn van een partitie. Verder bezit een leider twee belangrijke kenmerken.

- De leider heeft volledige toegang tot de data en is de enige die data kan sturen en ontvangen. Daarmee is de leider diegene die wordt aangesproken wanneer er iets in de data moet worden aangepast.

- De leider kent de data-achterstand van de replica's. Met fetch requests kunnen volgers achterhalen hoeveel data ze nog moeten inhalen.
- De leider zal de status van *out-of-sync* toekennen aan volgers die na tien seconden niets laten weten.

Follower

De andere brokers volgen de leader en worden followers genoemd. De followers hebben geen toegang tot de data, maar luisteren enkel naar de aanpassingen van de leader. Een applicatie kan, binnen dezelfde databankrack van de leader, een volger aanspreken. Dit gebeurt enkel wanneer het luisteren naar de volger resulteert in het snelst en meest correct ophalen van de data. Hiervoor moet de follower wel in-sync zijn. De volgers sturen *fetch requests* naar de leader om de lokale data te updaten met de meest recente versie.

Bij het uitvallen van een leader

Wanneer een leader opeens uitvalt, dan wordt er een broker als leader van een partitie toegekend. Dit gebeurt enkel als de replica *in-sync* is met de data. Er is een verschil met in-sync replica's en out-of-sync replica's.

- Een *in-sync* replica is een replica waarbij er geen verschil is tussen de replica en de leider. Zij bezitten de meest recente data. Enkel in-sync replica's komen in aanmerking om leider te worden.
- Een *out-of-sync* replica beschikt niet meer over de meeste recente data. De leader bepaalt wie *out-of-sync* is.

5.2.3 Producer

Een producer weet, afhankelijk van de metadata, naar welk topic en partitie zij zal sturen. Veel van de moeilijkheden worden voor de producer verborgen.

Configuratie

Belangrijke configuratie is wanneer de producer iets ziet als geslaagd. Er zijn drie verschillende waarden:

- acks=0 wijst erop dat de producer een bericht verstuurd en verder niets doet. De data verzenden is voldoende. Het is niet betrouwbaar, maar wel snel. Dit is interessant bij sensordata, maar minder interessant bij een banktransactie.
- acks=1 wijst erop dat de producer een bericht verstuurd én wacht op bevestiging van de broker. Dit leidt tot deelse geruststelling, want de dataverlies is beperkt. De producer geen weet op wat er na die stap gebeurt én of de data bij de *in-sync replica's* is terechtgekomen.
- acks=all wijst erop dat alle producers wachten op de leider, en alle *in-sync replicas* moeten de data ontvangen hebben. In het geval van minstens één *in-sync replica* is de producer zeker dat de data in minstens twee buffers zijn opgeslaan.
- min.insync.replicas=1 is een optionele parameter. Hierbij wordt er ingesteld hoeveel *insync-replica's* er per broker moet zijn ingesteld.
  - Replicatiefactor 3 met min-insync-replicas van 1 is gevaarlijk, want dat geeft aan dat de data verloren zijn wanneer de leader uitvalt. De producer zal denken dat het bericht is toegekomen.
  - Replicatiefactor drie met min-insync-replicas van 2 wilt zeggen er minstens één *out-of-sync* volger wordt toegelaten.
  - Replicatiefactor 3 met min-insync-replicas van 3 is functioneel ok, maar de beschikbaarheid van het systeem zal *tanken*. Hier verwacht je dat je géén trage volgers zal hebben.

Keys

Per boodschap wordt er een key verstuurd. Alle boodschappen met dezelfde sleutel zal naar dezelfde partitie worden verstuurd. Als er geen specifieke requirement is, dan wordt de key weggelaten. Het systeem zal de load verdelen, met behulp van een *round robin* manier, over de verschillende brokers.

5.2.4 Consumer

Consumers lezen van één of meer topics. De consumers worden per partitie toegekend. Als er een fout in de cluster voordoet, dan zullen de consumers zich daar automatisch van herstellen. Data wordt gelezen in een bepaalde volgorde in een partitie. Over de partities heen is er geen garantie dat de volgorde parallel loopt of identiek is.

Consumer group

Een groep wordt bijgehouden in de vorm van een applicatie. Alle consumers in een groep krijgen dezelfde ID toegekend. Iedere applicatie, met verschillende consumers, zal lezen van partities die de applicatie werd toegewezen. De partities worden niet onderling verdeeld. Een consumer zal altijd een volledige partitie benutten. Als er een partitie bijkomt, dan kan er een derde consumer worden toegevoegd. Het is zinloos om meer consumers te hebben, dan partities. Bij vier consumers en drie partities zal één consumer geen werk kunnen verrichten. Het aantal partities is de bovengrens voor het parallelisme dat een systeem kan bereiken.

Consumer offset

Binnen Kafka is er een speciale topic, namelijk de *consumer offset*. Dit is de staat van de topics per consumer. De analogie met een logboek wordt gelegd. Als de consumer alle data heeft verwerkt, dan moet de offset worden verlegd voor het lezen van de eerstvolgende offset. Het moment wanneer de offset wordt gecommit speelt een rol over hoeveel keer een bericht zal worden verwerkt. Alle gelezen berichten worden bijgehouden en zo heeft het systeem weet vanaf waar een consumer berichten mag beginnen verwerken.

5.2.5 Delivery Semantics

Er zijn drie verschillende delivery semantics:

- *At most once* impliceert dat de boodschap nooit twee keer zal verwerkt worden. De kans is dat de boodschap één keer werd verwerkt, maar ook dat de consumer géén boodschap heeft verwerkt.
- *At least once* wijst erop dat de boodschap zeker één keer werd verwerkt. Meerdere keren is mogelijk. Als de consumer uitvalt voor de commit, dan wordt de boodschap opnieuw verwerkt. Afhankelijk van de applicatie, idempotent of niet, dan kan dit worden gezien als een negatief.
- *Exactly once* betekent dat het bericht noch minder noch meer dan één keer mag worden gelezen.

Broker Discovery

Het maakt niet uit met welke broker er een verbinding wordt opgestart. De client zal een verbinding leggen met eender welke broker. Welke maakt niet uit, want je bent met de cluster verbonden als je met eender welke broker een verbinding kan opstarten. Iedere broker houdt metadata bij.

5.2.6 Zookeeper

Als je Kafka wilt draaien in productie, dan wordt Zookeeper aangeraden. Zookeeper wordt gebruikt om Kafka te beheren. Enkele functies van Zookeeper:

- Zookeeper houdt de brokers bij.
- Zookeeper voert het leader-election-algoritme uit. Als er een topic online/offline wordt gehaald, dan loopt dit eerst naar Zookeeper.

Oppositie

Zookeeper is op zich ook een gedistribueerd systeem. Nu wordt er gewerkt om Kafka zonder Zookeeper te laten draaien, maar dit werkt voorlopig enkel binnen een testomgeving.

5.3 Labo

Poortnummer 19092 wordt gebruikt. De broker maakt niet uit, want de actie zal altijd werken.

Lijst tonen

```
kafka-topics --bootstrap-server kafka1:19092 --list
kafka-topics --bootstrap-server kafka2:19093 --list
kafka-topics --bootstrap-server kafka3:19094 --list

kafka-topics --bootstrap-server 127.0.0.1:9092 --topic first_topic  --describe

kafka-topics --bootstrap-server kafka1:19092 --describe --topic lecture
```

Topic aanmaken

Opmerking: de replicatiefactor kan nooit groter zijn dan het aantal brokers.

```
kafka-topics --bootstrap-server 127.0.0.1:9092
       --create
       --topic first_topic
       --partitions 3
       --replication-factor 1

kafka-topics --create --topic lecture --partitions 3 --replication-factor 3
```

Topic verwijderen

```
kafka-topics --bootstrap-server 127.0.0.1:9092
  --delete
```

5.3.1 Producer

Messages aanmaken in bestaande topic

```
kafka-console-producer --bootstrap-server kafka1:19092
            --topic lecture
            --producer-property acks=all/0/1
```

Messages aanmaken in nieuwe topic

Eerst wordt er een waarschuwing gegeven, maar het topic zal worden aangemaakt met standaardparameters. Dit is standaard één replicatiefactor en één partitie, wat vaak van de benodigde parameters afwijkt. Het is beter om zelf een nieuwe topic te maken, al kunnen de standaardparameters wél in *server.properties* worden ingesteld.

5.3.2 Consumer

De consumer zal enkel nieuwe berichten opvangen. Als er geen nieuwe berichten zijn, dan wordt er niets opgehaald en blijft de terminal leeg. Enkel als de optie *from-beginning* wordt meegegeven, dan worden alle messages sinds het aanmaken van de topic getoond. Die volgorde is niet dezelfde als waarin de berichten werden verstuurd. De ordening van de topic is afhankelijk van de gekozen partitie.

```
kafka-console-consumer --bootstrap-server kafka1:19092
            --topic lecture
            --from-beginning
```

Consumer groups

Als een consumer hetzelfde commando tweemaal doet draaien, dan zal de uitvoer zich niet herhalen. Kafka houdt de offset bij van een consumer-group. Het tweede commando toont alle gekende consumer-groups. Met het derde commando kan er worden afgeleid hoeveel consumers er op dat moment in een groep zitten. Voor iedere partitie wordt de offset getoond van de laatste offset in de partitie en de laatste offset die werd geconsumeerd. Ofwel de log-end-offset en de current-offset.

```
kafka-console-consumer --bootstrap-server kafka1:19092
             --topic lecture
             --group group-lecture
             --from-beginning

kafka-consumer-groups --bootstrap-server kafka1:19092 --list

kafka-consumer-groups --bootstrap-server kafka1:19092 --describe
            --group group-lecture
```

Offsets resetten

De offset kan op vier verschillende manieren worden aangepast.

- To-earliest zal de offset naar het begin verplaatsen
- To-datetime maakt het mogelijk om de offset te verplaatsen naar de offset die het dichtste bij een specifiek gekozen datum
- Shift-by verlegt de offset met n-aantal lijnen
- By-period is gelijkaardig aan *to-datetime*, maar hier kan je enkel met periodes werken. Bijvoorbeeld een week geleden, een dag geleden, een uur geleden enzovoort.

```
kafka-consumer-groups --bootstrap-server kafka1:19092
            --group group-lecture
            --reset-offsets --to-earliest/to-datetime/by-period/shift-by
            --execute
            --topic lecture
```

Hoofdstuk 6

Kafka Java Programming

Dependencies

Het pom-bestand heeft twee dependencies nodig: kafka-clients en slf4j-simple.

6.1 Producer

6.1.1 Een Kafka Producer ontwikkelen

Een Kafka Producer in Java ontwikkelen omvat drie fasen:

1. Producer-eigenschappen instellen.
2. Producer aanmaken.
3. Data versturen naar Kafka.
4. Optioneel: Callback toevoegen

Eigenschappen instellen

De drie belangrijkste eigenschappen zijn:

- bootstrap.servers dat het adres van Kafka aangeeft
- key.serializer
- value.serializer

```
Properties kafkaProperties = new Properties();
String bootstrapServers = "localhost:9092";

kafkaProperties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG), bootstrapServers);
kafkaProperties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, bootstrapServers);
kafkaProperties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.
    class.getName());
```

Producer aanmaken

```
KafkaProducer<String, String> producer = new KafkaProducer<>(kafkaProperties);
```

Data versturen naar Kafka

```
ProducerRecord<String, String> record = new ProducerRecord<>("topic-naam", "message-inhoud-
    message");
producer.send(record);
producer.flush();
producer.close();
```

Callback gebruiken

De callback wordt in de send-methode gebruikt om logs bij te houden.

```
producer.send(record, new Callback(){
  @Override
  public void onCompletion(RecordMetaData recordmetadata, Exception e)
  {
    if (e == null){
      logger.info("Received new metadata"
      + "Topic: " + recordmetadata.topic()
      + "Partition: " + recordmetadata.partition()
      + "Offset: " + recordmetadata.offset());
    } else {
      logger.error("Error while producing", e);
    }
  }
})
```

Keys toevoegen aan de producer

```
for (int i = 0; i < NUM_MESSAGES; i++){
  String topic = "first-topic";
  String key = "id_" + Integer.toString(i % 10);
  String message = "Hello world " + Integer.toString(i);

  ProducerRecord<String, String> record = new ProducerRecord<>(topic, key, message);
}
```

6.1.2 Producer Internals

Het proces gebeurt in vier stappen:

1. Bij het aanspreken van de send() methode worden de key en value geserializeerd met de serializers. Deze werden verkregen bij het aanmaken van de KafkaProducer.
2. Als de standaardpartitioner niet wordt gespecifieerd, dan wijst Kafka zelf een partitioner aan. De standaardpartitioner is al dan niet eerder beschreven op een *round robin* manier.
3. Topic en partitie zijn gekend. De record wordt aan de batch van records toegevoegd. Die worden dan verstuurt naar hetzelfde topic en partitie.
4. In een aparte thread worden de batches van records naar de gepaste Kafka broker gestuurd.
5. Wanneer de broker de berichten ontvant, dan stuurt die een antwoord terug. Er zijn twee mogelijke scenario's:

- Als de broker erin slaagt om het bericht uit te schrijven, dan wordt er een RecordMetadata-object teruggegeven. Die bevat de topic, partitie en record-offset binnen een partitie.
- Als de broker er niet in slaagt om een bericht uit te schrijven, dan wordt er een fout teruggegeven. De producer zal proberen om het bericht enkele keren opnieuw te sturen totdat het opgeeft en een fout teruggeeft. Het aantal keren dat er iets opnieuw wordt gestuurd is een keuze van de ontwikkelaar.

6. Om de producer af te sluiten wordt er gewerkt met *flush* en *close* methoden.

6.2 Java Consumer

6.2.1 Een Java Consumer ontwikkelen

Een Kafka Consumer in Java ontwikkelen omvat vijf fasen:

1. Properties aanmaken
2. Een consumer aanmaken
3. De consumer koppelen of abonneren aan één of meerdere topics.
4. De consumer in een lus plaatsen.
5. De consumer op een gepaste en goedgekeurde manier laten afsluiten.

Consumer properties aanmaken

Er zijn vijf eigenschappen die een Kafka Consumer moet verkrijgen:

- *bootstrap.servers* ofwel de lijst van brokers waaraan de consumer zich kan aan verbinden.
- *key.deserializer* geeft aan hoe de sleutels worden gedeserialiseerd.
- *value.deserializer* geeft aan hoe de values worden gedeserialiseerd. In een best scenario lopen beide parallel met elkaar.
- *group.id* is een String-object dat aangeeft aan welke consumer-group de consumer toebehoort.
- *auto.offset.reset* bepaalt wat er moet gedaan worden als er geen offset bij aanvang is gekozen, of wanneer de huidige offset niet meer op de server bestaat. Er zijn drie mogelijkheden indien er geen offset werd gegeven:
  - *earliest* zal de consumer van het begin laten lezen.
  - *latest* zal de consumer vanaf de laatste offset laten lezen.
  - *none* zal een foutmelding geven als er geen offset werd teruggevonden.

```
1  private static final String GROUP_ID = "lecture-app";
2  private static final String TOPIC = "online";
3
4  Properties properties = new Properties();
5  properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
6  properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.
       getName());
7  properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
       class.getName());
8
9  properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
10 properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
```

Een consumer aanmaken

```
Consumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Een consumer koppelen of abonneren aan één of meerdere topics.

```
consumer.subscribe(Collections.singleton(TOPIC));
```

De consumer in een lus plaatsen

Dit is de eerste versie van hoe een Consumer in een lus wordt geplaatst. Consumers draaien vaak oneindig lang door, waardoor er hier met een oneindige loop wordt gewerkt. De poll-methode is het grote werkpaard hier, want deze methode handelt de volgende zaken af:

- De *poll* methode doet meer dan enkel data ophalen, daarnaast vraagt het een tijdsindicatie op. Die geeft aan hoe lang een Consumer moet wachten op nieuwe records.
- De poll-methode moet regelmatig gebeuren. Een conumser die geen poll doet, zal als dood worden beschouwd door de broker.
- Als er records beschikbaar zijn in de consumer buffer, dan zal poll deze records onmiddellijk teruggeven. Zo niet worden ze even opgehouden.
- Poll geeft een lijst van records mee. Deze worden geïtereerd en iedere record wordt individueel afgehandeld.
- Bij de eerste poll zal er een GroupCoordinator-object worden teruggegeven. Die consumergroup wordt gejoined en uiteindelijk zal het worden toegekend aan een partitie.
- Een rebalance wordt behandeld in de poll-methode.
- Als het niet-gebruiken van een *poll* langer is dan de *max.poll.interval.ms*, dan wordt de consumer als dood aangezien. Dan wordt de consumer uit de consumergroep gegooid. Daarom moet de code in de loop geen onvoorspelbaar interval meemaken. Als het verwerken te lang duurt, dan kan dit onopgemerkte problemen veroorzaken. Hou extra berekeningen buiten de while-lus.

```
while(true){
  ConsumerRecords<String,String> records = consumer.poll(Duration.ofMillis(100));

  for(ConsumerRecord<String,String> record:records){
  logger.info("key: " + record.key() = ", value: " + record.value());
  logger.info("Partition: " + record.partition() + ", Offset:" + record.offset());
  }
}
```

6.3 Java Threads

Een Java-programma kan uit meerdere execution threads bestaan. Iedere thread heeft een eigen *method-call stack* en een eigen *program counter*. Een thread kan tegelijkertijd met andere threads worden uitgevoerd. Daarnaast kunnen threads ook resources met elkaar uitwisselen. Dit maakt het programmeren moeilijk, want de toegang tot de gedeelde middelen moet correct worden opgevolgd.

Voorbeelden van threads

Er worden twee voorbeelden aangehaald:

- Een Kafka producer maakt een aparte thread aan om record-batches naar de correcte Kafka-brokers door te sturen.
- Een Kafka consumer stuurt *heart-beats* in een achtergrondservice.

Shutdown hook

De Java Virtual Machine of JVM laat het uitvoeren van *shutdown hooks* toe vooraleer een applicatie wordt afgesloten. Deze functies zijn ideaal wanneer er middelen worden afgenomen. Een shutdown hook is een aangemaakte, maar nog niet begonnen, thread. In Eclipse werkt dit niet, want in Eclipse is het klikken op de rode stop-knop geen normale *shutdown* van het JVM. Dit kan enkel worden uitgetest in de Vagrant-machine door met Ctrl+C te werken.

1. Als de JVM begint met de applicatie af te sluiten, dan worden alle geregistreerde hooks in een willekeurige volgorde genoteerd.
2. Na het uitvoeren van alle hooks zal de JVM stoppen.

```
1  public class ShutdownHookExample {
2    public static void main(String[] args){
3      System.out.println("The name of the main thread is: " + Thread.currentThread().getName());
4
5      Runtime.getRuntime().addShutdownHook(new Thread() {
6        @Override
7        public void run(){
8        System.out.println("Shutdownhook is executing in thread: " + Thread.currentThread().
       getName());
9        }
10     }
11   }
12 }
```

Atomic Boolean en het sluiten van de Consumer

- Het aanmaken van een AtomicBoolean buiten de main-methode.
- Het gebruiken van de AtomicBoolean binnen de main-methode. Een consumer moet op normale wijze gesloten worden. Dit gebeurt met de *close*-methode. Hiervan kan er ook een oneindige poll-lus van worden gemaakt.

```
1  final AtomicBoolean stopRequested = new AtomicBoolean(false);
2  final Thread mainThread = Thread.currentThread();
3
4  Runtime.getRuntime().addShutdownHook(new Thread() {
5    @Override
6    public void run() {
7      stopRequested.set(true);
8
9      try {
10       mainThread.join();
11     } catch (InterruptedException e) {
12       e.printStackTrace();
13     } finally {
14       System.out.println("Done waiting for main thread");
15     }
16   }
17 });
```

```
try {
  while (!stopRequested.get()) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));

    for (ConsumerRecord<String, String> record : records) {
      System.out.println("Received record with key: " + record.key() + "and value" + record.
    value());
    }
  }
} finally {
  System.out.println("About to close the consumer.");
  consumer.close();
  System.out.println("Consumer closed");
}
```

6.4 Commits en offsets

Kafka laat consumers toe om het systeem te laten gebruiken als een tracker per partitie.

- Kafka zelf zal geen records committen, want de consumers committen een laatste bericht dat ze zelf hebben kunnen verwerken. Er wordt simpelweg vanuit gegaan dat ieder bericht, voor het laatste, ook werd verwerkt.

- Een consumer zal de offset committen wanneer er een bericht naar Kafka wordt gestuurd, die zal namelijk de *_consumer_offsets* topic gaan updaten. Merk op dat de gecommitte offset diegen is van het volgende bericht dat de consumer wilt ontvangen.

6.4.1 Verloren berichten en dubbels

Er zijn verschillende maatregelen om dit tegen te gaan.

- Kafka slaat alle berichten in een *retention period* op. Zo krijgen consumers de kans om de berichten op hun eigen tempo op te halen. Bij een stroomstoring of dergelijke kan de consumer weer oppikken waar ze waren gebleven.

- Consumer groups laten consumers toe om van eenzelfde topic parallel te lezen. Iedere consumer in een consumergroep wordt toegekend aan een unieke verzameling van partities. Ieder bericht wordt enkel naar één consumer in de consumergroep gestuurd, om zo load balancing en fouttolerantie te voorzien in een consumergroup.

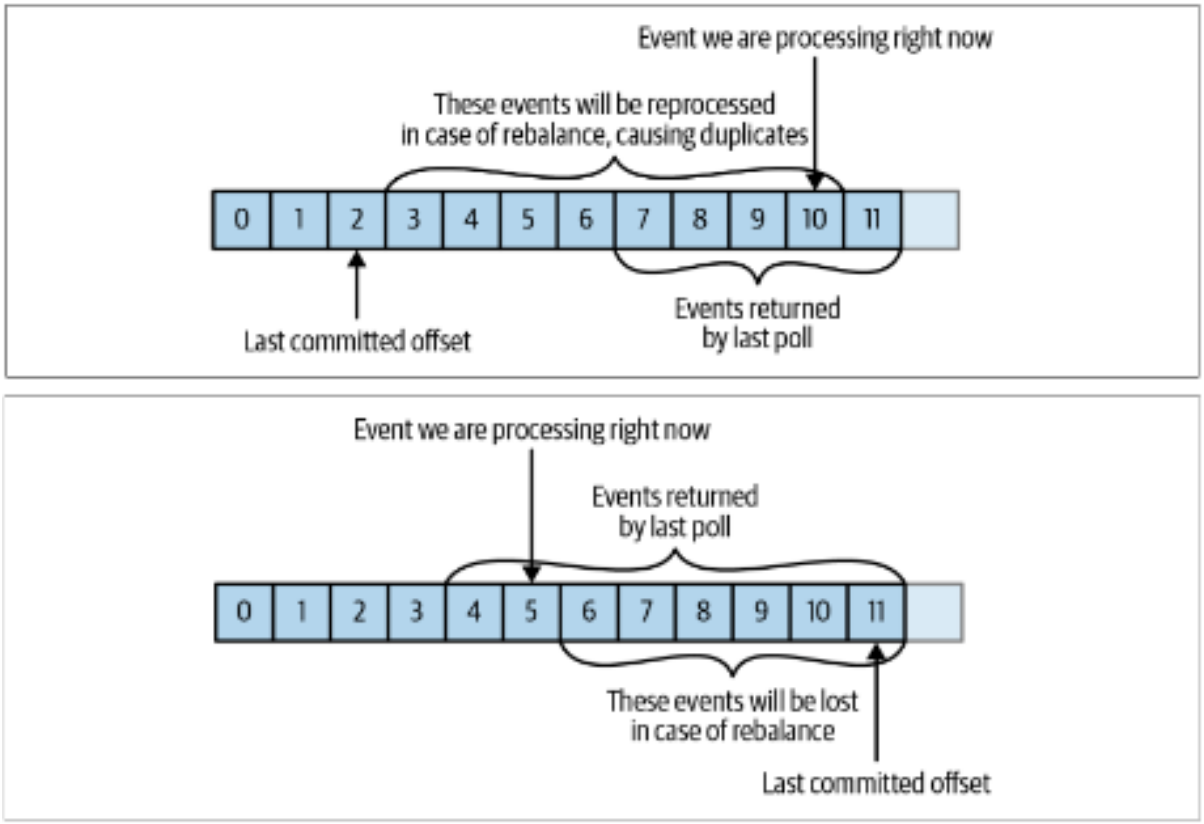
- Kafka laat producers toe om een *required acknowledgement level* aan ieder bericht toe te voegen. Dit belet hoeveel kopieën van het bericht naar Kafka moet worden geschreven vooraleer de producer *acknowledgement* krijgt.

6.4.2 API's om de offset te committen

Er zijn drie soorten commits:

- Automatisch is wanneer de *enable.auto.commit* optie op true staat.

- Synchroon en asynchroon worden gebruikt wanneer de *enable.auto.commit* op false staat.



Figuur 6.1: Bovenaan wordt, door de commit, een bericht gemist door de consumer. Onderaan haalt een consumer een bericht dubbel op.

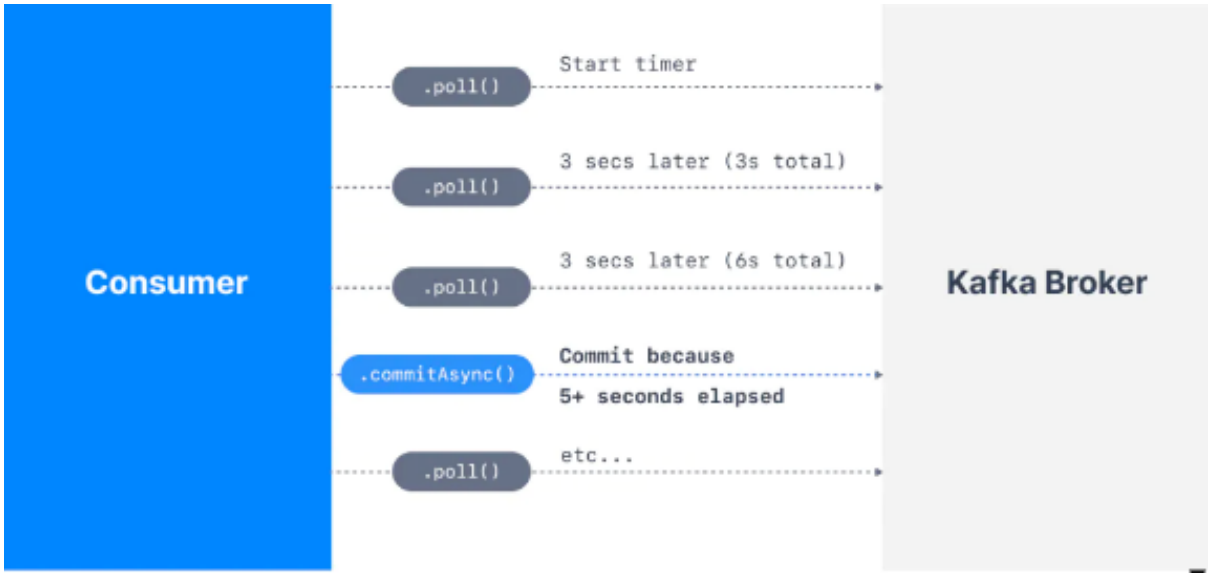
Automatische offset commit

De consumer zal iedere laatste offset van een poll elke vijf seconden, per standaard, committen naar de Kafka broker. De offset wordt teruggegeven in de laatste poll, als de consumer op tijd checkt. Het interval wordt aangepast met *auto.commit.interval.ms* en dit wordt geactiveerd met *enable.auto.commit=true*. Opmerking: Alle berichten moeten verwerkt zijn wanneer de poll werd opgeroepen. Bij het aanspreken van de *close*-methode wordt de offset ook gecommit.

Synchrone offset commits

Offsets worden enkel gecommit als de applicatie het expliciet zegt.

- Bij een *commitSync* zal de laatste offset teruggegeven worden met de poll-methode en wanneer de offset wordt gecommit.
- Deze vorm zal blijven proberen om te committen voor zolang er geen fout is waar er niet van hersteld kan worden. Als de commit faalt, dan wordt er een foutmelding gegooid.
- Een *commitSync* zal de laatste offset van *poll* teruggeven.
  - Als er wordt gecommit vooraleer de laatste berichten werden verwerkt, dan loopt het systeem het risico dat de gecommitte berichten niet verwerkt konden worden.
  - Als de commit na het verwerken van de berichten plaatsvindt, dan loopt het systeem het risico dat berichten dubbel worden verwerkt.



Figuur 6.2: Automatische offsets commits

```
properties.setProperty("enable.auto.commit","false");

try
{
  while(!stopRequested.get())
  {
    ConsumerRecords<String,String> records = consumer.poll(...);
    try
    {
      consumer.commitSync();
    } catch(CommitFailedException e) {LOGGER.error("commit failed", e);}
  }
}
  finally
{
  try
  {
    consumer.commitSync();
  } catch (CommitFailedException e){ LOGGER.error("commit failed");}
  consumer.close();
}
```

Asynchrone commit

Bij een synchrone commit wordt de applicatie geblokkeerd totdat de broker antwoordt op de commit-request. In tegenstelling tot een asynchrone commit API zal de consumer een commit-request sturen en gewoon doorwerken. De consumer werkt gewoon door, omdat de volgende commit mogelijks al succesvol is. Er wordt niets opnieuwe geprobeerd, want mogelijks wordt de vorige offset overschreven door een latere offset. De code volgt een gelijkaardige lijn als die van de synchrone commit.

Het combineren van een synchrone en een asynchrone commit

Een asynchrone commit is snel, maar het is niet geweten wanneer die succesvol is. Het doet er weining toe, want de volgende commit zal eigenlijk dienen als een *retry*. Als de consumer wilt sluiten, dan is er geen 'volgende commit', dus er wordt als sluitende commit een synchrone commit gebruikt met *close()*.

```
properties.setProperty("enable.auto.commit","false");

try
{
  while(!stopRequest.get())
  {
    ConsumerRecords<String,String> records = consumer.poll(...);
    consumer.commitAsync();
  }
}
  finally
{
  try
  {
    consumer.commitSync();
  } catch (CommitFailedException e) {LOGGER.error("commit failed",e)}
    {
      consumer.close();
    }
}
```

6.5 Apache Avro

Enkel String-objecten werden tot nu toe naar Kafka gestuurd. De data (de)serializeren gebeurde respectievelijk met StringSerializer en StringDeserializer. Het is niet aangeraden om een eigen Serializer of Deserializer te schrijven. Veel toepassingen verwachten dat gestructureerde objecten in en uit Kafka kunnen worden utigewisseld.

6.5.1 Avro

Avro is een taalonafhankelijk, serialisatieformaat dat schema's opstelt.

- Deze schema's worden vaak in JSON opgemaakt en dient om binaire bestanden te gaan serializeren. Ze zijn zelfomschrijvend waardoor de consumer, zonder toegang tot de initialisatiecode, het schema kan begrijpen.
- Avro gaat er vanuit dat er een schema aanwezig is wanneer bestanden worden gelezen en geschreven. Dit gebeurt vaak door het schema in bestanden zelf te gaan embedden, onder de vorm van metadata.
- Avro ondersteund verschillende soorten datatypen en bezit een goede ondersteuning voor *backward & forward compatibility*. Dit is ideaal voor systemen waarin data tussen meerdere systemen worden uitgewisseld.
- Avro moet aan de POM-file worden toegevoegd, zowel bij dependencies als bij plugin.

Voorbeeld Avro schema

Hieronder wordt er een skeletstructuur getoond. Dit schema houdt een type 'User' bij die drie velden heeft: *favourite number*, *favourite colour* en een *name*. Enkel het naamveld is verplicht, want de volgende twee waarden kunnen ook een null-waarde aannemen.

- Op basis van dit schema wordt er Java-code aangemaakt, maar het werkpaard hier is de Avro-compiler. De klassenaam zal hier *be.hogent.dit.tin.avro.User* noemen.

```
{
  "type": "record",
  "name": "User",
  "namespace": "be.hogent.dit.tin",
  "fields": [
```

```
  { "name": "name", "type": "string" },
  { "name": "favorite_number",  "type": ["int", "null"] },
  { "name": "favorite_color", "type": ["string", "null"] }
  ]
}
```

```
{
  "name": "Alice",
  "favorite_number": 7,
  "favorite_color": "red"
}
```

6.5.2 Een klasse aanmaken met Avro

Er zijn drie manieren:

- Constructor aanspreken met meerdere methoden.
- Alle eigenschappen in de constructor aanspreken.
- Aanmaken met de builder.

```
User user1 = new User();
user1.setName("Alice");
user1.setFavoriteNumber(7);
user1.setColor("red");

User user2 = new User("Dylan", 25, "violet");

User user3 = User.newBuilder().setName("Stijn")
                .setFavouriteColor(null)
                .setFavouriteNumber(314)
                .build();
```

6.5.3 De klasse serializeren

De drie gebruikers worden vervolgens op deze manier naar de disk geserializeerd.

1. Alleereerst wordt er een DatumWriter-object aangemaakt. Die zal de Java-objecten omzetten naar een in-memory serialized format. De SpecificDatumWriter-klasse wordt gebruikt bij aangemaakte klassen, zodat het schema later kan worden opgehaald.
2. Vervolgens wordt er een DataFileWriter aangemaakt. Die zal de geserializeerde records en het schema naar een specifiek bestand uitschrijven.
3. Met append worden de gebruikers in een bestand geschreven.
4. Het dataFileWriter-object wordt, eenmaal klaar, gesloten.

```
DatumWriter<User> userDatumWriter = new SpecificDatumWriter<>(User.class);
DataFileWriter<User> dataFileWriter = new DataFileWriter<>(userDatumWriter);

dataFileWriter.create(user1.getSchema(), new File("users.avro"));
dataFileWriter.append(user1);
dataFileWriter.append(user2);
dataFileWriter.append(user3);
dataFileWriter.close();
```

6.5.4 Een klasse deserializeren.

1. Eerst wordt er een DatumReader-object aangemaakt. In dit geval een SpecificDatumReader dat in-memory geserializeerde items zal omzetten naar instanties van de aangemaakte klassen. In dit geval is de klasse User.
2. De DatumReader en het bestand dat moet gelezen worden, wordt doorgegeven aan de DataFileReader. Dit object zal zowel het schema als de data van de disk lezen.
3. Itereer door de gebruikers in het bestand.
4. Het User-object wordt opnieuw gebruikt om zo zuinig mogelijk deze applicatie af te werken. Op deze manier hebben we minder *object-allocation*.

```
File file = new File("users.avro");
DatumReader<User> userDatumReader = new SpecificDatumReader<>(User.class);
DataFileReader<User> dataFileReader = new DataFileReader<>(file, userDatumReader);
User user = null;
while(dataFileReader.hasNext())
{
  user = dataFileReader.next(user);
  System.out.println(user);
}
dataFileReader.close();
```

Het aantal gebruikte schema's

Er worden twee schema's gebruikt. Het schema voor de writer is in het bestand en is nodig om te weten in welke volgorde de gebruikers werden uitgeschreven. Daarnaast is er een reader schema dat aangeeft welke velden er in een bestand zitten. De twee schema's verschillen op de manier hoe ze worden opgelost.

6.5.5 Een klasse deserializeren zonder het aangemaakte User-object

Data in Avro wordt altij bij het bijhorende schema opgeslaan. Daarom kan een geserializeerde item altijd worden gelezen, zonder dat het schema vooraf is gekend. Dit laat ons toe om zowel serializatie als deserializatie uit te voeren zonder het aangemaakte User-object. Een herwerking van het vorige voorbeeld ziet er als volgt uit:

Het aanmaken van de Users

1. Eerst wordt er een Parser-object gebruikt om het schema in te lezen en een Schema-object aan te maken.
2. Er is geen weet van hoe het schema eruitziet, dus er wordt gewerkt met GenericRecords. Dit object gebruikt het schema om te bevestigen of alles wel geldig is.
3. Als er een niet-bestaand veld wordt ingegeven, zoals "favourite_animal", dan wordt er een *AvroRuntimeException* teruggegeven in de compiler.

```
public class AvroVoorbeeldZonderUserObject
{
  private static final String SCHEMA_FILE_PATH = "src/main/resources/avro/Users.avsc";

  public static void main(String[] args) throws IOException
  {
    Schema schema = new Schema.Parser().parse(
      new File(SCHEMA_FILE_PATH)
    );

    GenericRecord user1 = new GenericData.Record(schema);
    user1.put("name","Alice");
```

```
13      user1.put("favorite_number", 7);
14      user1.put("favorite_color", "red");
15
16      GenericRecord user2 = new GenericData.Record(schema);
17      user2.put("name","Stijn");
18    }
19  }
```

Gebruikers uitschrijven

Gebruikers uitschrijven gelijkt zeer hard op de vorige uitwerking, maar enkel hier wordt er weer gewerkt met een generieke versie van het oorspronkelijke object.

- Het schema wordt gebruikt om te achterhalen hoe de GenericRecord moet worden geschreven.
- Daarnaast wordt het schema ook gebruikt om alle non-nullable velden te detecteren.

```
File file = new File("users.avro");
DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<>(schema);
DataFileWriter <GenericRecord> dataFileWriter = new DataFileWriter<>(datumWriter);
dataFileWriter.create(schema, file);
dataFileWriter.append(user1);
dataFileWriter.append(user2);
dataFileWriter.close();
```

Gebruikers inlezen

De gouden regel geld ook bij het inlezen. Alle objecten worden aangepast naar een generieke versie van datzelfde object.

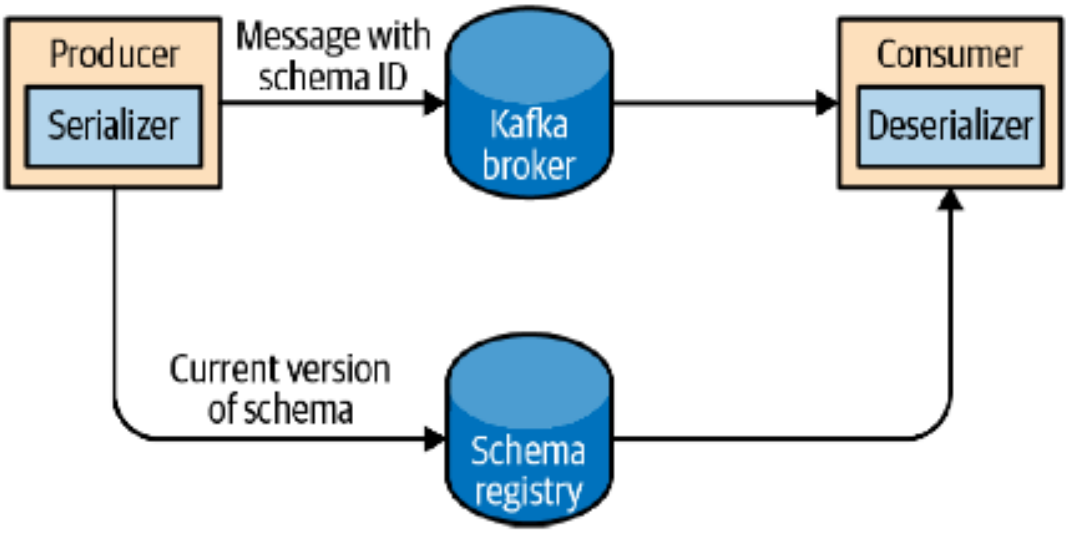
```
DatumReader<GenericRecord> datumReader = new GenericDatumReader<>(schema);
DataFileReader<GenericRecord> dataFileReader = new DataFileReader<>(file,datumReader);
GenericRecord user = null;
while(dataFileReader.hasNext())
{
  user = dataFileReader.next(user);
  System.out.println(user);
}
```

Het uitgeschreven bestand

De applicatie geeft de volgende output terug.

```
"
{"name: ""Alyssa, ""favorite_number: 256, ""favorite_color: null"}{"name: ""Ben, ""favorite_number:
    7, ""favorite_color: ""red}
```

Het uitgeschreven bestand is niet met het blote oog begrijpbaar. Daarvoor moeten we met *format-hex* het bestand bekijken. Daar merken we op dat het schema helemaal bovenaan het bestand zichtbaar is. Het schema maakt deel uit van het uitgeschreven bestand. Dit is OK wanneer het bestand groot is, want in dat geval bevat het bestand veel records ofwel veel data. De berichten in Kafka zijn echter van beperkte grootte, dus het schema in ieder bericht plaatsen zou veel overhead veroorzaken.



Figuur 6.3: Werking van een schema-register.

6.6 Het schema-register

Het volledige schema in ieder bericht opslaan zou leiden tot veel overhead. Weglaten kan ook niet, want Avro heeft het schema nodig om een record te kunnen lezen. De oplossing is om het schema elders op te slaan.

6.6.1 Werking van een schema-register

Bij alle Kafka-records worden enkel het ID van het schema opgeslaan. De consumers halen een record op en vragen dan ook aan het register welk schema er nodig is om een record te deserializern. De serializer en deserializer staat in voor het opslaan en ophalen van het schema. De consumers en brokers halen niets direct op vanuit het schema register.

6.6.2 Kafka-producer met Avro

In de volgende code worden er drie nieuwe zaken aan het properties-bestand toegevoegd. Merk op dat er voor de volgende code extra dependencies en repositories moeten worden toegevoegd in de POM-file.

- Er wordt een String als sleutel gebruikt, maar toch wordt er gevraagd aan Avro om de sleutel te serializeren. De twee config-klassen worden gebruikt om de keys en values van de berichten te serializeren.
- Daarnaast moet er ook een url richting het schema-register worden toegevoegd.

```
Properties props = new Properties();

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, io.confluent.kafka.serializers.
    KafkaAvroSerializer.class.getName());

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, io.confluent.kafka.serializers.
    KafkaAvroSerializer.class.getName());

props.put("schema.registry.url","http://localhost:8081");

Producer<String, be.hogent.dit.tin.avro.User> producer = new KafkaProducer<>(props);

List<User> users = new ArrayList<>();

...

for (User user : users)
{
  ProducerRecord<String,User> record = new ProducerRecord<>(TOPIC, user.getName().toString(),
    user);
  producer.send(record);
}

producer.flush();
producer.close();
```

6.6.3 Kafka Consumer met Avro

```
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, io.confluent.kafka.serializers.
    KafkaAvroDeserializer.class.getName());

props.put("schema.registry.url", "http://localhost:8081");
props.put("specific.avro.reader", "true");

Consumer<String, User> consumer = new KafkaConsumer<>(props);

ConsumerRecords<String, User> records = consumer.poll(Duration.ofMillis(500));

for(ConsumerRecord<String,User>record:records)
{
  System.out.println("Received record with key [" + record.key() + "]");
  System.out.println("Value of this record is " + record.value());
  System.out.println("The type of this value is " + record.value().getClass().getName());
}
```