Big Data 2019

Portfolio assignment - part one

Mette Strand Hornnes

Innholdsfortegnelse

1. Primary data source

Jeg har valgt å velge Oslo og omegn.

Jeg lastet tidlig ned en osm-fil som viste seg å bli altfor stor. Senere i prosjektet endte jeg derfor opp med å hente en ny, med en mindre del av Oslo.

Informasjon om data som er hentet:

Name: oslo

Coordinates: 10.73,59.918 x 10.759,59.933

Script URL:

https://extract.bbbike.org?sw_lng=10.73&sw_lat=59.918&ne_lng=10.759&ne_lat=59.933&format=osm.gz&city=oslo&lang=en

Square kilometre: 2

Granularity: 100 (1.1 cm)

Format: osm.gz

File size: 0.6 MB

SHA256 checksum:

7093b53dcaacacf9e37c8df93ffdd20e7040a57d3aaa8458aed808caf3a9ea33

MD5 checksum: 49bde295af7b6c6af451857c9d5c1c46

Last planet.osm database update: Thu Sep 26 16:22:32 2019 UTC

License: OpenStreetMap License

2. Assignment parts

2.1 Method for importing initial data into HDFS

IMPORTERE DATA

Det er flere måter å importere data inn til HDFS. For å lese/skrive data til HDFS kan man benytte seg av verktøy som Sqoop, Flume, Hadoop Distcp, Bruke HUE, File browser eller Benytte et JAVA API. Jeg har valgt å gjøre dette mer manuelt ved å benytte meg av Hadoop File system sine kommandoer.
Jeg utførte følgende:

1. Starter Hadoop: `start-all.sh`
2. Sjekker at Hadoop er oppe og kjører: `jps`
3. Legger fila inn i filsystemet: `hdfs dfs -put navnpåfil.osm /navnpåfil.osm`
    a. Eventuelt `hadoop fs -copyFromLocal navnPåfil.som /navnpåfil.osm`
4. Ser at filen er lagt i hadoop sitt filsystem: `hdfs dfs -ls`

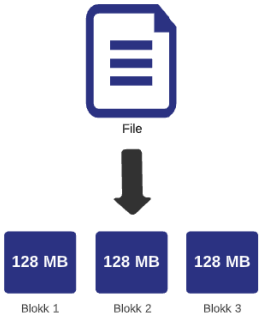
HVA SKJER I BAKGRUNNEN:

Litt HDFS-arkitektur
For å bedre kunne forklare hva som skjer ønsker jeg først å forklare litt om de ulike komponentene i Hadoop Distributed File System (HDFS).

Med HDFS bli data distribuert over flere ulike maskiner. Alle disse maskinene er koblet sammen og utgjør sammen et såkalt *Hadoop Cluster*. Hvert cluster består av, blant annet, en NameNode som har en rolle innenfor YARN-rammeverket og dermed håndterer filsystemet og har oversikt over ledige DataNoder, en Secondary nameNode som fungerer som avlastning og backup for NameNoden, DataNoder hvor selve dataen blir lagret og en ClientNode som styrer kommunikasjonen mellom NameNoden og DataNodene (edureka!, 2017, 12:00).

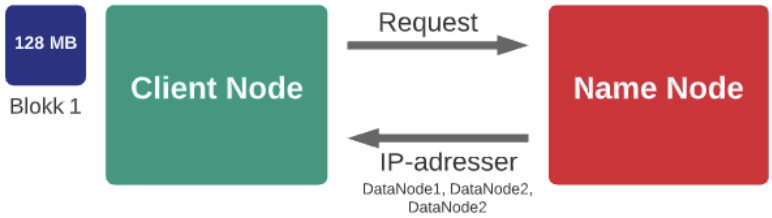
Skrive data til HDFS

Når filen skal skrives til HDFS deles denne først opp i blokker på 128 MB (Merk at størrelsen på disse blokkene kan variere noe).



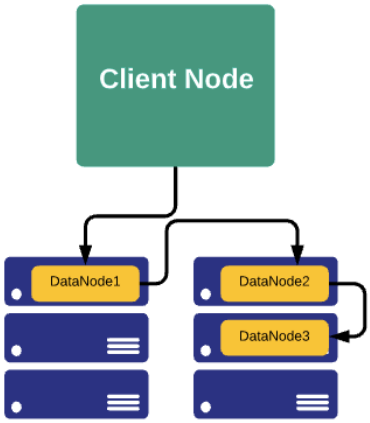
Deretter skjer følgende steg samtidig for hvert block:

1. Pipeline Setup
   I første steg kontakter ClientNoden Namenoden og gir beskjed om at den har en Block den trenger å lagre (Sender en request). NameNoden responderer med å returnere en liste med IP-adresser til 3 ulike DataNoder, da standard for antall kopier av en Block er 3.



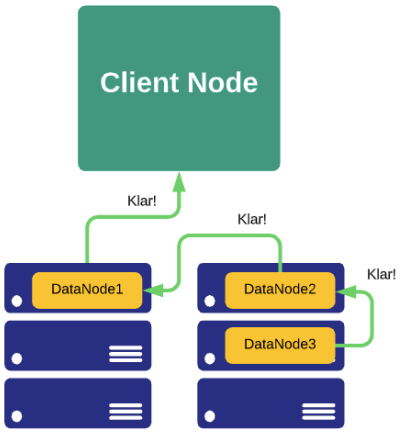
   Årsaken til disse kopiene er for å hindre at data blir tapt om en datanode eventuelt skulle sluttet å fungere.
   Den første replikaen av blocken vil lagres på den lokale maskinen, den andre på en annen rack (som består av flere DataNoder) og den tredje skal lagres på en annen DataNode på denne racken. Det vil maks være en duplikat per DataNode og kun to per rack.

   Når ClientNoden får denne listen returnert vil den kontakte den første DataNoden for å sikre at denne er klar til å ta imot data. Samtidig gir den beskjed om å kontakte den andre DataNoden, som igjen kontakter denne tredje for å sørge for at denne er klar til å ta imot data.



Skulle en av disse ikke gi et klarsignal, vil ClientNoden gå tilbake til NameNoden og be om IP-adresse til en ny DataNode. NameNode sjekker så hvilke DataNoder som er ledige og velger deretter ut en av de som har sendt såkalte «Heartbeats»-signaler om at den er ok, for å returnere tilbake IP-adressen til denne tilbake ClienNoden.

Når alle DataNodene 3 gir et OK-signal er pipelinen satt opp og hdfs er nå klar til å lagre dataen.



2. Lagring av Data

   På samme måte som kontrollen av DataNodene skjedde vil også lagringen foregå. ClientNoden kontakter den første for å lagre, den første kontakter den andre og den andre kontakter den tredje.

3. Bekreftelse på lagring - Acknowledge

   Når den siste DataNoden har lagret den tredje kopien vil den sende en såkalt ACK-melding (Acknowledge) til den forrige DataNoden for å gi beskjed om lagringen er gjort uten problemer. Slik fortsetter prosessen bakover til ACK-meldingen har nådd ClientNoden og denne får beskjed om at alle 3 Blockene er lagret.

Denne beskjeden videreføres til NameNoden slik at denne kan oppdatere sin informasjon om de ulike DataNodene (edureka!, 2017, 31:32).

MIN SETUP VS CLUSTER VS DUPLIKATER AV FILEN

Med et cluster vil data bli distribuert utover flere maskiner. Dette vil både bedre ytelsen, samt det vil bære enkelere å skalere opp eller ned ved å kun legge til eller trekke fra maskiner til clusteret. Da flere maskiner kjører minsker også dette sjansen for feil ettersom andre kan ta over om en av de skulle feile. Setup med cluster vil derfor være ganske fleksibelt og gi bedre ytelse.

Om HDFS er konfigurert til å bruke flere duplikater (replicas) av filen vil alt være distribuert utover på en maskin, og ulike deler av filen vil bli brukt for å parallell prosessering. Med dette hindrer vi også såkalte «single point failures» da data vil være tilgjengelig på andre noder selv om den skulle feile et sted.
Det fine med Hadoop er at dette ikke er noe vi egentlig trenger å tenke så mye på hvordan den kjører da dette blir abstrahert for oss. Men som sagt blir hovedforskjellen hvor mye data vi faktisk kan håndtere (Udemy, 2019, 00:00).

2.2 Method for updating the data

OPPDATERE DATA I HDFS

Å oppdatere data i HDFS kan være utforende da dette filsystemet er «immutable», altså uforanderlig. Filer som er lagt til filsystemet kan ikke endres. Samtidig finnes det heller ingen oppdaterings-kommandoer man enkelt kan benytte seg av. Likevel er det noen metoder man kan ta i bruk for å få gjort endringer. Jeg vil forklare fire ulike metoder. Hybrid Update, HBase Dimension Store og Merge and Compact Update (med «Good Enough» Update). De alle baserer seg på det grunnleggende som må til som er å gi filen ett nytt navn slik at det ikke vil skape problemer om filen blir brukt i andre jobber.

Hybrid Update

Hybrid Update-metoden benytter ETL (Extract Transform and Load) og SQL-programmering. Ved hjelp av et verktøt kalt Sqoop vil all data bli kopiert med jevne

mellomrom. Dette er ofte den første som blir tatt i bruk da den er såpass enkel å implementere.

Bakdeler med Hybrid Update er at denne metoden ikke er skalerbar og dermed ikke særlig effektiv ved store datamengder, da all alle record-ene må hentes hver gang en oppdateirng skal gjennomføres.

HBase Dimension Store

HBase er en NoSql-database som kjører på hadoop og har med dette innebygde verktøy for oppdatering av data. Denne metoden fungerer best om det kun er en eller et fåtall rader som skal modifiseres. Kreves det at hele tabellen må gås igjennom er HBase lite effektiv.

Merge And Compact Update

I motsetning til metodene over benytter denne seg av algoritmer for å oppdatere data. Her brukes altså innen annen spesifikk teknologi. Denne algoritmen kan implementeres med blandt annet både Java MapReduce og Spark. Algoritmen går igjennom seks steg:

1. **Kopier data:** En full kopi av dataen lagres i HDFS (Denne kopien blir ofte kalt «Masterdataen»).
2. **Laste inn ny data:** Den nye oppdaterte dataen blir deretter lastet inn i HDFS (Ofte kalt «Deltadata»)
3. **Flett data sammen:** Neste steg flettes den kopierte og oppdaterte sammen etter nøkkelfeltet.
4. **Plukk ut data:** Når man har kommet til denne delen vil man ha flere recorder for hver nøkkel. I denne delen vil algoritmen sørge for at det kun er en per nøkkel. Hvilken record som bli spart velges oftes utifra timestamp (Den som er sist endret).
5. **Skrive data midlertidig:** Resultatet av forrige steg vil nå bli skrevet til en midlertidig output (Da de fleste Hadoop-jobber ikke kan overskrive andre mapper).
6. **Overskriv masterdataen:** I dette siste steget vil dataen som ble midlertidig lagret i forrige flyttes til der masterdataen er lagret, og på den måten overskrive denne dataen.

Bakdelen med denne metoden er at man må lese og prosessere all data hver gang noe skal oppdateres, noe som er lite effektivt. Dette kan løses med noe kalt «Good Enough» Update

Denne metoden baserer seg på at noen data har større sjanse for å måtte bli oppdatert enn andre. Ved å flytte disse dataene (ofte de som sist ble oppdatert) i til egne sub-folders, vil Merge and Compact Update-metoden kun utføres på dataene som har størst sjanse for å skulle endres. På den måten slipper man å lese og prosessere alle dataene, og også ende opp med overflødige recorder (Deptula, 2015)

2.3 MapReduce and Spark

2.3.1 Calculate simple metrics

For disse oppgavene vil det beskrives hva som skjer I henholdsvis MapReduce og Spark. Jeg har valgt å måle ytelse på antall linjer med kode, samt kjøretid. Dette vil stå under hver beskrivelse i alle oppgavene jeg har klart å løse. All programmering løses i fil med angitt navn under helholdsvis Java- eler Scalamappe i innlveringen

I oppgave 3.3.3 vil en sammenligning av ytelsen bli gjort.

Explain what happens "behind the scenes" when you submit your program and during execution. – Er dette beskrevet godt nok?

Explain what would happen behind the scenes if the programs were to be run on a large cluster consisting of many nodes and using a much larger extract of OSM data. Will your programs still produce the same results? If the results differ, are they still correct? - Usikker på hvordan forklare dette

2.3.1.1 How many buildings is it in the extract you selected?

Løses i filen: `One_BuildingCount`

MapReduce:
I denne oppgaven vil mapperen gå igjennom hele osm-filen og dele den opp i tokens. For hvert ord som passer med k=»building» vil sammen med et ettall sendes til reduceren. Reduceren legge sammen alle disse tallene og printe ut resultatet.

Ytelse:
Antall kodelinjer i mapper og reducer: ca. 14
Kjøretid: 4.398 s

Spark:
I spark leses hele filen inn og splittes deretter i ulike ord (deles på mellomrom). For hvert av dissse ordne vil det sjekkes det er likt «K=»building». Om dette er tilfelle det samme med «true» legges til et ettall (Key-value par). Om ikke vil det settes en false. Deretter vil tallene legges sammen basert på om det er true eller false og vi få printet ut antall som er en building og antall som ikke er det. .

Ytelse:
Antall kodelinjer (uten initaliseringen): ca. 5
Kjøretid (uten initaliseringen) : 2.420 s

2.3.1.2 How many addr:street tags exist for each street?

Løses i filen: `Two_AddrStreetTagsPerStreet`

MapReduce:
Her settes startTag til å være <osm (Altså root-taggen til filen)**.**
I mapperen hentes alle «taggene» ut og sjekkes om den har attributten «add:street». Om det er tilfelle vil verdien (altså navnet på gaten), sammen med et ettall sendes til reduceren.
På samme måte som i oppgaven i over vil reudceren her legge sammen disse tallene for hvert gatenavn og printe resultatet.

Ytelse:
Antall kodelinjer i mapper og reducer: ca. 20
Kjøretid: 7.769 s

Spark:
I denne oppgaven (og de videre) vil først initialisering av Spark skje. Her settes blandt annet hva som er root- og radTag. I denne oppgaven bli radTaggen satt til å være «tag» slik disse sendes inn hver for seg.
Først vil det hentes ut attrobuttene k og v (key og value for nåværende tag) som deretter filtreres etter om nøkkelen er «add:street».

Disse add:street taggene vil nå grupperes etter verdien (altså gatenavnet) og telles, for å deretter printes til console.

Ytelse:
Antall kodelinjer (uten initaliseringen): ca. 3
Kjøretid (uten initaliseringen): 21.969 s

2.3.1.3 Which object in the extract has been updated the most times, and what object is that?

Løses i filen: `Three_ObjectMostUpdated`

MapReduce:
Også i denne oppgaven settes starttag til å være <osm.
I mapperen hentes alle node-tagger ut og gås igjennom. For hver av disse nodetaggene vil versjons- og idnummer hentes ut. Om dette versjonsnummere er større enn tidligere versjonsnummer som er hentet ut, vil denne tas vare på. Man sitter da igjen den noden med høyest versjonsnummer. Dette sendes så til reduceren.
Jeg ser at reduceren egentlig vil bli noe overflødig i denne oppgaven da jeg tar meg av all sjekk i mapperen. Jeg er noe usikker på hvordan dette burde gjøres, men jeg tenker at det ville vært mer optimalisert om de ulike node-taggene ville blitt sendt til reduceren og sjekk av versjonsnummer ville blitt gjort der.

Jeg ser at dette oppsettet i er optimalisert, da jeg får en ekstremt lang kjøretid. Jeg bør her heller sette startTag til å være node slik at disse kan kjøres til hver sin mapper og deretter settes sammen igjen i reducer. Dette vil forbedre ytelse mye. Da jeg ikke har fått det til fungere slik jeg ønsket har jeg latt koden min stå slik den nå er inntil videre.

Ytelse:
Antall kodelinjer i mapper og reducer: ca. 21
Kjøretid: 89.876 s

Spark:
I denne oppgaven vil radtag settes til å være «node», slik at disse sendes inn hver for seg (og kan eventuelt kjøres på ulike noder om dette er tilgjengelig).
For hver node hentes det ut id- og vesjonsattributten (samt settes alias for disse for enklere referering videre).

Videre vil sorteres denne tabellen synkende, slik at noden med det høyeste versjonsnummeren vil ligge øverst. Helt sist vises kun denne, ved å spesifisere at kun første rad skal vises.

Ytelse:
Antall kodelinjer (uten initaliseringen): ca. 3
Kjøretid (uten initaliseringen): 21.980 s

2.3.1.4 Which 20 highways contains the most nodes?

Løses i filen: `Four_20TopHighWayNodes`

MapReduce: - OBS! Mangler sortering i MapReduce
Istedenfor å sette startTag til <osm som i de tidligere oppgavene, settes nå denne til <way. Når dette gjøres vil way-taggene sendes til flere mappere (tidligere har hele xml-filen blitt sendt til samme). Fordelen med å dele opp slik jeg nå har gjort er at de nå vil kunne kjøre på flere noder om dette hadde vært tilgjengelig.
Mapperen vil nå gå igjennom way-taggen den er «tilsendt», samt dens barnetagger. For hver «barnetag» som er en nd-tag vil det plusses på en på telleren som har oversikt over dette.Deretter sjekkes det om en av barnetaggene som gås igjennom er en highway. Om dette er tilfelle vil id-en til denne wayen tas vare på og vil sammen med telleren sendes til reduceren.
Reduceren vil her for hver way legge sammen resultatet og printe dette.

Jeg har i denne oppgaven ikke fått til å sortere, samt vise kun de 20 høyeste. Jeg har vurdert å benytte meg av en hashMap hvor jeg legger alle wayene sammen med antall noder. For å deretter kunne sortere denne synkende og hente ut de 20 første. Jeg ser at dette kan være noe ueffektivt og har derfor enn så lenge ikke fått løst denne oppgaven helt.

Ytelse:
Antall kodelinjer i mapper og reducer: ca. 27
Kjøretid: 8.478 s

Spark:
I denne Sparkoppgaven settes radTaggen til å være way. For hver av disse wayene vil det hentes ut id-attributt, samt alle node-taggene for denne (med explode). Deretter filtreres denne på wayene som har en tag med nøkkel som er «highway». En ny kolonne legges så til denne

tabellen hvor antall noder for nåværende way telles opp. Tabellen sorteres så synkende på antall noder, før id-en og antallnoder plukkes ut (Da det er kun dette jeg ønsker å vise i tabellen) og de 20 øverste radene vises.

Ytelse:
Antall kodelinjer (uten initaliseringen): ca. 6
Kjøretid (uten initaliseringen): 12.127 s

2.3.1.5 What is the average number of nodes used to form the building ways in the extract?

Løses i filen: `Five_AverageNumOfNodesBuilding`

MapReduce:
I likhet med oppgaven over vil også denne sette startTag til <way slik at de sendes til ulike mappere.
Mapperen vil først hente ut antall nd-tagger for nåværend way. Deretter itereres det igjennom alle taggene for denne wayen og sjekker om en av disse har attributer som tilsier at det er en building. Om dette er tilfelle vil tekst og antall noder som er telt opp for denne wayen sendes til reduceren.
I reduceren vil nå legge sammen antall wayer (som er en buildingway), samt legge sammen verdien som blir sendt med. Med disse to tallene kan det nå enkelt finne gjennomsnittet som da printes ut.

Ytelse:
Antall kodelinjer i mapper og reducer: ca. 20
Kjøretid: 7.466 s

Spark:
I likhet med Sparkoppgaven før settes radTaggen til å være way. Her hentes nå id-attributt og alle «tag-tagger» ut med explode. Tabellen filtreres så på de wayene som har en tag med nøkkelattributt «building». Det legges så til en kolonne som vises summen av antall noder i denne wayen. Helt sist hentes denne kolonnen ut og gjennomsnittet av alle radene regnes ut og vises.

Ytelse:
Antall kodelinjer (uten initaliseringen): ca. 4
Kjøretid (uten initaliseringen): 14.536 s

2.3.1.6 How many ways of types ”highway=path”, ”highway=service”, ”high- way=road”, ”highway=unclassified” contains a node with the tag ”bar- rier=lift gate”

Løses i filen: `Six_NumOfLiftGate`

MapReduce:

Starttag settes til å være way, slik at de igjen sendes til ulike mappere.
Mapperen vil hente ut alle taggene for denne wayen som det deretter itereres igjennom. For hver tag vil det sjekkes den har nøkkelattributt «barrier» med verdi «lift_gate». Om dette er tilfelle setter jeg en boolean-variabel til true for å huske på dette. Det vil også sjekkes om den har attributt som tilsier at dette wayen er til highway. Om den er det vil den sendes til en ny sjekk for finne ut om den er av riktig type highway. Altså enten path, service, road eller unclassified (Denne sjekken er lagt ut i en egen metode for mer oversikt).
Om også denne sjekken returnerer true vil denne, sammen med et ettall sendes til reduceren om det tidligere har vist seg at den også er en liftgate. Om det ikke er tilfelle vil den sendes til reduceren med 0.
Reduceren vil så legge sammen alle verdiene som kommer inn for hver highway og deretter. printe dette.

Ytelse:

Antall kodelinjer i mapper og reducer: ca. 31
Kjøretid: 10.832 s

Spark:

RadTag settes til way. Id-attributt, samt alle tag-tagger for hver way hentes ut. Tabellen filtreres så på alle wayene med highway av riktig type (path, service, road eller unclassified). I neste steg hentes id-attributt og tag-tagger ut igjen til en ny tabell som deretter filtreres på de wayene som har en nøkkelattributt «barrier» med verdi «lift_gate».
Deretter vil de to tabellene man nå har kobles sammen (joines). Tabellene kobles på id-ene, slik at man nå ender opp med en tabell som inndeholder id til wayen, tag som er highway av riktig type og tag med barrier som er lift_gate.
Helt sist vil nå tabellen grupperes etter de ulike higway-typene og telles opp.
Når tabellen vises vil vi nå se hvor mange av de ulike highwayene som inneholder barrier=liftgate.

Ytelse:

Antall kodelinjer (uten initaliseringen): ca. 9

Kjøretid (uten initaliseringen): 19.711 s

2.3.1.7 Oppgaver som ikke er besvart

7. Which 15 highways contains the most number of traffic calming=hump?
8. Which building has the largest latitudinal extent? (biggest difference between the northernmost and southernmost node)

9 What is the longest way with tag highway?

2.3.2 Creative part

2.3.2.1 Informasjon om valgt data

Jeg har valgt å benytte oslobysykkel som ekstra data, jeg skal hente data fra og spørre spørsmål til. Dataen som her hentes inneholder turhistorikk fra Oslo Bysykkel. I denne CSV-filen ligger følgende informasjon for alle turer sålangt i november:
Tidspunktet turen startet, tidspunktet turen ble avsluttet, lengde på tur i sekunder, unik ID for startstasjon, navn på startstasjon, beskrivelse av hvor startstasjon er plassert, breddegrad for startstasjon, lengdegrad for startstasjon, unik ID for endestasjon, navn på endestasjon, beskrivelse av hvor endestasjon er plassert, breddegrad for endestasjon og lengdegrad for endestasjoninformasjonen

2.3.2.2 *Spørsmål 1 - Hvilke 10 steder i extracten er bysykkel mest populært å starte turen fra?*

Løses i filen: `CreativeOne_topTenCityCyclePlacesInExtract`

MapReduce: Oppgaven er foreløpig kun implementert med Scala

Spark:

Til lesingen av osm-filen setter jeg nå radTaggen til å være «tag», da det er disse jeg skal gå igjennom. Etter initialiseingen starter programmet med å hente ut de ulike stedene som finnes i ekstrakten jeg har valgt. Deretter oppretter jeg en ny tabellen som henter ut bysykkelstasjonsnavnene med en opptelling av antall ganger det er startet en tur fra denne. Etter dette joines disse to tabellene på sted og bysykkelstasjonsnavn, det plukkes ut hva som skal vises og sorterer tabellen etter antall ganger turen har startet herifra. Helt til slutt vises resultatet med de 10 første radene.

Ytelse:

Antall kodelinjer (uten initaliseringen): ca. 5

Kjøretid (uten initaliseringen): 37.200 s

2.3.2.3 Spørsmål 2 - Hvor finnes bysykkelstativer i ekstraktet? (Obs, kun de som er blitt brukt så langt i november)

Løses i filen: `CreativeTwo_whereAreTheCyclingStationsInTheExtract`

MapReduce: Oppgaven er foreløpig kun implementert med Scala

Spark:

Dette programmet er ganske lik som over. RadTag settes også her til å være «tag». Og likt som over starter programmet med å hente ut de ulike stedene som finnes i ekstrakten (oslo.osm) jeg har valgt. Deretter opprettes det en tabell hvor bysykkelstasjonsnavn og beskrivelse på hvor denne ligger hentes ut.
Etter dette kobles disse to tabellene sammen slik at jeg ender opp med en tabell over navn og beskrivelse på kun de stedene som finnes i ekstraktet.

Ytelse:

Antall kodelinjer (uten initaliseringen): ca. 5

Kjøretid (uten initaliseringen): 27.297 s

2.3.2.4 Why MapReduce or Spark is the correct or not the correct tool to answer them?

Foreløpig ikke besvart

2.3.3 Compare the performance of the MapReduce implementation and the Spark implementation

MapReduce og Spark har noen felles trekk da de begge er metoder å håndtere store datamengder på, er skalerbare og kan benytte opptil 1000 noder i et cluster (Xplenty, 2019), er det store forskjeller på de. De begge har feilhåndtering, men der MapReduce benytter replicas av data som fordeles utover, benytter Spark Resilient Distributed datasets (RDD) eller Dataframes (DF) (Xplenty, 2019). Både RDD og DF er statiske sett med objekter som fordeles ut på flere maskiner, men de skiller seg fra hverandre ved at DF er organisert inn i navngitte kolonner.

I oppgavene besvart i 3.3.1 og 3.3.2 måler jeg ytelse på henholdsvis MapReduce og Spark-programmene ved å se på antall linjer med kode, samt kjøretid. Der kommer det frem at det er nokså store forskjeller for MapReduce og Spark når det gjelder prosesseringshastighet og hvor lett er å bruke som er det jeg velger å fokusere på i denne sammenligningen.

Prosesseringshastighet:

Spark prosesserer data i RAM (in-memory caching), mens MapReduce skriver til disk etter hver mapping eller reducing-jobb. Dette gjør at Spark utkonkurrerer MapReduce når det kommer til ytelse (Xplenty, 2019). Ofte krever arbeidet at samme data må prosesseres flere ganger, ved at dette med Spark er tilgjengelig i minnet gjør Spark sin ytelse mye bedre (IntelliPaat, 2019). Faktisk kjører programmene opp til 100 ganger raskerene enn MapReduce i minnet, eller 10 ganger raskere på disk.

Så hvorfor er da mine kjøretider i Spark mer enn Mapreduce? Noe usikker på denne

Spark krever mye minne da den laster prosessene inn i minnet. Om man ikke har nok minne tilgjengelig vil ytelsen gå betraktelig ned, noe man kan se på mine kjøretider for Spark. Hos meg er MapReduce raskere da denne vil drepe prosessen så fort jobben er gjort. På den måten kan den lett kjøre sammen med andre prosesser uten at det utgjør særlig stor forskjell på ytelse (Xplenty, 2019).

Enklere:

Som man tydelig kan se utafra kodene er Spark mye enklere å benytte og skrive programmer i. Spark er en nyere metode for å håndtere store datamengder og har støtte for flere språk. Man har tilgang til et stort utvalg av API-er noe som gjør programmeringen og feilsøking enklere, samt gjør koden mindre. Dette syntes tydelig på sammenligningen av antall linjer med kode i programmene som er implementert (Educba, 2019).

3. Litteraturliste

Deptula, C. (2015). Hadoop: How to Update without Update. Hentet fra https://community.hitachivantara.com/s/article/hadoop-how-to-update-without-update

Educba (2019). MapReduce vs Apache Spark- 20 Useful Comparisons To Learn. Hentet fra https://www.educba.com/mapreduce-vs-apache-spark/

Edureka!, (2017, 9. mai). *Apache Hadoop Tutorial | Hadoop Tutorial For Beginners | Big Data Hadoop | Hadoop Training | Edureka* [Videoklipp]. Hentet fra https://www.youtube.com/watch?v=mafw2-CVYnA&fbclid=IwAR1mstik2Vsg2rHdIkMDerC5eMN-od1MwmbuHUh2h8JOCkLZryzUGn8K1_E

IntelliPaat (2019, 4. november). Spark vs MapReduce: Who is Winning? Hentet fra https://intellipaat.com/blog/spark-vs-map-reduce/

Udemy, (2019). *HDFS: What it is, and how it works* [Videoklipp]. Hentet fra https://www.udemy.com/course/the-ultimate-hands-on-hadoop-tame-your-big-data/learn/lecture/5951198#content

Xplenty (2019, 11. mars). Spark vs. Hadoop MapReduce. Hentet fra https://www.xplenty.com/blog/apache-spark-vs-hadoop-mapreduce/