

Portfolio assignment - part 1

Mette Strand Hornnes  
Studentnr: 171402

Innholdsfortegnelse

[1. Primary data source 3](#_Toc26379431)

[2. Assignment parts 4](#_Toc26379432)

[2.1 Method for importing initial data into HDFS 4](#_Toc26379433)

[2.2 Method for updating the data 8](#_Toc26379434)

[2.3 MapReduce and Spark 10](#_Toc26379435)

[2.3.1 Calculate simple metrics 10](#_Toc26379436)

[2.3.1.1 How many buildings is it in the extract you selected? 10](#_Toc26379437)

[2.3.1.2 How many addr:street tags exist for each street? 12](#_Toc26379438)

[2.3.1.3 Which object in the extract has been updated the most times, and what object is that? 13](#_Toc26379439)

[2.3.1.4 Which 20 highways contains the most nodes? 15](#_Toc26379440)

[2.3.1.5 What is the average number of nodes used to form the building ways in the extract? 17](#_Toc26379441)

[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” 19](#_Toc26379442)

[2.3.1. 7. Which 15 highways contains the most number of traffic calming=hump? 21](#_Toc26379443)

[2.3.2 Creative part 22](#_Toc26379444)

[2.3.2.1 Informasjon om valgt data 22](#_Toc26379445)

[2.3.2.2 Spørsmål 1 - Hvilke 10 steder i extracten er bysykkel mest populært å starte turen fra? 22](#_Toc26379446)

[2.3.2.3 Spørsmål 2 - Hvor finnes bysykkelstativer i ekstraktet? (Obs, kun de som er blitt brukt så langt i november) 23](#_Toc26379447)

[2.3.2.4 Why MapReduce or Spark is the correct or not the correct tool to answer them? 24](#_Toc26379448)

[2.3.3 Compare the performance of the MapReduce implementation and the Spark implementation 25](#_Toc26379449)

[3. Litteraturliste 28](#_Toc26379450)

# **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*](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  
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 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
   1. 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). Illustrasjoner er egenlaget.

Med HDFS bli data distribuert over flere ulike maskiner. Alle disse maskinene er koblet sammen og utgjør 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 et 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 også denne er klar til mottak.



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 har sendt såkalte «Heartbeats»-signaler om at den er ok) og returnerer IP-adressen til en av disse tilbake til ClienNoden.

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



1. 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.

1. 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 at lagringen er fullført 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 være enklere 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 hvordan den kjører egentlig ikke er noe vi trenger å tenke så mye på 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 verktøy 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 oppdatering 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 ingen annen spesifikk teknologi. Denne algoritmen kan implementeres med blant annet 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 (key-en).
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 oftest ut ifra 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 mappere).
6. **Overskriv masterdataen:** I dette siste steget vil dataen som ble midlertidig lagret i forrige steg 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) 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 jeg beskrive hva som skjer i henholdsvis MapReduce og Spark. Jeg har valgt å måle ytelse på antall linjer med kode (uten initilisering), oppstarts- og kjøretid. Da jeg har lagt merke til at disse kjøretidene varierer noe fra gang til gang har jeg kjørt de 4 ganger og tatt et gjennomsnitt av disse. Dette vil stå under hver beskrivelse i alle oppgavene jeg har klart å løse. All programmering løses i fil med angitt navn under henholdsvis Java- eler Scalamappe i innleveringen

I oppgave 3.3.3 vil en sammenligning av ytelsen bli gjort.

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

Løses i filen: One\_BuildingCount

**MAPREDUCE**  
MAPPEREN:   
I denne oppgaven sender jeg hele osm-filen til en mapper, som vil lese igjennom hele og dele den opp i tokens. For hvert ord som passer med k=»building» vil det sammen med et ettall sendes til reduceren.   
  
REDUCER:  
Reduceren mottar teksten «k=building» og en liste med ettall for hver gang dette står i osm-filen. I reduceren loopes det igjennom disse ettallene, summerer de sammen og deretter printes resultatet.   
  
IKKE OPTIMALISERT  
Jeg er klar over at denne metoden ikke er helt optimal da det er ønskelig å dele opp arbeidet i mindre deler som kan fordeles til ulike mappere og eventuelt til ulike noder i et cluster. Da jeg i denne oppgaven sender alt til en mapper som da må ta seg av hele filen, vil ikke dette arbeidet kunne fordeles og ytelsen blir dermed også mindre.   
Jeg har gjort en del forsøk på å få dette til, men mangel på tid gjorde at jeg dessverre så meg nødt til å prioritere andre deler. Forsøk som er gjort er å sette start- og sluttag til «node» slik at de ulike nodene blir sendt til hver sin mapper. Dessverre fikk jeg ikke dette til å fungere og måtte derfor gå tilbake til min opprinnelige versjon, som er den jeg nå har levert.

Resultat:

k="building" 2367

Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 14  
Oppstartstid: 2.265 s  
Kjøretid: 3.024 s

**SPARK**I sparkleses hele filen inn og splittes deretter i ulike ord (deles på mellomrom). For hvert av disse ordne sjekkes det om det er likt «K=»building». Om dette er tilfelle vil det samme med «true» legges til et ettall i et 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.

Resultat:  
  
buildingfalse: 960102  
buildingtrue: 2367  
  
Ytelse:  
Antall kodelinjer: ca. 5  
Oppstartstid: 5.972 s  
Kjøretid: 2.420 s

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

Løses i filen: Two\_AddrStreetTagsPerStreet

**MAPREDUCE**MAPPER  
Her settes start- og sluttTag til å være «node». Ved kjøring vil derfor de ulike node-taggene sendes til ulike mappere, noe som fører til at arbeidet kan utføres på ulike maskiner i et cluster. I mapperen hentes alle «taggene» i tildelt node ut. For hver «tagg» sjekkes det om den har attributten «add:street». Om dette er tilfelle vil verdien (altså navnet på gaten), sammen med et ettall sendes til reduceren.

REDUCER  
Reduceren mottar key-value-parene sendt fra mapper-jobbene og legger sammen valuen (i dette tilfelle ettallet) for hver like key. Deretter printer den ut resultatet.

Resultat   
Jeg har valgt å kunne legge ved de første linjene fra resultatet. Totalt antall linjer = 149

Adamstuen 1

Akersbakken 37

Akersveien 3

Alexander Kiellands   
plass 4

Anna Sethnes gate 7

Anton Schjøths gate 21

Armauer Hansens gate 21

Banksjef Frølichs gate 5

Benneches gate 13

Bentsebrugata 8  
Bergstien 21

Bergverksgata 4

Biermanns gate 10

Birkelunden 1

Bislettgata 4

Bjerkelundgata 9

Bjerregaards gate 112

Bolteløkka allé 14

Brageveien 17

Brandts gate   
Casparis gate 9

Christian Michelsens gate 11

Christies gate 44

Claus Riis’ gate 7

Colletts gate 104

Colletts gate x Louises gt 1

Dalsbergstien 35

Darres gate 18

Diriks’ gate 7

Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 20  
Oppstartstid: 2.265 s  
Kjøretid: 10.025 s

**SPARK**I denne oppgaven (og de videre) vil først initialisering av Spark skje. Her settes blant annet hva som er root- og radTag. I denne oppgaven blir radTaggen satt til å være «tag» slik disse sendes inn hver for seg og kan potensielt bli prosessert på ulike noder i et cluster.   
Først vil det hentes ut attributtene 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 konsollen.

Resultat:

  
  
Ytelse:Antall kodelinjer: ca. 3  
Oppstartstid: 14.950 s  
Kjøretid : 6.115 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**  
MAPPER  
I denne oppgaven settes start- og sluttTag 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 versjonsnummeret er større enn tidligere versjonsnummer som er hentet ut, vil denne tas vare på. Man sitter da igjen med det høyeste versjonsnummeret, samt id-en til den tilhørende noden. Dette sendes så til reduceren i et key-value-par.

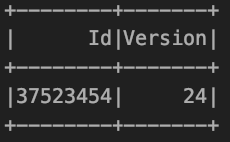
REDUCER   
Reduceren vil i denne oppgaven kun ta imot det høyeste versjonsnummeret og id-en til den tilhørende noden. Deretter vil den printe ut resultatet. Da denne oppgaven ikke er gjort på best mulig måte blir denne reduceren noe overflødig i dette tilfellet.

IKKE OPTIMALISERT  
Likt som i oppgave 2.3.1.1 er ikke denne helt optimalisert noe som fører til at jeg får en ekstremt lang kjøretid. Årsaken til dette er at istedenfor å la ulike mappere ta seg av ulike, mindre deler av arbeidet sender jeg inn hele osm-filen til en mapper.   
  
Likt som i den tidligere oppgaven har jeg forsøkt å sette start- og sluttTag til node slik at de ulike node-taggene kan sendes til ulike mapperne. Mapperne ville da ha hentet ut id til noden den skal håndtere og deretter sende denne sammen med versjonsnummeret til reduceren som et key-value par.   
Reduceren ville ha gått igjennom alle verdiene for å finne det høyeste versjonsnummeret. Deretter printet dette sammen med id-en til den tilhørende noden.

Resultat:  
Node with id 37523454 has been updated most times. Numbers of updates: 24

Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 21  
Oppstartstid: 2.136 s  
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 versjonsattributten (samt setter alias for disse for enklere referering videre).   
Videre vil denne tabellen sorteres synkende slik at noden med det høyeste versjonsnummeret ligger øverst. Helt sist vises kun denne ene, ved å spesifisere at kun første rad skal printes.

Resultat:  
  
  
  
Ytelse:  
Antall kodelinjer: ca. 3  
Oppstartstid: 21.578 s  
Kjøretid: 5.720 s

#### 2.3.1.4 Which 20 highways contains the most nodes?

Løses i filen: Four\_20TopHighWayNodes

**MAPREDUCE:**  
MAPPER  
Start- og sluttTag settes til way. På den måten vil de ulike way-taggene bli sendt til ulike mappere.   
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.

REDUCER  
Reduceren legger sammen antall mottatt for hver way og printer ut resultatene.   
  
MANGLER SORTERING  
Jeg har i denne oppgaven ikke fått å kun vise de 20 way-ene som inneholder flest noder. En av løsningene jeg vurderte for å få til dette var å benytte en hashMap. Her ville jeg ha lagt de ulike wayene som nøkler og antall noder som verdier til. Med en slik hashMap kunne jeg så ha sortert den synkende etter antall noder og deretter velge å printe ut de 20 første fra den. Da jeg ser at dette er noe ineffektivt har jeg ikke prioritert å sette opp dette, men lar oppgaven stå som den er.

Resultat:  
Da jeg printer ut alle nodene er det en ganske stor output (totalt 1443 linjer). Jeg velger derfor å legge ved de første 20 første linjene.

101261489 40

101263541 8

101263545 8

101263554 8

101263556 12

101263569 16

101263572 12

101263575 12

104687905 30

107436303 24

107919307 80

107958888 82

107958889 14

107958890 36

108225976 12

108225977 32

108225978 18

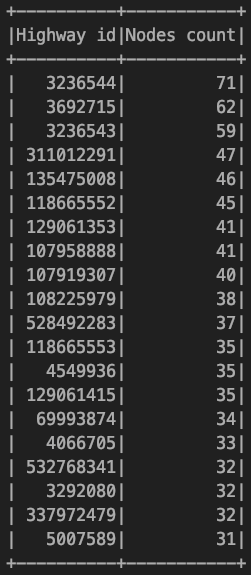
108225979 152

108324357 28

108324358 6

Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 27  
Oppstartstid: 3.897 s  
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 lik «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 antall noder plukkes ut (Da det er kun dette jeg ønsker å vise i tabellen) og de 20 øverste radene vises.

Resultat:  
  
  
  
Ytelse:  
Antall kodelinjer: ca. 6  
Oppstartstid: 14.295 s  
Kjøretid: 12.127 s

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

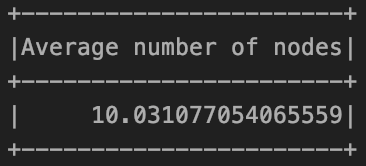
Løses i filen: Five\_AverageNumOfNodesBuilding

**MAPREDUCE**  
MAPPER  
Start- og sluttTag settes til «way» slik at de sendes til ulike mappere.   
Mapperen vil først hente ut antall nd-tagger for nåværende way. Deretter itereres det igjennom alle taggene for denne wayen og sjekker om en av disse har attributter som tilsier at dette er en building. Om dette er tilfelle vil tekst og antall noder som er telt opp for denne wayen sendes til reduceren.   
  
REDUCER  
I reduceren, som nå mottar antall nd-tagger for hver «buildingway», legges disse sammen. Her telles det også hvor mange buildingnoder som finnes. Med denne informasjonen finner man gjennomsnittet som deretter printes ut.

Resultat:

The average number of nodes used to form the building ways is: 10  
  
Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 20  
Oppstartstid: 3.011 s  
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.

Resultat:  
  
  
  
Ytelse:  
Antall kodelinjer: ca. 4  
Oppstartstid: 18.570 s  
Kjøretid: 9.496 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”

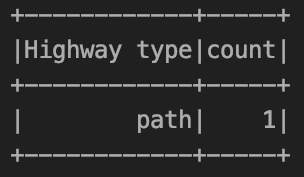
Løses i filen: Six\_NumOfLiftGate

**MAPREDUCE**  
MAPPER  
Starttag settes til å være way, slik at hver mapper kan behandle mindre deler av arbeidet.   
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 tilfellet 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 type highway sendes til reduceren med et ettall om det tidligere har vist seg at den også er en liftgate. Om dette ikke er tilfelle vil type highway sendes til reduceren med 0.  
   
REDUCER  
Reduceren vil så legge sammen alle verdiene som kommer inn for hver highway og deretter. printe dette.

Resultat:  
I min xml-fil finnes det ingen av de riktige typer highway som inneholder taggen bar-rier=liftgate. Det finnes også kun highway av typen path og service. Jeg får derfor følgende resultat:

Numbers of ways of type highway= path that has a node with the tag ”bar-rier=liftgate”: 0  
Numbers of ways of type highway= service that has a node with the tag ”bar-rier=liftgate”: 0  
  
Ytelse:  
Antall kodelinjer i mapper og reducer: ca. 31  
Oppstartstid: 2.194 s  
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 tagger ut igjen til en ny tabell. Denne 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 inneholder 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.

Resultat:  
Som nevnt i oppgaven med MapReduce har ikke min osm-fil noe match på det det søkes etter. For utskrift av denne i Spark har jeg derfor lagt til en Highway med korrekt type med barrier=liftgate for å vise hvordan utskriften ville ha blitt.   
  
  
  
Ytelse:  
Antall kodelinjer: ca. 9  
Oppstartstid: 15.188 s  
Kjøretid: 11.711 s

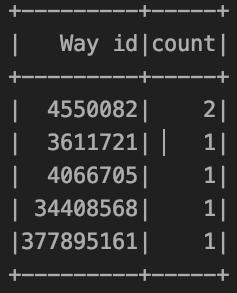
#### 2.3.1. 7. Which 15 highways contains the most number of traffic calming=hump?

Oppgaven er kun implementert med Spark.

Løses i filen: Seven\_15HighwaysMostTrafficCalmingHump

I denne oppgaven er jeg nødt til å lese både Noder og Wayer. Jeg setter derfor to ulike DataFramer, en hvor radTagg er node og en som er way. På den måten kan disse håndteres hver for seg. For de ulike wayene henter jeg ut tilhørende id og de ulike barnenodene med referansenumre for denne veien. Deretter hentes id og barnetagger for de ulike nodene, som deretter filtreres slik at man står igjen med de nodene som er en traffic calming med type hump. De to tabellene kobles nå sammen på id til nodene og referanseid til wayene. Avslutningsvis gjør jeg en opptelling av antall linjer per way-id og sorterer dette synkende.

Resultat:  
I mitt datasett har jeg totalt kun 6 humps og får da følgende resultat

  
  
  
Ytelse:  
Antall kodelinjer: ca. 6  
Oppstartstid: 30.810 s  
Kjøretid: 26.056 s

### 2.3.2 Creative part

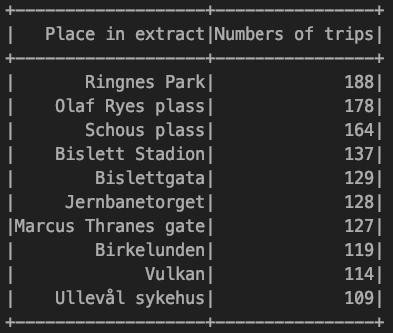
2.3.2.1 Informasjon om valgt data  
Jeg har valgt å benytte Oslo Bysykkel som ekstra data. Dataen som her hentes inneholder turhistorikk fra Oslo Bysykkel. I denne CSV-filen ligger følgende informasjon for alle turer så langt i november:  
Tidspunktet turen startet, tidspunktet turen ble avsluttet, lengde 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?

Oppgaven er kun implementert med Spark.

Løses i filen: CreativeOne\_topTenCityCyclePlacesInExtract

Til lesingen av osm-filen setter jeg nå radTaggen til å være «tag», da det er disse jeg skal gå igjennom. Etter initialiseringen starter programmet med å hente ut de ulike stedene som finnes i ekstrakten jeg har valgt. Deretter oppretter jeg en ny tabell 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.

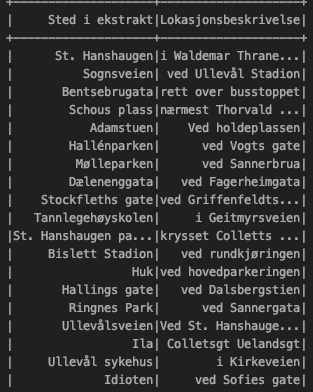
Resultat:  
  
  
  
Ytelse:  
Antall kodelinjer: ca. 5  
Oppstartstid: 20.353 s  
Kjøretid: 13.161 s

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

Oppgaven er kun implementert med Spark.

Løses i filen: CreativeTwo\_whereAreTheCyclingStationsInTheExtract

RadTag settes til å være «tag». 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.

Resultat:  
Viser her et utvalg av resultatet. Totalt er resultatet på 44 antall linjer.  
  
  
  
Ytelse:  
Antall kodelinjer: ca. 5  
oppstartstid: 31.453 s  
Kjøretid: 6.804 s

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

Jeg har kun implementert mine kreative oppgaver i Spark. En stor årsak til dette er Spark sitt enkle oppsett og måte å utføre disse oppgavene på. For en nybegynner i arbeid med Big data gjør Spark dette mye enklere. I dette prosjektet hvor noen deler har måttet bli prioritert over andre, falt valget på Spark nettopp på grunn av dens enkle måte å arbeide med.

I disse kreative oppgavene leser jeg inn fra to ulike filer. Dette krever mer prosessering og at samme data må prosesseres flere ganger. Ved at Spark kan ha dataen tilgjengelig i minnet gjør denne behandlingen av data mye raskere og derfor et bedre valg (IntelliPaat, 2019).   
Det kan være noen utfordringer om man mangler nok RAM (in-memory caching), slik at MapReduce ville vært fordelaktig. Dette noe jeg skriver mer om under punkt 2.3.3.

### 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 for å håndtere store datamengder, er skalerbare og kan benytte opptil 1000 noder i et cluster (Xplenty, 2019). Det er likevel noe som skiller de fra hverandre. 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.

SAMMENLIGNING AV PROGRAMMENE

|  |  |  |  |
| --- | --- | --- | --- |
| 1 How many buildings is it in the extract you selected? | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 14 | 2.265 s | 3.024 s |
| Spark | 5 | 5.972 | 2.420 s |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 How many addr:street tags exist for each street? | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 20 | 2.265 s | 10.025 s |
| Spark | 3 | 14.950 s | 6.115 s |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 Which object in the extract has been updated the most times, and what object is that? | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 21 | 2.136 s | 87.876 s |
| Spark | 3 | 21.578 s | 5.720 s |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 Which 20 highways contains the most nodes? | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 27 | 3.897 s | 8.478 s |
| Spark | 6 | 14.295 s | 12.127 s |

|  |  |  |  |
| --- | --- | --- | --- |
| 5 What is the average number of nodes used to form the building ways in the extract? | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 20 | 3.011 s | 7.466 s |
| Spark | 4 | 18.570 s | 9.496 s |

|  |  |  |  |
| --- | --- | --- | --- |
| 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” | | | |
|  | Kodelinjer | Oppstartstid | Kjøretid |
| MapReduce | 31 | 2.194 s | 10.832 s |
| Spark | 9 | 15.188 s | 11.711 s |

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 oppstarts og kjøretid. I oversikten over sammenligner jeg de programmene jeg har implementert i både MapReduce og Spark. Her kommer det frem nokså store forskjeller når det gjelder prosesseringshastighet og hvor lett er å bruke. Det er derfor dette jeg velger å fokusere på i denne sammenligningen.  
  
Prosesseringshastighet:  
Spark prosesserer data i RAM 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. Spark sin evne til å ha dataen tilgjengelig i minnet gjør at ytelsen ved bruk av denne teknologien utkonkurrerer MapReduce (IntelliPaat, 2019). Faktisk kjører programmene opp til 100 ganger raskere enn MapReduce i minnet, eller 10 ganger raskere på disk.   
  
Så hvorfor er da mange av mine kjøretider i Spark mer enn Mapreduce?   
Spark krever mye minne da den laster prosessene inn i RAM. 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). Det er på de aller fleste oppgavene selve initialiseringen og oppstart som tar ekstra tid i Spark. Utplukking og visning av data senere er som oftest en del raskere.

Enklere:  
Som man tydelig kan se utafra kodene er Spark, med Scala som programmeringsspråk, mye enklere. 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 (Educba, 2019). Dette syntes tydelig på sammenligningen av antall linjer med kode i programmene som er implementert.

# **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/

ILLUSTRASJONER

Forside:

Freepik (u.å). *Creative big data background concept*. Hentet fra https://www.freepik.com/free-photos-vectors/background

Øvrige illustrasjoner er laget selv.