NTNU
Kunnskap for en bedre verder

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4305 Big Data-Arkitektur

Faglig kontakt under eksamen: Kjetil Nørvåg/Heri Ramampiaro
Tlf.: 73596755/73591459

Eksamensdato: 28. mai 2016
Eksamenstid (fra-til): 09.00-13.00
Hjelpemiddelkode/Tillatte hjelpemidler: D: Ingen trykte eller håndskrevne hjelpemiddel tillatt.
Bestemt, enkel kalkulator tillatt. Annen informasjon:

Målform/språk: Bokmål
Antall sider (uten forside): 4
Antall sider vedlegg: 0

Kontrollert av:

_____
Dato Sign

Informasjon om trykking av eksamensoppgave
Originalen er:
1-sidig X 2-sidig □
sort/hvit X farger □

_____ Merk!
Studenter finner sensur i Studentweb. Har du spørsmål om din sensur må du kontakte instituttet ditt. Eksamenskontoret vil ikke kunne svare på slike spørsmål.

Oppgave 1 – Big Data – 5 %

a) Når man skal forklare Big Data snakker man ofte om de tre (eller flere) V'ene. Forklar de tre viktigste av disse.

Volume: refers to the amount of data being generated and that needs handling (such as storage and analysis). There are several factors contributing to the large increase in data volume in the Big Data era versus previously, the most important being the growth of social media platforms, and the utilization of sensors. Both social media (uploading high resolution images and videos) and sensors (internet of things, credit card scanners, other autonomous systems) generate enormous amounts of data.

Velocity: refers to the speed at which data is being generated, accumulated, ingested, and processed. E. g Twitter users generate millions of tweets every minute that need to be stored and processed, for example to analyze trends etc. Utilization of social media and sensors is therefore also a big factor in explaining the increase of velocity in the big data era compared to previously.

Variety: refers to the variety in the data being generated. Before, it was a lot more common that a datasource only generated structured data, while now, datasources can generate combinations of structured, semi-structured, and unstructured data. Previously, sources of data were mainly transactions involving financial, insurance, travel, healthcare, retail industries, and governmental and judicial processing. Now, these sources have expanded dramatically, and includes satellite data, internet data (clickstream and social media), research data, images, videos, emails, signal data (sensors).

Oppgave 2 – Hadoop – 20 % (alle deler teller likt)

a) Hva var viktige mål for Hadoop File system (HDFS), og hva er HDFS *ikke* egnet til?

De viktigste målene for HDFS var følgende:

1. Bruk av veldig store filer. Tanken bak dette var å redusere total tid brukt på aksess. Ved å bruke store filer kan man finne entrypointet for informasjonen man leter etter, og lese kontinuerlig i den store fila før man trenger å aksessere et nytt entrypoint.
2. Datastrøm-aksess. Det at man ønsker datastrøm-aksess er også litt av grunnen til at man lagrer data i veldig store blokker (filer). I big-data sammenheng er datastrømmer et viktig aspekt, og målet med datastrøm-aksess i hadoop er at man skal lagre store filer, aksessere fila i begynnelsen, og leser store deler av fila kontinuerlig fra begynnelsen og utover. Ved å bruke store filer trenger man mindre søking enn dersom man heller bruker mange små filer.
3. Bruk av hyllevare-maskiner. Dette er fordi hyllevaremaskiner ofte er billige, og det gjør

systemet skalerbart ved at man kan kjøpe flere maskiner og koble de opp mot systemet ved behov for større lagringsplass, eller eventuelt fjerne maskiner og selge dem eller flytte dem dersom behovet for lagringsplass minker.

HDFS er ikke egnet til

1. Data-aksess av mindre datablokker med krav til liten forsinking. Dette fordi dataen ligger lagra i store filer, og systemet er lagd for aksessmønsteret at man aksesserer entry-pointet til ei fil og leser kontinuerlig store deler av fila.
2. Mange små filer. Fordi dataen er lagra i store filer som beskrevet over, og også på grunn av måten dataen aksesseres på. Navnenoder i HDFS holder all oppslagsinfromasjon om hver fil i hovedminnet, noe som kan bli vanskelig å få til dersom man bruker mange store filer (tar for mye plass til å passe i hovedminnet)
3. Flere klienter som skriver til ei fil simultant. Dette går ikke i HDFS, mest på grunn av at man ønsker å gjøre ting enklest mulig for å forbedre feiltoleranse og feilhåndtering. Altså kan kun en klient skrive til ei fil på en gang, men flere kan lese fra samme fil på samme tid.
4. Vilkårlige oppdateringer i filer. I HDFS er det ikke mulig å oppdatere ei fil på andre måter enn å legge til data på slutten av fila. Dvs at HDFS er append-only. Dersom man vil gjøre endringer på en fil på andre måter enn å legge til noe på slutten, må man gjøre dette ved å kopiere innholdet i fila, gjøre endringen, og så opprette ei ny fil og skrive den oppdaterte fila til den nye. Dette er svært lite effektivt.

b) Beskriv arkitekturen til HDFS (bruk gjerne figur). Beskriv hvordan filer er lagret og node-typer.

HDFS består av racks som igjen består av et visst antall hyllevare-maskiner. Innad i et rack, går kommunikasjonen mellom maskiner gjennom en switch som ligger i racket. Det vil si at hvis en node (maskin) i et rack vil kommunisere med en annen node i samme rack, så sender den først informasjonen til switchen, som sender det videre til den aktuelle noden. Man sier at nettverksdistansen for kommunikasjon mellom to noder i samme rack er 2 (1 for første kommunikasjon fra sender-noden til switchen, og 1 for kommunikasjon fra switchen til

mottakernoden). Nettverksdistansen her er svært kort, som gjør kommunikasjon mellom maskiner i samme rack veldig effektiv.

I tilleg til dette kan racks kommunisere med andre racks i systemet. Denne kommunikasjonen går gjennom en annen switch som kobler racksene sammen. Nettverksdistansen mellom to noder i forskjellige racks blir da 4 (først 1 for kommunikasjon mellom sender-node og intern rack-switch, så 1 for kommunikasjon fra intern rack-switch til switchen som kobler rackene sammen, så 1 for kommunikasjon fra den switchen til den interne rack-switchen som mottakernoden ligger i, og til slutt 1 for kommunikasjon fra den switchen til mottakernoden).

Disse rackene kan også være lokalisert i forskjellige datasenter (som kan være i forskjellige bygninger, eller forskjellige geografiske lokasjoner). Da får man enda en switch som sørger for kommunikasjon mellom datasenter. Nettverksdistansen mellom to noder i forksjellige datasenter blir 6 (først 1 fra sender-noden til intern rack-switch, så 1 fra intern rack-switch til switch som kobler rackene sammen, så 1 fra den switchen til switchen som kobler datasentrene sammen, så 1 fra den switchen til switchen som kobler rackene i mottaker-datasenteret sammen, så 1 fra den switchen til intern rack-switch i mottaker-racket, og til slutt 1 fra intern rack-switch til mottakernoden).

Datablokkene er lagret på store filer som ligger på disse maskinene. I HDFS bruker man partisjonering og replikering. Partisjonering betyr at deler av ei fil kan være lagra på forskjellige maskiner. For eksempel for ei fil som består av datablokker A, B og C, så kan A og B være lagra på samme maskin, mens C kan ligge på en annen maskin. Replikering betyr at hver datablokk er lagra flere steder, som oftes på tre forskjellige maskiner (tre-veis replikering). Dette øker feiltoleransen, ved at dersom en maskin krasjer, så ligger fortsatt de filene som eksisterte på den maskina også på andre tilgjengelige maskiner som kan håndtere forskjellige forespørsler.

Ettersom HDFS bruker replikering, så må man ha en node som kjører en tjeneste som kan fortelle hvor de forskjellige filene ligger lagret. Denne kalles Namenode (navnenode), og kan ta inn et filnavn og svare på blant annet hvor mange replikasjoner det finnes av den fila, og hvor de ulike

replikasjonene ligger (ved bruk av blokk-identifikatorene som den aktuelle fila består av, samt info om hvilke datanoder blokkene ligger på). Navnenoda holder all informasjon i hovedlageret for å sørge for effektivitet, og bruker kun disken til back-up. Det at all informasjon i navnenoda skal kunne ligge i hovedlageret er også en god grunn til å ha store filer, ettersom hver fil vil bruke plass på navnenoda, og med færre filer så blir det mindre plass brukt på navnenoda. Back upen som er lagret på disk i navnenoda består av et namespace-image (en kopi av det som ligger i hovedlageret på et visst tidspunkt), og en edit-log (en log som består av alle oppdateringer som gjøres i hovedlageret), og etter en viss tid, vil namespace-image og edit-log merges sammen, slik at man igjen blir up-to-date med det som ligger i hovedlageret.

Selve dataen ligger lagra i datanoder. En datanode lagrer hver datablokk som ei fil, og har blokk-identifikatoren som del av filnavnene. I tillegg til selve datablokkene, så inneholder datanoda også metadata-informasjon om hver av blokkene som ligger på den noda. Den viktigste metadata-informasjonen er en sjekksum for hver blokk. Den fungerer sånn at når en klient skriver til ei fil, så blir det også lagd en sjekksum som lagres som meta-data i den aktuelle datanoda. Når den samme fila skal leses igjen, så sjekker regner datanoda ut sjekksummen for det som ligger i fila og sammenligner med sjekksummen den har lagra i metadataen. Dersom de stemmer overens, så kan fillesingen fortsette, men hvis ikke så har det skjedd noe galt med fila, og man må da lese fila fra en annen replikasjon.

c) Forklar hva som skjer når en klient skal lese en fil som er lagret i HDFS (inkl. interaksjon mellom noder).

Når en klient skal lese en fil som er lagret i HDFS, sender klienten først en forespørsel med filnavn til navnenoden ved bruk av et Hadoop API. Navnenoden vil da slå opp i katalogen den har liggende i hovedminnet, og returnere informasjon om fila til klienten. Denne informasjonen inkluderer blant annet blokkidentifikatorene til blokkene som fila ligger lagra på, samt hvilke datanoder de blokkene ligger på.

Når klienten mottar denne informasjonen, vil den sende leseforespørselen direkte til den/de aktuelle datanoden(e). Datanoden som mottar forespørselen vil da først sjekke at sjekksummen med den som ligger i blokka som skal leses stemmer over ens med den sjekksummen den har liggende i meta-data-fila for den blokka, og dersom det stemmer, så begynner datanoden å sende over dataen i fila direkte til

klienten.

Oppgave 3 – MapReduce og Spark– 10 % (alle deler teller likt)

Anta at man har en fil PersonInfo.txt som inneholder informasjon om navn, alder og lønn, dvs. format som dette:

Kari 45 450000
Ola 30 200000
Kate 30 500000
Pål 45 550000

Vi ønsker å finne gjennomsnittsinntekt for hver alder, dvs. resultat som dette (trenger ikke å være sortert):

45 500000
30 350000

a) Vis med pseudokode for *mapper* og *reducer* hvordan dette kan gjøres i MapReduce. Anta for enkelhet skyld at *value* til map er en post med feltene *age* og *salary*, dvs. bruk følgende som utgangspunkt:

```
public void map(key(name), value(age,salary))
public void reduce(key, Iterable values)
```

```
public void map(key(name), value(age, salary)){
     skriv value[age], value[salary]
}

public void reduce(key, Iterable values){
     n = 0
     total = 0
     for each val in values:
          n++
          total += val
     avg_salary = total/n
     skriv (key, avg_salary)
```

```
}
```

b) Vi ønsker nå å finne maks-lønn for hver alder ved hjelp av Spark. Anta at vi allerede har lest filen inn i en RDD av par (key,value)=(age,salary), dvs. RDD[(int,int)]. Vis hvilke(n) transformasjon(er) som må gjøres for å få en resulterende RDD der (key,value)=(age,maxSalary). Hint: viktige transformasjoner og handlinger ("actions") i Spark inkluderer `map, flatMap, filter, distinct, union, collect, count, countByValue, reduce, reduceByKey, groupByKey, values, sortByKey,` og `countByKey.`

```
val maxSalary = rdd.reduceByKey((a,b) => Math.max(a,b))
```

Oppgave 5 – Datastrømmer (streaming data) – 30 % (Alle deler teller likt)

Du skal analysere hvor mange ganger et tema om amerikansk valg og valgkamp blir nevnt i meldinger i sosiale media som Twitter.

a) Drøft karakteristikkene og/eller utfordringene med datastrøm. Nevn to andre eksempler hvor man er nødt til å håndtere en datastrøm (i tillegg til Twitter og eller sosiale media generelt).

Karakteristikker:

- Datastrømmer er kontinuerlige strømmer av data.
- Det er snakk om store mengder data
- Man vet ikke når, eller om de slutter. Kan være uendelige.

Utfordringer:

Den største utfordringen med datastrømmer er at den endrer seg raskt og kontinuerlig, og derfor trenger rask respons i sanntid.

Eksempler hvor man må håndtere datastrømmer:

- Signaldata, for eksempel sensordata fra sensorer i en bil for å gjøre beregninger for å unngå kollisjoner
- Transaksjonsdata, for å detektere om deler av systemet er nede som kan føre til for eksempel tapt inntekt.

b) Vi skiller mellom to typer spørringer når det gjelder datastrøm. Forklar hva disse er. Bruk eksempler til å støtte forklaringen din.

Når det gjelder datastrøm skiller vi mellom to typer spørringer:

- Ad-hoc spørringer: disse spørres kun en gang. Et eksempel er hvis man har en strøm av kjøpstransaksjoner og ønsker å vite prisen for det dyreste kjøpet på slutten av en dag. Da kan man spørre hvilken verdi som har vært den høyeste i løpet av dagen.

- Stående sørringer: disse spørres kontinuerlig. Et eksempel er hvis man har en strøm av twitter-tags, og alltid har lyst til å vite hvilken tag som er mest brukt for å kunne analysere trender i sanntid. Da kan man feks ha en stående spørring "hvilken tag er mest brukt i dag", som vil kunne gi ulike svar til ulike tidspunkter på dagen.

c) Se for deg at du skal finne ut hvor stor andel av meldingene er relatert til temaet "valg" og "valgkamp" i en gitt begrenset tidsperiode. Til dette formålet velger vi å bruke glidende-vindu prinsippet ("sliding window"). Anta at dette vinduet har en størrelse på 1000 twitter-meldinger (dvs. Tweets). Vis hvordan du går fram for å beregne denne andelen.

For de første 1000 twitter-meldingene i vinduet, kan man beregne andelen ved å gå gjennom alle meldingene i vinduet, finne ut hvor mange som inneholder ordene "valg" og/eller "valgkamp", og dele dette på 1000 (antall meldinger totalt i vinduet).

Deretter må man hver gang den kommer en ny tweet inn i vinduet sjekke om denne meldingen inneholder de aktuelle ordene. Hvis den gjør det, så endrer man snittet med (1-j)/N hvor N er antall tweets i vinduet (1000), og j =1 dersom den eldste tweeten i vinduet inneholdt ordene, og j = 0 dersom den eldste tweeten i vinduet ikke inneholdt ordene.

d) Kan problemet over sees på som en variant av "bit counting"? Begrunn svaret ditt.

Ja, man kan representere tweets som inneholder ordene man leter etter som 1 og tweets som ikke gjør det som 0. Deretter kan man telle antall enere i et gitt størrelse av et vindu osv.

e) Bruk "bloom filter"-prinsippet til å fylle ut tabellen nedenfor

Strømelement	**Hash-funksjon - h_1**	**Hash-funksjon - h_2**	**Filtrere Innhold**
			00000000000
39 = 10 0111	011 = 3	101 = 5	00010100000
214 = 1101 0110	1110 = 14 => 3	1001 = 9	00010100010
353 = 01 0110 0001	11001=25=> 3	00100 = 4	00011100010

Hint: bruk $h(x)= y \bmod 11$, der y er hentet henholdsvis fra oddetalls-bits fra x eller partalls-bits fra x.

f) Anta at vi vil analysere de 11 siste meldingene som er kommet inn. Generelt på twitter, vil mange av meldingene bli sendt på nytt av samme bruker for å markere sitt synspunkt. Andre brukere vil "re-tweete" for å få flere til å få med seg meldingene. Forklar hvordan vi kan bruke bloom-filtre for å filtrere bort slike meldinger. Gjør de antakelsene du finner nødvendig.

Her kan vi sende hver av de 11 meldingene gjennom et visst antall hash-funksjoner som alle gir et integer tall. Vi starter med et bloom-filter som består av kun 0er. Når vi sender den første meldingen

gjennom første hashfunksjon, vil vi få et tall. Vi bruker det tallet og den biten som ligger i den posisjonen i bloom-filteret til 1 (dersom resultatet fra hash-funksjonen er 3, teller vi fra 0-3 fra venstre side i bloom-filteret, og setter den aktuelle biten til 1). Dette gjør vi for alle hashfunksjonene. Når vi har gjort dette for den første meldingen, går vi videre og gjør det samme for neste melding. Dersom vi får en melding hvor resultatet fra alle hash-funksjonene er sett før, altså at bitene i bloom-filteren alle er 1, så kan vi anta at meldingen er sett før. Dette kan likevel gi falske positiver, ved at en kombinasjon av andre meldinger har ført til at bloom-filteret består av 1ere på de aktuelle posisjonene. Vi kan derfor risikere at vi registrerer en melding som sett før selv om den egentlig er ny.

Det eneste vi kan finne ut helt sikkert med et bloom-filter er om en melding ikke er sett før. Dette gjelder dersom resultatet fra en av hashfunksjonene for en melding gir et tall, og man kan se at biten på den posisjonen i bloom-filteret er 0. Da kan vi si helt sikkert at meldingen ikke er sett før.

g) Anta nå at når de 11 meldingene har kommet inn har vi fått en strøm av data som ser ut som dette: 10100101010. Kan vi ha sett meldingen som kan representeres ved y = 1111011 før? Begrunn svaret ditt.

Oddetall: 1101 = 13 % 11= 2

Partall: 111 = 7

Denne meldingen kan ha vært sett før, siden bitene i posisjon 2 og 7 fra venstre side i filteret er 1.

Oppgave 6 – Anbefalingssystem (recommender systems) – 20 % (6% på a.i, 4% på a.ii, 10% på b)

Du er nyansatt i et nytt firma som vil spesialisere seg på strømming av film. En av oppgavene dine er å utvikle gode anbefalingsalgoritmer og metoder.

a) En del av metoden du foreslår går ut på å gi brukeren mulighet til å "rate" filmene for så bruke dette til å finne ut hvilke filmer systemet deres skal anbefale senere. Anta at brukerne deres har "rated" følgende 10 filmer med 3 eller flere stjerner:

```
Jurasic Park (Fantasi/SciFi), Harry Potter (Fantasi/Adventure),
ET (SciFi), Lord of the Rings (Fantasi/Adventure), Alien (SciFi),
Terminator (SciFi), 101 Dalmatians (Adventure/Family), Titanic
(Romantic), Sleepless in Seattle (Romantic) og Mr. Bean (Comedy).
```

i. Forklar hvordan du vil gå fram for å anbefale neste film til denne brukeren. Gjør de antakelsene du finner nødvendig.

Dersom metoden med rating har blitt implementert for alle brukere, ville jeg ha brukt item-item

anbefaling. Det vil si at jeg ser på hvordan de ulike filmene er ratet av de forskjellige brukerne, og bruker likhetsestimering for å finne en estimert rating for neste film.

ii. Ville du brukt innholdsbasert ("content-based") anbefalingsmetode eller "collaborative filtering"? Begrunn svaret ditt.

Her ville jeg brukt collaborative filtering, i og med at brukerne rater filmene og slik den nødvendige informasjonen er tilgjengelig. Collaborative filtering er generelt bedre enn content-based.

b) Anta følgende brukerratingstabell.

movies \ users	1	2	3	4	5	6	7	8
1	1		3			5		
2			5	4			4	
3	2	4		1	2		3	
4		2	4		5			4
5			4	3	4	2		
6	1		3		3		A	2

Bruk "***item-item collaborative filtering***"-metoden til å foreslå bruker nr. 7 sin rating av film nr. 6. Dvs. hva blir ratingverdien A? Du må vise mellomregningen.

Til denne oppgaven vil du trenge følgende formler:

Pearson Correlation similarity - likhet mellom vektor x, og vektor y:

$$sim(x,y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})(r_{ys} - \overline{r_y})}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \overline{r_y})^2}}$$

der r_{xs} er bruker s sin rating på film x og $\overline{r_x}$ (overline) er gjennomsnitt av alle rating-ene på film x. **Vektet gjennomsnitt (weighted average)** for en brukers ratinger:

$$r_{ix} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{jx}}{\sum s_{ij}}$$

r_{ix} er her bruker x sin rating på film i, mens s_{ij} er likhet (similarity) mellom ratingene til film i og j

sim(6,1) = 0.53

sim(6,2) = 0.37

sim(6,3) = 0.05

sim(6,4) = 0.29

sim(6,5) = 0.41

sim(6,6) = 1

Vektet snitt = (4*0.37 + 3*0.05)/(0.37+0.05) = 3.88

NTNU
Kunnskap for en bedre verder

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4305 Big Data-Arkitektur

Faglig kontakt under eksamen: Kjetil Nørvåg/Heri Ramampiaro

Tlf.: 73596755/73591459

Eksamensdato: 16. mai 2017

Eksamenstid (fra-til): 09.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: D: Ingen trykte eller håndskrevne hjelpemiddel tillatt.

Bestemt, enkel kalkulator tillatt. Annen informasjon:

Målform/språk: Bokmål

Antall sider (uten forside): 4

Antall sider vedlegg: 0

Kontrollert av:

_____

Dato Sign

Informasjon om trykking av eksamensoppgave Originalen er:

1-sidig X 2-sidig □

sort/hvit X farger □

_____ Merk!

Studenter finner sensur i Studentweb. Har du spørsmål om din sensur må du kontakte instituttet ditt. Eksamenskontoret vil ikke kunne svare på slike spørsmål.

Oppgave 1 – Hadoop – 15 % (alle deler teller likt)

a) Forklar replikering i HDFS.

Replikering i HDFS går ut på at datablokkene blir lagret på mer enn en plass. Som oftest er det tre-veis replikering, som vil si at en datablokk vil bli lagret på tre forskjellige noder. Disse kan være på helt forskjellige lokasjoner. Replikering bidrar til feiltoleranse, ettersom dersom en node krasjer, så ligger fortsatt blokkene som var lagret på den noden på andre noder som kan håndtere forespørsler.

b) Hva er hensikten med NameNode, og hvilken informasjon har denne lagret?

NameNode skal virke som et oppslagsverk for metadata om filene som er lagret i systemet. NameNode kan få inn et filnavn, og returnere blant annet antall replikeringer, blokkidentifikatorene til blokkene som fila består av, samt hvilke datanoder disse blokkene ligger på. Når en kllient for eksempel vil lese en fil, kontakter den NameNode og spør om denne informasjonen. Når NameNode svarer klienten, har klienten den nødvendige informasjonen den trenger for å kontakte aktuelle datanoder direkte. All informasjon som NameNode har kontroll på ligger i hovedminnet for å sørge for mest mulig effektivitet.

c) Hva er input og output på hhv. Map og Reduce-funksjonene?

Input for Map-funksjonen er (key, value)-par, og output er List(key, value)

Input for Reduce-funksjonen er (key, Iterable(value)), og output er List(key, value)

Oppgave 2 – Spark – 20 % (alle deler teller likt)

Anta at man har en fil *countries.tsv* (tsv = tab-separert) som inneholder informasjon om land og karakteristiske termer for landene (i dette tilfellet kan dere anta en tab "\t" per linje, mellom land og term-listen):

```
France wine,art,trains,wine,trains
Canada glaciers,lakes,hockey,maple,grizzly
Norway fjords,fjords,glaciers,trolls
Japan trains,sushi,origami,sushi,fuji
Argentina wine,glaciers,football,glaciers
```

I denne oppgaven skal dere for hver av deloppgavene under vise hvordan de kan løses vha. Spark transformasjoner/aksjoner (Scala, Python eller Java).
Hint: Splitte en tekststreng x basert på "\t" (tab): val y = x.split("\t")

a) Lag en RDD med navn ”data” basert på filen, hvor hver post/objekt er en tekststreng (String)

inneholdende en linje fra filen.

countries_schema = StructType([StructField(“Country”), StringType(), False),

StructField(“CharacteristicTerms”, ArrayType(StringType())),

])

data = spark.read.format(“csv”) \

.option(“delimiter”, “\t”) \

.schema(“countries_schema”)

.load(“countries.csv”)

Fra løsningsforslag:

val data = sc.textFile("countries.tsv").

b) Lag en ny RDD ”data2” der hvert objekt i ”data” er en ”array/list of strings”, der første tekst streng er land og tekststreng nummer to inneholder termene. Eksempelresultat (Python):

```
[['France', 'wine,art,trains,wine,trains'],
 ['Canada', 'glaciers,lakes,hockey,maple,grizzly'],
 ['Norway', 'fjords,fjords,glaciers,trolls'],
 ['Japan', 'trains,sushi,origami,sushi,fuji'],
 ['Argentina', 'wine,glaciers,football,glaciers']]
```

```
val data2 = data.map(x => x.split("\t"))
```

c) Lag en ny par-RDD (pair RDD) ”data3” der nøkkel (key) er land, og verdi (value) er en “array/list of strings” av karakteristiske termer, eksempelresultat:

```
[('France', ['wine', 'art', 'trains', 'wine', 'trains']),
 ('Canada', ['glaciers', 'lakes', 'hockey', 'maple',
 'grizzly']), ('Norway', ['fjords', 'fjords', 'glaciers',
 'trolls']),
 ('Japan', ['trains', 'sushi', 'origami', 'sushi', 'fuji']),
 ('Argentina', ['wine', 'glaciers', 'football',
 'glaciers'])]
```

```
val data3 = data2.map(x => (x(0), x(1).split(",")))
```

d) Finn antall karakteristiske termer i datasettet (ikke inkludert navn på landene).

data3.flatMap(x => x._2).count()

e) Finn antall *distinkte* karakteristiske termer i datasettet.

data3.flatMap(x => x._2).distinct().count()

f) Basert på ”data3”, lag en RDD som inneholder antall distinkte termer for hvert land.
Eksempelresultat:

```
[('Argentina', 3), ('Norway', 3), ('France', 3), ('Canada', 5),
('Japan', 4)]
```

```
val data4=data3.flatMapValues(x => x).distinct().map(x => (x._1,
1)).reduceByKey((x,y)=> x+y).collect()
```

Oppgave 4 – MinHashing – 10 % (alle deler teller likt)

lx	Element	S 1	S2	S3	S4	lx2
0	p	0	0	0	1	3
1	a	0	1	1	1	2
2	g	1	1	0	0	1
3	i	1	1	1	1	0
4	k	0	1	0	0	4

a) Forklar ”shingling” og hensikten med ”shingling”.
Shingling er brukt for å dele inn et dokument i et sett basert på en gitt verdi for K. Dersom man for eksempel har et dokument og ønsker å bruke 2-shingles for ordene i dokumentet, finner man alle par av ord som kommer etter hverandre, og hver av disse parene blir en shingle. Eksempel med dokument d = “Jeg har eksamen nå”
2-shingles: {Jeg har, har eksamen, eksamen nå}

Hensikten med dette er å gjøre sammenligning enklere og mindre ressurskrevende. Man kan for eksempel definere at et dokument må ha minst en matchende shingle for å være like nok til videre analyse.

lx	Element	S 1	S2	S3	S4	lx2
0	p	0	0	0	1	3
1	a	0	1	1	1	2
2	g	1	1	0	0	1

3	i	1	1	1	1	0
4	k	0	1	0	0	4

b) Figuren ovenfor viser en forekomstmatrise for elementene p/a/g/i/k i settene S1/S2/S3/S4.
Hva er MinHash-signaturene gitt permutasjonene lx og lx2?

	s1	s2	s3	s4
h1	inf	inf	inf	inf
h2	inf	inf	inf	inf

	s1	s2	s3	s4
h1	inf	inf	inf	0
h2	inf	inf	inf	3

	s1	s2	s3	s4
h1	inf	1	1	0
h2	inf	2	2	2

	s1	s2	s3	s4
h1	2	1	1	0
h2	1	1	2	2

	s1	s2	s3	s4
h1	2	1	1	0
h2	0	0	0	0

	s1	s2	s3	s4
h1	2	1	1	0
h2	0	0	0	0

Oppgave 5 – Datastrømmer (streaming data) – 30 % (alle deler teller likt)

Anta at du er ansatt i et firma som analyserer interessetrender for det nye kameraet Sony A9 som Amazon nettopp har startet å selge. Firmaet ditt har bestemt seg for å gjøre dette basert på sosiale media data, inkl. Twitter og Facebook.

a) Gå ut fra at du starter med å bestemme deg for finne antall meldinger som nevner ”Sony A9” ved å bruke såkalte stående spørring (”Standing Query”). Forklar hva som er forskjellen(e) mellom ”standing query” og ”ad-hoc query”. Hvordan kan ”standing query” løse oppgaven?

Forskjellen mellom standing query og ad-hoc query er at ad-hoc spørres kun en gang som en vanlig spørring, mens standing query spørres kontinuerlig og til enhver tid.

Standing query kan løse oppgaven ved å spørre “hvor mange meldinger har nevnt ‘Sony A9’ så langt?”. Svaret på denne querien vil endres hver gang det kommer en ny melding som nevner kameraet.

b) Siden vi kan se sosiale media data som datastrøm har de også de andre karakteristikker og/eller utfordringer enn statiske data. Drøft hva disse er.

Disse karakteristikkene er at:

- I en datastrøm er det kontinuerlig generering av data’
- Det er snakk om veldig store mengder data
- Dataen er ubegrenset. Man vet ikke når eller om datastrømmen slutter.
- Har varierende format og struktur. kan for eksempel bestå av både tekst, bilder og videoer.
- Dataen kommer i høy hastighet.

Den største utfordringen med datastrømmer kommer av den enorme mengden og at den endrer seg svært raskt. Dette krever rask respons i sanntid som er en stor utfordring. En annen er at ettersom datastrømmer er ubegrensede mtp når de slutter, så kan man ikke lagre hele strømmen.

c) Som alternativ til ”standing query” velger du ”bit counting” for løse deler av oppgaven. Forklar hvordan kan oppgaven oversettes til ”bit counting”.

For å bruke bit counting kan man gjøre om strømmen til en bitstrøm, hvor meldinger som ikke inneholder “Sony A9” gir bitverdi 0, mens meldinger som gjør det gir bitverdi 1. Man kan da telle antall 1ere ved bruk av for eksempel DGIM-algoritmen for å finne ut hvor mange meldinger som inneholder “Sony A9”.

d) Analysen din skal se på trendene de siste 2 ukene og du ser for deg antall ”klikk” og ”kjøp” av produkter i Amazon direkte som en del av analysen. Se for deg at du skal finne ut hvor stor andel av ”klikk” og ”kjøp” er gjort på ”Sony A9”. Til dette formålet velger du nå å bruke glidende vindu-prinsippet (“sliding window”). Anta at dette vinduet har en størrelse på 500 ”klikk” og ”kjøp” tilsammen. Forklar hvordan du går fram for å beregne denne andelen.

For det aller første vinduet, dvs de 500 første verdiene i strømmen, er det bare å telle antall klikk og kjøp, og dele dette på total antall i vinduet (500).

Når det kommer en ny verdi inn i vinduet:

Dersom den nye verdien er klikk eller kjøp, må vi endre andelen med (1-j)/N, hvor N er antall i vinduet (500), og j er 1 dersom den eldste meldingen i vinduet inneholdt klikk eller kjøp, og 0 dersom den ikke gjorde det.

e) Når vi først er inne på ”klikk” drøft hvordan ”bloom filter” kan være nyttig til å finne antall klikk. Gjør de antakelsene du finner nødvendig.

Man kan ikke direkte bruke bloom filter for å finne totalt antall klikk, men man kan bruke det for å for eksempel filtrere ut klikk fra samme person. Dersom man ønsker å finne antall klikk fra distinkte brukere på en side, kan man for eksempel hashe på både URLen til siden og brukerIDen til brukeren som klikker, og på den måten sjekke om et klikk fra en bruker ikke har vært registrert før.

f) Bruk “bloom filter”-prinsippet til å fylle ut tabellen nedenfor

Strømelement	**Hash-funksjon - h_1**	**Hash-funksjon - h_2**	**Filtrere Innhold**
			00000000000
56 = 11 1000	100 = 4	110 = 6	00001010000
428 = 1 1010 1100	10010 = 18 => 7	1110 = 14 => 3	00011011000
875 = 11 0110 1011	11001 = 25 => 3	10111 = 23 => 1	01011011000

Hint: bruk *h1(x)=y1 mod 11* og *h2(x)=y2 mod 11* som hash-funksjoner, der verdiene av *y1 og y2* er hentet henholdsvis fra oddetalls-bits fra *x* og partalls-bits fra *x*.

Oppgave 6 – Anbefalingssystem og sosiale grafer (recommender systems and so cial graphs) – 20 % (alle deler teller likt)

Firmaet du er nyansatt i er spesialist på anbefalingssystemer. Din oppgave er å utvikle gode anbefalingsalgoritmer og metoder.

a) Med fokus på fordeler og ulemper sammenlikn ”collaborative filtering” mot ”content-based recommendation”.

Collaborative filtering bruker rating av et produkt gitt av brukere som har erfaring med produktet. Denne ratingen kan være både eksplisitt (rating 1-10) eller implisitt (feks ved å se på hvor lenge en bruker har sett på en youtube-video før han klikket seg bort).

Content-based recommendation baserer seg på å se på innholdet i produktet. Dersom produktet

man vil lage anbefalingssystem for er filmer, kan man da for eksempel se på regissør, skuespillere, språk, lengde osv.

Collaborative filtering har en tendens til å gi bedre resultater, altså mer riktige anbefalinger enn content-based recommendation. Derfor burde man bruke collaborative filtering der man har mulighet (særlig hvis brukerne kan rate produktene man vil anbefale). Det er likevel noen problemer med collaborative filtering. Såkalt cold start problem oppstår når man har enten et helt nytt produkt eller en helt ny bruker. Dersom det er snakk om et nytt produkt, betyr det at ingen har ratet det produktet enda, og man får derfor problemer med å utføre likhetsestimering. Dette ville ikke vært et problem dersom man brukte content-based recommendation, da man uansett ville hatt tilgang på informasjonen man trenger. Når det gjelder en ny bruker, så er problemet at man ikke vet noe om smaken til brukeren enda. En løsning på dette kan være å ha en generell og generisk anbefaling som viser produkter som de alle fleste liker, fram til man har samlet mer data fra brukeren.

SE BILDE UNDER

Collaborative Filtering	**Content-based Recommender**
Fordeler: - Fungerer på alle type produkt - Lett og intuitivt å sette opp basert bruker/produkt-matrisen	**Fordeler**: - Trenger ikke data på andre brukere - Kan gi anbefaling til brukere med unike smak - Kan anbefale nye og ikke populære produkt (ingen cold start på nye produkter) - Bakgrunnen for anbefaling kan forklares til bruker
Ulemper: - Cold-start-problemet: trenger nok brukere i systemet til å finne "match" - Sparsitet: veldig få produkt har ratings. Problemer med å finne nok bruker-ratings på produkt - First raters: Kan ikke anbefale produkt som ikke har ratings fra før - Popularitets bias: Kan ikke anbefale produkt til brukere med unike smak. Mest tendens til å anbefale populære produkt.	**Ulemper**: - Avhengig av god feature engineering, i.e., utfordrende med å hente ut features på alt - Anbefaling til nye brukere er avhengig av å kunne ha en god brukerprofil - Overspesialisering, i.e., anbefalingene har tendens til å være kun innen brukerens profil (lav grad av serendipity)

b) Anta at du vil bruke bruker-relasjoner, som feks. "friends" og "followers" fra sosiale media, til å bygge opp brukerprofiler. Slike relasjoner kan tegnes i en graf. Bruk dette som utgangspunkt til å svare på følgende spørsmål:

i. Forklar forskjellene på "*overlapping*" og "*non-overlapping communities"* i grafer. Hvordan kan vi enkelt finne ut om to grafer overlapper.

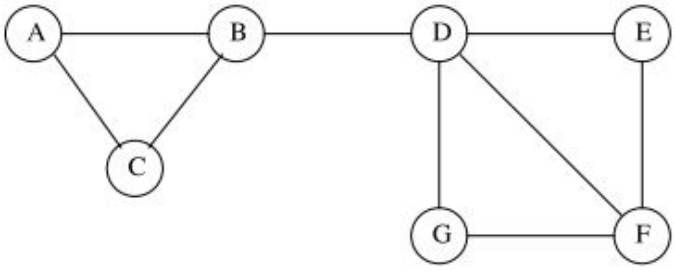
Forskjellen mellom overlappende og ikke-overlappende communities i grafer, er at i overlappende communities, så kan en node i grafen tilhøre flere communities (clusters), mens i ikke-overlappende communities, så kan en node i grafen maks tilhøre en community.

Vi kan enkelt finne ut om to grafer overlapper ved å bruke en "adjacency matrix" som er fylt med alle punktene i grafen, og se på kant-tettheten. Jo større tetthet av kanter, jo større er sannsynligheten for at grafene overlapper.

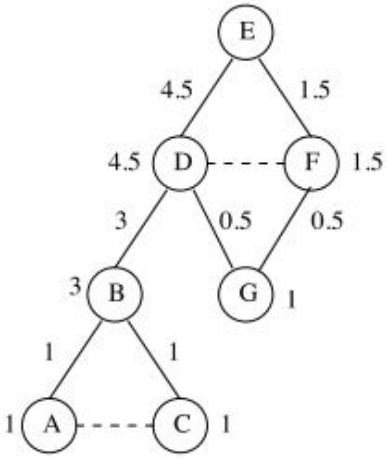
ii. Hva er hovedhensiktene med "community detection"? Forklar.

Hovedhensiktene med community detection er å finne noder som har noe til felles med hverandre. Dette kan for eksempel være personer på facebook som har mange felles venner, eller det kan være en gruppe med proteiner som interagerer med andre proteiner på en viss måte. Når man har funnet slike grupperinger, kan man finne ut hva de har til felles, og bruke dette til for eksempel videre analyse eller markedsføring.

c) Girvan-Newmans metode for beregning av "betweenness" er en vanlig metode innen graf-mining-teori. Vis hvordan du går fra Figur 1 til Figur 2 nedenfor ved å bruke Girvan-Newmans metode til å beregne "*betweenness*" i grafen i Figur 1.

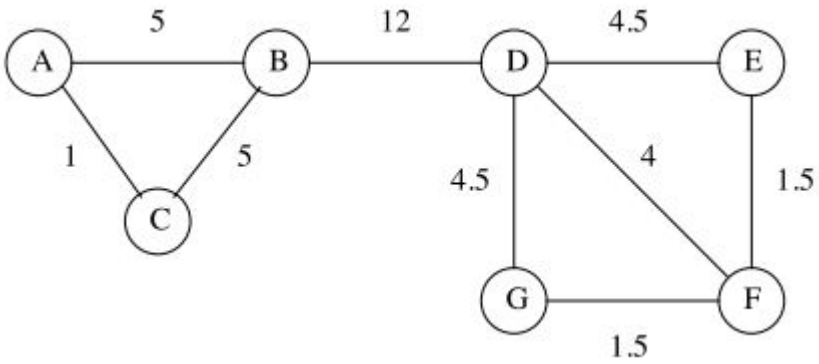


Figur 1: Start-graf



Figur 2: Steg 3 i første iterasjon.

d) Hvordan kan resultatfiguren nedenfor brukes til detektere ”communities”?



Denne figuren viser edge betweenness mellom nodene i grafen. For å detektere communities/ finne cluster, kan man i første steg fjerne den kanten med høyest edge betweenness, altså kanten mellom B og D. Alle noder fra og med b og til venstre for b må gå gjennom den kanten for å komme til D og alle noder til høyre for d. Det er derfor den kanten har høyest edge betweenness.

Framside

Department of Computer Science

Examination paper for TDT4305 Big Data Architecture

Academic contact during examination: Kjetil Nørvåg and Heri Ramampiaro
Phone: 41440433 and 99027656

Examination date: May 28th

Examination time (from-to): 1500-1900

Permitted examination support material: D: No tools allowed except approved simple calculator. Other information:

Students will find the examination results in Studentweb. Please contact the department if you have questions about your results. The Examinations Office will not be able to answer this.

1 Problem 1 – Hadoop – 10 % (all having same weight)

1. Explain two techniques that together make fault tolerance on data nodes (DataNodes) unnecessary.

1. Replikering. Replikering i HDFS går ut på at datablokkene blir lagret på mer enn en plass. Som oftest er det tre-veis replikering, som vil si at en datablokk vil bli lagret på tre forskjellige noder. Disse kan være på helt forskjellige lokasjoner. Replikering bidrar til feiltoleranse, ettersom dersom en node krasjer, så ligger fortsatt blokkene som var lagret på den noden på andre noder som kan håndtere forespørsler.

2. Sjekksummer. Den fungerer sånn at når en klient skriver til ei fil, så blir det også lagd en sjekksum som lagres som meta-data i den aktuelle datanoda. Når den samme fila skal leses igjen, så sjekker regner datanoda ut sjekksummen for det som ligger i fila og sammenligner med sjekksummen den har lagra i metadataen. Dersom de stemmer overens, så kan fillesingen fortsette, men hvis ikke så har det skjedd noe galt med fila, og man må da lese fila fra en annen replikasjon.

2. Explain *combiner* in MapReduce, and what is the purpose of this.

Hensikten med combiner i MapReduce er å redusere kommunikasjonskostnaden. Dette er aktuelt dersom man har en map-funksjon som produserer mange par med samme nøkkel. Da kan combiner-funksjonen utføre delvis sammenslåing av parene som har samme nøkkel, som oftest ved bruk av samme funksjon som Reduce. Dette kan minke datamengden man er nødt til å sende over nettverket.

3. What is the purpose of using data pipelines when writing blocks in HDFS?

NB! Ligger tekst fra læreren i nynorsk

Hensikten med å bruke data pipelines under skriving av blokker i HDFS er å redusere kommunikasjonskostnaden for klienten, som da kun trenger å sende dataen til ei node i stedenfor til tre (ved tre-veis replikering). Man flytter altså replikeringsbyrden vekk fra klienten. Og i tillegg der ein har fleire rack vil ein redusere kommunikasjon mellom rack (om to replika vert skrive til remote rack vil ein kun ha behov for ein inter-rack kommunikasjon).

2 Problem 2 – Spark – 10 % (all having same weight)

Tuddel is a subscription based streaming service for music. Information about all songs that are streamed is stored in a log, in order to be able to analyze, perform recommendations, and calculate royalties to artists. For each song streamed, a line will be generated in the log, in the following format (comma separated):
TimeStamp,UserName,Artist,SongName

Assume the following example dataset stored in the file streamed.csv:

1,u1,a1,s1
2,u2,a1,s2
3,u1,a1,s2
4,u3,a1,s1
5,u1,a2,s3
6,u2,a1,s2
7,u2,a1,s2
8,u2,a1,s2

This dataset has already been loaded into an RDD named s:

val s = sc.textFile("streamed.csv").map(_.split(","))

For each of the subproblems below, you should show how they can be solved using Spark transformations/actions (Scala, Python or Java).

1. Create an RDD containing number of songs streamed for each artist. Example results:
   (a2,1)
   (a1,7)

```
val new_rdd = s.map(a => (a(2),1)).reduceByKey(_+_)
```


2. Create an RDD that for each line has name of user and number of song he/she has streamed. Example results: u3 1
   u2 4
   u1 3

```
val new_rdd2 = s.map(a => (a(1),1)).reduceByKey(_+_)
```

3. Find number of *distinct* songs that have been played. Example results:
   3

```
val new_rdd3 = s.map(a => (a(3)).distinct().count()
```

Maximum marks: 10

4 Problem 4 – MinHashing – 10 %

Explain the main features of LSH (locality sensitive hashing) for documents, incl.:

1. The purpose of LSH.

Hensikten med LSH er å redusere antall par som må sammenlignes. Uten LSH kan dette ta altfor lang tid. LSH er en metode for å finne kandidatpar som er "like nok" til at de må sammenlignes med en likhetsfunksjon senere.

2. Input and output.

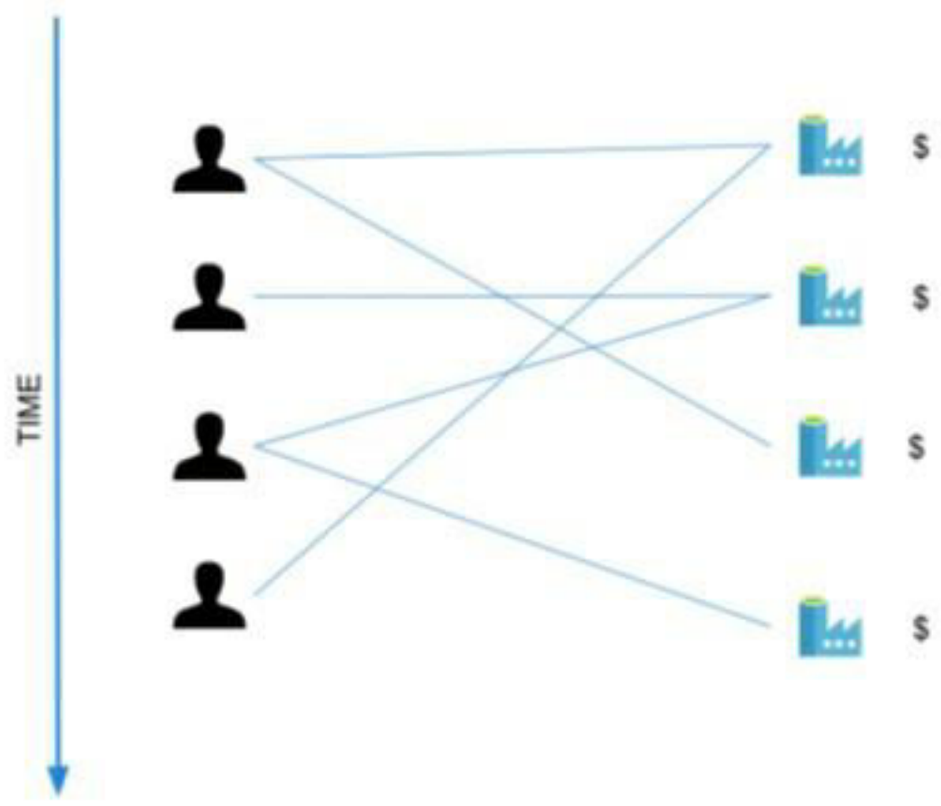
Inputen til LSH er en signaturmatrise, som er generert ved bruk av shingling og minhashing. Output er en liste med kandidatpar som må evalueres av en gitt likhetsfunksjon.

3. Contents of important data structure(s), and algorithm.

Algoritmen er i enkle trekk at man først deler opp signaturmatrisen i b antall bands (bånd), som alle inneholder r rader hver. Deretter går man gjennom hvert dokument for hvert bånd, og hasher disse delene av signaturene til buckets med et visst antall hash-funksjoner. Alle som ender opp i samme bucket etter at man har hashet alle signaturene i hvert band, er kandidatpar. Det at de er kandidatpar, betyr at de må evalueres.

Maximum marks: 10

5 Oppgave 5 – Adwords – 5 %

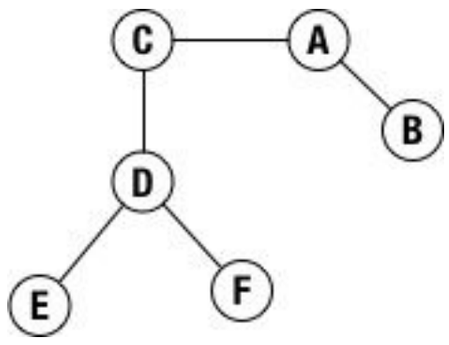


Given the figure above, explain the Adwords problem and a Greedy algorithm for solving this problem. What is the disadvantage of using this algorithm versus an optimal solution? A minor change to the greedy algorithm can improve results. Explain this change.

Fill in your answer here

6 Oppgave 6 – 15%

1. Explain briefly what steps you need to execute to find "communities" in a graph. Use the following figure to support your explanation. (6%)



2. Explain the main purpose of analyzing social graphs. (4%)

3. Explain the main differences between **Storm** and **Spark**. Explain the advantages of the **AsterixDB**'s Feed System has compared to both **Storm** and **Spark**. (5%)

Fill in your answer here

Maximum marks: 15

7 Oppgave 7 – 20%

1. a. What is "Bloom Filter" and what is it used for? Use an example to support your explanation. (4%)

   b. Use Bloom filter to complete the following table. Assume that we use h as a hash-function, and that it is defined as h(x)= y mod 11, where y is extracted from the odd number bits from xor even number bits from x. E.g., h1(39) = 011 = 3,
   h2(39) = 101 = 5, osv. (6%)

Strømelement	Hash-funksjon - h_1	Hash-funksjon - h_2	Filtrere Innhold
			000 0000 0000
85 = **101 0101**			
214 = **1111 1010**			
353 = **01 0110 0011**			

2. Assume that you will find fractions of unique queries from the last month in a stream of search queries. Due to space and time limitation, you cannot check all incoming queries and must use sampling and decide to read only every 10th of the questions (i.e. 10% sampling). Explain why this will not give the correct answer? What would be a more correct approach to sample the stream of search queries? (5%)

3. Imagine you are using social media like Twitter to analyze, e.g., trends. In particular, you are interested in knowing when a product is mentioned by people and how often they are mentioned. You propose using the bucket principle to count how many times this product is mentioned. Explain how the bucket principle on data stream works in this case. (5%)

Fill in your answer here

Maximum marks: 20

8 Oppgave 8 – 15%

1. So-called cold start is a challenge when using collaborative filtering in recommended systems? Explain what is meant by cold start and when this can occur. Further, discuss possible solutions for the cold start problem. (6%)

TDT4305

2. Assume the following user rating table:

USERS

PRODUCTS	1	2	3	4	5	6	7	8
1	1		3			5		
2			5	4			4	
3	2	4		1	2		3	
4		2	4		5			4
5			4	3	4	2		
6	1	P	3		3			2

Assume further that you will use "item-item collaborative filtering".

a. Explain what is the difference between item-item collaborative filtering and user-item collaborative filtering. (4%)

b. Explain how you will proceed to compute the predicted / estimated value of P. Make the assumptions you find necessary. (5%)

Fill in your answer here

Maximum marks: 15

Framside / Front page

Institutt for datateknologi og informatikk

Eksamensoppgave i TDT4305 Big Data Architecture

Eksamensdato: 23. mai 2020

Eksamenstid (fra-til): 09:00 – 13:00

Hjelpemiddelkode/Tillatte hjelpemidler: A / Alle hjelpemidler tillatt

Faglig kontakt under eksamen:
Tlf.: 99 02 76 56

Teknisk hjelp under eksamen: NTNU Orakel
Tlf: 73 59 16 00

ANNEN INFORMASJON:

Gjør dine egne antagelser og presiser i besvarelsen hvilke forutsetninger du har lagt til grunn i tolkning/avgrensing av oppgaven. Faglig kontaktperson skal kun kontaktes dersom det er direkte feil eller mangler i oppgavesettet.

Lagring: Besvarelsen din i Inspera Assessment lagres automatisk. Jobber du i andre programmer – husk å lagre underveis.

Juks/plagiat: Eksamen skal være et individuelt, selvstendig arbeid. Det er tillatt å bruke hjelpemidler. Alle besvarelser blir kontrollert for plagiat. *Du kan lese mer om juks og plagiering på eksamen her*.

Kildehenvisninger: Alle oppgaver skal besvares "med egne ord" for å vise forståelse.

Varslinger: Hvis det oppstår behov for å gi beskjeder til kandidatene underveis i eksamen (f.eks. ved feil i oppgavesettet), vil dette bli gjort via varslinger i Inspera. Et varsel vil dukke opp som en dialogboks på skjermen i Inspera. Du kan finne igjen varselet ved å klikke på bjella øverst i høyre hjørne på skjermen. Det vil i tillegg bli sendt SMS til alle kandidater for å sikre at ingen går glipp av viktig informasjon. Ha mobiltelefonen din tilgjengelig.

Vekting av oppgavene: Som vist i oppgavesettet. Alle deloppgaver innenfor en oppgave teller

likt. **OM LEVERING:**

Besvarelsen din leveres automatisk når eksamenstida er ute og prøven stenger, forutsatt at minst én oppgave er besvart. Dette skjer selv om du ikke har klikket «Lever og gå tilbake til Dashboard» på siste side i oppgavesettet. Du kan gjenåpne og redigere besvarelsen din så lenge prøven er åpen. Dersom ingen oppgaver er besvart ved prøveslutt, blir ikke besvarelsen din levert.

Trekk fra eksamen: Ønsker du å levere blankt/trekke deg, gå til hamburgermenyen i øvre høyre hjørne og velg «Lever blankt». Dette kan ikke angres selv om prøven fremdeles er åpen.

Tilgang til besvarelse: Du finner besvarelsen din i Arkiv etter at sluttida for eksamen er passert

[1] Oppgave 1

Oppgave 1 – Systemer for datastrømmer (15%)

1. Drøft prinsippene bak håndteringen av datastrømmer i AsterixDB. Hvordan skiller dette seg fra Spark og Storm?

HOVEDMOTIVASJON BAK ASTERIXDB

De eksisterende løsningene som eksisterer for big data og håndtering av store datamengder er såkalte lappeteppesystemer, det vil si systemer som kobler sammen mange ulike teknologier for å møte kravene til de forskjellige bruksområdene innen big data. For eksempel har man vanlige databasesystemer som MySQL og MongoDB, som i big data sammenheng ofte er koblet sammen med løsninger som Hadoop eller Hive. På den måten kan man håndtere blant annet lagring av data og håndtering av spørringer når det kommer til store datamengder på en måte som både er feiltolerant og skalerbar. Det at man ofte er nødt til å koble sammen mange ulike systemer og teknologier for å kunne utvikle en ønsket applikasjon er ikke optimalt. Dette er fordi det krever stor kompetanse om alle de ulike systemene og teknologiene, samt at det er tidkrevende å få alle delene til å virke sammen.

Tanken bak AsterixDB er å lage et såkalt “One Size Fits a Bunch”-system. Det vil si at det er et eget system som skal inneholde mye av funksjonaliteten som man ellers kun kan oppnå ved å koble sammen mange ulike systemer. Spesifikt gjelder dette funksjonalitet for parallelle databasesystemer, funksjonalitet for semi-strukturert datahåndtering, og funksjonalitet for data-intensiv beregning. I AsterixDB har man også tatt hensyn til at mange allerede er vant til å bruke systemer som Spark og Storm, så man har derfor mulighet til å bruke feks Spark sine grensesnitt, og samtidig få tilgang til all annen funksjonalitet som finnes i AsterixDB.

HVORDAN FUNGERER ASTERIXDB MTP INGESTION AV DATA

Datastrømmer som kommer fra eksterne kilder og går inn til persistent lagring i et Big Data Management System kalles feeds i AsterixDB. For å håndtere feeds, så har man noe man kaller Data Feed Management, som har som oppgave å vedlikeholde den kontinuerlige strømmen av data. Altså, når data skal injiseres inn i AsterixDB, henter man først data fra en ekstern datakilde (feks twitter API), og sender dette inn til en såkalt Feed Adaptor. Denne har som oppgave å konvertere dataen som kommer inn til det formatet som AsterixDB forstår, som kalles Asterix Data Model. Deretter blir dataen, som nå er på ADM-format enten pullet eller pushet inn i databasen. Dersom man bruker push, blir data lagt inn i databasen med en gang det blir tilgjengelig ved at det blir pushet inn til AsterixDB fra den eksterne datakilden, mens dersom man bruker pull, så henter man dataen fra den eksterne datakilden når det er spesifisert (feks etter en viss tid). Man kan også velge å prosessere feeden på andre måter enn bare å endre format til ADM før dataen legges inn i databasen. For eksempel kan man legge til felter med ekstra informasjon.

Det er også mulig å splitte dataen man får inn fra den eksterne kilden inn i flere feeds. Da er det mulig å prosessere hver av disse feedene parallelt. Den feeden som består av data som kommer direkte fra kilden kalles primary feed, mens feeds som får data fra andre feeds kalles secondary feeds. På denne måten kan man hente dataen bare én gang, men prosessere den mange ganger på mange forskjellige måter parallelt, og ende opp med flere resultat-datasett som lagres i databasen.

AsterixDB vs. both:

AsterixDB:
+ Et komplett system med innebygd effektiv lagringshåndtering av bigdata som også kan utvides med eksisterende lagringssystemer
+ Sofistikerte «recovery»-mekanismer som kan håndtere både «soft og hard failures»
+ Tillater å ta inn høyhastighetsdata med effektiv håndtering

---

+ Lar man håndtere data både i batch og ren strøm (sanntidsbehandling/analyse) form
+ Feed kan programmeres til å håndtere én input-strøm og produsere flere strøm av resultat-outputs («fetch once, compute many» prinsippet).
- Komplisert – kan være for mange moving parts, noe som gjør det vanskelig å vedlikeholde effektivt
- Proprietære spørringsspråk
- Ikke like utbredt som Storm og Spark

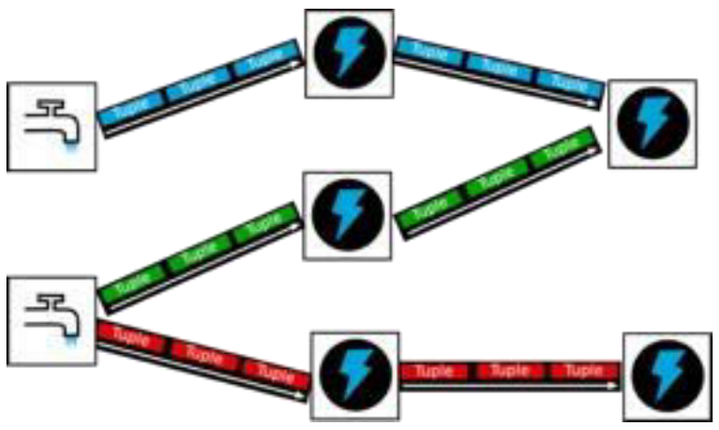
Spark:
+ Veldig enkel å ta i bruk
+ Kan kobles til standard og eksisterende systemer som kan gi mye fleksibilitet
+ Micro-batching kan gjør de veldig naturlig med «windowing» (aka. sliding windows)
- Bruker tupler som kan være litt vanskelig sammen Java
- Micro-batching gjøre det ikke rettfrem med håndtering og analyse i sanntid
- Uten innebygd naturlig datalagring og recovery-håndtering
- Fortsatt «single point of failure» i strømmen

Storm:
+ Veldig enkel å ta i bruk
+ Kan kobles til standard og eksisterende systemer som kan gi mye fleksibilitet
+ Sanntidsstrømhåndtering
+ Innbygd user-defined function-mulighet som gjøre det mulig å implementer funksjoner selv.
- Uten innebygd naturlig datalagring og recovery-håndtering
- Sanntidsfokus gjøre det litt vanskelig å implementer «windowing»

2. Vi skiller mellom tre forskjellige meldingsleveringsgarantier (Delivery semantics). Forklar hva disse tre garantiene betyr i praksis. Lag en tabell som forklarer hvilke garanti(er) som henholdsvis AsterixDB, Spark og Storm klarer å oppfylle. Til dette skal du bruke følgende mal:

	Garanti	**Eksempel**
AsterixDB		
Spark	Exactly once	Meldinger som sendes blir garantert levert. Dette krever komplekse implementasjoner, og kan føre til høy forsinkelse. Eksempel på bruksområde kan være finansielle applikasjoner hvor det er avgjørende at alle meldinger blir levert
Storm	At most once eller At least once	-At most once kan brukes hvis man skal analysere twitter-trender, og det ikke er viktig at absolutt alle tweets blir analysert -At least once kan brukes til for eksempel produktanalyse, hvor det er viktig å få med all data, og hastigheten ikke er avgjørende.

3. Forklar følgende figur av et Storm system:



Figuren viser topologien i Storm. Til venstre i figuren ser vi to spouts. En spout er en datakilde som genererer streams av tupler (altså datastrømmer). Figuren viser streams (markert med 3x “Tuple” som går ut fra spoutene, og inn til såkalte bolts (markert med lynsymbol i figuren). Disse boltene tar altså en stream som input, og prosesserer dataen på en eller annen måte (kan feks være filtrering, aggregering eller andre funksjoner), før de sender ut en ny stream med den prosesserte dataen som output.

Fra toppen på venstre side i figuren har vi altså en spout (datakilde) som sender ut en stream til en bolt. Denne bolten prosesserer streamen på en viss måte, og sender den prosesserte streamen videre til en ny bolt som igjen prosesserer den på en annen måte. Nederst til venstre har vi enda en spout som sender ut to streams. Den sendes opp til en bolt som prosesserer den og sender videre til den siste bolten som ble brukt av den øverste spouten. Den andre sendes ned gjennom de to boltene nederst på bildet, og blir altså prosessert i begge disse.

EKSTRA:

Storm arkitektur:

Man kan si at storm-arkitekturen består av 4 forskjellige nivåer. På øverste nivå har man master-noden, som kalles numbus. Oppgaven til master-noden er å koordinere alle aktivitetene i systemet. Neste nivå er for cluster-koordinering, og består av et antall noder som kalles zookeepers. Disse har altså til oppgave å koordinere clusterne i systemet. Tredje nivå består vanligvis av et høyere antall noder enn andre nivå, og disse nodene kalles Supervisors. Oppgaven til supervisors er å sette i gang worker-noder, som ligger på fjerde nivå. Disse worker nodene kjører selve arbeidsprosessene i systemet. Én supervisor kan både ha kontroll over én, flere, eller eventuelt ingen worker-noder.

Storms vs Spark:

En av de mest fundamentale forskjellene mellom Storm og Spark, er at prosesseringsmodellen i Storm er prosessering av hendelsesstrøm (event stream), altså at dataen kommer og prosesserers kontinuerlig. I Spark derimot, deler man opp dataen i “micro-batches”, og prosesserer en og en micro-batch.

En annen stor forskjell mellom Storm og Spark, er meldingsleveringsgarantiene de oppfyller. I storm kan man ha enten “At most once”, eller “At least once”. Man har altså mulighet til å spesifisere hvilken av de to meldingsgarantiene man vil bruke ut ifra hvilken use case man har, noe som bidrar til fleksibilitet. “At most once” betyr at meldinger blir forsøkt sendt bare én gang, og dersom det skjer en feil under sendingen av meldingen, så vil den ikke bli sendt på nytt. Det er altså mulighet for permanent tapte meldinger. “At least once” betyr at meldinger i utgangspunktet sendes bare én gang, men dersom det skjer noe som gjør at meldingen ikke blir levert, så vil systemet forsøke å sende den samme meldingen på nytt. Spark oppfyller meldingsleveringsgarantien “Exactly once”, som betyr at alle meldinger som blir sendt, garantert blir levert. Derfor trenger man aldri å sende meldinger på nytt. Man kan derfor si at pålideligheten er høyere i Spark enn i Storm, men Storm har mer fleksibilitet når det kommer til meldingsleveringsgaranti.

Neste forskjell er forsinkelsen. Storm har forsinkelse på delsekunder, altså under ett sekund, mens Spark har forsinkelse på sekunder. Spark har dermed høyere forsinkelse enn Storm, noe som blant annet henger sammen med at meldingsleveringsgarantien til Spark krever mer kompleksitet og tid enn Storm sin.

Det er også noen forskjeller i hvilke programmeringsspråk man kan bruke. Java, Scala og Python kan brukes med både Storm og Spark, men i Storm kan man i tillegg bruke Clojure og Ruby.

Siste hovedforskjell, som så vidt også ble nevnt i starten, er at dersom man ønsker å bruke batching av dataen, så må man ved bruk av Storm bruke andre verktøy til dette, mens i Spark er det native støtte for batching.

Generelt sett, så burde man bruke Storm når det er snakk om svært store mengder data som må analyseres i sanntid, og hastighet er mer viktig enn pålitelighet, mens man burde bruke Spark der hvor det er mindre krav til hastighet og mindre tilgjengelig lagringsplass, men større krav til pålitelighet, altså at alle meldinger kommer fram. Man burde også bruke Spark der hvor man tenker å bruke iterativ batch-prosessering. En iterasjon består typisk av Extract, Transform, og Load, som betyr at man henter data, transformerer den på et eller annet vis, og deretter laster den på nytt. Dette er høyst aktuelt i blant annet maskinlæring, hvor man først trener opp en modell, evt tester den, og så klassifiserer det som brukeren vil ha klassifisert. Altså er Spark svært anvendelig hvis man har en strøm av batch-data, og skal klassifisere dataen kontinuerlig og i sanntid. Man kan også bruke batching i Storm, men da må man bruke tredjepartsverktøy, som ikke er optimalt i forhold til å ha native støtte for batching. Det samme kan sies for Spark når det kommer til kontinuerlige strømmer. Da kan man simulere strømmer ved å lage batchene så små som mulig, men dette er heller ikke like optimalt som å ha native støtte for kontinuerlige strømmer dersom det er strømmer som passer best for den aktuelle use casen.

2 Oppgave 2

Oppgave 2 – Håndtering av datastrøm (15%)

REI Coop ønsker å få oversikt over kundene sine kjøpsvaner. De bestemmer seg derfor å lage et system som registrerer hvor ofte en spesifikk kunde er innom nettsidene deres og ser på deres produkt. Basert på dette skal du svare på følgende spørsmål. Gjør de antakelsene du finner nødvendige.

1. Drøft hvordan du ville løse denne oppgaven ved hjelp av stående spørring (standing queries).

En stående spørring er en spørring som stilles kontinuerlig og til enhver tid. I denne sammenhengen kunne man hatt en spørring som spør “hvor mange ganger har den spesifikke kunden sett på produkt X?”. Svaret på denne spørringen vil da oppdateres hver gang den aktuelle brukeren er inne på siden og ser på det spesifikke produktet X.

2. Kan denne oppgaven løses ved hjelp av «bit counting»? Forklar.

Man kan løse oppgaven ved hjelp av bit counting ved først å lage en bitstrøm hvor man har bitverdi 1 dersom det var den spesifikke kunden som var inne på nettsiden og så på produktet, mens man har bitverdi 0 dersom det var en annen kunde som så på produktet. Så kan man telle antall 1ere i strømmen, eller estimere antallet for eksempel ved bruk av DGIM-algoritmen.

3. Anta at du skal bruke buckets til å løse oppgaven. Hvordan fungerer buckets? Hvordan ville du gå frem for å løse oppgaven?

Buckets i DGIM algoritmen er brukt til å estimere antall 1ere i en bitstrøm. algoritmen bruker log2(N) bits for å representere et vindu av N bits. Hver bit som kommer inn i vinduet har en egen timestamp, hvor den første biten som kommer vil ha timestamp 1, neste vil ha timestamp 2 osv. Man deler vinduet inn i buckets som inneholder 1ere og 0ere. Alle størrelsene for buckets må være 2^x, altså blir første størrelse $2^0 = 1$, andre blir $2^1 = 2$, tredje blir $2^2 = 4$ osv. Dersom man har en bøtte av størrelse 4, betyr det at det er fire 1ere i bøtta. Biten lengst til høyre i bøtta må alltid være en 1er. Størrelsene på bøtta øker alltid fra høyre side av vinduet. Altså er den minste bøtta alltid lengst til høyre, og den største er alltid lengst til venstre. For hver størrelse, er det alltid maks 1 eller 2 bøtter. Så man kan ikke ha tre bøtter av samme størrelse. Ved bruk av alle disse reglene, ville jeg altså ha delt inn vinduet av bitstrømmen (hvor bitverdi er 1 dersom kunden som ser på produktet er den vi leter etter, og 0 hvis det er noen andre) inn i buckets. La oss si at lengst til høyre i vinduet så blir det to buckets av størrelse 1 (dvs det er én bit i hver av bøttene). Det kommer en ny bit. Hvis denne biten er 0 gjør vi ingen endringer, men hvis biten er 1, blir det 3 bøtter av størrelse 1, som ikke er tillatt. Da merger vi sammen de to eldste 1er bøttene til en bøtte av størrelse 2. Da kan den nye biten bli med i vinduet som en bøtte av størrelse 1. Dersom vi allerede hadde to bøtter av størrelse 2 før vi foretok mergingen, må vi fortsette prosessen videre til venstre i vinduet ved å merge de to bøttene av størrelse 2 sammen til en bøtte av størrelse 4. Dette må vi gjøre helt til vi har maks 1 eller 2 bøtter av hver størrelse.

Dette kan vi bruke for å estimere antall 1ere i de nyeste k bitene, hvor k < n. Dette gjør vi ved å summere alle bøttene som har slutt-timestamp mindre eller lik k bits i fortiden. Når det gjelder den siste bøtta, legger vi til bare halvparten av størrelsen, siden vi ikke vet hvor mange 1ere i den siste og største bøtta som fortsatt er i vinduet og ikke utenfor. På grunn av denne “gjettingen” for siste bøtte, får vi kun et estimat av antallet. Dette estimatet har en maksimal error på 50%, altså vil antallet vi estimerer alltid være innenfor +-50% av det eksakte antallet.

3 Oppgave 3

Oppgave 3 – Anbefalingssystemer (20%)

1. Drøft når du ville ha valgt å bruke «content-based recommendation» og når du ville ha valgt å bruke «collaborative filtering» som basis for anbefalingsmetode i et anbefalingssystem. Ta med fordeler og
ulemper med begge som en del av forklaringen din.

Collaborative filtering bruker rating av et produkt gitt av brukere som har erfaring med produktet. Denne ratingen kan være både eksplisitt (rating 1-10) eller implisitt (feks ved å se på hvor lenge en bruker har sett på en youtube-video før han klikket seg bort).

Content-based recommendation baserer seg på å se på innholdet i produktet. Dersom produktet man vil lage anbefalingssystem for er filmer, kan man da for eksempel se på regissør, skuespillere, språk, lengde osv.

Collaborative filtering har en tendens til å gi bedre resultater, altså mer riktige anbefalinger enn content-based recommendation. Derfor burde man bruke collaborative filtering der man har mulighet (særlig hvis brukerne kan rate produktene man vil anbefale). Det er likevel noen problemer med collaborative filtering. Såkalt cold start problem oppstår når man har enten et helt nytt produkt eller en helt ny bruker. Dersom det er snakk om et nytt produkt, betyr det at ingen har ratet det produktet enda, og man får derfor problemer med å utføre likhetsestimering. Dette ville

ikke vært et problem dersom man brukte content-based recommendation, da man uansett ville hatt tilgang på informasjonen man trenger. Når det gjelder en ny bruker, så er problemet at man ikke vet noe om smaken til brukeren enda. En løsning på dette kan være å ha en generell og generisk anbefaling som viser produkter som de alle fleste liker, fram til man har samlet mer data fra brukeren.

SE BILDE UNDER

Collaborative Filtering	**Content-based Recommender**
Fordeler: - Fungerer på alle type produkt - Lett og intuitivt å sette opp basert bruker/produkt-matrisen	**Fordeler**: - Trenger ikke data på andre brukere - Kan gi anbefaling til brukere med unike smak - Kan anbefale nye og ikke populære produkt (ingen cold start på nye produkter) - Bakgrunnen for anbefaling kan forklares til bruker
Ulemper: - Cold-start-problemet: trenger nok brukere i systemet til å finne "match" - Sparsitet: veldig få produkt har ratings. Problemer med å finne nok bruker-ratings på produkt - First raters: Kan ikke anbefale produkt som ikke har ratings fra før - Popularitets bias: Kan ikke anbefale produkt til brukere med unike smak. Mest tendens til å anbefale populære produkt.	**Ulemper**: - Avhengig av god feature engineering, i.e., utfordrende med å hente ut features på alt - Anbefaling til nye brukere er avhengig av å kunne ha en god brukerprofil - Overspesialisering, i.e., anbefalingene har tendens til å være kun innen brukerens profil (lav grad av serendipity)

2. Tabellen nedenfor viser hvordan 5 av deres brukere (U1 – U5) har «rated» filmene (M1 – M5) de ser på med verdi fra 0 til 10. Fyll ut tabellen for å finne ut hvilke av de andre brukerne som er mest lik bruker 4 (dvs. U4). For å finne ut denne likheten skal du bruke **Pearson correlation coefficient.**

User-ID /Movie-ID		**M 2**	**M 3**	**M 4**	**M 5**	**pearson(Ui, U4) (user-user)**
U1	7	6	7	4	5	-0.87
U2	6	7	0	4	3	-0.327
U3	0	3	3	1	1	0.089
U4	1	2	2	3	3	1
U5	1	0	1	2	3	0.681

Svar: Bruker 5 er mest lik bruker 4.

4 Oppgave 4

Oppgave 4 – Hadoop og Spark – 15 %

1. Forklar hvordan skriving til fil foregår i HDFS.

Når en klient skal skrive til fil i HDFS, vil den først kontakte navnenoda med informasjon om blokka som skal skrives. Navnenoda legger til blokka i katalogen sin, og returnere tilbake til klienten hvilke datanoder blokka skal lagres på. Ettersom man bruker replikering i HDFS, vil det være snakk om flere datanoder, og ikke bare én. Likevel trenger klienten bare å sende blokka til én datanode, som igjen sender videre til neste datanode, som igjen sender til neste osv. Dette kalles pipelining. Dersom man har flere racks, vil den første replikeringen av blokka bli lagret på det lokale racket, mens de to andre (ved tre-veis replikering) vil bli lagret i andre racks, ofte på andre lokasjoner. Dette er for å få feiltoleranse mellom racks, slik at hvis noe skjer med det ene racket så har man fortsatt dataen tilgjengelig på andre oppegående racks. Når en datanode har mottatt og skrevet blokka ferdig, gir den beskjed om dette til navnenoda sånn at navnenoda kan oppdatere det som må oppdateres i katalogen sin som feks sjekksummen for blokka.

2. Forklar konseptene "narrow dependency" og "wide dependency", og hvordan disse påvirker ytelse.

Narrow dependency:
Hver partisjon i parent-RDD brukes av maks én partisjon i child-RDD. Ettersom partisjonene derfor er uavhengig av andre partisjoner, kan disse operasjonene gjennomføres parallelt, og på samme maskin, og man trenger ikke kommunikasjon mellom noder. Recovery fra feil er også enkelt her, i og med at alt kan foregå på samme maskin. Altså er kommunikasjonskostnaden her veldig lav. Eksempler på transformasjoner som har narrow dependency er map og union.
Wide dependency:
Partisjonene i parent-RDD kan brukes av flere partisjoner i child-RDD. Derfor kan det hende at én partisjon i parent-RDD må sendes til flere av partisjonene i child-RDD, som krever kommunikasjon mellom noder. Dette øker kommunikasjonskostnaden, og påvirker derfor ytelsen negativt.

5 Oppgave 5

Oppgave 5 – MinHashing – 20 %

Forklar "shingles", "MinHashing", og LSH.

Shingling er brukt for å dele inn et dokument i et sett basert på en gitt verdi for K. Dersom man for eksempel har et dokument og ønsker å bruke 2-shingles for ordene i dokumentet, finner man alle par av ord som kommer etter hverandre, og hver av disse parene blir en shingle. Eksempel med dokument d = “Jeg har eksamen nå”
2-shingles: {Jeg har, har eksamen, eksamen nå}
Generelt er k-shingling prosessen hvor man konverterer en input (feks dokumenter) til sett med alle strenger av lengde k som finnes i inputen. Disse strengene kan for eksempel være characters, ord, eller noe annet.

Hensikten med dette er å gjøre sammenligning enklere og mindre ressurskrevende. Men selv med shingles er slike sammenligninger for ressurskrevende når man har mange nok dokumenter man skal sammenligne. Derfor bruker vi MinHashing på shinglene for å komprimere dem. Dette gjøres ved å sette opp en karakteristisk matrise, hvor kolonnene representerer set (som kommer fra shingling og representerer dokumentene), og kolonnene representerer elementer som er inneholdt i settene. Man setter en 1er i rad R og kolonne K dersom elementet for rad R er inneholdt i sett K, og 0 dersom det ikke er det. Deretter hasher man hver kolonne til en liten signatur på en sånn måte at dersom den hashede signaturen for to sett er like, så er settene også like. Man hasher altså for å redusere plassen det tar og tiden det tar å sammenligne, samtidig som man bevarer likheten. Fremgangsmåten er å bruke I antall forskjellige hash-funksjoner, og representere hvert set S med de I verdiene av Hmin(S) for disse I hash-funksjonene. På denne måten kan man estimere Jaccard-likheten mellom to set S og T. Hvis vi lar Y være antall hash-funksjoner som gir Hmin(S) = Hmin(T), så kan vi estimere Jaccard-likheten med Y/I, hvor I altså er det totale antall hash-funksjoner.

Dette reduserer tiden man trenger for å sammenligne dokumenter betraktelig. Likevel kan det likevel ta altfor lang tid, dersom man for eksempel skal sammenligne millioner av dokumenter. Det er der siste steg, LSH kommer inn. LSH brukes for å finne kandidatpar. Dette er par som er såpass like at likheten må evalueres eksplisitt. Dette reduserer antall par man må evaluere likheten til betraktelig. LSH tar inn en signaturmatrise, altså outputen fra MinHashing som input. Algoritmen er i enkle trekk at man først deler opp signaturmatrisen i b antall bands (bånd), som alle inneholder r rader hver. Deretter går man gjennom hvert dokument for hvert bånd, og hasher disse delene av signaturene til buckets med et visst antall hash-funksjoner. Alle som ender opp i samme bucket etter at man har hashet alle signaturene i hvert band, er kandidatpar. Det at de er kandidatpar, betyr at de må evalueres.

6 Oppgave 6

Oppgave 6 – Adwords – 15 %

Forklar "Balance-"algoritmen (i kontekst av Adwords).

GREEDY ALGORITME

Gir ad-plassen til hvilken som helst budgiver som byr mest

BALANCE ALGORITME

Gir ad-plassen til den budgiveren som har størst resterende budsjett

GENERALIZED BALANCE