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

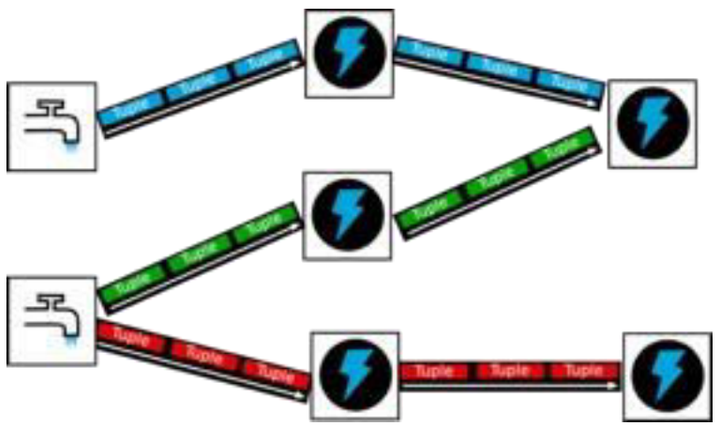
Spark:
+ Veldig enkel å ta i bruk
+ Kan kobles til standard og eksisterende systemer som kan gi mye fleksibilitet
+ Micro-batching kan gjør de veldig naturlig med «windowing» (aka. sliding windows)
- Bruker tupler som kan være litt vanskelig sammen Java
- Micro-batching gjøre det ikke rettfrem med håndtering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