

Big Data 2019



Portfolio assignment - part two

Mette Strand Hornnes

Innholdsfortegnelse

1. Choosing the right storage technology.....	3
1.1 Column Store Database - Apache Cassandra	3
1.1.1 Om Column Store Database & Cassandra:	3
1.1.2 Datasett	5
1.1.3 Spøringer på dette datasettet	5
1.1.4 Kilde til datasett.....	6
1.1.5 Ved annen bruk av dataene.....	6
1.2 Graph Database - Neo4j	6
1.2.1 Om Graph Database & Neo4j.....	6
1.2.2 Datasett – Facebook Data	8
1.2.3 Spøringer på dette datasettet	8
1.2.4 Kilde til datasett.....	9
1.2.5 Ved annen bruk av dataene.....	9
1.3 Key-value database - Riak	10
1.3.1 Om key-value database & Riak	10
1.3.2 Datasett	11
1.3.3 Spøringer på dette datasettet	12
1.3.4 Kilde til datasett.....	13
1.3.5 Ved annen bruk av dataene.....	13
1.4 Document Store Database - Mongo DB.....	14
1.4.1 Om Document Store Database & Mongo DB	14
1.4.2 Datasett	14
1.4.3 Spøringer på dette datasettet	14
1.4.4 Kilde til datasett.....	14
1.4.5 Ved annen bruk av dataene.....	14
2. Practical use of one of the storage technologies.....	15
2.1 Oppsett av Neo4j.....	15
2.2 Importere datasettet inn I database	16
2.3 Egendefinert spørring.....	17
2.4 Adding.....	19
2.5 Removing	20
2.6 Modifying.....	21
2.7 Ekstra	21
3. Kildeliste:.....	24

1. Choosing the right storage technology

For å kunne besvare oppgavene i denne delen av porteføljen best mulig har jeg valgt å for hver databasemodell beskrive hvordan denne fungerer, fordeler, samt bruksområder. Deretter vil oppgavene for datasettet besvares

1.1 Column Store Database - Apache Cassandra

1.1.1 Om Column Store Database & Cassandra:

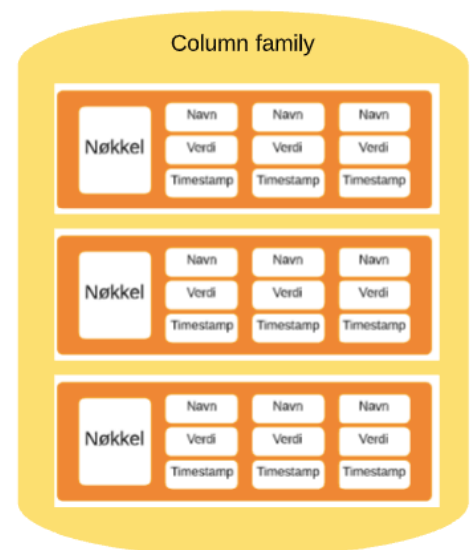
I motsetning til vanlig database hvor data ligger radvis, vil data i column store-database ligge i kolonner som er lagret hver for seg. Dette gjør blant annet operasjoner som krever aggregering veldig raskt, da man kun trenger å lese utvalgte kolonnene istedenfor for hele raden.

Komponenter:

Denne type database benytter et konsept kalt keyspace som er et skjema for hvor dataene befinner seg. Dette keyspacet inneholder alle Column families som inneholder rader med de ulike kolonnene.

Hver rad har en unik nøkkel som identifiserer denne raden.

Hver kolonne for hver rad inneholder navn, verdi og timestamp for dato og tid for når data ble lagret.



Fordeler med Column Store DB:

- Kan komprimeres - Da hver kolonne kan ha hver sin datatype
- Raskt med aggregeringsoperasjoner – Da det er raskt å hente data fra ulike kolonner
- Enkelt å skalere – Velig lett å kun legge til en kolonne om dette trengs

Egner seg til:

- Applikasjoner som er geografisk lagret utover flere datasentre
- Applikasjoner som kan tåle noe inkonsistens i replikaene for en kort tid
- Applikasjoner som krever lagring av store volum med data (hundrevis av TB)

Vanlige bruk:

- Sikkerhetsanalyser som benytter nettverkstrafikk
- Big Science
- Markedsanalyse
- Nettskalering – Som søk

Apache Cassandra - Eksempel på en Colum Store DB

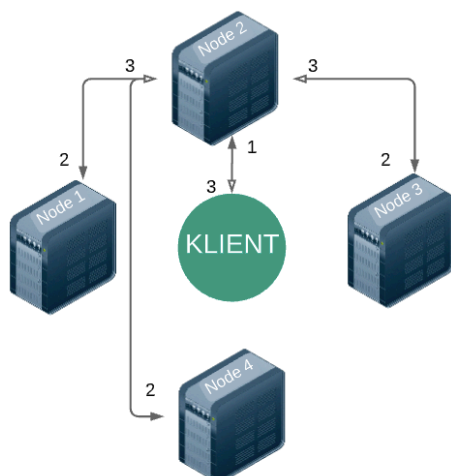
Cassandra er et distribuert databasesystem som er designet for å håndtere store volum med strukturert data. Dataene som brukes kan være fordelt utover ulike maskiner. Dette gir høy skalerbarhet, gjør det mulig å håndtere store mengder data raskt (mye raskere enn for eksempel MySQL), samt gir høy tilgjengelighet. Cassandra er dermed laget med tanke på at feil på en maskin (software eller hardware) kan oppstå og hindrer derfor «Single point of failures» (Guru99, u.å).

Arkitektetur

Arkitekturen til Cassandra består kun av noder (maskiner/servere) eller clustere (samling av noder) og ingen master. Alle nodene spiller samme rolle og er selvstendige samtidig som de er koblet til de andre nodene med et peer-to-peer-system. Hver node kan skrives til eller leses fra uavhengig av hvor dataene ligger, noe som også hindrer Single point of failures.

Med en såkalt gossip-protokoll deles informasjonen om data gjennom clusteret hvert sekund. Det lages også kopier (replicas) av dataene som legges på andre noder, noe som igjen er med på å hindre feil og tap av data (IntelliPaat, 2019).

Skrijving av data skjer på følgende måte:



1. Klienten skriver til en av nodene
2. Fra denne skrives replicas til andre noder (antall bestemmes av replication factor som settes ved oppsett)
3. Nodene bekrefter lagring tilbake til den første noden, som igjen bekrefter tilbake til klienten om at lagring er fullført (Acknowledge-melding).
4. Når klienten mottar denne ACK-meldingen skrives info om datalagringen til en commit-logg.

1.1.2 Datasett

For Column store database har jeg valgt følgende datasett:

Supermarket sales - Historical record of sales data in 3 different supermarkets

Link til datasett:

<https://www.kaggle.com/divyeshardeshana/warehouse-and-retail-sales>

Hvorfor denne datamodellen?:

Dette datasettet inneholder informasjon over salg gjort i ulike supermarkeder. Da mye av denne informasjonen vil være nyttig for å finne ut ting som hvordan pris påvirker salget, hvor og hva som selges vil column store db være den riktige måten å lagre denne dataen på. I motsetning til ved en tradisjonell Row-stored database hvor man måtte ha lest mye data som ikke er av interesse, vil man enkelt kunne hente ut kun de kolonnene som er av interesse for å få den markedsanalysen man ønsker. På grund av Column stored DB sine sin gode evne til å utføre aggregeringsoperasjoner ville lagring og håndtering av slik type data passet denne. Med slike dataer vil det også hende at ny informasjon om dataen må legges til. Dette gjøres enkelt i denne type databasemodell ved å kun legge til en ny kolonne (Columnar Database, u.å).

1.1.3 Spøringer på dette datasettet

Typiske spøringer på dette datasettet kan være:

- 1 Hvilken måned ble det totalt solgt mest?
- 2 Hvilke varer blir det solgt mest av?
- 3 Hvor mye blir det gjennomsnittlig solgt for hvert år?

Her vil ønsket varekode, måned eller år sendt med som parametere for å få svar på ønskede spøringer. Da Column stored DB (med for eksempel databasemotoren Cassandra) benyttes for denne dataen trengs det kun å leses fra kolonnene år, måned og varekode (avhengig av hvilke av spøringene over som blir gjort). Deretter kan man enkelt benytte aggregeringer (som «sum» og «avg») på disse kolonnene for å få det resultatet man ønsker. Da det å bruke

slike aggregeringer er en av styrkene til Column stored DB passer denne til å kunne svare på slike spørsmål og kunne få en analyse av salg.

1.1.4 Kilde til datasett

For dette datasettet er det ikke beskrevet noe kilde til hvordan dataene her blitt hentet ut. En mulig kilde til denne dataen kan være fra et varehus sitt elektroniske ekspedisjons- eller kassesystem over solgte varer.

Prosessen for å produsere/innhente denne dataen passer denne typen av databasemodell, da dataen enkelt kan hentes ut og tilpasses til en kolonnebasert modell.

Da det ikke står beskrevet noe nærmere bruksområder for dataene vil jeg igjen nevne det samme som gjort over, altså at datasettet er nyttig til markedsanalyse. Å kunne få oversikt over ting som totalt solgt, gjennomsnitt av salg og hvor det selges mest er noe man absolutt ville kunne bruke dette datasettet til. Som nevnt over vil denne typen databasemodell være passende til dette på grund av sin evne til å lese kun utvalgte kolonner.

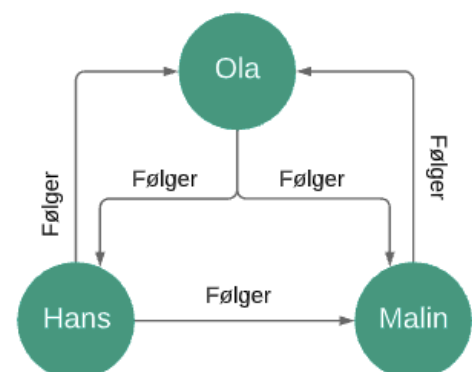
1.1.5 Ved annen bruk av dataene

Jeg tenker at Column Based DB egner seg godt til langtidslagring og månedsanalyse av dataene. For lagring og analyse av data dag til dag, eller flere ganger om dagen kan det tenkes at denne lagringsmetoden vil være for resurskrevende for å lagring store mengder data. Om man ønsker å raskt kunne hente ut data mange ganger om dagen, vil muligens key-value være et bedre alternativ.

1.2 Graph Database - Neo4j

1.2.1 Om Graph Database & Neo4j

Med Graph Databaser er data representert som noder og kanter. Hver node i databasen representerer en enhet (For eksempel en person på Twitter) med informasjon i key-value par og kantene representerer relasjonen mellom disse (For eksempel er «venn til»). Disse relasjonene er viktig i Graph databaser. Disse relasjonene



gjør det lett og med korte spørringer mulig å få ut data som med RDBMS ville krevd kompliserte spørringer med kanskje flere joiner.

Ulike typer grafer innenfor denne:

Grafene kan se og oppføre seg ulikt. Juan F. Cía nevner følgende i sin internettartikkel fra 2015:

- Undirected graphs: Noder og relasjoner er utskiftbare og går begge veier. Venner på Facebook er et eksempel på denne typen
- Directed graphs: Noder og relasjoner kan ikke gå begge veier. Et eksempel på dette er følgere på Twitter hvor en kan følge en annen uten at denne personen følger tilbake.
- Graphs with weight: Relasjoner mellom noder er vektet noe som gjør det mulig å utføre noen operasjoner før eller etter andre.
- Graph with labels: Denne typen har en slags merkelapp som forteller hva slags node der er, samt relasjon det er mellom noder. Eksempel:
 - En node kan være en Person, en forballspiller og Student
 - Relasjoner kan være venner, kollega, søster osv.
- Property graph: Den mest komplekse typen hvor vi kan tildele noder og relasjoner ulike egenskaper. Eksempel:
 - En node Person har egenskaper som id, navn, alder osv. som legges om key-value par
 - Relasjonen kan ha id.

Fordeler med Graph Database:

- Høy og stabil ytelse (Spesielt når det kommer til data som skal være koblet sammen):
 - Graph DB er rask da kun den delen av databasen som trengs for å respondere på spørringen trenger å traverseres.
 - Ytelsen holder seg som oftest til tross for økning av data.
- Fleksibelt:
 - Lett å endre innhold og relasjonen mellom dataene
 - Passer til den smidige måten systemer bli laget på i dag, ved at ting kan endres underveis.
- Lett å gjøre spørringer: Ingen joiner.

Egner seg til:

Graph Databaser egner seg meget godt til graflignende spørringer, som for eksempel å finne korteste vei mellom to eller flere elementer.

Neo4j - Eksempel på en Graph Database

Neo4j er den ledende Graph Database – motoren. Denne benytter altså grafer for å representere data og relasjonen mellom disse og gjør det lett, med språket Cypher (CQL), å gjøre spørringer (Stackshare, u.å).

1.2.2 Datasett – Facebook Data

For Graph Database har jeg valgt følgende datasett:

Facebook Data - Exploratory Data Analysis giving insights from Facebook dataset

Link til datasett:

<https://www.kaggle.com/sheenabatra/facebook-data>

Hvorfor denne datamodellen?:

Dette datasettet inneholder informasjon om ulike personer, samt en relasjon til andre. Ved å lagre dette som en Graph DB ville det vært enkelt å lage hver person som en node, sette informasjonen som navn, alder og fødselsdato som egenskaper, samt kunne sette ulike relasjoner til andre noder. Det vil være naturlig å hente informasjon om venner og venners venner ut ifra et slikt datasett, noe som Graph DB løser enkelt. Både ved å gjøre spørringen mindre avansert og raskere da det kun trenger å gå igjennom delen av databasen som kreves for å kunne svare på spørringen.

Da Graph DB gjør det enkelt å endre på innhold og relasjoner mellom noder passer dette slik type data bra, da denne dataen ofte kan endres ved at man for eksempel får nye venner.

Facebookdata er også noe som typisk vil kunne vokse ganske raskt i størrelse. Dette er også noe Graph DB håndterer fint uten at det påvirker ytelsen.

1.2.3 Spørringer på dette datasettet

Typiske spørringer på dette datasettet kan være (x er en node:person i datasettet)

1. Hvem er x sine venner
2. Hvem er venner til x sin venner

3. Hvem av x sine venner som «liker VG»

Her blir id-en til x, samt i den siste spørringen september, sendt med som parametre.

Graph DB løser en slik spørring enkelt ved å gå til noder som med relasjonstype «venner» er koblet til node x, og eventuelt videre fra disse nodene videre.

1.2.4 Kilde til datasett

Datasettbeskrivelse

This exploratory data analysis gives insights from Facebook dataset which consists of identifying users that can be focused more to increase the business. These valuable insights should help Facebook to take intelligent decision to identify its useful users and provide correct recommendations to them.

Som beskrevet i beskrivelsen er kilden til dette datasettet Facebook.

Proessen som har produsert dataen gjør det enkelt å opprette relasjoner mellom brukere, noe som gjør Graph DB godt egnet.

Som beskrevet over vil dette datasettet hjelpe til med å kunne gi riktig anbefalinger til riktige brukere. For å gjøre dette kreves det at databasen raskt kan søke igjennom noder (brukere) som har en relasjon til ulike sider eller andre personer. Noe som, også nevnt over, Graph DB gjør.

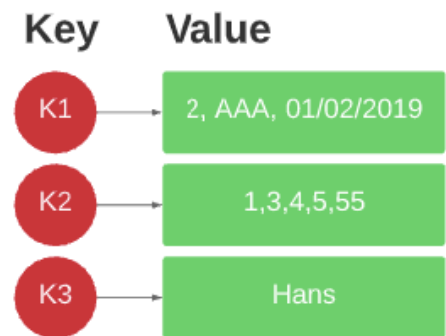
1.2.5 Ved annen bruk av dataene

Denne type database har, som nevnt over, høy ytelse spesielt når det kommer til data som skal være koblet sammen. Den egner seg godt til å utføre avanserte spørringer hvor disse relasjonene må knyttes sammen. Ved langtidslagring og analyse vil det ofte være et større behov for å utføre spørringer med aggregeringer (for eksempel addisjon eller å finne gjennomsnitt av ulike egenskaper). I slike tilfeller vil en Column-based DB ha høyere ytelse og derfor være bedre egnet

1.3 Key-value database – Riak

1.3.1 Om key-value database & Riak

Key-value database (også kjent som key-value store eller key-value store database) er en type NoSQL-database som benytter seg av unike nøkler og verdier for å lagre data (Ian, 2016). I verdiene kan alt som passer applikasjonen lagres. Dette er den enkleste databasemodellen da den ikke er et skjema slik som enkelte andre databaser er, men er mer som en assosiativ array. Nettopp på grund av denne enkeltheten til modellen er det heller ikke mulig med avanserte spørringer. Man kan kun nå data via nøkkelen og nøkkelen bestemmer hvor dataen vil lagres.

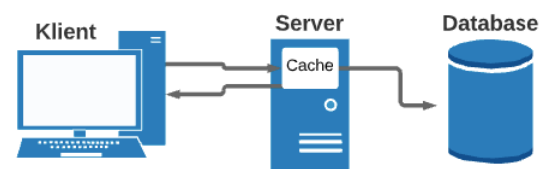


Få nøkkelen fra hashfunksjoner

En kjent måte å få genert en nøkkel er å kjøre verdien igjennom en hashfunksjon. Med hashing-algoritmen vil man få ut en hashcode, som ofte baserer seg på flere av verdiene som skal tilknyttes denne. De samme verdiene inn vil alltid produsere den samme nøkkelen og forespørselen vil bli håndtert på samme server. Dette hindrer at flere kan lese/skrive til samme sted samtidig.

Fordeler med Key-value Database:

- Rask
 - Key-Value er kjent for å være rask på grunn av sitt enkle assositive array-lignende oppsett.
 - En av årsaken er dens evne til å holde data i minnet. Data leses kun inn fra disk første gang og ved oppdateringer, mens ellers kun fra minne.
 - Dette krever at minne frigjøres før ny data kan lagres. Det er ulike algoritmer som håndterer dette, hvor den vanligste er LRU (Least Recently Used) da det antas at data som ikke nylig er brukt er minst sannsynlig vil brukes igjen.



- Enkelt
 - Om man ikke har behov for avanserte modeller med joiner og henting av data fra flere kolonner, passer denne modellen godt og gjør det ikke mer avansert enn det behøver å være.
- Fleksibelt
 - Da Key-value databaser er fleksible passer den veldig godt om datatyper skal endre seg ofte, eller om det trengs støtte for ulike datatyper for samme attributt.
- Skalerbarhet
 - Key-value databaser lar seg lett skalere horisontalt.
 - Dette gjøres enten med såkalt Master-slave replication hvor det legges til flere servere som kan svare på forespørsler (via en master-server) eller med Masterless replication hvor de ulike nodene skriver til andre noder (antall etter replication factor).

Egner seg til:

Key-value databaser egner seg godt til use cases hvor det kun trengs å nå id-en for å hente ut dataene eller om datatyper endres ofte. Den lagrer verdier som «blob» (Binary Large Object) og bryr seg derfor ikke noe om hva som lagres.

Typiske use cases for denne type lagring er:

- Lagring av Sessions

Her har hver bruker en unik id som brukes for å gjøre et oppslag på dataen.

- Handlekurv

Ved netthandel kreves det ofte at databasen kan håndtere millioner med lagring og forespørsler per sekund. Da Key-value både er skalerbart og rask egner denne seg til dette formålet.

(Ripon, Lecture 22 Key Value Database, 7. november 2019)

1.3.2 Datasett

For Key-value Database har jeg valgt følgende datasett:

Random Shopping cart - Contains a list of products separated in carts

Link til datasett:

<https://www.kaggle.com/fanatiks/shopping-cart>

Hvorfor denne datamodellen?:

Som nevnt over egner Key-value databaser seg godt til lagring av handlekurver.

Lagring av handlekurver krever ofte at databasen takler enormt mange forespørsler og endringer per sekund, ofte samtidig av flere brukere. Det er også et krav at data ikke mistes data selv om en node skulle gå ned. Dette gjør at dette datasettet, som inneholder handlekurver med varer, er passende for denne type database (Reeve, 2013).

Dette datasettet inneholder handlekurver med id som kan settes som key, samt en rekke kolonner med varer som kan settes som value. Dette er altså ikke salg som enda er gjort, men en database som holder på en liste med varer frem til kjøpet blir gjennomført. Den må derfor ikke sammenlignes med datasettet valgt for Column Store Database hvor jeg valgte et datasett over salg og hvor man ofte ønsker å gjøre aggregeringer på ulike kolonner. Dette er ikke noe som ville vært mulig med Key-value database.

Et eksempel på lagring vil se slik ut:

Key : Value

Handlekurvid : {"shampoo", "hand soap", "waffles", "cheese", "hand soap"}

1.3.3 Spøringer på dette datasettet

Spørningen man kan gjøre på dette datasettet og med key-value databaser generelt er ganske begrenset da man kun kan nå data via Key-en.

Spøringer som vil kunne være nyttig å utføre på dette datasettet handler om å få ut informasjon om verdier som er lagret for hver handlekurv.

En måte å utføre spøringer til key-value database er med et HTTP Interface (Ofte kalt RESTful). Med REST benytter man ulike metoder for å utføre handlinger. Disse metodene (også kalt verb) bestemmer hva som skal gjøres med path og argumenter som blir sendt med.

En oversikt over noen av Metodene:

POST: For opprettelse av data

GET: For lesing av data

PUT: For oppdatering eller erstatte data

DELETE: For å slette data

(REST API Tutorial, u.å)

Typiske spørringer på dette datasettet kan være (x er en node:person i datasettet)

1. Hva er innholdet i handlekurv med key x
2. Vis liste med alle key til alle handlekurvene
3. Slett handlekurv med key x

For visning av både innhold i handlekurv og alle keyene benyttes metode GET. For å vise innhold sendes id til handlekurv (x) med som parameter. For visning av keyene sendes det med et såkalt query ?keys=true :

For sletting av handlekurv vil man benytte metode DELETE og sende med key for handlekurv (x).

1.3.4 Kilde til datasett

Prosessen som har laget dataene er ikke beskrevet under beskrivelse for dette datasettet, annet enn at det er tilfeldig laget. En prosess for å produsere akkurat dette datasettet kan for eksempel være lagring fra et nettsteds handlekurv for ulike brukere. Prosessen som har gitt oppgav til dataen kan tenkes å generere store mengde data raskt. Dette er noe som, beskrevet over, Key-value DB håndtere fint.

Samtidig vil bruken av dataen kreve rask og ofte simultan prosessering, noe denne typen databasemodell med sin enkle måte å lagre data på vil utføre på en god måte. Da den heller ikke har krav til datatype som lagres i value forenkler dette lagringsprosessen.

1.3.5 Ved annen bruk av dataene

Denne type database har høy ytelse når data må lagres, hentes eller prosesseres raskt. Ved dag-til-dag-bruk vil derfor egne seg godt. Motsatt vil den ikke være særlig effektiv til analyse og ved langtidslagring da man kun når dataene via Key-en. Samtidig er det lite struktur på dataene som ligger lagret i value noe som gjøre det mer utforende å analysere dette. For slik langtidslagring og analyse vil Column Based DB være et bedre alternativ.

1.4 Document Store Database - Mongo DB

1.4.1 Om Document Store Database & Mongo DB

Foreløpig ikke besvart

1.4.2 Datasett

Foreløpig ikke besvart

1.4.3 Spøringer på dette datasettet

Foreløpig ikke besvart

1.4.4 Kilde til datasett

Foreløpig ikke besvart

1.4.5 Ved annen bruk av dataene

Foreløpig ikke besvart

2. Practical use of one of the storage technologies

Jeg har valgt benytte Database motor Neo4j, med følgende datasett som nevnt over:

<https://www.kaggle.com/sheenabatra/facebook-data>

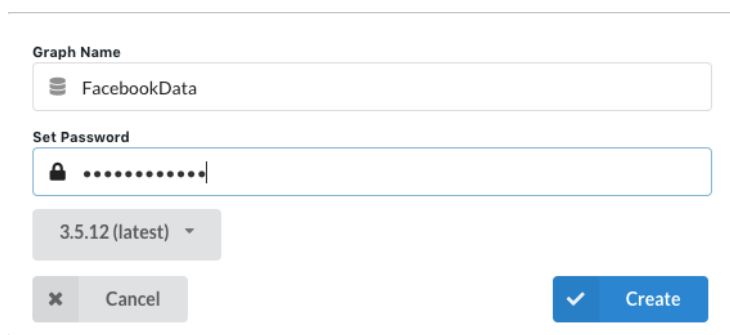
Dette er et datasett som inneholder informasjon om noen Facebookbrukere.

2.1 Oppsett av Neo4j

Jeg har lastet ned Neo4j Desktop som jeg brukes for å laste inn data og kjøre spørringer i.

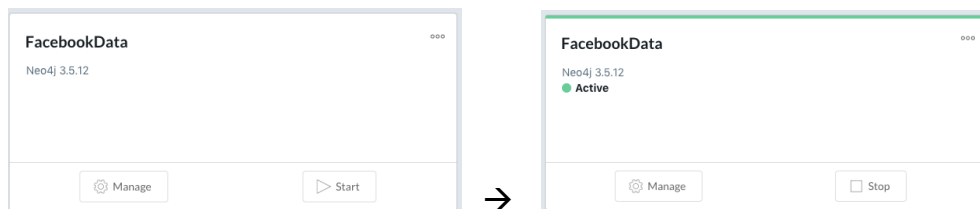
For å få opp og gå måtte jeg få kontakt med Neo4J-serveren.

1. Dette gjorde jeg ved å først legge til en ny graf

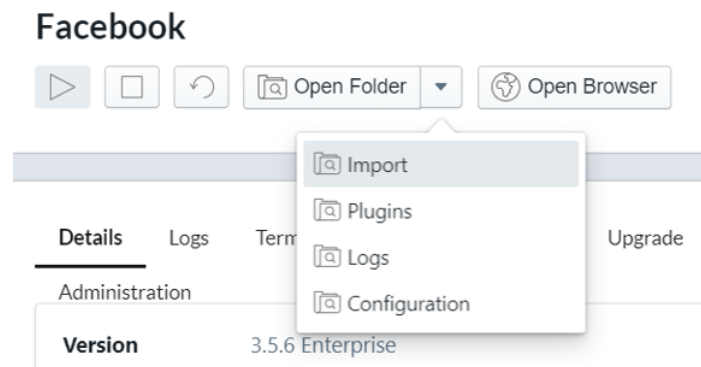


The image shows the 'Create Graph' dialog box in Neo4j Desktop. It has a 'Graph Name' field with 'FacebookData' entered. Below it is a 'Set Password' field with a masked password. A version dropdown shows '3.5.12 (latest)'. At the bottom are 'Cancel' and 'Create' buttons.

2. Navn på graf, samt passord er satt starter jeg opp denne grafen:



3. Da jeg ønsker å få inn data fra en CSV-fil importere jeg denne til Neo4j Desktop.



4. Når dette er klart kan jeg nå gå til "Neo4j Browser" og nå databasen på følgende adresse: bolt://localhost:7687 med brukernavn neo4j og passord satt tidligere.

2.2 Importere datasettet inn I database

LESER INNHOLD FRA DATASET

Her starter jeg med å sjekke at datasettet blir korrekt lest inn og at man får et resultat.

```
LOAD CSV WITH HEADERS FROM 'file:///pseudo_facebook.csv' AS row
RETURN row
```

Får da følgende resultat:



OPPRETTE NODER AV DATASET

Når jeg ser at dette er ok kan jeg opprette noder av dataene:

```
LOAD CSV WITH HEADERS FROM 'file:///pseudo_facebook.csv' AS row
CREATE (p:Person {
gender: row.gender,
dob_month: toInteger(row.dob_month),
userid: toInteger(row.userid), mobile_likes:
toInteger(row.mobile_likes),
dob_year: toInteger(row.dob_year),
www_likes: toInteger(row.www_likes),
www_likes_received: toInteger(row.www_likes_received),
mobile_likes_received: toInteger(row.mobile_likes_received),
friendships_initiated: toInteger(row.friendships_initiated),
likes_received: toInteger(row.likes_received),
friend_count: toInteger(row.friend_count),
```



```
tenure: toInteger(row.tenure),  
age: toInteger(row.age),  
dob_day: toInteger(row.dob_day),  
likes: toInteger(row.likes)}  
)
```

Når jeg kjører denne får jeg beskjed om at 99003 noder er opprettet

```
$ LOAD CSV WITH HEADERS FROM 'file:///pseudo_facebook.csv' AS row CREATE
```

```
Added 99003 labels, created 99003 nodes, set 1485043 properties, completed after 4335 ms.
```

SETTER AT ID SKAL VÆRE UNIK:

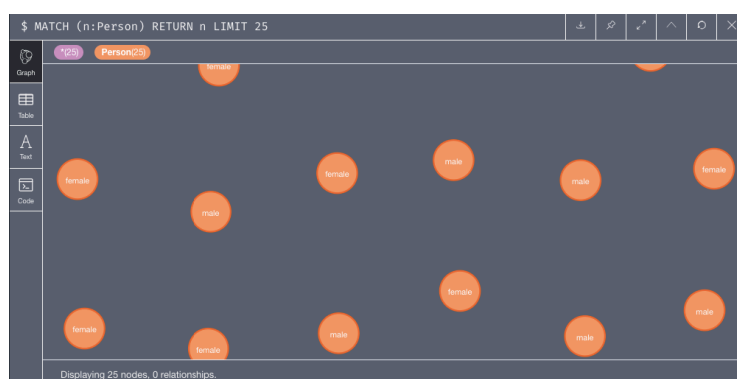
Etter at dette var gjort så jeg at det var nødvendig å sette en constraint på userid som sørger for at jeg ikke ender opp med like id. Utfører derfor følgende spørring:

```
CREATE CONSTRAINT ON (n:Person)  
ASSERT n.userid IS UNIQUE
```

SE ALLE NODER

Kan nå se at de ligger inne som noder:

```
MATCH (n:Person)  
RETURN n LIMIT 25
```



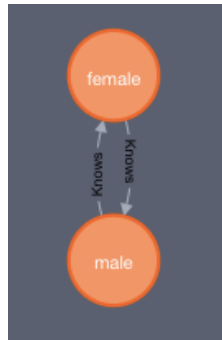
2.3 Egendefinert spørring

Da jeg ønsker å kjøre en av spørringene som sjekker venner, er jeg først nødt til å opprette relasjoner mellom ulike noder.

OPPRETTER RELASJONER (KNOWS)

Jeg kjørte først et par av følgende spørring hvor jeg la til relasjoner en og en

```
MATCH (a:Person), (b:Person)
WHERE a.userid = 2094382 AND b.userid = 1192601
CREATE (a)-[:Knows]->(b)
RETURN a,b
```

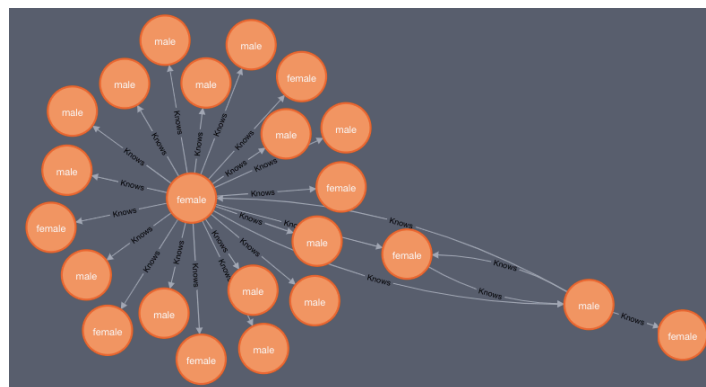


Deretter la jeg til en relasjon hvor en node skal kjenne alle som er født I 1999.

```
MATCH (a:Person), (b:Person)
WHERE a.userid = 1932519 AND b.dob_year = 1999
CREATE (a)-[:Knows]->(b)
RETURN a,b
```

Viser så resultatet og får resultatet under. (Viser kun 25 av nodene)

```
MATCH p=() - [r:Knows] -> ()
RETURN p
LIMIT 25
```



UTFØRER EGENDEFINERT SPØRRING \rightarrow *Hvem er venner til x sin venner*

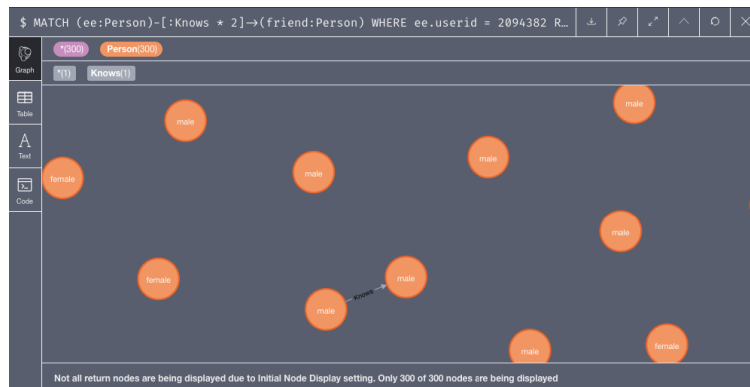
I mitt tilfelle blir spørringen mer konkret dette: «Hvem kjenner de som node med userid 2094382 kjenner?»

```

MATCH (ee:Person)-[:Knows * 2]->(friend:Person)
WHERE ee.userid = 2094382
RETURN friend

```

Får nå ut alle bekjente av bekjente til 2094382. Her kan man se at det er kun en felles relasjon



2.4 Adding

Noe som er en særlig vanlig del av GraphDB er relasjoner. Da datasettet fra før ikke inneholdt informasjon om relasjoner mellom brukerne var dette noe jeg la til selv som nevnt over.

Her vil jeg derfor fokusere på å legge til en ny node (Person) og legge til en label på en node som allerede finnes.

EN NY NODE (PERSON)

Legge til en ny node blir ganske lik som gjort over. Forskjellen er her at jeg istedenfor å lese inn data fra CSV-fil setter jeg inn verdiene direkte:

```

CREATE (p:Person {gender: "male",
dob_month: 12,
userid: 999999,
mobile_likes: 5,
dob_year: 1990,
www_likes: 55,
www_likes_received: 2,
mobile_likes_received: 3,
friendships_initiated: 54,
likes_received: 43,
friend_count: 0,
tenure: 6age: 30dob_day: 13,
likes: 5}
)

```

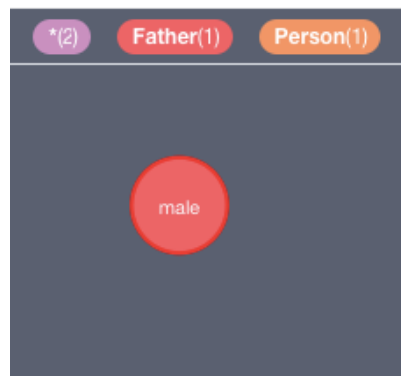
Kan nå med følgende spørring se at noden er lagt til. (Dette kan også gjøres samtidig med opprettelsen av noden)

```
MATCH (n:Person {userid: 999999})
RETURN n
```

SETTE TIL EN NY LABEL PÅ EN NODE

Legger her til en Label “Father” til på node med brukerid 999999.

```
MATCH (n:Person {userid: 999999})
SET n:Father
RETURN n
```



2.5 Removing

Her vil jeg vise hvordan jeg sletter en node, label og relasjon mellom to noder.

SLETTE NODE:

Ønsker her å slette en node med userid 1733186. Da jeg tidligere satt denne til unik vil det ikke være fare for at fler slettes.

```
MATCH (node:Person {userid: 1733186})
DETACH DELETE node
```

SLETTE LABEL:

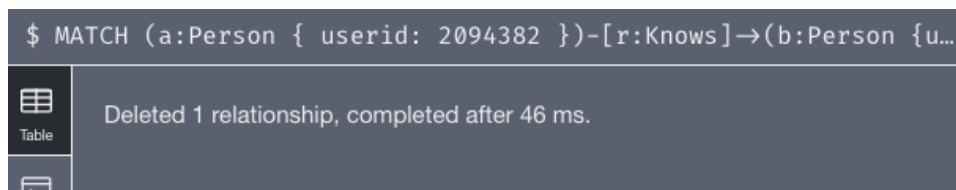
Fjerner her labelen jeg satt på node med bruker id 999999 tidligere. Etter kjøring av denne ser jeg at en label med Father ikke lenger finnes, da denne noden var den eneste med denne.

```
MATCH (n:Person {userid: 999999})
REMOVE n:Father RETURN n
```

SLETTE RELASJON:

Velger her å slette relasjonen mellom node 2094382 og 1192601. Benytter her referanse til selve relasjonen (r) for å kunne slette denne.

```
MATCH (a:Person {userid: 2094382})-[r:Knows]-(b:Person {userid: 1192601})
DELETE r
```



2.6 Modifying

Velger her å endre en egenskap (property) til en node (Person)

```
MATCH (n {userid: 1376108})
SET n.age = 10
SET n.dob_year = 2009
RETURN n.userid, n.age, n.dob_year
```

Ser nå at følgende informasjon er registrert på denne personen:

n.userid	n.age	n.dob_year
1376108	10	2009

// Endre navn på egenskap??

2.7 Ekstra

OPPDATERE ANTALL VENNER

Da en av egenskapene som er lagret for en person er antall venner (friend_count) ønsker jeg å oppdatere denne etter hvor mange personen kjenner. Når jeg sjekker hvor mange de ulike personene kjenner kjører jeg følgende spørring:

```
MATCH (n)-[Knows]->()
RETURN n.userid AS `Person`, COUNT(Knows) AS `Antall relasjoner`
```

Jeg ser da at flere av personene er fått en relasjon til andre personer

Person	Antall relasjoner
2094382	3
1192601	1
1932519	1926

Jeg ønsker å få satt denne informasjonen på egenskapen «friend_count» for den personen det gjelder og utfører følgende spørring:

```
MATCH (n)-[r:Knows]->()
WITH n, count(r) AS `Antall relasjoner`
SET n.friend_count = `Antall relasjoner`
```

Jeg utfører så en spørring for å vise en av personene.

```
MATCH (n: Person {userid: 1932519})
RETURN n
```

Ser her at egenskapen har blitt endret:

```
{
  "gender": "female",
  "dob_month": 3,
  "userid": 1932519,
  "mobile_likes": 0,
  "dob_year": 2000,
  "www_likes": 0,
  "www_likes_received": 0,
  "mobile_likes_received": 0,
  "friendships_initiated": 0,
  "likes_received": 0,
  "friend_count": 1926,
  "tenure": 0,
  "age": 13,
  "dob_day": 28,
  "likes": 0
}
```

GJENNOMSNIITT ANTALL RELASJONER

Da jeg nå har oppdatert informasjon om antall bekjenskaper for alle noder kan jeg enkelt finne ut gjennomsnittet av antall bekjenskaper

```
MATCH (n:Person)
RETURN avg(n.friend_count)
```

Får da følgende resultat:

avg(n.friend_count)

196.37689766976712

ENDRER PROPERTY-NAME

Da det i mitt tilfelle og bruk av dataen passer bedre at egenskapen «friend_count» kalles «knows_count» ønsker jeg å endre dette. Jeg føler at «Friend» i dette tilfelle blir feil da relasjonen «Knows» kan kun gå en vei. I mitt datasett kan altså en Node kjenne en annen, men denne noden kjenner ikke den tilbake.

Jeg utfører følgende spørring for å endre navn på egenskap:

```
MATCH (n:Person)
SET n.knows_count = n.friend_count
REMOVE n.friend_count
RETURN n
```

Da jeg ikke kan modifisere navnet på egenskapet direkte velger jeg å opprette en ny egenskap med navn “knows_count” og sette denne lik «friend_count». Deretter sletter jeg egenskapen «friend_count».

3. Kildeliste:

Cia, J.F. (2015, 22. mai). Neo4j: What a graph database is and what it is used for. Hentet fra <https://bbvaopen4u.com/en/actualidad/neo4j-what-graph-database-and-what-it-used>

Columnar Database (u.å). Columnar Database: A Smart Choice for Data Warehouses. Hentet 11. November 2019 fra <https://www.columnardatabase.com/>

Guru99 (u.å). Cassandra Tutorial for Beginners: Learn in 3 Days. Hentet fra <https://www.guru99.com/cassandra-tutorial.html>

Ian (2016, 31. juni). What is a Key-Value Database? Hentet fra <https://database.guide/what-is-a-key-value-database/>

IntelliPaat (2019, 5. september). Brief Architecture of Cassandra. Hentet 18. november 2019 fra <https://intellipaat.com/blog/tutorial/cassandra-tutorial/brief-architecture-of-cassandra/>

Reeve, A. (2013, 25. november). Big Data Architectures – NoSQL Use Cases for Key Value Databases. Hentet fra https://infocus.dellemc.com/april_reeve/big-data-architectures-nosql-use-cases-for-key-value-databases/

REST API Tutorial. (u.å). Using HTTP Methods for RESTful Services. Hentet 17. November 2019 fra <https://www.restapitutorial.com/lessons/httpmethods.html>

Stackshare (u.å). Cassandra vs Neo4j. Hentet 12. november 2019 fra <https://stackshare.io/stackups/cassandra-vs-neo4j>