Erfaringsrapport

Introduksjon/hensikt

Denne rapporten detaljerer våre erfaringer ved bruk av grafdatabasen Neo4 for geografiske data. Datasettet som vi tar utgangspunkt i og er grunnlaget for utarbeiding av denne erfaringsrapporten er Administrative Enheter Kommuner, som finnes her: Administrative Enheter [DØD LENKE]

Systemet vi utarbeider denne rapporten for er Ubuntu.

Dataene i dette datasettet er naturlig lenket:

- Norge har sine fylker
- Fylker har sine kommuner, og fylkene har sine nabofylker
- Kommuner har sine kommunegrenser representert som koordinater, og disse koordinatene har sine neste koordinater.
- Vi ønsker ikke å sette inn et koordinat mer enn en gang, og flere koordinater må kunne lenke til sine nabokommuner.

Rapporten vil være delt opp i følgende seksjoner:

1. Installasjon og konfigurering av Neo4j
2. Behandle JSON i Neo4j- browser (grafisk)
3. Neo4j og Java-applikasjoner
4. Visualisering av grafer
5. Oppgradering innenfor major releases 3.3.2 -> 3.3.3

Forord...

Det er hensiktsmessig å demonstrere:

- Kode
- Konfigurasjonsfiler
- Spørringer mot database

- Kommandoer

Dette illustreres på følgende måter slik at leser enkelt kan kjenne igjen konteksten.

Cypher

```
cql$ MATCH (n) RETURN n
```

Shell

```
usr$ sudo apt-get install <package>
```

Config filer

```
...

# Endringer her

...
```

Kode

```
public static void main(String[] args) {
    String var1 = "Hei";
}
```

Gradle

```
build.gradle
```

```
...
```

Ressurser

Neo4j-manual

Neo4j publisert operasjonsmanual for 3.3.x utgivelsene:

https://neo4j.com/docs/operations-manual/3.3/introduction/

Manualen finnes historisk. Benytter man en annen versjon enn 3.3.x, erstatter man versjonsnummeret i URL'en til det aktuelle versjonsnummeret.

Denne dekker de aller fleste temaer man støter på.

Neo4j og Java

Referanser:

https://neo4j.com/docs/java-reference/current/

JavaDoc:

https://neo4j.com/docs/java-reference/3.3/javadocs/

Dokumentasjon av grensesnittet

APOC: Neo4j og brukerdefinerte prosedyrer:

APOC:

https://github.com/neo4j-contrib/neo4j-apoc-procedures

Et bibliotek av prosedyrer skrevet i Java som kan kalles på fra Cypher.

Dokumentasjon:
https://neo4j-contrib.github.io/neo4j-apoc-procedures/

Neo4j-contrib

https://github.com/neo4j-contrib

3.part moduler til Neo4j med ulike formål

Andre referanser

http://finnposisjon.test.geonorge.no/

For søk på koordinater. Brukt for å validere innsettinger.

Installasjon og konfigurering

Informasjon

Installasjonsveiledning for forskjellige systemer finnes her:
https://neo4j.com/docs/operations-manual/current/installation/

Denne oppdateres ved nye utgivninger.

Vi har benyttet oss av installasjon for Debian baserte systemer:

https://neo4j.com/docs/operations-manual/current/installation/linux/debian/

For vårt prosjekt, har vi benyttet oss av Neo4j Community Edition. Denne versjonen har noe redusert funksjonalitet, deriblant mangler den støtte for kjøring av flere databaser samtidig. Vi har benyttet oss av versjon 3.3.3 og Java 8 på Ubuntu, og dette vil være utgangspunktet ved utarbeiding av denne erfaringsrapporten. Har man flere versjoner av Java installert, beskriver operasjonsmanualen hvordan man skal håndtere det her: Flere Java versjoner

Neo4j 3.3.3 krever Java 8 runtime. Dersom det ikke er installert, må det gjøres. Installasjon av Java 8 beskrives her:

https://neo4j.com/docs/operations-manual/3.3/installation/linux/debian/

Neo4j-installasjon

Etter Java 8 er på plass, kan man ta for seg installasjon av Neo4j.

Installer med apt-get:

```
usr$ sudo apt-get install neo4j=3.3.3
```

Neo4j anbefaler at man benytter `systemctl` for å starte/stoppe/etc databasen.

Se manualsiden for systemctl for andre kommandoer som systemctl støtter.

Neo4j-oppstart

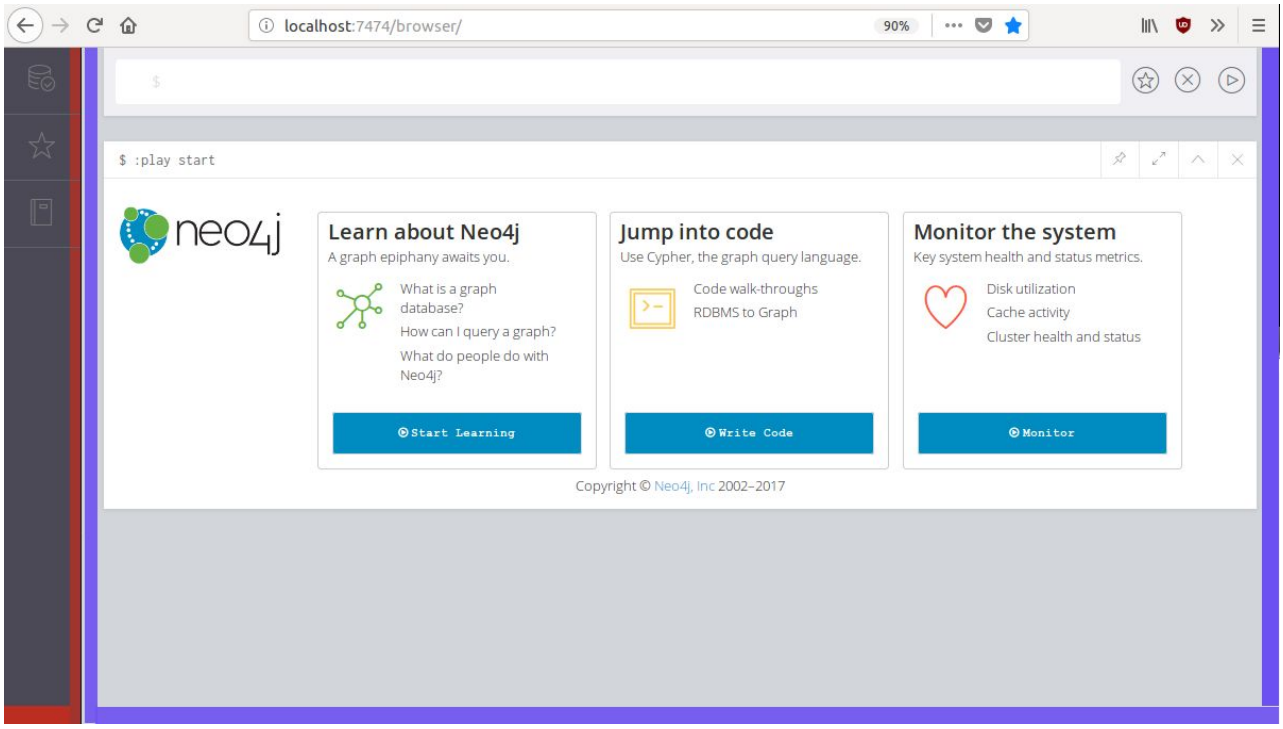
Starte Neo4j:

```
usr$ systemctl start neo4j
```

Stoppe Neo4j

```
usr$ systemctl stop neo4j
```

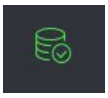
Når databasen er startet, starter det en tjeneste som kan nås via nettleser og shell. Naviger til `http://localhost:7474` og man blir møtt med en skjermbilde tilsvarende dette:



Dette vinduet er delt inn i 2 seksjoner:

Rødt:

- Meny med tre menyknapper.



Database Information
Viser metadata for den aktive databasen:

- Plass på disk
- Versjon
- Neo4j-versjon
- Informasjon om hvilke typer noder, labels og properties som eksisterer i databasen

Favorites
Meny for å lagre scripts, spørringer m.m. Supplerer også endel eksempler på scripts, prosedyrer, etc.

Documents
Grunnleggende referansemateriale for bruk av databasen.

Blått:

- Det øverste hvite feltet er kommandolinjebasert interaksjon mot databasen. Dette er stedet for å plassere spørringene dine.
- Alt under dette feltet viser en historikk over spørringene dine og hvilke resultater de har gitt. Resultatene representeres som en graf eller en liste, alt ettersom hva man ber om.

Neo4j har opprettet en database ved navn ***graph.db*** for deg automatisk. Default bruker er neo4j og passordet skal være passordet man benytter for å logge på maskinen. For Neo4j Community Edition, kan man kun ha en database kjørende, men man kan ha flere databaser liggende på en maskin.

Her finnes et par læringsmoduler som kan benyttes for å gjøre seg kjent med Neo4j og spørringer i CQL (Cypher Query Language), og det anbefales at leser går gjennom disse.

Lag en graf

I Neo4j-browser seksjonen for spørringer, klikk på knapp for “Write Code” under “Jump into Code” elementet og kjøre “Movie Graph” skriptet. Grafen som blir opprettet her, vil bli brukt i noen eksempler i denne rapporten. Denne grafen er nå laget i databasen **graph.db.**

Opprette og endre aktive databaser

Å opprette en ny database i Neo4j er veldig enkelt. Neo4j leser en konfigurasjonsfil ved oppstart. Her finnes bl.a innstillinger for sikkerhet, nettverk og hvilken database som er satt som aktiv database. Dersom den aktive databasen ikke eksisterer ved oppstart av Neo4j, opprettes den.

For Debian-systemer er databasene lokalisert i mappen `/var/lib/neo4j/data/databases/`

```
usr$ ls /var/lib/neo4j/data/databases/
graph.db
```

Konfigurasjonsfilen finner du her:

`/etc/neo4j/neo4j.conf`

Åpne neo4j.conf med skriverettighet

```
usr$ sudo nano /etc/neo4j/neo4j.conf
```

neo4j.conf

```
...
# The name of the database to mount
# dbms.active_database=graph.db
dbms.active_database=ny-database.db
...
```

Dette er helt på toppen av dokumentet. Dupliser linjen med med active_database, kommenter ut den gamle og sett et nytt navn på en database. Lagre og lukk filen, start Neo4j og den nye databasen vil bli laget og satt som aktiv database. List innhold i databasemappen og bekreft at den nye databasen eksisterer.

```
usr$ ls /var/lib/neo4j/data/databases/
graph.db  ny-database.db
```

I Neo4j Enterprise Edition kan man ha flere databaser aktive på en gang.

Databaseadministrasjon

Databasen kan administreres via:

1. Grafisk brukergrensesnitt
2. Ved å kjøre cypher-shell i kommandolinjen

Neo4j via kommandolinje:

```
usr$ cypher-shell
username: neo4j
password: ****************
```

Erfaringer

Installasjon av Neo4j kan gjøres ved få, enkle steg. Det er få avhengigheter til andre tjenester. Det eneste man må være observant på, er hvilken versjon av Java Runtime du har på maskinen din.

Behandle JSON med Neo4j

GEOJSON, GML, SOSI

Vi brukte en del tid på å kartlegge hvilket dataformat som var mest gunstig å arbeide med i Neo4j og hvilke verktøy som eksisterte for å parse dataen. Neo4j har ikke innebygget støtte for å arbeide med JSON-data, kun CSV. Omsider oppdaget vi APOC, som er et 3.parts bibliotek som tilbyr en rekke brukerdefinerte prosedyrer, deriblant en prosedyre for lasting av json filer. Vi benyttet denne blogposten som en introduksjon av temaet:

https://neo4j.com/blog/intro-user-defined-procedures-apoc/

Installasjon av APOC

Finn din Neo4j versjon

```
usr$ neo4j --version
neo4j 3.3.3
```

Finn passende APOC versjon

Det finnes flere APOC-versjoner, og disse er kompatible med forskjellige neo4j-versjoner. Oversikt over APOC-Neo4j kompatabiletet finnes her:
https://github.com/neo4j-contrib/neo4j-apoc-procedures#version-compatibility-matrix

For vår Neo4j-version (3.3.3), er APOC-version 3.3.0.2 den vi må benytte.

APOC ble et naturlig valg, og det at biblioteket hadde en prosedyre for å laste inne JSON, førte til at vi valgte å arbeide med datasettet i GEOJSON-format. JSON er også et format som vi kjenner fra før.

Installasjon av APOC

Last ned .jar filen som ligger i repositoriet som samsvarer med din Neo4j-versjon og legg den i /plugins mappen. For Debian-systemer er stien til mappen **/var/lib/neo4j/plugins**.

Bruker du et annet system, konsulter denne matrisen for å finne fil-lokasjoner til systemet du bruker:

https://neo4j.com/docs/operations-manual/current/configuration/file-locations/

Nå som APOC er installert, må du gi Neo4j rettigheter til å kjøre prosedyrene som APOC tilbyr. Dette gjøres via Neo4j sin konfigurasjonsfil:

Filsti på Debian-systemer er: ***/etc/neo4j/neo4j.conf***

Åpne denne filen med en teksteditor du ønsker å bruker (vi bruker nano)

```
usr$ sudo nano /etc/neo4j/neo4j.conf
```

neo4j.conf

```
...
# Sandbox settings for procedures. Catch-all statement that gives procedures
unrestricted   # access.
# Additional adjustments can be done by naming the specific procedures to be
allowed.
# To enable unrestricted access for procedures, uncomment this line
dbms.security.procedures.unrestricted=apoc.*
...
```

Søk etter linjen i bold i konfigurasjonsfilen. Dersom den ikke eksisterer, legg den til. Kommentarene ovenfor er egendefinerte for å beskrive hva innstillingen gjør.

Denne endringen lar deg kjøre alle prosedyrer som er definert i apoc pakken. Disse tillatelsene kan finmaskes ved å spesifisere underpakker og prosedyrenavn etter behov.

Verifisere installasjonen

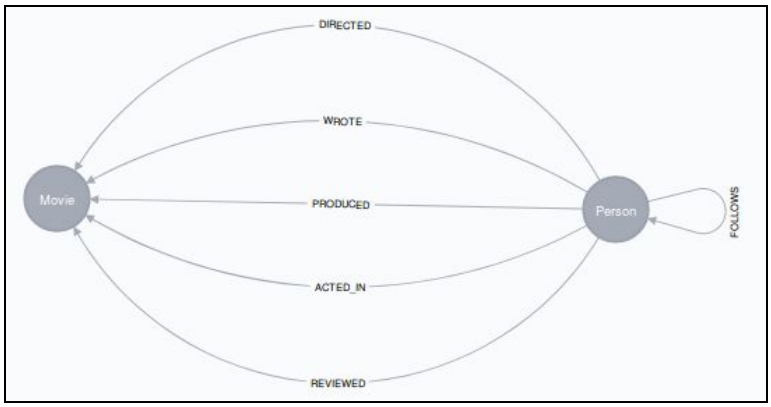
For å verifisere at installasjonen har gått bra, kan man kjøre et kall på en av de brukerdefinerte prosedyrene som APOC supplerer.

Start Neo4j-databasen og naviger til localhost:7474 i nettleseren. I dette eksempelet tar vi utgangspunkt i at “Movie Graph” skriptet er kjørt på databasen.

Skriv inn følgende i kommandolinjen øverst.

```
cql$ CALL apoc.meta.graph
```

Denne prosedyrer returnerer en visuell representasjon av de noder og forhold som eksisterer i grafen på et overordnet nivå. For “Movie Graph” vil metagrafen se slik ut.



Nå vet vi at APOC er installert og fungerer korrekt, og prosedyrene som finnes i APOC er eksponert for Neo4j.

Dokumentasjon for APOC finnes her:

https://neo4j-contrib.github.io/neo4j-apoc-procedures/

Behandle JSON med APOC

Her er en liste over ressurser som kan være nyttige for leser.

https://neo4j.com/docs/developer-manual/current/cypher/clauses/with/

https://neo4j.com/docs/developer-manual/current/cypher/clauses/merge/

https://neo4j.com/docs/developer-manual/current/cypher/clauses/unwind/

JSON-datasett: https://hotell.difi.no/api/json/difi/geo/fylke

Her er hensikten å vise at man kan behandle JSON i Neo4j ved hjelp av APOC-prosedyrene.

Her viser vi forholdsvis enkelt CQL kombinert med prosedyrekall. Datasettet vi bruker er en liste med navn over alle fylker i Norge med tilhørende fylkenummer, og hentes via et API.

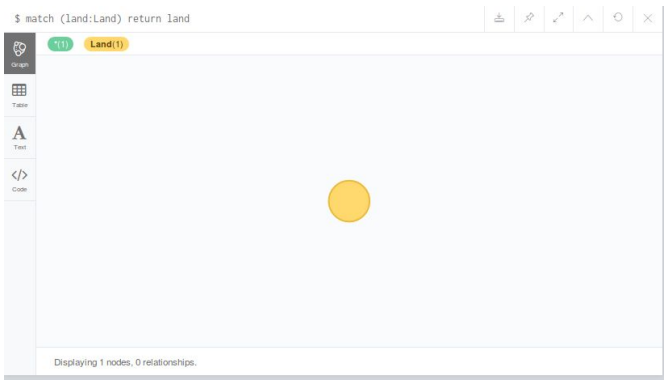
Dersom man JSON-filer lagret lokalt, må man gi APOC-prosedyrene tilgang til å importere lokale filer.

Gi APOC-prosedyrer tilgang til lokal import:

```
...
apoc.import.file.enabled=true
...
```

Lag en land-node, denne skal lenke til alle fylker:

```
cql$ MERGE (land:Land{ navn:"Norge", id:"NO"})
```



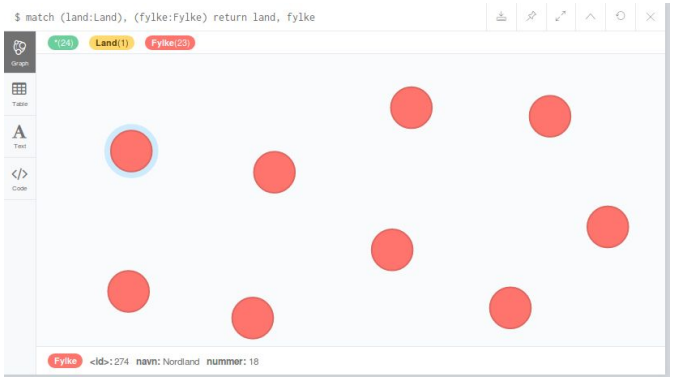
Hent JSON-data og opprett noder for hvert fylke:

```
cql$ WITH "https://hotell.difi.no/api/json/difi/geo/fylke" AS url
     CALL apoc.load.json(url) YIELD value
     UNWIND value.entries AS fylke
     MERGE (fylkenode:Fylke{nummer:fylke.nummer}) ON CREATE SET
           fylkenode.navn = fylke.navn
```

Dersom du har JSON-filer lokalt på disk:

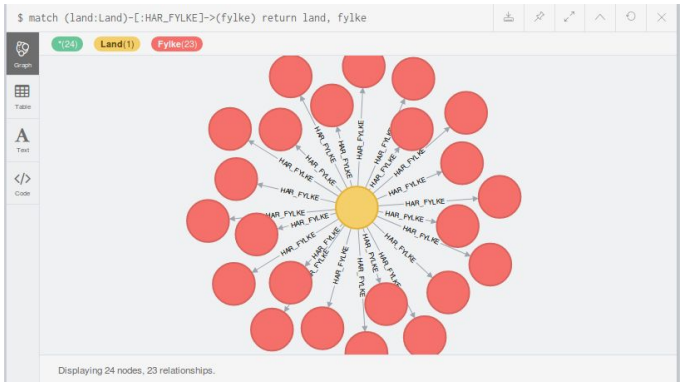
```
cql$ -- Forutsetter at du har gitt ACOP tilgang på lokal import av filer
     WITH "file://///path/to/json/data.json"
     ... identisk som spørringen mot API
```

Nå har vi land-node “Norge” og alle fylkenoder med tilhørende navn og fylkenummer. Neste steg i prosessen er å lenke de, slik at land har en [:HAR_FYLKE] relasjon til fylker.



Match alle fylker og landnoder med navn "Norge" og opprett relasjon

```
cql$ MATCH (f:Fylke), (l:Land{navn:"Norge"})
     MERGE (l)-[:HAR_FYLKE]->(f)
```



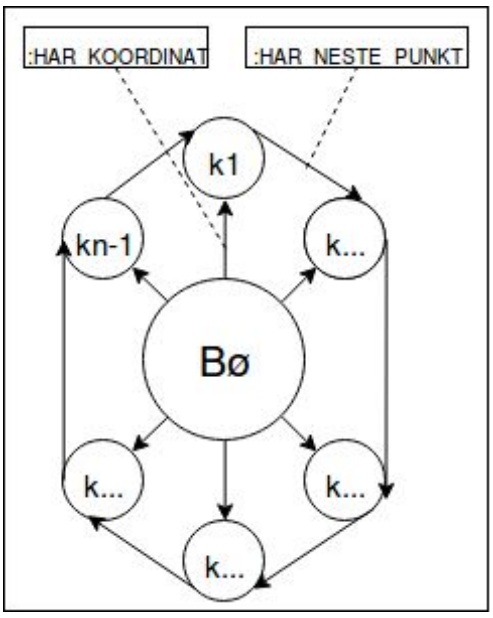
,

Behandle GEOJSON med APOC

Vår databehandling handler stort sett om geografiske data. Denne seksjonen tar for seg våre erfaringer med å behandle GEOJSON. Datasettet Administrative Enheter Kommuner finnes som nedlastbare filer i GEOJSON-format. Datasettene er forholdsvis store, og besluttet av den grunn å arbeide med datasett for en enkelt kommune, slik at vi kunne sette oss inn i strukturen. Datasettet for Bø kommune egnet seg fint for for dette formålet.

Vi er i hovedsak interessert i koordinatene som utgjør grensene til de administrative enhetene. For Bø kommune, er vi interessert i koordinatene som utgjør kommunegrensen.

Kartlegge hvordan grafen skal se ut

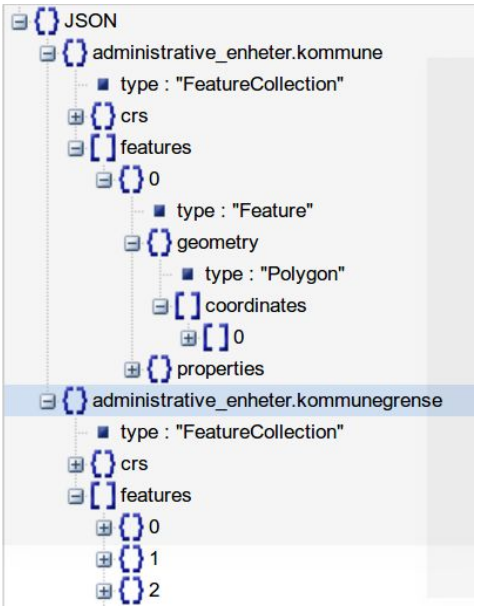


Vi ønsker å opprette en node som representerer Bø kommune, og denne noden har relasjonen [:HAR_KOORDINAT] mot noder som inneholder koordinater. Koordinatnodene skal ha en [:HAR_NESTE_PUNKT] relasjon til neste koordinatnode.

Til venstre kan man se en illustrasjon over hvordan vi ønsker at dataen skal representeres for en kommune. Denne strukturen skal være lik for alle kommuner.

For datasettet som gjelder alle fylker og kommuner i Norge ønsker vi at kommuner skal kunne dele koordinater med nabokommuner. På samme vis skal koordinater for kommuner deles med koordinater for fylkegrenser og kommuner i nabofylker. Fylker skal også kunne dele koordinater med nabofylker.

Datasettets utforming



Her er et utsnitt av datasettet for Bø kommune. Vi har benyttet http://jsonviewer.stack.hu/ for en menneskevennlig visning av JSON- filer. Objekter i illustreres med { }, tabeller illustreres med [ ]. Kommunegrensene er å få i to typer features, enten som "Polygon", eller som en samling av "LineString"-segmenter. Når vi arbeidet med dataen i CQL, valgte vi å bruke "Polygon".

Polygon er et sett avkoordinater, og i her utgjør settet en flate som beskriver kommunegrensen for Bø. Punktene ligger sekvensielt, slik at punkt[i] sin [:NESTE_PUNKT] blir punkt[i+1]. For punkt[n-1] blir [:NESTE_PUNKT] punkt[0].

Alle disse punktene har også en node "Kommune" som har en [:HAR_KOORDINAT] relasjon mot alle punkter for sin kommunegrense. Dette trodde vi at skulle være relativt greit å få til i CQL.

Innlasting av JSON-data for en kommune

Først må vi pakke ut JSON-data. Datasettene er tilgjengelige som nedlastbare filer, og APOC må derfor gis **tilgang til lokal import av filer.**

For navigasjon i JSON-objekter, bruker man vanlig dot-notasjon.

Her støtte vi på et problem med utformingen av datasettene. Toppnivå objektene i datasettene er

administrative_enheter.kommune og **administrative_enheter.kommunegrense.**

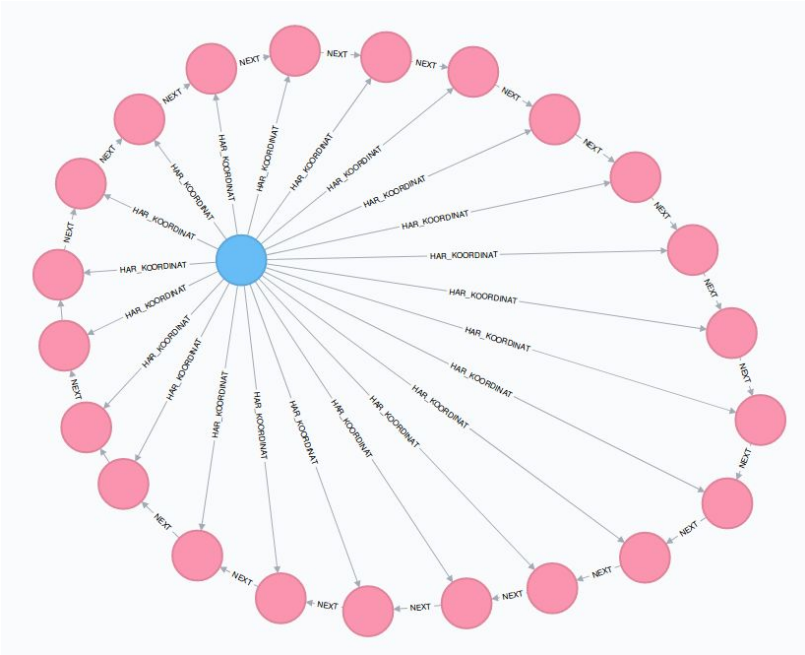
I Neo4j pakker man ut lister med **UNWIND json.path.** Siden de to øverste nivåene er keys, tolker ikke UNWIND navnene slik de er tenkt. Vi løste dette ved å manuelt endre navnene til **administrative_enheter_kommune** og **administrative_enheter_kommunegrense.** Etter

denne endringen var gjort, kunne vi bruke dot-notasjon for navigasjon. Dersom man skal jobbe med ett datasett per fylke, denne endringen gjøres for hvert fylke.

For dette eksempelet har vi benyttet datasettet for Kvitsøy kommune, som er en liten øy i Rogaland. Kvitsøy sin kommunegrense består av 23 koordinater, som gjør det enkelte å validere innsettingen grafisk i Neo4j-Browser.

```
cql$ WITH "file:////path/to/json/kvitsoy.geojson" as url
     CALL apoc.load.json(url) YIELD value
     UNWIND value.administrative_enheter_kommune AS adm
     UNWIND adm.features[0] as features
     WITH features.geometry.coordinates[0] AS koordinatListe
     UNWIND RANGE(0, LENGTH(koordinatListe)-2) as idx
     MERGE (kommune:Kommune{navn:"Kvitsøy"})
     MERGE (k1:Koordinat{
       lat:koordinatListe[idx][0],
       lon:koordinatListe[idx][1]
       })
     MERGE (k2:Koordinat{
       lat:koordinatListe[idx+1][0],
       lon:koordinatListe[idx+1][1]
       })
     MERGE (k1)-[:NEXT]->(k2)
     MERGE (kommune)-[:HAR_KOORDINAT]->(k1)
     MERGE (kommune)-[:HAR_KOORDINAT]->(k2)
```

Her itererer vi over alle koordinatene i listen over koordinater.



Videre må det opprettes en kommunenode for Bø i Telemark. Denne kommunen skal ha relasjon til alle koordinater i settet. Hver koordinat i settet skal ha en relasjon til sitt neste koordinat i settet. Det siste koordinatet i settet skal ha en relasjon til det første koordinatet i settet. Grafen for en enkelt kommune resulterer da i en sirkulær Linked List.

Det er nå tid og sted for å overføre det vi har lært for en kommune over til alle kommuner i et gitt fylke. Vi jobber videre med datasettet for Telemark kommune.

På grunn av ressursbruken til en slik spørringen under, har vi redusert antallet kommuner vi ønsker å lage en graf av til 5. Bø og nabokommunene Lunde, Notodden, Sauherad, Kviteseid og Seljord. Ved nærmere inspeksjon av datasettet, finnes ikke Lunde kommune, og Nome kommune har to oppslag med forskjellige koordinatsett. Vi tar ikke høyde for dette annet en av vi velger bort Lunde og Nome i innsettingen

```
cql$ UNWIND value.administrative_enheter_kommune AS adm
     WITH adm.features AS kommuner
       UNWIND [4, 5, 11, 12, 14] as kidx
```

```
  UNWIND kommuner[kidx] AS kommune
  UNWIND kommune.properties.navn[0].navn AS kNavn
  UNWIND kommune.geometry.coordinates AS koordinatListe
  UNWIND RANGE(0, SIZE(koordinatListe)-2) as idx
  MERGE (ko:Kommune{navn:kNavn})
  MERGE (k1:Koordinat{
            lat:koordinatListe[idx][0],
            lon:koordinatListe[idx][1]
            })
  MERGE (k2:Koordinat{
            lat:koordinatListe[idx+1][0],
            lon:koordinatListe[idx+1][1]
            })
  MERGE (k1)-[:NEXT]->(k2)
  MERGE (ko)-[:HAR_KOORDINAT]->(k1)
  MERGE (ko)-[:HAR_KOORDINAT]->(k2)
```

Forklaring:

Vi definerer en URL som peker til filen vi ønsker å behandle. Det foretas så et kall på en apoc-prosedyre der returverdien defineres i en variabel ‘value’. Deretter pakkes JSON-data som vi er interessert i ut til alias ‘adm’.

```
...   WITH "file:////path/to/json/telemark.geojson" as url
      CALL apoc.load.json(url) YIELD value
      UNWIND value.administrative_enheter_kommune AS adm
```

Her definerer vi en JSON-tabell som inneholder JSON-objekter, der hver rad inneholder data for en gitt kommune i fylket som en ressurs vi skal bruke videre i skopet definert av WITH. WITH-klausulen lager et nytt skop hver gang det brukes. Aliaset ‘kommuner’ kan nå aksesseres via tabell-indekser direkte.

```
...   WITH adm.features AS kommuner
```

Vi er interessert i et utvalg kommuner i Telemark, og har derfor definert en tabell som inneholder tabellreferansene til de kommunene vi vil lage en graf av. UNWIND pakker ut liste-literalen vi har definert, og for hver iterasjon, får aliaset ‘kidx’ en ny heltallsverdi som vi kan bruke til å indeksere tabellen ‘kommuner’.

```
...   UNWIND [4, 5, 11, 12, 14] as kidx
```

Bruk ‘kidx’ til å aksessere JSON-tabellen ‘kommuner’. UNWIND pakker ut resultatet til et nytt JSON-objekt kalt ‘kommune’, som vi kan bruke dot-notasjon på. Kommuneobjektet vårt inneholder blant annet kartdata og navn for en gitt kommune.

```
...   UNWIND kommuner[kidx] AS kommune
```

Vi pakker ut navnet til kommunen, gitt alias ‘kNavn’.

```
...   UNWIND kommune.properties.navn[0].navn AS kNavn
```

Vi pakker ut en liste der hver indeks har en ny liste med koordinatene. [0] for Øst-koordinat, [1] for Nord-koordinat.

```
...   UNWIND kommune.geometry.coordinates AS koordinatListe
```

For å opprette nødvendig [:NEXT] relasjon mellom koordinatene, må vi kunne indeksere et koordinat og dets neste koordinat, definert av rekkefølgen av koordinatene i polygonet. SIZE-funksjonen returnerer lengden til en liste. Siden vi er interessert i å få tak i det neste koordinatet, trekker vi fra antall steg vi kommer til å indeksere oss fremover. For dette eksempelet indekserer vi ett steg fremover, så vi tar 2 fra totalen, fordi tabellen starter på 0.

```
...   UNWIND RANGE(0, SIZE(koordinatListe)-2) as idx
```

Oppretter en Kommune-node med navn, dersom den ikke eksisterer.

```
...   MERGE (ko:Kommune{navn:kNavn})
```

Oppretter en Koordinat-node for ‘idx’ sin nåværende verdi med lat og lon, får henholdsvis verdiene til Øst og Nord koordinat.

```
...   MERGE (k1:Koordinat{
                lat:koordinatListe[idx][0],
                lon:koordinatListe[idx][1]
```

```
        })
```

Oppretter en Koordinat-node for det neste koordinatet i listen. Derfor ‘idx+1’.

```
...     MERGE (k2:Koordinat{
            lat:koordinatListe[idx+1][0],
            lon:koordinatListe[idx+1][1]
        })
```

Oppretter en [:NEXT] relasjon mellom koordinatene. Siden det første og siste koordinatet i polygonene er det samme, blir grafen opprettet som en sirkulær linket liste.

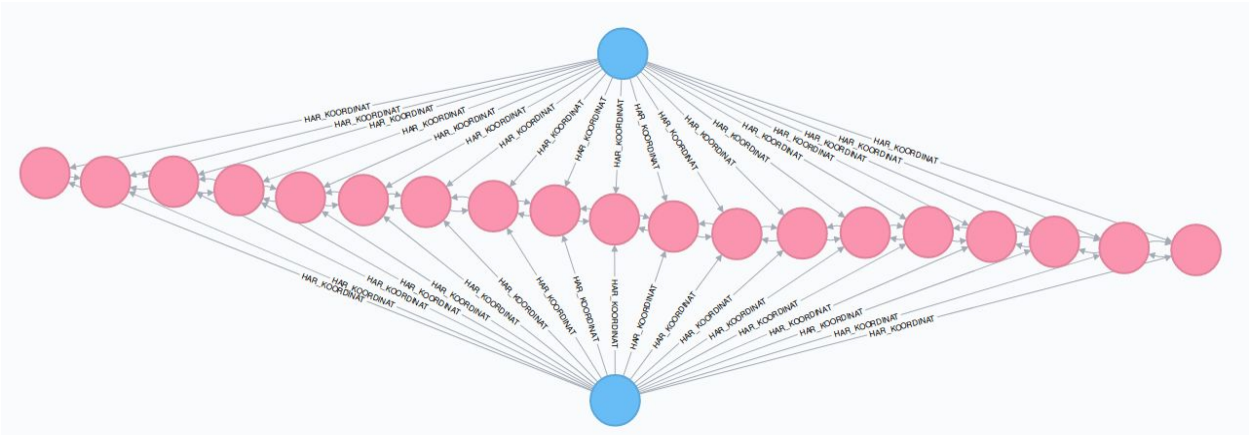
```
...     MERGE (k1)-[:NEXT]->(k2)
```

Oppretter en [:HAR_KOORDINAT]- relasjon mellom nåværende Kommune-node og koordinatene.

```
...     MERGE (ko)-[:HAR_KOORDINAT]->(k1)
        MERGE (ko)-[:HAR_KOORDINAT]->(k2)
```

Dette skripetet ble kjørt på en av prosjektdeltakernes maskin. For å prosessere disse 5 kommunene, tok det 6.5 minutter. Vi kjørte også en annen variasjon av dette skriptet som lager en graf av alle kommunene i et fylke. Dette tok 64 minutter å utføre. Årsaken til denne tidsbruken er at MERGE oppretter noder og mønstre, dersom de ikke eksisterer fra før. Det vil si at MERGE søker gjennom alle mønstre som allerede er opprettet, og dersom den finner en match, bruker den matchen i stedet for å opprette ny. Dette er for å unngå duplikater, men det har vannvittig stor innvirking på prosesseringstiden for spørringen. Spørringene kan nok optimaliseres noe, men vi har gått bort i fra denne løsningen, og jobber videre med Java Embedded Database.

Utsnitt av grensen mellom Bø og Notodden:

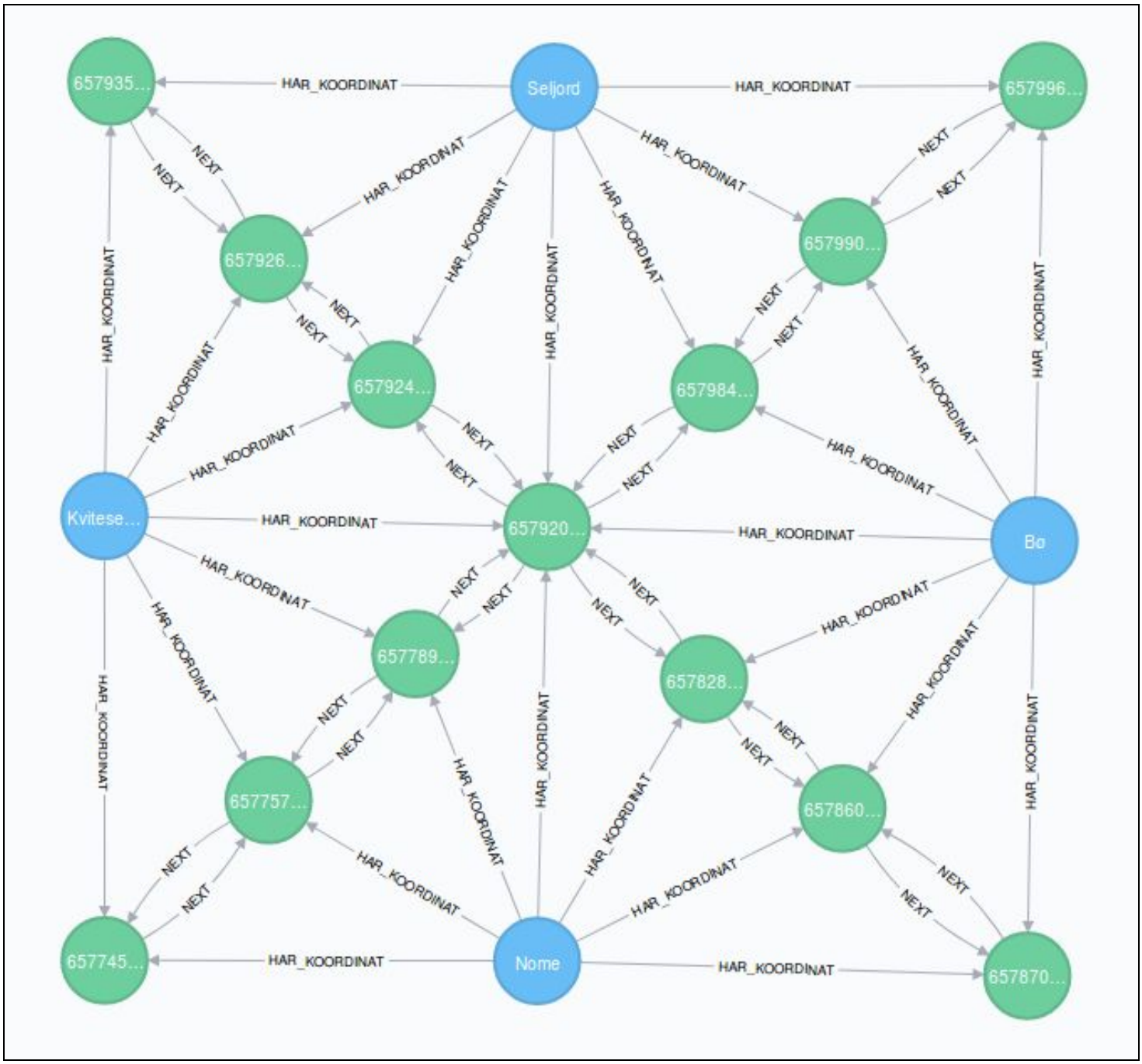


Dette skriptet ble kjørt på en av lab-maskinene.

```
cql$    WITH "file:////path/to/json/telemark.geojson" as url
        CALL apoc.load.json(url) YIELD value
        UNWIND value.administrative_enheter_kommune AS adm
        WITH adm.features AS kommuner
            UNWIND RANGE(0, SIZE(kommuner)-1) as kidx
            UNWIND kommuner[kidx] AS kommune
            UNWIND kommune.properties.navn[0].navn AS kNavn
            UNWIND kommune.geometry.coordinates AS koordinatListe
            UNWIND RANGE(0, SIZE(koordinatListe)-2) as idx
            MERGE (ko:Kommune{navn:kNavn})
            MERGE (k1:Koordinat{
                      lat:koordinatListe[idx][0],
```

```
                lon:koordinatListe[idx][1]
            })
        MERGE (k2:Koordinat{
                lat:koordinatListe[idx+1][0],
                lon:koordinatListe[idx+1][1]
            })
        MERGE (k1)-[:NEXT]->(k2)
        MERGE (ko)-[:HAR_KOORDINAT]->(k1)
        MERGE (ko)-[:HAR_KOORDINAT]->(k2)
```

Vi vet at Bø grenser til Kviteseid og at de hadde liten felles grense, så vi utførte en spørring som viste oss de punktene som var knyttet til begge kommuner. Grafen nedenfor viser et bearbeidet resultat av denne spørringen, og nodene for Bø og Kviteseid ble resultatet her. Vi fulgte stien i alle retninger, for å kontrollerer at relasjonene ble opprettet korrekt, og etter litt flytting av noder i det grafiske grensesnittet, ble resultatet slik du nå ser.


657935...
Seljord
657996...
657926...
657990...
657924...
657984...
Kvitese...
657920...
Bø
657789...
657828...
657757...
657860...
657745...
Nome
657870...
HAR_KOORDINAT
NEXT

Ovenfor ser man et utsnitt av kartet, der vi har brukt http://finnposisjon.test.geonorge.no/ for å finne posisjonen som spørringen resulterte i.

Spørringen som ble kjørt:

```
cql$  match
      p=(kviteseid:Kommune{navn:"Kviteseid"})-[:HAR_KOORDINAT]->(:Koordinat)<-
      [:HAR_KOORDINAT]-(bo:Kommune{navn:"Bø"}) return p
```

Erfaringer

Å bruke Neo4j, Cypher og APOC-biblioteket er egnet for innsetting av mindre datasett, for eksempel innsetting av data per fylke. Siden kommunegrensene for Telemark tok 64 minutter å prosessere, har vi valgt å ikke jobbe videre med denne løsningen for innsetting av alle kommuner og fylker i en database. For å redusere prosesseringstiden for større datasett, har vi behov for mer detaljert kontroll over nodene som lages, i tillegg til at vi trenger tilgang til forskjellige datastrukturer. Neo4j har utviklet et Java-API som er godt dokumentert, og vi har valgt å gå videre med en løsning med dette, kombinert med de datastrukturene og kontrollstrukturene som finnes i Java, for å lage en mer effektiv løsning.

Neo4j og Java-applikasjoner (Java Embedded Database)

Hva er Java Embedded Database

Java Embedded Database er en teknikk for å arbeide mot en Neo4j database lokalt gjennom en Java-applikasjon. Man refererer til mappen der databasen ligger, og instansierer den i Java-applikasjonen. Denne teknikken tillater utviklere å bruke Neo4j sitt Java-API direkte.

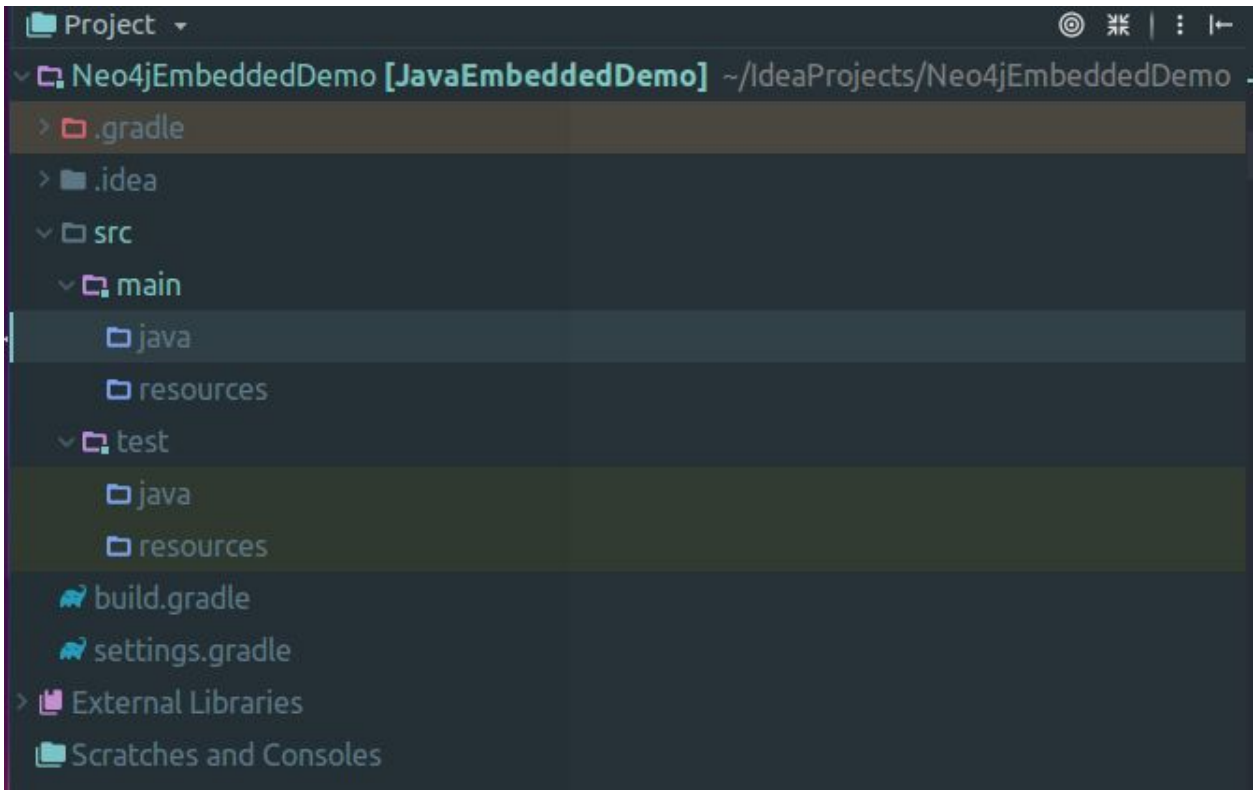
Neo4j tillater kun at en versjon av en database er instansiert på en maskin.

Neo4j Java-API

Vi har brukt IntelliJ som utviklingsmiljø mot databasen. Før man kan utvikle Java-applikasjoner som embedder database, må man ha tilgang på Java-API'et som Neo4j har utviklet. Vi har brukt Gradle som build-automasjons verktøy for å oppnå dette.

Fremgangsmåte:

1. Opprett et nytt Gradle prosjekt i IntelliJ. GroupID og ArtifactID velger dere selv. Følg veiviseren og opprett prosjektet med standardinnstillinger. Prosjektstrukturene ser da slik ut:



2. Som du ser, er det ikke opprettet noen Java-filer. Vi må ha et sted å starte programmet fra. Opprett klassen **"Main"** under java-mappen, og legg inn main-metoden i denne klassen. Skriver du inn "psvm" og trykker Enter, lager IntelliJ denne metoden for deg.
3. Opprett en avhengighet til Neo4j sitt Java-API
   build.gradle:

build.gradle

```
dependencies {
    …
    implementation "org.neo4j:neo4j:3.3.2" // Versjonsnummer for
databasen
    …
}
```

4. Synkroniser prosjektet og API'et lastes ned. Du har nå mulighet for å bruke API'et i prosjektet ditt.

Brukere og rettigheter

Før man går i gang med programmering mot databasen din, er det verdt å merke seg et par ting. Dette gjelder i hovedsak dersom man bruker Linux som operativsystem.

Når man oppretter en database, blir neo4j eier av databasen. Når man starter en Java-applikasjon mot databasen, kjøres den med din bruker. Nye filer og mapper som opprettes av brukeren din, har ikke neo4j tilgang til, og motsatt. Det er forskjellige måter å løse dette problemet på, men vi fant ut at det enklest er å melde brukeren som kjører applikasjonen inn i neo4j gruppen, og så sørge for at alle nye filer som blir opprettet i databasen, arver neo4j som gruppe.

Opprett en database for testing

Vi starter med å opprette en ny database med navn "nydb" som vil være testgrunnlaget. Hvordan du gjør dette, er beskrevet i seksjonen Opprette og endre aktive databaser.

Start og stopp neo4j-tjenesten, slik at databasemappen blir opprettet.

Skift mappe til **/var/lib/neo4j/data/database**.

```
usr$  cd /var/lib/neo4j/data/database
```

Legg brukeren din til i neo4j gruppen.

```
usr$  adduser [usr] neo4j
```

Filene som allerede ligger i databasemappen er feil i forhold til hva Java-applikasjonen trenger.
Vi endrer da eier på alle filene som ligger i denne mappen.

```
usr$  sudo chmod -R g=rw nydb.db
```

Filene har nå eierskap **[usr] neo4j**, for henholdsvis bruker og grupperettighet.
Vi ønsker at alle nye filer som opprettes i denne mappen, får likt eierskap.

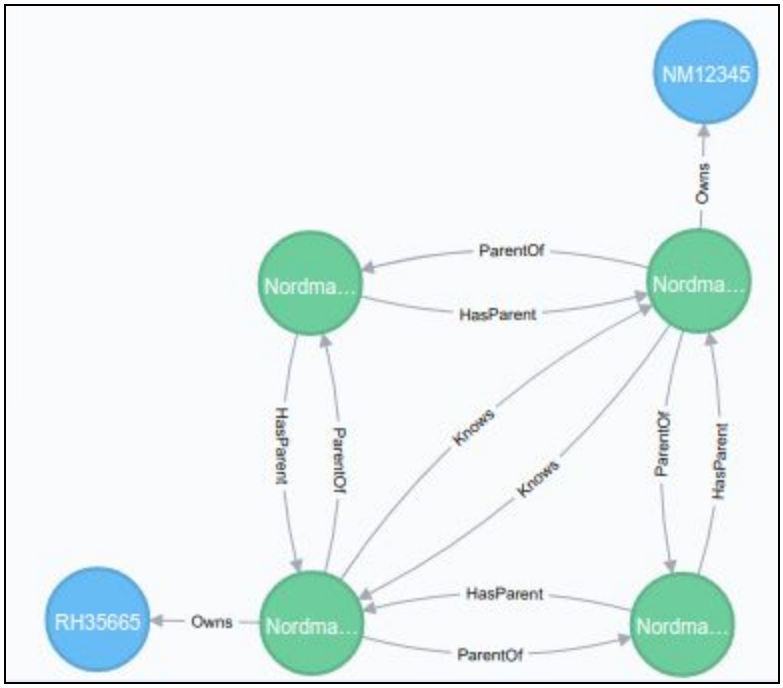
```
usr$  sudo chmod g=rwxs nydb.db
```

Gratulerer, vi har nå spart dere for mye feilsøking.

Embedde databasen i en Java-applikasjon

Vi går nå gjennom de grunnleggende stegene man må gjøre for å embedde databasen og utføre en transaksjon. Programkoden for å opprette graf som består av fire personer og to biler, og fornuftige relasjoner mellom de, finner man HER[LINK TIL BUNN].

I denne seksjonen lager vi for enkelthetskyld en graf som består av en person, en bil og et eierskapsforhold mellom de.



All kode i denne seksjonen ligger i **Main.java** i det nye test-prosjektet som vi nylig opprettet.

For å embedde databasen i en Java-applikasjon, må man først ha en referanse til filen/mappen hvor databasen ligger.

```
// Filen til databasen man ønsker å benytte

private static final File DB_DIR = new File("/var/lib/neo4j/data/databases/nydb.db");
```

Vi oppretter databasen i main-metoden. Du har nå en instans av databasen “nydb.db” i Java-applikasjonen. Oppretting av noder og relasjoner skjer ved hjelp av metoder definert i GraphDatabaseService-klassen

```
GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
GraphDatabaseBuilder dbBuilder = dbFactory.newEmbeddedDatabaseBuilder(DB_DIR);
```

```
GraphDatabaseService grapdb = dbBuilder.newGraphDatabase();
```

Før man kan opprette noder og relasjoner, må man definere typene av node og relasjoner man har tenkt å benytte i databasen. Dette gjøres ved hjelp av enum-klasser som implementerer grensesnittene Label og RelationshipType, definert av Neo4j sitt Java-API. Vi definerer disse som indre klasser i Main.java.

```
// Nodetyper som man kan bruke i grafen
public enum NodeType implements Label {
    Person,
    Car
}

// Forhold som man kan bruke i grafen
public enum RelationType implements RelationshipType {
    Knows,
    Owns,
    HasParent,
    ParentOf
}
```

Vi starter så en transaksjon mot databasen. All kode som påvirker databasen må skje i en transaksjon som man starter, og så avslutter når datainnsetting og uthenting er ferdig. Grafdatabasen må så stenges ned, slik at Java-applikasjonen ikke lenger har lås på databasefilen. Under testing kan det være lurt å slette alle noder og relasjoner før man setter inn data. Dette utfører vi med en rå Cypher Query Language spørring.

```
try (Transaction tx = graphdb.beginTx()) {
    // Slett alle noder og relasjoner i databasen før annen databaseinteraksjon.
    graphdb.execute("MATCH (n) DETACH DELETE n");

    …
    // Databaseinteraksjon skjer her
    …
    tx.success();
    tx.close();
}
graphdb.shutdown();
```

Vi går tilbake til transaksjonen og oppretter noder og relasjoner. Vi oppretter her kun et par forskjellige noder og relasjoner for å demonstrere. Hele Main-klassen finner du HER[LINK TIL BUNN]

```
…
// Innenfor transaksjonen
```

```
// Opprett en bil-node og angi egenskaper
Node saab = grapdb.createNode(NodeType.Car);
saab.setProperty("regnr", "RH35665");
saab.setProperty("model", "9-3");
saab.setProperty("førstegangsreg", "19990406");

// Opprett Person-node og angi egenskaper
Node ola = grapdb.createNode(NodeType.Person);
ola.setProperty("fornavn", "Ola");
ola.setProperty("etternavn", "Nordmann");
ola.setProperty("alder", 45);

// Opprett forhold mellom personer og biler
ola.createRelationshipTo(saab, RelationType.Owns);
…
```

Prosessen vil være tilsvarende dersom man oppretter flere noder og relasjoner av typene definert i de indre klassene. Ønsker man flere typer, kan legge de til i disse klassene.

Dersom du nå kjører denne klassen, vil forløpet bli som følger:

1. Instansier databasen i Java-applikasjonen
2. Start en transaksjon
3. Slett alle eksisterende noder og relasjoner
4. Utfør datainnsetting og uthenting
5. Utfør transaksjonen
6. Lukk instansen av databasen

Dersom du nå starter databasen ved **systemctl** og åpner det grafiske brukergrensesnittet og kjører følgende spørring, vil du se den grafen du nettopp har opprettet:

```
cql$  MATCH (n) return n;
```

Husk!
Før du kjører en Java-applikasjon som embedder en Neo4j-database, må du stoppe databasetjenesten **systemctl stop neo4j**. Dersom du ikke gjør det, vil applikasjonen kaste unntak på at en annen prosess har databasen kjørende. Det programmet som bruker databasefilen, får en lås på databasen, og vil ha enerett på å bruke databasen til låsen er åpnet.

For å demonstrere,
https://raw.githubusercontent.com/gitsieg/Neo4jEmbeddedDemo/master/src/main/java/Main.java

Java-embedded database for innsetting av geografiske data

Bakgrunn

Vi valgte å gå for denne løsningen for innsetting av geografiske data, grunnet datasettets størrelse og lang prosesseringstid ved innsetting når man benytter Cypher Query Language.

Når man har tilgang på data- og kontrollstrukturene som finnes i Java, kan man redusere tidsordenen for innsetting på bekostning av økt minnebruk. Med andre ord så flytter man den logikken som Neo4j utfører ved oppretting av noder og relasjoner inn i Java-applikasjonen.

Datasett

Datasettet for Administrative Enheter er tilgjengelig på land, fylke og kommune basis. Vi har valgt å benytte oss av datasett på fylkenivå, som igjen inneholder data på kommunenivå.

Dataene prosesseres fylke for fylke, og innad hvert fylke prosesseres hver kommune. Da koordinatdataene ligger i korrekt rekkefølge i datasettene er dette utgangspunkt for relasjonene mellom de.

Hvert fylkes datasett er innsatt i grafdatabasen som individuelle transaksjoner, da minnebruken ellers blir for stor.
Dette fører til en utfordring med tanke på relasjoner, og vi tok i bruk en slags ORM for å komme rundt dette.

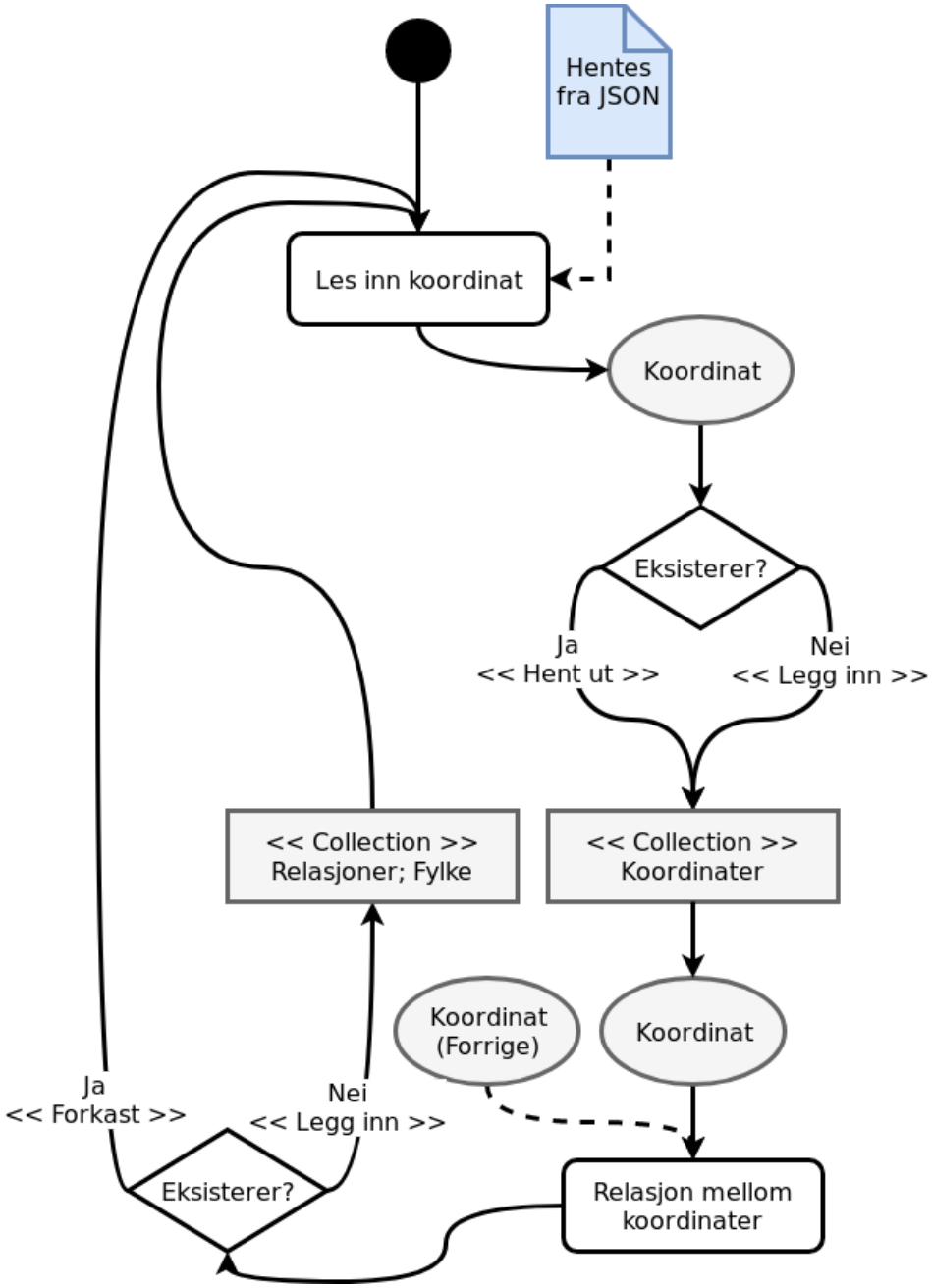
I LibGraph er det definert to klasser; **Koordinat** og **Forhold**. Koordinat-objekter representerer koordinater slik de er gitt i datasettene, mens Forhold-objekter representerer relasjoner mellom koordinater enten i fylkes- eller kommunesammenheng.
Klassene implementerer Comparable slik at vi kan bruke de i TreeSet-samlinger. TreeSet egner seg best i denne sammenhengen da både søk og sortert innsetting foregår i log(n)-tid, og fordi vi uansett ikke ønsker duplikater (slik duplikater er definèrt i hver modell/klasse.)

Når vi ved gjennomløping av datasettene møter på koordinater opprettes et Koordinat-objekt, dette objektet sjekkes opp mot koordinat-samlingen for å finne ut om det allerede eksisterer, vi ønsker ikke duplikater.
Eksisterer koordinatet *ikke* i samlingen setter vi det inn i grafdatabasen og i koordinatsamlingen.
Eksisterer koordinatet forkastes det, og vi henter i stedet ut det korresponderende koordinatet (objektet) fra koordinat-samlingen.

Vi har definért to typer relasjoner, for hhv. fylke og kommune. Dette er nødvendig for å kunne traversere relasjoner riktig.

De to typene settes inn i hver sin samling, og eksisterer som NESTE_PUNKT_KOMMUNE/FYLKE i databasen.

For hvert koordinat opprettes et relasjonsobjekt, og vi sjekker den korresponderende samlingen (basert på om vi er i fylke- eller kommune-kontekst) om relasjonen allerede eksisterer. Gjør den det forkastes den, og vi går videre. Eksisterer den ikke setter vi den inn i samlingen og legger den til i grafdatabasen.



Visualisering av grafer

https://cyneo4j.wordpress.com/installation-instructions/