Datainnsetting i Neo4j

Innholdsfortegnelse

1 Innledning

1.1 Innhold og struktur

Denne rapporten detaljerer våre erfaringer ved bruk av grafdatabasen Neo4j i behandling av geografiske data.

Erfaringene beskrives todelt:

1. Veiledning med eksempler og beslutninger, delt inn i seksjoner.
2. Oppsummerende erfaringer for hver seksjon.

Det kan være nyttig å lese igjennom en seksjons erfaringer i forkant av selve kapittelet.

Rapporten kan også sees på en innføring i grafdatabasen Neo4j. Rekkefølgen av kapitler følger i stor grad forløpet til prosjektarbeidet. Du vil få best utbytte av denne rapporten om du arbeider med temaene som tas opp samtidig som du leser.

1.2 Datagrunnlag

Datasettet Administrative Enheter er grunnlaget ved utarbeiding av denne erfaringsrapporten. Datasettet finner du på geonorge, søk på Administrative Enheter: https://kartkatalog.geonorge.no/metadata/kartverket/

Datasettet er naturlig lenket:

- Norge har sine fylker
- Fylker har kommuner, nabofylker og en fylkegrense.
- Kommuner har nabokommuner og kommunegrenser.
- Fylkegrenser og kommunegrenser består av et sett koordinater.
- Koordinater har et neste koordinat, og samlingen definerer en grense for kommune eller et fylke.
- Et koordinat kan ha forbindelse til flere koordinater, fylker og kommuner.

1.3 Kort om Neo4j og mål

Neo4j bruker en graf-modell ved navn “Graph-Property Model. Enkelt forklart så kan både noder og kanter (kanter er relasjoner i Neo4j) ha egenskaper tilknyttet seg. Mer informasjon om hva Neo4j og grafmodellen er, finner du her: https://neo4j.com/developer/graph-database/. Dette er også kort beskrevet i denne rapporten under delkapittelet Neo4j.

Vi har som mål å lenke dataen på best mulig måte ved å å bruke Neo4j. Koordinatene beskriver fylker og kommuner sin relasjon til hverandre. Vi besluttet derfor at duplikater av noder i databasen ikke skulle tillates.

1.4 Kode, kommandoer og konfigurasjonsfiler

I rapporten er det hensiktsmessig å vise kode, kommandoer og innhold konfigurasjonsfiler. Vi demonstrerer dette på følgende måter slik at du enkelt kan se hvor ting utføres når du leser rapporten.

Cypher:

Enkel forklarende tekst:

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

Shell:

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

Config filer:

name.extension:

```
…
# Endringer her
…
```

Kode:

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

2 Neo4j Installasjon

2.1 Generell informasjon

Vi har benyttet oss av Neo4j versjon 3.3.3 og Java 8 på Linux.

Installasjonsveiledning av Neo4j på forskjellige operativsystemer finnes her:
https://neo4j.com/docs/operations-manual/current/installation/

Vi har installert Neo4j for Debian-baserte systemer. Installasjonsguide finnes her:
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.

Har man flere versjoner av Java installert, beskriver operasjonsmanualen hvordan man skal håndtere dette her:
https://neo4j.com/docs/operations-manual/3.3/installation/linux/debian/#multiple-java-versions

Neo4j 3.3.3 krever Java 8 Runtime. Dette må installeres dersom du ikke har det fra før. Installasjon av Java 8 beskrives her:
https://neo4j.com/docs/operations-manual/3.3/installation/linux/debian/

Databasen kan administreres ved hjelp av:

- Neo4j-Browser
- Cypher-shell (kommandolinjebasert verktøy)

Under prosjektarbeidet har vi kun benyttet oss av Neo4j-Browser. Informasjon om verktøyet Cypher Shell finnes her:
https://neo4j.com/docs/operations-manual/3.3/tools/cypher-shell/

2.2 Neo4j-installasjon

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

Installer Neo4j v3.3.3 med apt-get:

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

2.3 Neo4j-oppstart og Neo4j-Browser

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

Se manualsiden for **`systemctl`** for andre kommandoer som systemctl støtter.

Starte Neo4j:

```
usr$ sudo systemctl start neo4j
```

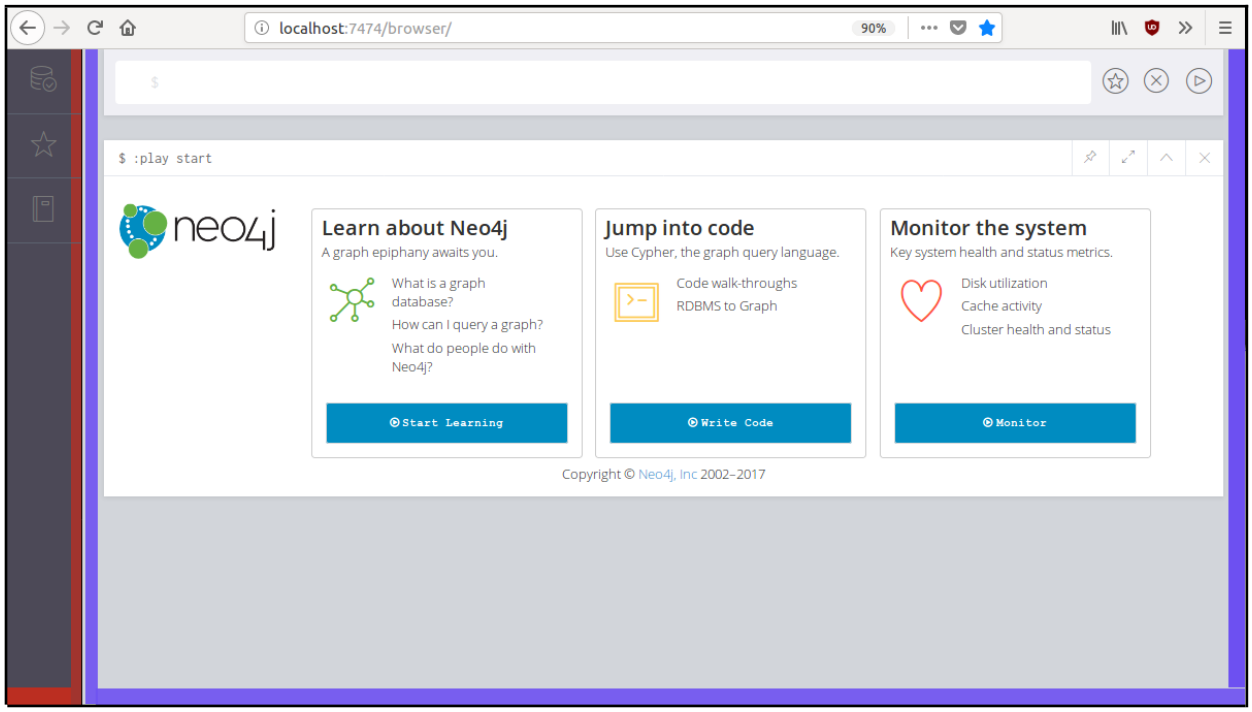
Stoppe Neo4j:

```
usr$ sudo systemctl stop neo4j
```

Se status på Neo4j:

```
usr$ systemctl status neo4j
```

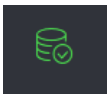
Når databasen er startet, tilgjengeliggjøres det en tjeneste som kan nås via nettleser. Denne tjenesten omtales heretter som **Neo4j-Browser**, og er Neo4j sitt grafiske brukergrensesnitt. Vi har brukt Firefox og Google Chrome. Naviger til i nettleseren din `http://localhost:7474` og man blir møtt med en skjermbilde tilsvarende dette:



Neo4j-Browser er delt inn i 2 seksjoner:

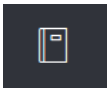
Rødt:

- 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 noen eksempler på scripts, prosedyrer, osv..

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.
  [Enter] kjører spørringer, [Shift][Enter] lager en ny linje, om man ønsker formattéring.
- Alt under dette feltet viser en historikk over spørringene dine og hvilke resultater de har gitt. Resultatene representeres som en graf eller en liste, avhengig hva man ønsker.

Neo4j oppretter en database ved navn ***graph.db*** for deg ved førstegangskjøring. Standard bruker er 'neo4j' og passordet skal være passordet man benytter for å logge på maskinen sin systembruker. I Neo4j Community Edition kan man kun ha en database kjørende, men flere databaser kan eksistere samtidig på maskinen.

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

2.4 Lag en graf

Neo4j tilbyr et forhåndsdefinert datasett kalt Movie Graph. Denne egner seg godt for innføring i Neo4j og Cypher og brukes forøvrig som grunnlag for en tutorial.

I Neo4j-Browser, klikk på knapp for "Write Code" under "Jump into Code" elementet og kjør "Movie Graph" skriptet. Grafen som blir opprettet her, vil bli brukt i noen eksempler i denne rapporten. "Movie Graph" opprettes og lagres i databasen **graph.db**. Du kan nå kjøre spørringer mot denne databasen.

2.5 Opprette og endre aktive databaser

Om fillokasjoner for Neo4j:
https://neo4j.com/docs/operations-manual/current/configuration/file-locations/

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 med det navnet som er definert i konfigurasjonsfilen.

For Debian-systemer (herunder Ubuntu) er databasene lokalisert i katalogen **/var/lib/neo4j/data/databases/**.

List innhold:

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

Konfigurasjonsfilen er lokalisert her: **/etc/neo4j/neo4j.conf**. For å opprette en ny database, må man åpne filen med skriverettighet og definere en ny aktiv database.

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
…
```

Innstillingen man må sette er helt i starten av konfigurasjonsfilen. Dupliser linjen med med **active_database** og kommenter ut den gamle med **#.** Velg et navn på den nye databasen din. Vi har valgt å kalle den 'ny-database.db'. Lagre og lukk filen.

Når du starter Neo4j, leses **neo4j.conf** og den nye aktive databasen vil bli opprettet. Ønsker du en bekreftelse på at databasen er opprettet, kan du liste innholdet i katalogen **databases**.

List innhold:

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

Jobber du lokalt og ikke ønsker å styre med autentisering, kan dette også settes opp i konfigurasjonsfilen

neo4j.conf:

```
…
# Whether requests to Neo4j are authenticated
# To disable authentication, uncomment this line
```

```
dbms.security.auth_enabled=false
…
```

2.6 Erfaringer

2.6.1 Installasjon

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. Dersom du har flere, må du velge en standardversjon som Neo4j kan bruke.

2.6.2 Spørrespråket CQL (Cypher Query Language)

Spørrespråket som Neo4j har utviklet for interaksjon mot databaser, er intuitivt og ligner til dels på MySQL. Det anbefales at leser tar seg tid til å gjøre de treningsmodulene som Neo4j-Browser tilbyr. Gjør oppslag på nye klausuler underveis og sett deg inn i de mønstre som er gjeldende for din database.

2.6.3 Generelt

Det er nyttig å sette seg inn i filstrukturen til Neo4j, spesielt hvor databasefilene og konfigurasjonsfilene ligger.

Neo4j Community Edition, som vi benytter, støtter som sagt ikke flere instanser av databaser. Denne versjonen egner seg veldig bra for å lære seg Neo4j, men Neo4j Enterprise Edition vil være mer passende i større utviklingsmiljøer.

3 Neo4j og data i JSON format

3.1 Valg av format

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 lese datasettene.

Datasettet vi arbeider med er Administrative Enheter. Dataeier er Kartverket og datasettet er tilgjengelig i formatene GeoJSON, GML og SOSI på GeoNorge sine nettsider.

Neo4j har ikke innebygget støtte for å arbeide med JSON, GML og SOSI. Det er kun støtte for CSV. For å løse dette fant vi frem til APOC, et 3.parts bibliotek for Neo4j som tilbyr en rekke brukerdefinerte prosedyrer. Biblioteket har blant annet en prosedyre for lasting av JSON-filer. Dette, kombinert med at vi er fortrolige med å bruke JSON, var årsaken til at vi valgte å hente ut datasett i GEOJSON-formatet.

Vi benyttet denne blogposten som en introduksjon og veiviser for installasjon og bruk av APOC: https://neo4j.com/blog/intro-user-defined-procedures-apoc/

I dette kapittelet vil det være en del bruk av Cypher Query Language. Vi anbefaler derfor at du har grunnleggende kjennskap til Cypher Query Language.

3.2 Installasjon av APOC

3.2.1 Finn din Neo4j versjon

Sjekk neo4j versjon i shell:

```
usr$ neo4j –version
neo4j 3.3.3
```

3.2.2 Finn passende APOC versjon

Det finnes flere APOC-versjoner, og disse er kompatible med forskjellige neo4j-

versjoner. En kompatabilitetsoversikt 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. Denne versjonen av APOC er kompatibel med alle Neo4j 3.3.X utgivelser.

3.2.3 Installasjon av APOC

Last ned .jar filen fra det repositoriet som samsvarer med din Neo4j-versjon. Lenke til repositoriet vi har benyttet finnes her: https://github.com/neo4j-contrib/neo4j-apoc-procedures/releases/3.3.0.2.

Kopier/flytt den til **`/plugins`** mappen. For Debian-systemer er stien til mappen **`var/lib/neo4j/plugins`**.

Bruker du et annet operativsystem, sjekk denne oversikten for å finne fil-lokasjoner til operativsystemet ditt: https://neo4j.com/docs/operations-manual/current/configuration/file-locations/

Nå som APOC er installert, må man gi Neo4j rettigheter til å kjøre prosedyrene som APOC tilbyr. Dette gjøres via Neo4j sin konfigurasjonsfil: ***/etc/neo4j/neo4j.conf***.

Åpne denne filen med din foretrukne tekstbehandler (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 specific procedures.
# Uncomment this line to enable unrestricted acces for procedures.
Dbms.security.procedures.unrestricted=apoc.*
…
```

Søk etter linjen **`dbms.security.procedures.unrestricted=apoc.*`** i konfigurasjonsfilen. Dersom den ikke eksisterer, legg den til. Kommentarene ovenfor er egendefinerte og beskriver hva innstillingen gjør. Denne innstillingen lar deg kjøre alle prosedyrer som er definert i apoc pakken. Disse tillatelsene kan finmaskes ved å spesifisere underpakker og prosedyrenavn etter behov.

3.2.4 Verifisere installasjonen

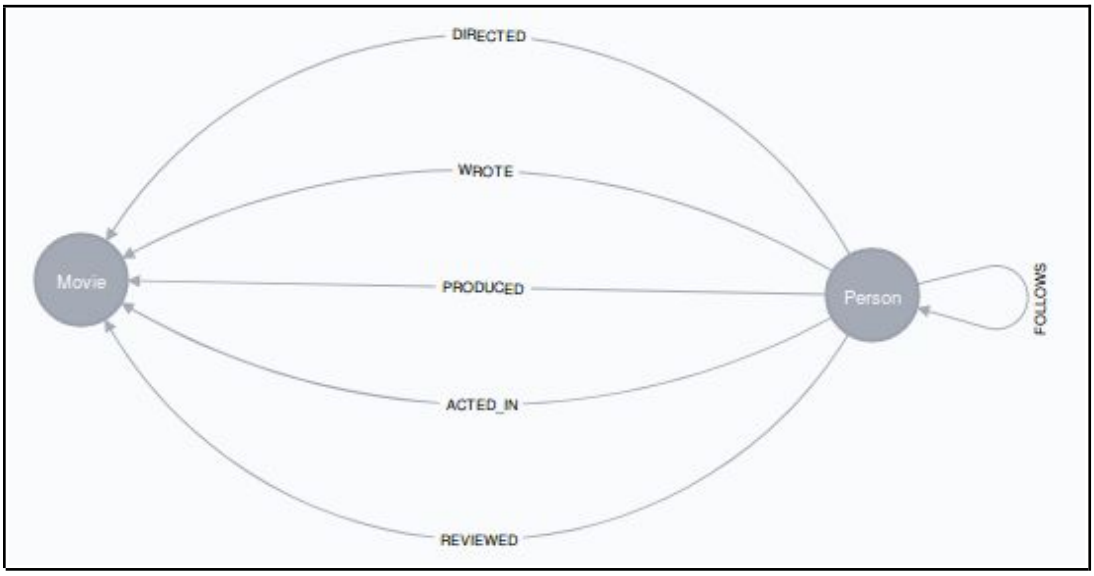
For å verifisere at installasjonen er vellykket, kan man kjøre et kall på en av de brukerdefinerte prosedyrene som APOC supplerer.

Start Neo4j-databasen og åpne Neo4j-Browser. I dette eksempelet tar vi utgangspunkt i at “Movie Graph” skriptet er kjørt på databasen.

Generer metagraf for databasen:

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

Denne prosedyren returnerer en graf som inneholder noder og forhold som eksisterer i grafen på et overordnet nivå. For “Movie Graph” vil metagrafen se lik ut som grafen i figuren under.



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

Dokumentasjon for APOC finnes her: https://neo4j-contrib.github.io/neo4j-apoc-procedures/

3.3 Behandle JSON med APOC

Relevante linker for Cypher:
https://neo4j.com/docs/developer-manual/current/cypher/

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 enkel 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.

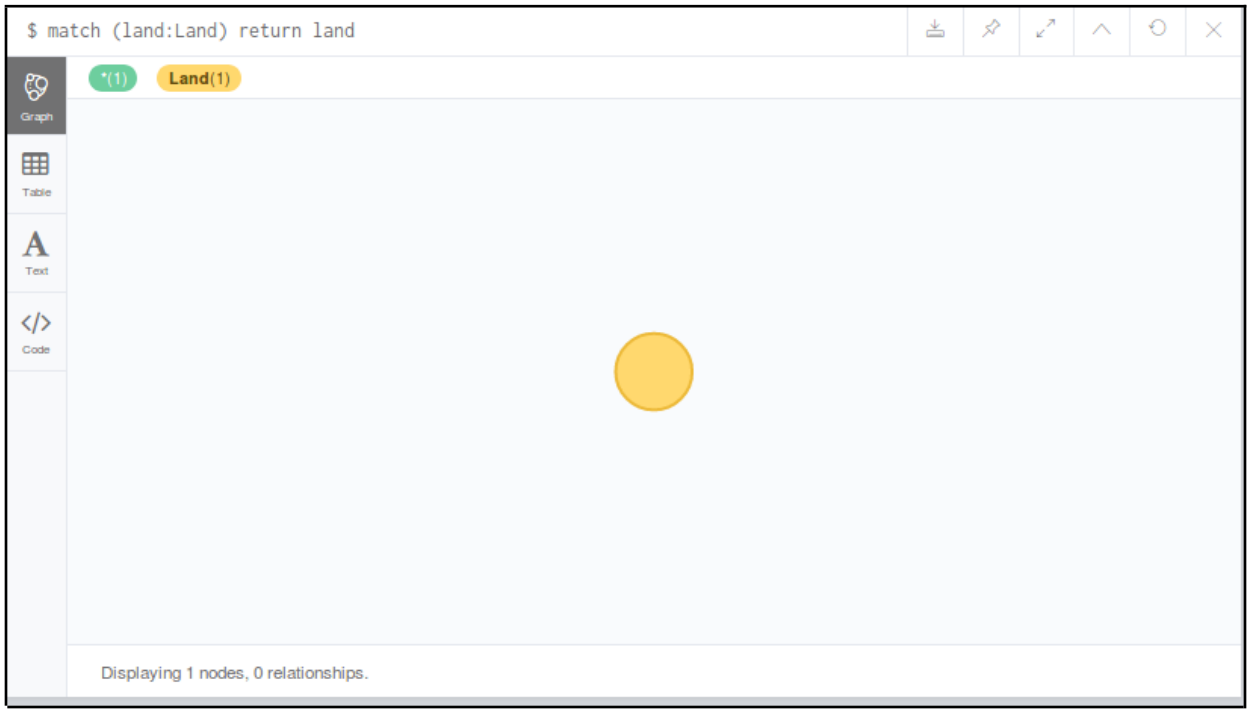
Dersom man skal bruke JSON-filer lagret lokalt, må man gi APOC-prosedyrene tilgang til å importere lokale filer.

neo4j.conf:

```
…
# Allows local file import for APOC
apoc.import.file.enabled=true
…
```

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

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



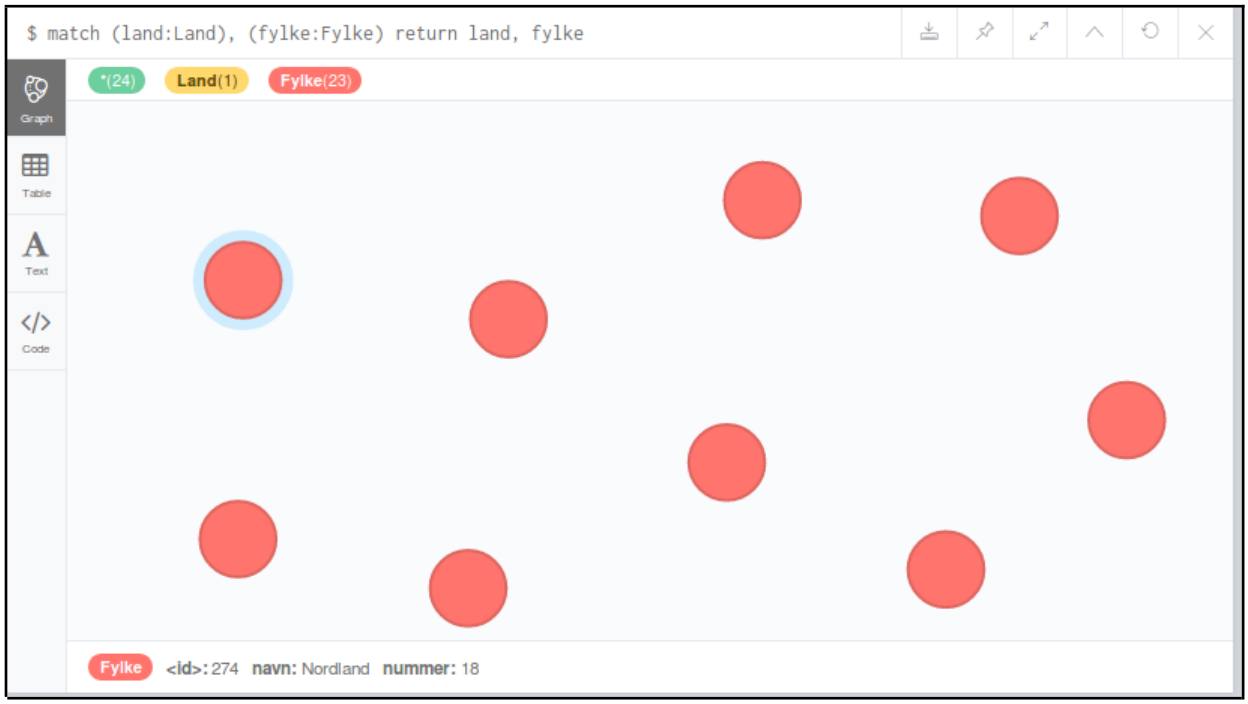
Hent JSON-data og opprett noder for hvert fylke:

```
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:

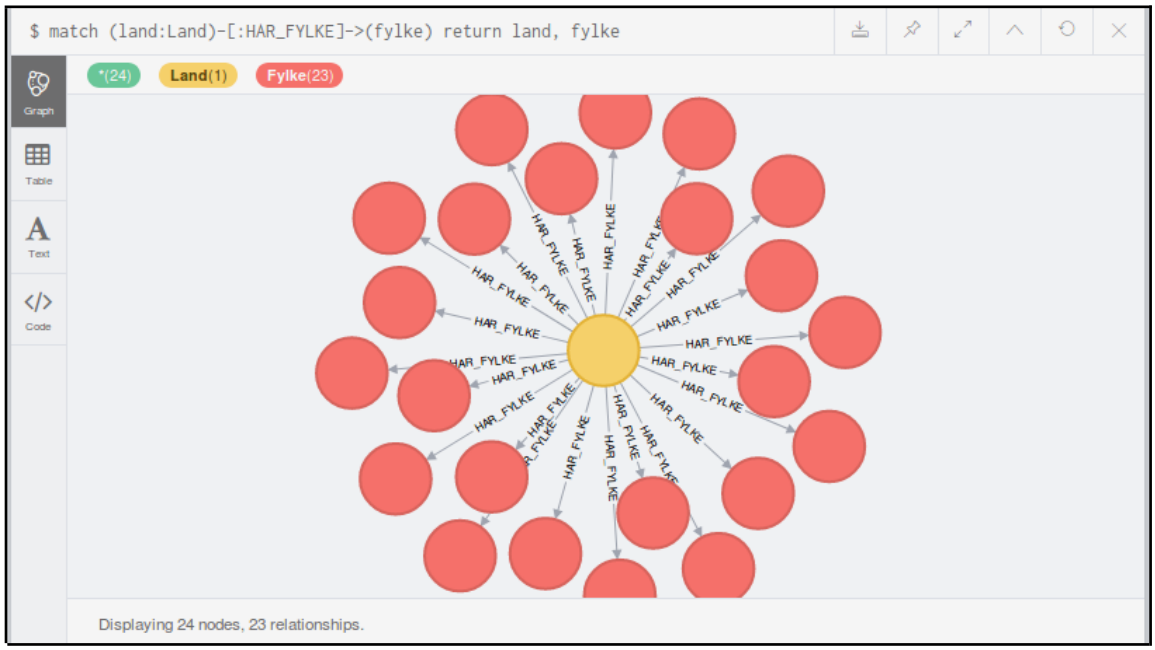
```
cql$ // Forutsetter at APOC har tilgang på lokal import av filer
     WITH "file://///path/to/json/data.json"
     … identisk som spørringen mot API
```

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:

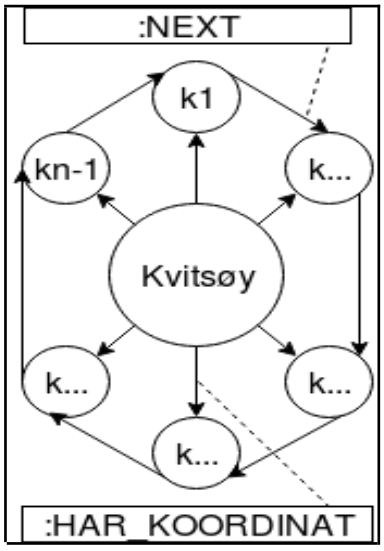
```
cql$ MATCH (f:Fylke), (l:Land{navn:"Norge"})
     MERGE (l)-[:HAR_FYLKE]->(f)
```



3.4 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 Kvitsøy kommune egnet seg fint for for dette formålet, og vil være datagrunnlaget vi benytter i det kommende eksempelet..Vi er i hovedsak interessert i koordinatene som utgjør grensene til de administrative enhetene, henholdsvis fylke og kommunegrense. For Kvitsøy kommune, er vi interessert i koordinatene som utgjør kommunegrensen.

3.4.1 Kartlegge hvordan grafen skal se ut

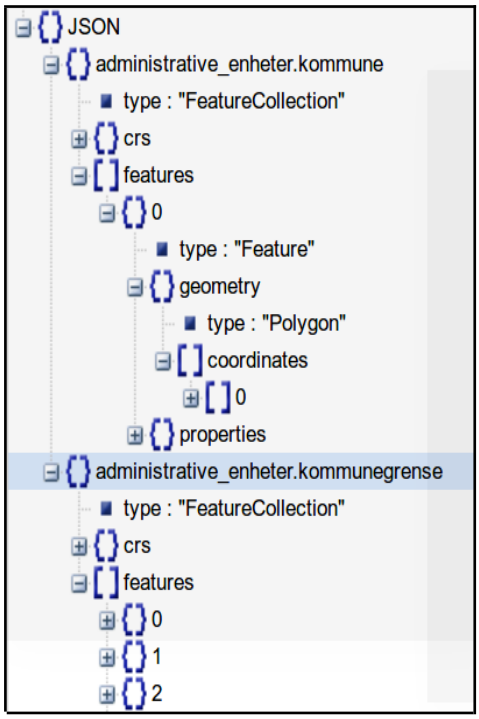


Vi ønsker å opprette en node som representerer Kvitsøy kommune, og denne noden har relasjonen [:HAR_KOORDINAT] mot noder som inneholder koordinater. Koordinatnodene skal ha en [:NEXT] relasjon til neste koordinatnode.

Til venstre ser 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 har 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. Vi har nå en modell som grunnlag. Neste steg er å kartlegge strukturen i datasettet.

3.4.2 Datasettets utforming og overføring til graf

Her er et utsnitt av datasettet for Kvitsøy kommune. Vi har benyttet



http://jsonviewer.stack.hu/ for en menneskevennlig visning av JSON- filer. Objekter i illustreres med **{}**, og tabeller illustreres med **[]**. Kommunegrensene er å få i to typer features, enten som "Polygon", eller som en samling av "LineString"- segmenter.Vi har besluttet vi å bruke "Polygon".

Polygon er et sett av koordinater, og i her utgjør settet en flate som beskriver kommunegrensen til Kvitsøy. Punktene ligger sekvensielt, slik at **punkt[i]** sin **[:NEXT]** blir **punkt[i+1]**. For **punkt[n-1]** blir **[:NEXT]** relasjonen **punkt[0]**. Koordinatlistene i to nabokommuner kan ha definert polygonene i motsatt rekkefølge. Dette muliggjør to **[:NEXT]** relasjoner i hver sin retning mellom to koordinater.

Alle disse punktene har også en node "Kommune" som har en **[:HAR_KOORDINAT]** relasjon mot alle koordinatene for sin kommunegrense. Dette stemmer veldig godt overens med den grafen vi la frem som forslag over.

3.4.3 Innlasting av JSON-data for en kommune

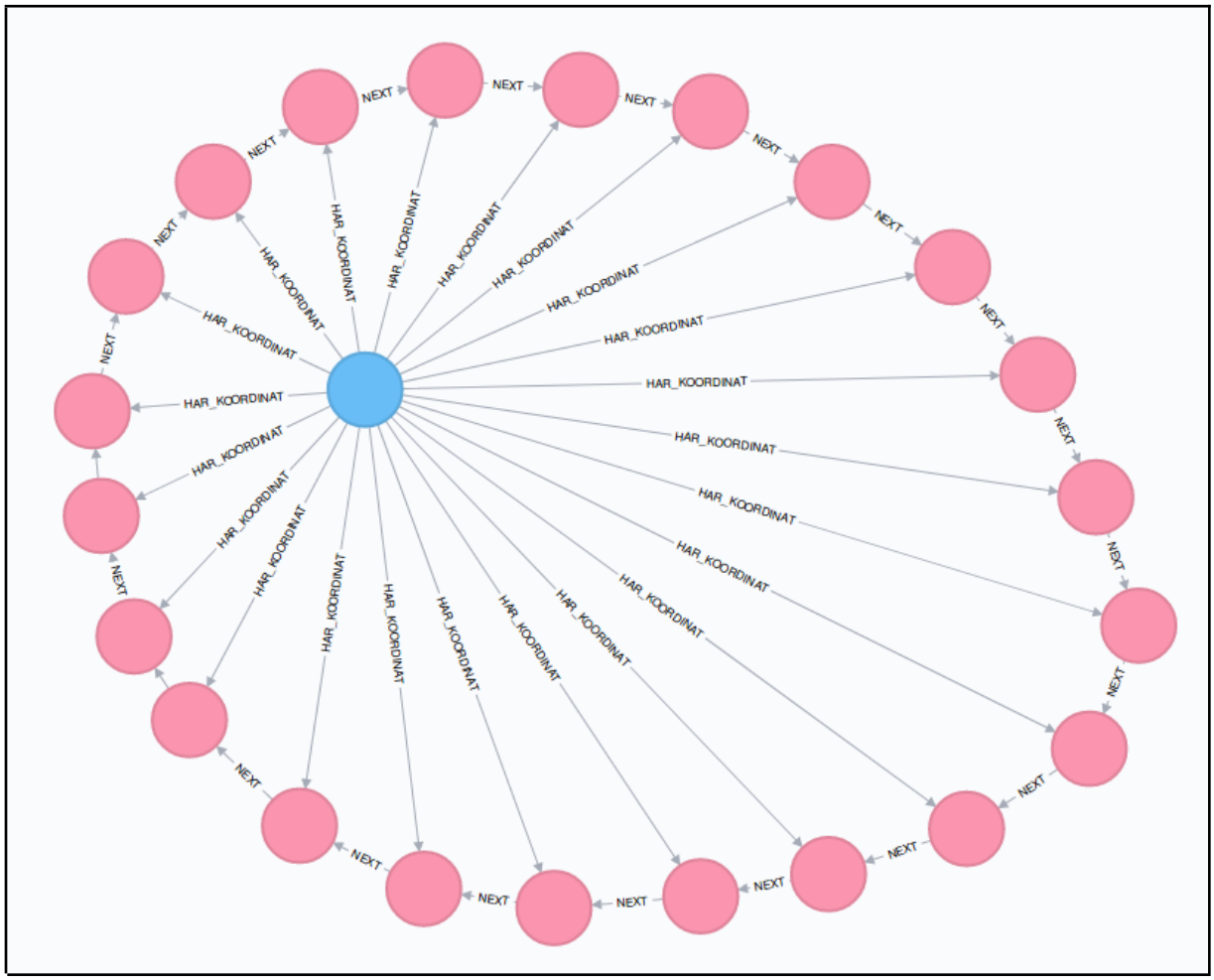
Først må vi pakke ut JSON-data. Datasettene er tilgjengelige som nedlastbare filer, og APOC må derfor gis **tilgang til lokal import av filer.**

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, må denne endringen gjøres for hvert fylke.

Som vi nevnte tidligere, benytter vi datasettet for Kvitsøy kommune, som er en liten øy i Rogaland. Kvitsøy sin kommunegrense består av 23 koordinater, som gjør det enkelt å validere innsettingen visuelt i Neo4j-Browser i etterkant. Følgende spørring oppretter en graf av dette datasettet. En mer detaljert forklaring av spørringen og

hva som skjer på hver linje kommer senere.

```
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)
```



Som man kan se av figuren ovenfor, er resultatet veldig bra for innsetting på kommunebasis. Dersom vi overfører denne fremgangsmåten for hver kommune i et fylke, skal dataen blir korrekt lenket.

3.4.4 Innsetting av JSON-data for utvalgte kommuner

Vi har valgt å benytte datasettet for Telemark fylke, og vi tar heretter for oss innsetting utvalgte kommuner i dette fylke. Vi vil også avslutte kapittelet med innsetting av alle kommuner i Telemark. Denne teknikken kan enkelt overføres til datasett for andre fylker, da strukturen er lik.

Vi ser at innsetting av større datasett vil ta tid, da innsetting ikke skjer effektivt. Dette kommer vi tilbake til senere. På bakgrunn av dette, har vi redusert antallet kommuner vi ønsker å lage en graf av. Vi har valgt å sette inn 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$ 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 [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:

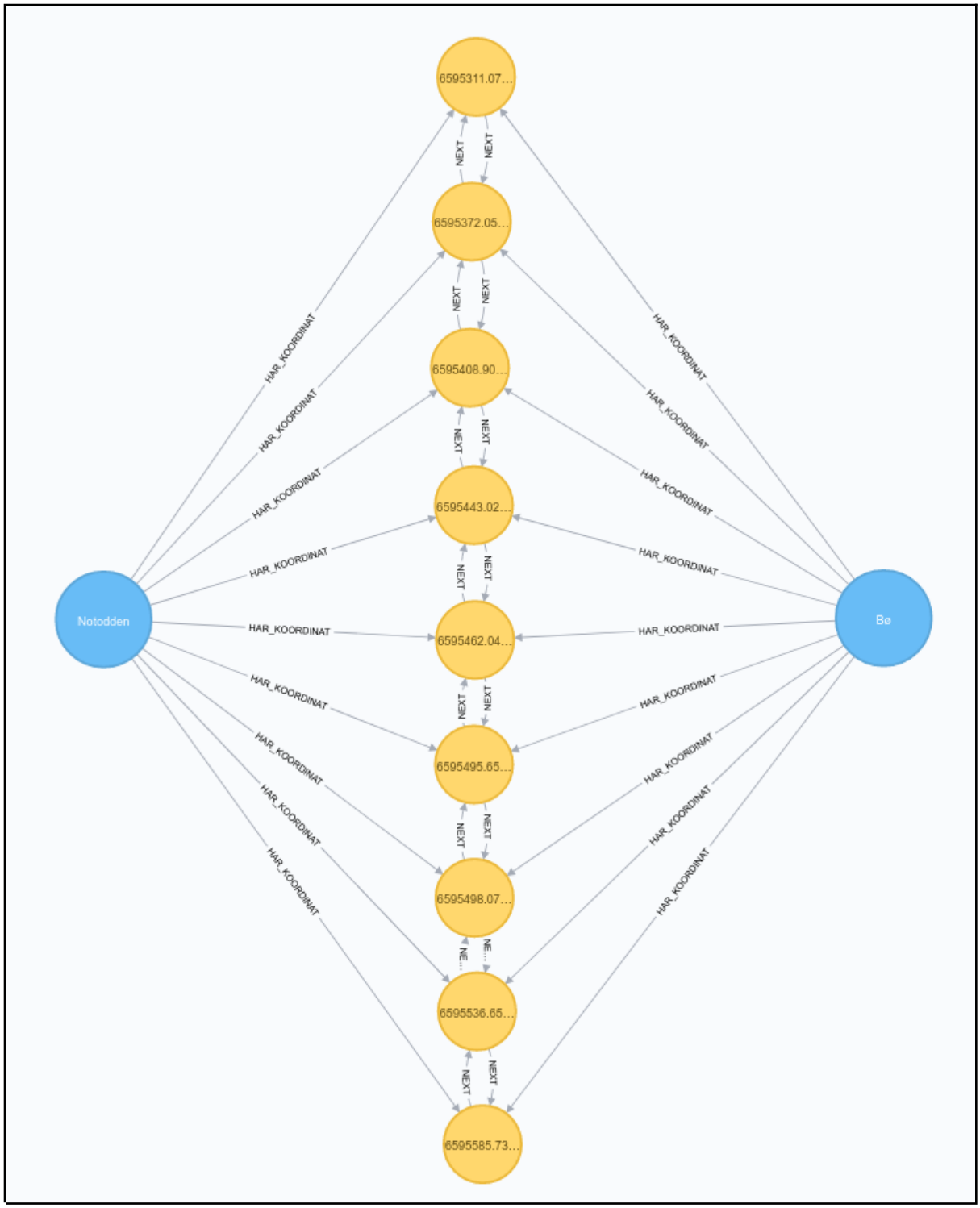
```
...   MERGE (k1)-[:NEXT]->(k2)
```

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

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

Dette skriptet ble kjørt på en prosjektdeltakers maskin. Det tok 6.5 minutter å prosessere disse 5 kommunene. Vi kjørte også en annen variasjon av dette skriptet som lager en graf av alle kommunene i et fylke som tok 64 minutter å utføre. Vi sier litt om årsaken til dette i neste avsnitt: Innsetting av JSON-data for alle kommuner i et fylke.

Utsnitt av grensen mellom Bø og Notodden:



3.4.5 Innsetting av JSON-data for alle kommuner i et fylke

For innsetting av alle kommuner og tilhørende kommunegrenser i et fylke, er fremgangsmåten veldig lik som for innsetting av utvalgte fylker. Forskjellen er at man itererer over alle kommunene i et fylke i stedet for utvalgte indekser. Dette gjøres med følgende: **`UNWIND RANGE(0, SIZE(kommuner)-1) as kidx`**.

Det var her tidsbruk for innsetting begynte å bli et problem. Som vi nevnte tok det 64 minutter å sette inn dataen. Årsaken til denne tidsbruken er at **`MERGE`** oppretter 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.

I skriptet under tar vi igjen utgangspunkt i Telemark fylke.

```
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)
```

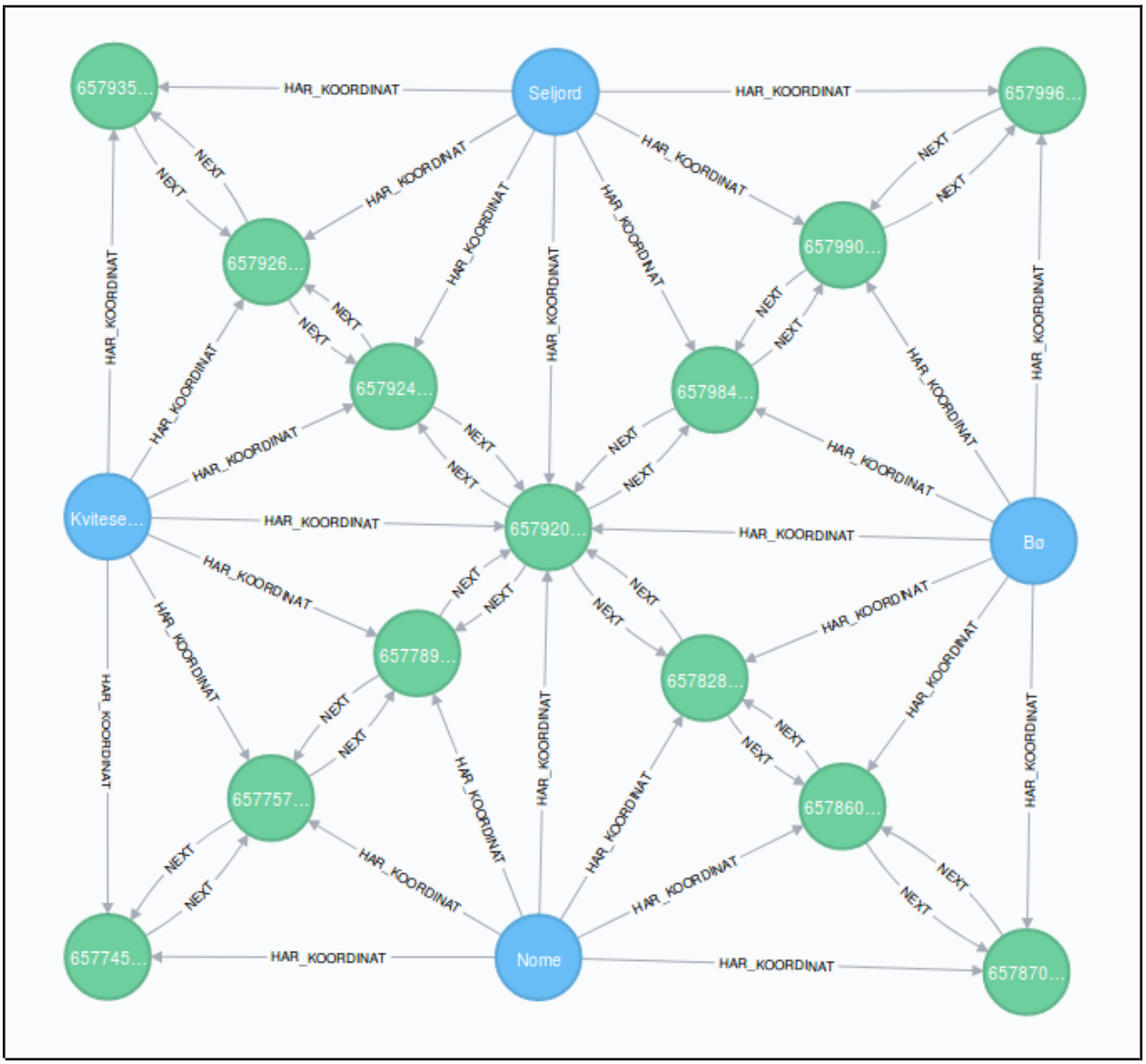
3.4.5.1 Eksempler på spørringer etter innsetting

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 kontrollere at relasjonene ble opprettet korrekt. Etter litt flytting av noder i det grafiske grensesnittet, ble resultatet slik som grafen under.

Spørringen som ble kjørt:

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



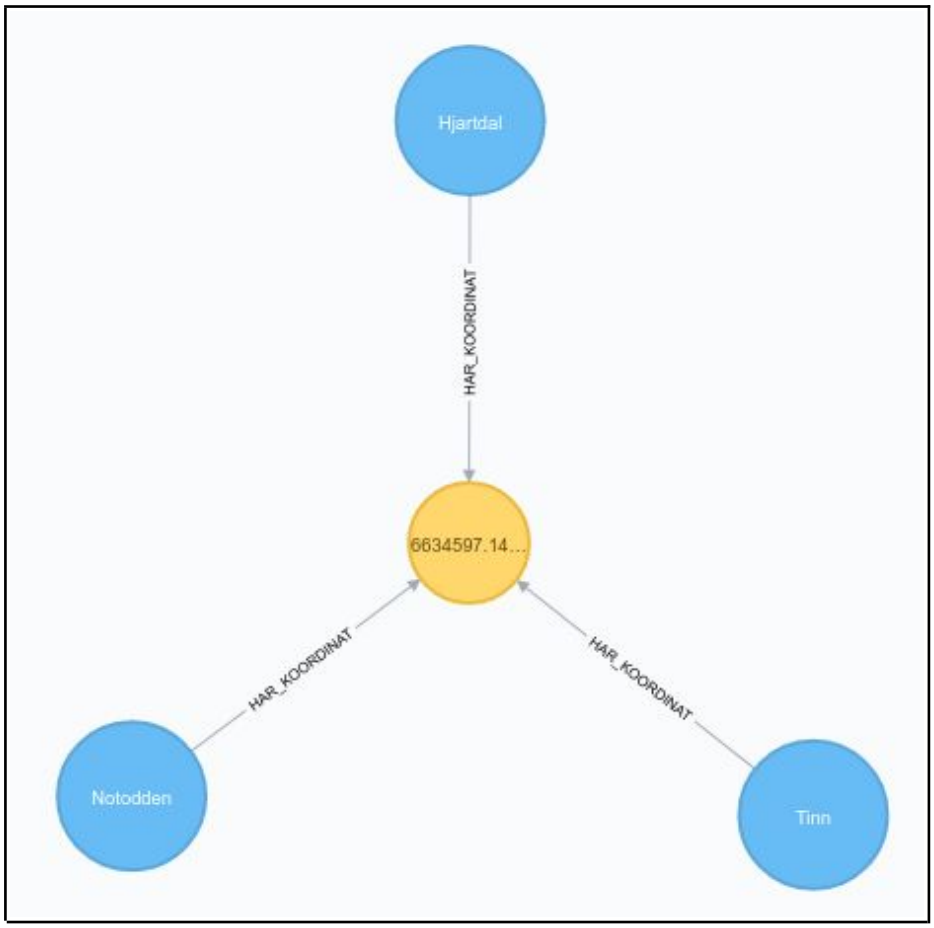
Vi har brukt http://finnposisjon.test.geonorge.no/ for å finne posisjonen som spørringen resulterte resulterte i. I figuren under ser man et utsnitt av kartet rundt

dette punktet. Dette gjorde vi for å verifisere at nodene og relasjonene er satt korrekt. Dette stemmer bra overens med resultatet vi fikk fra spørringen.

Felles grense mellom Notodden, Tinn og Hjartdal:

Spørring for figur nedenfor:

```
cql$ MATCH (hjartdal:Kommune{navn:"Hjartdal"})-[:HAR_KOORDINAT]->(k)
     <-[:HAR_KOORDINAT]-(tinn:Kommune{navn:"Tinn"}),
     (k)<-[:HAR_KOORDINAT]-(notodden:Kommune{navn:"Notodden"})
     RETURN hjartdal, tinn, notodden, k
```



3.5 Erfaringer: Neo4j, GEOJSON og datasettet

3.5.1 JSON og Neo4j

Neo4j har som sagt ikke innebygget støtte for å behandle JSON. For å behandle JSON i Neo4j, har vi tatt i bruk et åpent 3.parts bibliotek ved navn APOC. En samling av 3.parts biblioteker for Neo4j finnes her: https://github.com/neo4j-contrib. Neo4j kunne med fordel ha lenket til en ressurs fra Neo4j-Browser som gir utviklere en introduksjon i behandling av JSON.

Neo4j har støtte for håndtering av CSV filer, og de har en egen treningsmodul som omhandler migrering av data fra en relasjonsdatabase til graf-format. Dette er et tema som denne erfaringsrapporten ikke tar for seg. Leser oppfordres til å gjøre dette på egenhånd.

Selve behandlingen av JSON i Neo4j er forholdsvis enkel, gitt at man har grunnleggende forståelse av Cypher Query Language.

3.5.2 Prosesseringstid

Innsetting av kommunegrenser i Telemark tok 64 minutter. Denne tidsbruken er en følge av vårt bruk av **`MERGE`** under innsetting for å hindre at noder og relasjoner dupliseres. **`MERGE`** forsøker å matche mønstrene som allerede eksisterer i databasen, og oppretter et nytt mønster dersom det ikke finnes fra før. Tiden det tar å kjøre skriptet vi detaljerte i avsnittet om Innsetting av JSON-data for alle kommuner i et fylke er årsaken til at vi ikke anbefaler denne fremgangsmåten for datasett av betydelig størrelse. Man kan redusere tidsbruken ved å bruke **`CREATE`**, som vil opprette noder og relasjoner uten kontroll av duplikater. Dersom datasettet man ønsker å lage en graf av ikke har strenge krav til duplikater, er det ikke problematisk å bruke denne fremgangsmåten.

3.5.3 Egnethet for innsetting

Neo4j, Cypher og APOC-biblioteket er egnet for innsetting av mindre datasett hvor det ikke er ønskelig med duplikater. For Administrative Enheter vil dette være innsetting av data per fylke, eller et utvalg kommuner i et fylket. Generelt så må man vurdere fremgangsmåte ut i fra datasettet man har valgt.

3.5.4 Alternativ fremgangsmåte

For å redusere prosesseringstiden for innsetting av datasettet Administrative Enheter, har vi behov for mer detaljert kontroll over nodene og relasjonene som lages. Neo4j har utviklet et Java-API som er godt dokumentert, og vi har valgt å gå videre med en løsning som benytter dette API-et.

Ved å lage en Java-applikasjon, har vi tilgang på de data- og kontrollstrukturene som eksisterer i Java. Verktøyene som Java tilbyr kan bidra til duplikatsjekk på noder og relasjoner og sørge for korrekt lenking av data.

4 Neo4j og Java Embedded Database

4.1 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 katalogen der databasen ligger, og instansierer den som en tjeneste i Java-applikasjonen. Denne teknikken tillater utviklere å bruke Neo4j sitt Java-API direkte.

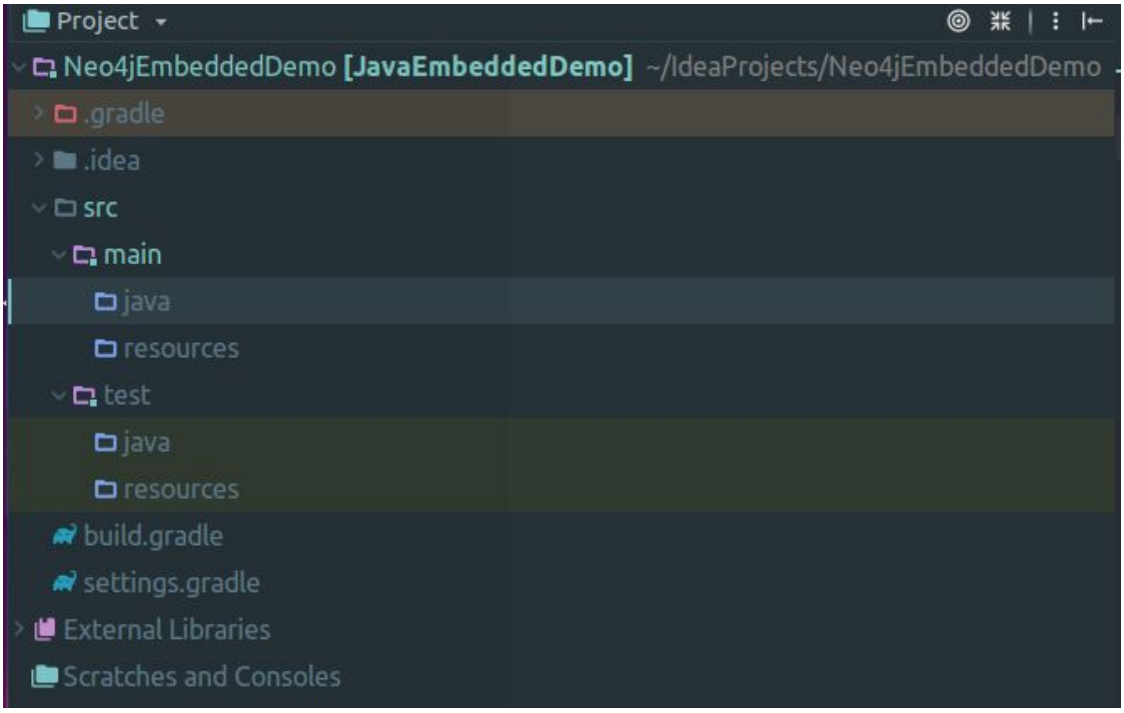
Neo4j tillater kun at en versjon av en database er instansiert på en maskin. Dette medfører at man må stenge ned Neo4j-tjenesten, eller satt en annen database som aktiv, når man kjører Java-applikasjonen som embedder databasen.

4.2 Neo4j Java-API

Vi har brukt IntelliJ som utviklingsmiljø for applikasjoner som embedder en Neo4j-database. 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 og lagt inn en avhengighet til Neo4j for å oppnå dette. Mer informasjon embedding av databasen og hvordan kan gjøres finnes her: https://neo4j.com/docs/java-reference/current/tutorials-java-embedded/.

Fremgangsmåten for Intellij og Gradle er som følger:

1. Opprett et nytt Gradle prosjekt i IntelliJ. GroupID og ArtifactID velger dere selv. Følg veiviseren og opprett prosjektet med standardinnstillinger. Prosjektstrukturen 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:

```
dependencies {
…
  implementation "org.neo4j:neo4j:3.3.3"
  // neo4j:x.x.x – erstatt x.x.x med din neo4j versjon.
…
}
```

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

4.3 Brukere og rettigheter

Før du går i gang med programmering mot databasen din, har vi litt informasjon vedrørende brukere og rettigheter på filer. Dette gjelder dersom du bruker Linux som operativsystem.

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

4.3.1 Valg av rettigheter

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 med **systemctl**, slik at databasekatalogen blir opprettet.

Skift mappe til /var/lib/neo4j/data/databases:

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

Legg brukeren din til i neo4j gruppen:

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

Filene som allerede ligger i databasekatalogen har feil rettigheter i forhold til hva Java-applikasjonen trenger.

Endre rettigher på alle filene som ligger i mappen:

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

Filene har nå grupperettighetene **read** og **write**.

Vi ønsker også at alle nye filer som opprettes i denne katalogen får tilsvarende grupperettigheter, selv om eier kan være annerledes

Endre grupperettigheter på databasekatalog:

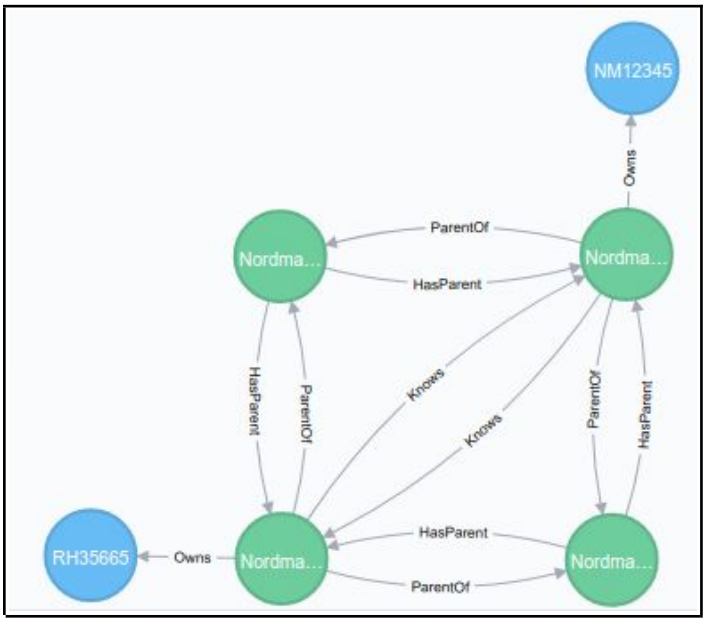
```
usr$ sudo chmod g=rwxs nydb.db
```

Nye filer som opprettes i denne katalogen arver nå grupperettighetene **read**, **write** og **execute.**

4.4 Embedde databasen i en Java-applikasjon

JavaDoc for Neo4j Java API: https://neo4j.com/docs/java-reference/3.3/javadocs/

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 du her: 8.11.Kildekode: Neo4jEmbeddedDemo



I denne seksjonen lager vi for enkelhets skyld 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 referanser til databasen i main-metoden. Du får nå instansene for å jobbe mot databasen "**nydb.db**" i Java-applikasjonen. Oppretting av noder og relasjoner skjer ved hjelp av objektmetoder 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 node- og relasjonstyper man skal benytte i databasen. Dette gjøres ved hjelp av enum typer som implementerer grensesnittene Label og RelationshipType, definert av Neo4j sitt Java-API. Vi definerer disse typene 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 deretter avslutter når innsetting og uthenting er ferdig. Grafdatabasen må så stenges ned, slik at Java-applikasjonen ikke lenger har lås på databasefilen. Hvordan man håndterer låsing i applikasjonen kommer vi tilbake til.

Under testing kan det være lurt å slette alle noder og relasjoner før man setter inn data. Dette utfører vi med en "raw" 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: #8.11.Kildekode: Neo4jEmbeddedDemo

```
…
/* 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 som enum, henholdsvis **`NodeType`** og **`RelationType`**. Ønsker man flere typer, kan de legges til her.

Dersom du nå kjører denne klassen (applikasjonen), 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.

Før du kjører en Java-applikasjon som embedder en Neo4j-database, må du stoppe databasetjenesten. Dersom du ikke gjør det, vil applikasjonen kaste unntak på at en annen prosess har lås på databasen. Det programmet som bruker databasefilen, får en lås på databasen, og vil ha enerett på å bruke databasen til låsen er åpnet. Dette har riktignok ingen negative konsekvenser, gitt at applikasjonen ikke gjør noe destruktivt før den sjekker tilgang på databasen. Derimot er det svært viktig å håndtere unntak som følge av feil under programkjøring. Uhåndterte feil under utførelse av en transaksjon kan føre til at databasen forblir i en transaksjonstilstand og at låsen opprettholdes.

Koden under representerer minstekravet for unntakshåndtering, kodeblokken er en utvidet versjon av den i avsnittet over, med ny kode uthevet. Dette forklares under:

```
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();
} catch (Exception e) {
    e.printStackTrace();
    if (tx != null)
        tx.failure();
} finally {
    if (tx != null)
        tx.close();
}
graphdb.shutdown();
```

Oppretter en transaksjon, som igjen setter lås på databasen:

```
tx = graphdb.beginTx();
```

Registrerer transaksjonen som suksessfull:

```
tx.success();
```

Registrerer transaksjonen som feilet:

```
tx.failure();
```

Utfører en commit eller rollback basert på hvorvidt transaksjonen er merket som suksessfull eller feilet:

```
tx.close();
```

På denne måten sikrer man at en feilkjøring uansett ville avslutte riktig, gitt at selve JVM'en ikke krasjer. Når en transaksjon av hvilken som helst grunn skulle feile, rulles den tilbake, og programmet lukkes. Tidligere fullførte transaksjoner beholdes naturligvis.

Start databasen med **systemctl**, åpne det grafiske brukergrensesnittet og kjør følgende spørring. Du vil nå se den grafen du nettopp har opprettet.

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

4.5 Erfaringer: Embedded Neo4j database

4.5.1 Bruke Neo4j sitt Java-API

Å få tilgang til Neo4j sitt Java-API er veldig enkelt, spesielt dersom man bruker IntelliJ som IDE og Gradle som build-verktøy. Bruker du en annen IDE og annet build-verktøy, skal fremgangsmåten være veldig lik. API-et er også enkelt å ta i bruk, gitt at man har kjennskap til Java og grafmodellen som Neo4j bruker.

4.5.2 Brukere og rettigheter

Når man embedder en database i Neo4j er det flere elementer man må ta høyde for. Dette gjelder spesielt rettigheter på filer og filhåndtering dersom man arbeider i Linux. Vi brukte en del tid på å løse dette. Det er derfor detaljert en fungerende løsning vedrørende filrettigheter under seksjonen Brukere og rettigheter.

5 Java-embedded og geografiske data

5.1 Kildekode/prosjekt

Kildekode er tilgjengelig på https://github.com/gitsieg/Neo4Java. Du kan laste dette ned, åpne prosjektet i IntelliJ og kjøre det på egen maskin. Instruks for hvordan du kan gjøre dette, finner du i Kjøre prosjekt på egen maskin.

5.2 Bakgrunn

Vi valgte å gå for denne løsningen for innsetting av geografiske data. Årsaken var 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 over til Java-applikasjonen.

5.3 Prosessering av datasettet

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.

Vi kommer til å referere til klasser i applikasjonen, og klassediagrammet finner du her: Klassediagram Neo4Java

Dataene prosesseres fylke for fylke, og innad hvert fylke prosesseres hver kommune. Koordinatdataene ligger i korrekt rekkefølge i datasettene og danner grunnlaget 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 Object Relational Mapping for å komme rundt dette.

I LibGraph er det definert to klasser; **Koordinat** og **Forhold**. Koordinat-objekter

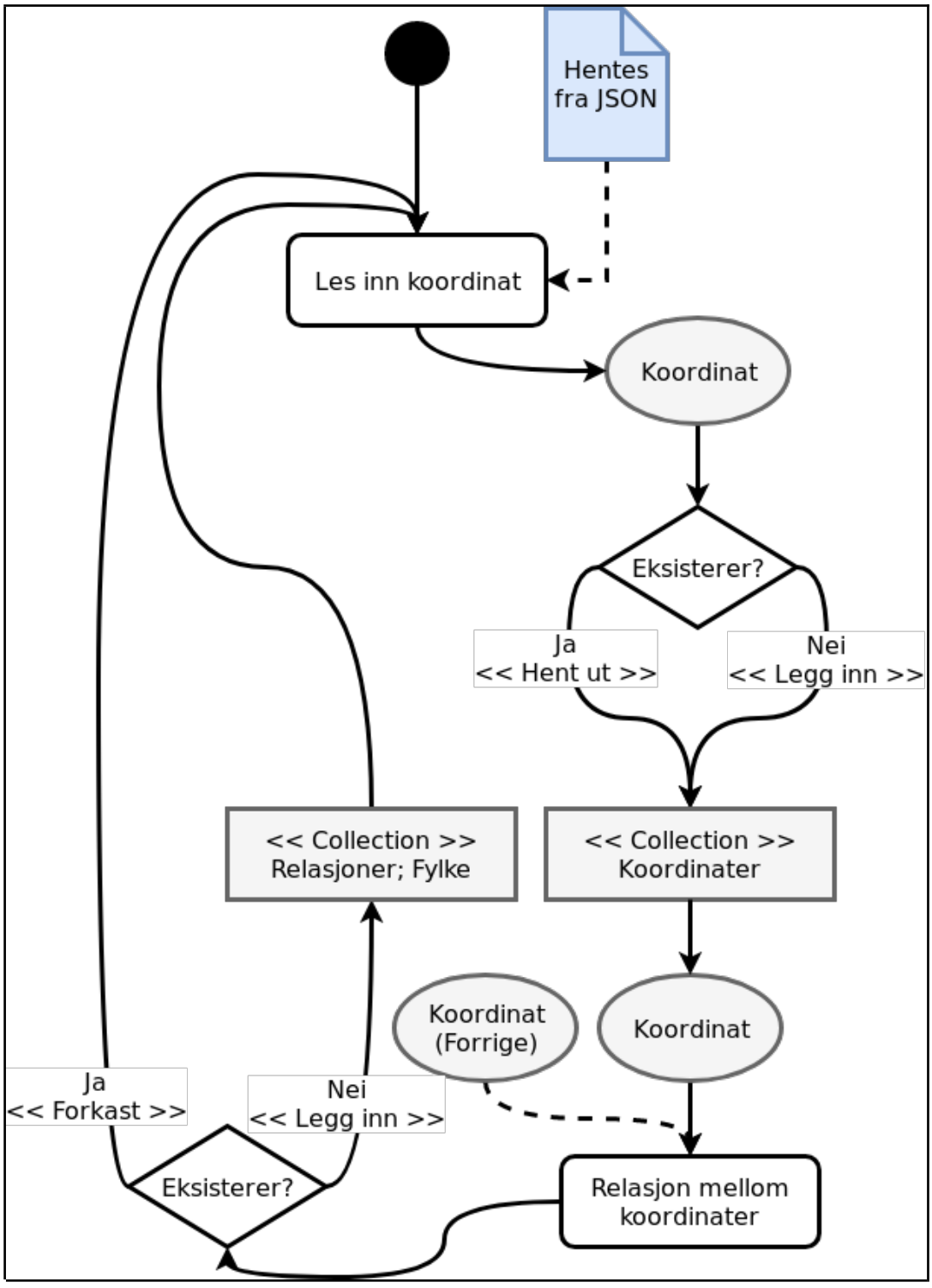
representerer koordinater slik de er gitt i datasettene, mens Forhold-objekter representerer relasjoner mellom koordinater enten i fylkes- eller kommunesammenheng. Klassene implementerer Comparable slik at vi kan bruke de i TreeSet-samlinger. TreeSet egner seg best i denne sammenhengen da både søk og 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, da duplikater ikke er ønskelig. 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.

Forløpet er visualisert i diagrammet under, gjennomløpingen av kommuner følger samme prinsipp og vi følte av den grunn det ikke var nødvendig å ta med.


Hentes fra JSON
Les inn koordinat
Koordinat
Eksisterer?
Ja
<< Hent ut >>
Nei
<< Legg inn >>
<< Collection >>
Relasjoner; Fylke
<< Collection >>
Koordinater
Koordinat (Forrige)
Koordinat
Ja
<< Forkast >>
Nei
<< Legg inn >>
Eksisterer?
Relasjon mellom koordinater

Denne prosessen er ganske omfattende, og for å forsikre oss om at ting forløper seg slik det skal har vi tatt i bruk command pattern. Vist i kodeblokken under:

```
…
// class LibGraph (...)
static void graphTransaction(GraphDatabaseService
                graphdb,TransactionCommand... commands){
  Transaction tx = null;
  try {
    for (TransactionCommand command : commands) {
      tx = graphdb.beginTx();
      command.performTransaction(graphdb);
      tx.success();
      tx.close();
    }
  } catch (Exception e){
    e.printStackTrace();
    if (tx != null)
      tx.failure();
  } finally {
    if (tx != null)
      tx.close();
  }
}
interface TransactionCommand {
  void performTransaction(GraphDatabaseService graphdb)
       throws Exception;
}

// class GraphLoader (...)
public void registerDatasets() {
  for (Node fylke : fylkeNoder) {
    LibGraph.graphTransaction(this.graphdb, txCmd -> {
      (...)
      // Innsettingskoden
      (...)
    }
  }
}
```

Arbeid mot databasen foregår ved kall på en metode *graphTransaction* i klassen LibGraph.

Denne metoden har ansvaret for start og slutt av en transaksjon samt feilhåndtering underveis. Klienten sender ved en (anonym) instans av TransactionCommand til denne metoden, hvis forhåndsdefinerte metode *performTransaction* inneholder det som skal utføres som en del av transaksjonen.

Ved å programmere mot et interface og la unntakshåndtering foregå implisitt ved hvert kall, forenklet vi arbeidet med innsetting.

5.4 Kjøre prosjekt på egen maskin

5.4.1 Hva du trenger

Du trenger:

- En datamaskin med Windows eller Linux.
- Du må ha IntelliJ installert.
- Neo4j og en Neo4j database av versjon 3.3.x.
- Du må kjenne til filstien til databasen på din maskin.
- Du må endre rettigheter på databasen i henhold til Brukere og rettigheter (dersom du benytter Linux som operativsystem).
- Ca 500Mb ledig diskplass til selve databasen.

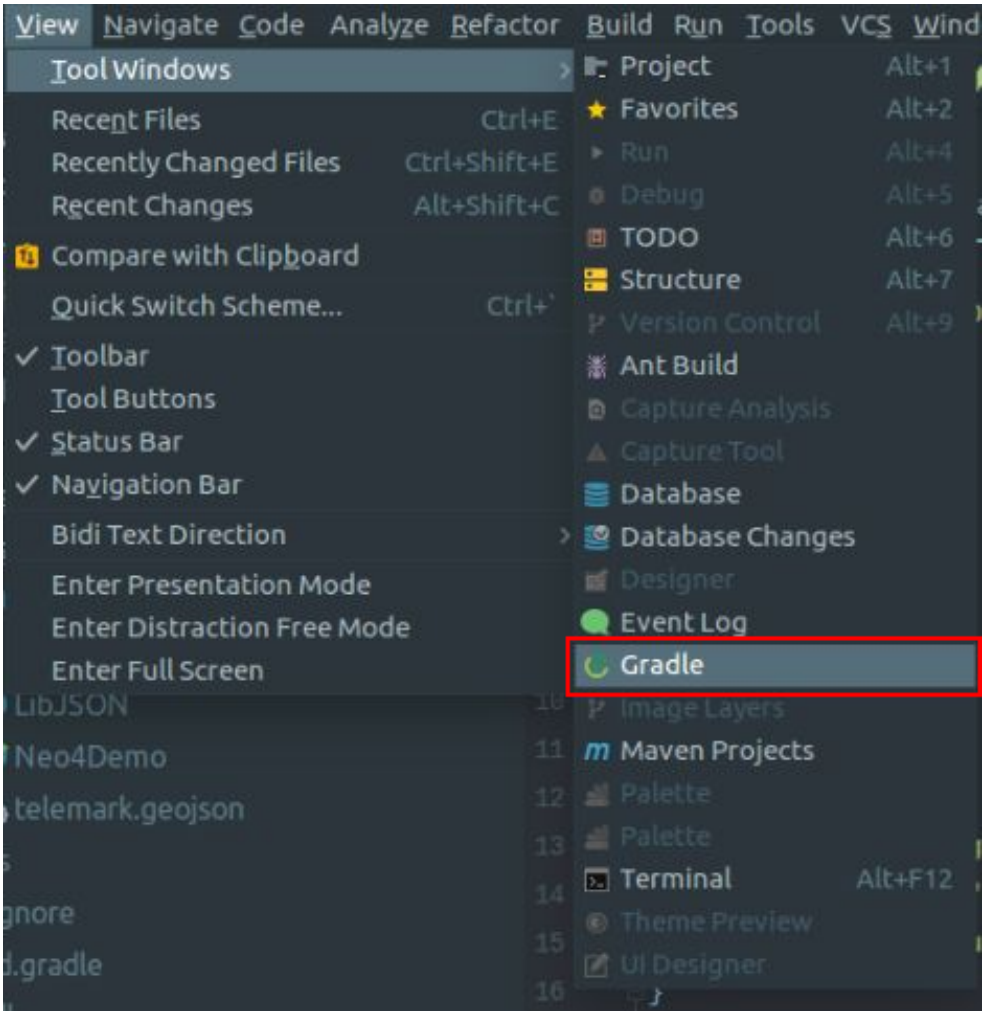
Nedlasting og oppsett av prosjekt:

Besøk https://github.com/gitsieg/Neo4Java og trykk “Clone or Download”. Last ned prosjektet som zip-fil. Pakk ut filen til en ønsket lokasjon. Alternativt kan du benytte git for å klone prosjektet. Åpne prosjektet i IntelliJ. Datasettene følger med i nedlastingen, og ligger i **res**-katalogen til prosjektet.

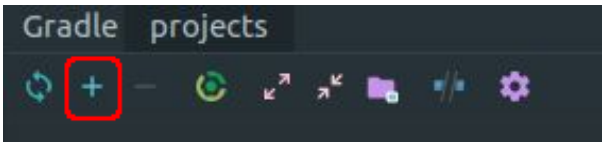
5.4.2 Registrere prosjektet som et Gradle-prosjekt

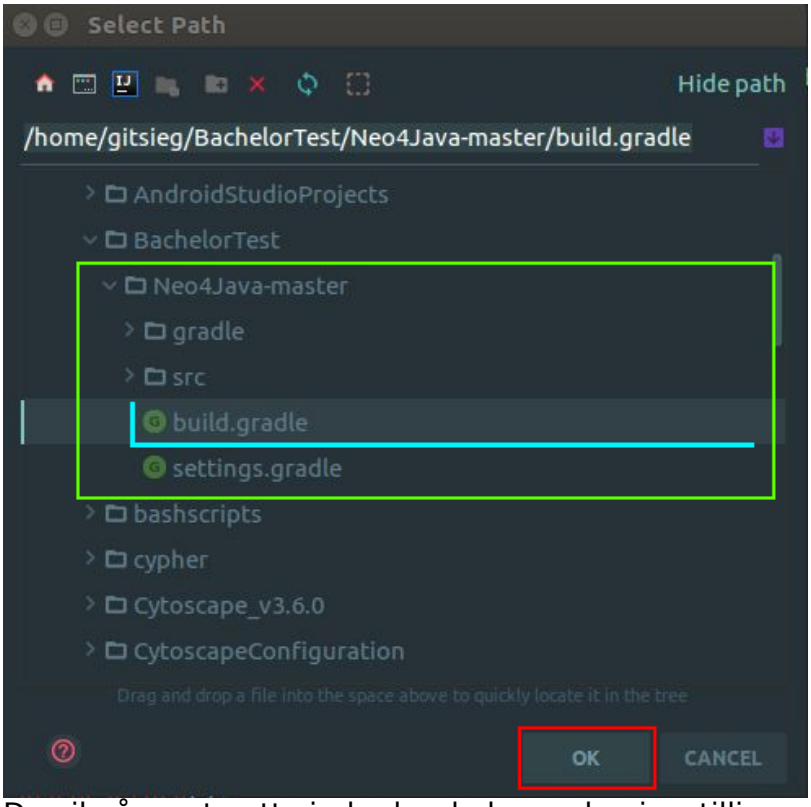
Gradle har ikke registrert at dette er et Gradle-prosjekt. Dette må du registrere manuelt i IntelliJ, slik at avhengigheter til eksterne biblioteker blir lastet ned og integrert i ditt prosjekt. For å gjøre dette, åpne “**View > Tool Windows > Gradle**” i navigasjonsmenyen til IntelliJ. Eksterne biblioteker som blir lastet ned er:

- org.json - bibliotek for tolkning av JSON-data
- org.neo4j - bibliotek for interaksjon med Neo4j-databaser.

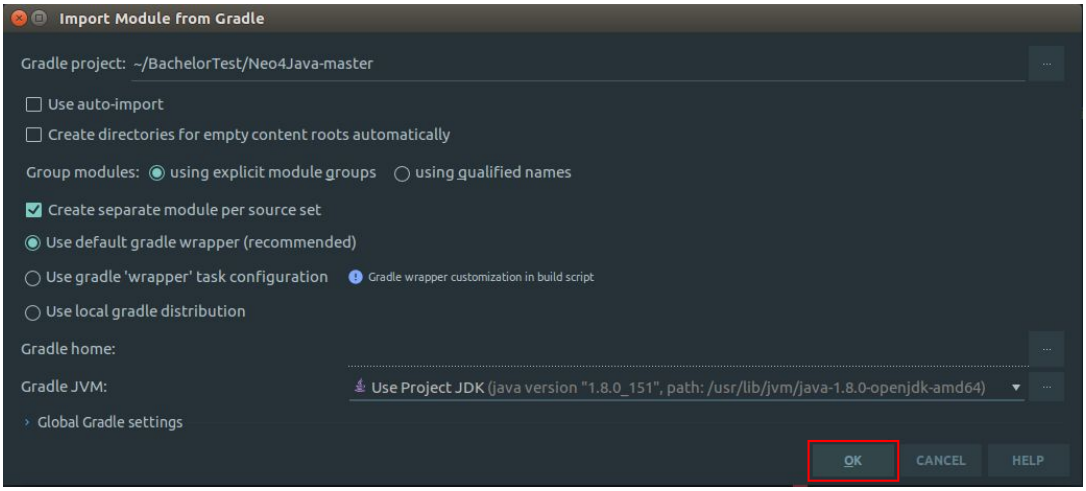


På høyre side i IntelliJ, vil du nå se en meny tilsvarende figuren ovenfor. Her kan du registrere en gradle-fil. Trykk på **+** tegnet, og du får opp en filvelger. Naviger til der du pakket ut prosjektet, og du finner gradle-filen (build.gradle) du er ute etter. Vi har merket den i blått. Velg denne og klikk **OK**.





Du vil nå se et nytt vindu der du kan velge innstillinger for importeringen. Klikk **OK** her også, uten å sette noen nye valg.



Gradle tar seg nå av nedlasting av de eksterne bibliotekene som applikasjonen trenger, henholdsvis et bibliotek for JSON-parsing og Neo4j sitt Java-API.

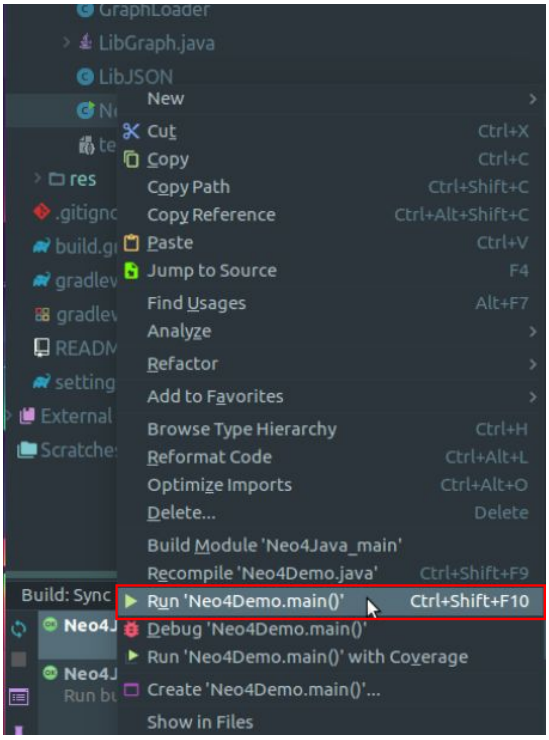
5.4.3 Oppdatere filsti til databasen

Filstiene i applikasjonen peker nå på prosjektdeltakernes databaser. Lokasjonen for databasen er tilsynelatende identisk på tvers av Linux-distribusjoner. For Windows-maskiner, er det sannsynligvis mer varierende.

Åpne klassen GraphLoader. I denne klassen er det deklarért to variabler, henholdsvis **WIN_DATABASE_PATH** og **LINUX_DATABASE_PATH**. Dersom du kjører Windows, erstatter du den filstien som ligger der, med din filsti til databasen. Gjør det samme dersom du kjører Linux.

5.4.4 Kjøre applikasjonen

Prosjektet tilbyr en klasse, **Neo4Demo**, som har en **main**-metode. Høyreklikk denne klassen og velg "**Run 'Neo4Demo.main()'**". Du kjører nå applikasjonen, og dersom rettighetene og filstier er satt korrekt, vil du applikasjonen ikke kaste noen unntak.



Det tar litt tid å prosessere alle datasettene, omtrent 30 sekunder på en av prosjektdeltakernes maskiner. Du vil bli informert i konsollen når et datasett er ferdig prosessert, og hvor mange koordinater som er opprettet.

```
/usr/lib/jvm/java-8-openjdk/bin/java ...
May 03, 2018 6:13:17 PM GraphLoader lambda$registerDatasets$1
INFO: Fil eksisterer:true
May 03, 2018 6:13:17 PM GraphLoader lambda$registerDatasets$1
INFO: Filsti:/home/kris/Student/Bachelor/Neo4Java/./src/res/json/fylker/NO-03.geojson
Antall koordinater: 5796
Antall forhold: 7645
```

Du kan nå starte opp Neo4j serveren og verifisere at dataen er innsatt.

5.5 Erfaringer: Java Embedded Database (Geodata)

5.5.1 Sammenligning av koordinat-noder

Det skjer en feil ved sjekking av koordinat-noder og oppretting av relasjoner som følger. Det ser ut til å være en feil i program-logikken men etter mye feilsøking har vi ikke klart å finne den.

Resultatet av feilen, hva enn den er, er at noen kommune-relasjoner (“NESTE_PUNKT_KOMMUNE”) blir doblet. Duplikatene peker aldri samme vei.

5.5.2 Minnebruk

Innsetting av store datasett som dette medfører betydelig behov for minne. Når vi tok utgangspunkt i bare én kommune og deretter ett fylke var ikke dette problem. Når vi gikk opp på landsnivå derimot eskalerte behovet for minne så betraktelig programmet krasjet.

Oppstykkingen av transaksjoner ved bruk av Command Pattern gjennomgått tidligere i denne seksjonen ble løsningen. Hver gang et fylke er gjennomgått, fullføres transaksjonen og minne blir frigjort. Omskrivingen til bruk av Command Pattern viste seg altså å ha flere fordeler.

5.5.3 Prosesseringstid

Tiden det tar å prosessere hele datasettet i Java er betraktelig redusert sammenlignet med bruk av Neo4j og CQL. Java-applikasjonen bruker ca 30 sekunder for Norge sine fylker og kommuner. Neo4j og CQL ca 1 time for kommuner i et fylke.

6 Vedlikeholdbarhet - Neo4j

6.1 Database-migrasjon

Neo4j har ingen innebygde verktøy for håndtering av migrasjoner, men det finnes et tredjepartsverktøy for dette

6.1.1 Liquigraph

Liquigraph baserer seg på java og Neo4j sin jdbc-driver, og migrasjoner blir definert i XML-dokumenter

Mer detaljert informasjon om bruk av Liquigraph finnes her.

6.2 Oppgradering

Det finnes utfyllende dokumentasjon om oppgradering av Neo4j her.
I høyre hjørne kan det velges manual for spesifikke versjoner, eventuelt er det mulig å hente ut manualen for spesifikke versjoner ved å endre URL'en direkte:
https://neo4j.com/docs/operations-manual/**current**/upgrade/
–> https://neo4j.com/docs/operations-manual/**3.1**/upgrade/

Det finnes også oppgraderingsguider her, der informasjon som blant annet hvorvidt en ny versjon har breaking changes befinner seg.

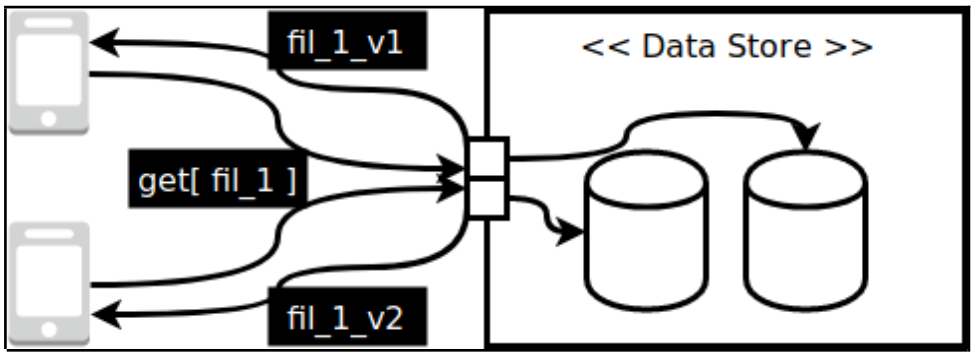
7 SQL vs NoSQL databaser

7.1 Grunnleggende

Relasjonsdatabaser tar utgangspunkt i ACID-prinsippene, for å garantere databasens innhold og struktur til enhver tid er korrekt.
Dette har betydelig kostnad når systemer skal skaleres opp da ACID-prinsippene krever at alle database-instanser må garantere samme innhold og interne struktur.
NoSQL-databaser tar derimot i bruk BASE-prinsippene, som ikke er en direkte [kontrast] til ACID, men heller en løsere implementasjon av de.

Base Availability går ut på at informasjon skal være tilgjengelig, men ikke nødvendigvis "korrekt".

Soft-state går ut på at datastores ikke trenger å være like hverandre til enhver tid. Ved skriving er det ikke garantert at alle instanser blir oppdatert samtidig og speiler hverandre, og dermed ved lesing er det ikke garantert at alle stores leverer samme data.
Det er heller opp til utvikler hvorvidt korrekthet av data er nødvendig ved lesing.



Istedenfor garanterer NoSQL databaser **Eventual Consistency.** Altså at datastores oppnår konsistens i data over tid; endringer propageres etter behov.

Grunnlaget for bruken av BASE-prinsippene ligger i hvilke oppgaver NoSQL databaser er ment å håndtere: distribuerte systemer i stor skala. For disse systemene ender relasjonsdatabaser opp med å være svært tungdrevne.

Sharding, eller horisontal skalering, altså å dele datalagringen på flere instanser, setter betydelige kommunikasjonskrav på et ACID-basert system. Alle instanser må

garantere korrekthet av data og struktur. All skriving til slike systemer medfører altså kommunikasjon om skrivingen til samtlige speilede instanser.

7.2 Graf-databaser

Graf-databaser baserer seg generelt på bruk av *index-free adjacency*, altså at noder ikke letes opp ved bruk av indekser men gjennom en serie av pekere; noder peker på hverandre. Disse pekerne kalles relasjoner.

Ved å basere seg på relasjons-traversering istedenfor indeksering er det mulig å gjennomløpe lange og komplekse relasjoner svært raskt. I tillegg er lagring av informasjon om noder og relasjoner svært dynamisk.

Graf-databaser baserer seg på følgende modeller: propertygraph-modell, hypergraph-modell og triples.

7.2.1 Neo4j

Neo4j er en ACID-kompatibel grafdatabase og benytter seg av Propertygraph-modellen.

Propertygraphs har følgende egenskaper:
Den er bygget opp av *noder* og *relasjoner* mellom disse, både nodene og relasjonene er alle bundet til hver sine *egenskaper* (properties). Disse egenskapene lagres i form av dokumenter med key-value par.

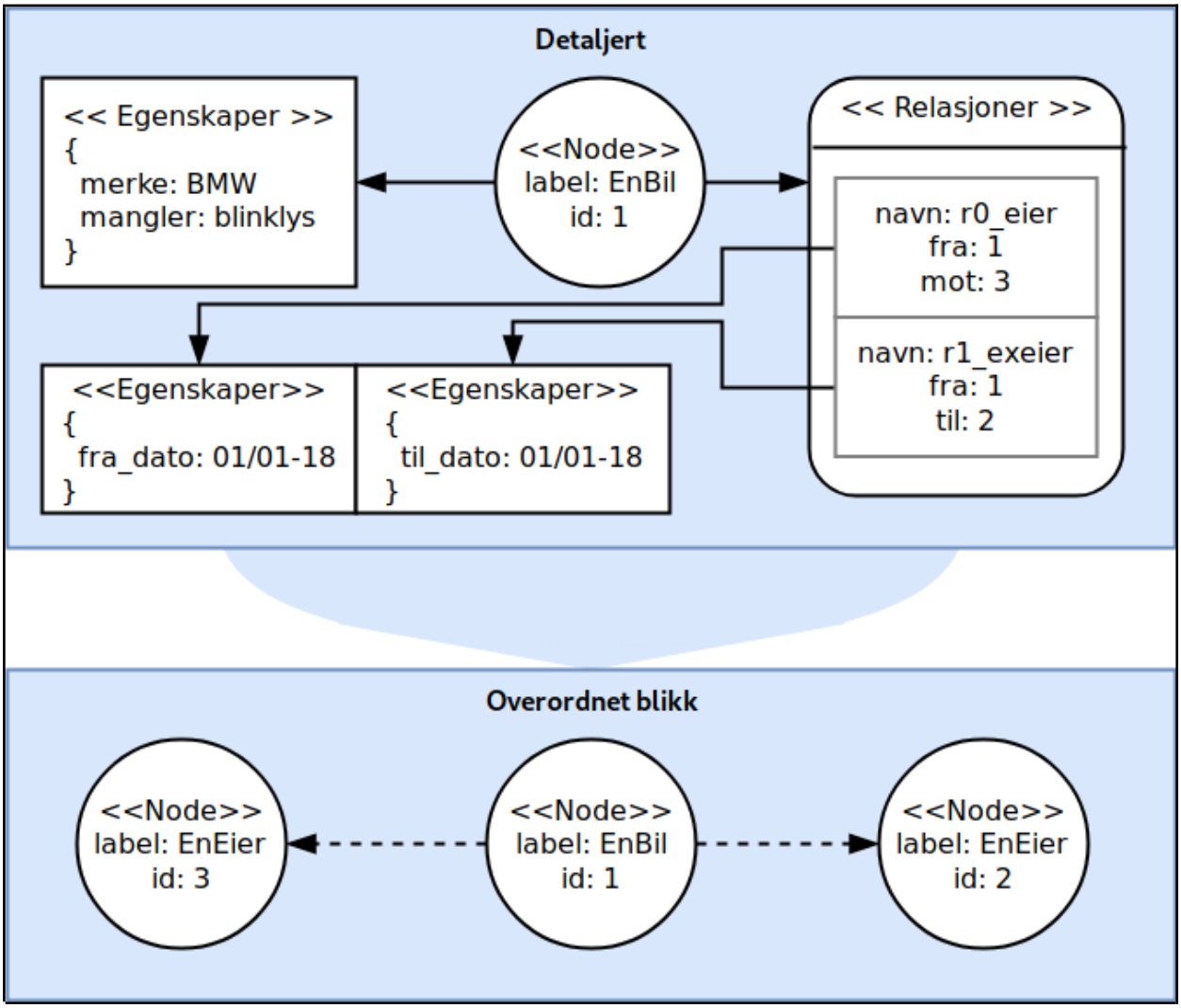
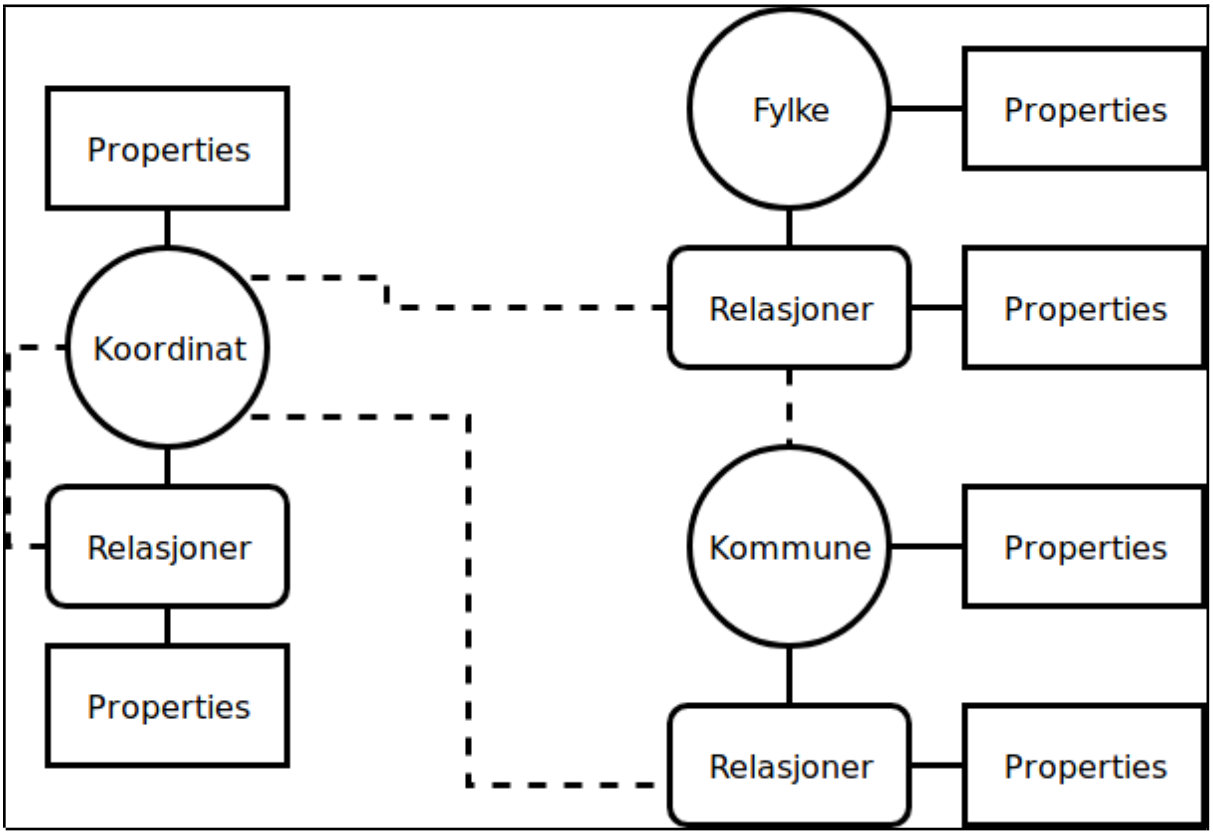


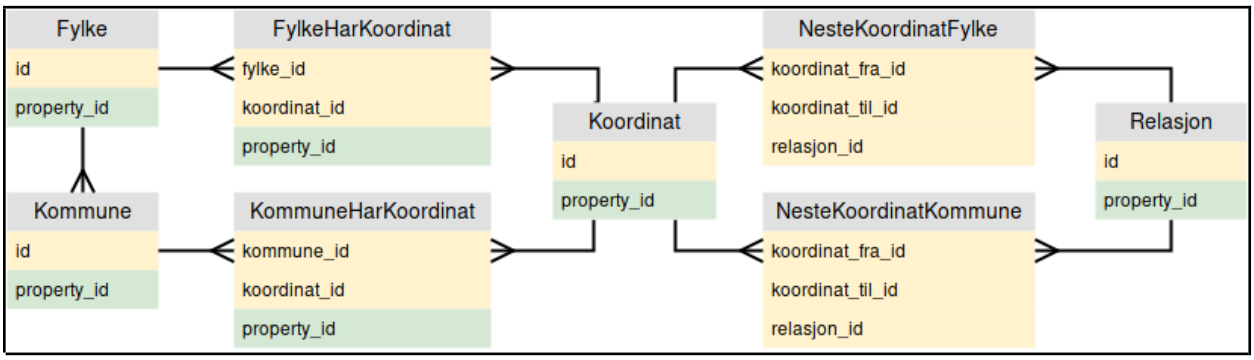
Diagram - *Sort pil representerer "en del av".*

7.2.2 Hvordan grafdatabasen ser ut for våre datasett



Det er tre noder, hhv Fylke, Kommune, og Koordinat. Alle noder har egenskaper (properties; ren data), og relasjoner til andre noder.
En relasjon har informasjon om start og slutt-node, samt sine egne egenskaper.

7.2.3 Hvordan en relasjonsdatabase kunne sett ut (simplifisert)



Som du ser av ER-diagrammet er antallet relasjoner langt høyere i relasjonsdatabasen sammenlignet med graf-modellen. Man får også betydelig overhead fra behovet for indeksering på hver tabell også.

I diagrammet er ikke tabellene for hver enkelt “nodes” egenskaper modellert modellert, men de grønne feltene representerer fremmednøkler som peker mot disse. Hver enkelt “node” X vil måtte ha hver sin unike [X]Property tabell, da innholdet i disse vil variere fra node til node.
Her støter relasjonsdatabaser på et annet problem: lite dynamiskt innhold. For å lagre nye egenskaper vil hele tabellens struktur måtte endres først. Dette er riktignok løst i nyere versjoner av MySQL/MariaDB, som nå har fått JSON-datatypen, eller ved bruk av TEXT/BLOB, osv.

8 Ressurser og vedlegg

8.1 Innføring i grafdatabasen Neo4j

Et gratis eksemplar av en bok som omhandler grafdatabaser og Neo4j:
https://neo4j.com/graph-databases-book/

8.2 Datasett

Nøyaktig lenke til datasettet varierer over tid, klikk på lenke og søk på Administrative Enheter: https://kartkatalog.geonorge.no/metadata/kartverket/

8.3 Property-Graph modellen

Kort om grafdatabaser og Property-Graph modellen:
https://neo4j.com/developer/graph-database/

8.4 Neo4j-manual

Neo4j sin operasjonsmanual for 3.3.x utgivelsene:

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

Manualen dekker flere temaer man støter på. Den finnes for hver release. Benytter man en annen versjon enn 3.3.x, erstatter man versjonsnummeret i URL'en til det aktuelle versjonsnummeret.
Nyest versjon finnes her: https://neo4j.com/docs/developer-manual/current/

8.5 Neo4j og Java

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

JavaDoc av Neo4j Java API:
https://neo4j.com/docs/java-reference/3.3/javadocs/

8.6 Neo4j Migrasjonsverktøy

http://www.liquigraph.org/

8.7 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/

Kompatabilitetsmatrise:
https://github.com/neo4j-contrib/neo4j-apoc-procedures#version-compatibility-matrix

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

8.8 Neo4j-contrib

Samling av open-source bidrag til Neo4j:
https://github.com/neo4j-contrib

8.9 Repositories for Java-Applikasjon

Demoprosjekt for embedding av Neo4j databaser:
https://github.com/gitsieg/Neo4jEmbeddedDemo

Java-applikasjon som tar i bruk Neo4j og geodata:
https://github.com/gitsieg/Neo4Java

8.10 Andre referanser

Søk på koordinater. Brukt for å validere innsettinger:
http://finnposisjon.test.geonorge.no/

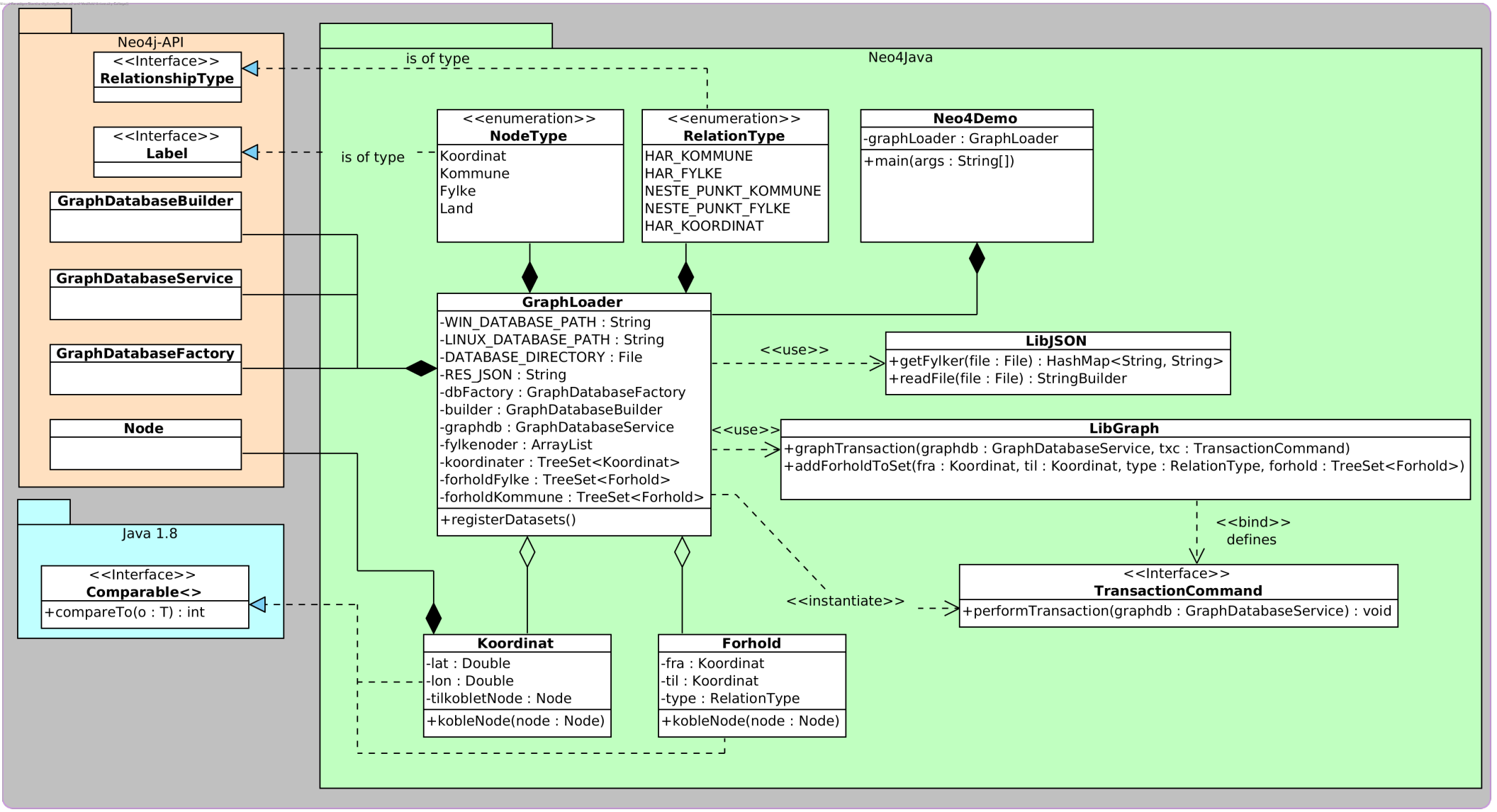
8.11 Kildekode: Neo4jEmbeddedDemo

https://raw.githubusercontent.com/gitsieg/Neo4jEmbeddedDemo/master/src/main/java/Main.java

```
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.factory.GraphDatabaseBuilder;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

import java.io.File;

public class Main {
    // Filen til databasen man ønsker å benytte
    private static final File DATABASE_DIRECTORY = new
File("/var/lib/neo4j/data/databases/nydb.db");

    // 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
    }

    public static void main(String[] args) {
        // Opprett databasen. Dersom den ikke eksisterer, opprettes
en ny
        GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
        GraphDatabaseBuilder dbBuilder =
```

```
dbFactory.newEmbeddedDatabaseBuilder(DATABASE_DIRECTORY);
        GraphDatabaseService grapdb = dbBuilder.newGraphDatabase();

        try (Transaction tx = grapdb.beginTx()) {
            grapdb.execute("match (n) detach delete n;");

            // 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 en bil-node og angi egenskaper
            Node honda = grapdb.createNode(NodeType.Car);
            honda.setProperty("regnr", "NM12345");
            honda.setProperty("model", "CR-V");
            honda.setProperty("førstegangsreg", "20180609");

            // 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 Person-node og angi egenskaper
            Node kari = grapdb.createNode(NodeType.Person);
            kari.setProperty("fornavn", "Kari");
            kari.setProperty("etternavn", "Nordmann");
            kari.setProperty("alder", 43);

            // Opprett Person-node og angi egenskaper
            Node per = grapdb.createNode(NodeType.Person);
            per.setProperty("fornavn", "Per");
            per.setProperty("etternavn", "Nordmann");
            per.setProperty("alder", 19);

            // Opprett Person-node og angi egenskaper
            Node stine = grapdb.createNode(NodeType.Person);
            stine.setProperty("fornavn", "Stine");
            stine.setProperty("etternavn", "Nordmann");
            stine.setProperty("alder", 17);
```

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

            // Opprett forhold mellom personer
            kari.createRelationshipTo(ola, RelationType.Knows);
            ola.createRelationshipTo(kari, RelationType.Knows);
            stine.createRelationshipTo(kari, RelationType.HasParent);
            stine.createRelationshipTo(ola, RelationType.HasParent);
            per.createRelationshipTo(kari, RelationType.HasParent);
            per.createRelationshipTo(ola, RelationType.HasParent);
            ola.createRelationshipTo(stine, RelationType.ParentOf);
            ola.createRelationshipTo(per, RelationType.ParentOf);
            kari.createRelationshipTo(stine, RelationType.ParentOf);
            kari.createRelationshipTo(per, RelationType.ParentOf);
            tx.success();
            tx.close();
        }
        // Lukk instansen av grafdatabasen
        grapdb.shutdown();
    }
}
```


Neo4j-API
<<Interface>>
RelationshipType
<<Interface>>
Label
GraphDatabaseBuilder
GraphDatabaseService
GraphDatabaseFactory
Node
Java 1.8
<<Interface>>
Comparable<>
+compareTo(o : T) : int
Neo4Java
is of type
is of type
<<enumeration>>
NodeType
Koordinat
Kommune
Fylke
Land
<<enumeration>>
RelationType
HAR_KOMMUNE
HAR_FYLKE
NESTE_PUNKT_KOMMUNE
NESTE_PUNKT_FYLKE
HAR_KOORDINAT
Neo4Demo
-graphLoader : GraphLoader
+main(args : String[])
GraphLoader
-WIN_DATABASE_PATH : String
-LINUX_DATABASE_PATH : String
-DATABASE_DIRECTORY : File
-RES_JSON : String
-dbFactory : GraphDatabaseFactory
-builder : GraphDatabaseBuilder
-graphdb : GraphDatabaseService
-fylkenoder : ArrayList
-koordinater : TreeSet<Koordinat>
-forholdFylke : TreeSet<Forhold>
-forholdKommune : TreeSet<Forhold>
+registerDatasets()
<<use>>
LibJSON
+getFylker(file : File) : HashMap<String, String>
+readFile(file : File) : StringBuilder
<<use>>
LibGraph
+graphTransaction(graphdb : GraphDatabaseService, txc : TransactionCommand)
+addForholdToSet(fra : Koordinat, til : Koordinat, type : RelationType, forhold : TreeSet<Forhold>)
<<bind>>
defines
<<instantiate>>
<<Interface>>
TransactionCommand
+performTransaction(graphdb : GraphDatabaseService) : void
Koordinat
-lat : Double
-lon : Double
-tilkobletNode : Node
+kobleNode(node : Node)
Forhold
-fra : Koordinat
-til : Koordinat
-type : RelationType
+kobleNode(node : Node)