

Dette dokumentet viser bruk av teknologiene

- Neo4j
- Mongoddb
- Cassandra
- Redis.
- Spark

Det lages en strømanalyse mot slutten av dokumentet, som kombinerer flere av teknologiene.

Etter dette er det i tillegg deler som forteller mer skriftlig om databasene

De brukes mot noen utvalgte datasett som inneholder data relatert til kraftmarkedet.

Dette arbeidet har vært utført av:

- Sindre Halsebakk
- Susanne Thorvaldsen
- Vemund V. Brynjulfsen

Datasett

Datasettene vi bruker inneholder nåtids- og historiske data for strømpriser, nettleie, værdata og magasindata.

For å gjøre dette skal vi se på fire ulike datasett: magasinstatistikk, strømpriser, nettleie og værstatistikk.

For å gjøre dette skal vi se på fire ulike datasett: magasinstatistikk, strømpriser, nettleie og værstatistikk.

På denne måten vil vi kunne se hvor stor innvirkning vær da spesielt nedbør, og årstider har på strømpriser i ulike områder. Vi skal hente ut historiske data om hvordan strømprisene endrer seg gjennom perioder.

Datasett: værstatistikk, spotprisstatistikk, magasinstatistikk, nettleie
Hvordan datasettene kan passe sammen

Datasettene kan bli har følgende kolonner de kan kobles over:

Magasinstatistikk: omnrnr, dato

Magasinstatistikk min/max/median: omnrnr, iso_uke

nettleiefylkesnitt: fylkenavn, fylkenr, tariffdato

nettleiehistorikk: fylkenr, fylkenavn, periodefradato, periodetildato

værstatistikk: navn, stnr, dato, fylke

priser_areas: prisområde

Fra omnr kan man få hvilket fylke en rad burde høre til.

Alle statestikkene kan kobles over:

Hvilket fylke en rad tilhører

Hvilken dato en rad tilhører

Værstatistikk

Kilde: Observasjoner og værstatistikk - Seklima (met.no)

Datasettet er en statistikk for forskjellige værmålinger for hvert døgn over en periode fra 29.08.2017 til 29.08.2023.

I tillegg har gruppen lagt på en kolonne for fylket som stasjonen er innenfor.

Datasettet inneholder følgende kolonner:

Fylke: Hvilket fylke stasjon er innenfor. (eks: Agder)

Navn: Navn på stedet (eks: Arendal Lufthavn)

Stasjon: Unik identifikator på stasjon (eks: SN36330)

Tid: Dato målingen ble gjort (eks: 29.08.2017)

Middel av middelvind fra hovedobs: Gjennomsnittlig vind for dagen i m/s (eks: 2)

Middeltemperatur: Gjennomsnittlig temperatur for gjennom dagen (eks: 15)

Makstemperatur: Største verdi for temperatur tatt over dagen (eks: 17)

Minimumstemperatur: Minste verdi for temperaturen tatt igjennom dagen (eks: 7)

Nedbør: Nedbør for dagen i mm (eks: 9)

Magasinstatistikk

Datasettet her gir informasjon om energilager og fyllingsnivåer over tid, for spesifikke datoer, uker og områder.

Områdeinformasjon knyttet til datasettet:

Omrnr: 0 = Hele landet Navn: Norge

OmrType: «NO»

Elspot:

Omrnr: 1 = Øst-Norge. Omfatter østlige del av Østfold fra Viken og nordover (bortsett fra

en del av Innlandet som ligger vest og nord for Vågåmo). Består av NVE-område 1.

Navn: NO 1 (Elspotområde 1)

OmrType: EL

Omrnr: 2 = Sørvest-Norge. Omfatter sørlige delen av Viken, mesteparten av Vestfold og Telemark, Agder, Rogaland og sørlige del av Vestland. Består av NVE-områdene 3, 4, og 5.

Navn: NO 2 (Elspotområde 2)

OmrType: EL

Omrnr: 3 = Midt-Norge. Omfatter nordre og vestlige del av Vestland, den del av Innlandet som ligger vest og nord for Vågåmo, Møre og Romsdal og Trøndelag til Tunnsjødal. Består av NVE-områdene 8, 9 og 13.

Navn: NO 3 (Elsporområde 3)

OmrType: EL

Omrnr: 4 = Nord-Norge. Omfatter resten av Trøndelag og Nord-Norge. Består av NVE-områdene 10, 11 og 12.

Navn: NO 4 (Elspotområde 4)

OmrType: EL

Omrnr: 5 = Vest-Norge. Omfatter midtre del av Vestland opp til Sognefjorden og Indre Sogn, og vestlig del av Viken og Innlandet. Består av NVE-områdene 2, 6 og 7.

Navn: NO 5 (Elspotområde 5)

OmrType: EL

Vassdrag:

Omrnr: 1 = Sørøst Norge. Østlandet, Agder-fylkene og deler av Rogaland. Navn:

VASS1 (Vassdragsområde 1)

Omrtype: Vass

Omrnr: 2 = Vest-landet. Resten av Rogaland, mesteparten av Hordaland og Sogn og Fjordane. Navn: VASS2 (Vassdragsområde 2)

OmrType: Vass

Omrnr: 3 = Midt-Norge. Møre og Romsdal, Trøndelag og sørlige Nordland. Navn:

VASS3 (Vassdragsområde 3)

OmrType: Vass

Omrnr: 4 = Nord-Norge. Resten av Nordland og nordover. Navn: VASS4 (Vassdragsområde 4)

OmrType: Vass

Nettleie:

Nettleie_fylkessnitt:

Dette datasettet viser statistiske verdier og gjennomsnittlige målinger relatert til

nettleiekostnader og energiforbruk for husholdninger i ulike fylker i Norge.

Nettleiehistorikk:

Denne informasjonen gir detaljert innsikt i nettleiepriser og -strukturer, inkludert fastledd, energiledd og effektledd for ulike tariffer og grupper. Det gir også informasjon om forbruksmengder, perioder og organisasjonsrelaterte detaljer.

Datasettene er hentet:

Swagger UI (nve.no)

<https://biapi.nve.no/magasinstatistikk/swagger/index.html>

<https://biapi.nve.no/nettleiestatistikk/swagger/index.html>

Spotprisstatistikk

Datasettet inneholder priser for de forskjellige prisområdene timesvis. Datasettet går tilbake til 01.01.2020.

MTU (CET/CEST): Timen det gjelder. (eks 01.01.2020 00:00 - 01.01.2020 01:00)

Day-ahead Price [EUR/MWh]: Spotprisen for timen. (eks 119.32)

Day-ahead price tilsvarende priser som blir gitt for strøm timesvis. Det er denne prisen som man henviser til når man snakker om hva strømprisen er.

Currency: Valutaen prisen er gitt i.

Area: prisområdet (eks BZN | NO1)

BZN (Bidding zone) er forskjellige markedsområder for salg og kjøp av strøm innenfor det europeiske strømmarkedet. F.eks BZN | RO gjelder for Romania, BZN | PL gjelder for Polen. Norge er delt inn i 5 områder (NO1, NO2, NO3, NO4, NO5).

Datasettet er nyttig for å se historiske og fremtidige strømpriser. For analyse kan man f.eks. sammenligne strømpriser med værforhold, nettleie og fyllingsgrad for å se hvordan disse henger sammen.

Kilde: entsoe.eu

Dataene kommer fra entsoe, som publiserer data og rapporter omhandlende det europeiske kraftmarkedet. Det er f.eks. data fra entsoe som Nord Pool bruker når det henviser til strømpriser.

Grafdatabase (Neo4j)

Design av Graph-database

Datasett

Vi har valgt ut to datasett fra forrige milepæl som vi skal basere denne databasen på. Datasettene beskriver henholdsvis magasinifylling og spotpris for et område i Norge. Vi ser to koblinger mellom dem basert på disse kolonnene:

| Datasett 1: Magasinstatistikk.csv | Datasett 2: Prices_Area.csv |
|--|--|
| Dato_Id | Date |
| OmrNr | Area |

Designprosessen

Vi bestemte oss for å bruke datasettene for magasinifylling og spotpris. Først lagde vi en oversikt over kolonnene i magasinifyllingsdatasettet og spotprisdasettet.

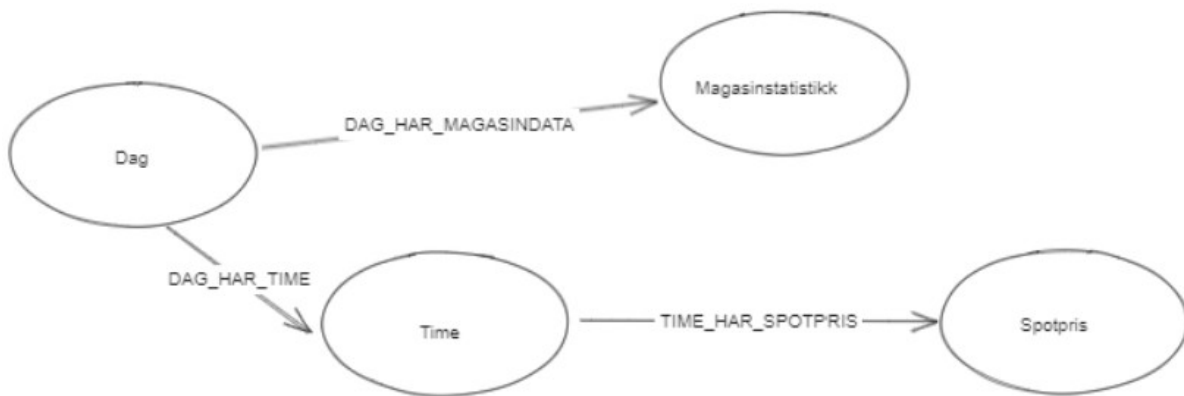
| | |
|---|---|
| Magasinstatistikk 1. dato_Id: Dato 2. omrnr: Type område EL, VASS eller NO 3. omrnr: Områdenummer 4. iso_aar: ISO-år, året i ISO 8601-format (f.eks. 2008, 2015) som er referert til. 5. iso_uke: ISO-uke, ukenummer i ISO 8601-format (f.eks. 51, 52) som er referert til. 6. fyllingsgrad: Fyllingsgraden til et energilager i desimalform. Dette representerer hvor mye av lagringskapasiteten som er fylt opp med energi. 7. kapasitet_TWh: Kapasitet i terawattimer (TWh) for det aktuelle energilageret. 8. fylling_TWh: Mengden energi i terawattimer (TWh) som er lagret i energilageret på tidspunktet. 9. neste_Publiseringdato: Datoen da neste oppdatering eller publisering av dataen er planlagt. 10. fyllingsgrad_forrige_uke: Fyllingsgraden for samme energilager i forrige uke. 11. endring_fyllingsgrad: Endringen i fyllingsgraden fra forrige uke til nåværende tidspunkt, uttrykt som desimalverdi. | Spotpris MTU (CET/CEST): Timen det gjelder. (eks 01.01.2020 00:00 - 01.01.2020 01:00) Day-ahead Price [EUR/MWh]: Spotprisen for timen. (eks 119.32) Area: prisområdet (eks BZN N01) |
|---|---|

Så tenkte vi for oss hvordan de to datasettene henger sammen. De henger sammen ved at de begge har en kolonne som refererer til elspotområde: omr_nr for magasinstatistikk, Area for spotpris. Hvis vi lager en node for hvert datasett, ser databasen slik ut:



Problemet med denne databasen er at magasinstatistikk og spotpris vil inneholde utrolig mye data. Hvis man skal ha en spotpris for en tid i et elspotområde, så må man se gjennom veldig mye data før man finner fram.

Det blir lettere hvis man kan splitte opp noden for magasinstatistikk dagvis, og splitte opp spotpris timesvis:

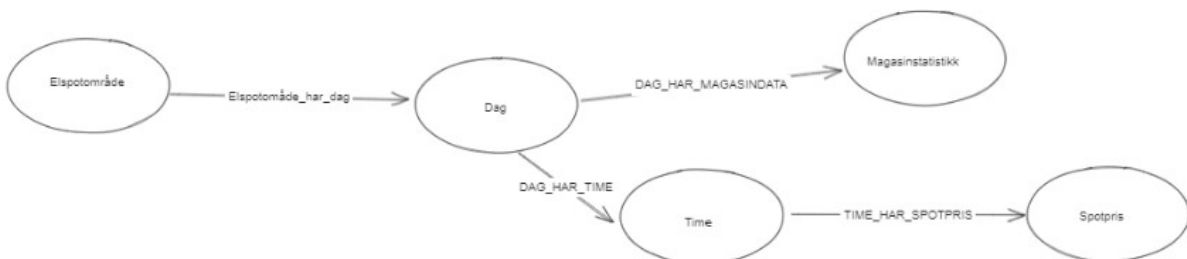


Men databasen har fortsatt problemer.

Spotpris og magasinstatistikk vil inneholde data for alle elspotområdene for hver dag/time.

Man er gjerne kun interessert i data for elspotområdet man selv bor i, så det blir unødvendig så mye data å se igjennom.

Istedenfor kan vi skille dagene etter elspotområde:

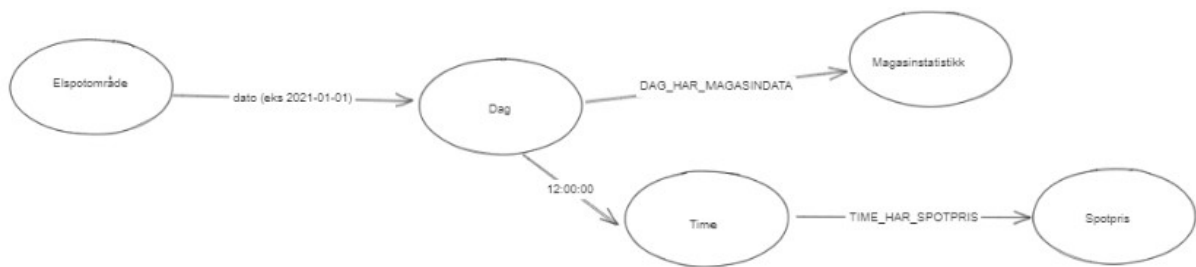


Nå begynner databasen å se mer ferdig ut.

Men man kan fortsatt gjøre noen justeringer.

Hvis vi har relasjonen fra elspotområde til dag være en eksakt dato, og relasjonen for time til spotpris være en eksakt time, så blir databasen bedre.

Da kan man hoppe mellom nodene etter relasjonstypen istedenfor å skjekke en dato/time verdi for alle de $360 \times (\text{mengde år})$ eller 24 (timer) relasjonene.



I tillegg blir spørringene enklere, fordi det er enklere å skrive simpelt navnet på en node eller en kant, enn å måtte referere til en verdi for en variabel.

F.eks. så er dette enklere å skrive:

MATCH

```
(:Uke) - [:`2021-03-04`] -> (:Dag)
```

enn dette:

MATCH

```
(:Uke{Dato: date("2021-03-04")}) - [] -> (:Dag)
```

I det første eksempelet er det mindre å skrive, og spørringen er mer oversiktlig.

Datainnlegging

Vi skal legge inn data i databasen.

Dataen vi skal legge inn er eksempeldata.

Naturen til datasettene våre er sånn at de kan ha veldig mange relasjoner.

Et elspotområde har noder for flere år, så for eksempel en dag har 24 timer, alle de timene har en spotpris. For hver dag skal det legges inn 24 timer, med spotprisene de har.

Siden vi skal legge inn data manuelt, må vi begrense oss.

Derfor inneholder hvert elspotområde kun 1 dag, og hver dag inneholder kun 7 timer.

For å opprette elspotområdene brukte vi denne spørringen:

CREATE

```
( N01:Område {Navn: 'N01', Område_nr:1, Område_type:'EL'} ),
( N02:Område {Navn: 'N02', Område_nr:2, Område_type:'EL'} ),
( N03:Område {Navn: 'N03', Område_nr:3, Område_type:'EL'} ),
( N04:Område {Navn: 'N04', Område_nr:4, Område_type:'EL'} ),
( N05:Område {Navn: 'N05', Område_nr:5, Område_type:'EL'} )
```

Spørringen lager 5 forskjellige noder for elspotområde, med 3 variabler.

For å lage datoer og relasjonen mellom elspotområde og dag, brukte vi denne spørringen:

```
MATCH (o:`Område`{Navn:'N01'})
CREATE
(o)-[:'2022-06-19']->(d:Dag {Dato = date("2022-06-19")})
```

Spørringen finner et elspotområde spesifisert fra verdien til variabelen "Navn", så lages det en kant fra elspotområdet til en ny node for en dato. Kanten har samme navn som datoen det gjelder. Noden for datoen har en variabel med verdi for datoen

For å lage timer og relasjonen mellom timer og dag, brukte vi denne:

```
MATCH
(d:Dag)
CREATE
(:Time {time: localtime('00:00')})<-[:'00:00']-(d),
(:Time {time: localtime('03:00')})<-[:'03:00']-(d),
(:Time {time: localtime('06:00')})<-[:'06:00']-(d),
(:Time {time: localtime('09:00')})<-[:'09:00']-(d),
(:Time {time: localtime('12:00')})<-[:'12:00']-(d),
(:Time {time: localtime('15:00')})<-[:'15:00']-(d),
(:Time {time: localtime('18:00')})<-[:'18:00']-(d),
(:Time {time: localtime('21:00')})<-[:'21:00']-(d)
```

Spørringen finner alle nodene for dager, så legges det på kanter til timesnoder for hver dagnode.

For å lage nodene for spotpris, og relasjonen mellom spotpris og timer, brukte vi denne spørringen:

```
MATCH
(o:`Område`{Navn:"N05"})-[:'2022-06-19']-(d)-[:'21:00']-(t)
CREATE
(t)-[:HAR_SPOTPRIS]->(:Spotpris{Valuta:"Euro",Day_ahead:150.04})
```

Spørringen starter på noden for elspotområde, så bruker den navnene på kantene mellom dagsområdet, dager og tider til å finne fram. Så lages det en kant som går fra timen til en ny node. Den nye noden kalles Spotpris, og har 2 variabler.

Så repeterte vi dette for alle de forskjellige timene til hver dato i databasen.

For å lage noden for magasinfylling og relasjonen mellom dag og magasinfylling.

MATCH

```
(o:Område{Navn: 'N01'}) - [: '2022-01-01'] -> (d)
```

CREATE

```
(d) - [:HAR_MAGASINSTATISTIKK] ->
```

```
(magasin:Magasinfylling
```

```
{Fyllingsgrad: 0.43423307, KapasitetTWh: 33.9288802, FyllingTWh:
```

```
14.7330065,
```

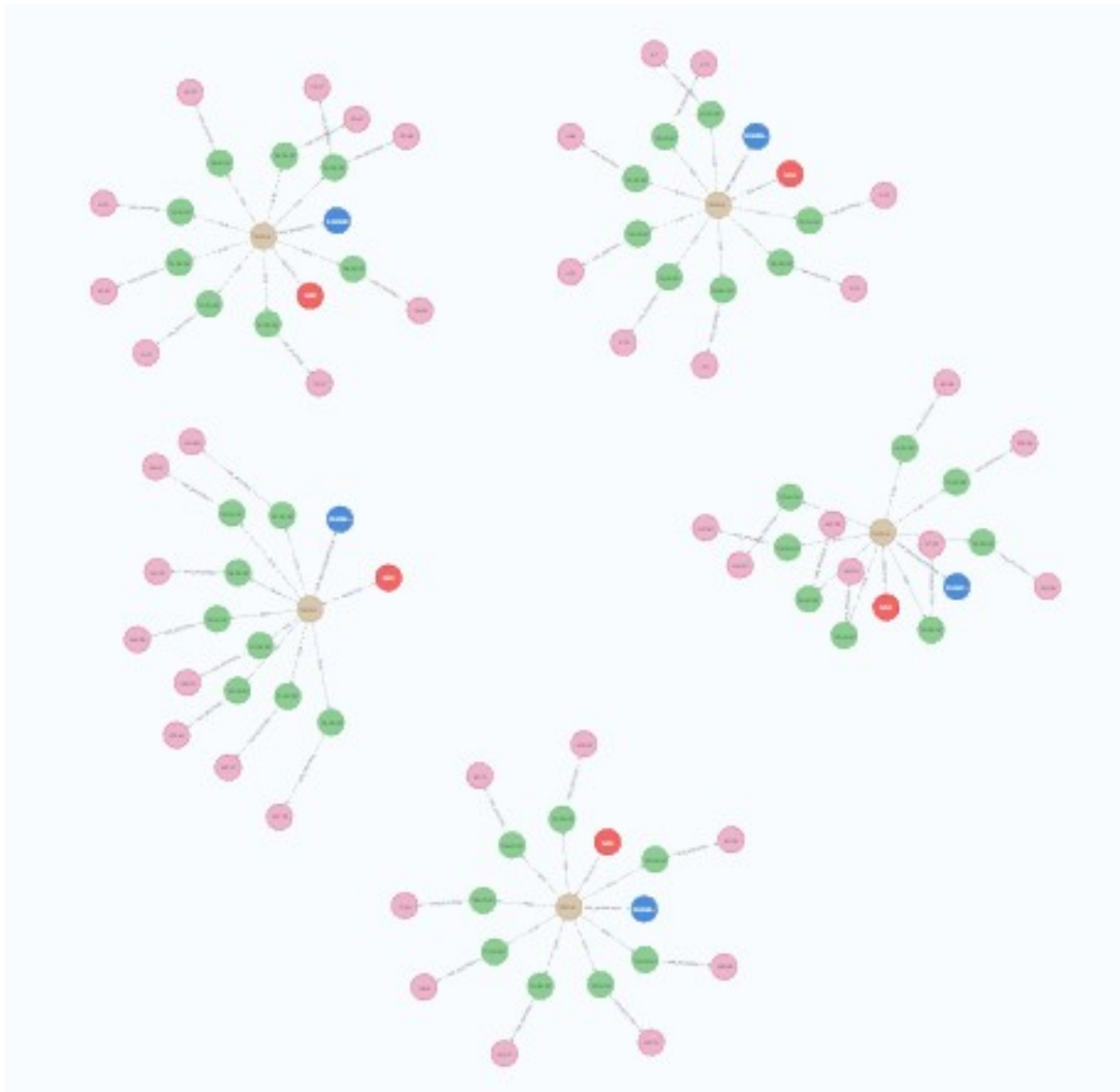
```
NestePubliseringsDato: date('2022-02-09') })
```

Spørringen starter fra en node for elspotområde, så hopper man til noden for en spesifikk dag med bruk av navnet på kanten mellom elspotområdet og den spesifikke dagen.

Deretter lages det en kant fra dagen til en node for magasinfylling. Magasinfyllingsnoden inneholder 4 variabler.

Så repeterte vi dette for alle de forskjellige dagene i databasen.

Den endelige databasen så slik ut:



Fargekoding for noder:

Rød node: Elspotområde

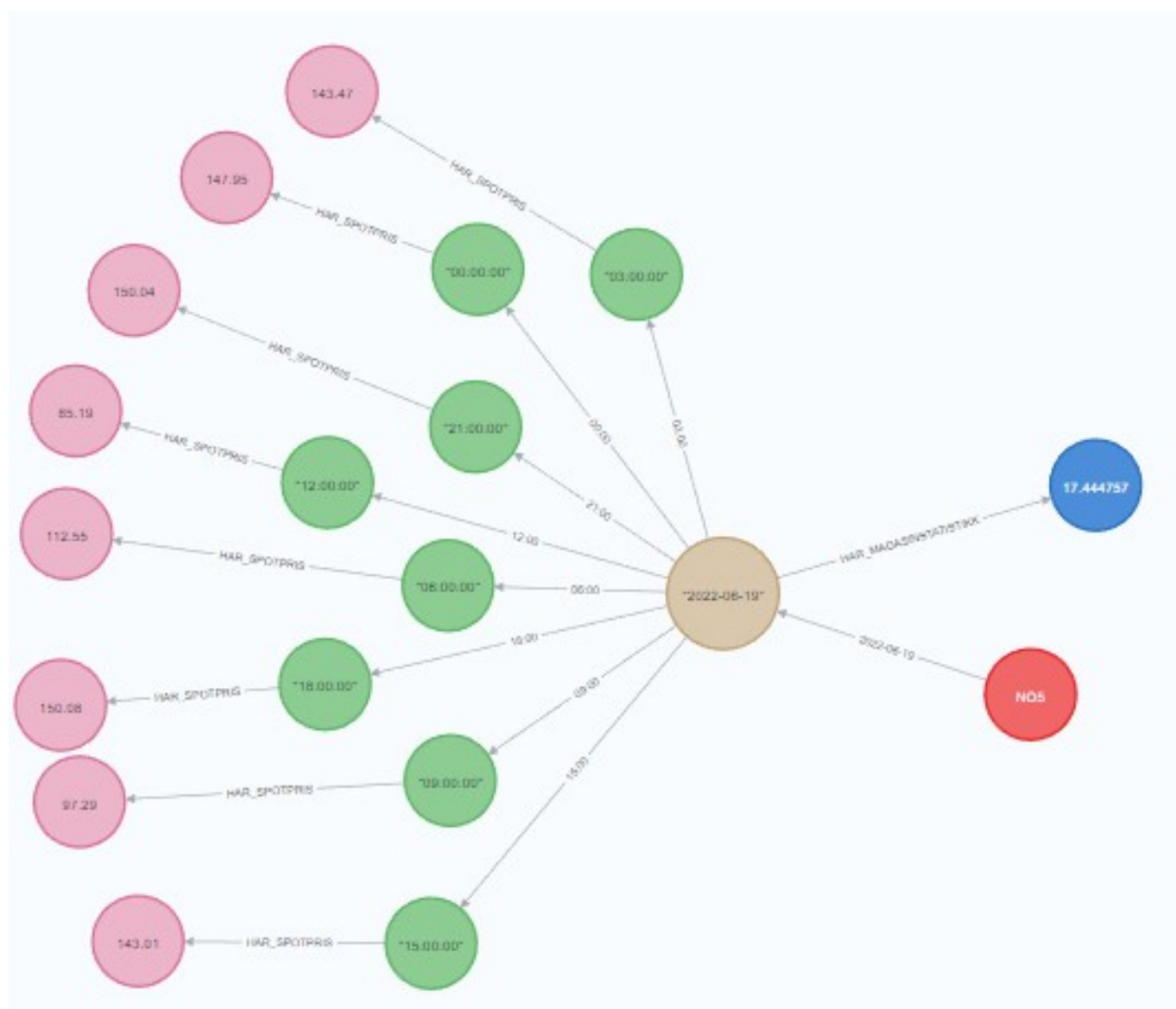
Brun node: Dag

Grønn node: Time

Rosa node: Spotpris

Blå node: Magasindata

Hver node for et elspotområde så slik ut:



Skriving av Cypher-spørringer

Vi har kommet frem til en liste over fem ulike interessante spørsmål vi ønsker å besvare. Basert på disse, ønsker vi å lage spørringer til databasen. Under følger en oversikt over spørringene vi vil bruke for å besvare disse spørsmålene sammen med forklaring og resultat.

Spørring: Spotpris per kvartal (Euro)

Her ønsker vi at spørringen skal returnere den gjennomsnittlige spotprisen for et kvartal i et elspotområde. Altså gjennomsnittspris for alle timene over en periode på 3 måneder.

Siden det bare er lagt inn 1 dag per elspotområde, så vil spørringen kun returnere gjennomsnittsprisen for den ene dagen.

MATCH

```
(o:Område{Navn: 'N01'})-[]->(d)-[]->(t:Time)-[]->(s)
```

WHERE

```
d.Dato >= date('2022-01-01') AND d.Dato <= date('2022-04-01')
```

RETURN

```
AVG(s.Day_ahead) AS `2022K1`
```

Spørringen starter i noden for elspotområde 1, så hentes alle dagene elspotnoden går mot, og deretter alle timene de forskjellige dagene går mot. Så hentes alle spotprisene som eies av de forskjellige timene.

Så filtreres dagene for kun de dagene som er innenfor intervallet på 3 måneder.

Til slutt tar man gjennomsnittet av alle spotprisene funnet.

Dette er resultatet:

| 2022K1 | |
|--------|--------------------|
| 1 | 108.22749999999999 |

Vi kan se at spørringen returnerer den forventede verdien.

| Spotprisdata | Omr nr | Dato | kl:00.00 | kl:03.00 | kl:06.00 | kl:09.00 | kl:12.00 | kl:15.00 | kl:18.00 | kl:21.00 |
|--------------|--------|------------|----------|----------|----------|----------|----------|----------|----------|----------|
| | 1 | 01.01.2022 | 119,98 | 92,36 | 77,65 | 93,73 | 120,02 | 120,71 | 121,77 | 119,6 |

Gj.snitt: 108,2275 ▾

For å teste ytterligere om spørringer fungerer, så legger jeg til 2 dager til Den ene dagen kommer til å være innenfor intervallet på 3 måneder, mens den andre datoen kommer til å være forbi intervallet.

Spørringen burde returnere gjennomsnittet av prisene for den allerede eksisterende dagen (som er innenfor intervallet), og den nye dagen som er innenfor intervallet. Dagen forbi burde ikke inkluderes i gjennomsnittet:

Jeg legger til dagene 02.04.2022 og 01.01.2022.
Her er koden for å legge til dagen 02.04.2022

MATCH

```
(o:Område{Navn:'N01'})
```

CREATE

```
(o)-[:`2022-04-02`]->(d:Dag{Dato:date('2022-04-02')}),  
(d)-[:`00:00`]->(:Time{time:localTime('00:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:184.42}),  
(d)-[:`03:00`]->(:Time{time:localTime('03:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:184.1}),  
(d)-[:`06:00`]->(:Time{time:localTime('06:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:184.3}),  
(d)-[:`09:00`]->(:Time{time:localTime('09:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:189.48}),  
(d)-[:`12:00`]->(:Time{time:localTime('12:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:180.96}),  
(d)-[:`15:00`]->(:Time{time:localTime('15:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:179.63}),  
(d)-[:`18:00`]->(:Time{time:localTime('18:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:185.08}),  
(d)-[:`21:00`]->(:Time{time:localTime('21:00')})-  
[:HAR_SPOTPRIS]->(:Spotpris{Day_ahead:194.78})
```

Fra noden for elspotområde lages en dag, og en relasjon til dagen som har likt navn som dagen. Så opprettes timene for dagen, relasjonene fra dagen til timene, spotprisen for timen, og relasjonen fra timen til spotprisen.

Når jeg kjører spørringen for å få kvartalsprisen, får jeg dette som resultat:

| 2022K1 | |
|--------|----------|
| 1 | 160.0625 |

Hvis jeg endrer spørringen til å returnere dagene, så får jeg kun de to dagene innenfor intervallet:

MATCH

```
(o:Område {Navn: 'N01'})-[]-> (d)-[]-> (t:Time)-[]-> (s)
```

WHERE

```
d.Dato >= date('2022-01-01') AND d.Dato <= date('2022-04-01')
```

RETURN d



Hvis jeg endrer spennet for intervallet slik at det inkluderer dagen som opprinnelig var forbi intervallet, så endres verdiene jeg får:

MATCH

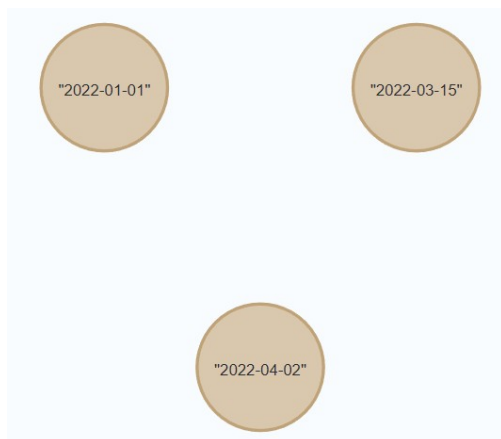
```
(o:Område {Navn: 'N01'})-[]-> (d)-[]-> (t:Time)-[]-> (s)
```

WHERE

```
d.Dato >= date('2022-01-01') AND d.Dato <= date('2022-04-02')
```

RETURN

d



MATCH

```
(o:Område {Navn: 'N01'})-[]-> (d)-[]-> (t:Time)-[]-> (s)
```

WHERE

```
d.Dato >= date('2022-01-01') AND d.Dato <= date('2022-04-02')
```

RETURN

```
AVG(s.Day_ahead) AS `2022K1`
```

2022K1

168.48958333333337

Designet av databasen gjør spørringen enklere på følgende måter:

MATCH

(o:Område{Navn: 'N01'}) - [] -> (d) - [] -> (t:Time) - [] -> (s)

WHERE

d.Dato >= date('2022-01-01') AND d.Dato <= date('2022-04-01')

RETURN

AVG(s.Day_ahead) AS `2022K1`

1. Den gir en intuitiv måte å hente ut dataene på. Man går fra dag til time, til spotpris.
2. Den gjør det lett å filtrere for hva man vil ha. Man kan lett spesifisere elspotområdet, og spesifisere dagene for intervallet.

Spørring: Spotprisen for en gitt dato og klokkeslett

Her ønsker vi at spørringen skal returnere spotprisen for en gitt dato og klokkeslett.

Etter de dataene vi har lagt inn i databasen forventer vi at første spørring skal returnere 120,2, og andre spørring skal returnere 14,22.

| Spotprisdata | Omr nr | Dato | kl:00.00 | kl:03.00 | kl:06.00 | kl:09.00 | kl:12.00 | kl:15.00 | kl:18.00 | kl:21.00 |
|--------------|--------|------------|----------|----------|----------|----------|----------|----------|----------|----------|
| | 1 | 01.01.2022 | 119,98 | 92,36 | 77,65 | 93,73 | 120,02 | 120,71 | 121,77 | 119,6 |
| | 2 | 30.01.2022 | 111,47 | 101,98 | 111,25 | 116,56 | 119,79 | 120,41 | 137,78 | 142,17 |
| | 3 | 10.04.2022 | 25,49 | 23,17 | 20,47 | 14,49 | 8,75 | 6,12 | 14,22 | 15,47 |
| | 4 | 07.08.2022 | 1,01 | 0,7 | 1,21 | 1,5 | 1,66 | 1,72 | 1,74 | 1,75 |
| | 5 | 19.06.2022 | 147,95 | 143,47 | 112,55 | 97,29 | 85,19 | 143,01 | 150,08 | 150,04 |

```
MATCH (o:Område {Navn: "N01"})-[]->(d:Dag {Dato: date('2022-01-01')})-[]->(t:Time)-[]->(s:Spotpris)
WHERE t.time = localTime("12:00")
RETURN s.Day_ahead AS Spotpris_KL_12_00_Euro_;
```

```
MATCH (o:Område {Navn: "N03"})-[]->(d:Dag {Dato: date('2022-04-10')})-[]->(t:Time)-[]->(s:Spotpris)
WHERE t.time = localTime("18:00")
RETURN s.Day_ahead AS Spotpris_KL_18_00_Euro_;
```

Dette er resultatet av begge spørringene:

| Spotpris_KL_12_00_Euro_ | |
|-------------------------|--------|
| 1 | 120.02 |

| Spotpris_KL_18_00_Euro | |
|------------------------|-------|
| 1 | 14.22 |

Spørring: Gjennomsnittspris for valgt dato og magasinkapasitet (Euro/TWh)

Her ønsker vi å hente ut gjennomsnittlig spotpris gjennom dagen, samt magasinkapasiteten den dagen. Ut fra dataen vi har lagt inn ønsker vi at denne spørringen skal returnere gjennomsnittspris på: 108,275 og magasinkapasitet på: 5,992993. Disse beregningene har vi kontrollert og vi ser at de stemmer overens med resultatet fra spørringen gjort mot databasen.

| | | | | | | | | | | |
|---------------------|------------|------------|----------|----------|--------------|-----------------|-------------|------------------------|----------|----------|
| Magasinfylling data | Dato | Omr nr | Omr type | | Fyllingsgrad | Kapasitet (TWh) | Fylling TWh | Neste publiseringsdato | | |
| | 01.01.2022 | 1 | EL | | 0,6531774 | 5,992993 | 3,9142497 | 1.11.2022 | | |
| Spotprisdata | Omr nr | Dato | kl:00.00 | kl:03.00 | kl:06.00 | kl:09.00 | kl:12.00 | kl:15.00 | kl:18.00 | kl:21.00 |
| | 1 | 01.01.2022 | 119,98 | 92,36 | 77,65 | 93,73 | 120,02 | 120,71 | 121,77 | 119,6 |

Gj.snitt: 108,2275 ▾

```
MATCH (o:Område {Navn: "N01"}) - [] -> (d:Dag {Dato:
date('2023-01-01')}) - [] -> (t:Time) - [] -> (s:Spotpris)
WITH AVG(s.Day_ahead) AS avgSpotpris, d
MATCH (d) - [] -> (m:Magasinfylling)
RETURN avgSpotpris, m.KapasitetTWh AS MagKapTWh;
```

Dette er resultatet:

| | avg Spotpris | MagKap Twh |
|---|--------------|------------|
| 1 | 108.275 | 5.992993 |

Spørring: Fyllingsgrad over en gitt dag

I denne spørringen ønsker vi å hente ut fyllingsgraden i et magasin for en gitt dag. Så i dette tilfellet vil vi at spørringen skal returnere fyllingsgrad på: 0,43423307 som vi ser i datasettet over de dataene vi har satt inn i databasen.

| Magasinfylling data | Dato | Omr nr | Omr type | Fyllingsgrad | Kapasitet (TWh) | Fylling TWh | Neste publiseringsdato |
|---------------------|------------|--------|----------|--------------|-----------------|-------------|------------------------|
| | 01.01.2022 | 1 | EL | 0,6531774 | 5,992993 | 3,9142497 | 1.11.2022 |
| | 30.01.2022 | 2 | EL | 0,43423307 | 33,928802 | 14,7330065 | 09.02.2022 |
| | 10.04.2022 | 3 | EL | 0,25474164 | 9,113162 | 2,321502 | 20.04.2022 |
| | 07.08.2022 | 4 | EL | 0,8738301 | 20,820948 | 18,193972 | 17.08.2022 |
| | 19.06.2022 | 5 | EL | 0,3941396 | 17,444757 | 6,8756757 | 29.06.2022 |

```
MATCH (o:Område {Navn: "N02"}) - [] -> (d:Dag {Dato:
date('2022-01-30')})
MATCH (d) - [] -> (m:Magasinfylling)
RETURN d.Dato as Dag, m.Fyllingsgrad AS FyllingsGrad
```

Denne spørringen begynner med en match setningen som starter på område (NO2) til angitt dato, også matcher vi dag med magasinfylling. Til slutt returner vi dato i form av Dag, og Fyllingsgrad.

Dette er resultatet:

| Dag | FyllingsGrad |
|--------------|--------------|
| "2022-01-30" | 0.43423307 |

Spørring: Fylling i TWh for nåtid eller gitt periode

Her ønsker vi å kjøre en spørring der vi kan hente ut fylling i TWh for en gitt dag.


Ut fra datasettet vi har brukt for å opprette databasen ønsker vi da at denne spørringen skal returnere en fylling TWh: 18,19372

| Magasinfylling data | Dato | Omr nr | Omr type | Fyllingsgrad | Kapasitet (TWh) | Fylling TWh | Neste publiseringsdato |
|---------------------|------------|--------|----------|--------------|-----------------|-------------|------------------------|
| | 01.01.2022 | 1 | EL | 0,6531774 | 5,992993 | 3,9142497 | 1.11.2022 |
| | 30.01.2022 | 2 | EL | 0,43423307 | 33,928802 | 14,7330065 | 09.02.2022 |
| | 10.04.2022 | 3 | EL | 0,25474164 | 9,113162 | 2,321502 | 20.04.2022 |
| | 07.08.2022 | 4 | EL | 0,8738301 | 20,820948 | 18,193972 | 17.08.2022 |
| | 19.06.2022 | 5 | EL | 0,3941396 | 17,444757 | 6,8756757 | 29.06.2022 |

```
MATCH (o:Område {Navn: "N04"}) - [] -> (d:Dag {Dato:
date('2022-08-07')})
MATCH (d) - [] -> (m:Magasinfylling)
RETURN d.Dato as Dag, m.FyllingTWh AS FyllingTWh
```

Vi begynner med en matchsetning som finner vei fra område (NO4) til dag, også en match setning som finner vei fra dag til magasinfylling. Deretter returnerer vi dag, og fylling TWh fra magasinsfylling.

Dette er resultatet:

| Dag | | FyllingTWh |
|--------------|---|------------|
| "2022-08-07" |  | 18.193972 |

Oppdatering av databasen

Betingelser for oppdatering

Ukentlig blir det utgitt ny magasindata.

Daglig blir det utgitt nye spotpriser.

Måling gjøres 3 dager før publiseringsdato.

Rutinen vår for å oppdatere databasen når nye data blir tilgjengelig er som følger:

1. Hver dag blir det laget en ny node for en dag, for hvert elspotområde. Det blir også laget timer som går til spotpriser
2. Ukentlig legger man på noder for magasindata, som gjelder for de forskjellige dagene det har kommet ut nye data for. Dette skjer etter hver publiseringsdato

Oppdatering av dataene ved bruk av Cypherspørringer:

Opprette dato:

```
CREATE (:Dag {Dato: date("dato")}), (:Dag {Dato: date("dato")}), (:Dag {Dato: date("dato")})
```

Koble til område:

```
MATCH (d:Dag), (o:`Område`)  
WHERE d.Dato = date("dato") AND o.Navn="legg til område"  
CREATE (o)-[:`dato`]->(d)
```

Legge til klokkeslett:

```

MATCH (d:Dag)
WHERE d.Dato=date('dato')
CREATE
(:Time {time: localtime('00:00')})<-[:`00:00`]- (d),
(:Time {time: localtime('03:00')})<-[:`03:00`]- (d),
(:Time {time: localtime('06:00')})<-[:`06:00`]- (d),
(:Time {time: localtime('09:00')})<-[:`09:00`]- (d),
(:Time {time: localtime('12:00')})<-[:`12:00`]- (d),
(:Time {time: localtime('15:00')})<-[:`15:00`]- (d),
(:Time {time: localtime('18:00')})<-[:`18:00`]- (d),
(:Time {time: localtime('21:00')})<-[:`21:00`]- (d)

```

Opprette spotpris:

```

MATCH
(o:`Område`{Navn:"område"}),
(o)-[dr:`dato`]->(d),
(d:Dag),
(t:Time),
(d)-[r:`klokkeslett`]->(t)
CREATE
(t)-[:HAR_SPOTPRIS]->(:Spotpris{Valuta:"Euro",Day_ahead:pris})

```

Denne spørringen oppretter magasinfylling, og kjøres da kun en gang i uken:

```

MATCH
(o:Område{Navn:'område'}),
(o)-[]->(d)
CREATE
(d)-[:HAR_MAGASINSTATISTIKK]->(magasin:Magasinfylling {
Fyllingsgrad:,
KapasitetTWh:,
FyllingTWh:,
NestePubliseringsDato: date('dato')
}))

```

Refleksjon

Designet til databasen ble jobbet med gjennom en prosess hvor vi begynte på tavle, der vi ønsket å finne naturlige koblinger mellom datasettene.

Etter dette tegnet vi det inn i et tegneprogram for å visualisere det bedre og mer konkret. Når vi følte vi hadde funnet et design som var organisert og ga mening, begynte vi å sette inn dataene med spørringer i Neo4j.

Under arbeidet med å utforme spørringene kom vi over noen problemer med at spørringene ble kjørt uten problemer, men ingenting ble returnert.

Forslag til forbedringer:

- Bedre strukturering av dataene
- Bruke egenskaper for å representere dag-nodene, slik at man slipper å hoppe gjennom flere noder for å søke seg frem til spotpris
- Gi relasjoner mer beskrivende navn, dette valgte vi fordi vi synes det var nokså oversiktlig - men til ettertanke hadde det kanskje vært bedre å kalle relasjonen mellom område og dag "har dag/dato" istedenfor selve datoen.
- Når man har noen verdier som repeterer seg igjen og igjen (som for dato og klokkeslett), så kan man gjøre det så man kan referere enklere til noden ved å la nodenavnet tilsvare en spesifikk verdi, istedenfor å måtte spesifisere en variabel i noden. For nodene for elspotområdene kunne vi hatt nodene oppkalt etter navnet på elspotområdet. Da hadde spørringene blitt enklere ettersom man hadde sluppet å referere til et variabel.

Dokumentdatabase (MongoDB)

Dokumentdatabase:

Installasjon:

Alle på gruppen fikk nedlastet mongoshell.

Vi brukte det til å opprette dokumentene, hente ut data og legge inn data.

Design av dokumentdatabase:

Med mongodb bruker vi datasettene for værdata og magasindata.

Dette kan brukes f.eks. til å se på hvordan nedbør henger sammen med fyllingsgraden i magasinene.

Vi antar at:

- Det er for det meste kommersielle og privatpersoner som kommer til å bruke databasen.
- Det betyr at folk er mest interessert i de dataene tilhørende det elspotområdet de selv bor i.
- Vi antar også at brukerne kommer til å svært sjeldent være interessert i værdataene sammen med magasindataene
- og at de nesten aldri kommer til å være interessert i værdataene for seg selv.

Først kan man tenke seg to dokumenter for hvert datasett:

```
magasin_data [  
    {  
      dato: ...  
      OmrType: ...  
      Fyllingsgrad: ...  
      Kapasitet_Twh: ...  
      Fylling_Twh: ...  
      Fyllingsgrad_forrige_uke: ...  
      Endring_i_fyllingsgrad: ...  
      Neste_pub_Dato: ...  
    }  
]  
  
værddata [  
    {  
      navn: 'navn på stasjon'  
      Nedbør: ...  
      Middelvind: ...  
      Middelterperatur: ...  
      Maksimumstemperatur: ...  
      Minimumstemperatur: ...  
      Fylke: ...  
      Område_nr: ...  
    }  
]
```

Men det blir svært vanskelig å hente ut data fra dette oppsettet.
Det blir en del jobb å filtrere for hvilket magasinområde det er snakk om.
Siden magasindataene og magasindataene er i forskjellige dokumenter uten noen
kobling, betyr det at spørringene kan bli litt vrie for å hente ut data.

Istedet kan man legge dataene fra værstasjoner for en dag sammen med magasindataene for en dag

```
magasin_data [  
    {  
        dato: ...  
        OmrType: ...  
        Fyllingsgrad: ...  
        Kapasitet_Twh: ...  
        Fylling_Twh: ...  
        Fyllingsgrad_forrige_uke: ...  
        Endring_i_fyllingsgrad: ...  
        Neste_pub_Dato: ...  
        værddata [  
            {  
                navn: 'navn på stasjon'  
                Nedbør: ...  
                Middelvind: ...  
                Middelterperatur: ...  
                Maksimumstemperatur: ...  
                Minimumstemperatur: ...  
                Fylke: ...  
                Område_nr: ...  
            }  
        ]  
    }  
]
```

Men det er fortsatt sånn at man gjerne må filtrere på magasinområdet det er snakk om. Istedet kan man lage et dokument for hvert magasinområde:

```
område_1 [  
    {  
      dato: ...  
      OmrType: ...  
      Fyllingsgrad: ...  
      Kapasitet_Twh: ...  
      Fylling_Twh: ...  
      Fyllingsgrad_forrige_uke: ...  
      Endring_i_fyllingsgrad: ...  
      Neste_pub_Dato: ...  
      værddata [  
        {  
          navn: 'navn på stasjon'  
          Nedbør: ...  
          Middelvind: ...  
          Middelterperatur: ...  
          Maksimumstemperatur: ...  
          Minimumstemperatur: ...  
          Fylke: ...  
          Område_nr: ...  
        }  
      ]  
    }  
  ]  
]
```

Men et problem er at man får med veldig mye værdata når man skal hente ut magasindata. Det er ikke alltid man er interessert i all den dataen. Man kan ha værdataene lagret i et annet dokument, så kan man bruke en foreign key til å referere til værdataene for et område på en dato.

```
område_1 [
  {
    dato: ...
    OmrType: ...
    Fyllingsgrad: ...
    Kapasitet_Twh: ...
    Fylling_Twh: ...
    Fyllingsgrad_forrige_uke: ...
    Endring_i_fyllingsgrad: ...
    Neste_pub_Dato: ...
    værdata_id: ...
  }
]

værdata_område_1 [
  {
    _id: ...    // værdata_id
    dag {
      værstasjoner [
        {
          navn: 'navn på stasjon'
          Nedbør: ...
          Middelvind: ...
        }
      ]
    }
  }
]
```

```

        Middeltemperatur: ...
        Maksimumstemperatur: ...
        Minimumstemperatur: ...
        Fylke: ...
    }
]
}
]

```

Da er databasen i mål.

Nå kan man lett filtrere dataene for dager, etter elspotområde, og man kan lett benytte værdataene, uten å få dem med når de er uinteressante.

I dokumentdatabasen setter vi opp totalt 10 dokumenter:

1. Område_1
2. Område_2
3. Område_3
4. Område_4
5. Område_5
6. Værdata_område_1
7. Værdata_område_2
8. Værdata_område_3
9. Værdata_område_4
10. Værdata_område_5

Disse kolleksjonene holder på informasjon om magasinstatstikk og værdata for et område. De fem første kolleksjonene vil da inneholde dette (Område_x):

Dato

OmrType

Fyllingsgrad

Kapasitet_TWh

Fylling_TWh
Fyllingsgrad_forrige_uke
Endring_i_fyllingsgrad
Neste_pub_dato

De fem neste vil inneholde på dette (Værdata_område_x):

dag - dato

Værstasjoner - dette vil være et innebygd dokument som inneholder værdata for den gitte dagen.

- Navn
- Nedbør
- Middelvind
- Middeltemperatur
- Maksimumstemperatur
- Minimumstemperatur
- Fylke

Databasen vil ha en god del flere lesninger enn skrivninger.

Databasen vil oppdateres kun 1 gang per dag (for værdata), i tillegg til 1 ekstra gang per uke (for magasindata).

Etter innsetting av data kan vi bruke optimistic locking for å gi databasen et versjonsnummer.

Hvis det er feildata i databasen kan vi gå tilbake til en tidligere versjon av den.

Det er ikke nødvendig at endringene i databasen vår synes med en gang, og dataene oppdateres også relativt sjeldent. Derfor kan det være lurt å bruke transaksjoner, siden det ikke vil gå så mye utover ytelsen til databasen.

Skriving av spørringer:

Fyllingsgrad over tid

Denne spørringene vil hente ut fyllingsgraden for et område over en viss periode, dette spesifiserer vi innenfor match dato.

Denne spørringen utføres da mot område 2, vi bruker match for å filtrere dokumenter i kolleksjonen basert på en bestemt tidsperiode. \$gte (greater than or equal), \$lte (less than or equal), brukes da for å opprette en tidsramme.

Group, grupperer dokumentene som oppfyller kravene fra match, og vi bruker id: null fordi vi vil beregne forskjellen i fyllingsgraden over hele perioden. \$first og \$last brukes for å hente første og siste verdi av fyllingsgrad.

\$project, beregner forskjellen i fyllingsgraden mellom start og sluttspunkt som blir hentet i \$group. Så bruker vi \$subtract for å trekke fyllingsgradstart fra fyllingsgradend, dette gir oss da DifferenceInFyllingsgrad, altså forskjellen i fyllingsgraden over denne perioden.

```
const docs = db.område_2.find({
  Dato: { $gte: ISODate("2018-08-12"), $lte: ISODate("2018-08-26") }
}).sort({ Dato: 1 }).toArray();

const start = docs[0].Fyllingsgrad;
const end = docs[docs.length - 1].Fyllingsgrad;

const DifferenceInFyllingsgrad = end - start;
```

```
[
  {
    _id: ObjectId("652693ae3eacadd3cbeba379"),
    Dato: ISODate("2018-07-01T00:00:00.000Z"),
    Fyllingsgrad: 0.73982006
  },
  {
    _id: ObjectId("652693ae3eacadd3cbeba37a"),
    Dato: ISODate("2018-07-08T00:00:00.000Z"),
    Fyllingsgrad: 0.7357676
  }
]
```

Fyllingsgrad nå

I denne spørringen ønsker vi å hente ut fyllingsgraden som er nå.

Vi bruker `findOne` for å hente opp dokumentet i `område_2`, og bruker kriterier `omrType` "EL" og `dato`, for å finne fyllingsgraden for denne datoen. Dette vil da returnere det første dokumentet med disse kriteriene. Vi setter fyllingsgrad: 1, for at den skal gi oss verdien til fyllingsgraden tilbake.

```
db.område_2.findOne({  
  OmrType: 'EL',  
  Dato: ISODate("2018-08-12")  
}, { Fyllingsgrad: 1 })
```

Output:

```
endelig> db.område_2.findOne({  
...   OmrType: 'EL',  
...   Dato: ISODate("2018-08-12")  
... }, { Fyllingsgrad: 1 })  
{ _id: ObjectId("652fad41f2bf71878738945d"), Fyllingsgrad: 0.6285738 }  
endelig> _
```

Vi ser at fyllingsgraden på område 2 for dagen 12-08-2018 er 0.6285738

Samlet nedbør og forskjell i fyllingsgrad på et visst område til en viss tid

Under match:

Det filtreres først etter datoverdier i dokumentet for magasindata og værdata.

Under lookup:

Så koples værdataen sammen med magasindataen via værdata_id. Dokumentet værdata_id knytter til, blir kalt værdata.

Under group:

Så manipuleres datene for å få summen av nedbør og forskjellen i fyllingsgrad. For fyllingsgrad tar man første og siste dokument av magasindata som blir funnet. For nedbør samler man alle verdiene fra værdata-dokumentene som blir funnet. Verdiene settes inn i variabler.

```
const docs = db.område_2.find({
  Dato: {
    $gte: ISODate("2018-08-12"),
    $lte: ISODate("2018-08-19")
  }
}).sort({ Dato: 1 }).toArray();

docs.forEach(doc => {
  const værdata = db.værdata_område_2.findOne({ _id:
doc.værdata_id });
  doc.værdata = værdata;
});

let totalNedbør = 0;
docs.forEach(doc => {
  doc.værdata.dag.forEach(dag => {
    dag.værstasjoner.forEach(stasjon => {
      totalNedbør += stasjon.værstasjon.nedbør;
    });
  });
});

const startFyllingsgrad = docs[0].Fyllingsgrad;
const endFyllingsgrad = docs[docs.length - 1].Fyllingsgrad;
const FyllingsgradDifference = endFyllingsgrad -
startFyllingsgrad;

print({
```

```
FyllingsgradDifference: FyllingsgradDifference,  
totalNedbør: totalNedbør  
});
```

```
[  
  {  
    _id: null,  
    'totalNedbør': 27.8,  
    FyllingsgradDifference: 0.021571200000000013  
  }  
]
```

Totalt nedbør for perioden har vært 27.8mm, og forskjellen i fyllingsgrad har vært 0.021

Endring i nedbør og endring i fyllingsgrad

Under match:

Det filtreres for dokumenter der datoen er mellom de spesifiserte dagene.

Under lookup:

værdata_id i magasindataen kobles til det matchende _id for værdataen for området.

Under group:

Av dokumentene som ble funnet, tar man verdien funnet i første dokument minus verdien funnet i siste dokument.

Verdiene settes som variabler.

Under project:

Man returnerer variablene.

```
const docs = db.område_2.find({
  Dato: {
    $gte: ISODate("2018-08-12"),
    $lte: ISODate("2018-08-19")
  }
}).sort({ Dato: 1 }).toArray();

docs.forEach(doc => {
  const værdata = db.værdata_område_2.findOne({ _id:
doc.værdata_id });
  doc.værdata = værdata;

  let nedbørValues = [];
  værdata.dag.forEach(dag => {
    dag.værstasjoner.forEach(stasjon => {
      nedbørValues.push(stasjon.værstasjon.nedbør);
    });
  });
  doc.nedbørValues = nedbørValues;
});
```

```
const startFyllingsgrad = docs[0].Fyllingsgrad;
const endFyllingsgrad = docs[docs.length - 1].Fyllingsgrad;
const startNedbør = docs[0].nedbørValues[0];
1].nedbørValues[docs[docs.length - 1].nedbørValues.length - 1];
```

```
const FyllingsgradDifference = endFyllingsgrad -
startFyllingsgrad;
const NedbørDifference = endNedbør - startNedbør;

print({
  FyllingsgradDifference: FyllingsgradDifference,
  NedbørDifference: NedbørDifference
});
```

```
[
  {
    _id: null,
    FyllingsgradDifference: 0.021571200000000013,
    NedbørDifference: -20.400000000000002
  }
]
```

Vi ser at forskjellen i fyllingsgrad for perioden har vært 0.0215, og forskjellen i nedbør har vært -20mm

Samlet fyllingsgrad, nedbør og Thw over tid

Under match:

Man finner dokumenter der datoen er innenfor intervallet til de spesifiserte datene.

Under lookup:

Man bruker værddata_id i magasindataen for å finne den relaterte værddataen er magasindataene innenfor det spesifiserte tidsintervallet.

Under group:

Man summerer verdiene i de funnede dokumentene.

Verdiene lagres i variabler.

Under porject:

Man returnerer variablene.

```
const docs = db.område_2.find({
  Dato: {
    $gte: ISODate("2018-08-12"),
    $lte: ISODate("2018-08-19")
  }
}).toArray();

docs.forEach(doc => {
  const værddata = db.værddata_område_2.findOne({ _id: doc.værddata_id });
  doc.værddata = værddata;
});

let totalNedbør = 0;
let totalFyllingsgrad = 0;
let totalThw = 0;

docs.forEach(doc => {
  totalFyllingsgrad += doc.Fyllingsgrad;
  totalThw += doc.Kapasitet_TWh;

  // Unwind værddata.dag.værstasjoner and sum up nedbør
  doc.værddata.dag.forEach(dag => {
    dag.værstasjoner.forEach(stasjon => {
      totalNedbør += stasjon.værstasjon.nedbør;
    });
  });
});

print({
  totalNedbør: totalNedbør,
  totalFyllingsgrad: totalFyllingsgrad,
  totalThw: totalThw
});
```

```
[
  {
    id: null,
    totalNedbør: 27.8,
    totalFyllingsgrad: 1.2787188,
    totalThw: 67.857604
  }
]
```

Oppdatering av databasen:

1. Samle inn og forberede data
2. Utføre oppdatering av databasen - oppdateringsoperatører \$set, \$push, \$pull, \$addToSet → disse brukes til å oppdatere allerede eksisterende dokumenter, så de brukes da hvis vi skal endre dataene i et dokument
3. Legge inn ny data - \$insertOne, \$insertMany → brukes når nye dokumenter skal legges inn
4. Etter at oppdateringer er gjort og nye data er lagt inn, er det viktig å sjekke konsistens. Viktig å vurdere hvordan disse nye dataene vil påvirke den dataen som allerede er i databasen.

Data om magasin statistikk publiseres en dag i uken (målingene blir gjort tre dager før publiseringsdatoen), oppdateringen av disse dataene vil da skje ettersom de blir publisert.

Værstatistikken blir publisert hver dag, og oppdateringen av denne dataen vil derfor skje en gang om dagen.

For å legge til værdata kan man bruke denne spørringen:

```
db.værdata_område_X.update(  
  { "dag.dato": new Date("yyyy-mm-dd") },  
  {  
    $push: {  
      "dag.værstasjoner": {  
        $each: [  
          {  
            værstasjon: {  
              Stasjon: "SNXXXXX",  
              navn: "navn på stasjon1",  
              "Middel_av_middelvind": 0,  
              "Middeltemperatur": 0,  
              "Maksimumtemperatur": 0,  
              "Minimumstemperatur": 0,  
              Nedbør: 0,  
              Fylke: "navn på fylke"  
            }  
          },  
          {  
            værstasjon: {  
              Stasjon: "SNXXXXX",
```

```

        navn: "navn på stasjon2",
        "Middel_av_middelvind": 0,
        "Middeltemperatur": 0,
        "Maksimumtemperatur": 0,
        "Minimumstemperatur": 0,
        Nedbør: 0,
        Fylke: "navn på fylke"
      }
    },
  ]
}
}}, { upsert: true });

```

Den putter inn ny data under dag.dato.værstasjoner, der dato matcher med den spesifiserte datoen i spørringen.

Hvis datoen ikke finnes fra før av i dokumentet, vil en ny dato bli lagd i samme slengen, som dataen kan puttes under.

Eksempel på å putte inn ny værdata i databasen:

```

db.værdata_område_2.update(
  { "dag.dato": new Date("2023-08-06") },
  {
    $push: {
      "dag.værstasjoner": {
        $each: [
          {
            værstasjon: {
              Stasjon: "SN47270",
              navn: "Karmøy-Hydro",
              "Middel_av_middelvind": 3.5,
              "Middeltemperatur": null,
              "Maksimumtemperatur": null,
              "Minimumstemperatur": null,
              Nedbør: 23,
              Fylke: "Rogaland"
            }
          },
          {
            værstasjon: {
              Stasjon: "SN40880",

```



```

        navn: "Hovden-Lundane",
        "Middel_av_middelvind": null,
        "Middeltemperatur": null,
        "Maksimumtemperatur": null,
        "Minimumstemperatur": null,
        Nedbør: 0,
        Fylke: "Rogaland"
    }
}
]
}
},
{ upsert: true }
);

```

For å legge til data om et magasinområde, må værdataen først legges til, fordi magasinområde har en værdata_id til id-er i værdata dokumentene.

For å legge til magasindata kan denne spørringen brukes:

```

db.område_X.insertOne([
  {
    Dato: ISODate("yyyy-mm-dd"),
    OmrType: "EL",
    Fyllingsgrad: 0,
    Kapasitet_TWh: 0,
    Fylling_TWh: 0,
    Fyllingsgrad_forrige_uke: 0,
    Endring_fyllingsgrad: 0
  }
]);

```

Så kan denne spørringen brukes til å legge til værdata_id:

```

db.område_X.aggregate([
  {
    $match: {
      Dato: ISODate("yyyy-mm-ddT00:00:00.000Z")
    }
  },

```

```

{
  $lookup: {
    from: "værddata_område_X",
    localField: "Dato",
    foreignField: "dag.dato",
    as: "joined_data"
  }
},
{
  $unwind: "$joined_data"
},
{
  $addFields: {
    værddata_id: "$joined_data._id"
  }
},
{
  $project: {
    joined_data: 0
  }
},
{
  $merge: {
    into: "område_X",
    whenMatched: "merge",
    whenNotMatched: "discard"
  }
}
}]

```

Refleksjon

Sånn så dokumentene for værdata for et område ut.

```
værdata_område_X [  
  {  
    _id: ...    // værdata_id  
    dato: ...  
    dag {  
      værstasjoner [  
        {  
          navn: 'navn på stasjon'  
          Nedbør: ...  
          Middelvind: ...  
          Middelsestemperatur: ...  
          Maksimumstemperatur: ...  
          Minimumstemperatur: ...  
          Fylke: ...  
        }  
      ]  
    }  
  }  
]
```

Men det er ikke noe poeng å ha værstasjoner [] under dag {}. Istedenfor kan man ta vekk dag {}, for så å ta værstasjoner på samme nivå som _id:

```
værdata_område_X [  
  {  
    _id: ...    // værdata_id  
    dato: ...  
    værstasjoner [  
      {  
        navn: 'navn på stasjon'  
        Nedbør: ...  
        Middelvind: ...  
        Middelsestemperatur: ...  
      }  
    ]  
  }  
]
```

```
        Maksimumstemperatur: ...
        Minimumstemperatur: ...
        Fylke: ...
    }
]
}
```

Et problem med oppsettet vårt var at det kan bli utfordrende å koble data fra forskjellige elspotområder. Vi tenkte at de fleste spørringer kommer til å være for å hente ut data for det elspotområdet de selv bor i.

I de tilfellene der data skal hentes ut fra flere elspotområder, så kunne vi f.eks. laget et dokument med med foreign keys til alle de forskjellige elspotområdene.

Kolonnefamiliedatabase:

Installasjon - DataStax og Docker:

Alle gruppe medlemmene har lastet ned og installert DataStax Desktop v2 for Windows via [denne lenken](#).

Vi støtte på problemer med å installere Docker via DataStax- installeren og måtte laste ned Docker via [denne lenken](#). Mest sannsynlig fordi vi først prøvde å installere DataStax V1.

Når vi nå har installert og har kjørt Docker, skulle vi være i stand til å starte Apache Cassandra-stacken i DataStax.

Alternativ installasjon - Cassandra med Web GUI:

Flere av oss hadde derimot store problemer med å ta i bruk DataStax. Vi bestemte oss dermed også for å prøve en alternativ fremgangsmåte etter tips fra medstudenter i emnets discordkanal. Vi forsøkte å opprette et web GUI for Cassandra. Dette gjorde vi på denne måten i Windows Powershell:

1. Start Cassandra med Docker

```
docker run --name cassandra-container -p 9042:9042 -d  
cassandra:latest
```

2. Pull Docker-image for Web GUI:

```
docker pull markusgulden/cassandra-web
```

3. Kjør webgrensesnitt for CQL:

```
docker run -d -p 3000:3000 \  
-e CASSANDRA_HOST_IPS=172.17.0.1 \  
-e CASSANDRA_PORT=9042 \  
-e CASSANDRA_USER=cassandra \  
-e CASSANDRA_PASSWORD=cassandra \  
markusgulden/cassandra-web
```

4. Se og kontroller at begge deler kjører

```
docker ps
```

5. Åpne nettleser

- http://localhost:3000
- Trykk på Execute i høyre/øvre hjørne
- Skriv spørringer

Design av kolonnefamiliedatabase:

For kolonnefamilie databasen bruker vi datasettene for nettleie og værdata.

Fylke

| fylke_navn | fylke_nr | dato | kvantum_per_fylke | snittEffektsEks | snittEffektInk | snittEnergiEks | snittEnergiInk | snittFastleddEks | snittFastleddInk | snittomregnetOreEks | snittOmregnetOreInk |
|------------|----------|------------|-------------------|-----------------|----------------|----------------|----------------|------------------|------------------|---------------------|---------------------|
| Oslo | 1 | 2018-08-07 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Agder | 1 | 2018-08-07 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Værdata

| fylke_navn | fylke_nr | dato | stasjon_navn | stasjon_id | middel_av_middelvind | maksimumstemperatur | minimumstemperatur | nedbør |
|------------|----------|------------|--------------|------------|----------------------|---------------------|--------------------|--------|
| Oslo | 1 | 2018-08-07 | 45 | | 0 | 0 | 0 | 0 |
| Agder | 1 | 2018-08-07 | 34 | | 0 | 0 | 0 | 0 |

Databasen kommer for det meste til å bli brukt for uthenting av data om det fylket en person bor i.

For å få enklere spørringer burde nettleie og værstatistikk-tabellene splittes opp etter fylke. Så istedenfor at det er en kolonnefamilie for alle fylkene, er det en kolonnefamilie for hvert fylke. Her er et eksempel på hvordan en tilfeldig kolonnefamilie for fylket.

| | | | | | | | | | | | | |
|------|------------|----------|------------|-------------------|-----------------|----------------|----------------|----------------|------------------|------------------|---------------------|---------------------|
| Oslo | Oslo | | | | | | | | | | | |
| | fylke_navn | fylke_nr | dato | kvantum_per_fylke | snittEffektsEks | snittEffektInk | snittEnergiEks | snittEnergiInk | snittFastleddEks | snittFastleddInk | snittomregnetOreEks | snittOmregnetOreInk |
| | Oslo | 1 | 2018-08-07 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Oslo | 1 | 2018-08-06 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | | | | |
|--------------|--------------|----------|------------|--------------|------------|----------------------|---------------------|--------------------|--------|--|
| Oslo Værdata | Oslo Værdata | | | | | | | | | |
| | fylke_navn | fylke_nr | dato | stasjon_navn | stasjon_id | middel_av_middelvind | maksimumstemperatur | minimumstemperatur | nedbør | |
| | Oslo | 1 | 2018-08-07 | | 45 | 0 | 0 | 0 | 0 | |
| | Oslo | 1 | 2018-08-07 | | 12 | 0 | 0 | 0 | 0 | |

I tillegg kan man inkludere id og timestamp som gjør det enklere å håndtere databasen.

| | | | | | | | | | | | | | | |
|------|------------|----------|------------|-------------------|-----------------|----------------|----------------|----------------|------------------|------------------|---------------------|---------------------|----|------------|
| Oslo | Oslo | | | | | | | | | | | | | |
| | fylke_navn | fylke_nr | dato | kvantum_per_fylke | snittEffektsEks | snittEffektInk | snittEnergiEks | snittEnergiInk | snittFastleddEks | snittFastleddInk | snittomregnetOreEks | snittOmregnetOreInk | id | created_at |
| | Oslo | 1 | 2018-08-07 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | now() |
| | Oslo | 1 | 2018-08-06 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | now() |

| | | | | | | | | | | | | | |
|--------------|--------------|----------|------------|--------------|------------|----------------------|---------------------|--------------------|--------|----|------------|--|--|
| Oslo Værdata | Oslo Værdata | | | | | | | | | | | | |
| | fylke_navn | fylke_nr | dato | stasjon_navn | stasjon_id | middel_av_middelvind | maksimumstemperatur | minimumstemperatur | nedbør | id | created_at | | |
| | Oslo | 1 | 2018-08-07 | | 45 | 0 | 0 | 0 | 0 | 0 | now() | | |
| | Oslo | 1 | 2018-08-07 | | 12 | 0 | 0 | 0 | 0 | 1 | now() | | |

Konsekvensene av konsistens

Det må sørges for at databasen brukes riktig.

Så det må f.eks. ikke lages kolonnefamilier for samme fylke flere ganger.

For det kan man bruke 'if not exists' når man oppretter kolonnefamilier:

```
CREATE TABLE IF NOT EXISTS kolonne_familie_navn ( )
```

LWT (Lettvektstransaksjoner):

LWT brukes til å sette forsikringer om at raden man setter inn er unik.

Det går utover ytelsen, men siden vår database sjeldent blir oppdatert (fylke oppdateres en gang i uken, værdata oppdateres en gang hver dag), så er ikke det noe å bekymre seg om. I vårt tilfelle vil LWT fungere slik:

```
INSERT INTO Oslo (fylke_navn, fylke_nr...) IF NOT EXISTS.
```

Hver rad i databasen må være unik fordi dato på Oslo vil alltid være unikt, og på Oslo værdata så må alltid kombinasjonen av dato og fylke_nr/fylke_navn være unikt.

For lesinger kan man bruke lav-consistency-nivå. Dette gir bedre leseytelse.

Spørringer for å opprette databasen:

dato kan tydeligvis representeres som INT. Timestamp kan også være int.

[Data Types | Apache Cassandra Documentation](#)

```
CREATE TABLE Fylke (  
  fylke_id UUID PRIMARY KEY,  
  fylke_navn TEXT,  
  fylke_nr INT,  
  created_at TIMESTAMP,  
  tariffdato map<TEXT, INT> //inneholder all data om  
  nettleie for en dato  
);
```

tariffdato<TEXT,INT> vil se slik ut:

Value-ene i MAPet er bare eksempler

```
{  
  tariffdato: ...  
  kvantum_pr_fylke: 0,  
  snitt_effekt_eks: 0,  
  snitt_effekt_ink: 0,  
  snitt_energi_eks: 0,  
  snitt_energi_ink: 0,  
  snitt_fastledd_eks: 0,  
  snitt_fastledd_ink: 0,  
  snitt_omregnet_ore: 0,  
  snitt_omregnet_ore_ink: 0,  
}
```

```
CREATE TABLE vaerdata (  
  data_id UUID PRIMARY KEY,  
  vaerdata_for_fylke TEXT,  
  created_at TIMESTAMP,  
  dag map<TEXT, FLOAT> //inneholder målinger til en  
  værstasjon for en dato  
);
```

dag <TEXT,INT> vil se slik ut:

Value-ene i MAPet er bare eksempler:

```
{
```



```
'navn_på_stasjon': 0,  
'stasjon_ISO': ...,  
'nedbør': 0,  
'middelvind': 0,...}
```

Skriving av spørringer:

Helt enkle spørringer:

Vi har også gjort tester med enkle spørringer. For eksempel:

1. Henter nedbør og dato for nedbør fra tabellen vaerdata_agder

```
SELECT  
    nedbor,  
    dato  
FROM milepael3v2.vaerdata_agder;
```

| nedbor | dato |
|---------------------|------------|
| 0.20000000298023224 | 2021-04-01 |
| 1.2999999523162842 | 2021-03-01 |
| 0.4000000059604645 | 2021-05-01 |
| 0.10000000149011612 | 2021-06-01 |
| 2.200000047683716 | 2021-01-01 |
| 0.20000000298023224 | 2021-02-01 |

Total: 6 results

2. Finner gjennomsnittlig nedbørsmengde for Møre og Romsdal

```
SELECT AVG(nedbor)
FROM milepael3v2.vaerdata_moere_og_romsdal;
```

Rows

system.avg(nedbor)

| |
|-------------------|
| 4.028571605682373 |
|-------------------|

Total: 1 results

3. Hent ut snitteffektink for datoen 2021-01-01

```
SELECT snitteffektink
FROM "milepael3v2".agder
WHERE fylke_navn = 'Agder' AND dato = '2021-01-01'
ALLOW FILTERING;
```



```
cqlsh:milepael3v2>
cqlsh:milepael3v2>
cqlsh:milepael3v2> SELECT snitteffektink
... FROM "milepael3v2".agder
... WHERE fylke_navn = 'Agder' AND dato = '2021-01-01'
... ALLOW FILTERING;

snitteffektink
-----
36.48706
```

4. Hente ut gjennomsnittlig snitteffekteks i Agder i perioden 2021-01-01 til 2022-01-01

```
SELECT AVG(snitteffekteks)
FROM "milepael3v2".agder
WHERE fylke_navn = 'Agder'
AND dato >= '2021-01-01'
AND dato < '2022-01-01'
ALLOW FILTERING;
```

```
cqlsh:milepael3v2> SELECT AVG(snitteeffekteks)
... FROM "milepael3v2".agder
... WHERE fylke_navn = 'Agder'
... AND dato >= '2021-01-01'
... AND dato < '2022-01-01'
... ALLOW FILTERING;

system.avg(snitteeffekteks)
-----
12.49985

(1 rows)

Warnings :
Aggregation query used without partition key
```

Oppdatering av databasen:

Oppdatering av databasen må gjøres hver dag, siden vi mottar ny data fra målestasjoner og ny nettleiedata daglig. En rutine for å håndtere dette kan for eksempel se slik ut:

1. Innsamling av nye data
2. Data forberedes og valideres - datavasking
3. En backup blir tatt av databasen i tilfelle feil (nodetool snapshot vurderes)
4. INSERT INTO nettleie-tabellen til et fylke:
INSERT INTO værddata-tabellen til et fylke:
5. Håndtering av eventuelle duplikatdata (overskrive, ignorere eller slette?)

Dette er et eksempel på dataene for nettleie som skal settes inn i databasen for en dag.

```
2021-06-01T00:00:00,Husholdning,Agder,42,2106950,12.49985,36.487064,0,0,2999.964,3749.955,27.49967,55.23684
2021-06-01T00:00:00,Husholdning,Innlandet,34,2904560,7.5115027,30.24704,1.4257596,1.7821995,3435.46,4294.3247,25.030985,52.146393
2021-06-01T00:00:00,Husholdning,Møre og Romsdal,15,1921971,16.278221,41.20976,2.666944,3.33368,2338.1055,2922.6318,28.608816,56.623
2021-06-01T00:00:00,Husholdning,Nordland,18,2160406,16.886642,33.572803,0,0,3778.967,3778.967,35.781475,52.467636
2021-06-01T00:00:00,Husholdning,Oslo,3,4054046,19.15,44.800003,0,0,1104,1380,24.67,51.700005
2021-06-01T00:00:00,Husholdning,Rogaland,11,3165960,19.50909,45.247883,0,0,2158.4924,2698.1155,30.301552,58.73846
2021-06-01T00:00:00,Husholdning,Troms og Finnmark Romsa ja
Finnmårku,54,2339459,13.828021,23.482668,0,0,2982.7773,2982.7773,28.741907,38.396557
2021-06-01T00:00:00,Husholdning,Trøndelag,50,3110795,20.358747,46.3105,0,0,2197.7185,2747.1482,31.347342,60.04624
2021-06-01T00:00:00,Husholdning,Vestfold og Telemark,38,3145200,15.326135,40.018814,0,0,2762.6743,3453.3428,29.139505,57.285526
2021-06-01T00:00:00,Husholdning,Vestland,46,4376385,18.902153,44.48965,0,0,1842.3121,2302.8901,28.113712,56.0041
2021-06-01T00:00:00,Husholdning,Viken,30,8817916,18.73776,44.283474,0.4942,0.61775,1332.8046,1666.0057,25.520393,52.761765
2021-06-01T00:00:00,Hytter og fritidshus,Agder,42,181434,12.499812,36.486958,0,0,3539.947,4424.9336,100.99848,147.11029
2021-06-01T00:00:00,Hytter og fritidshus,Innlandet,34,431103,8.558919,31.435036,21.8415,27.301874,2550.659,3188.3235,78.87784,119.33369
2021-06-01T00:00:00,Hytter og fritidshus,Møre og
Romsdal,15,49648,15.628061,40.365116,5.937456,7.42182,2726.5793,3408.224,85.57378,127.797264
2021-06-01T00:00:00,Hytter og fritidshus,Nordland,18,94549,14.663847,31.07359,0,0,3596.6174,3596.6174,104.57928,120.98902
2021-06-01T00:00:00,Hytter og fritidshus,Oslo,3,2099,19.15,44.800003,0,0,1728,2160,62.35,98.8
2021-06-01T00:00:00,Hytter og fritidshus,Rogaland,11,113220,20.358063,46.308514,0,0,2646.4363,3308.0454,86.51897,129.00964
2021-06-01T00:00:00,Hytter og fritidshus,Troms og Finnmark Romsa ja
Finnmårku,54,136542,14.223921,22.776812,0,0,3086.4346,3086.4346,91.38478,99.937675
2021-06-01T00:00:00,Hytter og fritidshus,Trøndelag,50,216306,18.904722,42.583275,0,0,1994.1418,2492.6772,68.75827,104.90021
2021-06-01T00:00:00,Hytter og fritidshus,Vestfold og Telemark,38,152145,23.300829,49.972454,0,0,3229.6555,4037.0693,104.04222,150.89919
2021-06-01T00:00:00,Hytter og fritidshus,Vestland,46,120484,23.624235,50.392544,0,0,2518.7375,3148.4219,86.592674,129.10309
2021-06-01T00:00:00,Hytter og
```

fritidshus,Viken,30,493273,19.925123,45.736946,1.2595584,1.5744481,2190.1614,2737.7017,75.05702,114.651825

Det kommer inn data for de forskjellige fylkene, for husholdning og 'Hytter og fritidshus'.

Så må man sette inn dataene i riktig tabell med tanke på fylke. Så f.eks hvis man skal sette inn data for Agder er det disse dataene som skal benyttes:

```
2021-06-01T00:00:00,Husholdning,Agder,42,2106950,12.49985,36.487064,0,0,2999.964,3749.955,27.49967,55.23684
2021-06-01T00:00:00,Hytter og fritidshus,Agder,42,181434,12.499812,36.486958,0,0,3539.947,4424.9336,100.99848,147.11029
```

Vi kan bruke BATCH for å forhindre race condition.

BEGIN BATCH og APPLY BATCH sikrer at det ikke er noen race-condition på verdiene i databasen (flere endringer på same verdi kommer ikke til å føre til feil verdi)

IF NOT EXISTS forhindrer duplikater i databasen.

For å sette inn dataene kan denne spørringen brukes:

```
BEGIN BATCH
INSERT INTO "milepael3v2".agder
(id, created_at, dato, fylke_navn, fylke_nr, kvantum_per_fylke, snitteffekteks, snitteffektink, snittenergieks,
snittenergink, snittfastledeks, snittfastleddink, snittomregnetoreeks, snittomregnetoreink, tariffgruppe)
VALUES ( uuid(), '2021-06-01 00:00:00', '2021-06-01', 'Agder',
42,2106950,12.49985,36.487064,0,0,2999.964,3749.955,27.49967,55.23684, 'Husholdning' ) IF NOT EXISTS;
APPLY BATCH;

BEGIN BATCH
INSERT INTO "milepael3v2".agder
(id, created_at, dato, fylke_navn, fylke_nr, kvantum_per_fylke, snitteffekteks, snitteffektink, snittenergieks,
snittenergink, snittfastledeks, snittfastleddink, snittomregnetoreeks, snittomregnetoreink, tariffgruppe)
VALUES ( uuid(), '2021-06-01 00:00:00', '2021-06-01', 'Agder',
42,181434,12.499812,36.486958,0,0,3539.947,4424.9336,100.99848,147.11029, 'Hytter og fritidshus' ) IF NOT EXISTS;
APPLY BATCH;
```

Men det utføres svært sjeldent endringer av en verdi, så BATCH er egentlig unødvendig. Ny data kan settes in slik:

```
INSERT INTO "milepael3v2".agder
(id, created_at, dato, fylke_navn, fylke_nr, kvantum_per_fylke, snitteffekteks, snitteffektink, snittenergieks,
snittenergink, snittfastledeks, snittfastleddink, snittomregnetoreeks, snittomregnetoreink, tariffgruppe)
VALUES ( uuid(), '2021-06-01 00:00:00', '2021-06-01', 'Agder',
42,2106950,12.49985,36.487064,0,0,2999.964,3749.955,27.49967,55.23684, 'Husholdning' ) IF NOT EXISTS;

INSERT INTO "milepael3v2".agder
(id, created_at, dato, fylke_navn, fylke_nr, kvantum_per_fylke, snitteffekteks, snitteffektink, snittenergieks,
snittenergink, snittfastledeks, snittfastleddink, snittomregnetoreeks, snittomregnetoreink, tariffgruppe)
VALUES ( uuid(), '2021-06-01 00:00:00', '2021-06-01', 'Agder',
42,181434,12.499812,36.486958,0,0,3539.947,4424.9336,100.99848,147.11029, 'Hytter og fritidshus' ) IF NOT EXISTS;
```

Det kommer også inn værdata hver dag fra forskjellige værstasjoner

Her er et eksempel på noe av dataen som kan komme i en dag:

```
Arendal Lufthavn;SN36330;01.11.2017;3,5;9,9;15,1;5,4;-;Agder
Lista Fyr;SN42160;01.11.2017;9,5;11,3;12,2;6,4;2,8;Agder
Kristiansand - Sørskleiva;SN39150;01.11.2017;-;10,9;-;-;0;Agder
Kvitfjell;SN13160;01.11.2017;3,9;2,2;3,3;0,6;-;Innlandet
Hamar Ii;SN12290;01.11.2017;-;2,1;3,8;-0,5;0;Innlandet
Jan Mayen;SN99950;01.11.2017;11,4;-4,2;2,1;-7,6;4,8;Jan Mayen
```

Hver dag kommer det inn data fra hver værstasjon i hele landet.

Dataene må filtreres på bakgrunn av fylke, så kan de settes inn slik:

```
INSERT INTO milepael3v2.vaerdata_agder (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00', 'Agder', 2, '2017-11-01', 'Arendal Lufthavn', SN36330,
3.5, 9.9,15,1.5,4, null)
IF NOT EXISTS;
```

```
INSERT INTO milepael3v2.vaerdata_agder (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00', 'Agder', 2, '2017-11-01', 'Hovden - Lundane',
SN42160,9.5,11.3,12.2,6.4,2.8)
IF NOT EXISTS;
```

```
INSERT INTO milepael3v2.vaerdata_agder (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00','agder', 2, '2017-11-01', 'Kristiansand - Sørskleiva',
SN39150, null, 10.9, null, null, 0)
IF NOT EXISTS;
```

```
INSERT INTO milepael3v2.vaerdata_innlandet (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00','innlandet', X, '2017-11-01', 'Kvitfjell', SN13160, 3.5,
9.9,15,1.5,4, null)
IF NOT EXISTS;
```

```
INSERT INTO milepael3v2.vaerdata_innlandet (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00', 'innlandet', X, '2017-11-01', 'Hamar Ii', SN12290, null,
2.1,3.8,-0.5,0)
IF NOT EXISTS;
```

```
INSERT INTO milepael3v2.vaerdata_jan_mayen (id, created_at, fylke_navn, fylke_nr, dato,
stasjon_navn, stasjon_id, middel_av_middelvind, maksimumstemperatur, minimumstemperatur, nedbor)
VALUES ( uuid(), '2017-11-01 00:00:00','jan mayen', Y, '2017-11-01', 'Jan Mayen', SN99950,
11.4,-4.2,2.1,-7.6,4.8)
IF NOT EXISTS;
```

Refleksjon

Kolonner og superkolonner

I kolonnefamiliene har vi inkludert kolonner for fylkenavn og fylkenr. Disse kolonnene er unødvendige siden hvilket fylke det er snakk om er allerede spesifisert i navnet på kolonnefamilien.

En mulighet hadde vært å inkludere værdata som en superkolonne under dataene for nettleie for et fylke. Det kunne man ha gjort med et MAP og en CUSTOM TYPE som kunne representert værdataene. Slik:

```
CREATE TABLE Oslo (  
  dato DATE,  
  tariffgruppe TEXT,  
  værdata MAP<DATE, Værstasjon>  
  ...  
)
```

```
CREATE TYPE Værstasjon (  
  navn TEXT,  
  maksimumstemperatur FLOAT,  
  nedbør FLOAT,  
  ...  
)  
-----
```

Mangler og feil

På grunn av at mye tid har gått med til feilsøking, prøving og feiling med oppsett av Cassandra, Datastax og Docker, henger vi litt etter skjema og ting ble dessverre ikke helt som vi hadde tenkt til å begynne med. Samtidig hadde vi også frist for individuell innlevering som måtte prioriteres før den til slutt ble flyttet en uke fremover.

Blant annet er vi klare over at utstrakt bruk av ALLOW FILTERING vil føre til ressursbruk som kan være kostbart. Vi har nå satt opp en enkel database med eksempeldata, og tenker at dette ikke vil være så farlig her og nå.

Men; i realiteten ville vi gjort endringer i tabellenes struktur og lagt en bedre strategi for nøkler for å optimalisere spørringene.

Vi har heller ikke klart å besvare alle spørringene vi ønsket å kjøre mot databasen vår. Vi sitter igjen med følgende spørsmål som ikke har klart å lage spørringer for på nåværende tidspunkt:

1. Forskjell i pris i forhold til hytte og fritidshus og husholdning
2. Hvilken dato er nettleien dyrest i et visst fylke (snittomregneteksink)

Valg av datasett for denne oppgaven:

Spørringene i denne delen av oppgaven blir ikke like interessante som de andre databasene vi har gjort, siden vi nå har brukt to datasett som egentlig ikke gir like mye mening uten noen av de andre datasettene. Vi valgte å gå for disse datasettene da det sto i oppgaven at vi måtte bruke alle fire settene, med ulike kombinasjoner. Vi tolket dette som at vi måtte bruke ulike kombinasjoner for hver database, og derfor ble det nettleie mot vær her.

Derfor vil også spørringene i denne databasen være ganske simple.

Alternativ database:

Ettersom vi slet så mye med spørringer og feilmeldinger, prøvde vi å sette opp en alternativ database. Denne lignet noe på det vi startet med første dagen vi prøvde oss på Cassandra. Her har vi en tabell for nettleiestatistikk, som da inneholder all dato for nettleiestatistikk fra ulike fylker. Primærnøkkelen her satt vi da til å være: dato og fylkenavn. Så lagde vi en tabell for hver fylke som holdt på værdato innenfor det gitte fylke, disse tabellene har primærnøkkel: StasjonId og tid.

Ut Fra denne kjørte vi følgende spørringer:

```

2 SELECT AVG(nedbor) AS gjennomsnittlig_nedbor
3 FROM milepael3v3.vaerdata_agder
4 WHERE fylke = 'Agder'
5 AND tid >= '2021-01-01' AND tid <= '2022-12-12'
6 ALLOW FILTERING;
7

```

Rows

gjennomsnittlig_nedbor

0.7333333492279053

```

2 SELECT fylke_navn, dato, snitt_omregnetOreEks
3 FROM milepael3v3.nettleiestatistikk
4 WHERE fylke_navn = 'Agder'
5 ALLOW FILTERING;
6

```

Rows

| fylke_navn | dato | snitt_omregnetoreeks |
|------------|---------------------------|----------------------|
| Agder | 2022-01-01 00:00:00 +0000 | 100.99878692626953 |
| Agder | 2021-01-01 00:00:00 +0000 | 27.499670028686523 |

```

1 SELECT tid, middel_middelvind, nedbor
2 FROM milepael3v3.vaerdata_agder
3 WHERE tid >= '2021-01-01' AND tid <= '2022-12-12'
4 ALLOW FILTERING;
5

```

ows

| id | middel_middelvind | nedbor |
|------------|--------------------|---------------------|
| 2021-01-01 | 1.2999999523162842 | 2.200000047683716 |
| 2021-01-02 | 1.100000023841858 | 0.20000000298023224 |
| 2021-01-03 | 1.7999999523162842 | 1.2999999523162842 |
| 2021-01-04 | 1.7999999523162842 | 0.20000000298023224 |
| 2021-01-05 | 1.600000023841858 | 0.4000000059604645 |
| 2021-01-06 | 0.6000000238418579 | 0.10000000149011612 |

Total: 6 results

Refleksjon 2:

Ettersom vi etter første innlevering hadde problemer med å sette opp et databasedesign som kunne kjøre spørringer uten allow filtering, har vi prøvd oss frem med litt andre alternativer.

Vi satte opprinnelig opp et design med en tabell for hvert fylke, med hver sin værdata og nettleiestatistikk. Dette førte til problemer med spørringer på tvers av fylkene, med mindre vi brukte ALLOW FILTERING.

Da prøvde vi et annet mulig design ved å kombinere nettleiestatistikken og værdata i en tabell, men dette designet ble også hindret av spørringer på tvers uten ALLOW FILTERING. Vi vurderte også et design med to separate tabeller, en for værstatistikk og en for nettleie statistikk, og en tabell for å lagre fylkes informasjonen. Men dette designet vil også kreve ALLOW FILTERING.

Vi har derfor sett oss nødt til å bruke tiden på de andre oppgavene slik at vi kan få de ferdige.

Nøkkel-verdi database (Redis)

Design nøkkel-verdidatabasen:

For redisdatabasen har vi valgt å bruke datasettene for magasinfylling og værdata. Hvordan databasen burde designes kommer an på hvordan man er interessert i å hente ut dataene.

Våres case handler om at kommersielle og private aktører skal enkelt kunne hente ut data for praktisk bruk, altså vurdere økonomiske beslutninger i tidlig og sen fortid. Derfor er man gjerne interessert i dataene på følgende måte:

- Man vil ha de nyeste tillagte dataene (f.eks. kapasitet i thw for nåværende uke)
- Man vil gjerne kunne hente data fra nåværende tid og bakover, for så å gjøre operasjoner som f.eks. å summere eller ta gjennomsnittet av de uthentede verdiene.

Man er sjeldent interessert i å hente ut dataene på følgende måte:

- Ha data for en spesifikk uke/dato langt tilbake i tid. Det er lite nyttig for en aktør å vite f.eks. fyllingsgraden på en tilfeldig dag for 2 år siden.

For dataene våre burde altså være raskt og lett å gå igjennom en hel del data, der man starter fra de seneste innlagte dataene.

- Hash egner seg bra til å representere dataene som objekter. F.eks. representere magasinstatistikk for en dag. Men det blir vanskeligere å gå igjennom all dataen fra dag til dag, sortert etter dato. Dataen er relatert til.
- Set garanterer unikheth for verdien. Men det er ingen nødvendighet for vår database at verdiene er unike. Dette kunne ha egnet seg mer hvis den vanlige aktør var ute etter data for kun en spesifikk dag, uavhengig av hvor langt tilbake datoen for dataen er.
- Sorted Set egner seg bra til å sortere dataene våre etter dato. Men det egner seg mindre til å gå igjennom en hel del data dagvis.
- List egner seg bra til å gå igjennom en hel del data dato for dato. Hvis man også passer på å legge inn dataene datoene dataene er for, så får det raskt å hente ut dataene nærmest nåværende dato, og man kan raskt få tak i dato fra datoen nærmest nåværende dato og bakover.

Lister tillater ikke å representere datasettene som objekter. Man kan ikke representere data for f.eks. elspotområde 1 som et objekt. Derfor må man bryte ned objektene datasettene representerer.

For hver kolonne i csv-filene kan man ha en liste. Så kan dataene indekseres i orden basert på dataene verdiene er for.

Datasettet for magasinfylling har følgende data:

dato_id
omrType
omnr
iso_aar
iso_uke
fyllingsgrad
kapasitet_TWh
fylling_TWh
neste_Publiseringsdato
fyllingsgrad_forrige_uke
endring_fyllingsgrad

Datasettet for værdata kan ha følgende data:

Navn
Stasjon
Tid
Middel av middelvind fra hovedobs
Middeltemperatur
Maksimumstemperatur
Minimumstemperatur
Nedbør
Fylke

Man er gjerne interessert i data for det elspotområdet man selv bor i, så dataene for magasindata kan puttes i lister etter elspotområdet de tilhører.

Værdataene er basert på fylke. Hvis man henter ut værdata, så er man alltid interessert i å vite hvordan værdataene er i forhold til magasinfyllingsdataene. Derfor burde også værdataene grupperes etter elspotområde. Man er heller aldri interessert i å vite navnet på stasjonen dataene kom fra, dataene for det kan utelates. Man kan ha en datastruktur som kartlegger fylkene til elspotområdene. Men man legger jo gjerne ikke til dataene direkte med databasevrktøyet. Gjerne bruker man et backend verktøy som man bruker databasen igjennom. Det er derfor mer naturlig å kartlegge fylkene til elspotområdene i backend. En aktør vil ofte ikke være interessert i å vite hvilket elspotområde som hører til hvilket fylke. Man kan godt ha en datastruktur som kartlegger fylker til elspotområder i tilfelle en aktør er interessert i å vite det. Men med tanke på kartlegging for å legge inn dataer i databasen, så er det best med en slik kartlegging i backend.

For å vite hvilken indeks dataene skal puttes inn på kan man f.eks. ha en datastruktur som kartlegger hvilken indeks som er for hvilken dato. Men en mye enklere og raskere løsning er heller å regne på datoene i backend. Så man kan f.eks. se på forskjellen i dager/uker for å vite hvilke spenn av indekser man skal hente data fra. Siden indeksregningen med datoer kan gjøres i backend, så kan dataene for dato/tid utelates fra databasen.

Etter disse vurderingene ender man opp med følgende lister for hvert elspotområde:

```
omrX_fyllingsgrad  
omrX_rkapasitet_TWh  
omrX_fylling_TWh  
omrX_neste_Publiseringsdato  
omrX_fyllingsgrad_forrige_uke  
omrX_endring_fyllingsgrad  
omrX_Middelvind  
omrX_Middeltemperatur  
omrX_Maksimumstemperatur  
omrX_Minimumstemperatur  
omrX_Nedbør
```

Disse listene kan argumenteres for å være unødvendige

```
omrX_neste_Publiseringsdato  
omrX_fyllingsgrad_forrige_uke  
omrX_endring_fyllingsgrad
```

Neste publiseringsdato er alltid en uke forover i tid. Så det er ikke nødvendig å vite hvilken dato dette skjer på. For å holde rede på dette kan man f.eks. heller oprette et variabel i backend som holder på verdien.

Fyllingsgraden forrige uke er også unødvendig å vite. Man kan simpelt gå en indeks bak. Det kan tenkes at det gjør databasen raskere. Men for å kun slippe å gå igjennom en ekstra indeks, så må man holde på mye data. Så det er bedre å utelate denne listen.

Endring i fyllingsgrad representerer endringen i fyllingsgrad mellom en aktuell uke, og uken før. Men her har man det samme tilfellet som med fyllingsgrad_forrige_uke. Det blir mye data å holde opp, umerksomt større hastighet.

Da ender man opp med disse listene i redis:

```
omrX_fyllingsgrad  
omrX_rkapasitet_TWh  
omrX_fylling_TWh  
omrX_neste_Publiseringsdato  
omrX_fyllingsgrad_forrige_uke  
omrX_endring_fyllingsgrad  
omrX_Middelvind  
omrX_Middeltemperatur  
omrX_Maksimumstemperatur  
omrX_Minimumstemperatur  
omrX_Nedbør
```

Innsetting av data inn i databasen

Som demodata valgte vi følgende dataer:

Værdata for område 2:

Hovden - Lundane;SN40880;29.05.2022;2,1;2,4;8,7;-4,5;0;Agder
Hovden - Lundane;SN40880;30.05.2022;2,5;6,1;12,6;-1,1;0,4;Agder
Hovden - Lundane;SN40880;31.05.2022;2,7,6;11,8;3,7;0;Agder
Hovden - Lundane;SN40880;01.06.2022;1,7;8,5;13,5;4,2,5;Agder
Hovden - Lundane;SN40880;02.06.2022;2,3;7,1;10,5;5,9;1,2;Agder
Hovden - Lundane;SN40880;03.06.2022;3,7,6;12,2;5,1;0;Agder
Hovden - Lundane;SN40880;04.06.2022;2,1;8,3;15,3;-2,6;0;Agder
Hovden - Lundane;SN40880;05.06.2022;2,2;9,3;17,1;-2,3;0;Agder
Hovden - Lundane;SN40880;06.06.2022;2,4;11,9;19,4;-0,5;0;Agder
Hovden - Lundane;SN40880;07.06.2022;2,8;10,4;17,1;3;0;Agder
Hovden - Lundane;SN40880;08.06.2022;1,9;9,4;14,4;1,7;3,9;Agder
Hovden - Lundane;SN40880;09.06.2022;2,3;11,7;16,8;6,8;0,1;Agder
Hovden - Lundane;SN40880;10.06.2022;2,7;10,8;15,5;8,7;0,1;Agder
Hovden - Lundane;SN40880;11.06.2022;4,2;9,12,3;5,1;0;Agder
Hovden - Lundane;SN40880;12.06.2022;3,5;7,2;9,4;7,2,1;Agder

Værdata for område 1:

Vestfossen;SN26630;29.05.2022;-;8,9;18;0,6;0;Viken
Vestfossen;SN26630;30.05.2022;-;12,3;18;7,5;0,3;Viken
Vestfossen;SN26630;31.05.2022;-;12,7;17;7,6;0;Viken
Vestfossen;SN26630;01.06.2022;-;12,7;18,6;10,3;0,7;Viken
Vestfossen;SN26630;02.06.2022;-;12,5;17;10,6;14,7;Viken
Vestfossen;SN26630;03.06.2022;-;13,5;19,5;6,6;4,6;Viken
Vestfossen;SN26630;04.06.2022;-;16,4;25,1;6;0,5;Viken
Vestfossen;SN26630;05.06.2022;-;15,7;23,4;5,9;0;Viken
Vestfossen;SN26630;06.06.2022;-;17,3;24,7;6,2;0;Viken
Vestfossen;SN26630;07.06.2022;-;15,1;23,5;11,1;0;Viken
Vestfossen;SN26630;08.06.2022;-;14,3;19,1;11,4;12,9;Viken
Vestfossen;SN26630;09.06.2022;-;14,7;19,1;9,3,1;Viken
Vestfossen;SN26630;10.06.2022;-;17,2;22,7;13,1;0,4;Viken
Vestfossen;SN26630;11.06.2022;-;15,19,9;9,7;0;Viken
Vestfossen;SN26630;12.06.2022;-;15,2;20,1;9,6;8,3;Viken

I demodatabasen er det kun data fra to værstasjoner i hvert sitt fylke.

Et et ekte eksempel kan man tenke seg når dataene kommer inn i backend, så sammenslås alle dataene for hvert elspotområde til ett datasett for hvert elspotområde, ved å ta gjennomsnittet av de endelige verdiene som legges inn i databasen være gjennomsnittet av alle verdiene som kom inn for et elspotområde.

Magasindata for elspotområde 1:

29/05/2022,EL,1,2022,21,0.33317775,5.992993,1.9967318,2022-06-08T13:00:00,0.22742738,0.10575037
05/06/2022,EL,1,2022,22,0.4182513,5.992993,2.506577,2022-06-15T13:00:00,0.33317775,0.08507356
12/06/2022,EL,1,2022,23,0.49069557,5.992993,2.9407349,2022-06-22T13:00:00,0.4182513,0.07244426

Magasindata for elspotområde 2:

29/05/2022,EL,2,2022,21,0.3054547,33.928802,10.363712,2022-06-08T13:00:00,0.25108355,0.05437115
05/06/2022,EL,2,2022,22,0.33530387,33.928802,11.376458,2022-06-15T13:00:00,0.3054547,0.029849172
12/06/2022,EL,2,2022,23,0.3778057,33.928802,12.818495,2022-06-22T13:00:00,0.33530387,0.042501837

Dataene for værdata og magasindata går 3 uker bak i tid fra datoen 12.06.2022.

Dataene legges til med følgende kommandoer:

Værdata for elspotområde 2:

RPUSH omr2_middelvind 3.5 4.2 2.7 2.3 1.9 2.8 2.4 2.2 2.1 3 2.3 1.7 2 2.5 2.1
RPUSH omr2_middeltemperatur 7.2 9 10.8 11.7 9.4 10.4 11.9 9.3 8.3 7.6 7.1 8.5 7.6 6.1 2.4 2.1 2.4
RPUSH omr2_maksimumstemperatur 9.4 12.3 15.5 16.8 14.4 17.1 19.4 17.1 15.3 12.2 10.5 13.5 11.8 12.6 8.7
RPUSH omr2_minimumstemperatur 7 5.1 8.7 6.8 1.7 3 -0.5 -2.3 -2.6 5.1 5.9 4 3.7 -1.1 -4.5
RPUSH omr2_nedbør 2.1 0 0.1 0.1 3.9 0 0 0 0 1.2 2.5 0 0.4 0

Værdata for elspotområde 1:

RPUSH omr1_middelvind null null null null null null null
RPUSH omr1_middeltemperatur 15.7 16.4 13.5 12.5 12.7 12.7 12.7 8.9
RPUSH omr1_maksimumstemperatur 23.4 25.1 19.5 17 18.6 17 17 18
RPUSH omr1_minimumstemperatur 5.9 6 6.6 10.6 10.3 7.6 7.6 0.6
RPUSH omr1_nedbør 0 0.5 4.6 14.7 0.7 0 0 0

magasindata for elspotområde 2:

RPUSH omr2_fyllingsgrad 0.3778057 0.33530387 0.3054547
RPUSH omr2_kapasitet_TWh 33.928802 33.928802 33.928802
RPUSH omr2_fylling_TWh 12.818495 11.376458 10.363712

magasindata for elspotområde 1:

RPUSH omr1_fyllingsgrad 0.4182513 0.33317775 0.49069557
RPUSH omr1_kapasitet_TWh 5.992993 5.992993 5.992993
RPUSH omr1_fylling_TWh 2.506577 1.9967318 2.9407349

Dataene blir lagt til høyre for hverandre.

Så data for den seneste datoen havner på indeks 0.

Konsekvenser av designet for konsistens

I vår database er hvert kolonne i et datasett sin egen liste.

Det betyr at alle verdier for f.eks. fyllingsgrad tilhørende et elspotområde kan være på en node, mens all data for f.eks. kapasitet TWh kan være på en annen node.

Det betyr at hvis en node er nede, så kan det være en del data som ikke er tilgjengelig.

Aktører som bruker vår tjeneste vil være mest interessert i de nyest innlagte dataene, så det er viktig at noder med nyligst innlagte dataer alltid er oppe.

En ting man kan gjøre er å splitte opp listene etterhvert som de blir lengre.

For eksempel så kan listene splittes opp også etter år eller kvartal.

Det å splitte opp listene kvartalvis kan være det enkleste, siden kvartaler er en enhet som gjerne brukes når man skal ha flere verdier.

Dette kan også gjøre databasen mer effektiv hvis man er ute etter spesifikke verdier midt i en liste. Istedenfor å iterere over en hel del verdier fra seneste dato og bakover, så kan man heller gå rett til listen for kvartalet som inneholder datoen.

Dette blir også tatt opp i refleksjonsbiten.

Spørringer

Før kommandoer for å utføre spørringene vises, kan det være greit å tenke ut å måte for å kartlegge datoer til indekser i backend.

Alle verdiene er i lister.

For å aksessere verdiene trenger man å vite hvilke indekser de er på.

Listene er organisert sånn at for hver uke, er det en verdi i magasindatalistene, og for hver dag er det en verdi i værdatalistene.

Tanken er at i backend bruker regner man forskjeller i datoer (ukesvis eller dagvis) til å aksessere elementene i databasen.

Man kan tenke seg en slik utregning for å regne forskjell i dager mellom to datoer:

$\text{dato1} - \text{dato2}$

Der dato1 er den seneste datoen, og dato2 er den tidligste datoen.

For dagen i dag bruker jeg benevnningen: idag, der idag representerer dagen

12.06.2022

Så hvis man skal ha magasindata mellom to datoer bruker man denne beregningen:

Laveste indeks: $(\text{idag} - \text{dato1}) / 7 = \text{int}(\text{resultat})$

Høyeste indeks: $(\text{idag} - \text{dato1} + \text{dato1} - \text{dato2}) / 7 = \text{int}(\text{resultat})$

$\text{int}()$ gjør at indeksene som regnes ut blir heltall.

Hvis man skal ha værdata mellom to datoer bruker man denne beregningen:

Laveste indeks: $\text{idag} - \text{neste_publiseringsdato} + \langle \text{laveste indeks for magasindata} \rangle * 7$

Høyeste indeks: $\text{idag} - \text{neste_publiseringsdato} + \langle \text{høyeste indeks for magasindata} \rangle * 7$

I backend kan man ha et variabel for neste publiseringsdato. Man tar med denne verdien i utregningen for å ikke sikre om at man er innenfor det samme ukesintervallet av datoer som for magasindata.

I redis trenger man ikke å bekymre seg for at man bruker en indeks utenfor lista, fordi da gir redis deg bare alle elementene innenfor lista.

Man får ikke en error.

Vi tenker oss at aktørene av tjenesten vår vil være interessert i følgende data:

- Gjennomsnittlig fyllingsgrad fra i dag til datoen 27.05.2022
- Gjennomsnittlig fylling_TWh delt på gjennomsnittlig nedbør fra i dag til datoen 27.05.2022
- Endring i fyllingsgrad fra i dag til datoen 01.06.2022
- Sum av fyllingsgrad for den alle områdene denne uken

- Endring i fyllingsgrad, endring i fyllingsgrad_THw, endring i kapasitet_THw fra i dag til datoen 01.06.2022
- Fyllingsgrad denne uken, kapasitet i TWh denne uken, fyllingsgrad i TWh denne uken

Gjennomsnittlig fyllingsgrad fra i dag til datoen 27.05.2022

Laveste indeks:

$$(12.06.2022-12.06.2022) / 7 = 0 / 7 = \text{int}(0) = 0$$

Høyeste indeks:

$$((12.06.2022-12.06.2022) + (12.06.2022 - 27.05.2022)) / 7 = (0 + 16) / 7 = \text{int}(2.28)=2$$

LRange omr1_endring_fyllingsgrad 0 2

```
(1) "2.506577"  
(2) "1.9967318"  
(3) "2.9407349"
```

Etter at man har fått tilbake resultatene så kan man beregne gjennomsnittet i backend:

$$\text{<liste av elementer>} / (\text{<høyeste indeks>} + 1)$$

Gjennomsnittlig fylling_TWh delt på gjennomsnittlig nedbør fra i dag til datoen 27.05.2022:

Laveste indeks for magasindata: $(12.06.2022-12.06.2022) / 7 = 0 / 7 = 0$

Høyeste indeks for magasindata: $((12.06.2022-12.06.2022) + (12.06.2022 - 27.05.2022)) / 7 = (0 + 16) / 7 = \text{int}(2.28)=2$

Laveste indeks for værdata: $0*7 = 0$

Høyeste indeks for værdata: $2*7= 14$

LRANGE omr1_nedbør 0 7

```
1) "0"
2) "0.5"
3) "4.6"
4) "14.7"
5) "0.7"
6) "0"
7) "0"
8) "0"
```

LRANGE omr1_fylling_TWh 0 2

```
1) "2.506577"
2) "1.9967318"
3) "2.9407349"
```

Når man har fått tilbake resultatene kan man beregne følgende i backend:

Gjennomsnitt for hver liste omr1_nedbør og omr1_fylling_TWh: <liste av elementer> / (<høyeste indeks> + 1)

Dele gjennomsnittene på hverandre: <gjennomsnitt fylling_TWh> / <gjennomsnitt nedbør>

Endring i fyllingsgrad fra i dag til datoen 01.06.2022

Laveste indeks: $(12.06.2022 - 12.06.2022) / 7 = 0 / 7 = \text{int}(0) = 0$

Høyeste indeks: $((12.06.2022 - 12.06.2022) + (12.06.2022 - 01.06.2022)) / 7 = (0 + 12) / 7$

$= \text{int}(1.71) = 1$

LINDEX omr1_fyllingsgrad 0

LINDEX omr1_fyllingsgrad 1

```
1) "0.4182513"  
2) "0.33317775"
```

Når man har fått tilbake resultatene kan man beregne følgende i backend:
element1 - element2

Her viser databasen seg å være ineffektiv. Man trenger kun to spesifikke verdier, men man må iterere over en hel del verdier som man ikke er interessert i. Dette diskuteres nærmere i refleksjonsdelen.

Sum av fyllingsgrad for den alle områdene denne uken

LINDEX omr1_fyllingsgrad 0

LINDEX omr2_fyllingsgrad 0

```
127.0.0.1:6379> LINDEX omr1_fyllingsgrad 0
"0.4182513"
127.0.0.1:6379> LINDEX omr2_fyllingsgrad 0
"0.3778057"
```

Vår eksempeldatabase i redis inneholder kun data for 2 områder.

Men i en database som inneholder all data for områdene, ville spørringene sett slik ut:

LINDEX omr1_fyllingsgrad 0

LINDEX omr2_fyllingsgrad 0

LINDEX omr3_fyllingsgrad 0

LINDEX omr4_fyllingsgrad 0

LINDEX omr5_fyllingsgrad 0

Så plusser man resultatene sammen:

fyllingsgrad_område1 + fyllingsgrad_område2

I en database med data for alle områdene ville man ha sammenlagt resultatet fra listene for alle områdene.

Endring i fyllingsgrad, endring i fyllingsgrad_THw, endring i kapasitet_THw fra i dag til datoen 01.06.2022

Laveste indeks: $(12.06.2022 - 12.06.2022) / 7 = 0 / 7 = \text{int}(0) = 0$

Høyeste indeks: $((12.06.2022 - 12.06.2022) + (12.06.2022 - 01.06.2022)) / 7 = (0 + 12) / 7 = \text{int}(1.71) = 1$

LINDEX omr1_fyllingsgrad 0

LINDEX omr1_fyllingsgrad 1

LINDEX omr1_fylling_TWh 0

LINDEX omr1_fylling_TWh 1

LINDEX omr1_kapasitet_TWh 0

LINDEX omr1_kapasitet_TWh 1

```
127.0.0.1:6379> LINDEX omr1_fyllingsgrad 0
"0.4182513"
127.0.0.1:6379> LINDEX omr1_fyllingsgrad 1
"0.33317775"
127.0.0.1:6379>
127.0.0.1:6379> LINDEX omr1_fylling_TWh 0
"2.506577"
127.0.0.1:6379> LINDEX omr1_fylling_TWh 1
"1.9967318"
127.0.0.1:6379>
127.0.0.1:6379> LINDEX omr1_kapasitet_TWh 0
"5.992993"
127.0.0.1:6379> LINDEX omr1_kapasitet_TWh 1
"5.992993"
```

Når man har fått tilbake resultatene kan man beregne følgende i backend:

endring i fyllingsgrad = fyllingsgrad_tidligste_dato - fyllingsgrad_seneste_dato

endring i fylling_TWh = fylling_TWh_tidligste_dato - fylling_TWh_seneste_dato

endring i kapasitet_TWh = kapasitet_TWh_tidligste_dato -

kapasitet_TWh_seneste_dato

Fyllingsgrad denne uken, kapasitet i TWh denne uken, fyllingsgrad i TWh denne uken

Fyllingsgrad denne uken:

LINDEX omr1_fyllingsgrad 0

"0.4182513"

Kapasitet_TWh denne uken

LINDEX omr1_kapasitet_TWh 0

"5.992993"

Fyllingsgrad_TWh denne uken

LINDEX omr1_fylling_TWh 0

"2.506577"

Samlet nedbør denne uken, samt endring i fyllingsgrad mellom denne uken og forrige uke:

LRANGE omr1_fyllingsgrad 0 1

LRANGE omr1_nedbør 0 7

```
127.0.0.1:6379> LRANGE omr1_fyllingsgrad 0 1
1) "0.4182513"
2) "0.33317775"
127.0.0.1:6379>
127.0.0.1:6379> LRANGE omr1_nedbør 0 7
1) "0"
2) "0.5"
3) "4.6"
4) "14.7"
5) "0.7"
6) "0"
7) "0"
8) "0"
127.0.0.1:6379>
```

Så i backend kan man gjøre følgende utregning for å få det endelige resultatet:

<fyllingsgrad> = fyllingsgrad_denne_ukene - fyllingsgrad_forrige_uke

<nedbør> = etterspurte_elementer_liste.sum()

Oppdatere databasen

Rutinen vår vil være som følger:

- Man holder rede på neste publiseringsdato.
- For hver dag som går, legger man inn værdatae. Da holder man rede på hvilke værstasjoner som tilhører hvilket elspotområde. Man tar gjennomsnittet av verdiene for værstasjonene som tilhører samme elspotområde, og legger inn den gjennomsnittlige dataen. Hvis en verdi er null, så betyr det at værstasjonen ikke måler verdien. Null verdiene blir ikke tatt med i utregningen.
- Når datoen for neste publiseringsdato innslår, legger man inn de nye magasindataene i de forskjellige listeme.

Hvis man har samme lister på flere noder, så må man oppdatere nodene de forskjellige nodene med de nye verdiene.

Hvis de nye dataene kommer inn på forskjellige noder, så kan man gjøre noe utregning på hver node, for så å samle de utregnede dataene på en node for å legge inn dataene der. Den noden vil da ha de nyeste dataene.

Noe værdata som kommer inn en dag

Vestfossen;SN26630;13.06.2022;-;15,5;21;7,5;0,1;Viken

Kjeller;SN4200;13.06.2022;2;14,9;20,5;8,6;-;Viken

Geithus;SN26500;13.06.2022;-;15,2;22,2;7;0;Viken

Magasindata for område 1 som kommer inn neste uke

19/06/2022,EL,1,2022,24,0.5360184,5.992993,3.2123542,2022-06-29T13:00:00,0.49069557,0.045322806

I demodatabasen bruke vi 12.06.2022 som seneste dato. La oss si at dagene går framover derfra. Over uka vil man ha følgende rutine:

- Når magasindataene for 12.06.2022 kommer inn, lagre verdien for neste publiseringsdato i et variabel. Som diskutert tidligere så kan dette lagres som et variabel i backend. Hvis man lagrer det i redis databasen istedenfor kan det gjøres på følgende måte: HSET publiseringsdatoer omr1 2022-06-22T13:00:00 omr2

Så kan man hente ut publiseringsdatoene slik: HGET publiseringsdatoer omr1

- På hver dag fram til neste publiseringsdato kommer det inn ny værdata. I databasen kan man ha et hash som kartlegger hver værstasjon til hvert elspotområde. Så kan man hente ut verdiene som dette: HMGET værstasjoner vestfossen, der den lagrede væredien kan være 1, som tilsier at vestfossen tilhører elspotområde 1. Så legger man sammen alle verdiene til værdataene for hvert elspotområde, og så lagrer man gjennomsnittet av disse verdiene inn i databasen.

Etter å gjøre ta gjennomsnittet for værdataene over, så ender man opp med dette reultatet:

Middelvind: 2

Middeltemperatur: 13

Maksimumstemperatur 17

Minimumstemperatur 5.6

Nedbør: 0.33

Dataene for denne dagen kan legges inn i databasen på følgende vis:

LPUSH omr1_middelvind 2

LPUSH omr1_middeltemperatur 13

LPUSH omr1_maksimumstemperatur 17

```
LPUSH omr1_minimumstemperatur 5.6
LPUSH omr1_nedbør 0.33
```

Kommandoene legger de nye verdiene til venstre i listen, så de får indeks 0.

```
127.0.0.1:6379> LPUSH omr1_middelvind 2
(integer) 9
127.0.0.1:6379> LPUSH omr1_middeltemperatur 13
(integer) 9
127.0.0.1:6379> LPUSH omr1_maksimumstemperatur 17
(integer) 9
127.0.0.1:6379> LPUSH omr1_minimumstemperatur 5.6
(integer) 9
127.0.0.1:6379> LPUSH omr1_nedbør 0.33
(integer) 9
127.0.0.1:6379> LINDEX omr1_middelvind 0
"2"
127.0.0.1:6379> LINDEX omr1_nedbør 0
"0.33"
127.0.0.1:6379>
```

- Etterhvert kommer man til datoen for neste_publiceringsdato.

Da skal det legges inn nye verdier inn i listene for magasindata.

fyllingsgrad: 0.5360184

kapasitet TWh: 5.992993

fylling TWh: 3.2123542

Dataene kan legges inn med følgende kommando:

```
LPUSH omr1_fyllingsgrad 0.5360184
LPUSH omr1_kapasitet_TWh 5.992993
LPUSH omr1_fylling_TWh 3.2123542
```

Kommandoen legger de nye dataene lengst til venstre i listen. Da får de nye verdiene indeks 0.

```
127.0.0.1:6379> LPUSH omr1_fyllingsgrad 0.5360184
(integer) 4
127.0.0.1:6379> LPUSH omr1_kapasitet_TWh 5.992993
(integer) 4
127.0.0.1:6379> LPUSH omr1_fylling_TWh 3.2123542
(integer) 4
127.0.0.1:6379> LINDEX omr1_fyllingsgrad 0
"0.5360184"
127.0.0.1:6379> LINDEX omr1_kapasitet_TWh 0
"5.992993"
127.0.0.1:6379>
```

Reflekter

Vi føler at det er greit å gjøre spørringer mot databasen slik den nå er designet.

Men med tanke på vedlikehold så kunne man hatt en hash som kartla alle værstasjonene til det elspotområdet de tilhører.

Man kunne også hatt en hash som kartla neste publiseringsdato for hvert elspotområde.

Egentlig så er publiseringsdatoen for alle elspotområdene den samme dagen, så da kunne man like godt hatt en string. Men hvis publiseringsdatoene plutselig hadde vært ulike for en eller annen grunn, så kunne en hash ha håndtert det bedre. Så en hash hadde vært sikrere.

Vi tenker et bedre design hadde vært hvis listene i databasen var splittet opp kvartalsvis.

F.eks. for spørringen for endring i fyllingsgrad fra i dag til datoen 01.06.2022, så trenger kun verdien assosiert med i dag og verdien assosiert med 01.06.2022.

Hvis man har et stort spenn mellom den seneste og tidligste datoen, så er det unødvendig å gå igjennom hele listen.

Dette kunne vært forbedret, f.eks. splittet opp listen mer.

Å splitte opp listen kvartalsvis tenker vi hadde gitt mest mening, ettersom man gjerne er ute etter data kvartalsvis.

Hvis en liste ikke inneholdt verdiene for spennet av datoer man er interessert i, så kunne man ha iterert en liste for det første kvartalet, så startet rett på det andre kvartalet.

Hvis man kun er interessert i to datoer som i endring i fyllingsgrad fra i dag til en annen dato,

så kunne man ha fått datoen i dag fra lista for nåværende kvartal, så kunne man ha gått rett til lista for kvartalet for den andre datoen, så startet fra den enden som er nærmest den ønskede datoen.

Spark it up!

Lasting av .CSV og skrivning til .Parquet:

Vi har skrevet dette programmet for å laste inn våre fire ulike datasett i csv-format og skrive dem til parquet-filer. Rutinen må kjøres for hvert datasett som skal behandles og den fungerer slik:

Først leses csv-filen ved å bruke read-funksjonen. Så brukes .option for å spesifisere at filen har header. Dette resulterer i en dataframe som vil inneholde dataene vi hadde i CSV-filen.

Deretter skrives dataene fra dataframen til en parquet-fil i en egen mappe ved å bruke funksjonen write.parquet()

Vi hadde store problemer med å skrive til parquet underveis. Og som et resultat kan man se at linje to er omtrent identisk med linje en. Eneste forskjell er spesifisering av header.

```
//Datasett 1:
```

```
val df1 = spark.read.format("csv")  
    .option("header", "true")  
    .load("C:/Spark/Datasett/Magasinstatistikk (1).csv")
```

```
val df1 = spark.read.csv("C:/Spark/Datasett/Magasinstatistikk  
(1).csv")
```

```
df1.write.parquet("C:/Spark/Datasett/Parquet/Magasinstatistikk  
(1).parquet")
```

```
//Datasett 2:
```

```
val df2 = spark.read.format("csv")  
    .option("header", "true")  
    .load("C:/Spark/Datasett/Nettleie_Fylkessnitt (1).csv")
```

```
val df2 = spark.read.csv("C:/Spark/Datasett/Nettleie_Fylkessnitt  
(1).csv")
```

```
df2.write.parquet("C:/Spark/Datasett/Parquet/Nettleie_Fylkessnitt
```

```
(1).parquet")
```

```
//Datasett 3:
```

```
val df3 = spark.read.format("csv")  
    .option("header", "true")  
    .load("C:/Spark/Datasett/prices_area.csv")
```

```
val df3 = spark.read.csv("C:/Spark/Datasett/prices_area.csv")
```

```
df3.write.parquet("C:/Spark/Datasett/Parquet/prices_area.parquet")
```

```
//Datasett 4:
```

```
val df4 = spark.read.format("csv")  
    .option("header", "true")  
    .load("C:/Spark/Datasett/vaerstatistikk_med_fylke.csv")
```

```
val df4 =  
spark.read.csv("C:/Spark/Datasett/vaerstatistikk_med_fylke.csv")
```

```
df4.write.parquet("C:/Spark/Datasett/Parquet/vaerstatistikk_med_fylke.parquet")
```

Aggregeringer og dataobjekter

a) Lesing av .PARQUET-filer:

Vi leste inn .Parquet-filene vi genererte i forrige oppgave ved å kjøre denne koden:

```
val df1 = spark.read.parquet("C:/Spark/Datasett/Parquet/Magasinstatistikk  
(1).parquet")  
val df2 = spark.read.parquet("C:/Spark/Datasett/Parquet/Nettleie_Fylkessnitt  
(1).parquet")  
val df3 = spark.read.parquet("C:/Spark/Datasett/Parquet/prices_area.parquet")  
val df4 =  
spark.read.parquet("C:/Spark/Datasett/Parquet/vaerstatistikk_med_fylke.parquet")
```

For hver dataframe vi har laget, bruker vi read-funksjonen og spesifiserer at det er en spesifikk parquet-fil på filstien C:/Spark/Datasett/Parquet som skal leses inn.

b) Aggregeringer:

c) Skrivning av dataobjekter til .CSV:

Plan videre, da vi dessverre har brukt for mye tid på å sette opp spark og har støtt på en del problemer underveis.

1. MongoDB:

- Utfør aggregeringer for MongoDB og lag en CSV-fil.
- Lag en CSV-fil med dataobjekter som skal settes inn i MongoDB, basert på aggregeringene.

2. Neo4j:

- Utfør aggregeringer for Neo4j og lag en CSV-fil.

- Lag en CSV-fil med dataobjekter som skal settes inn i Neo4j, basert på aggregeringene.
- 3. **Cassandra:**
 - Utfør aggregeringer for Cassandra og lag en CSV-fil.
 - Lag en CSV-fil med dataobjekter som skal settes inn i Cassandra, basert på aggregeringene.
- 4. **Kolonnebasert Database:**
 - Utfør aggregeringer for den kolonnebaserte databasen og lag en CSV-fil.
 - Lag en CSV-fil med dataobjekter som skal settes inn i den kolonnebaserte databasen, basert på aggregeringene.
- 5. **Redis:**
 - Utfør aggregeringer for Redis og lag en CSV-fil.
 - Lag en CSV-fil med dataobjekter som skal settes inn i Redis, basert på aggregeringene.

Vi har dessverre ikke fått tiden til å kjøre aggregeringer og skrive dataobjekter til CSV. Vi fikk satt opp en plan på hvordan vi skulle gjøre det, men her er hvordan vi ville gått frem:

Dataobjekter brukt i Cassandra:

Værdata:

- Fylkenavn
- Fylkenr
- Dato
- Stasjons_navn
- Stasjons_id
- Mittel_av_middelvind
- Maksimumstemperatur
- Minimumstemperatur
- nedbør

Fylkesnittnettleie:

- dato
- fylke_navn
- Kvantum_per_fylke
- Snitteffektseks
- Snitteffektsink
- Snitteregieks

- Snittenergiink
- snittfastleddeks
- snittfastleddink
- snittomregneroreeks
- snittomregnetoreink
- tariffgruppe

Kode for å skrive dataobjektene:

```
val selectedColumnsCassandra1 = df4.select("Fylkenavn", "Fylkenr", "Dato",
"Stasjons_navn", "Stasjons_id", "Middel_av_middelvind", "Maksimumstemperatur",
"Minimumstemperatur", "nedbør")
```

```
val selectedColumnsCassandra2 = df2.select("dato", "fylke_navn", "Kvantum_per_fylke",
"Snitteffektseks", "Snitteffektsink", "Snitteregieks", "Snittenergiink", "snittfastleddeks",
"snittfastleddink", "snittomregneroreeks", "snittomregnetoreink", "tariffgruppe")
```

Dataobjekter til CSV:

```
selectedColumnsCassandra1.write.csv("C:/Spark/Datasett/DataObjects/
Cassandra_data1.csv")
```

```
selectedColumnsCassandra2.write.csv("C:/Spark/Datasett/DataObjects/
Cassandra_data2.csv")
```

Dataobjekter brukt i MongoDB:

magasin_data

- Datp
- Omrtype
- Fyllingsgrad
- Kapasitet_TWh
- Fylling_TWh
- Fyllingsgrad_forrige_uke
- Endring_i_fyllingsgrad
- Neste_pub_dato

værdata

- Navn
- Nedbør
- Middelvind
- Middelterperatur
- Maksimumstemperatur
- Minimumstemperatur
- Fylke
- Område_nr

Kode:

```
val selectedColumnsMongoDB1 = df4.select("Navn", "Nedbør", "Middelvind",  
"Middeltemperatur", "Maksimumstemperatur", "Minimumstemperatur", "Fylke", "Område_nr")
```

```
val selectedColumnsMongoDB2 = df1.select("magasin_data", "Datp", "Omrtype",  
"Fyllingsgrad", "Kapasitet_TWh", "Fylling_TWh", "Fyllingsgrad_forrige_uke",  
"Endring_i_fyllingsgrad", "Neste_pub_dato")
```

Dataobjekter til CSV:

```
selectedColumnsMongoDB1.write.csv("C:/Spark/Datasett/DataObjects/  
MongoDB_data1.csv")
```

```
selectedColumnsMongoDB2.write.csv("C:/Spark/Datasett/DataObjects/  
MongoDB_data2.csv")
```

Graf database:

Magasindata:

- Dato
- Omr nr
- Omr type
- Fyllingsgrad
- Kapasitet (TWh)
- Fylling (TWh)
- Neste publiseringsdato

Spotprisdata:

- Omr nr
- Dato
- kl: 00.00
- Kl: 03.00
- Kl: 06.00
- Kl: 09.00
- Kl: 12.00
- Kl: 15.00
- Kl: 18.00
- Kl: 21.00

Kode:

```
val selectedColumnsGrafDB1 = df1.select("Dato", "Omr nr", "Omr type", "Fyllingsgrad",  
"Kapasitet (TWh)", "Fylling (TWh)", "Neste publiseringsdato")
```

```
val selectedColumnsGrafDB2 = df3.select("Omr nr", "Dato", "kl: 00.00", "kl: 03.00", "kl:  
06:00", "kl: 09:00", "kl: 12:00", "kl: 15:00", "kl: 18:00", "kl: 21:00")
```

Objekter til CSV:

```
selectedColumnsGrafDB1.write.csv("C:/Spark/Datasett/DataObjects/GrafDB_data1.csv")
```

```
selectedColumnsGrafDB2.write.csv("C:/Spark/Datasett/DataObjects/GrafDB_data2.csv")
```

Nøkkelverdi database:

Magasindata:

dato_Id
omrType
omrn
iso_aar
iso_uke
fyllingsgrad
kapasitet_TWh
fylling_TWh
neste_Publiseringsdato
fyllingsgrad_forrige_uke
endring_fyllingsgrad

Værdata:

Navn
Stasjon
Tid
Middel av middelvind
Middeltemperatur
Maksimumstemperatur
Minimumstemperatur
Nedbør

Fylke

Kode:

```
val selectedColumnsNokkelverdiDB1 = df1.select("dato_Id", "omrType", "omrnrr", "iso_aar",  
"iso_uke", "fyllingsgrad", "kapasitet_TWh", "fylling_TWh", "neste_Publiseringsdato",  
"fyllingsgrad_forrige_uke", "endring_fyllingsgrad")
```

```
val selectedColumnsNokkelverdiDB2 = df4.select("Navn", "Stasjon", "Tid", "Middel av  
middelvind", "Middeltemperatur", "Maksimumstemperatur", "Minimumstemperatur",  
"Nedbør", "Fylke")
```

Objekter til CSV:

```
selectedColumnsNokkelverdiDB1.write.csv("C:/Spark/Datasett/DataObjects/  
NokkelverdiDB_data1.csv")
```

```
selectedColumnsNokkelverdiDB2.write.csv("C:/Spark/Datasett/DataObjects/  
NokkelverdiDB_data2.csv")
```

Strømanalyse

Problemstilling

Vi ønsker å se hvorfor strømprisene har vært så høye det siste året.

Vi tenker å hente ut data fra en database med spark, så gjøre om dataene så de kan visualiseres i powerBI.

For fremstilling for brukere vil vi bruke diverse grafer for ting som produksjon, forbruk, strømpriser, eksport, import, værforhold, magasinfylling...

Vi har tenkt til å se på om forskjellige hendelser korrelerer, f.eks. hvis strømprisen blir høyere, vil forbruket være høyt i samme periode?

Arkitektur

Arkitektur for løsning: Vi bruker mongodb, scala spark, og powerBI - for å legge inn dataer, hente dem ut, og lage en analyse.

Vi legger inn dataene i mongodb. Så henter vi ut dataene med scala spark, og omformer dataframes slik at dataene blir lette å bruke i powerBI. I powerBI bruker vi dataframene til å skrive csv filer, som vi så importerer til powerBI. I powerBI lager vi diverse grafer for å fremstille funn. Fram til nå har vi brukt datasett for magasinfylling, værforhold, nettleie og strømpriser. For å gjøre analysen har vi funnet flere datasett. Disse datasettene inneholder data for: Strømpriser i andre land, import og eksport, kraftproduksjon og forbruk i forskjellige sektorer.

Dette passer med løsningen over ved at vi ender opp med grafer som kan brukes for å visualisere funn. Vi kan f.eks. ha data for forbruk og strømpriser i mongodb. Så kan vi hente ut dataene i spark, omforme på dem så de er lett å visualisere i powerBI. Til slutt vil man ha visualiseringen av funnene i en graf.

Databasemotor

Vi tenkte originalt å bruke Cassandra som databasemotor. Dette er fordi cassandra er en raskt databasemotor, samtidig som en innebygde funksjonaliteter, og har en datastruktur som verken er for simpel (det blir for vanskelig å representere deler av dataene i databasen), eller for detaljert (det blir plundrete å gjøre spørringer og å håndtere dataene). Men vi slet mye med å få koblet Cassandra med spark. Etterhvert som tiden begynte å gå, så konkluderte vi at vi heller måtte prøve en annen databasemotor.

Vi vurderte redis, men vi konkluderte at redis ville gjøre det svært vanskelig å håndtere dataene. Da var det gjerne at man måtte komme opp med strukturer, for å håndtere andre strukturer, som så igjen håndterte andre strukturer. I redis konkluderte vi at det var spesielt vanskelig å håndtere dato verdier. Siden våre dataer er timeseries, så ville redis gjort håndtering av dataene vanskelig.

I mongodb prøvde vi å lage kolleksjoner for hver instans i datasettene, etter å ha omformet datasettene i mongodb til hvordan vi tenkte de best ville passe med spørringer og bruk av databasen. For å gjøre dette prøvde vi å bruke insertOne, men det gikk fryktelig tregt. Vi måtte istedenfor bruke insertMany. Men da kunne vi ikke ha en kolleksjon for hver instans. Da måtte vi istedenfor ha hver instans som en subkolleksjon. Det ble derfor mye data i hver kolleksjon.

Noe vi kunne ha gjort er å ha delt opp dataene kvartalsvis, men vi fikk etterhvert lite med tid. Men med tanke på konsistens hadde det vært mer optimalt å ha delt opp kolleksjonene etter instanser eller kvartaler.

I mongodb brukte vi følgende kode for å legge inn dataene:

Koden refererer til forskjellige csv-filer, omformer dataene, for så å putte dem inn i en kolleksjon.

```
...

use BD_analyse_rapport_db //spesifiser databasen <-----

path_to_folder = "..." + "/" // <-- spesifiser path til folderet med alle dataene

// ===== Day ahead NO (Norske strømpriser) =====

const fs = require('fs');

var csvFilePath = path_to_folder + "prices area/prices_area.csv";

//referer til fil
var csvData = fs.readFileSync(csvFilePath, 'utf8');

// Splitter fil i rader
var csvRows = csvData.split("\n");

// Henter ut header for kolonnenavn
var header = csvRows[0].split(",");
header = header.map(function (field) {
  return field.trim();
});
```

```

var documents = [];

// Itererer over resten av radene, og legger dem inn i listen over dokumenter
for (var i = 1; i < csvRows.length; i++) {
  var row = csvRows[i].split(",");
  var document = {};

  //Gjør om dato verdi i datasett til en datetime verdi
  for (var j = 0; j < header.length; j++) {
    if (header[j].trim() === 'Date') {
      document['DateTime'] = new Date(`${row[j].trim()}T00:00:00Z`);
    }
    document[header[j]] = row[j].trim();
  }

  documents.push(document);
}

// Bruker insertMany for å legge inn alle dokumentene
db.getCollection('day_ahead_NO').insertMany(documents);

console.log("Insertion completed");

// ===== SE =====
// Svenske strømpriser
// Nesten akkurat den samme prosessen som over

var csvFilePath = path_to_folder + "SE/day_ahead_SE.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');

var csvRows = csvData.split("\n");

var header = csvRows[0].split(",").map(field => field.trim());

var documents = [];

for (var i = 1; i < csvRows.length; i++) {
  //console.log(csvRows.length)
  var row = csvRows[i].split(",");
  var document = {};

  header.forEach((field, index) => {
    if (field === 'Date') {
      document['DateTime'] = new Date(`${row[index].trim()}T00:00:00Z`);
    } else if (field === 'Day-ahead Price [EUR/MWh]') {
      document[field] = parseFloat(row[index].trim());
    } else {
      document[field] = row[index].trim();
    }
  });

  documents.push(document);
}

db.getCollection('day_ahead_SE').insertMany(documents);

// ===== DE =====

```

```

// Tyske strømpriser
// Nesten akkurat den samme prosessen som for Svenske og Norske strømpriser

var csvFilePath = path_to_folder + "DE/day_ahead_DE_LU.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');
// Split the CSV data into rows
var csvRows = csvData.split("\n");

// Get the header row to use as field names
var header = csvRows[0].split(",").map(field => field.trim());

// Array to store documents
var documents = [];

// Iterate through the remaining rows (starting from 1)
for (var i = 1; i < csvRows.length; i++) {
  //console.log(csvRows.length)
  var row = csvRows[i].split(",");
  var document = {};

  // Map CSV columns to document fields
  header.forEach((field, index) => {
    if (field === 'Date') {
      document['DateTime'] = new Date(`${row[index].trim()}T00:00:00Z`);
    } else if (field === 'Day-ahead Price [EUR/MWh]') {
      document[field] = parseFloat(row[index].trim());
    } else {
      document[field] = row[index].trim();
    }
  });

  documents.push(document);
}

db.getCollection('day_ahead_DE_LU').insertMany(documents);

// ===== værddata =====

const fs = require('fs');

var csvFilePath = path_to_folder + "værdstatistikk/værddata.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');

var csvRows = csvData.split("\n");

var header = csvRows[0].split(",").map(field => field.trim());

var documents = [];

for (var i = 1; i < csvRows.length; i++) {

  var row = csvRows[i].split(",");
  var document = {};

  // Map CSV columns to document fields
  header.forEach((field, index) => {

```



```

        if (field === 'Tid') {
            document['DateTime'] = new Date(row[index].trim());
        } else if (field === 'Vind' || field === 'Nedbør') {
            document[field] = parseFloat(row[index].trim());
        } else {
            document[field] = row[index].trim();
        }
    });

    documents.push(document);
}

db.getCollection('værddata').insertMany(documents);

// ===== import export =====

// Datasett for import og eksport av strøm

const fs = require('fs');

var csvFilePath = path_to_folder + "import eksport/import_eksport.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');

var csvRows = csvData.split("\n");

var header = csvRows[0].split(",");
header = header.map(function (field) {
    return field.trim();
});

var documents = [];

for (var i = 1; i < csvRows.length; i++) {
    var row = csvRows[i].split(",");
    var document = {};

    for (var j = 0; j < header.length; j++) {

        if (header[j].trim() === 'Date') {
            document['DateTime'] = new Date(`${row[j].trim()}T00:00:00Z`);
        }

        document[header[j]] = row[j].trim();
    }

    documents.push(document);
}

db.getCollection('import_eksport').insertMany(documents);

console.log("Insertion completed");

// ===== forbruk og produksjon =====
// Forbruk og produksjon av kraft i Norge

const fs = require('fs');
```

```

var csvFilePath = path_to_folder + "forbruk produksjon/forbruk_produksjon.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');

var csvRows = csvData.split("\n");

var header = csvRows[0].split(",");
header = header.map(function (field) {
    return field.trim();
});

var documents = [];

for (var i = 1; i < csvRows.length; i++) {
    var row = csvRows[i].split(",");
    var document = {};

    for (var j = 0; j < header.length; j++) {
        if (header[j].trim() === 'Date') {
            document['DateTime'] = new Date(`${row[j].trim()}T00:00:00Z`);
        }
        document[header[j]] = row[j].trim();
    }

    documents.push(document);
}

db.getCollection('forbruk_produksjon').insertMany(documents);

console.log("Insertion completed");

// ===== forbruk produksjon i forskjellige markeder =====

const fs = require('fs');

var csvFilePath = path_to_folder + "forbruk og produksjon i forskjellige markeder/"
// refererer til en filsti

//Henter ut flere filer fra filstien. Hver fil vil inneholde data for
//kraft produksjon og forbruk i en sektor
var files = fs.readdirSync(csvFilePath);
files.forEach(function (file) {

    var fullPath = path.join(csvFilePath, file);

    var csvData = fs.readFileSync(fullPath, 'utf8');

    var csvRows = csvData.split("\n");

    var header = csvRows[0].split(",");
    header = header.map(function (field) {
        return field.trim();
    });

    var documents = [];

    for (var i = 1; i < csvRows.length; i++) {
        var row = csvRows[i].split(",");

```

```

var document = {};

var year = parseInt(row[1]);
var month = parseInt(row[2]);

        //Datasettet er splittet opp månedsvis.
        //bruker en kolonne for år og måned, til å lage en datetime verdi
    if (!isNaN(year) && !isNaN(month)) {
        // Directly create 'DateTime' field with day set to 01
        document['DateTime'] = new Date(Date.UTC(year, month - 1, 1));

        document[header[0]] = row[0].trim();

        documents.push(document);
    }
}

var collectionName = file.replace(/ /g, '_').replace(/.csv/g, "");

db.getCollection(collectionName).insertMany(documents);

})

// ===== Magasinstatistikk =====

var csvFilePath = path_to_folder + "Magasinstatistikk.csv";

var csvData = fs.readFileSync(csvFilePath, 'utf8');

var csvRows = csvData.split("\n");

var header = csvRows[0].split(",").map(field => field.trim());

var documents = [];

for (var i = 1; i < csvRows.length; i++) {

    var row = csvRows[i].split(",");
    var document = {};

    header.forEach((field, index) => {
        if (field === 'date') {
            document['DateTime'] = new Date(row[index]);
        } else if (field === 'area') {
            document[field] = row[index].trim();
        } else {
            document[field] = parseFloat(row[index].trim());
        }
    });

    documents.push(document);
}

db.getCollection('magasindata').insertMany(documents);

```

Scala spark og powerBI

I scala spark importerte vi kolleksjoner i mongodb som dataframes. Så gjorde vi diverse endringer på dataframene for at dataene skulle bli lette å bruke i powerBI. Vi måtte blant annet endre på tidskolonner, gruppere data etter tid, og kombinere dataframes...

Først er det greit å se på strømprisene over tid.

Denne koden henter ut data fra mongodb, for strømpriser i Norge. Sluttresultatet er en dataframe som inneholder strømpriser elspotvis, og for hele landet, dagvis.

```
// ===== IMPORT =====

// importerer nødvendige ting
import org.apache.spark.sql.Session
import org.apache.spark.sql.functions.{year, month, col, avg, make_date, regexp_replace}
import org.apache.spark.sql.types.DecimalType
import org.apache.spark.sql.functions._

// ===== SAMLER DAY AHEAD PRISER I NORGE =====

// henter data fra mongodb for strømpriser i norge og innenfor forskjellige elspotområder
val day_ahead_NO = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "day_ahead_NO").load()

// dropper unødvendige kolonner i dataframen
val day_ahead_NO_dagvis = day_ahead_NO.drop("_id", "DateTime")

// Lager dataframe for snittpris over hele landet
val day_ahead_without_area = day_ahead_NO_dagvis.drop("Area")

val day_ahead_NO_all = day_ahead_without_area.groupBy("Date").agg(
  avg("Day-ahead Price [EUR/MWh]").alias("avg_day_ahead_for_all"),
)

// Splitter opp dataframe etter område
val uniqueArea = day_ahead_NO_dagvis.select("Area").distinct().as[String].collect()

val dataFramesArea = uniqueArea.map { area =>
  val DF = day_ahead_NO_dagvis.filter(col("Area") === area)
  val areaDF = DF.withColumnRenamed("Day-ahead Price [EUR/MWh]", "Day ahead " + area)
  val areaDF_without_Area = areaDF.drop("Area")
  areaDF_without_Area
}

// Grupperer dataframene
def joinTwoDataFrames(df1: DataFrame, df2: DataFrame): DataFrame = {
  df1.join(df2, Seq("Date"), "inner")
}

val joinedDataFrameAreas = dataFramesArea.reduce(joinTwoDataFrames)

joinedDataFrameAreas.show()

// Grupper dataframene med dataframe for snittpris over hele landet
val joinedDataFrame = joinedDataFrameAreas.join(day_ahead_NO_all, Seq("Date"), "inner")
```

```
// ===== SKRIVER TIL CSV FIL =====
```

```
val basePath = "C:/Users/Greencom/Desktop/BD_analyse"
```

```
val filePath = s"$basePath/day_ahead.csv"
```

```
joinedDataFrame.write.option("header", "true").csv(filePath)
```

```
...
```

Den endelige dataframen ser slik ut:

| Date | Day ahead N02 | Day ahead N04 | Day ahead N03 | Day ahead N01 | Day ahead N05 | avg_day_ahead_for_all |
|------------|---------------|---------------|---------------|---------------|---------------|-----------------------|
| 2020-02-26 | 10.23 | 17.86 | 17.86 | 10.3 | 10.3 | 13.309999999999999 |
| 2020-04-13 | 2.44 | 2.31 | 2.31 | 2.44 | 2.44 | 2.388 |
| 2021-11-03 | 97.07 | 16.56 | 19.77 | 97.07 | 97.07 | 65.508 |
| 2022-10-05 | 50.49 | 23.49 | 29.59 | 50.49 | 50.49 | 40.910000000000004 |
| 2023-01-21 | 140.17 | 47.19 | 72.74 | 140.17 | 140.17 | 108.088 |
| 2023-05-01 | 90.53 | 46.42 | 57.59 | 90.53 | 93.79 | 75.772 |
| 2023-05-18 | 75.36 | 7.16 | 7.16 | 73.16 | 73.16 | 47.199999999999996 |
| 2020-06-24 | 1.48 | 1.61 | 1.61 | 1.48 | 1.48 | 1.532 |
| 2021-12-23 | 244.72 | 39.08 | 39.08 | 244.72 | 244.72 | 162.464 |
| 2022-10-07 | 27.84 | 7.76 | 7.76 | 27.42 | 27.42 | 19.64 |
| 2023-04-17 | 110.1 | 36.32 | 88.55 | 110.1 | 110.1 | 91.03399999999999 |
| 2023-04-21 | 93.02 | 35.35 | 56.14 | 93.02 | 93.03 | 74.11200000000001 |
| 2023-04-28 | 103.49 | 45.47 | 97.23 | 103.49 | 103.49 | 90.63399999999999 |
| 2020-06-08 | 1.78 | 5.51 | 5.51 | 1.78 | 1.78 | 3.272 |
| 2020-09-12 | 11.76 | 9.22 | 10.93 | 11.76 | 11.76 | 11.086 |
| 2020-11-12 | 6.3 | 5.74 | 5.74 | 6.3 | 6.3 | 6.076 |
| 2021-04-06 | 32.92 | 18.87 | 18.87 | 36.64 | 36.42 | 28.744000000000007 |
| 2022-05-17 | 179.47 | 12.06 | 12.17 | 179.47 | 179.47 | 112.52799999999999 |
| 2023-02-10 | 89.33 | 21.78 | 26.66 | 89.33 | 89.33 | 63.286 |
| 2020-01-05 | 30.7 | 28.69 | 29.65 | 30.7 | 30.7 | 30.088 |

Strømpris over tid

● Gjennomsnitt ● NO1 ● NO2 ● NO3 ● NO4 ● NO5

| Tid | Gjennomsnitt | NO1 | NO2 | NO3 | NO4 | NO5 |
|----------|--------------|------|-------|------|------|------|
| jan 2020 | 0.7K | 0.1K | 0.1K | 0.1K | 0.1K | 0.1K |
| jan 2021 | 1.4K | 1.3K | 1.3K | 1.3K | 1.3K | 1.3K |
| jan 2022 | 4.0K | 5.4K | 13.7K | 6.0K | 6.0K | 6.0K |
| jan 2023 | 2.9K | 8.0K | 8.0K | 5.4K | 5.4K | 5.4K |
| jul 2023 | 0.8K | 0.8K | 0.8K | 0.8K | 0.8K | 0.8K |

De høyeste prisene har vært i NO2, som dekker sørlandet.

Man kan se at fra desember 2020 begynte strømprisene å stige, og ved slutten av 2021 begynte strømprisene å øke kraftig.

Denne koden henter ut data fra mongodb, og bruker dataene til å lage en dataframe som inneholder kraftproduksjon for vind-, vann- og varmekraft.

```
...

// ===== HENTER DATAFRAMES =====

// henter data fra mongodb for kraftproduksjon i forskjellige markeder

val vind = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "Vindkraft").load().drop("_id")

val vann = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "Vannkraft").load().drop("_id")

val varme = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "Varmekraft").load().drop("_id")

val day_ahead_NO = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "day_ahead_NO").load().drop("_id", "Date")

// ===== SAMLER ETTER DAGVIS =====

// Gjør om på datokolonner

val vind_date = vind.withColumn("DateTime", date_format(col("DateTime"), "yyyy-MM-dd"))

val vann_date = vann.withColumn("DateTime", date_format(col("DateTime"), "yyyy-MM-dd"))

val varme_date = varme.withColumn("DateTime", date_format(col("DateTime"), "yyyy-MM-dd"))

val day_ahead_NO_date = day_ahead_NO.withColumn("DateTime", date_format(col("DateTime"), "yyyy-MM-
dd"))

val day_ahead_without_area = day_ahead_NO_date.drop("Area")

val day_ahead_NO_all = day_ahead_without_area.groupBy("DateTime").agg(
  avg("Day-ahead Price [EUR/MWh]").alias("day_ahead"),
)

val vind_date_all = vind_date.groupBy("DateTime").agg(
  avg("Elektrisk kraft").alias("kraft vind")
)

val vann_date_all = vann_date.groupBy("DateTime").agg(
  avg("Elektrisk kraft").alias("kraft vann")
)
```

```
)
```

```
val varme_date_all = varme_date.groupBy("DateTime").agg(
```

```
    avg("Elektrisk kraft").alias("kraft varme")
```

```
)
```

```
// ===== SLÅR SAMMEN DATAFRAMENE =====
```

```
val vind_pris = day_ahead_NO_all.join(vind_date_all, Seq("DateTime"), "inner")
```

```
val vann_varme = vann_date_all.join(varme_date_all, Seq("DateTime"), "inner")
```

```
val vind_pris_vann_varme = vind_pris.join(vann_varme, Seq("DateTime"), "inner")
```

```
// ===== EKSPORTERER DATAFRAME =====
```

```
val basePath = "C:/Users/Greencom/Desktop/BD_analyse"
```

```
val filePath = s"$basePath/kraft_produksjon_markeder.csv"
```

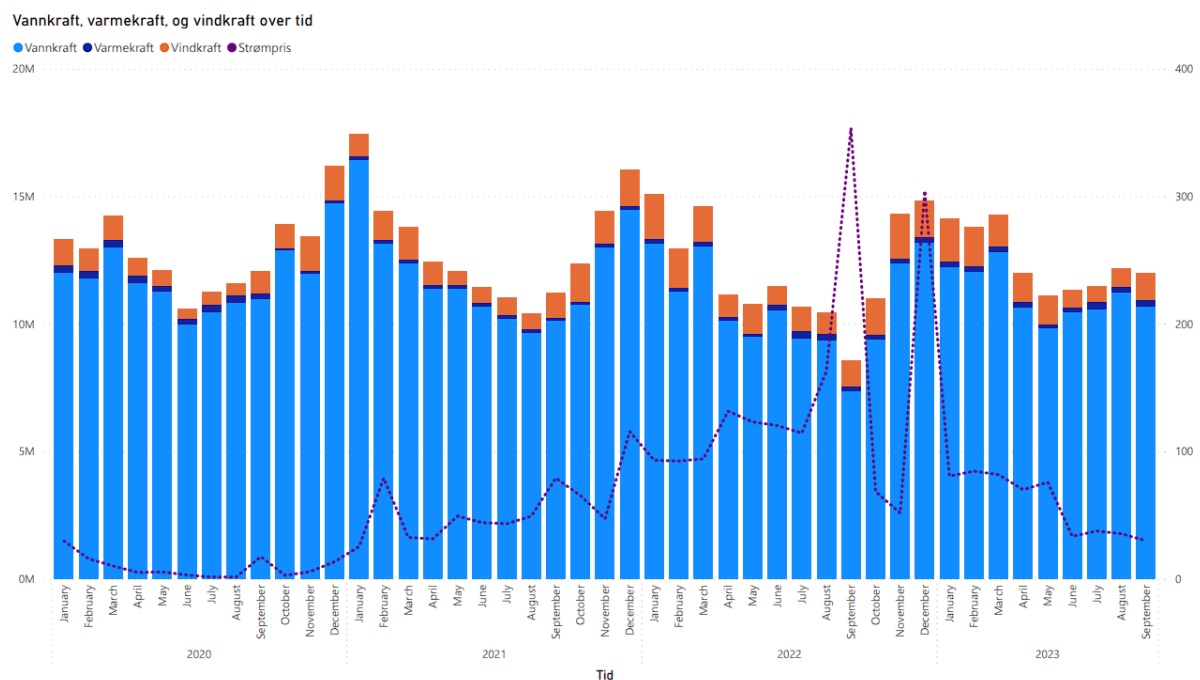
```
vind_pris_vann_varme.write.option("header", "true").csv(filePath)
```

```
...
```


Den endelige dataframen ser slik ut:

| DateTime | day_ahead | kraft vind | kraft vann | kraft varme |
|------------|--------------------|------------|-------------|-------------|
| 2020-01-01 | 29.660000000000004 | 1024285.0 | 1.2018606E7 | 293536.0 |
| 2020-02-01 | 15.803999999999998 | 853722.0 | 1.18063E7 | 273631.0 |
| 2020-03-01 | 10.094 | 942781.0 | 1.3010771E7 | 282876.0 |
| 2020-04-01 | 5.018 | 686210.0 | 1.1616737E7 | 285703.0 |
| 2020-05-01 | 5.4620000000000001 | 600285.0 | 1.1282714E7 | 224906.0 |
| 2020-06-01 | 2.994 | 372085.0 | 9995953.0 | 204024.0 |
| 2020-07-01 | 1.4220000000000002 | 506181.0 | 1.0451614E7 | 289328.0 |
| 2020-08-01 | 1.436 | 445913.0 | 1.0838428E7 | 282888.0 |
| 2020-09-01 | 17.387999999999998 | 858422.0 | 1.0993E7 | 197008.0 |
| 2020-10-01 | 2.706 | 922942.0 | 1.2907275E7 | 83002.0 |
| 2020-11-01 | 5.6 | 1338367.0 | 1.1950499E7 | 142425.0 |
| 2020-12-01 | 13.392 | 1359399.0 | 1.472094E7 | 138460.0 |
| 2021-01-01 | 25.324 | 872583.0 | 1.644846E7 | 130855.0 |
| 2021-02-01 | 78.972000000000001 | 1120002.0 | 1.3172279E7 | 119301.0 |
| 2021-03-01 | 32.465999999999994 | 1271373.0 | 1.2382901E7 | 135441.0 |
| 2021-04-01 | 31.314 | 916171.0 | 1.1375454E7 | 135515.0 |
| 2021-05-01 | 49.236000000000004 | 532029.0 | 1.139609E7 | 151766.0 |
| 2021-06-01 | 44.106000000000001 | 613563.0 | 1.0671065E7 | 141840.0 |
| 2021-07-01 | 43.198 | 691798.0 | 1.021155E7 | 147494.0 |
| 2021-08-01 | 49.162 | 600008.0 | 9657224.0 | 141366.0 |

Denne dataen kan brukes til å lage følgende graf:

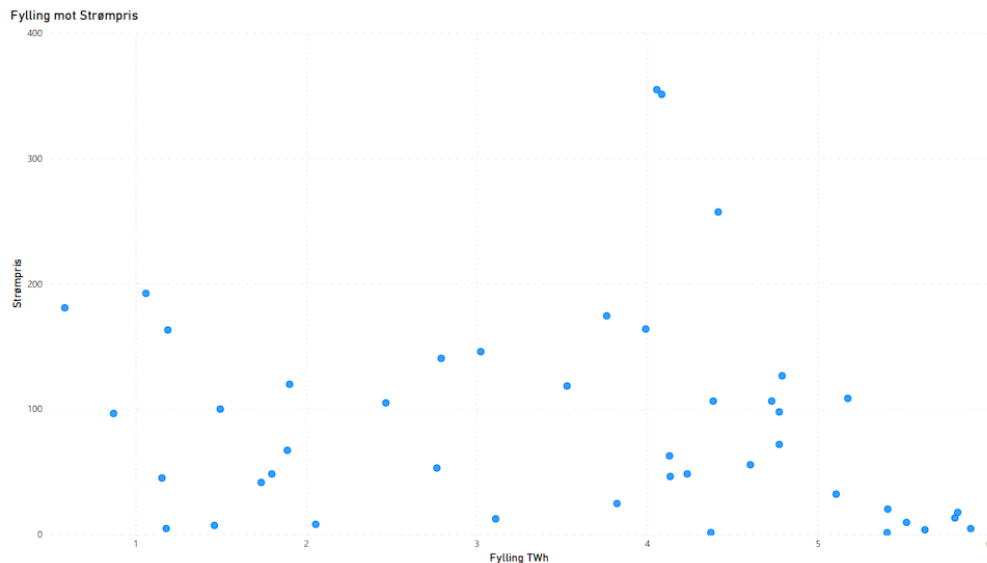


Vi ser at den aller største andelen av kraftproduksjon i Norge består av vannkraft.

Vi ser også noen interessante ting. Akkurat da strømprisen er høyest i september 2022, så er det et dypp i kraftproduksjonen. Dette kan tyde på at produksjonen av vannkraft har noe å si om hvorfor strømprisene er høye.

De samme dataene kan brukes til å lage et scatter plot.

Hvis det er en korrelasjon mellom vannkraftproduksjon og strømpris, så vil punktene på plottet være diagonale. Det er ikke lett å se en korrelasjon her.



Vi kan nå se på vannkraftproduksjon, og strømprisen.

Nå ønsker vi å hente ut data for strømpriser, nedbør, og magasinfylld i de forskjellige elspotområdene.

Denne koden henter ut data fra mongoddb, og lager en dataframe som inneholder data for nedbør og magasinfylld månedsvis.

```
...
// ===== SAMLER MAGASINDATA MÅNEDSVIS
=====

// hent ut dataframe for magasinfylld
val magasindata = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db ").option("collection", "magasindata").load()

// dropper kolloner man ikke trenger
val magasindata_fylling = magasindata.drop("_id", "fyllingsgrad_forrige_uke", "kapasitet_TWh",
"endring_fyllingsgrad")

// får tak i år og måned
val magasin_fylling_month_year = magasindata_fylling.withColumn("year",
year(col("DateTime"))).withColumn("month", month(col("DateTime")))

// Grupperer etter år, måned og område
val magasin_fylling_month_year_avg = magasin_fylling_month_year.groupBy("area", "year", "month").agg(
  make_date(col("year"), col("month"), lit(1)).alias("date"),
  avg("fylling_TWh").alias("avg_fylling_TWh"),
  avg("fyllingsgrad").alias("avg_fyllingsgrad")
)

// dropper måned og år kolonnene
```

```

val magasin_fylling_avg = magasin_fylling_month_year_avg.drop("month", "year")

// ===== SAMLER NEDBØR MÅNEDSVIS =====

// hent ut dataframe for værdata
val værdata = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "værdata").load()

// dropper _id og vind
val nedbør_dagvis = værdata.drop("_id", "Vind")

// får tak i år og måned
val nedbør_dagvis_month_year = nedbør_dagvis.withColumn("year",
year(col("DateTime"))).withColumn("month", month(col("DateTime")))

// Grupperer etter år, måned og område
val nedbør_dagvis_month_year_avg = nedbør_dagvis_month_year.groupBy("Elspot", "year", "month").agg(
  make_date(col("year"), col("month"), lit(1)).alias("date"),
  avg("Nedbør").alias("avg_nedbør"),
)

// dropper måned og år kolonnene
val nedbør_avg = nedbør_dagvis_month_year_avg.drop("month", "year")

// ===== SAMLER DAY AHEAD PRISER I NORGE MÅNEDSVIS =====

//henter ut data for strømpriser
val day_ahead_NO = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "day_ahead_NO").load()

// dropper _id og currency
val day_ahead_NO_dagvis = day_ahead_NO.drop("_id")

// får tak i år og måned
val day_ahead_NO_month_year = day_ahead_NO_dagvis.withColumn("year",
year(col("DateTime"))).withColumn("month", month(col("DateTime")))

// Grupperer etter år, måned og område
val day_ahead_NO_month_year_avg = day_ahead_NO_month_year.groupBy("Area", "year", "month").agg(
  make_date(col("year"), col("month"), lit(1)).alias("date"),
  avg("Day-ahead Price [EUR/MWh]").alias("avg_day_ahead"),
)

// dropper måned og år kolonnene
val day_ahead_NO_avg = day_ahead_NO_month_year_avg.drop("month", "year")

// ===== SAMLER DATAFRAMES =====

// Gjør om navnet på kolonnen Elspot til area
val nedbørWithArea = nedbør_avg.withColumnRenamed("Elspot", "area")
val day_ahead_NO_area = day_ahead_NO_avg.withColumnRenamed("Area", "area")

// joiner de to dataframene etter area og date
val joined_magasin_nedbør = magasin_fylling_avg.join(nedbørWithArea, Seq("area", "date"), "inner")
val joined_magasin_nedbør_dayAhead = joined_magasin_nedbør.join(day_ahead_NO_area, Seq("area",
"date"), "inner")

```

```

// Select the desired columns for the new DataFrame
val combinedDF = joined_magasin_nedbør_dayAhead.select("area", "date", "avg_fylling_TWh", "avg_nedbør",
"avg_day_ahead")

// ===== SPLITTER DFs ETTER OMRÅDE =====

// Lager dataframes for hvert elspotområde for magasinfylling
val combinedDFunique = combinedDF.select("area").distinct().as[String].collect()

val dataFrames = combinedDFunique.map { area =>
  val areaDF = combinedDF.filter(col("area") === area)
  areaDF
}

// ===== Eksporterer dataframes-ene til csv filer =====

val basePath = "C:/Users/Greencom/Desktop/BD_analyse"

dataFrames.foreach { areaDF =>
  val areaValue = areaDF.select("area").head().getString(0)
  val filePath = s"$basePath/nedbør_fylling_day_ahead$areaValue.csv"

  // Write DataFrame to CSV with header
  areaDF.write.option("header", "true").csv(filePath)
}

...

```

Med koden over ender man opp med dataframes for hvert elspotområde.

Her er et eksempel på dataframe for elspotområde 2.

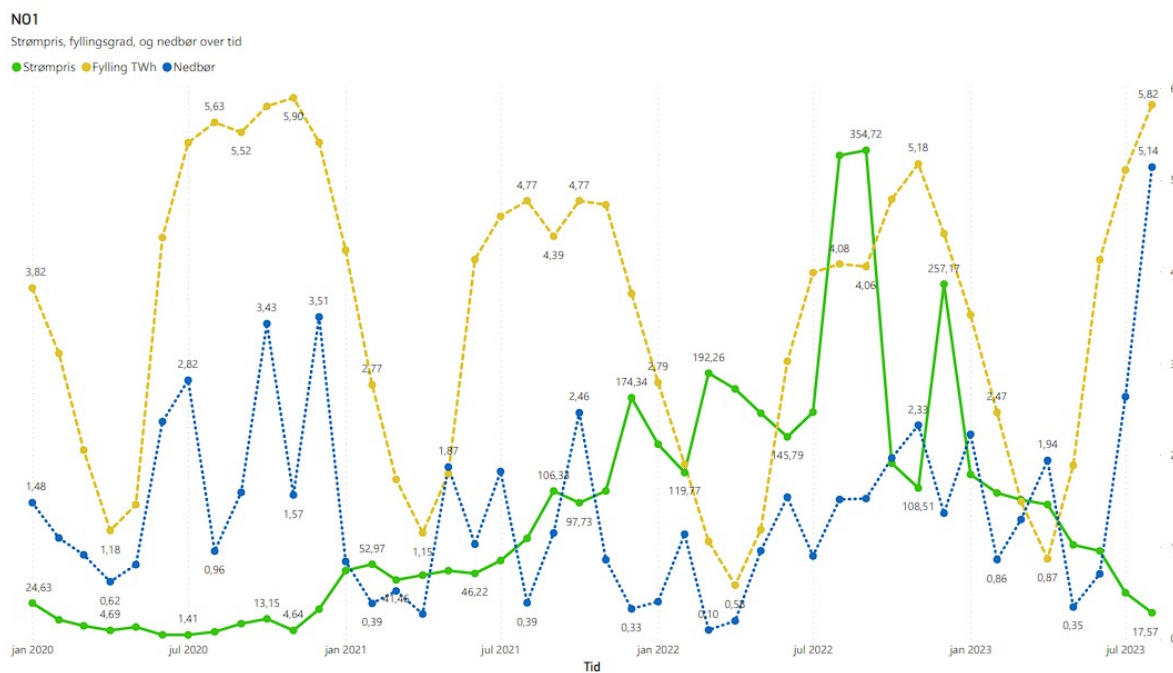
| area | date | avg_fylling_TWh | avg_nedbør | avg_day_ahead |
|------|------------|--------------------|--------------------|--------------------|
| N02 | 2022-03-01 | 10.06697525 | 0.4035483870967741 | 192.26354838709682 |
| N02 | 2023-06-01 | 19.430701 | 1.0639999999999998 | 82.44999999999999 |
| N02 | 2023-01-01 | 20.519049799999998 | 5.103870967741934 | 118.37838709677418 |
| N02 | 2023-07-01 | 23.900714 | 4.118709677419354 | 60.26903225806453 |
| N02 | 2020-03-01 | 20.2583598 | 2.915806451612904 | 8.00967741935484 |
| N02 | 2022-01-01 | 15.6965277 | 1.987741935483871 | 140.4470967741935 |
| N02 | 2020-12-01 | 31.47927425 | 5.528709677419354 | 20.182580645161288 |
| N02 | 2022-04-01 | 7.061824849999999 | 0.5293333333333332 | 180.64633333333334 |
| N02 | 2022-02-01 | 13.540103 | 3.1446428571428577 | 119.7725 |
| N02 | 2020-05-01 | 15.4279079 | 1.2390322580645161 | 7.164516129032257 |
| N02 | 2023-08-01 | 25.568225333333334 | 3.7006896551724138 | 63.68032258064516 |
| N02 | 2021-01-01 | 27.754643999999995 | 1.9319354838709677 | 48.140645161290315 |
| N02 | 2022-05-01 | 7.838608600000001 | 1.694193548387097 | 163.40096774193543 |
| N02 | 2020-10-01 | 32.25167175 | 4.881290322580644 | 13.151290322580644 |
| N02 | 2022-10-01 | 20.797386 | 4.243548387096774 | 126.89935483870966 |
| N02 | 2020-06-01 | 23.49215875 | 2.9003333333333333 | 1.4553333333333333 |
| N02 | 2021-05-01 | 16.9806132 | 1.8974193548387097 | 48.29354838709677 |
| N02 | 2020-01-01 | 24.42596 | 4.572903225806451 | 24.628387096774194 |
| N02 | 2021-09-01 | 18.48521125 | 1.5623333333333334 | 106.33033333333333 |
| N02 | 2020-07-01 | 29.85195775 | 4.107741935483871 | 1.4129032258064516 |

Disse dataene kan brukes til å lage følgende grafer:

Grafen viser strømprisen, fylling TWh og nedbør over tid.

Vi kan se at mengden nedbør følger fylling i TWh.

I midten av 2022 er strømprisen høyest. På det tidspunktet er det ikke spesielt lav fylling, men det er veldig lav fylling og nedbør i perioden rett før.



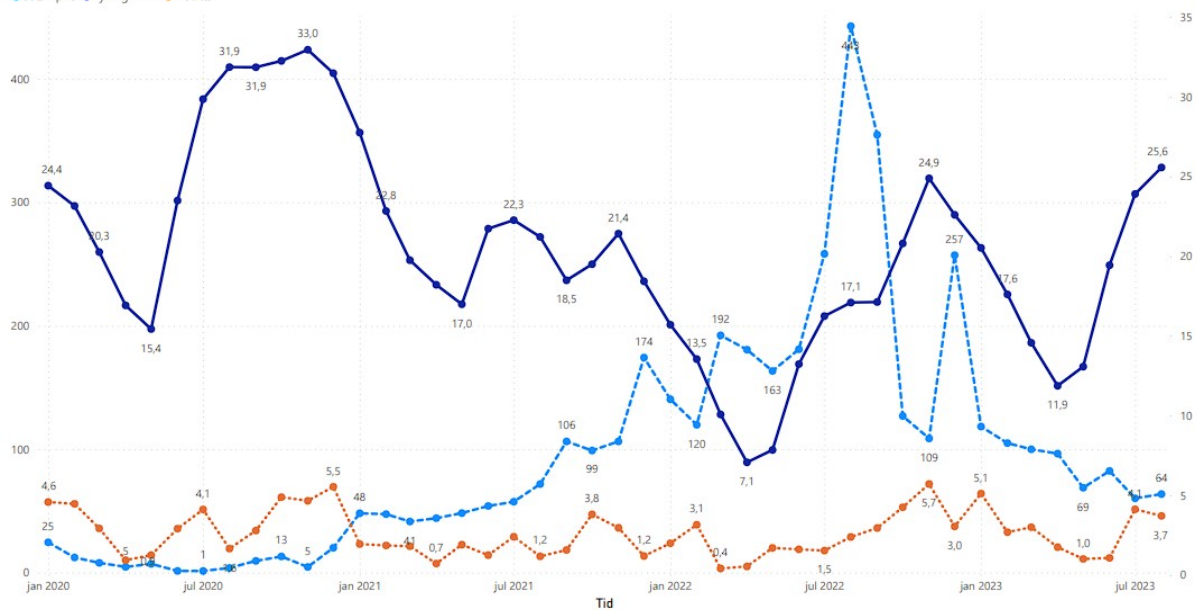
Her er de andre områdene:

Likt som med NO1, ser man at det er rekordlav fylling rett før perioden med rekordhøye strømpriser.

N02

Strømpris, Fylling TWh, og Nedbør over tid

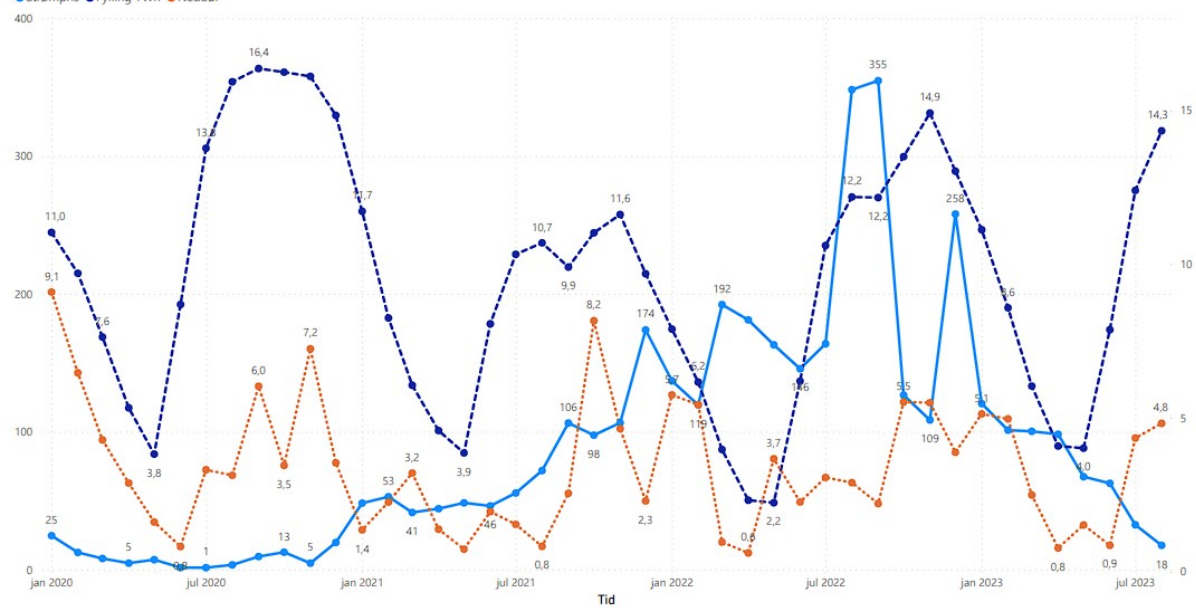
● Strømpris ● Fylling TWh ● Nedbør



N05

Strømpris, Fylling TWh, og Nedbør over tid

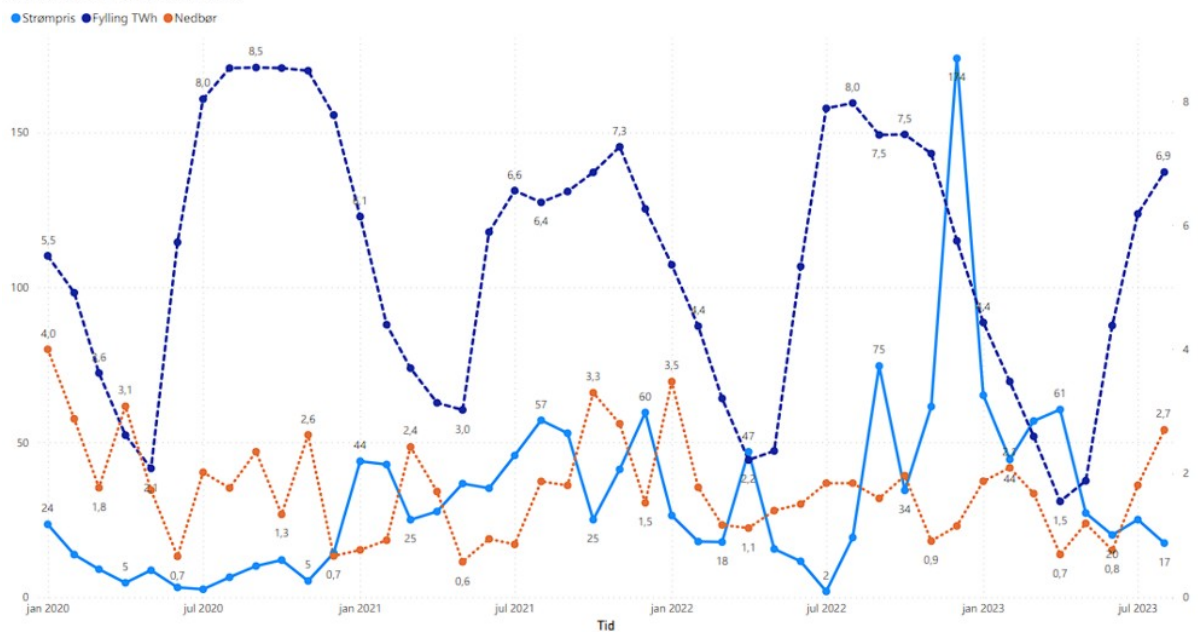
● Strømpris ● Fylling TWh ● Nedbør



I NO3 og NO4 er ikke den samme korrelasjonen som nevnt over like tydelig.

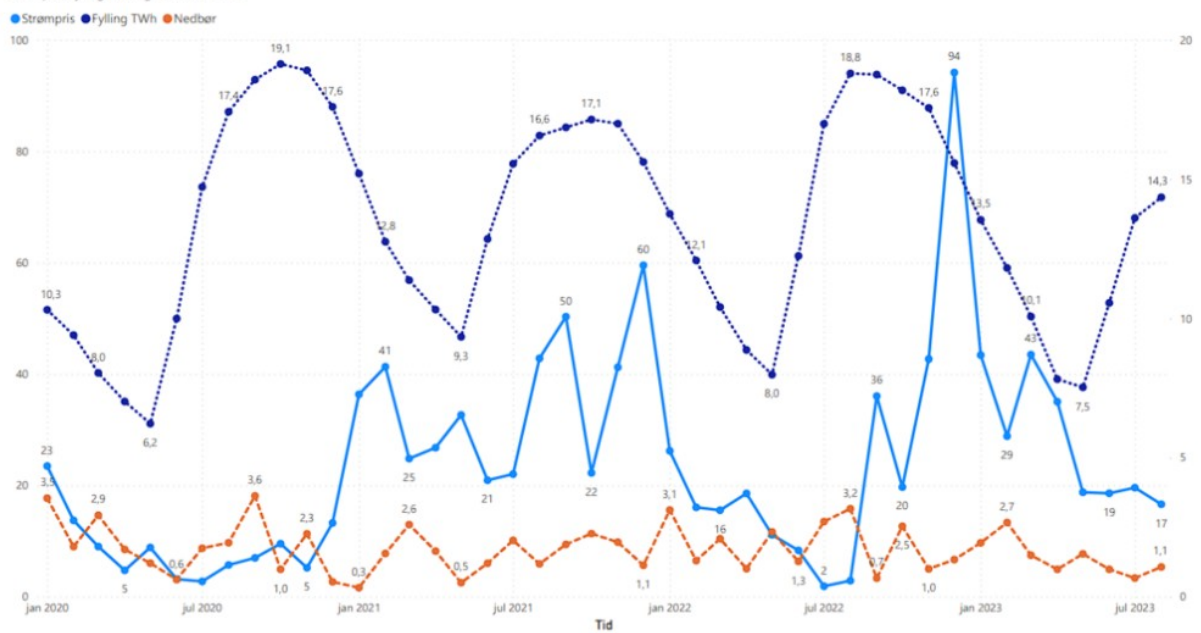
NO3

Strømpris, Fylling TWh og Nedbør over tid



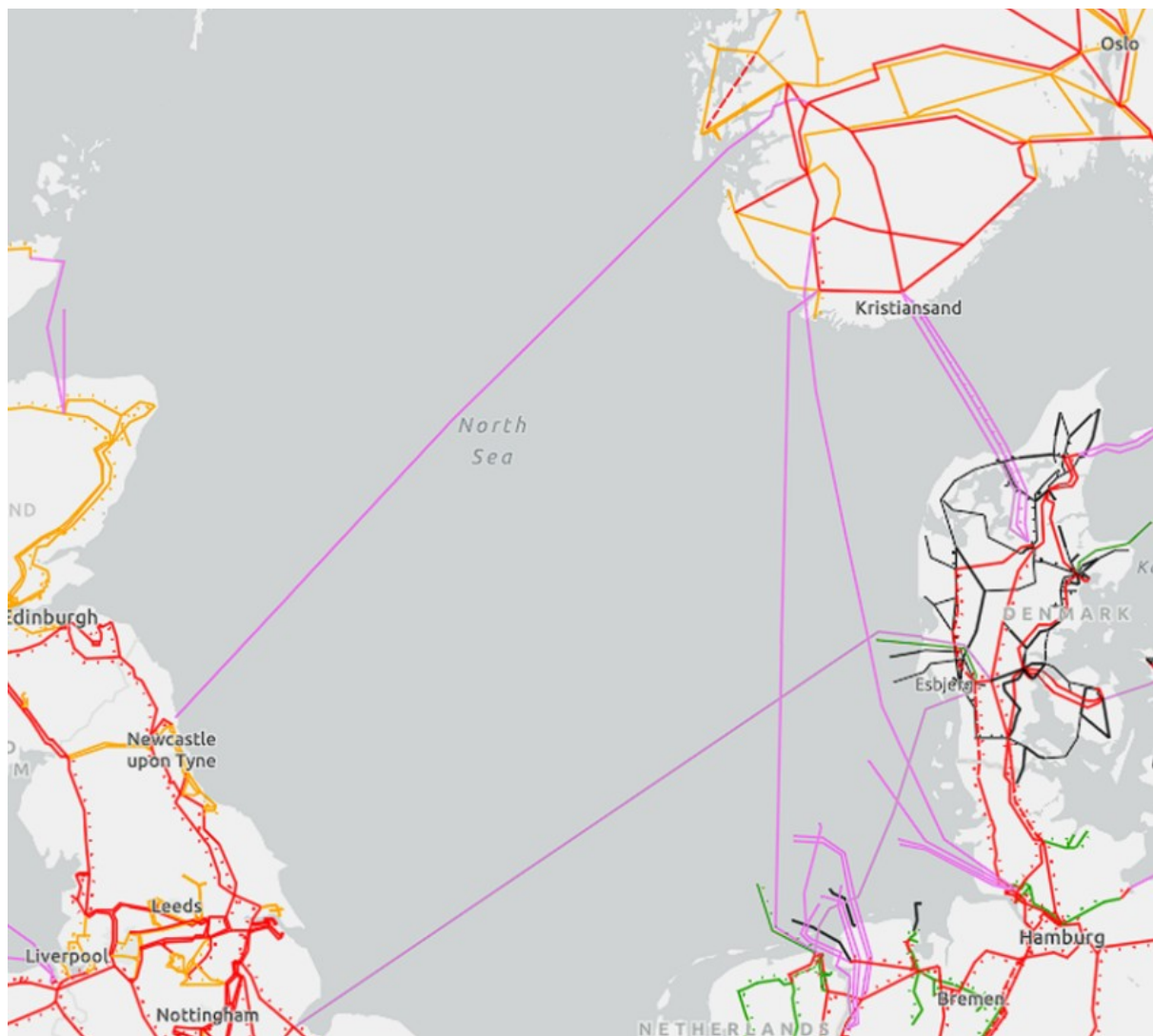
NO4

Strømpris, Fylling TWh og Nedbør over tid



De laveste strømprisene var i NO4 og NO3.

Den høyeste strømprisen var i NO2. Det var også fra NO2 .



Vi ser at kablene for kraftoverføring med utlandet går fra NO2.

Fram til nå er det sett mer på tilbudssiden av strømprisene. Men hva med etterspørselssiden?

Det kan være interessant å se på betydningen av kraftflyt med utlandet, for å se om det har påvirket strømprisene. Man kan også se på forholdet mellom produksjon og etterspørsel av kraft.

Denne koden henter ut data fra MongoDB for produksjon, forbruk, import og eksport. Dataene blir så kombinert i en dataframe som går daglig.

```
...
// ===== FORBRUK EKSPORT OG PRODUKSJON IMPORT
=====

// ===== HENTER DATAFRAMES =====

val forbruk_produksjon = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "forbruk_produksjon").load()
val import_eksport = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "import_eksport").load()
val day_ahead_NO = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"BD_analyse_rapport_db").option("collection", "day_ahead_NO").load()

// ===== SAMLER ETTER DAGVIS =====
val day_ahead_without_area = day_ahead_NO.drop("Area", "_id")
val import_eksport_d = import_eksport.drop("Hour", "DateTime", "_id")
val forbruk_produksjon_d = forbruk_eksport.drop("Hour", "DateTime", "_id")
import_eksport_d.show()

val day_ahead_NO_all = day_ahead_without_area.groupBy("Date").agg(
  avg("Day-ahead Price [EUR/MWh]").alias("day_ahead"),
)

val forbruk_eksport_all = forbruk_produksjon_d.groupBy("Date").agg(
  avg("Production"),
  avg("Consumption")
)

val import_eksport_all = import_eksport_d.groupBy("Date").agg(
  avg("Export"),
  avg("Flow from NO"),
  avg("Import")
)

// ===== KOMBINERER DATAFRAMES ETTER TID
=====

val import_eksport_produksjon_forbruk = import_eksport_all.join(forbruk_eksport_all, Seq("Date"), "inner")
val import_eksport_produksjon_forbruk_pris = import_eksport_produksjon_forbruk.join(day_ahead_NO_all,
Seq("Date"), "inner")

// ===== EKSPORTERER DATAFRAMEN TIL CSV
=====

val basePath = "C:/Users/Greencom/Desktop/BD_analyse"
```

```
val filePath = s"$basePath/import_eksport_produksjon_forbruk_pris.csv"

import_eksport_produksjon_forbruk_pris.write.option("header", "true").csv(filePath)

...
```

Den resulterende dataframen ser slik ut:

| Date | avg(Export) | avg(Flow from NO) | avg(Import) | avg(Production) | avg(Consumption) | day_ahead |
|------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| 2020-02-26 | 2864.044583333333 | 2845.048333333333 | 18.997916666666665 | 22407.5 | 19554.708333333332 | 13.309999999999999 |
| 2020-04-13 | 349.2333333333333 | -2646.787083333333 | 2996.021666666667 | 13325.083333333334 | 15885.5 | 2.388 |
| 2021-11-03 | 4978.742916666667 | 4812.257500000001 | 166.48499999999999 | 20882.041666666668 | 15976.208333333334 | 65.508 |
| 2022-10-05 | 1688.255833333333 | -375.65625 | 2063.9137499999997 | 13140.291666666666 | 13573.75 | 40.910000000000004 |
| 2023-01-21 | 4500.338750000001 | 4194.652916666667 | 305.6854166666667 | 23141.708333333332 | 18982.041666666668 | 108.088 |
| 2023-05-01 | 2616.309583333333 | 173.79708333333323 | 2442.513333333333 | 14715.958333333334 | 14567.875 | 75.772 |
| 2023-05-18 | 3674.2850000000003 | 1899.9870833333337 | 1774.2987500000002 | 14670.708333333334 | 12680.208333333334 | 47.199999999999996 |
| 2020-06-24 | 2377.2791666666662 | 2342.8579166666664 | 34.420833333333334 | 14265.25 | 11924.166666666666 | 1.532 |
| 2021-12-23 | 4057.9337499999997 | 3211.852916666667 | 846.08 | 22189.208333333332 | 19187.625 | 162.464 |
| 2022-10-07 | 1483.138333333333 | -1558.342083333333 | 3041.4808333333335 | 12131.125 | 13657.708333333334 | 19.64 |
| 2023-04-17 | 4475.614583333335 | 4159.432083333334 | 316.1825 | 18837.541666666668 | 14700.958333333334 | 91.03399999999999 |
| 2023-04-21 | 3019.8237499999996 | 1394.03875 | 1625.7849999999999 | 15191.25 | 13780.708333333334 | 74.11200000000001 |
| 2023-04-28 | 3314.5925 | 2897.61 | 416.9825000000001 | 18127.625 | 15111.75 | 90.63399999999999 |
| 2020-06-08 | 2404.0562500000005 | 2401.2016666666664 | 2.8549999999999999 | 15552.166666666666 | 13153.833333333334 | 3.272 |
| 2020-09-12 | 2369.8349999999996 | 2338.1179166666666 | 31.71833333333333 | 15400.958333333334 | 12629.75 | 11.086 |
| 2020-11-12 | 3509.08625 | 3460.7137500000001 | 48.37291666666667 | 19687.416666666668 | 16246.541666666666 | 6.076 |
| 2021-04-06 | 940.9095833333334 | -1093.37375 | 2034.283333333333 | 16682.291666666668 | 17628.916666666668 | 28.744000000000007 |
| 2022-05-17 | 3330.114583333332 | 2473.9275 | 856.1870833333336 | 15219.333333333334 | 12723.291666666666 | 112.52799999999999 |
| 2023-02-10 | 2869.8125 | 571.0916666666668 | 2298.720833333333 | 19168.666666666668 | 18616.958333333332 | 63.286 |
| 2020-01-05 | 1108.1845833333334 | 4.871666666666708 | 1103.31375 | 17754.333333333332 | 17830.041666666668 | 30.088 |

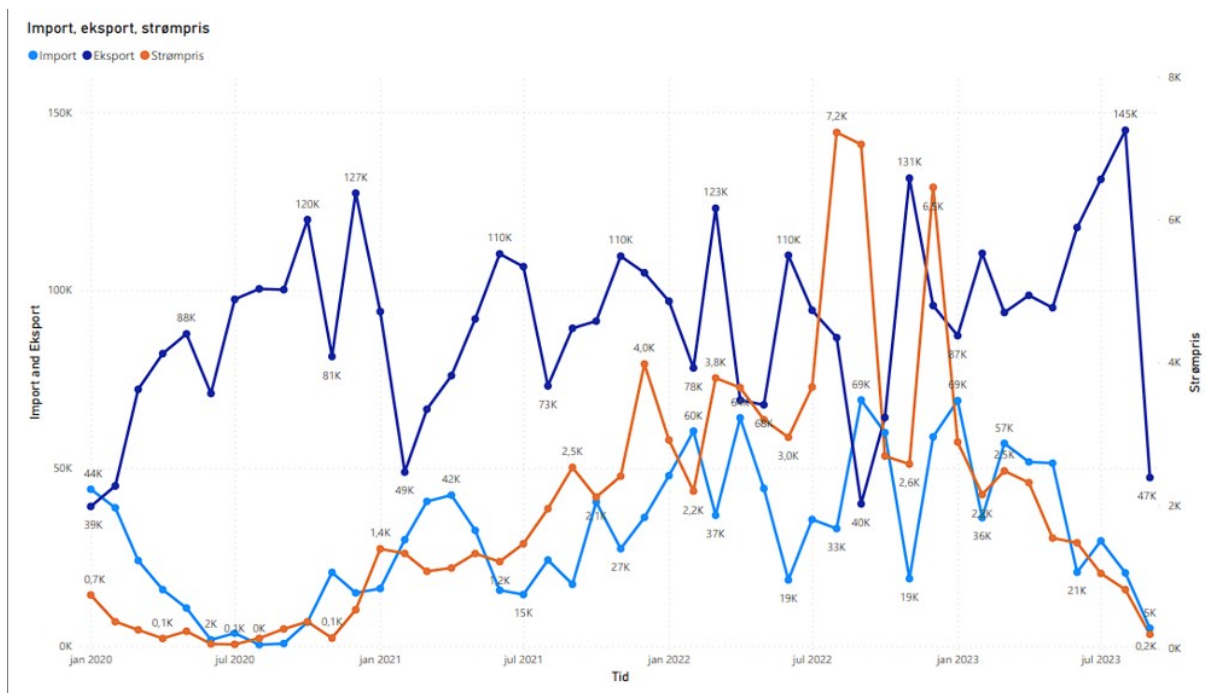
Den viser eksport av kraft, import, produksjon, forbruk, strømpriser, og flyt hver dag.

Denne dataen kan brukes til å få frem følgende grafer i powerBI.

Her ser man import, eksport, og strømpriser over tid for hele landet.

Med første øyekast er det ikke noen klar sammenheng mellom de tre.

Man ser at da strømprisene er høyest, er importen størst, og eksporten lavest, i perioden rett før har eksporten vært høyere, og importen lavere. Når strømprisene begynner å falle igjen, faller også importen, mens eksporten fortsetter å vokse.

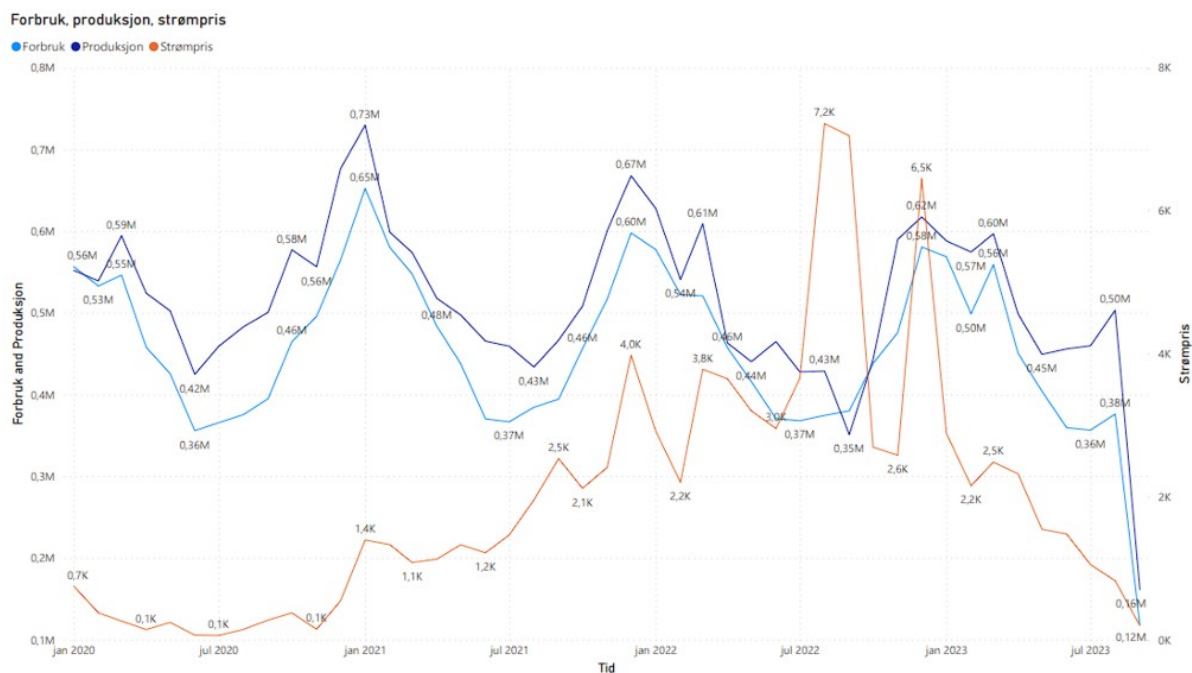


Denne grafen viser produksjon av kraft, forbruk, og strømpriser over tid.

Vi ser at forbruket og produksjonen følger hverandre.

Produksjonen er nesten alltid litt høyere enn forbruket.

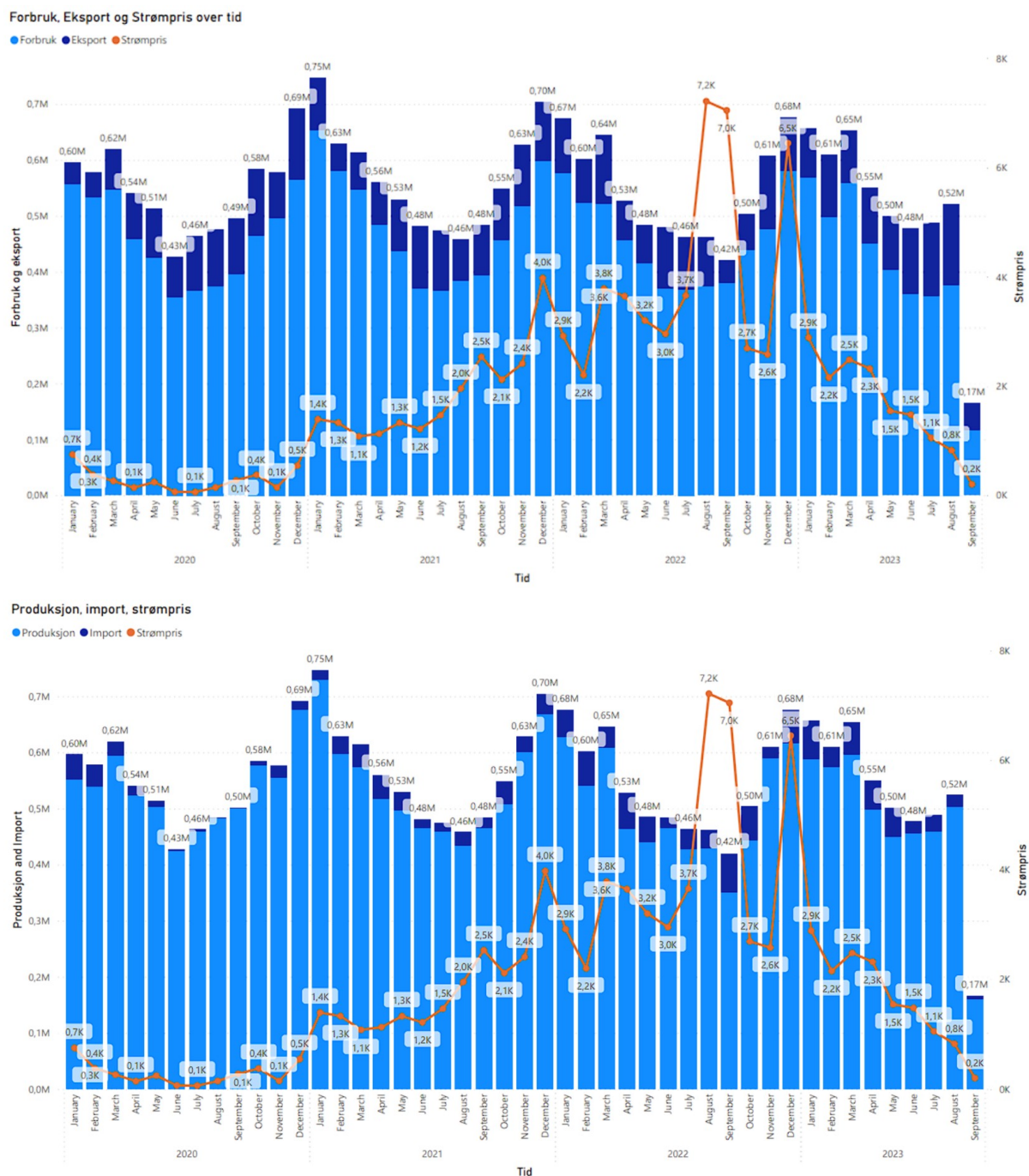
Forbruket og produksjonen går rytmisk opp og ned. Vi ser ikke klart noe klart tegn her på noe som kan ha hatt innflytelse på strømprisen. forbruket og produksjonen holder seg på samme nivå som før de høye strømprisene. I perioden strømprisen er høyest er også produksjonen lavest, men forbruket er også lavt i den perioden.



Man kan også bruke dataene til å summere eksport og forbruk (senker tilbud), og import og produksjon (øker tilbud).

De følgende grafene viser forbruk og eksport sammen, og produksjon og import sammen.

De to histogrammene følger samme rytme. Det er heller ikke klart her noe som kan ha forårsaket høye strømpriser. Strømprisene er høyest i den tiden produksjon og import er lavest, men på den samme tiden er også forbruk og eksport lavest.



Det virker som om det som har hatt størst innflytelse på strømprisen har vært lav magasinfylling, på grunn av lite nedbør. I perioden det er et tydelig dupp i fylling, er også strømprisen høyest. Det virker som om det er mindre korrelasjon mellom strømprisen og forbruk & eksport.

Ifølge vår analyse er det altså primært små mengder nedbør som har gitt de høye strømprisene.

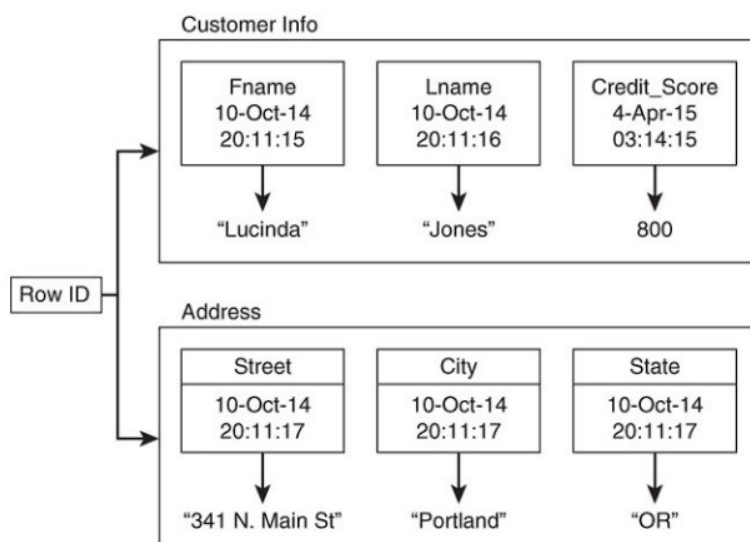
Mer om - Kolonnefamilie

Kort om hva kolonnefamilie

Kolonnefamilie er en type NoSQL database.

Data lagres i tabeller innenfor forskjellige rader. Disse radene har et row-id som fungerer som en id til dataene, så er det tabeller tilknyttet denne row-id'en.

I boken vises dette slik:



Hva er cassandra?

Apache cassandra er en NoSQL database basert på kolonnefamilie.

Den er åpne kildekode.

Det ble initielt laget av Facebook for å være en slags kombinasjon av Amazons Dynamo og Google Bigtable.

I Cassandra bruker man et språk kalt Cassandra Query Language (CQL) for å lage spørringer mot databasen.

Spørrespråket er inspirert av SQL.

Jeg tror at grunnen til at spørrespråket er veldig likt SQL er fordi når man lagde CQL, så var muligens tanken at man kan bruke SQL litt som en mal, slik at man ikke trenger å tenke ut alt av hvordan syntax-en burde være.

Jeg tror også det er fordi det da er det lett å komme i gang med spørrespråket hvis man kan SQL fra før av (Noe man sannsynligvis gjør).

Organisering av database

Databasen i en kolonnefamilie kalles for et keyspace.

Row key forklares i boka som å være en unik identifikator til flere kolonnerader i en database. Til sammen vil disse kolonnene uttrykke informasjon om en "entitet", for eksempel en person eller et produkt.

I cassandra kaller man gjerne kolonnefamilie for et table.

Så når man lager en kolonnefamilie så bruker man det som nøkkelord i spørringen:

```
CREATE TABLE column_family (  
    id UUID PRIMARY KEY,  
    column1 text,  
    column2 int  
);
```

I Cassandra må schema være spesifisert før man legger til data. Det betyr at før man kan legge til/endre data i en kolonne, så må kolonnen for dataene være definert når man lager kolonnefamilien.

Jeg tror at dette er gjort for å gjøre databasen mer oversiktlig.

Ved at man må definere schemaet først, gjør man så det settes regler for hvilke verdier man kan fylle ut når man setter inn data. Det betyr også at når man senere putter inn data, så har man noe å forholde seg til.

I cassandra lagres dataene med en timestamp, som vil ligge i en kolonne.

I cassandra vil hver rad inneholde følgende:

et navn, en datatype, og et timestamp.

I cassandra er kolonnene sortert etter kolonnenavnene.

Grunnen til at timestamp inkluderes er fordi da vet man når en rad sist ble endret, så kan man bruke det til å sammenligne hvilken versjon av dataene i de forskjellige nodene som er av nyeste versjon.

Datatyper

I boka sies det at Cassandra tilbyr rundt 20 forskjellige datatyper.

For å være eksakt så er det 21 forskjellige typer.

Hvis man inkluderer timestamp, og forskjellige typer lister, så er det 25 datatyper.

Map og tuple fungerer som key-value pairs.

Flere av disse datatypene kan i tilfeller oppgis med bruk av andre datatyper. For eksempel så kan datatypen time oppgis som en string og integer.

Jeg tror at grunnen til at man kan oppgi noen datatyper i andre datatyper, er fordi det gir mer fleksibilitet i designet av databasen.

La oss si at man f.eks. bruker et MAP, og vil ha en del forskjellige integer verdier, men så vil man også f.eks. en dato eller et uuid. Da kan man fortsatt bruke integer til å lage det.

Da kan man oppgi datoen/uuid-en som en integer, og Cassandra vil skjønne at dataverdien skal behandles som en dato/uuid.

I tillegg kan man definere "custom datatype".

Superkolonne

I Cassandra kan man også ha noe kalt superkolonne.

I Cassandra er kolonnefamilier lagret på samme fil.

Så det er viktig å holde kolonner som sannsynligvis refereres til sammen, i samme kolonnefamilie. Når man bruker en kolonnefamilie oppfordres det til å denormalisere data. Altså i relasjonsdatabase vil man dele opp dataen i flere tabeller, men i kolonnefamilie vil man kombinere dataene i samme kolonnefamilie.

I Cassandra lages en superkolonne ved at man lager en spesiell variabeltype, som kan inneholde flere variabler.

Slik som dette:

```
CREATE TYPE custom_datatype (  
    field1 text,  
    field2 int  
);
```

```
CREATE TABLE column_family (  
    id UUID PRIMARY KEY,  
    column1 text,  
    column2 int,  
    column3 <custom_datatype>  
);
```

Noen steder vil man se at man kan lage en superkolonne ved å bruke en `map<>` som definerer key-value pairs.

Men dette er en utdatert måte å lage en superkolonne.

Hvis man bruker `map` må alle key-verdiene, og value-verdiene være av samme datatype, noe som kan gjøre det vanskelig å lage databasen. Som sagt tidligere, så tror jeg det faktisk at `map` kun aksepterer en datatype, innflytelse på at man kan spesifisere noen datatyper i andre datatyper.

Andre datatyper, men det er ikke alltid det er fleksibelt nok. I tillegg så kan det være at man må sette seg litt inn i hvordan datatypen kan lagres på en måte som gjør at den kan fungere som en annen datatype.

I dag anbefales det at man bruker custom types for å lage kolonnefamilier.

| Column | | |
|---------------|----------------|-----------------|
| name : byte[] | value : byte[] | clock : clock[] |

| Super Column | |
|---------------|----------------------------|
| name : byte[] | cols : map<byte[], column> |

Random Partitioner

I implementasjoner av kolonne databaser må dataene i databasen fordeles over noder.

Dette betyr at rader med row-keys fordeles over forskjellige noder.

I Cassandra gjøres dette på en måte som kalles random partitioner, som betyr at radene er fordelt tilfeldig over forskjellige noder.

Dette resulterer i at dataene skal bli fordelt i nærheten av likt over de forskjellige nodene.

Dette gjøres med hashing. Hashing bruker verdier fra en kolonnefamilien til å generere en så tilfeldig så mulig tallverdi som kan brukes i indeksen.

Så i Cassandra resulterer hashing i at dataene blir fordelt likt over nodene.

I tillegg kan hashing være svært raskt ($O(1)$), så det gjør opplesningen av verdier raskere enn det ville vært med en del andre algoritmer.

Alternativt til hashing så kan man også bruke et bloom-filter for å finne data. Bloom filter er noe dårligere, men også billigere med tanke på ressurser.

Gossip protocol

En database kan være organisert i forskjellige noder (servere).

Disse nodene inneholder deler av databasens data.

Så det er et problem hvordan disse nodene skal snakke sammen for å dele informasjon.

Boken nevner at et problem å løse er hvordan nodene skal snakke med hverandre. Hvis en node har en viss status, og skal melde om dette til alle de andre nodene, så blir det veldig mange beskjeder som går til hver server.

Hvis N er antall servere, så er $N*(N-1)$ Antall beskjeder som trengs for å oppdatere nodene.

Boka nevner så en løsning på dette kalt gossip protocol.

Istedenfor at hver node oppdaterer alle de andre nodene om seg selv, så kan heller en node oppdatere andre noder om seg selv, og alle nodene den vet om, så de nodene gjør det samme. Da vil man redusere antall beskjeder som sendes.

Cassandras gossip protocol er litt annerledes. Cassandra har med syn-ack protokoll i sin gossip protocol, i tillegg så sendes det med versjonskontroll.

- En node initierer en samtale med en annen node.
- Den initierende noden sender en initierende beskjed (syn beskjed).
- Den andre noden svarer med at den har fått beskjeden (en ack beskjed)
- Etter å ha fått tilbakemelding, sender den initierende noden melding om at den fikk tilbakemeldingen (en ack 2 beskjed).
- I løpet av samtalen blir hver node oppdatert om den andre noden og nodene de kjenner til.
- I tillegg sendes det med versjonsinformasjon, slik at nodene som kommuniserer kan vite hvem av dem som har de meste oppdaterte dataene.

Så når man designer en database i cassandra, så er det viktig å bruke timestamp for at databasen skal kunne vite hvilke data som er av nyeste versjon.

Jeg tror at grunnen til at Cassandra bruker TCP med syn-ack istedenfor UDP er fordi det er viktig med konsistens i databasen, så da kan ikke kun deler av dataene overføres.

Klystre

Boken nevner at det finnes to typer arkitektur vanligvis brukt i distribuerte databaser. Disse er “multiple node” og “peer-to-peer”.

I “Multiple node” består databasen av forskjellige typer noder, og en sentralisert server for å konfigurere klystre. I en Multiple node database er det en mester-node som håndterer informasjon om nodene.

Cassandra bruker Peer-to-Peer, der man kun har en type node. Det betyr at det ikke er noen mester som håndterer nodene.

Hver node er derfor ansvarlig for å holde rede på informasjon om klysteret, altså hvilken data som tilhører hvilken node, og hva av informasjon som er nyeste.

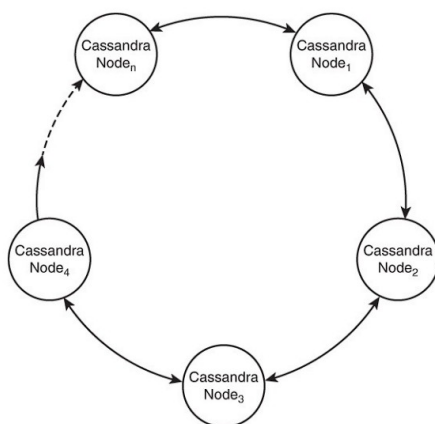


Figure 9.10 Cassandra uses a peer-to-peer architecture in which all nodes are the same.

Jeg tror at Cassandra bruker en peer-to-peer-modell fordi det virker som det å håndtere databasen, og hastigheten til databasen blir raskere.

Man trenger ikke å sette opp en spesiell mesternode. Hver node kan lages på samme måte, og fortsatt kunne dekke alle behov til databasen.

Jeg tror det også gjør Cassandra raskere fordi nodene trenger ikke å vente på en mesternode, men hver node kan initiere operasjoner sammen med andre noder.

Informasjonsentropi

I praksis når man lagrer data i en database vil det oppstå informasjonsentropi.

Hvis en instans i databasen sier at et entitet har en viss status, og en annen instans sier at entiteten har en annen status, så har databasen inkonsistens.

Cassandra løser informasjonsentropi på følgende måte:

- En server kan starte en anti-entropi sesjon med en annen server.
- Den initierende serveren sender en hash datastruktur utarbeidet fra dataene i en kolonnefamilie.
- Den mottakende serveren kalkulerer også en hashstruktur med sin egen versjon av kolonnefamilien.
- Hvis de to resulterende hashstrukturene ikke er like, vet man at en av kolonnefamiliene har gammel dato.
- Da finner serverene ut hvilken kolonnefamilie som er utdatert, og oppdaterer serveren med den utdaterte kolonnefamilien.

Så som sagt tidligere, så er det viktig å bruke timestamp når man lager databasen, fordi det er med timestampen cassandra vet hvilken data som er den nyeste.

Lese operasjoner

I praksis lagres data til en database på forskjellige servere.

En database programvare må derfor kunne håndtere etterspørseler mot servere.

F.eks. hva skjer hvis dataen som etterspørres av brukeren ikke er tilstede på serveren?

Tidligere ble det sagt at hver node inneholder informasjon om de andre nodene, altså hvilke data de inneholder.

Så hvis en node leser forespørsel mot data den ikke selv har, så sender den forespørselen videre til den noden som har dataen.

Noden som har dataen sender så dataen tilbake til noden som fikk forespørselen, så sender den noden igjen dataen videre tilbake til brukeren.

Skrive operasjoner

I teorien tar databaser og lagrer data fra brukere gjennom spørringer.

Men i praksis kan det være situasjoner med skrijving som man må håndtere.

Hva hvis dataene som skrive operasjonen sendes til, ikke lagres på den noden?

I det tilfellet sendes forespørselen videre til den noden som som besitter den aktuelle row-id.

Men hva om noden med row-id'en er nede?

Hvis noden med row-id'en er nede, så lagres skriveoperasjonen istedenfor.

Periodisk sjekkes det om den andre serveren er oppe igjen.

Hvis den er oppe igjen, sendes skriveforespørselen.

Cassandra omtales for å være bra på å håndtere skriveoperasjoner.

Jeg tenker at dette er på grunn av hvordan den kan ta i mot skrive operasjoner til tross for at serveren med dataene som det skrives til ikke er tilgjengelig.

Mer om - Grafdatabase

En generell innføring i grafdatabase

Hva er en graf:

En graf representerer et sett med objekter disse kalles noder eller vertex, som er koblet sammen med kanter. Disse grafene brukes til å modellere komplekse sammenhenger mellom dataelementer. (forelesning om grafdatabaser)

Grafdatabaser:

Grafdatabaser er databaser som er bygget opp av noder og kanter, der hver node kjenner til sin nabonode, og det vil ta like lang tid å finne informasjon om en nabonode uansett hvor mange noder som finnes i databasen. De er optimalisert for navigasjon av kanter, og de er spesielt effektive når det er snakk om å håndtere oppgaver som for eksempel anbefalingssystemer.

I sammenligning med RDBMS databaser er grafdatabaser som regel mye raskere når det kommer til oppgaver som krever navigering gjennom komplekse sammenhenger. De krever altså mindre kode for å implementere slike funksjoner, også for sanntids anbefalinger, der det kreves rask og effektiv navigasjon gjennom data vil grafdatabaser være mer foretrukne. (forelesning om grafdatabaser, Geitle, M)

Kildehenvisning:

Forelesninger

Emneboken Sullivan D. NoSQL for Mere Mortals®. 1st edition. Addison-Wesley Professional; 2015.

Cassandra dokumentasjonen: [Welcome to Apache Cassandra's documentation! | Apache Cassandra Documentation](#)

Elementer av grafdatabasen:

VERTEX:

En node representerer et objekt som for eksempel land, personer, byer, kinoer eller for vårt tilfelle vannmagasiner. Disse nodene har en egen unik identifikator, som i likhet med en radnøkkel i en familiekollonedatabase og en primærnøkkel i en relasjonsdatabase. (Sullivan, 2015, s.298)

Som tidligere nevnt kan en node representere hvilket som helst objekt, som har en relasjon til et annet objekt. Nodene kan også ha egenskaper, som da for eksempel ved at en person kan ha navn, alder og kjønn, eller at en by kan ha innbyggertall og areal. (Sullivan, 2015, s.298)

EDGE:

Kanter i en grafdatabase utgjør forholdene eller relasjonene mellom nodene. Nodene har ofte forhold til andre ting innenfor den samme kategorien, så tar vi utgangspunkt i en person så kan denne personen da ha ulike forhold til ulike personer. Kantene vil da beskrive relasjonen mellom personene.

I likhet med nodene kan også kantene ha egenskaper, som i et familietre kan kantene ha egenskaper som for eksempel "adopsjon" eller "biologisk", og "gift".(Sullivan, 2015, s.299)

En annen egenskap som ofte brukes når det kommer til kanter, er vekt. Disse representerer en verdi av forholdet, disse kan representere, kostnad, distanse eller en annen form for målbare enheter om relasjonene mellom objektene.(Sullivan, 2015, s.299)

Det finnes to ulike kanter:

- Directed
- Disse kantene har en retning for å forklare forholdet mellom objektene, retningen viser da hvor relasjonen begynte eller hvor den går fra. For eksempel ved at i et familietre vil kanten være rettet fra foreldre til barn for å visualisere forholdet.
- Undirected o En kant trenger ikke alltid ha en retning, for eksempel kan en graf over en motorveier være urettet, da det kan være trafikk begge veier

(Sullivan, 2015, s.301)

Under ser vi en graf med to noder, og en kant. Disse kan representere en person(node a) med et forhold (kanten) til en annen person(node b).

Ulike typer grafer

- Urettet og retter graf

En urettet graf er en graf der kantene ikke er rettet fra eller mot noder og en rettet graf er en graf med kanter som er rettet fra og mot noder og som da beskriver forholdet dypere.. Som beskrevet tidligere om rettede og urettede kanter er prinsippet det samme når det kommer til denne type graf.

- Flow Network

En flow network graf er en graf med rettede kanter, og hvor hver kant har en kapasitet. Hver node har et sett med utgående og ett sett med inngående kanter, summen av de inngående kantene kan ikke være høyere enn kapasiteten til de utgående.

Unntak: Source noder, disse nodene har kun utgående kanter og Sinks noder, disse har kun inngående kanter.

- Todelt graf (bipartite graph)

Dette er en graf med to distinkte sett av noder, der hver node i et sett er kun lenket til noder i det andre settet. Disse grafene er nyttige når man modellerer forhold mellom ulike typer objekter.

- Multigraf

En multigraf er en graf med flere kanter mellom nodene. Dette er nyttig når man har noder som har flere forhold til en annen node, dette kan være for eksempel når det kommer til transport mellom ulike plasser, eller sosiale nettverk.

• Vektet graf
I denne type graf har hver kant en nummer knyttet til seg, dette nummeret beskriver en verdi kanten har. Dette er mye brukt for å finne den korteste veien mellom nodene. En populær metode for å finne den korteste veien til en node er Dijkstra's algorithm.

(Sullivan, 2015, s.308-311)

Fordeler ved grafdatabaser:

Raskere spørringer:

I relasjonsdatabaser må man utføre sammenslåinger (join-operasjoner) for å finne koblinger eller forhold. Denne prosessen kan være tidkrevende, spesielt når man må gjøre dette mange ganger på store tabeller.

I motsetning til dette så trenger man ikke å utføre sammenslåinger i en grafdatabase. I stedet kan vi navigere langs kantene fra en node til en annen. Denne tilnærmingen er enklere og raskere. (Sullivan, 2015, s.293)

En enklere modellering av databaser:

Modelleringsprosessen kan også bli forenklet når man jobber med grafdatabaser. Når man jobber med relasjonsdatabaser begynner vi ofte med å modellere de viktigste enhetene i systemet. Dette fungerer greit helt til vi får flere enheter som interagerer med samme enhet, som i en relasjonsdatabase kalles "mange-til-mange" relasjon. Dette krever da at man oppretter en ekstra tabell, og må lage komplekse spørringer for å håndtere dette. Når det kommer til grafdatabaser er dette ikke nødvendig, her kan vi enkelt se forholdene gjennom kantene. (Sullivan, 2015, s.295)

Flere relasjoner mellom ulike noder:

I en grafdatabase har du mulighet til å bruke flere typer kanter, dette tillater oss å kunne modellere flere relasjoner mellom nodene. Dette er også mulig når det kommer til relasjonsdatabaser, men dette er lettere å forstå når man bruker en grafdatabase hvor du kan følge kantene, og få et visuelt bilde av det. (Sullivan, 2015, s.296)

Unikt ved grafdatabaser:

Når det er snakk om databaser er operasjoner som innsetting, lesing, oppdatering og sletting av data vanlige. Disse er da også relevante når det kommer til grafdatabaser, men grafdatabaser skiller seg ut ved å tilby noe ekstra funksjonalitet som er spesielt egnet til å kunne håndtere komplekse sammenhenger mellom relasjoner i data. (Sullivan, 2015, s.303)

Disse unike operasjonene er sammenslåing av grafer, kryssing av grafer og grafovergang. Sammenslåing av grafer innebærer å kombinere noder og kanter fra to ulike grader. Dette vil gi en helhetlig graf som vil representere dataene i en mer omfattende sammenheng. (Sullivan, 2015, s.303)

Kryssing av grafer eller graf skjæring, innebærer å identifisere og beholde de samsvarende elementene i to grafer. Dette gjør at vi kan fokusere på det som er delt mellom grafene og kan være nyttig når vi skal analysere overlappende data eller relasjoner. (Sullivan, 2015, s.304)

Graftraversing, involverer å systematisk bevege seg gjennom nodene for å kunne utføre bestemte handlinger, som for eksempel å hente eller oppdatere informasjonen til nodene. Dette kan sammenlignes ved å planlegge en reiserute mellom ulike byer på et kart, der veiene vil representere kantene. (Sullivan, 2015, s.305)

Valg av grafdatabaseprogramvare:

Neo4j

I vårt prosjekt har vi brukt Neo4j da dette var programvaren gitt av foreleser. Dette er en åpen kildekode database, og ble originalt utgitt i 2007. Neo4j er kjent for sin skalerbarhet, som gjør den i stand til å kunne håndtere milliarder av noder, men også for høy tilgjengelighet og feiltoleranse. (forelesning om grafdatabaser, Geitle, M)

I Neo4j er en graf satt sammen av noder og kanter, der hver node har en intern identifikator og kan være merket med en eller flere labels. Hver node kan også ha egenskaper, som er representert som nøkkel-verdi par. Nøkkelen er da en streng, mens verdien kan være en atomisk verdi eller en array av atomiske verdier (boolean, numeriske og strengverdier). Kantene i Neo4j har også unike identifikatorer, en retning (disse kan være enveis eller toveis), en start- og en sluttnode, en type og egenskaper. (forelesning om grafdatabaser, Geitle, M)

De har offisielle drivere for flere programmeringsspråk, inkludert Java, .NET, JavaScript og Python, men de har også tredjepartsdrivere for en rekke andre språk. Spørrespråket som brukes i Neo4j er "Cypher", dette er inspirert av SQL og SPARQL. (forelesning om grafdatabaser, Geitle, M)

Diskusjonsdel:

Gjennom teorien om grafdatabaser oppnår vi en dypere forståelse av underliggende konsepter som definerer denne databasetypen. Grunnlaget består som tidligere nevnt av en grafstruktur der data er organisert i noder og kanter. Fleksibiliteten i denne organiseringen gjør at grafdatabaser er spesielt egnet for håndtering av komplekse relasjoner og nettverksstrukturer.

Skalerbarhet og ytelse:

Boken gir et viktig perspektiv på skalerbarheten i grafdatabaser, noe som er avgjørende når det kommer til å kunne håndtere veksten i antall noder og kanter, samt økning i antall brukere, og økning i antall og størrelse på egenskaper på noder og kanter. Denne veksten kan lede til økte krav til databasetjeneren, med implikasjon om at mange grafdatabaser er designet for å kjøre på en enkelt tjener. I slike tilfeller blir vertikal skalering en nødvendighet. (Sullivan, 2015, s.327)

Dette står i kontrast til Neo4j som er designet for både vertikal og horisontal skalering. Neo4j har altså evne til å dele belastningen over flere tjenere, som gir mulighet for å håndtere større skalerbarhet mer effektivt. Denne tilnærmingen muliggjør også håndtering av betydelig større grafer, noe som er nyttig i tilfeller der det er behov for å analysere komplekse nettverk, som for eksempel i sosiale nettverk eller anbefalingssystemer.

Transaksjonshåndtering og pålitelighet:

I boken fokuseres det på nødvendigheten av å sikre data tilgjengelighet og konsistens, spesielt i tilfeller der det kan oppstå hardwarefeil eller er nødvendig med vedlikehold. Den nevner en to-fase-commit mekanisme som et middel for å oppnå høy tilgjengelighet og konsistens. Dette sikrer at endringene i en primær database replikeres nøyaktig i en backup database, med vekt på at begge databasene må være konsistente for at transaksjonen kan betraktes som fullført. (Sullivan, 2015, s.57)

Neo4j støtter ACID transaksjoner, dette sikrer pålitelighet og konsistens i behandlingen av data. Dette betyr at hver transaksjon i Neo4j er atomisk, konsistent, isolert og holdbar. De tilbyr også mekanismer for sikkerhetskopiering og gjenoppretting for å håndtere systemfeil. Dette er avgjørende for å sikre dataintegritet og tilgjengelighet.

To-fase commit-metoden som er diskutert i boken fokuserer på konsistens mellom primære og backup databaser, i Neo4j er det mer fokus på å direkte opprettholde ACID-egenskaper innenfor hver enkelt transaksjon. Dette er med på å sikre at komplekse transaksjoner kan utføres på en sikker måte.

Spørrespråk:

I boken nevnes både Cypher og Gremlin. Cypher er et deklarativt, SQL-lignende språk som tilbyr en tilnærming til relasjonelle databaser for det som er kjent med det. Cypher til syntaks forenkler interaksjonen med graf-databaser og effektiviserer utførelsen av komplekse spørringer. Gremlin gir en mer iterativ tilgang til gratraversering som kan være foretrukket i mer komplekse eller spesialiserte scenarioer. (Sullivan, 2015, s.319)

Praktisk bruk av Neo4j:

Boken bidrar til å gi verdifull grunnlag av teori for forståelse av grafdatabaser, mens Neo4j viser hvordan denne teorien omsettes til praksis. Neo4j beveger seg noe utover de prinsippene beskrevet i boken, ved å introdusere en rekke funksjoner som forsterker den praktiske anvendelsen av grafdatabaser. Med en kombinasjon av god skalerbarhet, ytelse, og robuste sikkerhetsfunksjoner fremstår Neo4j som et god løsning for komplekse datasett. Videre tilbyr også Neo4j verktøy for visualisering, noe som er betydelig når det kommer til å forstå sammenhengene i en grafdatabase.

Oppsummering av sammenligning:

Fra hva boken skriver om databasers skalerbarhet og ytelse, viser Neo4j i praksis hvordan man kan ta i bruk avansert teknikk og innovativ arkitektur som brukes for å takle utfordringer som kommer med store og komplekse databaser. Med en kombinasjon av horisontal og vertikal skalering, og gode spørremotorer er Neo4j en plattform som kan løse utfordrende problemer innenfor grafdatabaser.

Både Neo4j og boken anerkjenner viktigheten av transaksjonsbehandling og pålitelighet. Selv om begge legger vekt på nødvendigheten av robuste transaksjonsbehandlinger, går Neo4j utover den teoretiske delen ved å implementere praktiske løsninger som er skreddersydd for dagens behov. Dette viser til den raske utviklingen innen databaseteknologi og fremhever hvordan nåtidens løsninger kan tilpasse seg.

Kildehenvisning:

Sullivan,D. (2015) NoSQL for Mere Mortals. (Første utgave). Addison-Wesley
ISBN-13: 978-0-13-402321-2 ISBN-10: 0-13-402321-8

.....Individuelt bidrag 1 - Sindre

Innledning

I denne oppgaven har jeg vurdert å ta for meg både dokumentdatabaser og grafdatabaser siden dette er de to temaene som stikker seg ut som interessante for meg. Jeg konkluderte med at førstnevnte vil passe bra for denne oppgaven, siden jeg har hatt litt erfaring med grafdatabaser tidligere i emnet Databasesystemer og at jeg ønsker å ta fatt på et tema som for meg er helt nytt og ukjent.

I denne oppgaven kommer jeg først til å gå gjennom grunnleggende teori om dokumentdatabaser og NoSQL for å gi kontekst, men også for å samle stoff for egen orientering. Så, til slutt kommer jeg til å se nærmere på hvilke forskjeller som finnes mellom innholdet i emnets teoribok og dokumentasjonen til programvaren og hvorfor disse forskjellene oppstår. I dette tilfellet vil jeg sammenligne boken «NoSQL for Mere Mortals» av Dan Sullivan opp mot dokumentasjonen til MongoDB. Jeg skal også se nærmere på hvordan disse forskjellene vil påvirke bruken av databasen.

Bakgrunn

Om NoSQL

Før jeg setter i gang med å forklare hva dokumentdatabaser er, må vi vite hva NoSQL er og hvorfor NoSQL-databaser oppstod. Begrepet NoSQL vil på fagspråket bli brukt om databaser som ikke er relasjonelle. Begrepet NoSQL står enten for «Not only SQL» eller «Non SQL», avhengig av hvem man spør. Essensen er at NoSQL-databaser ikke lagrer data på samme måte som relasjonsdatabaser, hvor ulike tabeller med data blir koblet sammen via nøkler, altså delte egenskaper. I NoSQL-verden finnes det fire ulike typer databaser:

1. Dokumentdatabaser
2. Kolonnefamiliedatabaser
3. Grafdatabaser
4. Nøkkel-verdidatabaser

Så; litt historie. På slutten av 2000-tallet sank kostnadene for lagring til et lavt nok nivå at man innså at dataduplisering ikke ville være et like viktig moment å optimalisere for under utvikling av databaser. Siden prisene for lagring nå var blitt lave nok, kunne man tenke nytt når det gjaldt datalagring. Som en konsekvens av lavere kostnader for lagring, så man også større behov for databaser som både kunne lagre og håndtere store datavolumer raskere. Ikke bare var datamengdene blitt større enn tidligere, men de oppstod også i flere former enn tidligere. Strukturerte, ustrukturerte og polymorfiske data førte også til et behov for veldig fleksible databaser. Hint: Schemaless databaser.

Så for å oppsummere; utviklingen av NoSQL-databaser var et produkt av følgende faktorer:

- Lavere kostnader for lagring
- Lavere kostnader fører til økt behov for datalagring og spørringer mot data
- Data oppstår i flere former enn tidligere på grunn av punktene over
- Behov for mer fleksibilitet

Grunnleggende teori om Dokumentdatabaser

Dokumenter

Dokumentdatabaser lagrer data i dokumenter som har en struktur som kan minne om JSON-objekter. Så hvert dokument kan inneholde flere felter bestående av nøkkel-verdipar.

La oss ta et banalt eksempel; en nøkkel være «id» og verdien være «1234». Disse utgjør til sammen et felt i et dokument. Et dokument kan ha flere felter. Slik som for eksempel «navn» og «Sindre». Dokumentet vil nå få en struktur som ser sånn ut:

```
{  
  "id": 1234,  
  "navn": "Sindre"  
}
```

Så vi ser at verdiene kan være tall og strenger. Men de kan også være boolske verdier, lister eller forskjellige objekter. Selv om dokumentdatabaser som oftest ligner JSON i sin struktur, er det også verdt å nevne at det også er mulig å lagre data i formater som XML, PDF eller diverse Office-formater i dokumentdatabaser.

Har vi flere like dokumenter, kan vi lagre dem i det som kalles en samling eller «Collection» som det kalles på fagspråket. Jeg har utvidet eksempelet over til å ta for seg mer data for å vise strukturen til en samling.

```
{  
  "brukere": [  
    {  
      "id": 1234,  
      "navn": "Sindre",  
      "email": "sindrelh@hiof.no",  
      "alder": 29,  
      "adresse": {  
        "gate": "BRA Veien 4",  
        "by": "Halden",  
        "postnummer": "1757"  
      }  
    },  
    {  

```

```
    "id": 5678,  
    "navn": "Tim",  
    "email": "tim@hiof.no",  
    "alder": 34,  
    "telefon": "+4712345678"  
  }  
]  
}
```

Så, vi ser ovenfor en samling i en database som har navnet «brukere». Det vi også ser her i dette eksempelet er at de to dokumentene ikke er helt like. Sindre har ikke et felt for telefonnummer og Tim har ikke et felt for adresse. Så dokumentene kan ligne hverandre, men de kan fint ha forskjellige egenskaper.

Det er verdt å nevne at god praksis er å holde irrelevante dokumenter adskilt i egne samlinger. Et eksempel på dårlig praksis kan være å lagre kunder og ordrelinjer i samme samling. Det er mulig å gjøre, men de har strengt tatt ikke noe til felles med hverandre og burde lagres i sine respektive samlinger – adskilt fra hverandre. Dette gjør det lettere å gjøre spørringer mot databasen senere.

Spørringer mot dokumentdatabaser

Vanlige CRUD-operasjoner kan gjøres mot dokumentdatabaser i form av spørringer. Dokumentdatabaser som MongoDB bruker et eget spørrespråk kalt MQL. Så, i dette eksempelet ønsker å demonstrere hvordan jeg opprettet databasen med tilhørende samling i forrige eksempel og hvordan jeg gjorde enkle operasjoner i databasen ved å skrive MQL-spørringer i MongoShell.

1. Oppretting av database

use eksempelDatabase

2. Oppretting av samling og innsetting av dokument: Setter inn brukeren Sindre

```
db.brukere.insertOne({
  id: 1234,
  navn: "Sindre",
  email: "sindrelh@hiof.no",
  alder: 29,
  adresse: {
    gate: "BRA Veien 4",
    by: "Halden",
    postnummer: 1757
  }
})
```

3. Legge til nytt dokument i samlingen: Brukeren Tim

```
db.brukere.insertOne({
  id: 5678,
  navn: "Tim",
  email: "tim@hiof.no",
  alder: 34,
  telefon: "+4712345678"
})
```

4. Hente ut alle dokumenter hvor navn er Tim

```
eksempelDatabase> db.brukere.find({navn: "Tim"})
[
  {
    _id: ObjectId("652da7b061172726ba91c7dd"),
    id: 5678,
    navn: 'Tim',
    email: 'tim@hiof.no',
    alder: 34,
    telefon: '+4712345678'
  }
]
```

5. Oppdatere dokument: Her alder for en bruker Tim med ID 5678

```
eksempelDatabase> db.brukere.updateOne({id: 5678}, {$set: {age: 30}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

6. Slette dokument: Sletting av bruker med ID 1234, altså brukeren Sindre

```
eksempelDatabase> db.brukere.deleteOne({id: 1234})
{ acknowledged: true, deletedCount: 1 }
eksempelDatabase> db.brukere.find()
[
  {
    _id: ObjectId("652da7b061172726ba91c7dd"),
    id: 5678,
    navn: 'Tim',
    email: 'tim@hiof.no',
    alder: 34,
    telefon: '+4712345678',
    age: 30
  }
]
```

Dokumentdatabaser vs. Relasjonsdatabaser - Fordeler

Her ser vi på noen fordeler sett med øyne som kommer fra verdenen innen relasjonsdatabaser.

- Schemaless

Som vi så i eksemplene på forrige side, så var dokumentene ulike i sin oppbygging og vi så at oppbyggingen av dokumenter kunne variere i antall felter og innhold. Dette kommer av at dokumentdatabaser ikke er avhengige av et fast schema for å kunne legge til nye dokumenter i databasen. Dette er i mange tilfeller en fordel som gjør at man oppnår langt større fleksibilitet sammenlignet med tradisjonelle relasjonsdatabaser – hvor det er gitt føringer på hvordan data eksisterer ved at man har definert tabeller, kolonner, nøkler og constraints. Vi oppnår med andre ord polyformisme for dataene.

- Skalerbarhet

En dokumentdatabase tar ikke hensyn til redundans i like stor grad som en relasjonsdatabase gjør. Data i en dokumentdatabase kan oppstå flere ganger samtidig over forskjellige dokumenter. Derfor er det en fordel at dokumentdatabaser er mulig å skalere databasen vertikalt. Dette betyr at man kan legge til flere servere i en klynge for å legge til rette for mer ressurser etter hvert som man ser behov for det.

- Embedded Documents

Man kan se på embedded documents som dokumenter inni dokumenter. Dette konseptet gjør det mulig for brukere å lagre data som er relaterte til hverandre i et enkelt dokument. Dette gjør det mulig for dokumentdatabaser å fungere uten å bruke joins. Dette påvirker i stor grad lesehastigheten for databasen i positiv retning, sammenlignet med relasjonsdatabaser.

- Sharding

Sharding er en prosess hvor man deler dokumentdatabasen opp i de forskjellige dokumentene den består av og distribuerer disse dokumentene på tvers av

servere. Ett dokument blir da regnet som en shard. Hvis serveren er konfigurert for å replikere data, vil denne sharden være tilgjengelig på flere servere.

Mer om - Dokumentdatabase

Som vi vet, er dokumentdatabaser en sentral del av NoSQL-teknologien. De tilbyr fleksibel datalagring, som kan være fordelaktig for moderne applikasjoner som kan dra nytte av skalerbarhet og mulighet for hurtige justeringer i tilfeller hvor behovene endrer seg. MongoDB har på mange måter satt standarden for hva som forventes fra implementasjoner av en dokumentdatabase. Til nå har vi sett grunnleggende bakgrunnsteori for dokumentdatabaser. Nå skal vi se nærmere på forskjellene mellom MongoDB sin dokumentasjon og teorien som nevnes i boken vi bruker i dette emnet – «NoSQL for Mere Mortals» skrevet av Dan Sullivan.

Forskjeller - teori og praksis, påvirkning av bruk

Fleksibilitet

Teorien i boken setter fokus på fleksibiliteten som kan tilbys av dokumentdatabaser. Man kan lagre dokumenter uten å definere en struktur på forhånd på hvordan disse dokumentene skal eksistere. Mens i praksis kan dette i mange tilfeller være et tveegget sverd. Man kan med MongoDB ende opp med inkonsekvenser i dataene hvis man ikke gjør dette på rett måte.

For eksempel kan vi ta for oss en nettbutikk. Vi skal lagre produktinformasjon i vår database.

```
{  
  "produktId": 123,  
  "productNavn": "T-skjorte",  
  "pris": 500.00  
}
```

En stund senere blir det bestemt at man skal legge til en beskrivelse og tilgjengelige størrelser for hvert produkt. Med MongoDBs dynamiske schema, kan man enkelt legge til disse feltene uten å endre schemaet for de eksisterende produktene. Vi får dermed dette resultatet:

```
{  
  "produktId": 456,  
  "productNavn": "Jeans",  
  "pris": 1500.00,  
  "beskrivelse": "En flott jeans",  
  "størrelser": "S, M, L, XL"
```



```
"beskrivelse": "Blå jeans",  
"stoerrelse": ["S", "M", "L", "XL"]  
}
```

Fordelene er at hvis man har MongoDB-databaser i et prosjekt hvor ting endres raskt, slik at databasen må reflektere dette, vil dynamiske schema fremskynde utviklingen av prosjektet på grunn av fleksibilitet. Ulempen oppstår når man får en rotete database og ikke har kontroll over fleksibiliteten. Dette ender til slutt med inkonsistens i data og vanskeligheter ved å utføre spørringer mot databasen

Skalerbarhet

Boken nevner også horisontal skalerbarhet av dokumentdatabaser ved at man distribuerer data over flere servere. Når man bruker MongoDB i praksis vil man ofte oppdage at optimal skalerbarhet krever en balanse mellom datamodell, infrastruktur og konfigurasjon – noe som er krevende.

Fordelene er at man kan ved å bruke sharding for å skalere vertikalt, oppnå større ytelse ved arbeid på store datasett. Ulempen er at dette krever en god forståelse for både data og bruken av dem. Man må blant annet konfigurere sharding-nøkler. Feil konfigurering av sharding kan redusere ytelsen – så her må man tenke seg nøye om.

Vi kan se tilbake på eksempelet som tar for seg en nettbutikk. Hvis vi ser for oss at nettbutikken har tatt seg opp i løpet av kort tid og nå inneholder ordre med flere millioner oppføringer, kan vi vurdere å implementere sharding og skalere vertikalt for å ta høyde for nettbutikkens vekst. Vi må først og fremst gjøre noen avgjørelser når det kommer til sharding.

1. Shard Key

- a. Først må vi velge Shard Key. Dette er det viktigste avgjørelsen man tar. Dette vil påvirke hvordan dataene blir distribuert over på flere servere. Etter å ha undersøkt, velger vi å bruke kundeID som nøkkel fordi nettbutikken selger internasjonalt og ordrene er jevnt fordelt blant kunder bosatt over hele verden.

2. Antall shards

- a. Vi bestemmer oss for å fordele data på tre ulike servere, slik:
 - i. Shard 1: kundeID fra 1 à 1.000.000

ii. Shard 2: kundeID fra 1.000.000 à 2.000.000

iii. Shard 3: kundeID fra 2.000.000 à 3.000.000

Nå ser vi at når en kunde legger inn sin ordre, vil databasen raskt identifisere hvilken shard ordren skal til basert på kundeID. Så vi ser at i et tilfelle hvor vi har en kundeID som er 2.123.456 så vil vi finne ordren på Shard 2.

Fordelen ved å gjøre dette er at siden hver shard bare håndtere en del av den totale datamengden databasen består av, vil vi oppnå høyere ytelse i form av raskere skrive-og-leseoperasjoner. Vi har også en umiddelbar fordel i det at hvis en shard skulle gå ned, vil ikke dette påvirke hele databasen. Ufordringene rundt dette er som tidlgiere nevnt balansering. Hvis en stor andel av de aktive kundene i nettbutikken i eksempelet skulle falle innenfor en bestemt shard, kan den serveren naturligvis som en konsekvens bli belastet mer. En annen utfordring vil være re-sharding hvis man ser at nettbutikken nok en gang har vokst seg stor nok.

Spørringer

Boken kan ha en tendens til å fremstille spørringer mot dokumentdatabaser som en enklere oppgave å utføre enn det de kanskje egentlig er. Spesielt liker boken å sammenligne spørringene mot dokumentdatabaser og relasjonsdatabaser. Ja – å kjøre spørringer mot en dokumentdatabase kan ved første øyekast se enkelt ut. Men mye av jobben er allerede gjort, ved at man på forhånd har designet dokumentdatabasen på grunnlag av hvilke spørringer man ønsker å kjøre mot den i fremtiden – dermed vil spørringene i enkelte tilfeller se enklere ut. Praktisk bruk av MongoDB vil avsløre en litt annen virkelighet enn den boken fremstiller.

Hvis vi tar for oss nettbutikken som et eksempel nok en gang, så ser vi fordeler ved at vi raskt kan hente ut ordre for en spesifikk kunde. Dette går raskt fordi vi har konfigurert sharding og MongoDB vet hvilken shard som skal besøkes ved å bruke sharding-nøkkelen i spørringen. Samtidig, hvis vi har indeksert kundeID, vil henteoperasjonen bli enda raskere siden databasen kan direkte kan finne riktig plassering på disken – noe som absolutt burde vurderes.

```
db.orders.find({ "kundeID": 2.123.456 })
```

Men la oss si at vi ønsker å gjøre spørringer som innebærer dataaggregering. Dette vil potensielt kunne by på problemer avhengig av faktorer. Som for eksempel hvor godt planlagt databasen er for denne typen spørringer. La oss ta en titt på en potensiell spørring:

```

db.orders.aggregate([
  {
    $match: {
      "ordreDato": { $gte: new Date(Date.now() - 7*24*60*60*1000) }
    }
  },
  {
    $group: {
      _id: "$produktKategori",
      avgOrdreVerdi: { $avg: "$ordreVerdi" }
    }
  }
])

```

Denne spørringen vil finne gjennomsnittsverdien av alle ordrer som er lagt inn av kunder den siste uken. Fordelt på produktkategori – eksempelvis t-skjorter, jeans, genser, jakker osv. Dette vil bli komplekst fordi spørringen først må filtrere gjennom alle mulige ordrer for uken. Deretter må den gruppere etter produktkategori, noe som krever beregning. Så må den lete over flere shards. Dette vil potensielt være krevende på grunn av det store datavolumet som kan være involvert i denne operasjonen.

Vi ser at selv om MongoDB støtter komplekse spørringer, kan de likevel være krevende å gjennomføre på en god måte. Man burde legge til grunn at man kjenner sine data godt og planlegger godt hvilke spørringer man skal kjøre mot database for å lykkes. Man burde også vurdere tiltak som for eksempel indeksering og ulike strategier for sharding.

Hvorfor finnes disse forskjellene?

Boken nevner for det meste veldig generelle brukssituasjoner, men det skal godt gjøres å dekke alle mulige tilfeller av utfordringer og scenarier som kan oppstå når man jobber med dokumentdatabaser og MongoDB. I virkeligheten må man ofte skreddersy løsninger som svar på et problem, i stedet for å tenke på best practice-tilfeller som boken presenterer.

MongoDB er designet for å være fleksibel nok slik at implementasjonen kan tilpasses de aller fleste behov. Og dette er nok grunnen til at vi ser disse forskjellene som er nevnt tidligere. MongoDB kan være passende å implementere i veldig mange tilfeller, men det kan alltid oppstå tilfeller hvor MongoDB ikke egner seg.

Samtidig, vil MongoDB fortsette å utvikle seg i takt med den teknologiske innovasjonen innen infrastruktur og hardware. Forskjellene mellom teori og praksis vil dermed potensielt bli mindre etter hvert som nyere versjoner blir lansert.

Kildeliste

Marius Geitle. (2023). "[Dokumentdatabaser, konsistens og CAP Teoremet.](#)" In *Forelesning*.

Høgskolen i Østfold.

What is nosql? NoSQL databases explained. (n.d.). MongoDB. Retrieved October 25, 2023, from <https://www.mongodb.com/nosql-explained>

What is MongoDB? — MongoDB Manual. (n.d.). Retrieved October 25, 2023, from <https://www.mongodb.com/docs/manual/>

Introduction to mongodb — mongodb manual. (n.d.). Retrieved October 25, 2023, from <https://www.mongodb.com/docs/manual/introduction/>

