FFI-RAPPORT

18/01676

Stordatasystemer og deres egenskaper

Audun Stolpe
Bjørn Jervell Hansen
Jonas Halvorsen

Stordatasystemer og deres egenskaper

Audun Stolpe
Bjørn Jervell Hansen
Jonas Halvorsen

Forsvarets forskningsinstitutt (FFI) 25. mars 2019

Emneord
Stordata
Dataanalyse
Informasjonssystemer
Informasjonsteknologi

FFI-rapport
18/01676

Prosjektnummer
1430

ISBN
P: 978-82-464-3176-5
E: 978-82-464-3177-2

Godkjennere
Espen Skjelland, *forskningsdirektør*
Ole-Erik Hedenstad, *forskningsleder*

Dokumentet er elektronisk godkjent og har derfor ikke håndskreven signatur.

(U) Sammendrag

Begrepet *stordata* er på mange måter mer dekkende som en betegnelse for en samfunnsutvikling enn for en bestemt teknologi, og har så langt ikke fått noen entydig definisjon. Isteden karakteriseres stordata som regel ved hjelp av de såkalte tre V'ene *Volume*, *Velocity* og *Variety*: Stordata er data av forskjelligartet natur (*Variety*), som kommer i store mengder (*Volume*) og/eller har hyppig oppdateringsfrekvens (*Velocity*) og som et resultat av dette ikke lar seg effektivt håndtere eller bearbeide ved hjelp av tradisjonelle metoder.

I denne rapporten tar vi for oss en samling egenskaper som vi mener karakteriserer systemer designet for å håndtere stordata:

- støtte for dataanalyse
- programmeringsmodell
- skalerbarhet
- tilgjengelighet vs. konsistens
- sårbarheter og feiltoleranse
- primærminnesystemer vs. persistenssystemer
- støtte for iterative beregninger,
- input/output-profil
- gjenbrukbarhet av data

Disse egenskapene kan tjene som en huskeliste over hvilke avveiinger man må gjøre når man skal designe en systemløsning for stordataproblemer.

Rapporten skisserer også fem hovedklasser av komponenter som benyttes når man komponererer stordataløsninger:

- tabulære databaser
- grafdatabaser
- strømmesystemer
- programmeringsrammeverk
- analyse- og visualiseringssystemer

Når man har et stordataproblem som fordrer systemstøtte, vil karakteristikaene til en slik løsning være avhengig av både problemet som skal løses og dataene som skal benyttes. Det finnes imidlertid et rikt utvalg av komponenter som kan settes sammen i henhold til disse karakteristikaene. Mange av disse komponentene er også fritt tilgjengelig, noe som gjør at design av stordataløsninger i stor grad kan betraktes som skreddersøm av åpne komponenter.

(U) Summary

The concept of *Big Data* remains elusive to define, yet is commonly characterized by the three Vs *Volume*, *Velocity*, and *Variety*. That is, a big data problem is typically indicated by data being so diverse (*Variety*), in such large quantities (*Volume*), and/or moving at such a speed (*Velocity*) that it doesn't lend itself to efficient processing using traditional methods and systems.

In this report we identify the following capabilities that together caracterize typical big data systems:

- support for data analytics
- programming model
- scalability
- availability vs. consistency
- vulnerabilities and fault tolerance
- in-memory vs. disk storage
- support for iterative computations
- input/output profile
- data reusability

The report proceeds to give an outline of the main design choices with respect to these capabilities, including associated trade-offs, which need to be considered when constructing big data systems.

The report also presents five main categories of big data system components:

- tabular databases
- graph databases
- streaming systems
- programming frameworks
- analysis and visualization systems

The main lesson is that the characteristics of a big data system is very much dependent on the problem at hand and the data available, for which there are numerous components available that can be put together accordingly. Many of these components are available as open source solutions, making designing big data solutions to a large extent tailoring open source components.

Innhold

1 Innledning

Denne rapporten er et første skritt i et kompetanseoppbyggingsarbeid på stordatasystemer- og utfordringer i tråd med FFIs strategiske satsing på temaet. Vekten er derfor lagt på teknologifronten innen området, og rapporten bør leses som en veileder til et informert teknologivalg snarere enn en behovsutredning mtp. stordatasystemer for militære formål. Teknologiene beskrevet i rapporten, og hvorvidt de er egnet for militære formål, vil i det videre bli studert gjennom eksperimenteringsaktivitet ved FFI.

Begrepet *stordata* er på mange måter mer dekkende som en betegnelse for en samfunnsutvikling enn for en bestemt teknologi. Denne samfunnsutviklingen kalles gjerne *informasjonssamfunnet*, et begrep som forbindes med et samfunn der produksjon, spredning og utnyttelse av informasjon er en vesentlig økonomisk, politisk og kulturell aktivitet. Drivkraften i denne utviklingen er selvsagt de digitale kommunikasjonsteknologiene generelt og Internett spesielt. Internett opptrer mer og mer som et globalt informasjonsreservoar der alt fra offentlige forvaltningsdata til data om individuelle borgere, f. eks. bevegelser og handlevaner, til syvende og sist munner ut. Som et resultat av dette har produksjonen av informasjon i samfunnet eksplodert, med dyptgripende konsekvenser for alle sektorer, bl. a. økonomi, utdannelse, helse, administrasjon, forsvar, o.a.

Denne utviklingen øker stadig i moment, og alle prognoser tilsier at den vil fortsette slik i mange år fremover. Det anslås at 90 % av all data som i dag er tilgjengelig over Internett er produsert i løpet av de siste to årene (Marketing 2017). Vekstraten er estimert til svimlende 2,5 exabytes om dagen (dvs. 2,5 trillioner eller 2×10^{18} bytes hver eneste dag). Sosiale medier alene står for enorme mengder data. Her er noen utvalgte tall:[1]

- Hver dag er nesten 1,3 milliarder mennesker aktive på Facebook (Socialbakers 2018*a*).
- YouTube vokser med mer enn 4 millioner timer med video hver dag (Micro Focus 2017).
- Twitter publiserer 500 millioner tweets om dagen (Socialbakers 2018*b*) – 82 % av brukerne er mobile.
- Instagram vokser med ca. 47 000 bilder i minuttet (Micro Focus 2017).

Disse tallene til tross; sosiale medier er kun en av mange innløpselver til det digitale datahavet, og brorparten av informasjonen som produseres i samfunnet tjener selvsagt andre hensikter.

En spesielt viktig kilde er den økende strømmen av (vanligvis stedsbestemt) informasjon som genereres av nettbrett, telefoner, kjørecomputere, radiofrekvenslesere, kameraer, satellitter, fly og en lang rekke andre ting som ikke er datamaskiner i en tradisjonell forstand. Denne utviklingen går under flere navn. Mest vanlig er tingenes internett (*the internet of things*), men allestedsnærværende databehandling (*ubiquitous computing*) og regnekraftige omgivelser (*ambient computing*) er også begreper som er i bruk.

Gartner har utarbeidet prognoser som tilsier at 8,4 milliarder nettbevisste gjenstander vil være i bruk i verden i løpet av 2017 (en 31 % økning fra 2016), et tall som vil stige til over 20 milliarder i 2020 (Gartner 2017). Her er noen tall brutt ned på typer av ting:

[1] Disse tallene er ikke kvalitetssikret og bør bare betraktes som en antydning av størrelsesorden.

- Ved begynnelsen av 2017 produserer mobile enheter omlag 8 exabytes (10^{18}) data årlig.
- International Data Corporation (IDC) spår 31% økning av kroppsnære (*wearable*) enheter som ur, pulsklokker, e-tekstiler og AR-briller fra 2016 – 2020. Det tilsier 82,5 millioner slike enheter i 2020.
- Business Insider spår at 75 % av alle biler som produseres i 2020 vil være utstyrt med nettbevisste kjørecomputere (Business Insider 2015).
- Trafikk til lands, til sjøs og i luften er direkte tilgjengelig over internett i sanntid gjennom satellittkommunikasjon, transpondere og kameraer. Skipstrafikken i Norge i 2016 genererte ca. 550 AIS-meldinger i sekundet.

Denne utviklingen er ikke begrenset til sivil sektor, men gjelder i aller høyeste grad også Forsvaret. Om ikke annet, fordi også Forsvaret benytter sivil teknologi som GPS-sensorer, mobiler og nettbrett, og fordi sivile kilder er viktige for etterretningsformål, situasjonsforståelse og bildebygging.

Men langt viktigere per i dag er selvsagt Forsvarets egne sensorplattformer. Spørsmålet om hvordan disse skal utnyttes best mulig blir stadig mer presserende ettersom dataproduksjonen er i ferd med å eksplodere også på dette området. For eksempel:

- Videostrømmene fra cockpiten i den norske F-35 flåten genererer ca. 30 – 35 terabyte (10^{12}) i året – og dette er kun én av mange sensorer i en F-35.
- Hydrofondata fra ubåt utgjør typisk rundt 1 terabyte per døgn
- Omlag 7 milliarder AIS-meldinger passerer gjennom norske basestasjoner og satellitter hvert år.
- En enkelt drone i NATOs Allied Ground Surveillance (AGS)-program er ventet å produsere ca. 12 terabyte data per flyvning.

I tillegg til all denne, la oss kalle den, dynamiske sanntidsinformasjonen, er det selvsagt fortsatt slik at svært mye informasjon som produseres av og for Forsvaret ligger lagret i tradisjonelle fagsystemer og registre. Dette gjelder både strukturert informasjon (slik som i en ORBAT-database, Forsvarets Helseregister eller Forsvarets HMS-rapporteringssystem), og ustrukturert informasjon som det vanligvis er mye mer av (f.eks. rapporter, satellittbilder, videoopptak og sonardata).

1.1 De tre V-ene

Vi har så langt sagt at begrepet stordata kan forstås som en betegnelse på en samfunnsutvikling, men det kan også forstås som et navn på enkelte typer ulike, men innbyrdes relaterte informasjonsbehandlingsproblemer. Det er vanskelig å gi en mer presis definisjon av stordatabegrepet enn dette, og enda vanskeligere å gi en definisjon av stordata*systemer*. Begrepet brukes da også ganske forskjellig av konsulenter, programvarehus og forskere.

Det finnes imidlertid et minste felles multiplum som de aller fleste er enige om. Dette er en liste av egenskaper som, siden den ble lansert i Laney (2001), er blitt kjent som "*the three V's of big data*".

Volum

Størrelsen på et datasett er det mest opplagte kjenntegnet ved et stordataproblem. Man kan, vel å merke, ikke operere med absolutte tall her, men må definere størrelse relativt til en gitt kapasitetsbegrensning i minne og prosessorkraft. Dersom mer enn én maskin er nødvendig for å lagre og behandle et datasett, så er datasettet 'stort' for alle praktiske formål, og derfor stort i den intenderte tekniske forstanden.

Per definisjon henger derfor stordata nært sammen med parallell og distribuert databehandling. Siden behovet for regnekraft og minne overgår det én enkelt maskin kan tilby, dreier stordata seg i stor grad om å koordinere ressurser over en klynge (*cluster*) av maskiner. Algoritmer som er i stand til å bryte opp en beregningsoppgave i mindre, uavhengige delproblemer er en avgjørende del av dette.

Velocity (hastighet)

En datamengde bør også regnes som stor dersom informasjonen genereres svært raskt og skal behandles i sanntid. Typiske eksempler på slike systemer vil være systemer for hendelsesdeteksjon og bildebygging. I slike systemer flyter gjerne informasjonen inn fra flere kilder samtidig, og må analyseres fortløpende for at forståelse av den gitte situasjonen skal holde seg så fersk som mulig.

Legg merke til at dette er en definisjon av størrelse som ikke først og fremst er relativ til lagringskapasitet, men til det man kan kalle evnen til å absorbere en strøm. Et system er i stand til å absorbere en strøm av data dersom alle planlagte analyseoppgaver og informasjonssøk returnerer umiddelbart og med oppdatert informasjon. Begge disse kravene er nødvendige da det andre av dem opplagt avhenger av det første.

Også høyhastighets datastrømmer krever en distribuert prosesseringsmodell. Det er generelt fordelaktig å skyve det som kan gjøres av preprosessering så langt ned langs (data-)rørledningen som mulig, og det er nødvendig med betydelig redundans i systemet som beskyttelse mot brudd.

Variety (heterogenitet)

Stordatasystemer henter vanligvis data fra mange ulike og i utgangspunktet uavhengige kilder. Dataene kan komme fra lokale systemer, slik som regneark eller databaser, fra vanlige tekstfiler, fra sosiale mediestrømmer og fra fysiske sensorer slik som kameraer, GPS satellitter m.m. Formatene og medietypene vil vanligvis variere mye, og de vil vanligvis være utviklet uavhengig av hverandre for uavhengige formål.

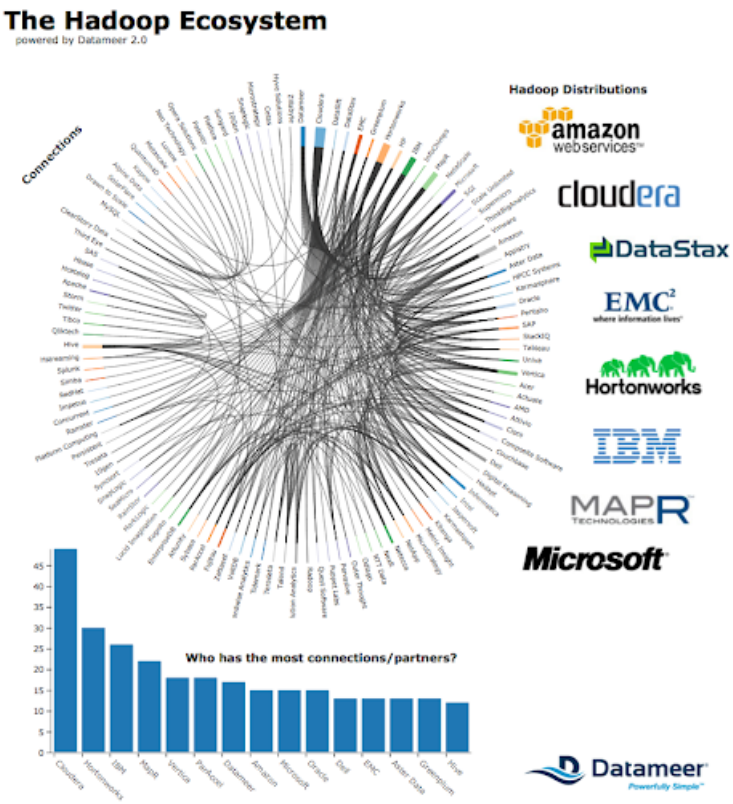
Et stordatasystem må være i stand til å utvinne verdifull informasjon ved å konsolidere slike svært heterogene kilder – dvs. ved å omforene ulike typer data. For eksempel er det ikke uvanlig for et stordatasystem å sammenstille bildefiler og videostrømmer med tekstfiler og serverlogger.

Informasjonen som flyter gjennom et stordatasystem vil derfor ofte måtte raffineres gjennom en serie steg som omfatter søk, harmonisering, analyse og visualisering. Hvert av disse stegene er som regel støttet av avanserte teknikker og algoritmer fra relaterte disipliner som rubriseres under fellesbetegnelsen datavitenskap. Eksempler er statistisk modellering, maskinlæring, naturlig språk, maskinoversettelse og prediktiv analytikk.

1.2 Om denne rapporten

Antallet databaser, programvarebiblioteker, visualiseringsverktøy etc. som markedsfører seg som stordatasystemer, i en eller annen forstand av ordet, er allerede svært stort og hurtig økende.

Bare Apache Hadoop-økosystemet inneholder minst 120 ulike verktøy (jf. figur 1.1). Apache Hadoop er et programvarerammeverk for distribuert lagring og prosessering av store datamenger med programmeringsmodellen MapReduce (jf. kapittel 2.2), og det som vanligvis kalles Apache Hadoop-økosystemet (jf. figur 1.1) er en samling av programvarebiblioteker, tilbehør og verktøy som er bygget over eller i tilslutning til Hadoop. Vi beskriver Hadoop nærmer i kapittel 3.4.1.



Figur 1.1 Hadoopøkosystemet.[2]

Og MapReduce er bare én type system. Dersom man flytter blikket over på for eksempel strømmesystemer (jf. kapittel 3.3), er mangfoldet også påfallende stort. Allerede i en mye sitert "*state-of-the-art*"-undersøkelse fra 2012 (Cugola & Margara 2012) beskriver og sammenlikner forfatterne 34 forskjellige systemer for strømresonnering og kompleks hendelsesdeteksjon (*complex event detection*) som de regner som modne og stabile programvareprosjekter.

Det sier seg selv at det ikke er noe poeng i å forsøke å skrive en "*state-of-the-art*"-rapport som gir en komplett innholdsfortegnelse, og langt mindre en punkt-for-punkt sammenligning av alle disse systemene.

[2] https://www.datameer.com

I denne rapporten har vi derfor valgt å legge oss på en annen linje. Den består av to trinn. Først forsøker vi å ta stilling til hvilke avveininger det er viktig å ta hensyn til før man velger en stordatateknologi. Mer spesifikt, presenterer vi en liste av egenskaper som beskriver *oppførselen* til et distribuert system med henblikk på slike ting som feiltoleranse, sårbarhet, konsistens og ytelsesprofil (kapittel 2).

Dernest sorterer vi stordatasystemer inn i fem hovedklasser, beskriver disse klassenes karakteristiske egenskaper i henhold til listen i kapittel 2, og beskriver noen utvalgte systemer fra hver klasse. De fem typene vi landet på er:

- Tabulære databaser: databaser som til syvende og sist er basert på å gjøre oppslag på nøkler,
- Grafdatabaser: databaser hvor informasjon kodifiseres i semantiske (i en vid forstand) nettverk,
- Strømmesystemer: sanntidssystemer som tolker data fortløpende,
- Programmeringsrammeverk: en programvarestabel og et API for å utvikle stordataprogrammer, og
- Analyse- og visualiseringssystemer: systemer som tolker data med avanserte algoritmer og presenterer resultatene i forskjellige diagrammer og plott.

Vi skisserer i tillegg fire militære anvendelser, som bevisst er valgt ut for å illustrere noen av de egenskapene eller aksene som er listet i kapittel 2, og som denne rapporten er strukturert rundt. Det finnes utvilsomt mange andre.

Å lese denne rapporten i sin helhet er mest relevant for lesere med faglig interesse for temaet. Lesere som er ute etter en oversikt over temaet, kan lese kapitlene 1 og 2, innledningen til kapittel 3 samt kapitlene 4 og 5.

2 Viktige egenskaper ved et stordatasystem

Selv om ingen er spesielt ivrige etter å forplikte seg til en endelig definisjon av begrepet stordata, er de tre V'ene presentert i kapittel 1.1 etterhvert blitt en de facto standard: "Big data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation." (Gartner IT Glossary 2018). Med andre ord dreier et stordataproblem seg ikke nødvendigvis bare om datamengder, men også om opprinnelse, betydning, formater og hastigheter.

> Big Data consists of extensive datasets – primarily in the characteristics of volume, variety, velocity, and/or variability – that require a scalable architecture for efficient storage, manipulation, and analysis.[3]

Implisitt i denne karakteriseringen ligger det åpenbart en erkjennelse av at et stordataproblem ikke er én bestemt ting. Begrepet 'stordata' må snarere forstås som en samlebetegnelse for beregningsoppgaver som enten er for komplekse eller som vokser for raskt til at de kan håndteres av én enkelt maskin. I dette perspektiver er et stordataproblem simpelthen en oppgave som fordrer et parallelt og distribuert prosesseringsparadigme for å kunne løses effektivt.

Det er også verdt å merke seg at det her fokuseres på manipulering og analyse av disse dataene. I dette ligger en påminnelse om at stordata først har verdi når de blir utnyttet.

Det er vanskelig å si mer spesifikt enn dette hva et stordatasystem er. Det man kan si med én gang er at det ikke finnes noen generisk programvaresuite som kan anvendes på et hvilket som helst problem uten noen form for tilpasning. Tvertimot må et valg av programvarepakke tuftes på en inngående analyse og forståelse av hvilket problem det er man forsøker å løse: Hva slags data dreier det seg om; komplekse eller enkle, homogene eller heterogene? Hva slags algoritmer ser du for deg å bruke; maskinlæringsalgoritmer, sosiale nettverksalgoritmer? Hvor oppdaterte trenger dataene å være? Er noe nedetid akseptabelt?

Med andre ord, å velge rett programvare for et bestemt stordataproblem er ikke en triviell oppgave, men krever at man på bakgrunn av en forståelse av problemet har en noenlunde klar tanke om hvilke egenskaper man ønsker at systemet skal ha.

I dette kapitlet har vi laget en liste over noen av de egenskaper vi betrakter som spesielt viktige. Ulike valg her vil gi store utslag i hva slags system man ender opp med; hvordan det oppfører seg, hva det er velegnet til og hva det er uegnet til. Listen pretenderer ikke å være komplett.

2.1 Støtte for dataanalyse

Å oppdage mønstre, foreslå konklusjoner og støtte beslutningsprosesser – mer generelt å foredle rådata til nyttig informasjon – er selvsagt det endelige siktemålet for de aller fleste anvendelser av

[3] http://www.nist.gov/itl/bigdata/bigdatainfo.cfm

stordatasystemer. Dataanalyse er i dag en samlebetegnelse som omfatter et stort antall av teknikker fra svært forskjellige disipliner slik som sosiologi, statistikk, diskret matematikk og logikk. Nedenfor følger et utvalg:

Prediktiv analytikk omfatter teknikker fra statistikk, maskinlæring og datautvinning (*data mining*) for å forutse trender basert på en analyse av historiske data. Modellene spenner fra regresjonsanalyse av enkle lineære trender, til avansert multivariat statistikk som benyttes i nevrale nettverk. SAP Predictive Analytics [4] og Apache Mahout [5] er mye brukt.

Hendelsesbehandling (*event processing*) står for et sett av teknikker for å identifisere og behandle hendelser i en kontinuerlig strøm av sanntids- eller nær sanntidsdata. Hensikten er å kunne abstrahere meningsfulle situasjoner fra lavnivå datastrømmer for å respondere raskt og fortløpende. Hendelsesdeteksjon omfatter statistiske såvel som logikkbaserte teknikker. Eksempler på slike systemer er Cayuga[6] og IBM System S[7].

Virksomhetsetteretning (*business intelligence*) er en samlebetegnelse for strategier og teknologier som gir historiske, nåværende og prediktive syn på en virksomhet, ofte i form av statistikk over produksjonstall, omsetning, logistikk o.l. Fellesfunksjoner for slike systemer som spesialiserer seg på virksomhetsetterretning inkluderer rapportering, analyse (ofte i sanntid) av rapporteringskuber, hendelsesdeteksjon, resultatstyring/planlegging o.a. Slike systemer er ofte designet for å kunne sammenstille og utnytte både strukturerte og ustrukturerte data og støttes ofte av en varehusmodell slik som for eksempel en datainnsjø (*data lake*). Sisense[8] og Dundas[9] er eksempler på systemer som spesialiserer seg på dette feltet.

Sosial nettverksanalyse (SNA) er en tverrfaglig disiplin basert på grafteori, statistikk, spillteori og sannsynlighetsregning som studerer organisering og evolusjon av grupper samt posisjonen og innflytelsen til enkeltpersoner. SNA omfatter i dag algoritmer slik som identifikasjon av nettsamfunn (*community detection*), beregning av innflytelse og sentralitet (*betweenness* og *centrality*), avsløring av rykter og sporing av ryktespredning, sentimentanalyse og kartlegging av informasjonsflyt og sårbarheter. Gephi[10] og ORA[11] er eksempler på populære SNA-verktøy.

Automatisk resonnering er et felt i skjæringspunktet mellom informatikk og matematisk logikk som studerer de algoritmiske egenskapene til forskjellige former for deduktiv logikk. Automatisk resonnering kan brukes til å utlede nye data fra historiske data basert på en logisk beskrivelse av problemområdet. Innenfor stordatafeltet brukes automatisk resonnering mest til å harmonisere heterogene datakilder, såkalt OBDI (*ontology based data integration*), og til å forsterke informasjonssøk, såkalt OBDA (*ontology based data access*) (Calvanese et al. 2009). RDFox[12] og Ontop[13] er velkjente eksempler på slike systemer.

Cybersikkerhet: SIEM og UBA. SIEM (*security information and event management*) er et voksende felt innenfor cybersikkerhet som dreier seg om å lagre, analysere og korrelere sikkerhetsrelevant

[4] https://www.sap.com/products/analytics/predictive-analytics.html
[5] https://mahout.apache.org/
[6] http://www.ccs.neu.edu/home/mirek/OldProjects/Cayuga/index.htm
[7] https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534
[8] https://www.softwareadvice.com/bi/prism-profile/
[9] https://www.softwareadvice.com/bi/dundas-bi-profile/
[10] https://gephi.org/
[11] http://netanomics.com/
[12] https://www.oxfordsemantic.tech/
[13] http://ontop.inf.unibz.it/

informasjon slik som autentiseringshendelser, antivirus-hendelser, innbrudd o.a. Et SIEM-system assisterer analytikeren i å etablere en normaltilstand for nettverket som overvåkes, og å identifisere avvik. SIEM-systemer brukes derfor ofte som støtteverktøy for å oppfylle regler og forskrifter for IKT-sikkerhet.

UBA (*user behaviour analytics*) skiller seg fra SIEM ved å fokusere mindre på systemhendelser, og mer på brukeraktiviteter. Et typisk UBA-system benytter en kombinasjon av regelbasert AI, maskinlæring og statistikk for å bygge en profil som etablerer normal adferd for en bestemt bruker. Betydelige uregelmessigheter knyttet til f.eks. nettaktivitet, hvilke applikasjoner personen starter, hvilke filer han eller hun åpner etc. kan deretter rapporteres som potensielle trusler. UBA brukes ofte som et komplement til SIEM for å styrke vernet mot trusler fra innsiden.

Stordatasystemer som spesialiserer seg på cybersikkerhet, eksempelvis Cynet[14], BayDynamics[15] og Splunk[16], tilbyr som regel både SIEM- og UBA-funksjonalitet.

2.2 Programmeringsmodell

En programmeringsmodell kan betraktes som den grunnleggende stilen eller idiomet som et rammeverk legger opp til at utviklere skal uttrykke seg i. Programmeringsmodellen tilbyr et sett av abstraksjoner og datastrukturer som lar utvikleren beskrive et problem på en måte som kan oversettes til en parallell beregning over en maskinklynge. Den fungerer med andre ord som en bro mellom utviklerens problem og det underliggende distribuerte systemet, og er designet for å oppnå økt utviklerproduktivitet, ytelse og overførbarhet til beslektede problemer og beslektede systemer. Det er ingen nødvendig sammenheng mellom en programmeringsmodell og datastrukturene et stordatasystem er bygget over, men det er heller ikke uvanlig at de er avstemt i forhold til hverandre. F. eks. vil en distribuert database som lagrer data i tabeller ofte, men ikke alltid, tilby en programmeringsmodell som er drevet av et deklarativt spørrespråk, typisk en variant av SQL. Utover dette nøyer vi oss med å beskrive to eksempler her. Se Wu et al. (2017) for en mer utfyllende liste:

Funksjonell programmering er i ferd med å bli normen i stordataverdenen med rammeverkene Apache Spark og Apache Flink som flaggskip. Det er spesielt to konstruksjoner som gjør funksjonell programmering til et velegnet idiom for å uttrykke parallelle og distribuerte beregninger:

- Førsteklasses funksjoner: er funksjoner som kan holdes i variabler og utveksles mellom prosesser.
- Høyereordens funksjoner: funksjoner som aksepterer andre funksjoner som input og som kan returnere funksjoner som output.

Disse to aspektene til sammen gjør at en distribuert beregning kan modelleres på en deklarativ og intuitiv måte. Førsteklassesfunksjoner spesifiserer hvilke beregninger som skal sendes til de ulike maskinene i en maskinklynge, mens høyereordens funksjoner spesifiserer hvordan resultatene

[14] https://www.cynet.com/about-us/
[15] https://baydynamics.com/
[16] https://www.splunk.com/

skal kombineres. Det mest velkjente eksempelet på en slik modell er MapReduce hvor en *map*-funksjon brukes for å transformere data på flere maskiner samtidig og en *reduce*-funksjon brukes til å aggregere og kombinere resultatene. Navnet MapReduce refererer opprinnelig til proprietær Google-teknologi, men har siden blitt assosiert med Apache Hadoop som er en åpen implementasjon av den samme teknologien.

Aktørmodellen (*actor model*) er en meldingsorientert datamodell for parallell prosessering hvor uavhengige beregninger modelleres som *aktører* som kan kommunisere med hverandre ved å sende hverandre beskjeder. Modellen er ofte brukt i grafdatabaser hvor noder er aktører i denne forstanden. Et konkret eksempel er programmeringsmodellen *Gather-Apply-Scatter* hvor utvikleren implementerer tre funksjoner for hver node i en graf: *gather*-funksjonen samler beskjeder fra nabonoder, *apply*-funksjonen oppdaterer tilstand basert på beskjedene som er mottatt, og *scatter*-funksjonen sender nye beskjeder basert på den nye tilstanden. Utveksling av beskjeder skjer parallelt, men samles opp i runder (maks én beskjed fra én node til en annen i én runde) kalt *supersteg* som i seg selv utføres som en sekvens. *Gather-Apply-Scatter*-modellen rubriseres derfor vanligvis som en *Bulk-Synchronous-Parallel*-modell, hvilket betyr at den er sekvensiell betraktet som en serie med steg, men parallell innenfor hvert av dem (Junghanns et al. 2017*a*). Eksempler på systemer er grafdatabasen Google Pregel[17] og dens åpne variant Apache Giraph[18].

Valget av programmeringsmodell har stor betydning for alt fra produktivitet til vedlikehold og kunnskapsoverføring, og bør vurderes på bakgrunn av hvilket problem som skal løses samt tilgjengelig kompetanse.

2.3 Skalerbarhet

Skalerbarhet defineres gjerne som evnen et system har til å håndtere en økende mengde arbeid, eller evnen systemet selv har til å vokse for å imøtekomme en slik økning. Med 'en økende mengde arbeid' mener man typisk mer data og/eller behovet for mer regnekraft, men begrepet kan, og i mange sammenhenger bør, forstås videre enn dette. Generelt kan man skille mellom de følgende betydningene:

Skalerbarhet i datamengde og regnekraft (*load scalability*). Som nevnt allerede dreier dette seg om å øke tilgangen på minne og regnekraft for å imøtekomme tyngre arbeidsoppgaver og større input. Det er vanlig å skille mellom *horisontal* og *vertikalt* skalerbarhet i denne forstanden. Å skalere *vertikalt* betyr å øke regnekraften og/eller lagringskapasiteten til en enkelt maskin f.eks. ved å øke antallet prosessorer. Å skalere *horisontalt* betyr å øke antallet maskiner i en maskinklynge. Stordatasystemer er nesten uten unntak og per definisjon horisontalt skalerbare.

Funksjonell skalerbarhet. Også her kan man skille mellom to underkategorierer: det man kan kalle *utvidbarhet* (*extensibility*) dreier seg om å kunne legge til ny funksjonalitet med minimal kostnad. Et system som er skalerbart i denne forstanden tilbyr typisk et åpent programmeringsgrensesnitt eller en plug-in-arkitektur. Et eksempel på et system som er utvidbart i denne forstanden er analysesystemet Semantica Pro[19]. Den andre underkategorien består av rammeverk og systemer som er *multimodale* i

[17] https://blog.acolyer.org/2015/05/26/pregel-a-system-for-large-scale-graph-processing/
[18] http://giraph.apache.org/
[19] http://www.semanticresearch.com/semantica-pro

den forstand at de støtter flere forskjellige programmeringsmodeller og datastrukturer. Et multimodalt rammeverk vil la deg representere data i grafer, aggregater, tabeller og strømmer, samt å bevege data mellom dem uten behov for å bytte verktøy. Moderne programmeringsrammeverk som Apache Spark[20] og Apache Flink[21] er multimodale i denne forstanden, mens Google Pregel, Apache Giraph eller Neo4j[22], som eksempler på det motsatte, er dedikerte grafdatabaser.

Organisatorisk skalerbarhet. Et system kan sies å være organisatorisk skalerbart dersom det er designet for at ulike grupper av brukere med ulike privilegier og ulik innsynsrett skal kunne samhandle på ett enkelt distribuert system. I praksis betyr dette gjerne at systemet implementerer en eller annen form for rollebasert aksesskontroll og/eller lagdeler applikasjonen i graderingsnivåer. Et eksempel på førstnevnte er Apache Accumulo[23] som er en distribuert nøkkel/verdi-database basert på BigTable-teknologi[24] fra Google. Et eksempel på sistnevnte er analysesystemet Semantica Pro[25].

Skalerbarhet på tvers av generasjoner. Et system som er designet for å kunne lese og forstå arkiverte data selv etter at nyere versjoner av systemet lanseres (eller nyere versjoner av programvaren det avhenger av), er skalerbart i denne forstanden. I den grad man kan si noe generelt om problemet, dreier dette seg om å gjøre applikasjoner mest mulig generiske og datasettene mest mulig selvbeskrivende. RDF-baserte systemer (se kapittel 3.2.2) slik som Amazon Neptune[26], Semantica Pro og Stardog[27] vil ofte være skalerbare på tvers av generasjoner i denne forstanden.

2.4 Tilgjengelighet vs. konsistens

En viktig egenskap som bør tas i betraktning når man velger et stordatasystem er det som gjerne kalles systemets CAP-profil hvor CAP står for *Consistency*, *Availability* og *Partition tolerance*. Disse respektive egenskapene kan forklares slik:

- Konsistens (C): betyr at alle maskinene i en klynge ser de samme dataene samtidig.
- Tilgjengelighet (A): betyr at systemet responderer innen rimelig tid på enhver forespørsel.
- Robusthet (P) (*Partition tolerance*): betyr at systemet fortsetter å fungere normalt også når noen av maskinene i klyngen ikke lenger kan nås (f.eks. som en følge av nettverksfeil eller strømbrudd).

Et velkjent teorem kalt Brewer's teorem, presentert som CAP-teoremet i Gilbert & Lynch (2002), viser at ingen parallelle distribuerte systemer kan garantere alle disse tre egenskapene samtidig. Mer spesifikt sier teoremet at et distribuert system alltid må velge å ofre konsistens eller tilgjengelighet når nettverket fragmenteres.

Ulike stordatasystemer inngår ulike kompromisser når det gjelder CAP-egenskapene, avhengig av hvilken bruk systemet er designet for. Det er viktig å vite hvilket kompromiss dette er når man velger

[20] https://spark.apache.org/
[21] https://flink.apache.org/
[22] https://neo4j.com/
[23] https://accumulo.apache.org/
[24] https://cloud.google.com/bigtable/
[25] https://www.semanticresearch.com/semantica-pro
[26] https://aws.amazon.com/neptune/
[27] https://www.stardog.com/

stordataløsning siden det i stor grad styrer oppførselen til systemet: dersom du trenger å kunne garantere at dataene dine er konsistente og at klyngen tåler maskiner som feiler eller forsvinner, kan du ikke samtidig garantere at systemet vil være 100% responsivt til enhver tid. Det omvendte gjelder også: dersom du behøver tilgjengelighet og robusthet, må du være forberedt på å leve med at forskjellige versjoner av dataene kan være i omløp i systemet – ihvertfall midlertidig.[28]

2.4.1 ACID vs. BASE

I relasjonsdatabaser står ACID (Atomicity, Consistency, Isolation, Durability) for et sett med egenskaper som er ment å garantere gyldigheten av en lese- eller skriveoperasjon, selv om mange forskjellige brukere arbeider mot den samme databasen samtidig, og selv i tilfelle feil, strømbrudd etc.

En lese eller skriveoperasjon i en relasjonsdatabase involverer ofte flere tabeller, typisk pga. fremmednøkler og joins. ACID-egenskapene sikrer at en slik kompleks operasjon behandles som en logisk enhet. *Atomicity*, for eksempel, garanterer at en kompleks operasjon enten lykkes helt eller ignoreres: dersom deler av operasjonen ikke kan fullføres så svikter hele og databasen forblir uendret. Konsistens (*consistency*), på den annen side, sikrer at en operasjonen bringer databasen fra én gyldig tilstand til en annen. Det vil si at alle data som skrives til databasen må være gyldige i henhold til skjema og funksjonelle avhengigheter etc., og at dataene alltid er konsistente i den forstand at ingen klienter kan lese en gammel verdi som allerede er endret av en annen (også kalt *sterk konsistens* eller *serialiserbarhet*). I databaselitteraturen kalles en slik ACID-sertifisert operasjon gjerne en *transaksjon*. En overføring av midler fra en bankkonto til en annen er et vanlig eksempel på en ACID-transaksjon. Transaksjonen innebærer flere mindre lese- og skriveoperasjoner, slik som debitering av en konto og kreditering av en annen, som enten må fullføres i sin helhet eller ignoreres fullstendig.

Som et slags polemisk kontrapunkt til ACID, beskriver mange stordatasystemer seg selv som BASE-systemer, en forkortelse for *Basically available*, *Soft state*, *Eventually consistent* (Vogels 2009). Sammenliknet med ACID er BASE en svak konsistensmodell som ikke garanterer at kun én versjon av dataene er i omløp til enhver tid. Den garanterer snarere at systemet vil *konvergere* mot samme versjon over tid, og derfor at konsistens vil gjenopprettes til syvende og sist (*eventually*). Man kan betrakte denne modellen som en måte å bytte bort sterk konsistens for responsivitet.

2.5 Sårbarheter og feiltoleranse

Generelt kan man si at et system er feiltolerant dersom det kan velge å degradere fremfor å avslutte dersom én eller flere komponenter svikter. Degradering kan her bety flere ting. Det kan bety at data

[28] Det bør nevnes at det er en del uenighet (jf. https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html) om nytteverdien av CAP-teoremet. Mange hevder at CAP-teoremet gir en så forenklet fremstilling av distribuerte systemer at det ikke sier noe vesentlig om hvordan virkelige systemer oppfører seg. De fleste aksepterer gyldigheten av diskusjonen som CAP-teoremet hadde til hensikt å starte, om avveininger av tilgjengelighet- og konsistensegenskaper i distribuerte systemer, men argumenterer for at teoremet må raffineres for å reflektere dagens systemflora. PACEaLC er en annen modell som har blitt foreslått, som i tillegg til konsistens og tilgjengelighet, også faktorerer inn forsinkelse (*latency*).

går tapt, men at systemet fortsetter som normalt over det resterende datasettet. Det kan bety lengre beregningstid og/eller responstid, og det kan bety inkonsistens; forskjellige versjoner av datasettet er synlige for forskjellige brukere, samtidige oppdateringer kan komme i konflikt osv.

En feilhåndteringsstrategi tar som regel sikte på et bestemt blandingsforhold mellom de tidligere nevnte CAP-egenskapene, og er vanligvis bygget inn i selve systemarkitekturen til et stordatasystem i form av en algoritme for replisering av spredte data og beregninger. Tre vanlige arkitekturer er:

Master/slave. En master/slave-arkitektur har én maskin eller prosess, la oss kalle den *drivernoden*, som kontrollerer og koordinerer alle andre, la oss kalle dem *arbeiderne*. Drivernoden delegerer arbeid og data til resten av maskinklyngen, den vedlikeholder metadata, f.eks. en indeks over innholdet på de forskjellige arbeiderne, og den regulerer skrive og lesetilgang til det distribuerte filsystemet. En master/slave-arkitektur utnytter disse metadataene bl.a. til å vedlikeholde konsistens i CAP-forstand, men har til gjengjeld ett enkelt sårbarhetspunkt; dersom drivernoden svikter så er i praksis all informasjonen i klyngen tapt. Det er derfor ikke helt galt å si at master/slave-systemer velger konsistens fremfor robusthet. Et velkjent eksempel på et slikt system er Hadoop som beskrives nærmere i kapittel 3.4.1.

Peer-to-peer. I en slik arkitektur er det ingen koordinerende node. Alle noder er like mye verdt og hver av dem kan håndtere forespørsler fra en klient. Denne arkitekturen har den fordelen at nye noder kan legges til klyngen sømløst, uten behov for å oppdatere en indeks eller et sett av metadata mer generelt. Det finnes heller ikke noe spesielt sårbarhetspunkt og systemet vil i prinsippet kunne fortsette å være tilgjengelig så lenge noen maskiner i klyngen fungerer som normalt. Imidlertid er det ingen koordinerende node som kontrollerer skrivetilgang, slik som i et master/slave-system. Noe forenklet kan man si at en peer-to-peer-arkitektur vektlegger tilgjengelighet og robusthet fremfor konsistens. Et eksempel på et slikt system er Cassandra[29].

Avstamningsgrafer (*lineage graphs*). *Avstamning* er et konsept for feilhåndtering som er basert på å lagre funksjonelle avhengigheter mellom datasett snarere enn datasettene selv. Når en funksjon appliseres til et datasett lagrer ikke systemet resultatet, men kun en peker til det opprinnelige datasettet sammen med den nevnte funksjonen. Grafen som består av slike pekere og funksjoner, kalles en *avstamningsgraf*. Avstamningsgrafen er nyttig dersom deler av nettverket feiler, fordi systemet kan rekalkulere nødvendige data fra avstamningsgrafen snarere enn å kopiere data på mange forskjellige maskiner. På grunn av at avstamningsgrafer gir minimal datareplisering brukes de i primærminne-(*in-memory*)systemer, dvs. i systemer som benytter hovedminne for datalagring. Dersom ikke dataene persisteres på en annen måte, kan de derfor i prinsippet gå tapt dersom maskiner i klyngen mister strøm. Et eksempel på et system som benytter avstamningsgrafer er Apache Spark som er nærmere beskrevet i kapittel 3.4.2.

Det er viktig å ta stilling til hvilke sårbarheter man velger, samt hvordan man kompenserer for dem, idet man velger et stordatasystem for en bestemt oppgave: et primærminne-system kan og bør suppleres med en lagringsløsning som persisterer beregningsgrunnlaget, et master/slave-system kan gjøres mindre sårbart ved å kopiere drivernoden etc.

[29] http://cassandra.apache.org/

2.6 Primærminnesystemer vs. persistenssystemer

Det går et viktig skille mellom stordatasystemer som lagrer data i hovedminnet, på den ene siden, og systemer som skriver data til disk på den andre. Vi vil i det følgende kalle den første gruppen for primærminnesystemer (*in-memory systems*) og den andre for persistenssystemer, men understreker at disse merkelappene kun er ment for å kontrastere systemer utifra denne ene egenskapen.

Et persistenssystem er et datalagringssystem slik vi vanligvis forstår det, dvs. et system som skriver og leser til og fra en eller flere harddisker i en maskinklynge, til og fra en redundant matrise av uavhengige disker (RAID) eller noe liknende. Poenget er at dataene lagres på et ikke-flyktig medium som tåler strømbrudd, dvs. *persisteres*.

En primærminnedatabase (*in-memory database*) er et datalagringssystem som ikke skriver data til filer, men holder dem i primærminnet (RAM) så lenge prosessen er aktiv. Primærminnedatabaser er raskere enn persistensdatabaser fordi lese- og skriveoperasjoner tar mye kortere tid. Primærminnesystemer kan derfor være et nødvendig valg for applikasjoner der responstid er avgjørende.

Primærminnesystemer har utviklet seg hurtig, spesielt innenfor stordatafeltet, siden midten av 2000-tallet, mye på grunn av fremveksten av multiprosessorarkitekturer som kan adressere store minneområder, og på grunn av fallende priser på RAM.

Primærminnesystemer har den opplagte sårbarheten at RAM er flyktig. Dersom strømmen av en eller annen grunn forsvinner så er dataene borte. Det utvikles imidlertid stadig nye transistorbaserte former for minne slik som Flashminne, som har den egenskapen at de kan lagre data selv etter at strømtilførselen er slått av.

Enkelte eksempler på primærminnesystemer på stordataområdet er VoltDB[30], MemSQL[31] og Apache Ignite[32] som er distribuerte relasjonsdatabaser designet for sanntidsanalyse av hurtige data, og Redis[33] som er en distribuert nøkkel/verdi-database som kan lagre datastrukturer slik som lister, mengder og bitmaps.

2.7 Støtte for iterative beregninger

Å lese og skrive til disk er en svært kostbar operasjon tidsmessig. Målt i nanosekunder tar det omtrent hundre ganger så lang tid å lese 1 MB sekvensielt fra harddisk som fra RAM. Dette er en kostnad som har stor kumulativ effekt for *iterative beregningsoppgaver*.

En iterativ beregning er en prosess som består av en serie steg der hvert mellomresultat er avledet fra det foregående og utgjør beregningsgrunnlaget til det neste. Svært mange analyseoppgaver er iterative i denne forstanden, og antall iterasjoner kan være svært høyt. Her er noen eksempler:

[30] https://www.voltdb.com/
[31] https://www.memsql.com/
[32] https://ignite.apache.org/
[33] https://redis.io/

- mønstergjenkjenning med nevrale nettverk
- sentralitet (innflytelse og plassering) i sosiale nettverk
- statistisk regresjonsanalyse
- ruteplanlegging

Her er noen eksempler på algoritmer som ikke er iterative:

- beregne aggregater fra en tabell, f.eks. summen av numeriske verdier i en kolonne
- svare på standard SQL spørringer
- gjøre oppslag i en tabell

En iterativ prosess er nødt til å huske mellomresultatet som beregningen avhenger av. Dersom det er mange av dem, og hvert av dem må skrives til disk, vil de samlede tidskostnadene svært raskt overstige akseptable grenser.

I slike tilfeller er det derfor viktig å velge et stordatasystem som har støtte for iterative beregninger, hvilket i praksis vil si at det tilbyr et prosesseringsrammeverk som lagrer mellomresultater i primærminnet. Dette forbedrer ytelsen med flere størrelsesordener, og gjør kunstig intelligens-teknologi som maskinlæring og grafanalyse praktisk gjennomførbart i stordataskala.

Apache Spark og Apache Flink er eksempler på slike primærminnebaserte prosesseringsrammeverk, mens Apache Hadoop er et eksempel på et system som ikke støtter iterasjon i denne forstanden.

2.8 Input/output-profil

CRUD er en forkortelse for det som vanligvis betraktes som de fire grunnleggende operasjonene i datalagringssystemer. Disse operasjonene – her gjengitt uten forsøk på å oversette – er *C*reate, *Read*, *U*pdate og *D*elete. Tabell 2.1 gir konkrete eksempler på disse operasjonene i noen vanlige protokoller.

Databasesystemer varierer med hensyn til hvilke av disse operasjonene de vektlegger, og dette kan kalles systemets input/output-profil. Input/output-profilen gjenspeiler som regel en bakenforliggende filosofi dvs. en tiltenkt funksjon eller idiomatisk bruk av systemet. Fra tilstrekkelig avstand kan man skille mellom to hovedgrupper av systemer, OLAP- og OLTP-systemer:

Et OLTP-system (*online transactional processing*) er designet for å håndtere et stort antall samtidige transaksjoner av typen Create, Update eller Delete. Leseoperasjoner er vanligvis predefinerte og enkle. OLTP-systemer vektlegger dataintegritet og konsistens i flerbrukermiljøer, og effektivitet måles i antall mulige transaksjoner per sekund.

Et OLAP-system (*online analytical processing*) er typisk designet for virksomhetsetterretning (*business intelligence*) og statistikkproduksjon, og benytter typisk multidimensjonale matriser, såkalte rapporteringskuber, som datamodell. Et OLAP-system er vanligvis optimalisert for multidimensjonale analyseoperasjoner (slik som *roll-up*, *drill-down*, *slicing* og *pivoting*) i sanntid eller nær sanntid, og vektlegger følgelig leseoperasjoner fremfor skriveoperasjoner.

Operasjon	SQL	HTTP	DDS
Create	INSERT	PUT/POST	write
Read	SELECT	GET	read/take
Update	UPDATE	PUT/ POST/ PATCH	write
Delete	DELETE	DELETE	dispose

Figur 2.1 CRUD-operasjoner i SQL, HTTP og DDS.

Det kan være vanskelig å definere en klar grense mellom OLAP- og OLTP-systemer, og selv om OLTP og OLAP ofte brukes som teknisk terminologi – skjønt innholdet varierer veldig fra kilde til kilde – bruker vi dem ikke slik i denne rapporten. Her betegner de snarere endepunktene av et spektrum som indikerer hvordan et system vekter de forskjellige CRUD-operasjonene mot hverandre. Vi vil for eksempel snakke om et system som nær OLAP-enden av spekteret dersom det er optimalisert for gjentakende leseoperasjoner og enkle ikke-iterative beregninger slik som gjennomsnitt eller sum.

2.9 Gjenbrukbarhet av data

En relasjonsdatabase forholder seg til et skjema som beskriver tabeller, felter og datatyper samt relasjonen mellom dem. Fordelen med et skjema er at det er nyttig for datavalidering, mens ulempen er at det er komplisert og arbeidskrevende å endre og utvikle.

De fleste nye distribuerte lagringssystemene er av en type som ikke har et skjema. Dette gjelder for eksempel aggregatorienterte databaser og grafdatabaser. Mangelen på et skjema er faktisk et populært trekk som ofte fremheves som et fortrinn, da det gjør at systemutviklere kan konsentrere seg om å modellere problemdomenet uten å bekymre seg for strukturelle endringer. Skjemaløshet gjør det også enklere å gjøre raske tilpasninger til endrede krav.

Det er allikevel viktig å være klar over at et skjema først og fremst har en selvstendig verdi som dokumentasjon, men også og at det kan være misvisende å kalle en lagringsløsning skjemaløs bare fordi skjemaet ikke er eksplisitt definert.

Et aggregatorientert system (jf. kapittel 3.1), for eksempel, har alltid et *implisitt skjema* som defineres av programkoden som leser fra databasen. Mer spesifikt må programkoden lese og forstå informasjonen som er lagret i databasen, hvilket innebærer slike ting som å kjenne til hvordan dataelementer er lagret i forhold til hverandre samt hvilke datatyper de instansierer.

En konsekvens av dette er at informasjon om hva dataene betyr, hvordan de skal tolkes, flyttes fra datasettet selv og inn i applikasjonen. Det er viktig å være klar over at dette går på bekostning av gjenbrukbarhet.

Gjenbrukbarhet – det at dataene kan forståes utenfor sin opprinnelseskontekst – er viktig for mange ting: det er viktig for integrasjon og utveksling mellom systemer, det er viktig for arkiveringsverdien til dataene – arkiverte data bør ikke være avhengige av programkode for å kunne forstås – og det er viktig for at ny programvare eller nye versjoner av gammel programvare skal være bakoverkompatibel.

Gjenbruksverdien av et datasett fremmes generelt av én eller flere av disse tingene (jo flere jo bedre):

- rike metadata basert på åpne standardiserte kodelister eller terminologier
- en praksis for navngivning (relasjoner, objekter, begreper, etc.) som gir entydige identifikatorer og reduserer risikoen for navnekollisjoner
- en datamodell som er semantisk i den forstand at den er i stand til å uttrykke betydningen til og forholdet mellom dataelementer

XML-databaser er en gruppe av lagringssystemer som et stykke på vei oppfyller disse kriteriene. Blant de mest populære er ExistDB[34] og BaseX[35] som begge er applikasjonsplattformer bygget utelukkende rundt XML og XML-relaterte teknologier (spørringer i XQuery og XPath, skjemaer i DTD, RelaxNG og XMLSchema etc.). Problemet er at XML ikke er særlig velegnet for distribuert og parallell prosessering, og vi vet per dags dato ikke om noen XML-database som skalerer horisontalt.

Et bedre alternativ fra et gjenbruk- og utvekslingsperspektiv vil være grafdatabaser (jf. kapittel 3.2). Grafer er en semantisk modell i ovennevnte forstand: forholdet mellom dataelementer og begreper i problemdomenet er eksplisitt kodet i grafen og bidrar til å flytte tolkningen av dataene ut av programkoden og over i datasettet selv. Spesielt interessante i denne sammenhengen er den undergruppen av grafdatabaser som er bygget over W3C-standarden RDF[36], fordi RDF er en datamodell som er designet for å standardisere terminologier med identifikatorer som er universelt utvetydige. Eksempler på RDF-databaser som kan kjøres over maskinklynger omfatter Amazon Neptune[37], som selges som en skytjeneste, samt Stardog[38] og OpenLink Virtuoso[39] som kan konfigureres til å kjøre på én eller flere servere.

[34] http://www.exist-db.org/exist/apps/doc/
[35] http://basex.org/
[36] https://www.w3.org/RDF/
[37] https://aws.amazon.com/neptune/
[38] https://www.stardog.com/
[39] https://virtuoso.openlinksw.com/

3 Hovedtyper av stordatasystemer

I informatikken er en datastruktur definert som en måte å organisere informasjon i en datamaskin. Dersom datamengdene er store er det nødvendig å bruke gode datastrukturer som er optimalisert for å løse rett type beregning på kortest mulig tid. En god datastruktur for et stordatasett kjennetegnes av at den

- passer problemet i den forstand at problemet naturlig lar seg modellere slik,
- minimerer antallet beregninger CPU-en må gjøre,
- er plasseffektiv, og
- kan distribueres og vedlikeholdes effektivt.

Stordatasystemer kan grupperes etter hvilke datastrukturer de er bygget over. Med en hensiktsmessig grovsortering dreier det seg om de følgende tre typene:

- Systemer som representerer data i *tabeller*: en tabell bør her forstås svært vidt som en funksjon som korrelerer en nøkkel med en verdi. I de fleste slike systemer er verdiene som lagres *aggregater*. Ikke ulikt objekter i en objektdatabase er aggregater en samling av relaterte objekter vi ønsker å betrakte som en enhet. Et aggregat kan f.eks. representere en person, med felter for fødselsdato, bostedsadresse, etc. Vi tenker følgelig på disse systemene som *aggregatorienterte* og beskriver dem nærmere i kapittel 3.1.
- Systemer som representerer data i *grafer*: en grafdatabase er en database som lagrer data i form av et semantisk nettverk av objekter eller noder som er forbundet med navngitte relasjoner eller kanter. Grafdatabaser er designet for assosiative data som er innbyrdes forbundet på mange ulike vis, og som ofte danner komplekse mønstre som er vanskelig å modellere i vanlige relasjonsdatabaser. Vi beskriver grafdatabaser nærmere i kapittel 3.2.
- Systemer som beregner *datastrømmer*: strømmesystemer er systemer som er designet for å behandle vilkårlig lange sekvenser av sanntidsdata. Flere og flere typer av data er tilgjengelig som sanntidstrømmer. Dette er tildels et resultat av utbredelsen av sensorer i samfunnet, eksempelvis stedsbevisste mobile enheter, værsensorer, sosiale mediestrømmer, o.a.

Det kan argumenteres for at denne grupperingen representerer tre hovedtyper av lagrings- og prosesseringssystemer som derfor fortjener en nærmere beskrivelse. Kapittel 3.1 beskriver tabulære systemer, som igjen er underdelt i *aggregatorienterte systemer* i kapittel 3.1.1 og *NewSQL*-systemer i kapittel 3.1.2. Kapittel 3.2 beskriver *grafdatabaser*, mens kapittel 3.3 beskriver *strømmesystemer*.

I tillegg til slike elementære databehandlingssystemer, finnes det en rekke mer avanserte rammeverk som er bygget over én eller flere av de ovennevnte typene, f.eks. med grafer, strømmer eller aggregater som standard datamodell, og som er ment å forvalte flere ledd i verdikjeden fra innsamling til analyse av data. Vi tenker på dem som totalløsninger – uten at mye vekt bør legges på dette begrepet.

Vi vil ikke forsøke oss på noen finmasket typologi av totalløsninger i dette kapittelet, men nøyer oss med å gjøre en grovsortering der vi skiller mellom *programmeringsrammeverk* på den ene siden, og *analyse- og visualiseringssystemer* på den andre. Listen har ingen pretensjoner om å være komplett, og beskriver kun noen utvalgte eksempler.

Et programmeringsrammeverk er i korte trekk et sett av moduler eller biblioteker som tilbyr generisk funksjonalitet som enkelt kan tilpasses et spesifikt behov. Kapittel 3.4 beskriver noen eksempler.

Et analyse- og visualiseringssystem, på den annen side, er et verktøy, som regel et grafisk brukergrensesnitt, som hjelper folk å forstå betydningen av data ved å presentere mønstre og trender i dataene på forskjellige vis, ofte ved å plassere dem i en visuell kontekst. Kapittel 3.5 beskriver noen eksempler.

Leseren bør merke seg at det ikke er noen klar linje mellom disse typene. Et stordata programmeringsrammeverk kommer nesten alltid med avanserte analysealgoritmer, og ofte også med enkle visualiseringsverktøy og adaptere til mer avanserte (f.eks. rammeverket Apache Spark[40] og det webbaserte analysegrensesnittet Apache Zeppelin[41]). Omvendt har visualiseringsverktøy ofte en plug-in-arkitektur som gjør dem konfigurerbare til et punkt hvor de nærmer seg rammeverk selv (f.eks. analyseverktøyet Semantica Pro[42]).

3.1 Tabulære databaser

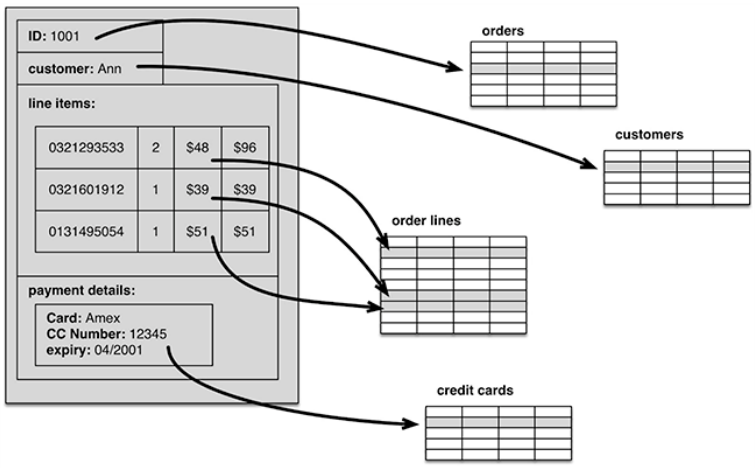
Når det er snakk om å lagre data i tabeller, tenker de fleste på relasjonsdatabaser. Relasjonsmodellen ble først definert av Edgar F. Codd i 1969. I begynnelsen av 70-tallet ble den implementert i prototypesystemer slik som IBM System R og systemet INGRES, sistnevnte utviklet ved Universitetet i California.

I over 30 år ble relasjonsdatabaser brukt for så å si ethvert lagringsbehov som fordrer mer struktur enn et enkelt filsystem, ofte til tross for at datamodellen ikke passet relasjonsmodellen særlig godt. Det er f. eks. bred enighet om at å bruke relasjonsdatabaser til å representere grafer (f.eks. transportruter) og nettverk (f.eks. sosiale nettverk) simpelthen er å bruke relasjonsmodellen på en unaturlig og gal måte. Dette leder uunngåelig til økt kompleksitet og høyere vedlikeholdskostnader ved at man er nødt til å utvikle programvare som oversetter mellom den gitte datamodellen (i dette eksempelet grafer) og relasjonsmodellen.

I databaselitteraturen er dette kjent som *the impedance mismatch problem*, som på norsk kan oversettes med *objektrelasjonell overgangsmotstand*, se figur 3.1. Generelt har dette å gjøre med at relasjonsmodellen – som består av tupler, tabeller, fremmednøkler osv. – ikke reflekterer naturlige objekter i problemdomenet eller representasjonen av dem i programvaren (Sadalage & Fowler 2012). Ta en bestillingsordre som eksempel: den innholder felter som hører naturlig sammen, slik som kundeidentifikasjon, kortinformasjon, adresse, vare, etc. I en relasjonsdatabase vil det derimot ikke finnes noe slikt som en bestillingsordre per se, snarere er en bestillingsordre noe som rekonstrueres ved å kombinere rader fra ulike tabeller som peker til hverandre.

Et annet problem med klassiske relasjonsdatabaser er at de er vanskelige å distribuere over en maskinklynge, og derfor at de skalerer dårlig til tera- og petabyte av data. Dette er mye på grunn av relasjonsdatabasenes normaliserte datamodell og fulle ACID-støtte som gjør at antallet joins og låste tråder påvirker ytelsen negativt (Hecht & Jablonski 2011).

[40] https://spark.apache.org/
[41] https://zeppelin.apache.org/
[42] http://www.semanticresearch.com/semantica-pro
[43] http://www.informit.com/articles/article.aspx?p=2266741



Figur 3.1 Eksempel på objektrelasjonell overgangsmotstand. En applikasjon arbeider med et aggregat av informasjon som gjenspeiler naturlige objekter i problemdomenet. En relasjonsdatabase krever at dette aggregatet deles opp i uavhengige felter som lagres i forskjellige tabeller.[43]

3.1.1 Aggregatorienterte databaser

Aggregertorienterte systemer er en undergruppe av det som med en vag samlebetegnelse kalles NoSQL-systemer – et begrep som også omfatter grafdatabaser (jf. kapittel 3.2).[44] Det eneste egentlige fellestrekket ved NoSQL-systemer er at de oppgir relasjonsmodellen og ACID-garantiene til fordel for alternative datamodeller og mindre stringente konsistensmodeller.

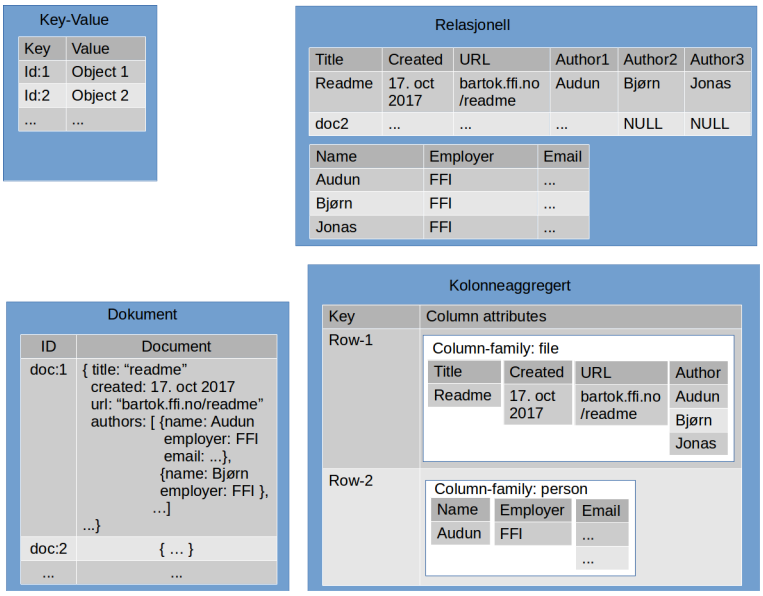
Aggregatorienterte databaser kjennetegnes ved at lagringsstrukturene er designet for å reflektere de naturlige objektene i problemdomenet, ikke ulikt objektdatabasene som ble utviklet på 1980-tallet. De egner seg godt for behandlingsoppgaver der algoritmene kan begrenses til å operere på enkeltaggregater med sjeldne eller ingen behov for krysskobling eller sammenstilling. Under denne antagelsen er heller ikke streng konsistens i ACID-forstand nødvendig. Det er som regel tilstrekkelig å sørge for konsistens for hvert aggregat betraktet for seg.

Det er vanlig å dele aggregatorienterte databaser i tre undergrupper:

- Nøkkel/verdi-databaser (*key/value*)
- Dokumentdatabaser
- Kolonnefamiliedatabaser

Det som skiller disse tre kategoriene fra hverandre er kun graden av strukturrestriksjoner de påtvinger aggregatobjektet: nøkkel/verdi-strukturen har høyest grad av frihet og kolonneaggregert minst mtp. strukturform. Færre strukturrestriksjoner gir friere muligheter for å representere data, men som regel på bekostning av effektivitet ifm. gjenfinning. For øvrig deler de alle den egenskapen at artefaktene er nøkkelbaserte, som vist i figur 3.2.

[44] Forkortelsen NoSQL har en ganske tilfeldig opprinnelse og en tilsvarende uklar mening, men leses ofte som 'not only SQL' dvs. som et polemisk kontrapunkt til tradisjonelle relasjonsdatabaser.



Figur 3.2 Datamodeller.

3.1.1.1 Nøkkel/verdi-databaser.

En nøkkel/verdi-database er enkelt forklart en stor, distribuert og feiltolerant hashtabell eller assosiativ matrise (*associative array*) som er spredt over en horisontalt skalerbar maskinklynge. Mer abstrakt kan nøkkel/verdi-databaser betraktes som et konsept for å endre, lagre og gjenopprette en distribuert oppslagstabell eller ordbok.

Det som kjennetegner nøkkel/verdi-databaser er at dataene som er lagret under en nøkkel er en ustrukturert ‘klump’. Det er ingen måte å beskrive datatyper på, og det er ingen krav til at forskjellige verdier følger et felles sett av strukturrestriksjoner.

I enkelte nøkkel/verdi-databaser, for eksempel, Redis[45] trenger ikke aggregatet som er lagret å være et programobjekt, men kan være en hvilken som helst datastruktur. Redis støtter lagring av lister, mengder og matriser og den kan også utføre enkle operasjoner slik som mengdeteoretisk snitt, union og diff.

Nøkkel/verdi-databaser gir betydelig fleksibilitet. De løser f.eks. problemet med objektrelasjonell overgangsmotstand ved at fullstendige programobjekter kan serialiseres og lagres i databasen som sådan. Kostnaden er imidlertid at dataene er avhengige av programlogikken for å kunne tolkes, og derfor i svært liten grad er gjenbrukbare utenfor sin opprinnelige sammenheng. De vil heller ikke uten videre støtte skalerbarhet på tvers av programvaregenerasjoner (jf. kapittel 2.3).

Fordi nøkkel/verdi-databaser ikke vedlikeholder en indeks utover nøklene, kan de ofte bruke mindre minne enn relasjonsdatabaser til å lagre den samme informasjonen. Dette kan gi store ytelsesgevinster i visse arbeidsbelastninger, men det kommer selvsagt helt an på arbeidet.

Nøkkel/verdi-databaser er velegnet for å lagre ustrukturerte data slik som bilder, video, sonardata, etc.

[45] https://redis.io

De egner seg også godt for å lagre enkle tilstander som trenger å hentes raskt, typisk brukerprofiler, preferanser, sesjonsinformasjon og loggdata.

Nøkkel/verdi-databaser er derimot ikke en god match dersom du trenger å representere relasjoner i naturlig koblede data, slik som f.eks. sosiale nettverk, transport- og logistikksystemer, ruteplanleggingsystemer, m.m. De er heller ikke et godt valg dersom du trenger å søke i dataene basert på komplekse kriterier.

Mange nøkkel/verdi-databaser distribueres ved hjelp av en en teknikk som kalles *sharding*, noe som betyr at datasettet deles horisontalt i segmenter basert på en ordning av nøklene og at hvert segment lagres på forskjellige maskiner.

Sharding gjør det enkelt å skalere opp en maskinklynge etterhvert som et datasett vokser, men det gir også enkelte sårbarheter det er viktig å være klar over: nøkkelen til en dataverdi bestemmer hvilken maskin i maskinklyngen et dataelement er lagret på. Dersom vi antar at nøklene er leksikalsk sortert vil en nøkkel f4b19d79587d, som starter med en 'f', kunne bli sendt til en annen node enn nøkkel ad9c7a396542 som startet med en 'a'. Det følger at dersom maskinen som holder på f4b19d79587d blir utilgjengelig, så vil alle med nøkler tilstrekkelig nære 'f' i alfabetet bli utilgjengelig, og man vil heller ikke kunne lagre data under slike nøkler.

Enkelte systemer, slik som f.eks. Riak[46] lar deg derfor kontrollere repliseringsfaktoren til dataene, og, mer generelt, spesifisere hvordan CAP-egenskapene skal vektes (jf. kapittel 2.4). Man kan for eksempel angi at mer enn halvparten av nodene må respondere på en forespørsel før en leseoperasjon regnes som fullført. Disse innstillingen gjør det mulig å finjustere feiltoleranse for hhv. skrive og leseoperasjoner basert på et gitt behov.

Alle nøkkel/verdi-databaser kan gjøre oppslag på nøkler, men støtter få søkefunksjoner utover det. Dersom man har behov for å søke etter og/eller analysere data basert på en bestemt egenskap, så kan man altså vanligvis ikke gjøre det. Ikke nok med det, det er også nødvendig å kjenne nøklene, dvs. å vite hvilke identifikatorer som er brukt. De fleste nøkkel/verdi-databaser vil ikke gi deg en liste av alle nøkler, og selv om de gjorde det så ville det å hente en liste over alle nøkler for så å lete etter en verdi ved å gå gjennom hele listen være en svært ineffektiv prosess.

Enkelte nøkkel/verdi-databaser slik som Riak kompenserer for dette ved å tilby fulltekstsøk over verdier. Riak f.eks. indekserer verdier ved hjelp av Apache Solr[47].

Allikevel, dette gjelder kun tekst, så i det generelle tilfellet er det ikke noe annet alternativ enn å kjenne sine nøkler, og optimalisering av søkefunksjonalitet vil hovedsaklig være basert på en hensiktsmessig design av nøklene selv, f.eks. i form av at objekter tilhørende samme naturlige gruppe får nøkler med delt prefiks. Dette leder til "ordnede" nøkler (jf. *ordered key-value stores*), som muliggjør intervallsøk (BerkeleyDB,[48] InfinityDB,[49] MemcacheDB,[50] LMDB[51]). Utvider man derimot funksjonaliteten forbi dette, ved å tillate at metadata inkluderes i verdiene, havner man over i kategorien dokumentdatabase.

[46] http://basho.com/products/riak-kv/
[47] http://lucene.apache.org/solr/
[48] https://www.oracle.com/database/berkeley-db/index.html
[49] https://en.wikipedia.org/wiki/InfinityDB
[50] http://memcachedb.org/
[51] http://www.lmdb.tech/doc/

Dagens nøkkel/verdi-databaser bruker ulike konsistensmodeller som eksemplifiserer hele spennet fra svak konsistens til ACID. ACID er dog svært kostbart beregningsmessig, og de fleste nøkkel/verdi-databaser ligger på CP-siden av CAP-spekteret (Dynamo,[52] Oracle NoSQL Database,[53] Project Voldemort,[54] Riak[55]).

Hva gjelder persistering vs. primærminne finner vi også mange varianter. Noen nøkkel/verdi-databaser er primærminnesystemer (Aerospike,[56] Apache Ignite,[57] Redis,[58] Hazelcast,[59]) , mens andre lagrer data på flashenheter (SSD) og/eller disk (Apache Ignite, Aerospike, Couchbase,[60] LevelDB[61]). Disse kategoriene er ikke gjensidig utelukkende da mange primærminnesystemer har en *"spill-over"* funksjonalitet som gjør at data vil persisteres når primærminnet er fullt.

Eksempel: Apache Accumulo. Apache Accumulo er et system som opprinnelig ble utviklet hos NSA (*National Security Agency*) som en fleksibel, rask og horisontalt skalerbar nøkkel/verdi-database med *cellebasert tilgangsstyring*. Accumulo ble avhendet til Apache Software Foundation i 2011, i form av åpen kildekode, og er i dag topp fem blant programvare av sitt slag.

Accumulo bygger på Googles BigTable-modell, men utvider den med en sikkerhetsmekanisme kalt cellebasert sikkerhet: hvert nøkkel/verdi-par har sin egen sikkerhetsetikett som avgjør hvorvidt en bruker har anledning til å lese eller skrive til verdien av denne nøkkelen. Disse sikkerhetsetikettene kan betraktes som atomære eller primitive, mens synligheten til en kolonne vil være en logisk kombinasjon av sikkerhetsetikettene på cellene. Slike komplekse etiketter gir detaljert kontroll over hvilke roller som gir lese- og/eller skriverettigheter til enhver tid basert på datasettet slik det ser ut i øyeblikket.

Cellebasert tilgangsstyring gjør det mulig å lagre data på forskjellige graderingsnivåer i én og samme tabell. Brukere får tilgang kun til de nøklene de er klarert for. Dette gjør Accumulo til en sjelden fugl i den forstand at det er mulig å samarbeide på tvers av graderingsnivåer i ett og samme system. Accumulo er, med andre ord, et eksempel på det som i kapittel 2.3 ble kalt organisatorisk skalerbarhet.

Accumulo tilbyr også en programmeringsmodell basert på såkalte *iteratorer* for å gjøre beregninger over datasettet på serversiden. Denne programmeringsmodellen er grovt sett funksjonelt ekvivalent med MapReduce-modellen og brukes til å aggregere/kombinere verdier fra mange nøkkel/verdi-par til én beregnet verdi.

Se figur 3.3 for en rask oppsummering av Accumulo.

[52] https://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
[53] http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html
[54] http://www.project-voldemort.com/voldemort/
[55] http://basho.com/products/riak-kv/
[56] https://www.aerospike.com/
[57] https://ignite.apache.org/
[58] https://redis.io
[59] https://hazelcast.com/
[60] https://www.couchbase.com/
[61] https://github.com/google/leveldb

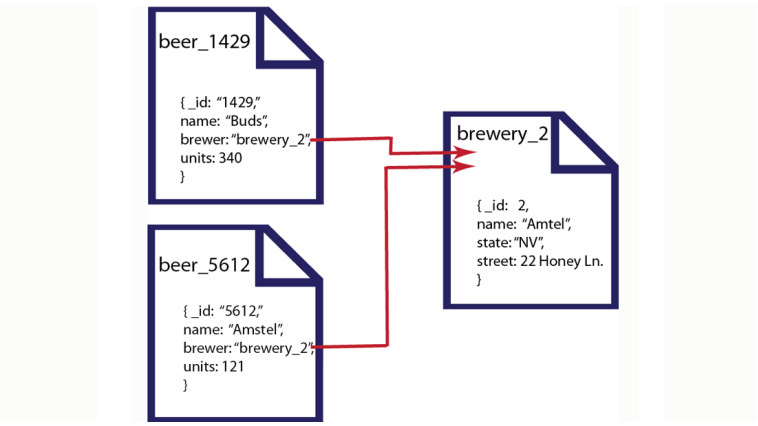
Analysestøtte	Enkle nøkkelbasert søk. Tekstindeksering av verdier.
Programmeringsmodell	Iterator/MapReduce
Skalerbarhet	Horisontalt skalerbar lagringsplass og regnekraft. Organisatorisk skalerbar
Tilgjengelighet vs. konsistens	ACID-konsistens på enkeltnøkler. Svak konsistens for distribuerte søk.
Sårbarhet og feiltoleranse	Feiltoleranse vha. datareplisering og persistering
Primærminne vs. persistens	Persistenssystem (Apache Hadoop HDFS, jf. kapittel. 3.4.1)
Iterative beregninger	Raskt for enkle repeterende operasjoner. Lite egnet for iterative oppgaver.
Input/output-profil	På OLTP-siden av spekteret
Gjenbrukbarhet av data	Lav. Avhengig av applikasjonslogikk.

Figur 3.3 Apache Accumulo, raskt oppsummert.

3.1.1.2 Dokumentdatabaser

Dokumentdatabaser er en underklasse av nøkkel/verdi-databaser. Forskjellen ligger i hvordan dataene behandles: i en nøkkel/verdi-database er dataene ustrukturerte og ugjennomsiktige. De er, med andre ord, rå bytestrenger uten metadata som det er opp til klientprogramvaren å kjenne betydningen av. Data i en dokumentdatabase, derimot, er representert i et *semistrukturert*, format slik som XML, JSON, BSON e.l.

Semistrukturerte dokumenter kan inneholde metadata som gjør datasettet mer selvbeskrivende. Det er f.eks. mulig (og vanlig) å skille mellom en serie med tall som uttrykker et telefonnummer, et tall som uttrykker et postnummer og et som uttrykker en dato, se figur 3.4.



Figur 3.4 Nøkler og JSON-verdier i en dokumentdatabase.[62]

Som nøkkel/verdi-databaser er også dokumentdatabaser i en strikt forstand skjemaløse, hvilket vil si at formen til et aggregat kan variere fra nøkkel til nøkkel og at nøsting av elementer er tillatt. Kortfattet kan man altså definere dokumentmodellen som en nøkkelindeksert samling av vilkårlig nøstede semistrukturerte dokumenter som deler format, men varierer fritt i form.

[62] https://developer.couchbase.com/documentation/server/3.x/developer/dev-guide-3.0/compare-docs-vs-relational.html

Siden dokumenter er skjemaløse, er dokumentdatabaser på mange måter mer fleksible enn relasjonsdatabase. Det er imidlertid andre kostnader forbundet med denne fleksibiliteten. Som et eksempel er det ikke nødvendig å skrive applikasjonslogikk for å håndtere nullverdier i dokumentdatabaser: I en relasjonsdatabase fordrer skjemaet at hvert objekt i en tabell har de samme egenskapene og at egenskapene, dvs. kolonnene, har en verdi. Dersom en slik verdi ikke eksisterer må kolonnen eksplisitt settes til `null`, noe som ofte skaper mye unntakslogikk i programkode. I en dokumentdatabase, derimot, kan egenskapene variere fritt fra objekt til objekt, og en egenskap uten en verdi for et gitt objekt kan simpelthen utelates fra beskrivelsen av dette objektet.

Skaleringsmodellen i en dokumentdatabase varierer fra produkt til produkt, men skiller seg ikke nevneverdig fra nøkkel/verdi-databaser: de fleste dokumentdatabaser er bygget over en master/slave-arkitektur (jf. kapittel 2.5) der skrivetilgang kontrolleres av drivernoden (*the master*), mens lesetilgang administreres av arbeiderne som deler og repliserer dataene seg imellom.

Dette gir de forventede sårbarhetene som følger fra avhengigheten av drivernoden. Det finnes dog mange tilpasninger av master/slave-konseptet blant dokumentdatabaser som kan øke tilgjengeligheten og robustheten. Som eksempel kan man nevne såkalte *replikasett* i MongoDB, som er beskrevet nærmere i kapittel 3.1.1.2 nedenfor.

Dokumentdatabaser, som NoSQL-systemer generelt, er som oftest BASE-systemer, hvilket vil si at de velger responsivitet framfor sterk konsistens, jf. kapittel 2.4.1. Det er allikevel vanlig å gi transaksjonsgarantier som likner på ACID for alle operasjoner som kun manipulerer ett dokument. For mange bruksområder er dette godt nok, da dokumentene i en dokumentorientert database representerer objekter i problemdomenet som holder relevante data sammen i ett og samme aggregat. Dokumentdatabaser er med andre ord designet nettopp for å redusere avhengigheten av å kombinere data på tvers av tabeller, som er de komplekse operasjonene som ACID, på sin side, er designet for.

MongoDB, for eksempel, gir ingen ACID-garantier på tvers av dokumenter, men støtter atomisitet for operasjoner på et enkelt dokument. Det finnes selvsagt brukstilfeller hvor denne begrensningen til enkeltdokumenter er hemmende, dvs. tilfeller hvor det vil være ønskelig og naturlig å la operasjoner spenne over mer enn ett dokument. De fleste nyere dokumentdatabaser gir derfor ACID-garantier for *mengden av dokumenter som inngår i operasjonen*. Disse systemene markedsfører seg gjerne som ACID-systemer per se, men leseren bør merke seg at også disse systemene, som alle andre parallelle distribuerte systemer, ikke kan garantere sterk konsistens *og* tilgjengelighet samtidig dersom det er en mulighet for at noen noder i maskinklyngen kan svikte.

Blant nyere systemer som tilbyr ACID-garantier i ovennevnte forstand finner man MarkLogic, [63] OrientDB, [64] Azure CosmosDB,[65] CrateDB[66] m.fl.

Forskjellige dokumentdatabaser tilbyr forskjellig funksjonalitet for å søke i data, men felles for dem alle er

- at de lagrede dokumentene representerer naturlige aggregater eller objekter i programkoden, og

[63] https://www.marklogic.com/
[64] https://orientdb.com/
[65] https://docs.microsoft.com/en-us/azure/cosmos-db/introduction
[66] https://crate.io/

- at dokumentene har søkbar struktur.

Det første punktet innebærer at felter med informasjonfragmenter eller felter som er relevant for hverandre som en hovedregel vil kunne hentes fra ett enkelt dokument (jf. diskusjonen om objektrelasjonell overgangsmotstand innledningsvis). Det andre punktet innebærere at felter som er relatert på denne måten er søkbare og at dokumenter kan filtreres ut med komplekse uttrykk.

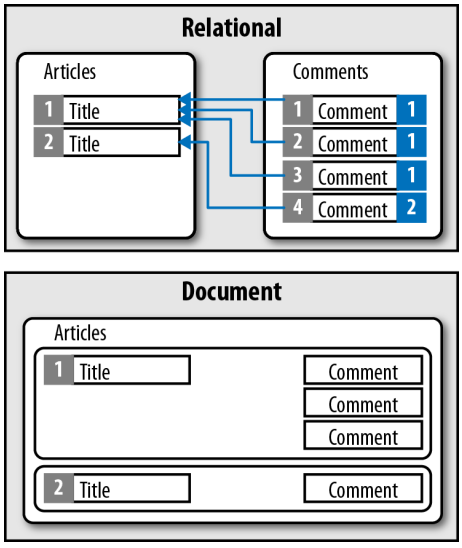
MongoDB, for eksempel, har et spørrespråk basert på JSON. Eksempelet nedenfor henter alle dokumenter i databasen `inventory` som har status "A" og lagerantall mindre enn 30 (eksempelet er hentet fra dokumentasjonen av MongoDB).

```
db.inventory.find({{ status: "A", qty: {$lt: 30}})
```

Spørringen tilsvarer det følgende SQL uttrykket:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

Dette skiller seg fra nøkkel/verdi-databaser der man for hvert oppslag på en nøkkel må hente alle dataene som er assosiert med den nøkkelen, for så å analysere dem i programkoden. Dokumentdatabaser ligger derfor, hva spørringer angår, nærmere relasjonsdatabaser enn nøkkel/verdi-databaser, jf. figur 3.5.



Figur 3.5 En illustrasjon av forholdet mellom dokumentdatabaser og relasjonsdatabaser.

Hvilket API eller spørrespråk som tilbys varierer mye fra produkt til produkt, og det finnes ingen standard.

Enkelte dokumentdatabaser tilbyr funksjonalitet for å aggregere verdier og lagre mellomresultatene som selvstendige dokumenter. I databaselitteraturen er dette kjent som *materialiserte*

tabeller/perspektiver (*materialized views*). Et eksempel på dette er CouchDB. CouchDB tilbyr en søkefunksjonalitet som er i stand til å sammenstille rådata med slike materialiserte perspektiver. Tenk deg for eksempel at du ofte ønsker å sammenligne drivstofforbruket til en bestemt panservogn med gjennomsnittet for en sammenliknbar klasse kjøretøy. Med CouchDB kan du implementere en map/reduce-rutine for å preprosessere og lagre gjennomsnittet, og senere sammenstille data fra dette med annen informasjon om konkrete panservogner etter utført oppdrag.

Eksempel: MongoDB. MongoDB er en gratis dokumentdatabase som publiseres som åpen kildekode under en kombinasjon av GNU Affero General Public License og Apache License. Den er en interessant videreutvikling av master/slave-arkitekturen som øker både tilgjengeligheten og robustheten til et slikt system, dog på bekostning av lagringsplass.

MongoDB er basert på konseptet om et *replikasett*, en klynge maskiner som speiler hverandres data og som implementerer en konsensusprotokoll for å velge en *primærnode*. Primærnoden mottar og kontrollerer skriveoperasjoner til systemet og er funksjonelt ekvivalent med drivernoden i en konvensjonell master/slave-arkitektur.

De andre maskinene i et replikasett kalles *sekundærnoder*. Deres oppgaver er å kopiere operasjonsloggen fra primærnoden slik at alle noder kan rekonstruere det samme datasettet, og å velge en ny primærnode dersom den nåværende driveren skulle bli utilgjengelig over en viss tid. Selv om en klient av ulike årsaker ikke er i stand til å kommunisere med primærnoden, og derfor ikke kan skrive til systemet, så kan den fortsatt *lese* fra sekundærnodene. Videre, dersom primærnoden forblir utilgjengelig over en viss tid – et tidsintervall som angis ved en konfigurasjonparameter ved oppstart av systemet – vil sekundærnodene i replikasettet velge en ny primærnode og gjenoppta normal drift. Alle framtidige skriveoperasjoner betjenes av den nye primærnoden, mens sekundærnodene begynner å kopiere data fra denne. Dersom den gamle primærnoden skulle våkne, så vil den slutte seg til replikasettet som en ny slave og vil begynne å speile den nye primærnoden.

Se figur 3.6 for en rask oppsummering av MongoDB.

Dataanalyse	Nøkkel-baserte søk, geo-analyse, m.m.
Programmeringsmodell	Funksjonell/MapReduce
Skalerbarhet	Horisontalt skalerbar vha. *sharding* (se 3.1.1.1)
Tilgjengelighet vs. konsistens	Prioriterer konsistens
Sårbarheter og feiltoleranse	Feiltoleranse vha. replikasett med valgt primærnode
Primærminne- vs. persistenssystem	Persistenssystem, men med mulighet for å koble til primærminne-lagring
Støtte for iterative beregninger	God støtte for iterative beregninger når den er satt opp med primærminne-lagring
Input/output-profil	Nærmere OLTP enn OLAP
Gjenbrukbarhet av data	Lav. Avhengig av applikasjonslogikk

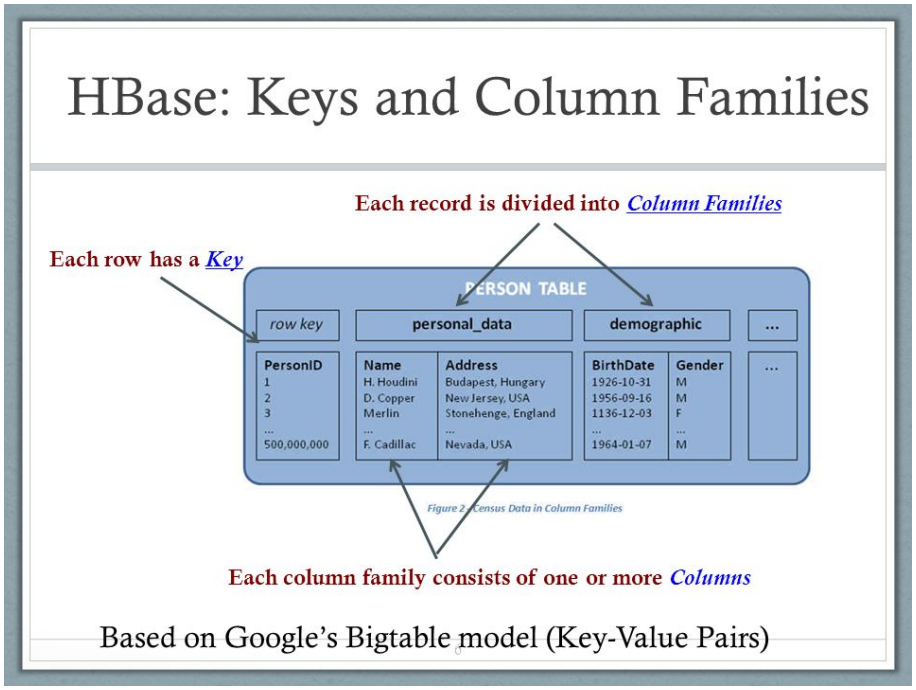
Figur 3.6 MongoDB, raskt oppsummert.

3.1.1.3 Kolonneorienterte databaser

En konvensjonell relasjonsdatabase er *radorientert*. En SQL-spørring som ber om verdiene til bestemte attributter (tilsvarende kolonner) i en gitt tabell, leser hele rader fra disk om gangen, og

filtrerer vekk uønskede attributter etter at raden er overført til primærminnet.

Kolonneorienterte databaser er transposisjonen av dette prinsippet forsåvidt som kolonneorienterte databaser partisjonerer en tabell *vertikalt* og lagrer den som en samling av separate individuelle *kolonner* (Abadi et al. 2013), se figur 3.7.



Figur 3.7 Kolonnefamilier i HBase.[67]

Selv om forskjellen mellom disse to kan virke subtil, har den dyptgripende konsekvenser: I kolonneorienterte databaser er det å legge til kolonner en billig operasjon og kan gjøres på en rad-til-rad basis. Hver rad kan med andre ord ha forskjellige kolonner, noe som sparer både plass og regnekraft sammenliknet med `null`-verdier. Det følger at kolonneorienterte databaser ikke har et skjema i en relasjonell forstand, og derfor at betydningen av individuelle dataelementer ikke er dokumentert i datasettet selv, men må skrives inn i applikasjonslogikken (jf. gjenbrukbarhet i kapittel 2.9). Allikevel, kolonneorienterte databaser, som relasjonsdatabaser, er essensielt et tabulært konsept, og har en liknende tabulær grafisk representasjon som relasjonsdatabaser. Man kan kanskje si at kolonneorienterte databaser er midtveis mellom nøkkel/verdi-databaser og relasjonsdatabaser (Redmond & Wilson 2012). De tre mest populære kolonneorienterte databasene er Hypertable,[68] HBase[69] og Cassandra.[70]

Grunnen til at vi diskuterer kolonneorienterte databaser under *aggregatorienterte stordatasystemer* er framveksten av NoSQL-systemer hvor det sentrale aggregatet er *kolonnefamilier*. En kolonnefamilie er en mengde kolonner som hører sammen på samme måte feltene i et JSON-dokument gjør i dokumentdatabaser.

[67] https://slideplayer.com/slide/12948836/
[68] http://www.hypertable.com/
[69] https://hbase.apache.org/
[70] http://cassandra.apache.org/

Man kan tenke på kolonnefamilier som en nøstet funksjon, eller som en aggregatstruktur med to nivåer. Det første nivået består av et sett av nøkler som identifiserer rader, dvs. tupler som består av data som karakteriserer det samme objektet. Det andre nivået, som altså er korrelatet til nøkkelen som identifiserer raden, er selv en funksjon som gir en verdi for hvert element av et utvalg av kolonner. Figur 3.7 gir en illustrasjon av hvordan dette konseptet er implementert i Apache HBase.

Kolonnefamilier er et godt egnet lagringsformat for store datasett (petabyte- og exobytestørrelse) fordi datamodellen, som en kolonneorientert modell, gjør det enkelt å la dataene partisjoneres og spres over svært store maskinklynger.

Kolonnefamilier tillater at nye rader og kolonner legges til mens systemet kjører, noe som gjør kolonnefamilier til en svært fleksibel datamodell som kan tilpasse seg heterogene data raskt. De er ikke like fleksible som dokumentdatabaser, riktignok, da kolonnefamilien selv som regel må være predefinert.

De positive siden av dette er at predefinerte kolonnefamilier (med typede felter) kan utnyttes for indeksering og optimalisering av søk. De fleste kolonnefamiliedatabaser tilbyr derfor et strukturert spørrespråk som minner en god del om SQL. Faktisk kan man med en viss rimelighet betrakte kolonnefamiliedatabaser som et skritt tilbake mot SQL-databaser, men med utgangspunkt i en arkitektur som ulikt relasjonsdatabaser er bygget for maskinklynger. Denne bevegelsen tilbake til SQL og relasjonsmodellen er en trend i tiden, som er en eksplisitt målsetting med de såkalte NewSQL-databasene som vi beskriver i kapittel 3.1.2.

Kolonnefamiliedatabaser brukes i dag til bl.a. brukeradferdsanalyse (jf. kapittel 2.1), analyse av sosiale medier, hendelseslogger, tingenes internett applikasjoner, o.a.

Eksempel: Apache Cassandra Apache Cassandra er en av de mer populære kolonnefamiliedatabasene på markedet, og brukes bl.a. av Netflix, eBay, GitHub og Instagram. Cassandra ble opprinnelig utviklet for Facebook, men ble gjort til åpen kildekode under Apache-parasollen i 2010.

Cassandra er bygget på en peer-to-peer-arkitektur uten en drivernode (*masterless*, jf. kapittel 2.5), hvilket vil si at en hvilken som helst lese- eller skriveoperasjonen kan betjenes av hvilken som helst maskin i klyngen. Cassandra er derfor svært responsivt og tilgjengelig, men – som man bør forvente – bytter dette mot sterk konsistens (jf. kapittel 2.4.1).

Følgelig mangler Cassandra transaksjoner i tradisjonell ACID-forstand (jf. kapittel 2.4.1), hvor man har gjensidig utelukkende komplekse lese- og skriveoperasjoner som enten lykkes eller feiler i sin helhet. En konsekvens av dette er at Cassandra ikke er konsistent for individuelle rader; samtidige oppdateringer av samme rad i samme tabell kan endre kolonner på inkonsistente vis – gitt at de ikke er primærnøkler.

Cassandra tilbyr imidlertid noe de kaller *tunable consistency* som lar brukeren bestemme hvordan vektingen av konsistens mot tilgjengelighet skal gjøres for hver enkelt operasjon. Den grunnleggende idéen er at en operasjons *konsistensnivå* spesifiserer hvor mange maskiner med kopier av de berørte dataene som trenger å svare for å betrakte en operasjon som suksessfull. Cassandra definerer de følgende nivåene, og andre:

ONE Én enkelt maskin må bekrefte.

```
CREATE COLUMNFAMILY POI (
        ssn int PRIMARY KEY,
        navn varchar,
        bosted varchar,);


 INSERT INTO POI (ssn, navn, bosted)
 VALUES (15047******,
         'Audun Stolpe',
         'Ski');
```

Figur 3.8 Den øverste CQL-spørringen oppretter en kolonnefamilie for "persons of interest" (POIs). Den nederste lagrer en spesifikk rad i denne kolonnefamilien.

TWO To maskiner må bekrefte.

THREE Tre maskiner må bekrefte.

QUORUM Et flertall $(n/2 + 1)$ av de n maskinene i klynga må bekrefte.

Dette gjelder leseoperasjoner. Skriveoperasjoner sendes alltid til alle kopier uavhengig av konsistensnivå.

Cassandra leveres med spørrespråket CQL (*The Cassandra Query Language*), som er et SQL-liknende språk, dog mye enklere, som skjuler alle detaljer knyttet til lagringen, spredning og replisering. Figur 3.8 gir et eksempel. Det finnes CQL drivere for Java, Python, Node.JS, Go og C++.

Hva gjelder input/output-egenskaper, befinner Cassandra seg på OLTP-siden av spekteret. Fordi den har en peer-to-peer-arkitektur uten en drivernode kan Cassandra håndtere svært mange små skriveoperasjoner på kort tid, og egner seg derfor godt blant annet til applikasjoner som behandler tidsserier.

Se figur 3.9 for en rask oppsummering av Cassandra.

Dataanalyse	Brukeradferdsanalyse, analyse av sosiale medier, m.m.
Programmeringsmodell	Funksjonell/MapReduce
Skalerbarhet	Horisontalt skalerbar i datamengde
Tilgjengelighet vs. konsistens	Lar brukeren konfigurere balansen
Sårbarheter og feiltoleranse	Peer-to-peer-arkitektur
Primærminne- vs. persistenssystem	Kan settes opp til å bruke primærminne
Støtte for iterative beregninger	God støtte i de tilfeller den settes opp til å bruke primærminne
Input/output-profil	På OLTP-siden av spekteret
Gjenbrukbarhet av data	Lav. Avhengig av applikasjonslogikk

Figur 3.9 Cassandra, raskt oppsummert.

3.1.2 NewSQL-databaser

NewSQL er en betegnelse som brukes om en klasse distribuerte relasjonsdatabaser som forsøker å å skalere til store datamengder like godt som NoSQL systemer, men uten å ofre ACID-garantier eller SQL. Med andre ord ønsker disse systemene å oppnå den samme skalerbarheten som NoSQL-systemer fra midten av 2000-tallet, men uten å forsake relasjonsmodellen (med SQL) og transaksjoner slik de ble utviklet på 1970- og 80-tallet (Pavlo & Aslett 2016).

Dette er en vanskelig blanding av egenskaper å sjonglere, som kun i senere tid har blitt en reell mulighet. Dette skyldes bl.a. framveksten av *in-memory computing*, og fallende priser på SD RAM. Utviklingen av maskinvare har nådd et punkt hvor minne nå er så billig at alle, med unntak av de aller største, databasene kan holdes fullstendig i minnet som samlet sett tilhører en maskinklynge.

SAP HANA, [71] MemSQL,[72] CockroachDB,[73] Clustrix,[74] NuoDB,[75] Google Spanner[76] og Altibase[77] er eksempler på kommersielle NewSQL-systemer som aktivt utvikles og promoteres.

På grunn av transaksjonsgarantiene, vil NewSQL databaser ofte kunne egne seg godt til OLTP-applikasjoner (jf. kapittel 2.8), det vil si applikasjoner som fordrer et stort antall transaksjoner som (1) er kortvarige, (2) aksesserer en liten delmengde av datasettet ved hjelp av en indeks, og (3) er gjentakende.

I motsetning til tradisjonelle relasjonsdatabaser, som er radorienterte, er de fleste NewSQL systemer kolonneorienterte, hvilket vil si at data er samlet i enkeltkolonner snarere enn i rader. Fordelen med dette er at en SQL-spørring kun vil trenge å lese de radene den angår, mens ulempen er at når antallet kolonner i spørringen øker så øker kommunikasjonskostnadene mellom de maskinene som lagrer dem. Dersom man kan foregripe hvordan disse spørringene ser ut, og det er grunn til å forvente at de ikke endrer seg nevneverdig, vil en kolonnestruktur kunne tilpasset det spesifikke problemet og gi en NewSQL database svært høy ytelse også for analytiske OLAP arbeidsbyrder. Flere NewSQL-systemer markedsfører seg derfor som *hybrid transactional/analytical processing*.

En annen interessant egenskap ved NewSQL-systemer er at databaseskjemaet er valgfritt. Man kan altså velge å bruke databasen som en dynamisk og fleksibel lagringsløsning som ikke er kresen på datatyper og som tilpasser seg heterogene data, eller man kan bruke den som et system som sjekker sin egen integritet og vedlikeholder tabeller i henhold til predefinerte regler. Skjemaløs drift har også enkelte fordeler knyttet til vedlikehold. For eksempel trenger ikke utviklere å skrive unntakslogikk for å håndtere `null`-verdier i glisne (*sparse*) tabeller.

Et interessant spørsmål er hvorvidt noen NewSQL-databaser er bakoverkompatible med programkode som bruker konvensjonelle databaseobjekter. Vi vet ikke svaret på det per dags dato.

[71] https://www.sap.com/products/hana.html
[72] https://www.memsql.com/
[73] https://www.cockroachlabs.com/
[74] https://www.clustrix.com/
[75] https://www.nuodb.com/
[76] https://cloud.google.com/spanner/
[77] http://altibase.com/

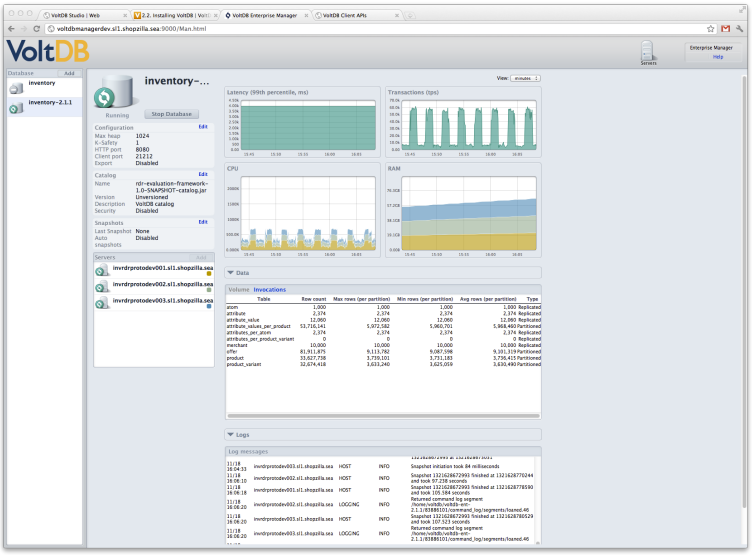
Eksempel: VoltDB VoltDB,[78] se illustrasjon i figur 3.10, er en NewSQL-database som kombinererer sterke ACID-garantier med maskinklynger og distribuert databehandling.

I tillegg til å støtte tradisjonell transaksjonsbehandling (OLTP) (VoltDB kan håndtere millioner av enkle transaksjoner per sekund) har VoltDB en rekke andre anvendelser som tradisjonelle SQL-databaser vanligvis ikke støtter, f.eks. (Stonebraker & Weisberg 2013):

- vedlikehold av tilstand i internettspill,
- riskanalyse i online handelsapplikasjoner, og
- vedlikehold av konsistens i distribuerte filsystemer.

Hovedgrunnen til at VoltDB er i stand til å tilby ACID-garantier – tilsynelatende kontra CAP-teoremet (jf. kapittel 2.4) er at det er et primærminnesystem, noe som gjør at minne effektivt kan låses, kopieres og oppdateres.

På grunn av sin horisontalt skalerbare maskinklyngearkitektur er VoltDB et fleksibelt verktøy som er velegnet for virksomhetsetterretning (*business intelligence*) og enhver sanntidsanalyse over datastrømmer – spesielt når analysen trenger å loggføre tilstand og/eller historikk.



Figur 3.10 VoltDB.

Som nevnt tidligere har primærminnesystemer den opplagte sårbarheten at primærminnet er flyktig, hvilket betyr at dataene forsvinner ved systemsvikt eller strømbrudd. VoltDB er imidlertid kompatibelt med brorparten av Hadoop-økosystemet, og kan kombineres med en egnet slik 'backend' for persistering.

VoltDB er et av de ledende systemene av NewSQL-typen, og var et av de første til å markedsføre seg med merkelappen *hybrid transactional/analytical processing*). Hvordan VoltDB plasserer seg langs aksene vi benytter i denne rapporten er vist i figur 3.11.

[78] https://www.voltdb.com/why-voltdb/

Dataanalyse	Virksomhetsetterretning, analyse over datastrømmer, m.m.
Programmeringsmodell	Funksjonell
Skalerbarhet	Tilrettelagt for horisontal skalerbarhet
Tilgjengelighet vs. konsistens	Sterke ACID-garantier
Sårbarheter og feiltoleranse	
Primærminne- vs. persistenssystem	Primærminnesystem
Støtte for iterative beregninger	God støtte for iterative beregninger
Input/output-profil	Hybrid transactional/analytical processing
Gjenbrukbarhet av data	Avhengig av valg av databaseskjema

Figur 3.11 VoltDB, raskt oppsummert.

3.2 Grafdatabaser

Grafer eller nettverk er en svært vanlig struktur i verden omkring oss. Eksempler inkluderer sosiale nettverk, økosystemer, transportnettverk, kjemiske nettverk og terroristnettverk. Alle disse systemene kan representeres som grafer der individuelle objekter er innbyrdes forbundet på forskjellige måter som til sammen danner sammenhengende, heterogene nettverk. En grafdatabase er en database som er designet for å uttrykke, lagre og behandle slike nettverk maskinelt.

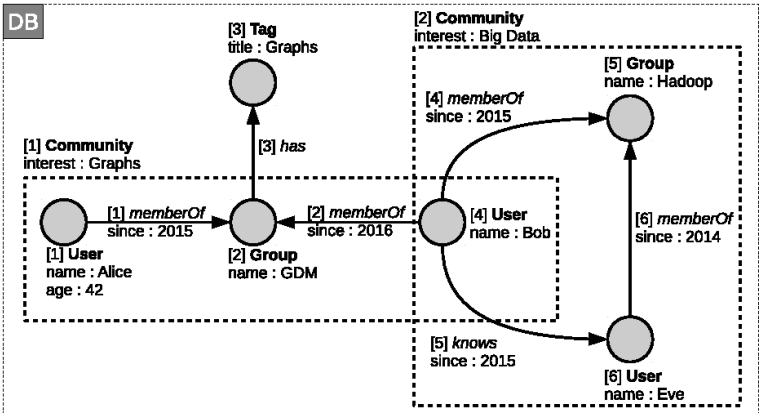
Grafer har lenge vært en populær mekanisme for kunnskapsrepresentasjon innen kunstig intelligens. De tilbyr en visuelt tiltalende og intuitiv måte å uttrykke kognitive strukturer, og er matematisk veldefinert og velegnet for algoritmisk behandling. Populariteten har skutt i været de siste årene i takt med utviklingen av matematisk sosiologi, mer spesifikt *sosial nettverksanalyse*, og den stadig økende utbredelsen av sosiale medier. Vi går nærmere inn på dette i kapittel 3.2.3.

Det finnes to hovedtyper av distribuerte grafdatabaser. Den ene typen består av systemer som er utviklet som grafdatabaser fra bunnen av og som ikke tilbyr noen annen funksjonalitet enn grafprosessering. Den andre typen består av generelle stordatasystemer som tilbyr programvarebiblioteker for grafbehandling, men som i tillegg er bygget for batch-prosessering, overvåking av strømmer, maskinlæring, o.a. Vi gir noen eksempler på hver av typene i kapittel 3.2.5.

En annen noenlunde klar måte å sortere grafdatabaser på er å skille mellom hvilke grafmodeller som benyttes. I matematisk terminologi er en graf simpelthen en mengde elementer, gjerne kalt *noder* eller *punkter*, som er forbundet med binære relasjoner kalt *kanter*. Hver node representerer et informasjonselement i grafen, mens hver kant representerer et forhold. Denne definisjonen er matematisk og abstrakt og kan selvsagt realiseres på mange forskjellige vis. Ved å variere den indre strukturen til nodene og kantene kan man definere forskjellige konkrete grafmodeller.

Som et interessant eksempel kan vi nevne nøstede grafer, eller grafer der nodene selv er grafer. Denne modellen kalles også *hypernodemodellen* og illustrert i figur 3.12. En interessant egenskap ved hypernodemodellen er at den er rik nok til å representere hvordan forskjellige nettverk forholder seg til hverandre. Dette har interessante anvendelser innenfor etterretning og overvåking, f.eks. som støtteverktøy for å analysere forholdet mellom ulike, potensielt overlappende, samfunn eller interessegrupper på sosiale medier (jf. figur 3.12). Andre anvendelser er også tenkelige: Hypernodemodellen kan f.eks. brukes til å representere kommunikasjonslinjene som forbinder mobile ad hoc-nettverk, og de kan brukes til å rangere informasjon i ulike nettverk etter sannsynlighet

og/eller opprinnelse.[79]



Figur 3.12 Representasjon av nettsamfunn i en hypernodegraf (Junghanns et al. 2017b).

For kommersielle grafdatabaser er imidlertid den enkle grafmodellen uten nøstede elementer en de facto standard. Vi lar det derfor være med hypernoder, og setter andre komplekse grafmodeller til side i denne omgang.

Dagens marked har langt på vei konvergert rundt de to konkrete grafmodellene som kalles hhv. *The Property Graph Model* (PGM) og *The Resource Description Framework* (RDF). PGM-grafer støttes av mange kommersielle produkter og har stor industriell utbredelse. RDF brukes på sin side som underliggende teknologi for flere stordataløsninger som f.eks. Semantica Pro[81] og Amazon Neptune[82]), og støtter automatisk resonnering og regelbasert kunstig intelligens. Vi beskriver PGM og RDF nærmere nedenfor.

3.2.1 Attributtgrafer (PGM)

The Property Graph Model beskriver en bestemt type rettede multigrafer – vi kaller dem *attributtgrafer* her – som har fått stor utbredelse i databasemarkedet. Attributter i denne sammenhengen skal forstås som nøkkel/verdi-par, og grafene kalles attributtgrafer fordi både noder og kanter kan være annotert med et vikårlig antall slike par. Oppsummert kan attributtgrafer beskrives slik (Robinson et al. 2013):

- Attributtgrafer består av noder, kanter, merkelapper/navn og attributter elelr nøkkel/verdi-par.
- Noder kan inneholde et ubestemt antall attributter. Man kan tenke på noder som dokumenter som lagrer data i et enkelt tabulært nøkkel/verdi-format der nøkkelen er en tekststreng og verdien er en datatype slik som et tall, en liste, e.l.
- Noder kan være merket. Et merke er en tekststreng som angir hvilken type node en node er. Merkelapper klassifiserer følgelig noder, og antyder hvilken rolle de spiller i en gitt modell.

[79] Gradoop[80]er et hypernodesystem bygget over Hadoop (jf. kapittel 3.4.1) som implementerer hypernodemodellen. De fleste RDF-databaser støtter også hypernodemodellen gjennom den delen av RDF-spesifikasjonen som omhandler navngitte grafer (*named graphs*).

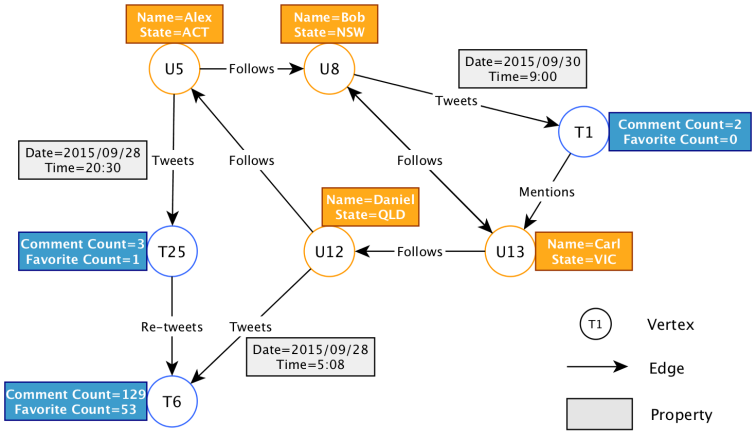
[80]https://dbs.uni-leipzig.de/research/projects/gradoop

[81]http://www.semanticresearch.com/semantica-pro

[82]https://aws.amazon.com/neptune/

- Kanter kobler sammen noder og representerer relasjoner mellom dem. En kant har en retning og et navn som til sammen antyder den semantiske betydningen av relasjonen.
- Kanter kan også ha attributter som gir mer informasjon om betydningen av en relasjon utover dens navn.

Figur 3.13 gir et eksempel på en attributtgraf over handlinger på Twitter.



Figur 3.13 En attributtgraf. [83]

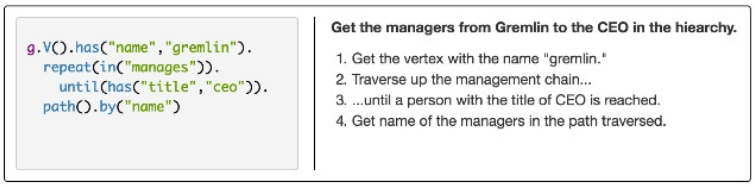
Muligheten for å annotere relasjoner med data kan være en nyttig måte å supplere med informasjon som kan utnyttes algoritmisk. Et typisk tilfelle vil være å gi kanter en vekt, altså et tall, som representerer en kostnad assosiert med å traversere den. Vekten til en kant kan f.eks. representere geografisk avstand eller kapasiteten til en kommunikasjonskanal, og dette kan i sin tur benyttes sammen med standard algoritmer fra grafteori til å beregne eksempelvis informasjonsflyt eller transportruter med minimal sammenlagt kostnad.

Attributtgrafer kan betraktes som et nyttig kompromiss mellom en tabulærorientert og en graforientert datamodell: både noder og kanter kan betraktes som navngitte oppslagstabeller, hvilket vi si at ikke all informasjon trenger å representeres som stier i grafen. Spesielt vil enkle metadata om noder og kanter (slik som personnavn, alder og kjønn i tilfellet der noden representerer en person) med fordel kunne representereres som nøkler og verdier i en nodetabell, mens stier kan forbeholdes mer semantisk betydningsfulle relasjoner mellom forskjellige noder, f.eks. bekjentskapsrelasjoner. Her finnes det selvsagt ikke noen klare regler som kan erstatte erfaring og dømmekraft.

Apache TinkerPop[84] er et programmeringsgrensesnitt for attributtgrafer som nærmer seg en de facto standard. Alle TinkerPop systemer er forenlige og kan integreres og/eller utveksle informasjon uten videre. TinkerPop abstraherer vekk lagringsstrukturen og grensesnittene til det underliggende databasesystemet slik at valget av teknologi for en bestemt applikasjon ikke nødvendigvis går på bekostning av skalerbarheten til systemet: én applikasjon kan være best tjent med en primærminne grafdatabase som støtter ACID-transaksjoner, mens en annen fordrer distribuerte BASE-systemer som er skalert for tilgjengelighet og høy trafikk. Dersom begge er TinkerPop-systemer skal de allikevel kunne sammenføyes uten større besvær.

[83] https://minjianblog.wordpress.com/2015/10/11/graph-databases/
[84] http://tinkerpop.apache.org/

I TinkerPop-suiten av teknologier ligger også spørrespråket Gremlin.[85] Gremlin kan best beskrives som et traverseringsspråk som lar deg navigere i en graf med utgangspunkt i en gitt node. Figur 3.14 gir et eksempel.



Figur 3.14 Gremlin: et graftraverseringsspråk.[86]

Det finnes andre mer kraftige deklarative språk basert på mønstergjenkjenning. Et eksempel er Cypher, som er et SQL-liknende språk utviklet for grafdatabasen Neo4j.[87] At Cypher er deklarativ betyr at en Cypher-spørring, til forskjell fra Gremlin, beskriver tupler av dataelementer snarere enn å beskrive hvordan man finner dem. At Cypher er basert på mønstergjenkjenning betyr at *alle* tupler som svarer til databeskrivelsen velges som svar. Som sådan har Cypher mer til felles med SPARQL,[88] et spørrespråk spesifikt designet for RDF-grafer som vi kommer tilbake til under.

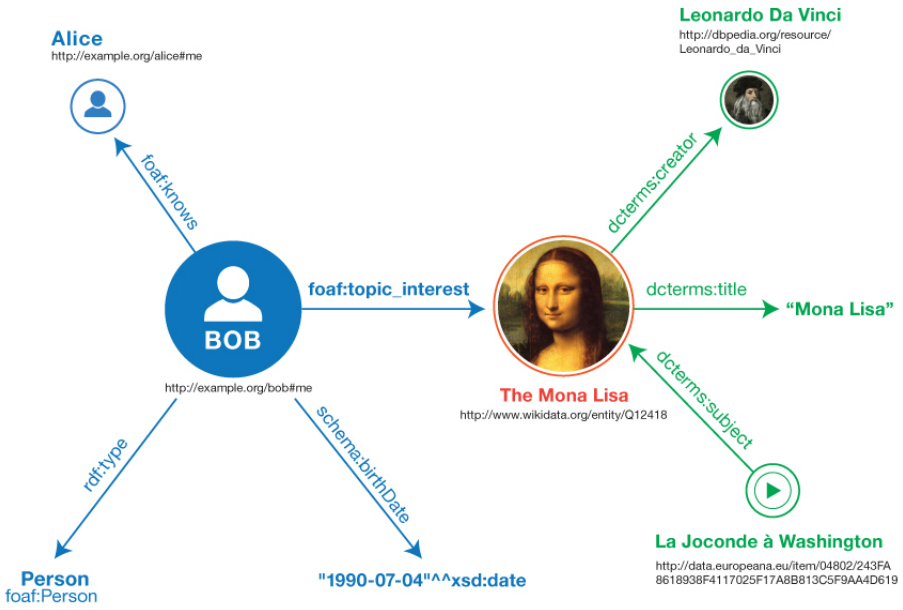
3.2.2 The Resource Description Framework (RDF)

RDF (*The Resource Description Framework*) ble opprinnelig designet i 1999 som en modell for maskinleselige beskrivelser av lenker mellom nettsider. Tanken var i utgangspunktet å støtte mer betydningsorientert navigering og søk gjennom å gjøre semantikken til hypertekst mer eksplisitt enn i ren HTML.

Gjennom årene siden, har RDF vokst til å bli en modell for generell kunnskapsrepresentasjon som er spesielt innrettet for dataintegrasjon og federering på Internett. Sagt annerledes er RDF en standard for å beskrive data i et web-miljø på en måte som gjør tolkningen av dataene uavhengig av opprinnelsessystem og fysisk plassering. RDF-standarden oppnår dette ved å utnytte internettets egen adresseringsmekanisme, altså URIer, som identifikatorer for dataelementer. Hvert dataelement eller objekt man ønsker å si noe om forankres dermed i en nettadresse, dvs. i ett veldefinert, unikt punkt innenfor rommet av alle IP-adresser. RDF-objekter kan derfor refereres til uten tvetydighet fra et hvilket som helst annet punkt på nettet, og denne referansen er selv en nettadresse som kan brukes til å navigere til det relevante datasettet. Dette kalles gjerne *lenkede data*.

Alle påstander i RDF uttrykkes med tripler på formen subjekt-predikat-objekt der hver av triplens bestanddeler er en nettadresse. En samling av tripler utgjør til sammen en RDF-graf som matematisk betraktet er en rettet multigraf med navngitte eller merkede kanter. Ulikt PGM-modellen kan ikke RDF-noder og -kanter inneholde ytterligere data i form av nøkkel/verdi-par eller noe annet. Noder og kanter i RDF er rene identifikatorer, og all informasjon må uttrykkes i grafstrukturen selv. Figur 3.15 gir et eksempel.

[85] https://github.com/tinkerpop/gremlin/wiki
[86] https://tinkerpop.apache.org/gremlin.html
[87] https://neo4j.com/
[88] https://www.w3.org/TR/rdf-sparql-query/



Figur 3.15 En RDF-graf.[89]

En egenskap ved RDF-standarden som har generert mye oppmerksomhet er at den har et logisk fundament som gjør den kompatibel med mer abstrakte beskrivelsesspråk slik som OWL (*The Web Ontology Language*). Mens RDF uttrykker forhold mellom enkeltobjekter, klatrer OWL et trinn lenger på abstraksjonsstigen, opp til generelle relasjoner mellom klasser av ting. En beskrivelse av generelle forhold mellom klasser er essensielt en logisk teori om vesentlige konseptuelle sammenhenger innenfor et område. Den kalles derfor gjerne en ontologi (en teori om det værende) eller, med en muligens mer edruelig terminologi, en *begrepsmodell*.

Som oftest brukes en begrepsmodell som et abstraksjonslag, eller som en linse over RDF-data. De generelle påstandene i begrepsmodellen uttrykker hvordan data fra ulike kilder forholder seg til hverandre betydningsmessig, og glatter ut semantisk heterogenitet: er disse typene koordinater forenlige? Er disse kodelistene overlappende, eller er felleselementene homonymer? Er denne klassen av kjøretøy pansrede? Og så videre.

Ved at OWL er så nøyaktig designet med hensyn på beregnbarhetsegenskaper, vil svaret på slike spørringer ofte kunne utledes automatisk, i en prosess som kalles kunstig resonnering, selv om svaret hverken er eksplisitt tilstede i datasettet eller på noen måte foregrepet av utgiveren. Det er i denne forstanden at RDF ofte kalles en semantisk teknologi: den gjør det mulig å beregne logisk konsekvens, dvs. hva påstander innebærer.

Figur 3.16 gir en kondensert sammenlikning av RDF og PGM.

3.2.3 Typiske analysekapabiliteter

Grafteori er en svært velstudert matematisk disiplin som står i et symbiotisk forhold til algoritmikk og kompleksitetsteori: svært mange algoritmiske problemer kan formuleres grafteoretisk og svært

[89] https://www.w3.org/TR/rdf11-primer/

	PGM-grafer	RDF-grafer
Type	*rettede multigrafer med navngitte kanter*	*rettede multigrafer med navngitte kanter*
Navn/ID	*strenger*	*URIer*
Noder	*Navn + nøkkel/verdi par*	Navn (ingen indre struktur)
Skjemaspråk	*Ingen*	*OWL* + andre

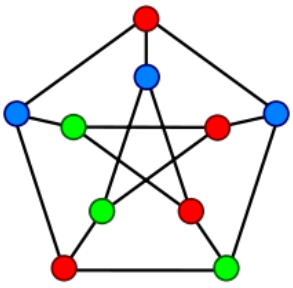
Figur 3.16 Sammenlikning av RDF og PGM.

mange grafteoretiske problemer har en algoritmisk løsning med kjente kompleksitetsegenskaper. Grafer er svært versatile og fleksible datastrukturer, og mange typer av problemer lar seg naturlig formulere som abstrakte grafproblemer. Algortimikk studerer hvordan disse problemene kan løses i en trinnvis deterministisk prosess, mens kompleksitetsteori er opptatt av å beskrive hvor store ressurser en algoritme vil kreve (typisk målt i regnekraft og minne) som en funksjon av størrelsen på et konkret problem (asymptotisk analyse). Et abstrakt grafproblem har som regel mange konkrete instanser som overfører teknikker og resultater til anvendte disipliner. Dette er selvsagt godt beskrevet andre steder (f.eks. Papadimitriou (1994)), og vi nøyer oss her med et par forsvarsnære eksempler:

3.2.3.1 Kognitiv radio og fargelegging

En kognitiv radio er en programmerbar radio som kan konfigureres til å benytte frekvenser eller tilgjengelige kommunikasjonskanaler på en måte som unngår eller reduserer interferens og støy. En slik radio oppdager automatisk tilgjengelige kanaler i det elektromagnetiske spekteret og endrer sine overføringsparametre dynamisk for til enhver tid å tillate mest mulig samtidig trådløs kommunikasjon innenfor et gitt geografisk område. Kognitiv radio studeres bl.a. i forbindelse med militær bruk av mobile ad hoc-nettverk i situasjoner der stasjonær infrastruktur enten er slått ut eller er for kostbar eller risikabel (se f.eks. Yu (2011)).

Kognitiv radio, og frekvensallokering mer generelt, er et grafteoretisk fargeleggingsproblem – jf. Maan & Purohit (2012) og Riihijarvi et al. (2005). I frekvens-/kanalallokeringsproblemet vil nodene i en graf representere radiosendere og/eller -mottakere. Kantene forbinder sendere som på en eller annen måte vil forstyrre hverandre dersom de sender på samme frekvens, f.eks. ved at de ligger for nære hverandre geografisk. Forskjellige frekvenser eller kanaler representeres med ulike farger.



Figur 3.17 En fargelegging av noder i en såkalt Petersen graf vha. tre farger – det minste mulige antallet.

Frekvensallokering er problemet om hvordan man fargelegger *nodene* i en graf på en slik måte at ingen noder som er forbundet med en kant har samme farge, jf. figur 3.17.

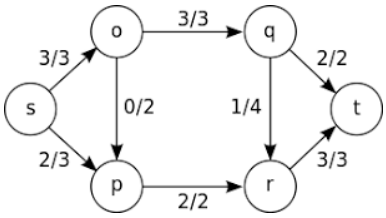
Fra kompleksitetsteori vet vi at dette fargeleggingsproblemet er et såkalt NP-problem (det kan beregnes i ikke-deterministisk polynomiell tid). Det vil si at maskinintelligens eller heuristikk er nødvendig for å hindre problemet i å vokse eksponensielt og oversvømme tilgjengelig regnekraft og minne. Det er dog investert mye forskning i algoritmer som yter godt i praksis, se Lewis (2015) for en oppdatert oversikt.

3.2.3.2 Logistikk og flytnettverk

Logistikk kan beskrives som kunnskapen om å planlegge, organisere og administrere ressurser i en kompleks operasjon, mer spesifikt om å opprettholde en flyt av ressurser fra et utgangspunkt til en destinasjon gjennom operasjonens varighet. En ressurs kan i denne sammenhengen være en vare slik som mat, utstyr og personell, eller det kan være abstrakte størrelser slik som tid, informasjon og tjenester.

I forsvarssammenheng dreier logistikk seg om å opprettholde forsynings- og kommunikasjonslinjer; en styrke som mangler begge deler er i praksis satt ut av spill. En logistikkoffiser er ansvarlig for å planlegge hvor og hvordan varer og tjenester skal forflyttes samt i hvilke mengder.

Dette problemet kan også formuleres grafteoretisk ved hjelp av flytnettverk (*flow networks*). Et flytnettverk er en rettet graf med én node markert som kilde og én node markert som mål. Hver kant i et flytnettverk bærer en bestemt ressursmengde, kalt flyt, som er begrenset av kantens kapasitet. Nettverket må oppfylle betingelsen at den samlede flyten inn til en node er lik den samlede flyten ut av den. Unntaket er kilden som kun har utgående flyt og målet som kun har innkommende flyt. En flyt (i bestemt form entall) i et nettverk er en tilordning av flyt (i ubestemt form entall) til hver kant i nettverket under denne begrensningen. Figur 3.18 gir et eksempel på et slikt nettverk.



Figur 3.18 Et flytnettverk, med kilde s og mål t. En kant som er markert n/m har kapasitet m og bærer flyt n<m.

Verdien til en flyt i et nettverk defineres som summen av flyten til hver av kantene som går ut fra kilden. Dette tallet representerer den samlede mengden ressurser som denne bestemte flyten transporter fra kilden til målet. I en logistikkoperasjon vil man være interessert i en maksimal flyt, dvs. i å transportere ressurser langs kantene i nettverket på en slik måte at nettverket utnyttes til sin fulle kapasitet. Dette problemet er kjent fra algoritmikk som *the maximum flow problem*. Kompleksitetsteori har vist at problemet kan løses i polynomiell tid, hvilket vil si at det er effektivt beregnbart i alle tilfeller. Det er som regel enkelt å finne implementasjoner av klassiske og velstuderte algoritmer, f.eks. Edmonds-Karp eller Dinics algoritme.

Fargelegging og maksimal flyt er kun to av mange eksempler på grafproblemer som har velstuderte algoritmiske løsninger (se f.eks. Garey et al. (1976)). Dette innebærer som regel at det finnes

optimaliserte algoritmer som er effektive i praksis, og som skalerer godt til store grafer. Grafer er en svært fleksibel datastruktur og abstrakte grafproblemer kan med litt kreativitet og domenekunnskap appliseres til mange forskjellige informasjonsforvaltningsproblemer hvor disse algoritmene kan utnyttes med god effekt.

Et problem slik som frekvensallokering eller logistikk er selvsagt ikke nødvendigvis et stordata-problem bare i kraft av å være et grafproblem – det avhenger av datamengdene. Når man nevner stordata og grafer i samme setning vil informatikere flest heller tenke på én av to ting; enten *sosial nettverksanalyse* eller *ontologibasert dataintegrasjon*. Dette er to anvendelser av grafdatabaser som med en viss rett kan sies å være iboende stordataproblemer: begge har det til felles at de forsøker å håndtere heterogene data fra ulike kilder i potensielt svært store mengder, den første av dem gjerne også i høy hastighet i form av sosiale mediestrømmer som Facebook, Twitter, o.a. Disse to anvendelsene fortjener en nærmere beskrivelse.

3.2.3.3 Sosial nettverksanalyse / Sosiometri

Sosial nettverksanalyse er en stordatateknologi som har fått mye oppmerksomhet de siste årene. Dette skyldes i stor grad noen få høyprofilerte selskaper som har skaffet seg store, ofte offentlige kontrakter innenfor etterretning, overvåking og politisk påvirkning.

Selskapet Palantir Technologies er et velkjent eksempel på hvordan stordatateknologier generelt og sosial nettverksanalyse stadig oftere tas i bruk av statlige organer. Produktet Palantir Gotham[90] brukes i dag av kontraterroranalytikere i USAs etterretningsfellesskap og Department of Defence (DoD). Systemet ble tidligere brukt av det amerikanske Recovery Accountability and Transparency Board for å ettergå og kontrollere offentlig forbruk (dog visstnok med vekslende hell (The Washington Times 2015)), samt av analytikere i det Canadiske forskningsprogrammet Information Warfare Monitor for å kartlegge cyberspace som strategisk domene. Palantir er også omtalt i kapittel 3.5.2.

Et annet eksempel som har fått mye pressedekning kommer fra selskapet Cambridge Analytica[91], et britisk konsulentselskap som kombinerer forskjellige former for *data mining*, deriblant sosial nettverksanalyse, med strategisk kommunikasjon for politisk påvirkning. Cambridge Analytica ble notoriske for sin såkalte mikroannonsering av politisk innhold på facebook i forbindelse med Donald Trumps valgkamp i 2016.

Sosial nettverksteori er en tverrfaglig disiplin basert på grafteori, statistikk, spillteori og sannsynlighet. En av dispilinens grunnleggere Jacob Moreno definerer feltets mål som *"the inquiry into the evolution and organization of groups and the position of individuals within them."* (Moreno 1934). Innenfor informatikken har sosial nettverksanalyse nærmest blitt ensbetydende med analyse av sosiale medier slik som Facebook, Twitter, YouTube og Instagram. Som Morenos definisjon antyder dreier det seg imidlertid om å traversere forbindelser mellom mennesker for å avsløre strukturelementer som gir en sosial gruppe en bestemt form og som bestemmer agenda, oppfatninger og den generelle informasjonsflyten innad i en gruppen eller mellom grupper. Digital sosial nettverksanalyse omfatter i dag algoritmer slik som

- identifikasjon av nettsamfunn (*community detection*),

[90] https://www.palantir.com/palantir-gotham/
[91] https://cambridgeanalytica.org/

- beregning av enkeltpersoners innflytelse og sentralitet (*betweenness* og *centrality*),
- avsløring av rykter og sporing av ryktespredning,
- sentimentanalyse og
- kartlegging av informasjonsflyt og sårbarheter.

Noen av disse vil være innebygget i kommersiell programvare slik som f.eks. den markedsledende grafdatabasen Neo4j[92] eller Amazons skybaserte RDF-database Neptune[93].

3.2.3.4 Ontologibasert dataintegrasjon

Ontologibasert dataintegrasjon (OBDA) er et forskningsfelt basert på automatisk ressonnering og regelbasert kunstig intelligens som utnytter RDF-standardens mere formelle aspekter (semantikk, kompleksitetsprofiler m.m.).

OBDA handler om å forene informasjon med overlappende betydning og relevans under et felles virtuelt spørregrensesnitt. Dette grensesnittet er en overordnet formell begrepsmodell (også kalt en ontologi) som uttrykker forholdet mellom datatyper i og på tvers av de ulike kildene (typisk, men ikke nødvendigvis, relasjonelle databaser). Disse kildene kan være utviklet og vedlikeholdt uavhengig av hverandre for ulike formål, og det virtuelle spørregrensesnittet vil ikke kreve at de endres. Mer spesifikt benyttes begrepsmodellen som en beregnbar spesifikasjon som gjør det mulig å omskrive spørringer mot begrepsmodellen til spørringer over de enkelte kildene, jf. figur 3.19.

Begrepsmodellen designes gjerne slik at den reflekterer brukerens foretrukne vokabular, noe som gjør en analytiker i stand til å uttrykke sitt informasjonsbehov med begreper som reflekterer hans eller hennes kompetanse.

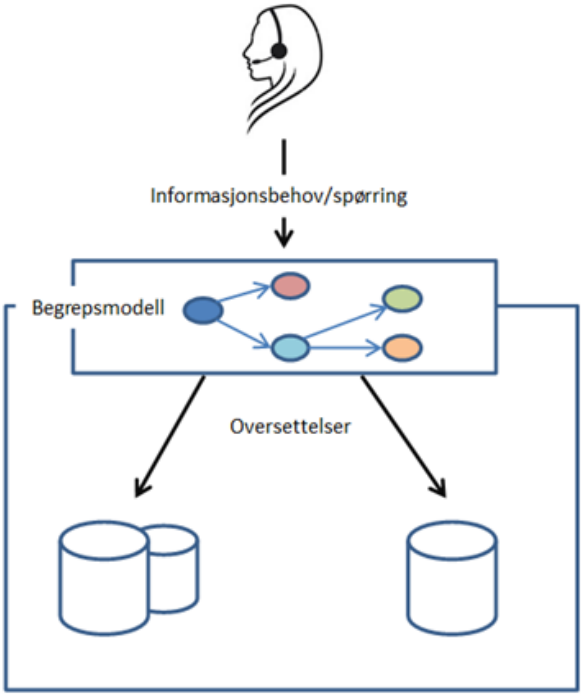
Et eksempel på en miltær anvendelse av denne teknologien er nærmere beskrevet i kapittel 4.4.

3.2.4 Arkitektur og ytelse

Svært mange konsepter og algoritmer innenfor grafteori er basert på å følge stier, dvs. å vandre fra ett objekt til et annet via relasjonene som forbinder dem. Dette er en beregningsoppgave som krever iterasjon, hvilket vil si at beregningsprosessen gjentas for å generere en serie av mellomresultater som med økende nøyaktighet approksimerer det ønskede resultatet.

Til tross for deres popularitet og modenhet, er mange konvensjonelle batch-orienterte stordatasystemer slik som MapReduce og Hadoop ikke gode på iterative beregningsoppgaver. For å gjenta et poeng fra kapittel 2.6, har det å gjøre med at disse systemene er persistenssystemer som er avhengige av å skrive mellomresultater til disk for hver iterasjon. Et antall nye primærminnegrafdatabaser har derfor sett dagens lys de senere årene. Det er naturlig i denne rapporten å begrense diskusjonen til de av dem som er dimensjonert for store datamengder, altså de som er utviklet for å spre data og beregninger over en vilkårlig og varierende stor klynge av maskiner (jf. kapittel 2.3 om horisontal skalerbarhet).

[92] https://neo4j.com/
[93] https://aws.amazon.com/neptune/



Figur 3.19 Ontologibasert dataintegrasjon.

På et generelt nivå kan man si at grafdatabaser, sammenliknet med relasjonelle databaser eller andre tabulære datamodeller, ofte er raskere for assosiative data som er tett forbundet med hverandre. Dette har å gjøre med kostnadene involvert i å følge lenker fra ett dataelement til et annet i tabulære datamodeller: selv om det ligger i ordet at relasjoner er en integrert del av relasjonsmodellen, så vil det å vandre i dataene gjennom å følge stier sammensatt av forskjellige relasjoner kreve at databasetabeller kombineres ved hjelp av *join*-operasjoner for å finne felles verdier. Selv om man muligens kun er interessert i én eller noen få rader i hver tabell, vil man også måtte prosessere overflødige rader som multipliseres med antallet kombinasjoner. I slike tilfeller vil relasjonsmodellen typisk tynges ned av store join-tabeller, sparsomt populerte rader og logikk for å håndtere nullverdier (Robinson et al. 2013).

Et annet problem med relasjonsdatabaser, dersom dataene er naturlige assosiative, er at en *join* av to tabeller ikke selv er et dataobjekt. Med attributtgrafer og RDF-grafer kan man knytte metadata til selve relasjonen, f.eks. ved å vekte en forbindelse, angi dens opprinnelse og gyldighet eller liknende. Relasjonelle relasjoner, derimot, kan ikke annoteres på denne måten. Siden grafer også er mer fleksible i den forstand at de ikke er er avhengige av et predefinert skjema som styrer hvordan et datasett vokser, kan man si at grafdatabaser er bedre tilpasset assosiative ad hoc datasett som endrer struktur over tid.

Utover disse generelle betraktningene, er det svært mange faktorer som avgjør hvor godt en grafdatabase yter på et bestemt område og hvor godt den skalerer. For eksempel kan den underliggende lagringsløsningen ha stor betydning for ytelsesprofilen til en grafdatabase. Grovt sett finnes det to typer:

- Opprinnelige (*native*) grafdatabaser er databaser som er designet som grafer fra bunnen av, med en lagringsstruktur som reflekterer noder og kanter i grafmodellen direkte. Karakteristisk for disse er at de benytter indeksfrie referanser mellom nodene i grafen, hvilket vil si at nodene i grafen fysisk peker til hverandre. Eksempler på slike systemer er Google Pregel og dens åpen kildekode-motstykke Apache Giraph, Amazons skybasert RDF-database Neptune og PGM-databasen Neo4j.
- Ikke-opprinnelige (*non-native*) grafdatabaser er grafdatabaser som er implementert over en arkitektur som i utgangspunktet ikke er designet for grafer spesielt. Det kan være en relasjonsdatabase, en dokumentdatabase slik som MongoDB eller en nøkkel/verdi-database slik som Apache Cassandra. I ikke-opprinnelige grafdatabaser er nodene og kantene i grafen lagret i en tabell, et dokument, eller hvilken som helst datastruktur den underliggende lagringsløsningen måtte tilby, og grafen utgjør et ekstra abstraksjonslag over den fysiske implementasjonen. Eksempler på slike grafdatabaser er RDF-databasen Virtuoso (relasjonell), ArangoDB (dokumentdatabase) og Titan (nøkkel/verdi).

En hovedforskjell mellom disse er at en ikke-opprinnelig database vil trenge en sekundær indeks for å kombinere data, noe som i sin tur gjør at slike databaser vil ha en tendens til å bli tregere for spørringer med mange joins etterhvert som datasettet øker i størrelse og antall koblinger. Ytelsen til en opprinnelig grafdatabase vil derimot forbli noenlunde konstant, da relasjoner er fysisk implementert som pekere mellom nodene direkte. Ikke-opprinnelige grafdatabaser på sin side vil ofte kunne utnytte at den underliggende lagringsløsningen er optimalisert for enkle beregningsoppgaver som gjentas mange ganger over et stort antall dataelementer med statisk struktur.

3.2.5 Eksempler

3.2.5.1 Amazon Neptune

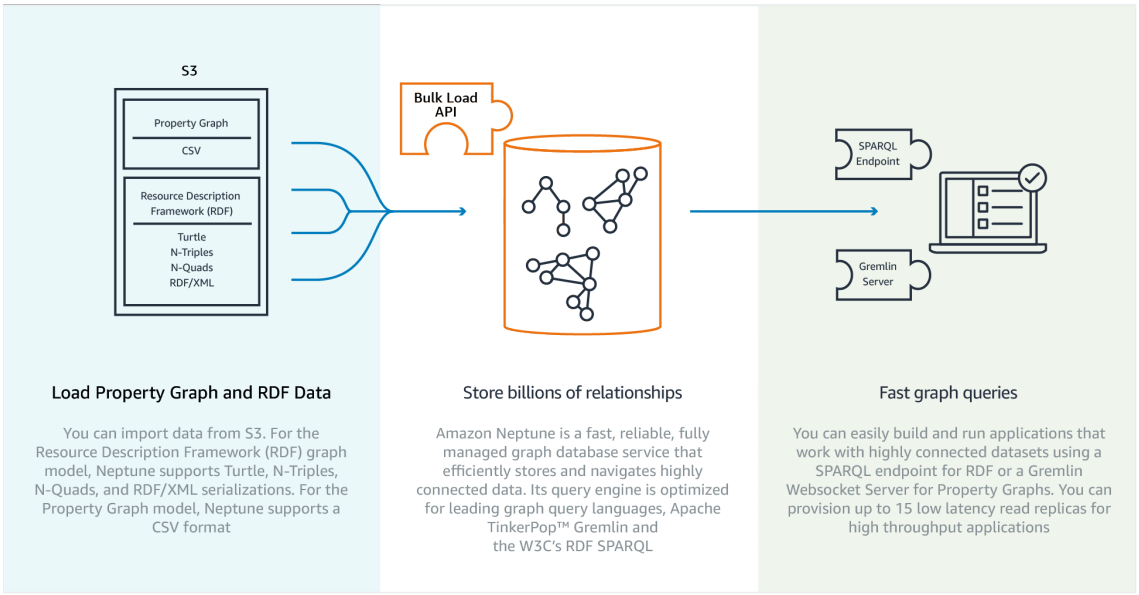
Amazon Neptune er en skybasert grafdatatjeneste som kjører på Amazon Web Services (AWS), og har blitt brukt for å håndtere data i forbindelse med svindeldeteksjon, medisin- og livsvitenskap samt nettverksoperasjoner.

Amazon Neptune støtter både attributtgrafer (PGM) og RDF samt de assosierte spørrespråkene Gremlin og SPARQL (se figur 3.20). Avansert prosessering av data tilbys gjennom Apache TinkerPop-grensesnittet.

Neptune er transaksjonsbasert, og tilbyr ACID-garantier samtidig med høy ytelse og oppetid. Systemet tilbyr bl.a. kontinuerlig backup i Amazon S3 som gjør det mulig å gjennopprette data med punkt-i-tid-presisjon. Neptune støtter for øvrig kryptering av data i transit samt backup.

Amazon Neptune er en viderutvikling av Systap BlazeGraph, som var en grafdatabase basert på åpen kildekode, spesialbygget for å kunne utnytte regnekraften i GPUer. Etter at Amazon kjøpte opp selskapet, har all videreutvikling av BlazeGraph som eget system opphørt og all videre innsats er kun på skybaserte Neptune. Grafmotoren i bunnen av BlazeGraph, og derfor også for Neptune, bruker RDF-datamodellen internt og utnytter så et standardisert serialiseringsregime for å håndtere

[94] https://aws.amazon.com/neptune/



Figur 3.20 Neptune-prosess[94].

attributtgrafer. Dette gjør det mulig å bruke både RDF- og attributtgraf-grensesnittene om hverandre på samme datasett uten å måtte endre data.

Se figur 3.21 for en overordnet oppsummering av Amazon Neptune.

Analysestøtte	Avansert søkemekanisme; mønstre/stier i grafen (SPARQL/Gremlin)
Programmeringsmodell	Iterator og traverseringsbasert (TinkerPop)
Skalerbarhet	Horisontal og vertikal skalerbar lagringsplass og regnekraft
Tilgjengelighet vs. konsistens	ACID
Sårbarhet og feiltoleranse	Feiltoleranse vha. kontinuerlig backup og datareplisering
Primærminne- vs. persistenssystem	Primærminne, skybasert (AWS)
Iterative beregninger	Egnet for iterative oppgaver (TinkerPop)
Input/output-profil	OLTP
Gjenbrukbarhet av data	Høy ved bruk av RDF, lav ved bruk av PGM

Figur 3.21 Amazon Neptune, raskt oppsummert.

3.2.5.2 Janusgraph

JanusGraph er en skalerbar, distribuert grafdatabase som er optimalisert for lagring og sanntids-analyse. Systemet støtter distribusjon over maskinclustere og legger til rette for eksekvering av avanserte spørringer og graftraversering i sanntid.

JanusGraph implementerer Apache TinkerPop-teknologistacken, som betyr at den kan regnes som en grafdatabase iht. attributtgraf-modellen (PGM). Spørrespråket som tilbys er Gremlin, som direkte følge av at den implementerer TinkerPop-spesifikasjonen.

JanusGraph er transaksjonsbasert og støtter både ACID og BASE etter behov. Persistering av grafer tilbys gjennom støtte for Apache Cassandra, HBase, Google Bigtable og Oracle BerkeleyDB, og

analysefunksjonalitet gjennom Apache Spark, Giraph og Hadoop. Egenskapene som systemet innehar, f.eks. ACID/BASE, vil derfor være diktert av hvilke komponenter man velger å bruke. For eksempel vil fokus på konsistens fordre HBase som persisteringsløsning, mens tilgjengelighet vil peke mot å bruke Cassandra.

Analysestøtte	Avansert søkemekanisme; mønstre og stier i grafen (Gremlin)
Programmeringsmodell	Iterator og traverseringsbasert (TinkerPop)
Skalerbarhet	Horisontal og vertikal skalerbar lagringsplass og regnekraft
Tilgjengelighet vs. konsistens	ACID eller BASE, etter behov
Sårbarhet og feiltoleranse	Feiltoleranse vha. datareplisering og persistering
Primærminne- vs. persistenssystem	Primærminne, men støtter persistering gjennom Apache HBase, Apache Cassandra, Google Bigtable eller Oracle BerkeleyDB
Iterative beregninger	Egnet for iterative oppgaver (TinkerPop)
Input/output-profil	OLTP og OLAP, basert på valg av underliggende komponenter
Gjenbrukbarhet av data	Lav. Avhengig av applikasjonslogikk

Figur 3.22 JanusGraph, raskt oppsummert.

JanusGraph er åpen kildekode, koordinert av The Linux Foundation der bl.a. Google, IBM, Hortonworks og Amazon er grunnleggere og partnere i prosjektet. Systemet er en videreutvikling av kodebasen til TitanDB, en distribuert grafdatabase som på grunn av endringer i eierskap og påfølgende konflikter[95] ikke lenger er aktivt utviklet.

Se figur 3.22 for en overordnet oppsummering av JanusGraph.

3.3 Strømmesystemer

Som tidligere nevnt flommer verden i dag over av data, og mange av disse dataene kjennetegnes av at de er strømmende. En strømmende datakilde gir er en kontinuerlig og vilkårlig lang sekvens av data. Eksempler på slike datakilder er værsensorer, videoovervåkningssystemer og datastrømmer fra sosiale medier.

En spesielt interessant kilde for strømmende data er tingenes internett (*Internet of Things – IoT*), som er en teknologitrend og et konsept basert på en idé om å koble sammen fysiske ting gjennom en virtuell representasjon av dem på internett. Hensikten er å gjøre det mulig for objekter som er fremstilt helt uavhengig av hverandre, og for ulike formål, å bli koordinert over verdensveven og å utveksle informasjon. Når flere og flere fysiske gjenstander slik som bygninger, biler, armbåndsur, kjøkkenapparater m.m. produseres med innebygget elektronikk og nettverkskapabiliteter, er dette en nærliggende tanke, og med internettoppkobling vil samtidig slike enheter kunne være datakilder.

Strømmende datakilder har potensiale til å bidra med fersk informasjon som kan hjelpe oss å bygge opp en forståelse av situasjonen rundt oss, men for å utnytte dette potensialet må vi være i stand til å behandle disse dataene raskt nok til at informasjonen som kan avledes fortsatt er nyttig.

Som eksempel kan man tenke på sanntidsanalyse av en strøm fra et overvåkingskamera brukt til perimetersikring rundt en kommandoplass. Hva er objektene som fanges opp av kameraet? Er noen

[95] https://www.datanami.com/2017/01/13/janusgraph-picks-titandb-left-off/

av dem trusler? Når kan man eventuelt forvente at de er innen rekkevidde av nærforsvarssystemene? Dette er spørsmål som en operatør vil ønske svar på så snart situasjonen oppstår.

De strømmende dataene må med andre ord analyseres i sanntid eller nær sanntid. Informasjonssystemer som er designet for å håndtere denne utfordringen kalles i denne sammenhengen for *strømmesystemer*.

Den største fordelen ved å behandle data som en strøm, er at analyse kan gjøres fortløpende og resultater kan leveres raskt. For å klare dette, holder og prosesserer strømmesystemer typisk data og resultater i minnet. På den måten unngås den tidkrevende lesingen og skrivingen av data og resultater til disk. Dette betyr imidlertid at strømmesystemer stiller store krav til minne, og det betyr også at med mindre man også har en prosess som også skriver data til disk vil dataene i et strømmesystem gå tapt når systemet slås av. De fleste strømmesystemer har imidlertid også mulighet for å persistere de strømmende dataene.

3.3.1 Prosessering og ytelse

Prosessering av strømmer kan gjøres som ren strøm-prosessering (dvs. at hvert enkelt dataobjekt behandles når det ankommer) eller mikro-batch-prosessering (dvs. at dataobjektene samles i små batcher før prosessering). Ren strøm-prosessering vil kunne gi lavere forsinkelse enn mikro-batching, men på bekostning av en høyere kompleksitet for å håndtere bl.a. kontrollmekanismer og feiltoleranse (Carbone et al. 2017).

En måte å gjøre strømmesystemer robuste med hensyn på feil i data eller (mellom-)resultater, er å basere seg på uforanderlige (*immutable*) data. Dette innebærer kumulativ lagring av data – nye data lagres når de ankommer og endres ikke. Er det behov for endringer introduseres endringene som oppdaterte, nye data. Dette står i kontrast til f.eks. relasjonelle databaser der oppdatering av eksisterende celler er en fundamental operasjon. Gjennom å gjøre dataene uforanderlige kan feil rettes ved å re-prosessere dataene (som altså ikke er endret siden de ankom systemet), eventuelt med feilaktige data fjernet. Å basere seg på uforanderlige data krever imidlertid en evne til å lagre enorme mengder data, og dersom dette skal kombineres med prosessering i sanntid stiller det store krav til hvor mye data som kan behandles i primærminnet.

Siden moderne strømmesystemer som Apache Spark og Apache Flink (se senere i dette kapitlet for beskrivelse av disse) er distribuerte primærminnesystemer, er ytelsen først og fremst avhengig av hvor mange noder systemet består av og hvor mye primærminne disse nodene har. Som et eksempel kan det vises til (Metzger & Ward 2018) der forfatterne beskriver hvordan man kan sette opp en instans av Apache Flink som prosesserer 1 000 000 meldinger i sekundet (2 GB/s) med en forsinkelse på under ett sekund.

3.3.2 Analyse av datastrømmer

Som tidligere nevnt er det typisk for analyseoppgaver på strømmende data at de ønskes besvart i sanntid eller nær sanntid. Siden strømmesystemer typisk holder data og mellomresultater i primærminnet er de godt egnet til dette.

Systemer for prosessering av datastrømmer har i hovedsak utviklet seg langs to hovedspor som har resultert i to hovedmodeller (Cugola & Margara 2012): Strømprosessering (*Data Stream Processing – DSP*) og kompleks hendelsesprosessering (*Complex Event Processing – CEP*). Målet for begge disse modellene er å prosessere data når de ankommer for å kunne gi umiddelbare varsler eller analysesvar. Dette står i kontrast til prosessering sentrert rundt tradisjonelle databaser der data lagres og indekseres i påvente av at en bruker skal be om at de prosesseres.

DSP kan spore sine røtter til tradisjonelle databaser og er i bunn og grunn en utvidelse av disse for å kunne håndtere en strøm av data framfor kun å forholde seg til en statisk database. Med DSP vil man typisk registrere spørringer som kontinuerlig stilles mot datastrømmen inntil de slettes eller deaktiveres. Dette betyr at svaret på en spørring vil endre seg etterhvert som nye data ankommer, dette til forskjell fra tradisjonelle databaser der spørringer utføres på kommando og forventes å gi eksakte svar. DSP innebærerer vanligvis at man prosesserer flere forskjellige datastrømmer, og at svarene på de kontinuerlige spørringene selv vil utgjøre en strøm.

CEP, på sin side, betrakter ankommende data som hendelsesnotifikasjoner (*event notifications*) som indikerer hendelser som kan filtreres og kombineres for å forstå fenomener i den virkelige verden. Denne modellen har sitt opphav i abonneringssystemer (*publish-subscribe-systemer*). CEP involverer ofte å sammenstille de ankomne hendelsene med informasjon fra andre, gjerne ikke-strømmende, kilder for å kunne bestemme hvilke bakenforliggende, komplekse hendelser som er opphavet til de ankomne dataene. Et eksempel kan være å konkludere med at observasjon av et helikopter satt sammen med oppfangede nødsignaler tyder på at en motstander er i ferd med å gjennomføre en redningsoperasjon.

3.3.3 Batch eller strøm?

Batch-behandling – å først samle opp store mengder data før disse behandles på én gang – sees gjerne på som motsatsen til strømbehandling, som betyr at data behandles med én gang de ankommer systemet.

Som tidligere nevnt, holder strømmesystemer typisk dataene som skal behandles i primærminnet (primærminnesystem, ref. 2.6). Det betyr at mengden data som kan prosesseres er begrenset av det tilgjengelige minnet, og selv om maskinvare med stort minne stadig blir billigere betyr det fortsatt at man må ha relativt kostbar maskinvare for å kunne prosessere store datamengder.

For batchprosessering, derimot, lagres dataene typisk på disk (persistenssystem, se kapittel 2.6) og behandles derfra. Siden prosessen å skrive og lese data fra disk er tidkrevende, er batchsystemer tregere enn systemer som kan holde alle nødvendige data og (mellom-)resultater i minnet.

Batch-systemer er begrenset av diskplass og hvordan prosesseringen parallelliseres. Siden diskplass bare blir billigere og billigere, og distribueringsalgoritmene bedre og bedre, kan batch-systemer typisk kjøres på relativt rimelig maskinvare.

Et eksempel på et mye brukt batch-system er Hadoop som beskrives nærmere i kapittel 3.4.1.

3.3.3.1 Lambda- og Kappa-arkitektur

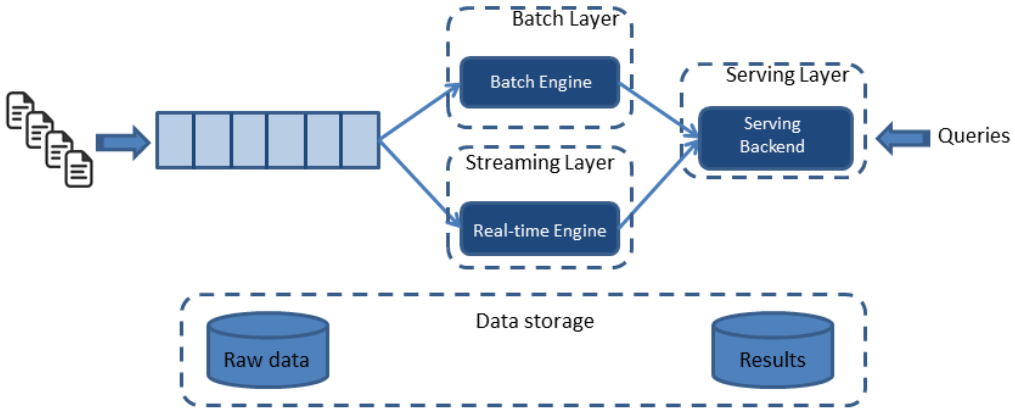
Lambda-arkitekturen (Marz & Warren 2015) ble presentert som en måte å forene batch- og strømprosessering av stordata. Den bygger på en erkjennelse at man trenger begge prosesserings-

regimene, både batch-prosesseringens tilrettelegging for å håndtere enorme mengder data og strømprosesseringens mulighet til å prosessere data raskt for å gi raske analysesvar.

Arkitekturen bygger på en lagvis tilnærming med tre lag, se figur 3.23:

- Batch
- Serving
- Speed

Batch-laget sørger for lagring av alle dataene, og kjører med jevne mellomrom en prosess som behandler de lagrede dataene. Utfordringene med batch-prosessering er at det er så tidkrevende at resultatene som er gitt på grunnlag av dataene prosessert av batch-laget vil være utdaterte. For å kompensere for dette har man speed-laget. Dette laget har som oppgave å prosessere dataene som er ankommet etter at siste batch-prosess ble satt igang. Et endelig analysesvar fra et slik system vil bli satt sammen i serving-laget.



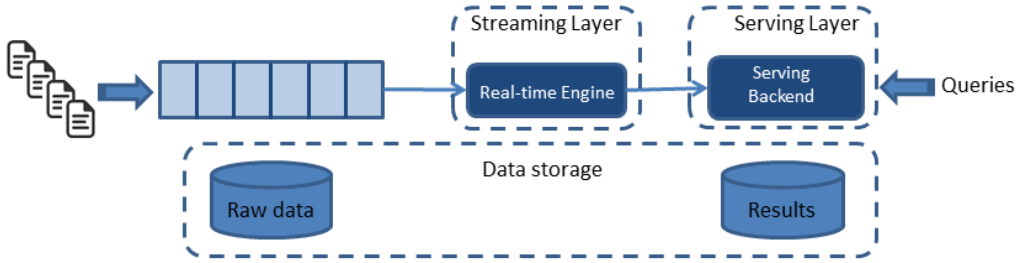
Figur 3.23 Lambda-arkitekturen(Seyvet & Viela 2016).

Lambda-arkitekturen var lenge den dominerende arkitekturen for stordatasystemer, helt til den ble utfordret av Kappa-arkitekturen presentert av LinkedIn-ingeniøren Jay Krebs i blogg-innlegget (Krebs 2014). Der tas det til orde for heller å behandle alt som strømmer. Batch-prosessering ses dermed på som et spesialtilfelle av strømprosessering. Hovedbegrunnelsen for denne tilnærmingen var å komme unna den systemmessige kompleksiteten ved lambda-arkitekturen. Sammensetting av resultater fra serving- og speed-laget har vist seg å være komplisert, og eventuelle nye algoritmer må holdes vedlike som separate programvarer for å kunne kjøres i de forskjellige lagene.

Et annet poeng med kappa-arkitekturen er at den utnytter de mer kapable strømmesystemene som nå finnes på markedet er i stand til å holde nok data i primærminnet til å gjøre batch-jobben der.

Kappa-arkitekturen kan illustreres som vist i figur 3.24.

Kappa-arkitekturens kjerne er uforanderlige (*immutable*) data. Håndtering av de innkommende dataene sees på som å håndtere en logg; Nye data blir lagt til loggen og aldri slettet. På den måten kan f.eks. re-prosessering av data med en oppdatert algoritme gjennomføres ved at dataene spilles av og prosesseres på nytt.



Figur 3.24 Kappa-arkitekturen(Seyvet & Viela 2016).

3.3.4 Eksempler

3.3.4.1 Apache Kafka

Apache Kafka er et distribuert meldingsverktøy som organiserer innkommende data i merkede køer (*topics*) slik at klienter kan tegne abonnement på data de er interessert i og motta disse når de ankommer Kafka. Dataene som lagres er uforanderlige (*immutable*) data. Det betyr at alle dataelementer som går gjennom Kafka lagres og aldri endres. Ved behov for reprosessering av resultater hentes bare de gamle dataene fram igjen.

Kafkas plass i et distribuert strømmesystem er først og fremst som en meldingsbuss som kan fange de strømmende dataene, lagre disse og gi dem videre til andre komponenter som skal stå for analyse og videre prosessering. I denne rollen er Kafka ledende på markedet, og brukes av store aktører som LinkedIn, Twitter, PayPal, Spotify og New York Times.

Se figur 3.25 for en overordnet oppsummering av Apache Kafka.

Dataanalyse	Prosesserer ikke data
Programmeringsmodell	Prosesserer ikke data
Skalerbarhet	Skalerbarhet i datamengde og regnekraft
Tilgjengelighet vs. konsistens	Prioriterer tilgjengelighet (AP) framfor konsistens (CP)
Sårbarheter og feiltoleranse	Replikering med en konfigurerbar replikeringsfaktor per kø
Primærminne- vs. persistenssystem	Primærminnesystem
Støtte for iterative beregninger	Prosesserer ikke data
Input/output-profil	Uforanderlige data, støtter mao. ikke Update og Delete.
Gjenbrukbarhet av data	Avhengig av formatet til informasjonen som sendes. Kafka kun videredistribuerer informasjonen

Figur 3.25 Apache kafka, raskt oppsummert.

3.3.4.2 Apache Spark Streaming

Apache Spark er et rammeverk designet for distribuert prosessering av store datamengder (se også kapittel 3.4.2 for en beskrivelse av Spark som programmeringsrammeverk) og Apache Spark Streaming er en utvidelse for strømprosessering. Spark er i utgangspunktet designet for batch-prosessering, så strømprosesseringen er implementert som såkalt mikro-batch-prosessering. De strømmende dataene samles i små batcher, ned i mikrosekunder, som deretter prosesseres.

Spark ble utviklet ved University of California, Berkeleys AMPLab, men ble i 2013 overlatt til Apache Software Foundation som åpen kildekode.

Spark ble utviklet som et svar på at den dominerende algoritmen for stordataprosessering, MapReduce, led under å være basert på hyppig, tidkrevende diskaksess og dermed relativt sen. For å bøte på dette holder Spark dataene som skal prosesseres i minnet istedenfor å lagre de på disk. Dette gjør Spark rask og velegnet til iterative prosesseringsoppgaver fordi den ikke trenger å skrive mellomresultater til disk. Spark benytter andre verktøy til persistering av data på disk, og brukes ofte sammen med HDFS (Hadoop Distributed File System).

Se figur 3.26 for en overordnet oppsummering av Apache Spark Streaming.

Dataanalyse	Maskinlæring, sosial nettverksanalyse, anbefalingssystemer m.m.
Programmeringsmodell	Funksjonell/MapReduce
Skalerbarhet	Horisontalt skalerbar lagring og regnekraft
Tilgjengelighet vs. konsistens	Vektlegger konsistens (CP)
Sårbarheter og feiltoleranse	Feiltoleranse vha. avstamningstrær
Primærminne- vs. persistenssystem	Primæreminne med mikro-batcher
Støtte for iterative beregninger	Velegnet for iterative oppgaver
Input/output-profil	Avhengig av lagringsløsning
Gjenbrukbarhet av data	Avhengig av lagringsløsning

Figur 3.26 Apache Spark Streaming, raskt oppsummert.

3.3.4.3 Apache Flink

Apache Flink er et strømmebasert rammeverk for distribuert dataanalyse, og ansett som en konkurrent til Apache Spark. Mens Spark er designet med utgangspunkt i batch-prosessering, er Flink designet med tanke på strømprosessering og prosesserer derfor dataene som en strøm (altså én etter én ettersom de ankommer systemet) istedenfor som mikro-batcher. Dette gjelder også batch-prosessering, som Flink gjør gjennom å samle opp data i en strøm over et gitt tidsrom og så gjøre prosesseringen når oppsamlingen er fullført.

Flink kommer med egne biblioteker for maskinlæring og grafprosessering, henholdsvis FlinkML Gelly og Flink Gelly.

Flink startet som Stratosphere – et akademisk prosjekt ved Berlins tekniske universitet – i 2010, og ble et Apache-prosjekt i 2014. Systemet sikter seg i stort inn på det samme markedet som Spark, og tilbyr en forbedring i ytelse på ren strømprosessering (Krettek & Winters 2017). Spark er imidlertid et mer modent produkt, og ligger fortsatt et hestehode foran Flink når det gjelder utbredelse og dermed tilgang til treningsmateriale og tredjeparts-biblioteker.

Som Spark kommer heller ikke Flink med egen lagringsløsning, men brukes ofte sammen med Hadoop Distributed File System.

Se figur 3.27 for en overordnet oppsummering av Apache Flink.

Dataanalyse	Hendelsesprosessering, maskinlæring, grafprosessering, m.m.
Programmeringsmodell	Funksjonell
Skalerbarhet	Horisontalt skalerbar lagring og regnekraft
Tilgjengelighet vs. konsistens	Vektlegger konsistens (CP)
Sårbarheter og feiltoleranse	Feiltoleranse gjennom kontinuerlig lagring av tilstand (*Asynchronous Barrier Snapshotting*)
Primærminne- vs. persistenssystem	Primærminne
Støtte for iterative beregninger	Velegnet for iterative oppgaver
Input/output-profil	Avhengig av lagringsløsning
Gjenbrukbarhet av data	Avhengig av lagringsløsning

Figur 3.27 Apache Flink, raskt oppsummert.

3.4 Programmeringsrammeverk

Et stordata (programmerings-)rammeverk er et ferdigpakket sett av programvarebiblioteker og lagringsløsninger som ved hjelp av brukerkode kan tilpasses spesifikke problemer og behov. Et rammeverk definerer som regel en standard måte å bygge og distribuere applikasjoner på, som gir et interoperabelt og gjenbrukbart kjøremiljø for programvare. Mer spesifikt vil et stordatarammeverk vanligvis inneholde:

- En programmeringsmodell (API) for å koble sammen databaseteknologi, algoritmer og visualiseringsverktøy,
- adaptere og parsere for å koble til ulike databaseteknologier og filformater,
- et administrasjonsverktøy for å konfigurere og drifte en maskinklynge (*cluster manager)*,
- programvarebiblioteker for statistisk analyse og maskinlæring og
- programvarebiblioteker for plotting av grafer, og andre visualiseringsverktøy.

Et rammeverk er som regel åpent for brukerdefinerte utvidelser, både når det gjelder kompatible lagringsløsninger og programvarebiblioteker.

3.4.1 Hadoop

Apache Hadoop er et av de tidligste rammeverkene for stordata. Kjernen i Apache Hadoop består av en lagringsløsning kalt *Hadoop Distributed File System* (HDFS), og en programmeringsmodell kalt MapReduce. Hadoop deler filer opp i blokker og sprer dem over maskinene i en klynge, med nok kopier til å håndtere korrupte partisjoner og nettverksfeil. Klientkode komprimeres og sendes til hver enkelt av disse nodene hvor den kjøres separat og parallelt.

MapReduce er en svært enkel programmeringsmodell basert på velstuderte idéer fra funksjonell programmering. I korte trekk går den ut på at en funksjon som skal appliseres til et distribuert datasett kan deles opp i en *map*-fase der forskjellige deler av datasettet filtreres og sorteres parallelt og uavhengig av hverandre, og en *reduce*-fase som kombinerer disse filtrerte og sorterte resultatene.

Som i stort sett alle stordatasystemer etter Hadoop, utnytter altså denne tilnærmingen lokaliseringen

av data ved at hver maskin i klyngen arbeider på de dataene den har tilgang til. Dette gjør at datasettet vil behandles raskere og mer effektivt enn ved å skalere opp maskinvare på én enkelt maskin.

Apache Hadoop består av følgende moduler:

- Hadoop Common: programvarebiblioteker for å manipulere andre moduler.
- Hadoop Distributed File System (HDFS): Et distribuert filsystem.
- Hadoop YARN: Et administrasjonsverktøy for maskinklynger.

Det er ikke lenger veldig vanlig å bruke Apache Hadoop som programmeringsrammeverk direkte. Hadoop var et av de tidligste åpen kildekode-rammeverkene på markedet, og er nå å regne som foreldet. Som programmeringsplattform lever imidlertid Hadoop i beste velgående, og mange nye stordatateknologier er bygget som abstraksjoner over Hadoop. Der er etterhvert mange av disse systemene, og de kalles med en samlebetegnelse for Hadoop-økosystemet: Her er noen eksempler:

- Apache Hive[96] er et datavarehusprosjekt bygget over Hadoop som er designet for å støtte spørringer for statistikkrelaterte analyseoppgaver over multidimenasjonale data, slik som aggregering, summering og såkalt *slicing-and-dicing*. Hive gir et SQL (eller SQL-liknende) grensesnitt for å søke i ulike databaser og filsystemer som er kompatible med Hadoop. Hive støtter et stort nok fragment av SQL til at mange fagsystemer som opprinnelig er utviklet for relasjonelle databaser, kan overføres til en stordataklynge. Apache Hive ble opprinnelig utviklet av Facebook, og brukes i dag av bl.a. Netflix.
- Apache Phoenix[97] er en parallell distribuert relasjonsdatabase som er designet for å gi et stordatasystem i OLTP-enden av CRUD-spekteret (jf. kapittel 2.8). Mer spesifikt er Phoenix en abstraksjon over Hadoop som tilbyr et standard SQL grensesnitt og standard JDBC databaseobjekter samt ACID transaksjoner over distribuerte datasett (jf. kapittel 2.5).
- Apache Kudu[98] er en kolonnedatabase (jf. kapittel 3.1.1.3) bygget over Hadoop. Som alle kolonnedatabaser organiserer Kudu data i kolonner snarere enn rader. Data som lagres i kolonner kan komprimeres og skannes effektivt, noe som sparer plass og gir hurtige leseoperasjoner. Kudu er med andre ord utviklet for rask analyse av sanntids- eller nær sanntidsdata.

Se figur 3.28 for en overordnet oppsummering av Apache Kafka.

3.4.2 Apache Spark

Apache Spark er i likhet med Hadoop et rammeverk som er basert på MapReduce modellen. Med andre ord tilbyr også Spark en programmeringsmodell hvor programlogikk uttrykkes ved å komponere parallelliserbare funksjoner slik som map, reduce, join og filter.

Utover dette er Hadoop og Apache Spark epler og pærer i den forstand at de egentlig ikke tjener de samme formålene. Hadoop er i hovedsak en distribuert datainfrastruktur som er designet for å spre

[96] https://hive.apache.org/
[97] https://phoenix.apache.org/
[98] https://kudu.apache.org/

Programmeringsmodell	Funksjonell/MapReduce
Skalerbarhet	Horisontalt skalerbar lagringsplass og regnekraft
Feiltoleranse	Feiltoleranse vha. datareplisering og persistering
CAP egenskaper	Vektlegger konsistens og feiltoleranse
Primærminne- vs. persistenssystem	Persistenssystem
CRUD-profil	På OLAP siden av spekteret
Ytelsesprofil	Raskt for enkle repeterende operasjoner. Lite egnet for iterative oppgaver
Gjenbrukbarhet	Lav. Avhengig av applikasjonslogikk
Analyse	Statistikkproduksjon og enkel businesslogikk

Figur 3.28 Hadoop, raskt oppsummert.

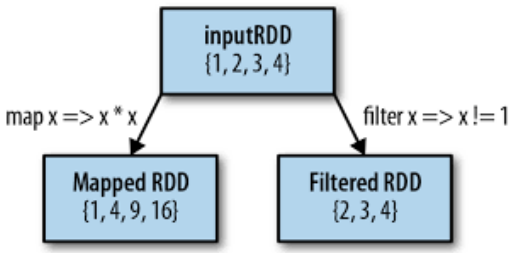
store datasett over en klynge av ordinære datamaskiner, noe som betyr at du ikke trenger å kjøpe og vedlikeholde dyr tilpasset maskinvare. Hadoop indekserer og repliserer disse dataene slik at de er tilgjengelig med minst mulig forsinkelse også når deler av et nettverk feiler.

Spark, på den annen side er primært et *prosesseringsverktøy*. Det er designet for effektivt å kunne utføre kompliserte regneoperasjoner på store datamengder, men delegerer selve lagringsfunksjonaliteten til andre verktøy, typisk HDFS (Hadoop Distributed File System).

Apache Spark er bygget over en sentral abstraksjon kalt RDDer (Resilient Distributed Datasets). RDDer er uforanderlige (*immutable*) datasett som eksisterer i minnet. At de er uforanderlige betyr at de kun kan leses. Et datasett i Spark endres nemlig strengt tatt aldri. Forandringer i datagrunnlaget representeres altså ikke ved å endre verdier i en eksisterende RDD, men snarere ved å lenke RDDer sammen i såkalte *avstamningstrær* (*lineage graphs*) som viser hvordan en ny RDD kan avledes fra en gammel ved hjelp av en navngitt operasjon.

Denne tilnærmingen gir to fordeler:

1. Det er mulig å holde store informasjonsmengder i minnet, noe som gir svært effektiv prosessering, også for iterative eller rekursive beregninger slik som de som er typiske i maskinlæring og sosial nettverksanalyse.
2. Avstamningsgrafer gir effektiv feilhåndtering siden systemets tilstand på et gitt punkt alltid kan gjenopprettes ved å repetere operasjoner langs grener i treet.



Figur 3.29 RDD avstamningstrær: datasettet nede til venstre (og høyre) eksisterer kun virtuelt og kan avledes fra roten med map-funksjonen som er angitt.

Spark-programmer er kjent for å være opp til hundre ganger raskere enn tilsvarende Hadoop-programmer for iterative oppgaver og strømmer. Spark er implementert i Scala og utnytter Scalas

funksjonelle programmeringskonstrukter for å uttrykke distribuerte beregninger. I praksis betyr dette at å skrive Spark-kode føles noenlunde som å skrive ordinær Scala-kode. Dette står i motsetning til Hadoop der den distribuerte lagringsmodellen er mye mer present – til dels brysomt – i programmeringsgrensesnittet.

Spark distribueres med mange velutviklede tredjepartsbiblioteker for stordataoppgaver, bl.a.:

- Spark Streaming er som navnet tilsier, et bibliotek for å interagere med strømmende data. Spark Streaming bruker mikrobatch-tilnærmingen til strømmer (jf. kapittel 3.3.1), hvilket i dette tilfellet vil si at en strøm representeres som en vilkårlig lang sekvens av diskrete RDDer. Denne representasjonen gjør at strømmende data kan manipuleres på samme måte som statiske data, og også kombineres med statiske data. Dette er et eksempel på det som i kapittel 2.3 ble kalt *funksjonell skalerbarhet*.
- Spark SQL er en abstraksjon over RDDer som gir et SQL-grensesnitt til et distribuert datasett. Dersom Spark SQL kombineres med Apache Hive som lagringsløsning, kan Spark SQL lese og skrive til et hvilket som helst Hive-støttet format inkludert tekstfiler, RC-filer (*Record Columnar File*), ORC (*Optimized RCFile*), Parquet (et popolært kolonneorientert format) og Avro (et RPC og serialiseringformat).
  For å støtte formater utover Hive kommer Spark SQL også med et API for å skrive adaptere til vilkårlige datakilder. Dette APIet implementeres ved å spesifisere en funksjon fra skjemaet og/eller typesystemet til forskjellige databasesystemer til Spark SQLs RDD-baserte datamodell. Spark distribueres med en håndfull ferdig implementerte adaptere, men det store flertallet av disse utvikles og vedlikeholdes av uavhengige tilbydere.
- Spark MLLib er et skalerbart maskinlæringsbibliotek som tilbyr optimaliserte algoritmer for alle vanlige, og en del uvanlige, maskinlæringsalgoritmer f.eks. regresjonsanalyse, dyp læring, klassifisering, beslutningstrær, gruppering/klyngedannelse (*clustering*) og mye annet.
- Spark GraphX er et grensesnitt for å representere og manipulere store mengder data i form av grafer. Som Spark Streaming utnytter også GraphX RDDer. Mer spesifikt benytter GraphX RDDer til å representere rettede attributtgrafer, jf. 3.2.1 GraphX er derfor funksjonelt skalerbart i samme forstand som Spark Streaming: grafdata kan kombineres med ordinære statiske data og strømmende data uten endring i programmeringsmodellen.

Kombinert med en distribuert lagringsløsning som Hive eller HDFS gir Spark, som de fleste andre nyere rammeverk, et totalt utviklingsmiljø for stordataapplikasjoner.

Programmeringsmodell	Funksjonell/MapReduce
Skalerbarhet	Horisontalt skalerbar lagringsplass og regnekraft
Feiltoleranse	Feiltoleranse vha. avstamningstrær
CAP egenskaper	Vektlegger konsistens og feiltoleranse
Primærminne- vs. persistenssystem	Primærminnesystem
CRUD-profil	Avhenger av lagringsløsning
Ytelsesprofil	Overlegent Hadoop på iterative beregninger
Gjenbrukbarhet	Avhengig av lagringsløsning
Analyse	Maskinlæring, sosial nettverksanalyse, anbefalingssystemer m. m.

Figur 3.30 Spark, raskt oppsummert.

3.4.3 The SANSA Stack

Mange stordatarammeverk slik som Apache Spark and Apache Flink (jf. kapittel 3.4) tilbyr grafanalyse gjennom tredjepartsbiblioteker. De støtter allikevel ikke *semantiske* standarder slik som RDF og OWL.

Slike rammeverk vil støte på velkjente utfordringer når de appliseres til heterogene datasett; ikke-standardiserte inputformater fordrer manuell kurering og tilpassing av dataene. Dette fører typisk til at mye tid brukes på preprosessering snarere enn på faktisk analyse (Lehmann et al. 2017).

Semantic Web-stacken er W3C-standardisert og har potensiale til å lette denne preprosesseringsoverheaden betraktelig: selv om det krever at det investeres mer tid i datamodellering, vil et datasett mye lettere kunne gjenbrukes i nye sammenhenger og for nye analyseoppgaver, noe som dramatisk reduserer kostnader over tid. Det finnes dessuten mange adaptere fra eksisterende lagringsløsninger til RDF (f.eks. R2RML standarden), som til sammen støtter sofistikert dataintegrasjon, f.eks. gjennom 'link analysis' og 'data fusion'-tilnærminger.

The Semantic Analysis Stack (SANSA)[99] er et programmeringsrammeverk som er spesielt designet for distribuert beregninger over store RDF-datasett. SANSA er bygget over Spark (jf. kapittel 3.4.2) og Flink (jf. kapittel 3.3.4.3), og arver derfor mange av egenskapene til disse rammeverkene; feiltoleranse og horisontal skalerbarhet, bruk av primærminnet for lagring av mellomresultater, map-reduce programmeringsmodell etc., jmf. figur 3.32. I tillegg implementerer SANSA all de sentrale Semantic Web-standardene og tilbyr

- spesialiserte serialiseringsmekanismer og partisjoneringsmetoder for RDF,
- en skalerbar motor for distribuerte SPARQL-spørringer over store datasett,
- en resonneringsmotor som utleder ny informasjon fra lagrede data og
- maskinlæringsalgoritmer tilpasset RDF-data.

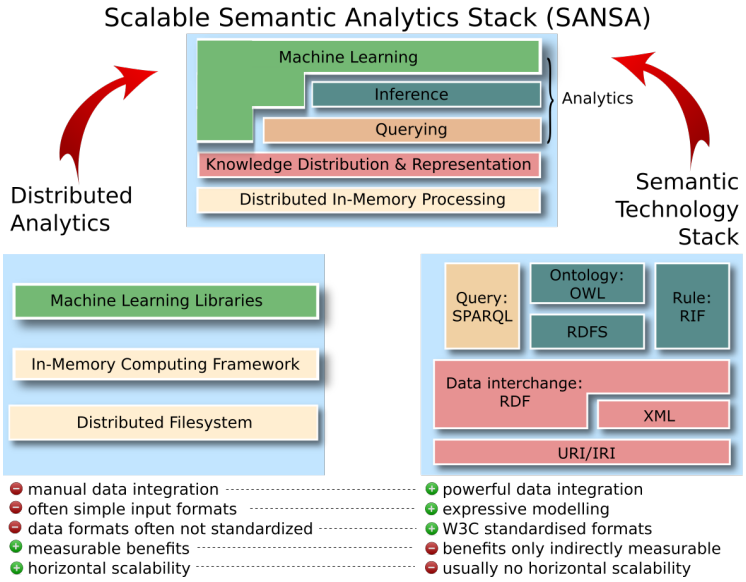
Pga. sine resonneringskapabiliteter har SANSA interessante anvendelser som ikke-semantiske stordatasystemer mangler; ontologibasert dataintegrasjon (jf. kapittel 3.19 og 4.4) og ontologibasert resonnering over strømmer er to eksempler.

SANSAs sett av maskinlæringsalgoritmer arver innholdet i de tilsvarende bibliotekene i Spark og Flink, men omfatter også en del algoritmer som utnytter grafstrukturen og begrepsmodellene som er spesifisert i RDF- og OWL-standardene. Eksempler omfatter link prediksjon og utledning av avhengighetsrelasjoner mellom objekter (*association rule mining*) og semantiske beslutningstrær.

3.5 Analyse- og visualiseringssystemer

Datavisualisering er et generelt begrep som beskriver enhver innsats for å hjelpe folk å forstå betydningen av data ved å plassere den i en visuell sammenheng. Mønstre, trender og korrelasjoner som kan gå uoppdaget i tekstbaserte data kan bli eksponert og gjenkjent lettere med datavisualiseringsprogramvare. Visualisering av data vil som regel kreve domenespesifikk framvisning, noe som ofte løses ved å ta i bruk hyllevare utvidet med skreddersydde tilpasninger.

[99] http://sansa-stack.net/



Figur 3.31 Sansaarkitekturen.

Programmeringsmodell	Funksjonell/Deklarativ/MapReduce
Skalerbarhet	Horisontalt skalerbar lagringsplass og regnekraft
Feiltoleranse	Feiltoleranse vha. avstamningstrær
CAP egenskaper	Vektlegger konsistens og feiltoleranse
Primærminne- vs. persistenssystem	Primærminnesystem
CRUD-profil	Avhenger av lagringsløsning
Ytelsesprofil	Som Apache SPARK, mer eller mindre
Gjenbrukbarhet	RDF-basert, tilrettelagt for gjenbrukbare data
Analyse	Kunstig resonnering, maskinlæring, sosial nettverksanalyse

Figur 3.32 SANSA, raskt oppsummert.

Analyse og visualisering av stordata er aspekter som er sterkt relatert, og systemer som utgir seg for å tilby visualisering av stordata vil som regel tilby analysefunksjonalitet. Grovt sett vil analysesystemer være komplette pakker, inkludert visualiseringsfunksjonalitet, mens såkalte stordata-visualiseringssystemer vil ha adaptere til en rekke etablerte stordatarammeverk for å realisere analysekapabiliteten.

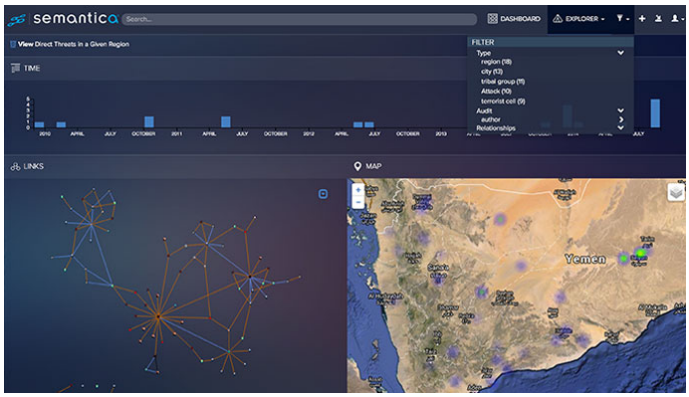
Analysesystemer som fokuserer på oppbygging og visualisering av nettverk og grafer, slik som Palantir Gotham[100], blir ofte framstilt som et prototypisk eksempel på et analysesystem for stordata. Slike systemer har som regel et omfattende integrasjonsrammeverk i bunn, som gjør det mulig å koble til funksjonalitet som f.eks. å ta inn både ustrukturert og strukturert informasjon samt å utføre data-mining over de samlede dataene. Prosesserte data kan så vises fram i det grafiske grensesnittet for videre visuell analyse. Foruten Palantir, er Semantica Pro Cortex, Neo4j, Visallo/Lumify og Siren.io eksempler på slike omfattende analysesystemer basert på underliggende integrasjonsrammeverk. Noen av disse systemene, slik som Palantir, baserer seg på tett kobling til selvutviklet analysefunksjonalitet, imens andre, slik som Semantica Pro, baserer seg på en løsere plug-in arkitektur der komponenter kan plukkes fra diverse leverandører.

[100] https://www.palantir.com/palantir-gotham/

Grensen fra analysesystemer med løs plug-in arkitektur, til såkalte visualiseringssystemer for stordata, er nokså uklar. Visualiseringssystemer som eksempelvis Tom Sawyer, KeyLines og Linkurious for linkanalyse, samt Tableau for business intelligence, er laget for å kunne lett kobles til annen dedikert hyllevare for analysefunksjonalitet.

3.5.1 Semantica Pro

Semantica Pro er en videreutvikling av programvaren SemNet, et verktøy for nettverksanalyse utviklet primært med henblikk på biologi og studiet av organiske strukturer.



Figur 3.33 Semantica Pro dashboard.

SemNet fikk oppmerksomhet fra etterretningssamfunnet etter angrepene på The World Trade Center i 2001, da amerikansk etterretning var på jakt etter et verktøy for å forvalte, sammenstille og analysere informasjonsbrokker som både kunne og burde sees i sammenheng med hverandre. Som en konsekvens ble selskapet Semantic Research etablert i 2001, og SemNet ble utviklet til et generelt etterretnings og overvåkingsverktøy kalt Semantica Pro.

Semantica Pro slik systemet er i dag er dog mer enn et verktøy for nettverksanalyse. Systemet er bygget over en lest som eksponerer et åpent programmeringsgrensesnitt for å muliggjøre tilpassing og utvidelser gjennom plug-ins. Dette er ikke helt ulikt arkitekturfilosofien bak editoren Eclipse, for å ta ett eksempel.

Standard-distribusjonen av Semantica Pro kommer i dag med et ganske stort utvalg av kapabiliteter, f.eks.:

- ansiktsgjenkjenning i live videostrømmer
- automatisk oversettelse av strømmer (video, chat, etc.)
- sentimentanalyse i sosiale medier
- sosial nettverksanalyse (sentralitet, innflytelse, etc.)

Semantica Pro er derfor i dag en desktopprogramvare som kan brukes til å håndtere svært varierte analytiske utfordringer på tvers av et bredt spekter av næringer og oppdrag, f.eks. innenfor finansetterretning slik som hvitvasking, etterforskning av tollsvindel, kartlegging av terrornettverk, o.a.

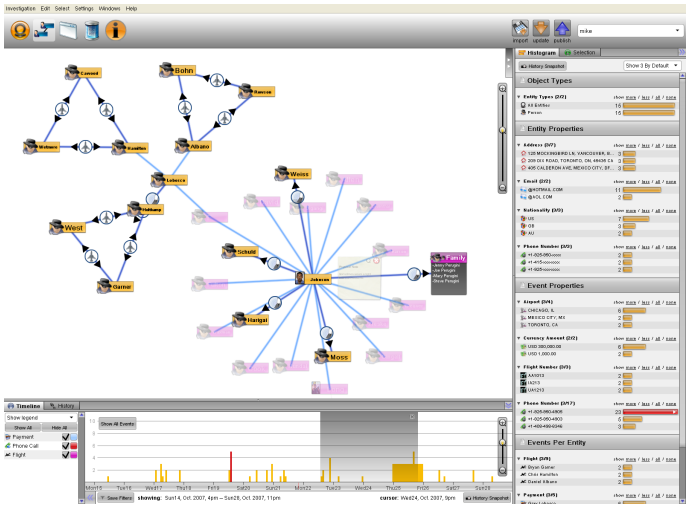
Siden 2016 har Semantic Research utvidet sitt kommersielle fotavtrykk, og i dag bruker flere *Fortune 500*-selskaper dette systemet for å bl.a. til å gjennomføre bedriftssikkerhetsundersøkelser, og identifisere og analysere sikkerhetstrusler. Semantica Pro har også fått en viss utbredelse innenfor Forsvaret.

3.5.2 Palantir

Palantir Gotham[101] er linkanalyseverktøy, beregnet spesielt for bruk i etterforskningsarbeid ifm. terrorismevirksomhet, finansiell svindel samt avanserte angrep på datasystemer og kritisk infrastruktur ("digital forensics"). Systemet har hatt flere store kontrakter innen forsvar-, toll- og justissektoren i USA, og her til lands har både politi- og tolletaten kontrakter med Palantir (se Politiforum (2018) og Aftenposten (2018)).

Bedriftsmodellen til Palantir[102] baserer seg på bruk av utplaserte ingeniører, såkalte *forward deployed engineers* i deres terminologi, som syr sammen systemet som kunden har bestilt. Dette inkluderer modellering av problemdomenet, integrasjon av eksterne kilder, o.l.

Produktet som kunden mottar ved kjøp av Palantir Gotham kan beskrives som å være skreddersydd og komplett, men lukket system, der funksjonaliteten som tilbys er begrenset til hva Palantir's bibliotek av (ofte egenutviklede) komponenter tilbyr. Se figur 3.34 for eksempel på brukergrensesnitt. Muligheten til å legge til funksjonalitet fra tredjepartsutviklede komponenter ser ut til å være meget begrenset. Dette er sådan på motsatt side av skalaen sammenliknet med f.eks. Semantica Pro, der systemet som selges i stor grad kan beskrives som et integrasjonsrammeverk for tredjeparts komponenter.



Figur 3.34 Palantir dashboard.[103]

[101] http://www.palantir.com/palantir-gotham/
[102] http://palantir.com
[103] https://digit.hbs.org/wp-content/uploads/sites/2/2018/04/palantirtech-link.png

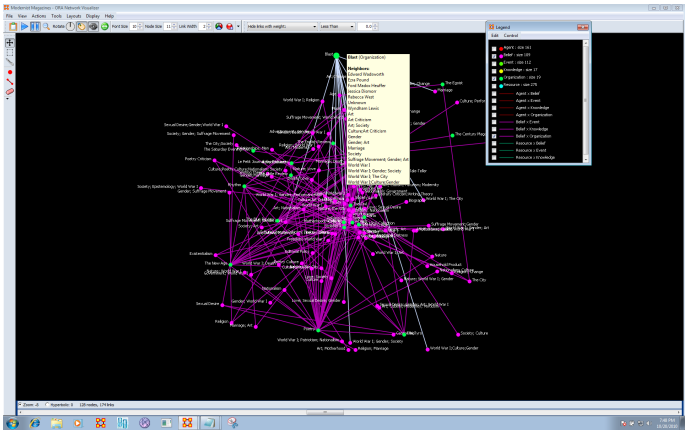
3.5.3 ORA

ORA[104] er et verktøy for sosial nettverksanalyse, utviklet ved Center for Computational Analysis of Social and Organizational Systems, Carnegie Mellon University og kommersialisert av Netanomics. Verktøyet inkluderer et brukergrensesnitt, se figur 3.35, og et stort bibliotek av algoritmer for grafanalyse på sosiale nettverk.

Med dette verktøyet kan en analytiker f.eks. se på et sosialt nettverk og få en idé om hvilke aktører som er sentrale i nettverket, hvilke roller de forskjellige aktørene har i nettverket og hvorvidt nettverket er sentralisert eller desentralisert m.m.

ORA har også algoritmer som egner seg for å analysere nettverk som er kjennetegnet av noder som er geografisk distribuert (geo-spatiale nettverk) og nettverk som endrer seg over tid (dynamiske nettverk), og er i stand til å håndtere nettverk opp til i størrelsesorden én million noder.

ORA benytter seg av det åpne formatet DyNetML, en variant av GraphML, for interoperabilitet med andre verktøy.



Figur 3.35 ORA: Sosial nettverksanalyse.

[104] http://netanomics.com/

4 Noen militære eksempler

I dette kapitlet søker vi å bruke egenskapene fra kapittel 2 til å diskutere tenkte stordataløsninger for et sett militære eksempler. Eksemplene er ikke uttømmende, men er valgt for å belyse avveiingene som må gjøres når man velger slike løsninger.

4.1 Eksempel 1: Digital arkivering

Forsvaret er, som alle offentlige enheter, underlagt en rekke krav til arkivering av informasjon, og tradisjonelle lagringsløsninger kan ikke alltid levere den smidigheten som skal til for å sikre optimale saksbehandlingsrutiner for journalføring og arkivering. Dette eksempelet illustrerer noen av vurderingene man må gjøre for å finne en best mulig løsning for tilfeller der man skal håndtere store mengder (*volume*) av informasjon.

Denne problemstillingen vil som regel ikke diktere sterke sanntidskrav, men må kunne holde store, og stadig økende, mengder informasjon over lang tid. Dette taler for at man her bør se til persistenssystemer framfor primærminnesystemer (se kapittel 2.6).

Informasjonen som er underlagt arkiveringskrav kan ikke enkelt slettes, så slike datalagre vil i utgangspunktet bare vokse. I en slik situasjon må man regne med at vertikal skalerbarhet ikke vil holde i lengden, men at man bør ha et system som er horisontalt skalerbart (se kapittel 2.3). Videre er det mange forskjellige organisatoriske enheter i Forsvaret som skal arkivere sin informasjon. For å legge til rette for dette bør man også ha organisatorisk skalerbarhet i tankene. Dette er ikke minst viktig i en oppstartsfase, da organisatorisk skalerbarhet legger til rette for at nye organisatoriske enheter enkelt kan koble seg til og gjøre sin informasjon tilgjengelig når nødvendige juridiske avklaringer er gjort.

Et system som er horisontalt skalerbart vil også være distribuert, og da må man gjøre en avveiing mellom konsistens og tilgjengelighet i henhold til CAP-teoremet (se kapittel 2.4). I dette tilfellet har man informasjon som er så viktig at den er pålagt arkivering, mens man ikke har noen åpenbare behov for alltid å ha mulighet til å hente informasjonen på minuttet. Det er derfor rimelig at man prioriterer konsistens foran tilgjengelighet og ergo sikter mot et CP-system.

En mulig løsning på denne typen problemstilling kan være såkalte *datavann* (*data lakes*) basert på forskjellige typer tabulære databaser slik disse er beskrevet i kapittel 3.1. Et datavann er en arkitekturtilnærming som utnytter den enorme lagringskapasiteten i et stordatasystem til å samle all den digitale informasjon som produseres i en gitt organisasjon på ett sted (eller mer presist; i samme sky eller maskinklynge). I prinsippet vil et datavann kunne fungere som en felles hukommelse eller kretsløp som gjør det mulig for ulike grupper innenfor organisasjonen å dele, gjenbruke og analysere de samme dataene.

Som avslutning av dette eksempelet er det verdt å merke seg at digital arkivering slik det er skissert her fordrer et forvaltningsregime som investerer nok ressurser i *metadata*, dvs. i annoteringer som dokumenterer opphavet til et dataelement, hvilke juridiske restriksjoner som hefter ved et datasett, eierskap, graderingsnivåer og ideelt sett all annen type informasjon som dataelementets livssyklus fordrer. Stordata i denne forstanden dreier seg med andre ord mye om digital arkiveringsteknikk.

4.2 Eksempel 2: Perimetersikring

Overvåking – systematisk og kontinuerlig observasjon – er en viktig oppgave for militære styrker for å bygge og vedlikeholde høy situasjonsbevissthet. En stadig høyere grad av digitalisering fører til at store mengder data lett blir tilgjengelig for dette formålet. Et eksempel på dette er video-overvåkning for perimetersikring av viktige objekter.

En slik installasjon vil kunne utføres av et antall videokameraer satt opp i perimeteren. Antallet vil være avhengig av størrelsen av objektet. Hvert enkelt av disse videokameraene kan generere 30 – 40 MB/s, noe som gjør at et slikt perimetersikringssystem vil generere mer data enn et menneske er i stand til å prosessere i sanntid hvis det er snakk om et objekt av noe størrelse som f.eks. en kommandoplass. I slike tilfeller vil teknologier som kan gjøre automatisk analyse på strømmende data for å varsle om mulige truende hendelser være til stor nytte.

Dette eksempelet illustrerer mao. noen av vurderingene man må gjøre for å finne en best mulig løsning for tilfeller der man skal håndtere informasjon som genereres raskt og behandles i sanntid (*velocity*).

For en slik oppgave vil det viktigste være å få analyseresultater i form av interessante hendelser raskt, og et system som baserer seg på primærminne vil være mer egnet enn et persistenssystem. Slike systemer har imidlertid svakheten at data som ikke er persistert vil forsvinne når systemet slås av eller strømmen blir brutt. Dersom det er et krav om at informasjonen som behandles også skal lagres, kan denne sårbarheten bøtes på gjennom å kombinere bruk av primærminne- og persistenssystemer.

Lokalt på en slik installasjon kan man forvente å ha rimelig kontroll på datakildene og hvilke mengder data det er snakk om. Vertikal skalering kan derfor være tilstrekkelig, men med store datamengder vil det kreve stordatamaskinvare som typisk er dyrere enn standard maskinvare. Hvis man kan ta seg råd til det vil man kunne slippe å bekymre seg for partisjoneringsproblematikk, og med det unngå begrensningene som pålegges av CAP-teoremet (se kapittel 2.4) og kunne oppnå både konsistens og tilgjengelighet.

I et slikt tilfelle vil man imidlertid kunne få utfordringer rundt feiltoleranse, siden det da er snakk om kun én sentral node som hvis den feiler vil gjøre at hele systemet feiler. Dette kan bøtes på gjennom å gjøre systemet distribuert, men et slikt designvalg vil gjeninnføre utfordringen rundt robusthet pga. partisjonering og fordre at man må gjøre en avveiing mellom konsistens og tilgjengelighet. I dette tilfellet vil det sannsynligvis være fornuftig å prioritere tilgjengelighet (AP-system) framfor konsistens (CP-system). Et slikt system vil ha svakheten at det kan gi advarsler som er basert på feil informasjon siden man ikke er garantert konsistens. Advarslene vil imidlertid kunne komme raskt pga. tilgjengelighet, og det virker rimelig at dette er å foretrekke framfor et system som heller venter med å gi potensielt viktige advarsler.

Et annet skalerbarhetshensyn som bør tas med i et slikt tilfelle er behovet for funksjonell skalerbarhet. Dette kan være nyttig dersom man ved en senere anledning ønsker å legge til for eksempel nye analysemuligheter.

En systemløsning for dette eksempelet vil naturlig ha et strømmesystem (se kapittel 3.3) som kjerne. Da vil man kunne legge til rette for at store mengder strømmende data kan behandles i sanntid eller nær sanntid.

4.3 Eksempel 3: Maritim overvåkning

Maritim overvåking er den systematiske og kontinuerlige observasjonen av aktivitetene i Norges havområder for å sikre Norges suverenitet og rettigheter. Det innebærer å samle data fra mange forskjellige kilder og bearbeide disse slik at de er egnet som støtte til beslutningstakere. Bearbeidingen kan også innbefatte analyse.

Maritim overvåking fordrer at informasjon fra mange forskjellige kilder blir utnyttet. Noen av disse kildene er:

- Rapporter fra Automatic Identification System (AIS)
- Informasjon fra maritime plattformer (fregatt, korvett, kystvakt, u-båt)
- Informasjon fra luftplattformer (P3C/N Orion, DA-20, F-16, F-35)
- Satellittbaserte radarsystemer
- Satellittbilder
- NATO Air Ground Surveillance (AGS)

Siden informasjonen kommer fra veldig forskjellige kilder, kommer den også på mange forskjellige former og formater. Et stordatasystem som skal støtte maritim overvåkning må derfor fremfor alt håndtere en stor variasjon (*variety*) i dataene.

Mange forskjellige organisasjoner bidrar med informasjonen som må utnyttes i maritim overvåkning. Forsvaret bidrar med informasjon fra sine plattformer, kystverket bidrar med AIS-informasjon og egne overvåkingsdata og NATO bidrar med informasjon samlet inn av AGS, for å nevne de mest fremtredende. Dette mangfoldet betyr at et samlende system for maritim overvåking bør støtte organisatorisk skalerbarhet (se kapittel 2.3).

En viktig bruk av maritime overvåkingsdata er deteksjon av fartøyer hvis bevegelsesmønster avviker fra det de selv rapporterer. Dette er i seg selv ikke noe ulovlig, men det kan tyde på smugling, tyvfisking eller annen lyssky virksomhet. Automatisering av slik deteksjon vil kreve en eller annen form for automatisk analyse av dataene, og et stordatasystem må legge til rette for dette. Maskinlæring er for eksempel en gruppe algoritmer som er mye brukt som grunnlag for systemer som gjør mønstergjenkjenning for å detektere slike avvik, og ønsker man et system som utnytter denne muligheten må det legges til rette for å støtte iterative beregninger (se kapittel 2.7).

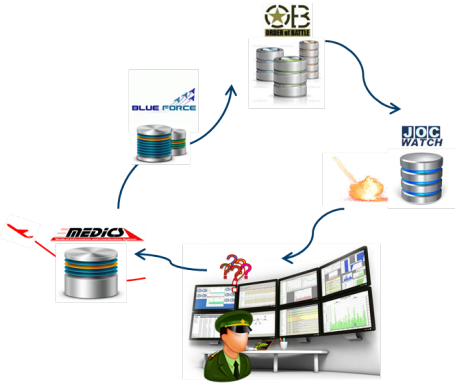
En stordataløsning som skal legge til rette for å håndtere variasjon slik som i dette eksempelet må sannsynligvis ha elementer fra alle de tre hovedtypene identifisert i denne rapporten – tabulære databaser, grafdatabaser og strømmesystemer. Tabulære databaser vil for eksempel kunne håndtere store samlinger satellittbilder, og grafdatabaser kan det legges til rette for grafbaserte analyser som nettverks- og linkanalyse. Strømmesystemer trengs for å bearbeide de strømmende dataene, som f.eks. AIS-meldinger, i sanntid.

4.4 Eksempel 4: Planlegging av evakueringsflygninger

Forestill deg en militær analytiker i et operativt hovedkvarter som har som oppgave å planlegge og overvåke evakueringsflygninger inn i et stridsområde. Det er spesielt viktig å holde et våkent

øye med evakueringsflygninger som er truet av fiendtlig aktivitet. Dersom en flygning er truet, er det analytikerens oppgave å lete etter vennlige styrker som er i stand til å nøytralisere trusselen i nærheten av landingsområdet. For å besvare dette informasjonsbehovet vil analytikeren vanligvis måtte kombinere informasjon fra flere ulike systemer, som vi normalt vil finne i et operativt hovedkvarter.

Analytikeren vil kanskje måtte konsultere en hendelseslogg for å følge utviklingen i stridsområdet, og et planleggingssystem for evakueringsflygninger for å avgjøre hvilke landingssoner som er truet. I tillegg vil hun naturlig nok ha behov for et «blåprikksystem» som viser hvor de vennlige styrkene står, og muligens også en Order of Battle-database for opplysninger om kapabilitetene til de ulike vennlige og fiendtlige stridsenhetene (se figur 4.1).



Figur 4.1 Dataintegrasjon for evakueringsflygninger.

Det er en tidkrevende og skjør prosess å kombinere alle disse opplysningene manuelt, og det fordrer at analytikeren kjenner de ulike informasjonssystemene og deres vanligvis ulike datamodeller og spørregrensesnitt. Gjenbruk av data utenfor intendert bruk (ref. kapittel 2.9), samt håndtering av *variasjon* er sådan sentrale utfordringer i dette tilfellet, da forskjellige informasjonssystemer ofte er utviklet og vedlikeholdt for separate formål uten felles grensesnitt eller datamodeller. Ontologibasert dataintegrasjon (OBDA, ref. kapittel 3.2.3.4) bruker virtuelle grensesnitt for å løse utfordringene dette gir.

Ontologibasert dataintegrasjon tilbyr en dynamisk og fleksibel løsning på analytikerens problem med å sammenstille informasjon. I et OBDA-system er integrasjonen kun virtuell og derfor løst koblet til de underliggende datakildene. Dette betyr blant annet at de underliggende datakildene ikke behøver å være utviklet eller vedlikeholdt for å utveksle informasjon med hverandre.

Som nevnt i kapittel 3.2.3.4 består kjernen i et OBDA-system av en begrepsmodell (også kalt en ontologi) som kan betraktes som en beregnbar spesifikasjon av hvordan typer av data i de underliggende kildene forholder seg til hverandre konseptuelt. Denne spesifikasjonen fungerer som et abstraksjonslag som legges over de underliggende kildene for å presentere dem for analytikeren i vårt tenkte tilfelle, som om de skulle være én kilde. Begrepsmodellen designes gjerne slik at den reflekterer brukerens foretrukne vokabular, noe som tillater vår analytiker å uttrykke sitt informasjonsbehov med begreper som reflekterer hennes kompetanse.

Selve datainnsamlingen foregår ved at analytikeren formulerer sitt informasjonsbehov ved hjelp av et spørrespråk som reflekterer begrepsmodellen. Siden begrepsmodellen uttrykker forhold mellom

typer av data i de underliggende kildene, gjør dette i sin tur det mulig å beregne hvilken informasjon som må hentes fra hvilke kilder, og hvordan den må kombineres for å svare på analytikerens informasjonsbehov. Denne oversettelsesprosessen gjør at et OBDA-system relativt enkelt kan tilpasses et vilkårlig antall kilder uten at kompleksiteten øker i brukerens øyne.

OBDA muliggjør gjenbruk av data og dataskjema på en måte som gir skalerbarhet både med tanke på antall datakilder, både i form av horisontal skalering så vel som inklusjon av nye typer kilder, samt varierende bruksområder ved hjelp av utbyttbare begrepsmodeller (ref. kapittel 2.3). En annen styrke med OBDA-tilnærmingen, er at den er feiltolerant iht. bortfallende datakilder, nettopp på grunn av abstraksjonsmodeller samt at informasjon ikke blir replisert lokalt, men innhentet run-time ved eksekvering av brukerspørringen.

Informasjonsbehovet skissert i evakueringsplanleggingen krever i tillegg et analysesteg ifm. å matche kapabilitetene til vennlige styrker opp mot den identifiserte trusselen. Denne type analyse-funksjonalitet kan støttes ved bruk av deduktiv resonnering (ref. kapittel 2.1), som OBDA-systemer vanligvis tilbyr.

5 Konklusjon

Stordata er et svært mangefasettert og forskjelligartet område som det egentlig ikke er plausibelt å regne som ett enkelt fagfelt. Som nevnt innledningsvis er begrepet kanskje mer dekkende som en betegnelse på en samfunnsutvikling enn for en bestemt teknologi. I dagens informasjonssamfunn er produksjon, spredning og utnyttelse av informasjon en vesentlig økonomisk, politisk og kulturell aktivitet, og mengden av data har for lengst overskredet det en enkelt maskin kan lagre og beregne. Av den grunn karakteriseres stordataproblemer ved hjelp av de såkalte tre V'ene *Volume*, *Velocity* og *Variety*. Stordata er data av forskjelligartet natur (*Variety*), som kommer i store mengder (*Volume*) og/eller har hyppig oppdateringsfrekvens (*Velocity*).

Som en respons til denne eksplosive veksten av informasjon, har det vokst frem en ny type programvaresystemer basert på prinsippet om å dele datalagring og beregningsoppgaver mellom et variabelt og skalerbart antall maskiner i en såkalt maskinklynge. Slike systemer har kapasitet til å håndtere nærmest vilkårlig store datamengder, men krever svært sofistikert koordinering mellom maskiner, og svært avanserte algoritmer for replisering og feilhåndtering.

Det finnes ulike måter å angripe dette problemet på som gir ulike egenskaper langs aksene responsivitet, konsistens og feiltoleranse. Noen systemer vil f.eks. garantere at alle maskinene i en klynge alltid arbeider på identiske kopier av dataene, men vil ikke samtidig garantere at systemet er responsivt til enhver tid. Hvordan et system vekter slike hensyn reflekterer systemets intenderte bruk og ytelsesprofil. De vil gjerne være optimalisert for én klasse av problemer, mens de typisk vil yte dårlig på andre områder. Et system som er optimalisert for nettverksanalyse, for eksempel, vil sjelden egne seg godt for virksomhetsetterretning og statistikkproduksjon.

Konsekvensen av dette er at man ikke kan forvente at én stordataløsning vil kunne dekke alle behov, slik som relasjonsdatabasene i sin tid langt på vei gjorde. Anskaffelse av en bestemt stordatateknologi bør derfor være forberedt av en grundig analyse av problemet, utarbeidet i samarbeid mellom teknologer og domeneeksperter. Første bud her er selvsagt å kjenne sitt domene, sitt problem og sine data. Dersom dataene ankommer raskt og problemet fordrer svar i sanntid, vil det kreve et helt annet system enn periodevis produksjon av analyseresultater over statiske, dvs. lagrede, data.

I denne rapporten har vi kompilert en liste over egenskaper vi mener det er viktig å ta med i betraktning når dette arbeidet skal gjøres. Disse egenskapene vil ha stor betydning for oppførselen og anvendeligheten til et system. Egenskapene vi har valgt å fremheve omfatter:

- naturlige beregnings- og analyseoppgaver for typen systemer
- programmeringsmodell
- skalerbarhetsegenskaper
- tilgjengelighet vs. konsistens
- sårbarhet og feiltoleranse
- tilnærming til datalagring: primærminnesystemer vs. persistenssystemer
- støtte for komplekse iterative eller rekursive beregninger
- støtte for input/output
- gjenbrukspotensialet til dataene

Rapporten deler videre stordatasystemer inn i fem hovedtyper. Denne inndelingen er, som alle slike typologier vil være, ikke den eneste mulige, og bør betraktes som en grov tommelfingerregel:

- tabulære databaser
- grafdatabaser
- strømmesystemer
- programmeringsrammeverk
- analyse- og visualiseringssystemer

Tilgangen på moden programvare er generelt svært god. Det finnes i dag en rik flora komponenter som kan settes sammen etter behov og utgjøre et system for stordata. De fleste av disse komponentene er også åpent tilgjengelig og gratis. Dette gir gode muligheter til å sette sammen systemer etter behov, og etter prøv-og-feil prinsippet. Det finnes også rimelig god tilgang på kompetanse i konsulentmarkedet. Alt i alt er det derfor ikke nødvendig i dag å kjøpe dyre skorsteinssystemer der man låser seg til én enkelt leverandør.

Vi har skissert fire militære anvendelser av stordatateknologier (jf. kapittel 4). Disse er bevisst valgt for å illustrere noen av de egenskapene eller aksene som er listet i kapittel 2 og som denne rapporten er strukturert rundt. Det er utvilsomt mange andre.

Denne rapporten er et første skritt i et kompetanseoppbyggingsarbeid på stordatasystemer- og utfordringer, og legger derfor vekt på teknologifronten innen området. Videre planer er å gjennomføre eksperimenter med de identifiserte teknologiene for å kunne avgjøre hvordan disse kan utnyttes for militære formål.

Referanser

Abadi, D., Boncz, P. & Harizopoulos, S. (2013), *The Design and Implementation of Modern Column-Oriented Database Systems*, Now Publishers Inc., Hanover, MA, USA.

Aftenposten (2018), 'Nå skal algoritmer og analyser av «big data» avgjøre hvem som blir sjekket ekstra nøye i tollen', *Aftenposten* .
URL: *https://www.aftenposten.no/norge/i/6nkkX0/Na-skal-algoritmer-og-analyser-av-big-data-avgjore-hvem-som-blir-sjekket-ekstra-noye-i-tollen*

Business Insider (2015), 'The 'connected car' is creating a massive new business opportunity for auto, tech, and telecom companies'.
URL: *http://www.businessinsider.com/connected-car-statistics-manufacturers-2015-2?r=US&IR=T&IR=T*

Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M. & Rosati, R. (2009), Ontology-based data access and integration, *in* L. Liu & M. T. Özsu, eds, 'Encyclopedia of Database Systems', Springer.

Carbone, P., Gevay, G. E., Hermann, G., Katsifodimos, A., Soto, J. & Markl, V. (2017), Large-scale data stream processing systems, *in* A. Y. Zomaya & S. Sakr, eds, 'Handbook of Big Data Technologies', Springer.

Cugola, G. & Margara, A. (2012), 'Processing flows of information: From data stream to complex event processing', *ACM Computing Surveys (CSUR)* **44**(3), 15.

Garey, M., Johnson, D. & Stockmeyer, L. (1976), 'Some simplified NP-complete graph problems', *Theoretical Computer Science* **1**(3), 237 – 267.

Gartner (2017), 'Gartner says 8.4 billion connected things will be in use in 2017, up 31 percent from 2016'.
URL: *https://www.gartner.com/newsroom/id/3598917*

Gartner IT Glossary (2018), 'Big data'.
URL: *https://www.gartner.com/it-glossary/big-data*

Gilbert, S. & Lynch, N. (2002), 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', *Acm Sigact News* **33**(2), 51–59.

Hecht, R. & Jablonski, S. (2011), Nosql evaluation: A use case oriented survey, *in* 'Proceedings of the 2011 International Conference on Cloud and Service Computing', CSC '11, Washington, DC, USA, pp. 336–341.

Junghanns, M., Petermann, A., Neumann, M. & Rahm, E. (2017*a*), *Management and Analysis of Big Graph Data: Current Systems and Open Challenges*, Springer International Publishing, Cham, pp. 457–505.

Junghanns, M., Petermann, A., Neumann, M. & Rahm, E. (2017*b*), Management and analysis of big graph data: Current systems and open challenges, *in* 'Handbook of Big Data Technologies'.

Krebs, J. (2014), 'Questioning the lambda architecture'.
URL: *https://www.oreilly.com/ideas/questioning-the-lambda-architecture*

Krettek, A. & Winters, M. (2017), 'The curious case of the broken benchmark: Revisiting apache flink vs. databricks runtime'.
URL: *https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime*

Laney, D. (2001), 3D Data Management: Controlling Data Volume, Velocity, and Variety, Technical report, META Group.

Lehmann, J., Sejdiu, G., Bühmann, L., Westphal, P., Stadler, C., Ermilov, I., Bin, S., Chakraborty, N., Saleem, M., Ngonga, A.-C. N. & Jabeen, H. (2017), Distributed semantic analytics using the sansa stack, *in* 'Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017)', Springer, pp. 147–155.

Lewis, R. R. (2015), *A Guide to Graph Colouring: Algorithms and Applications*, 1st edn, Springer Publishing Company, Incorporated.

Maan, V. & Purohit, G. N. (2012), 'Article: A distributed approach for frequency allocation using graph coloring in mobile networks', *International Journal of Computer Applications* **58**(6), 9–13.

Marketing, I. (2017), 10 key marketing trends for 2017, Technical report, IBM.
URL: *https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN*

Marz, N. & Warren, J. (2015), *Big Data. Principles and best practices of scalable real-time data systems*, Manning.

Metzger, R. & Ward, C. (2018), 'How to size your apache flink cluster: A back-of-the-envelope calculation'.
URL: *https://data-artisans.com/blog/how-to-size-your-apache-flink-cluster-general-guidelines*

Micro Focus (2017), 'How much data is created on the internet each day?'.
URL: *https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/*

Moreno, J. L. (1934), *Who shall survive? : a new approach to the problem of Human Interrelations*, Vol. 58 of *Nervous and mental disease monograph series*, Nervous and Mental Disease Publ., Washington.

Papadimitriou, C. H. (1994), *Computational complexity.*, Addison-Wesley.

Pavlo, A. & Aslett, M. (2016), 'What's really new with newsql?', *SIGMOD Rec.* **45**(2), 45–55.

Politiforum (2018), 'Amerikansk «big data»-gigant har signert avtale med politiet verdt 81 millioner kroner', *Politiforum* .
URL: *https://www.politiforum.no/artikler/amerikansk-big-data-gigant-har-signert-avtale-med-politiet-verdt-81-millioner-kroner/404434*

Redmond, E. & Wilson, J. R. (2012), *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*, Pragmatic Bookshelf.

Riihijarvi, J., Petrova, M. & Mahonen, P. (2005), Frequency allocation for wlans using graph colouring techniques, *in* 'Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services', WONS '05, IEEE Computer Society, Washington, DC, USA, pp. 216–222.

Robinson, I., Webber, J. & Eifrem, E. (2013), *Graph Databases*, O'Reilly Media, Inc.

Sadalage, P. J. & Fowler, M. (2012), *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1st edn, Addison-Wesley Professional.

Seyvet, N. & Viela, I. M. (2016), 'Applying the kappa architecture in the telco industry'.
URL: *https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry*

Socialbakers (2018*a*), 'Facebook statistics directory'.
URL: *https://www.socialbakers.com/statistics/facebook/*

Socialbakers (2018*b*), 'Twitter statistics directory'.
URL: *https://www.socialbakers.com/statistics/twitter/*

Stonebraker, M. & Weisberg, A. (2013), 'The voltdb main memory dbms', *IEEE Data Eng. Bull.* **36**, 21–27.

The Washington Times (2015), 'Recovery board folding with mixed track record of stimulus plan'.
URL: *https://www.washingtontimes.com/news/2015/sep/29/recovery-board-folding-with-mixed-track-record-of-/*

Vogels, W. (2009), 'Eventually consistent', *Commun. ACM* **52**(1), 40–44.

Wu, D., Sakr, S. & Zhu, L. (2017), *Big Data Programming Models*, Springer International Publishing, Cham, pp. 31–63.

Yu, F. R. (2011), *Cognitive Radio Mobile Ad Hoc Networks*, Springer Publishing Company, Incorporated.

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's MISSION

FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's VISION

FFI turns knowledge and ideas into an efficient defence.

FFI's CHARACTERISTICS

Creative, daring, broad-minded and responsible.

Om FFI

Forsvarets forskningsinstitutt ble etablert 11. april 1946. Instituttet er organisert som et forvaltningsorgan med særskilte fullmakter underlagt Forsvarsdepartementet.

FFIs FORMÅL

Forsvarets forskningsinstitutt er Forsvarets sentrale forskningsinstitusjon og har som formål å drive forskning og utvikling for Forsvarets behov. Videre er FFI rådgiver overfor Forsvarets strategiske ledelse. Spesielt skal instituttet følge opp trekk ved vitenskapelig og militærteknisk utvikling som kan påvirke forutsetningene for sikkerhetspolitikken eller forsvarsplanleggingen.

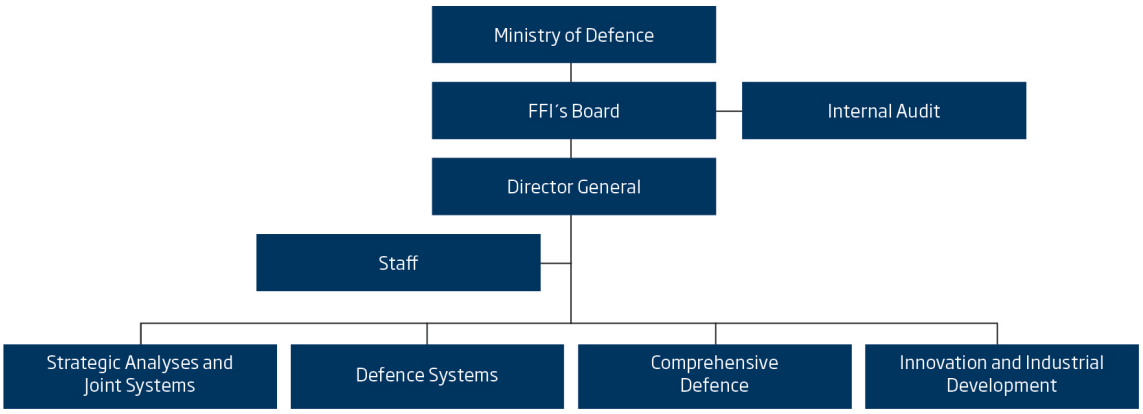
FFIs VISJON

FFI gjør kunnskap og ideer til et effektivt forsvar.

FFIs VERDIER

Skapende, drivende, vidsynt og ansvarlig.

FFI's organisation



Forsvarets forskningsinstitutt
Postboks 25
2027 Kjeller

Besøksadresse:
Instituttveien 20
2007 Kjeller

Telefon: 63 80 70 00
Telefaks: 63 80 71 15
Epost: ffi@ffi.no

Norwegian Defence Research Establishment (FFI)
P.O. Box 25
NO-2027 Kjeller

Office address:
Instituttveien 20
N-2007 Kjeller

Telephone: +47 63 80 70 00
Telefax: +47 63 80 71 15
Email: ffi@ffi.no

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment