# Sammendrag

Moderne kommersielle programvaresystemer leverer ofte tjenester til opptil flere hundre tusen brukere over Internett, det vil si ved hjelp av HTTP - applikasjonsprotokollen. Det er på slike systemer at NoSQL - databaser gjerne tas i bruk da de i vesentlig grad er i stand til å lagre stadig større datavolum som oppstår i et stadig raskere tempo mer effektivt. NoSQL - databaser er også mer horisontalt skalerbare enn de tradisjonelle relasjonsdatabasene, det vil si at de egner seg bedre til å dele ut og kopiere dataelementer over en klynge av databaseprosesser.

En av de største utfordringene innen drift av moderne kommersielle programvaresystemer som bankapplikasjoner, sosiale medier og netthandelssytemer er kunsten å minimalisere nedetid som følger av oppdatering av systemet. For mange store bedrifter som eier og admistrerer slike systemer er det totalt uaktuelt å dekommisjonere hele eller deler av systemet for å installere en liten programvareoppdatering eller resirkulere minne. Til det vil nedetiden til systemet medføre utålelige inntektstap. Derfor oppgraderer mange bedrifter systemene sine ”online”, det vil si at opppgraderingen gjøres uten å slå av en eneste datamaskin, og uten å forstyrre behandlingen av forespørsler fra brukere. Erfaringer fra industriene tilsier at slike levende oppgraderinger er lettere sagt enn gjort, især når det kommer til oppgraderinger av applikasjonens datamodell mens den opererer i et produksjonsmiljø.

Moderne programvaresystemer utvikles gjerne under en smidig utviklingssykel der nye versjoner, eller oppdateringer, publiseres til bruk opptil flere ganger om dagen. Slike oppdateringer kan endre programvaresystemets datamodell, eller ”skjema” som det heter i relasjonelle databaser. For å utføre slike opppgraderinger tryggest mulig blir systemet oppgradert på rullerende vis. Denne masteroppgaven setter som mål å realisere støtte for levende oppgradering av data-modeller i høytilgjengelige systemer uten nedetid ved å utvikle et eget administrasjonsverktøy til databasehandteringssystemet Voldemort. Dette verktøyet tillater applikasjonsutviklere å legge inn transformasjonsfunksjoner som kalles på ”lazy” vis når hver enkelt datatuppel aksesseres i databasen.

# Forord

Min masteroppgave presenterer et modulært programvarebibliotek som automatiserer oppdatering av semistrukturerte datamodeller i distribuerte, aggregatorienterte databasesystemer. Rapporten utgjør min besvarelse som vurderes i emnet TDT4900 - Datateknologi, masteroppgave, og utgjør samtidig mitt siste innleveringsarbeid i studieprogrammet MTDT - Datateknologi ved Norges Teknisk - Naturvitenskapelige Universitet i Trondheim. Oppgaven er basert på vitenskaplige kilder funnet og diskutert i løpet av fordypningsprosjektet jeg gjennomførte høsten 2017.

Formålet med oppgaven er å utforske hvordan prosessen med å oppgradere moderne webapplikasjoner som allerede kjører i et fungerende, aktivt produksjonsmiljø uten å slå av tjenesten. En egen løsning for denne problemstillingen er blitt implementert og testet i et realistisk oppgraderingsscenario for en typisk datamodell i en ekommersiell setting.

Den enkelte leser behøver ikke ha noen dype forkunnskaper om datamaskinvare eller operativsystemer. Det antas imidlertid at leseren er kjent med fenomenet ”prosess” i kontekst av operativsystemer, samt mønsteret for fjernt prosedyrekall, tradisjonelle databasesystemer, transaksjonsmønsteret og dets fire kvalitative egenskaper.

En stor, personlig takk rettes til min veileder Svein Erik, for gode, motiverende svar på mine spørsmål og usikkerheter rundt dette prosjektet, samt frie tøyler til å forme masteroppgaven etter eget ønske.

Rapporten er skrevet i LaTeX, og benytter en mal laget av Agus Ismail Hasan. [1] Takket være hans arbeid med denne malen sparte jeg mye tid på å sette opp dokumentets tekniske struktur, og det er derfor forfatteren krediteres i dette forordet.

Jeg vil også takke min tante, forhenværende lærer og utdannet logoped Nella Lovise Bugge, for hjelp med korrekturlesing av denne prosjektrapporten.

Trondheim, 5. mars 2018

Vegard Bjerkli Bugge

[1] Malen er tilgjengelig fra DAIM sin FAQ, `https://daim.idi.ntnu.no/faq_innlevering.php`

# Innhold

# Forkortelser

| Forkortelse | = | Definisjon |
|---|---|---|
| SQL | = | Structured Query Language |
| DDL | = | Data Definition Language |
| SLA | = | Service Layer Agreement |
| EC2 | = | Elastic Compute Cloud |

Kapittel 1

# Introduksjon

Dette kapitlet introduserer problemstillingen som oppgaven skal besvare og motivasjonen som ligger bak. Videre skisseres målene for løsningen av problemstillingen, og et eget delkapittel beskriver rammebetingelsene og gyldighetsområdet for denne løsningen. Siste delkapittel beskriver strukturen på oppgaven.

## 1.1 Bakgrunn

Moderne nettbutikker, offentlige nettbaserte tjenester, og nettbankapplikasjoner stilles svært strenge krav til tilgjengelighet. Aller helst skal en hvilken som helst kunde av en populær nettbutikk som Amazon kunne se på og legge ting i handlekurven, for deretter å betale for dem **når som helst, til alle døgnets tider**. At en vare blir lagt i handlekurven to ganger eller at en kunde leser utdatert informasjon om en vare like etter at den er blitt oppdatert har ikke så mye å si, for den slags småfeil lar seg alltid rettes opp i etterkant.

I tjenestenivåavtalen [1] (eng. "Service Level Agreement") til Amazon EC2 oppgis en tilgjengelighetsgaranti på 99,95 prosent (Bass et al., 2013). Til tross for denne høye prosenten tilgjengeligheten må programvarearkitekter som vil gjeste sine systemer på EC2 ta høyde for den halve prosentandelen der plattformen ikke er tilgjengelig for tjenesteleveranse.

[1] Tilgjengelig på url `https://aws.amazon.com/ec2/sla/`

Hvert sekund nedetid teller når det kommer til høyt-traffikerte tjenester på Internett som det sosiale mediet Facebook og tidligere nevnte Amazon sine skytjenester. Den 21. april 2011 hadde skyplattformen Amazon EC2 en periode med nedetid på fire dager (Bass et al., 2013). Dette tjenesteavbruddet rammet mange oppstartsselskaper som benyttet skyplattformen, inklusive Reddit, Quora og FourSquare. Schiller (2011) ved Information Today rapporterer at årsaken til hendelsen kom av en planlagt konfigurasjonsoppdatering som medførte at mange tjenestenoder mistet kontakten med backuptjenerne. Den samlede effekten av at alle nodene automatisk prøvde å gjenetablere forbindelsen førte til en overbelastning av forespørsler mot disse tjenerne.

En interressant bemerkelse fra denne episoden er at Netflix også var en hyppig bruker av plattformen på det tidspunktet webtjenesten gikk ned, uten at det gikk utover strømme - tjenestens egen tilgjengelighet. Forklaringen var at Netflix sine ingeniører tok høyde for den halve promillen som EC2 sin tjenestegaranti ikke dekket, blant annet ved å spre flere instanser av sine tilstandsløse tjenester utover flere av Amazon sine tilgjengelighetssoner (Bass et al., 2013).

Nedetid, den forventede tiden en plattform eller et programvaresystem ikke kan utføre dets definerte arbeidsoppgaver for dets brukere i løpet av en definert tidsperiode, er sterkt knyttet til systemets tjenestenivågaranti. Slike tilgjengelighetsgarantier baseres på beregninger med stokastiske modeller, for eksempel Markoff-analyse eller feil-tre (Bass et al., 2013). Ved hjelp av nevnte verktøy kan man anslå en forventningsverdi for hvor lang tid det vil gå mellom hvert feilscenario som rammer systemet slik at det blir totalt utilgjengelig for bruk.

Man kan også estimere en forventningsverdi for hvor lang det tar å reparere eller maskere nevnte feil slik at tjenester kan leveres av systemet som normalt. I lys av programvare som for eksempel databasesystemer er den førstnevnte verdien i praksis tiden fra en instans slås av til en ny startes opp, for eksempel ved en programvareoppdatering. Verdien til den andre variabelen påpeker tilsvarende hvor lang tid en programvarerestart tar.

Ut ifra en studie av flere dusin feilscenarier i storskala internettsystemer gjorde Oppenheimer et al. (2003) følgende konkluderende observasjoner: (1) operatørfeil er den hovedsaklige feilkilden i to av tre tilfeller; (2) operatørfeil har størst innvirkning på reparasjonstiden i to av tre internett-tjenester; (3) blant operatørfeil er konfigurasjonsfeil (feil syntaks, inkompatible argumenter) vanligst.

I en annen undersøkelse, der totalt 51 databaseadministratorer med varierende fartstid i yrket ble intervjuet, identifiserer Oliveira et al. (2006) i alt åtte kategorier feilscenarier som

oppstår i et databasesystem som kjører i et produksjonsmiljø: leveranse til produksjonsmiljø (deployment), ytelse (performance), strukturer i databasen (structure), tilgangsrettigheter (access-privilege), vedlikehold (maintenance), diskplass (space), feil i programvare (DBMS), og feil i maskinvare (hardware). I de fem førstnevnte er det databaseadministratoren som er den typiske hovedårsaken (i over 50 prosent av problemene som ble oppgitt under intervjuene) til at feil av disse typene oppstår.

Observerte trender innen flere forskjellige typer næringsvirksomhet, deriblant kundestøtte, industriell produksjon, e-kommers, finans, og banktjenester (Dumitraş et al., 2010; Choi, 2009) tilsier at det er et sterkt behov for distribuerte systemarkitekturer som støtter online-oppgraderinger. Oppgraderingsrutiner for kjørende databaseapplikasjoner som fordrer eller påtvinger nedetid er ikke lengre forsvarlige i lys av tjenestenivåavtalene som deres flerfoldige tusen klienter tilbys.

Den mest sentrale karakteristikken ved online-oppgradering, programvareoppgradere uten stopp i systemet, er at den gamle versjonen av applikasjonen må kjøre samtidig som den nye installeres, slik at tjenestene applikasjonen leverer ikke blir utilgjengelig for dets brukere. Choi (2009) kaller denne rutinen for ”hot rollover”. I tillegg må installasjonsoperasjonen ikke forstyrre applikasjonens leveranse av tjenester, e.g. behandling av innkommende HTTP-forespørsler.

Et annet sentralt problem innen online-oppgradering er kunsten å holde styr på pakkeavhengigheter. Dette må gjøres for å oppdage om den gamle og nye versjonen har delte avhengigheter, det vil si at begge avhenger av samme programvarepakke, men ikke nødvendigvis samme versjon av denne pakken. For at tjeneren skal unngå å miste data eller å gå ned må begge versjonene av en og samme pakke installeres på tjeneren. I praksis benytter oppdateringsprogrammet som handterer avhengigheter en form for manuelt skrevet konfigurasjonsfil der alle avhengigheter listes i form av par av unike pakkenavn og påkrevd versjon. For eksempel leser pakkehandtereren til NodeJS inn avhengigheter fra en JSON-fil med navn ”packages.json”, som vedlikeholdes av utviklerne selv.

Disse inputfilene er altså kilder til menneskelige feil, som for eksempel syntaksfeil, eller deprekeringsadvarsler. Det er bevist at problemet med å løse opp avhengigheter er NP-hardt ved å utføre en reduksjon (transformering av problemet og dets input) fra **3SAT** – problemet (Dumitraş and Narasimhan, 2009). Dermed er det grenser for hvor mange og store avhengigheter et programvaresystem kan ha før kjøretidskostnaden for avhengighetsbehandling (i for eksempel APT-registeret) vokser seg altfor stor.

Derfor har store aktører i industrien i de senere år innført prossessen *rullerende oppgra-*

*dering*, der programvaren på én etter én tjener i klyngen av tjenere blir oppdatert. Ved en automatisert rullerende oppgradering kan man i utgangspunktet kun gjennomføre patching av programvare, det vil si at brukergrensesnittet som applikasjonen tjener må i den nye versjonen være bakoverkompatibel med den gamle. Eventuelle konflikter må løses manuelt.

Opp igjennom det siste tiåret har det vært vanlig å oppgradere programvare som kjører i et system av flere instanser, eller prosesser, på rullerende vis. I denne manuelt kontrollerte oppgraderingsmetoden blir én etter én instans av den gamle versjonen av programmet avsluttet og erstattet med en instans av den nye versjonen. Et vesentlig problem med denne metoden er at applikasjonens datamodell er som regel realisert i et databasesystem som er instansiert i en separat prosess fra webapplikasjonsprosessen på en og samme fysiske tjenerdatamaskin. Dermed oppgraderes datamodellen til hver applikasjonsinstans på et annet tidspunkt enn koden til selve applikasjonen. Dette medfører til at det dsitribuerte produksjonsmiljøet befinner seg i en mikset tilstand - en uoppgradert applikasjonsposess kan potensielt interagere med en oppgradert datamodell og vice versa, noe som kan introdusere uante feilkilder til applikasjonen.

## 1.2 Oppgavens problemstilling og mål

Denne masteroppgaven opererer med en konkret definert problemstilling. Her presesnteres denne definisjonen, hvorpå et sett med konkrete, oppnåelige, og tidsbestemte målpunkter forbundet til definisjonen av problemstillingen også blir listet opp.

**Problemstilling:** Formålet med dette prosjektet er å implementere et høytilgjengelig programvaresystem bygget med en nøkkel-verdi-datamodell realisert med databasehandteringssystemet Project Voldemort, der oppgraderinger som involverer endringer

Oppgaven har hatt følgende overordnede mål:

1. Beskrive sammenhengen mellom kontinuerlig programvareleveranse og levende oppgradering av datamodeller

2. Modellere en modulær løsning der semistrukturerte datamodeller, også referert til som NoSQL-datamodeller, kan oppgraderes synkront med applikasjonstjenerne

## 1.3 Oppgavens struktur

Denne rapporten har følgende struktur. Kapittel 1 er introduksjonskapitlet, som illustrerer problemstilllingen som undersøkes og kort hvordan. Det beskrives hvordan litteratursøket ble gjennomført, samt hvordan analysen av de ulike oppgraderingsmetodene utføres.

Kapittel 2 omtaler relevant teori om NoSQL - datamodeller, en kort innføring i databasearkitektur, distibuerte systemer, tilgjengelighetskvaliteten til et distribuert programvaresystem, oppgradering av programvaren som utgjør noder i distribuerte systemer og nedetid i systemer som oppstår i forbindelse med programvareoppgradering av dem. Konseptet "levende oppgradering av programvaresystemer" defineres og forklares her. I teorikapitlet presenteres også NoSQL - DBMSet Project Voldemort, som vedlikeholdes av et dedikert utviklingslag hos LinkedIn.

Det tredje kapitlet beskriver et knippe oppgraderingsverktøy som på hvert sitt eget vis kommer i bukt med versjonmiks - problemet beskrevet i kapittel 1.1. I kapittel 3 sammenliknes løsningsforslagene for online oppgradering av distribuerte databasesystem, som ble lest om . Denne sammenliknignen er "fortrinnsvis" kvalitativ (det vil si at det har vist seg komplisert å produsere et optimalt datagrunnlag for en kvantitativ test) der vurderingskriteriene bunner i hvor tilgjengelig løsningen er for det allmenne marked, hvor forståelig publikasjonene som presenterer er for rapportens forfatter og popularitet i industrien - det er jo tross alt et problem av industriell undertone som besvares her.

Kapittel 4 kalles for *Levende oppgradering av datamodeller realisert med Project Voldemort*. Dette kapitlet vil presentere systemdesignet rundt en kommersiell webapplikasjon kalt "DBUpgradinator", som kan inndeles i en frontend-del med presentasjonslogikk formet av CSS - og JS - filer og en backend-del med kontroll-logikk skrevet i Java som ved hjelp av serialisering med binærdatakodingsbiblioteket Apache Avro snakker med datalageret, en instans av den distribuerte oppslagstabellen Voldemort. I kapittel 4 blir også en kontinuerlig oppdateringsleveranseløsning for hele applikasjonen, inklusive dets implisitte dataskjema, skissert.

Kapittel 5 beskriver hvordan måloppnåelse av tidligere beskrevede krav til datamodellevolusjonsverktøyet evalueres. Under testing vil webapplikasjonen DBUpgradinator gjennomgå en patch der den implisitte datamodellen sett fra applikasjonslagets perspektiv blir endret samtidig som en mengde genererte forespørsler tilsendes tjenerne slik at en vanlig arbeidslast med spørringer simuleres.

I kapittel 6 konkluderes evalueringen og i det beskrives forslag til videre kvantitativt felt-

arbeid som kan bygge på denne analysen.

Kapittel 2

# Teori

Dette kapitlet gir leseren en innføring i tre NoSQL - datamodeller, hvordan de skiller seg ut fra den relasjonelle datamodellen og hvordan deres forskjeller fra relasjonelle databaser har innvirkning på hvordan levende oppgradering av dem og migrasjon av eksisterende data i produksjonsmiljøet kan utføres i en smidig utviklingsprosess.

## 2.1 Aggregatorienterte datamodeller (også kjent som NoSQL)

Ifølge Sadalage and Fowler (2013) kan nøkkel-verdi-lagre (eng. key-value store), kolonne-familelagre (eng. column family store) og dokumentlagre (eng. documents store) ordnes under én og samme ”art” av NoSQL-databaser: Aggregatorienterte databasesystem (eng. aggregate oriented databases).

Den enkle måten å forklare hva det vil si å ha en aggregatorientert datamodell er ved å beskrive hva det ikke er. I den relasjonelle datamodellen deles forskjellige former for informasjon inn i relasjoner, og tilhørende data inndeles i atomiske, disjunkte enheter kalt *tupler*. En tupel er en flat, endimensjonal liste av verdier som hver korresponderer til nøyaktig ett attributt av relasjonen tupelen er lagret i.

Det foreligger derfor visse begrensinger på denne datastrukturen. Til eksempel kan ikke en enkelt tupel nøstes inn i en annen, og hvert attributt i tupelen har én atomisk korresponderende verdi, aldri en liste av verdier. Her kan man kontre med at nyere versjoner av MariaDB støtter JSON-objekter som datatype. JSON-objekter er potensielt komplekse

dataenheter som kan inneholde nøstede datastrukturer. Men et objekt ansees av datalageret som én kohesiv, helhetlig dataenhet i form av streng, så det argumentet faller. Ettersom tupler er den laveste dataenheten i den relasjonelle modellen kan man si at spørringer opererer med og returnerer (et helt antall) tupler (Sadalage and Fowler, 2013).

Men så har vi den aggregatorienterte modellen, en måte å tenke på data som tillater den enkelte datamodellør å definere dataenheter i stedet for å tvinge vedkommende til å konformere med en forhåndsbestemt minste enhet, slik tilfellet er i den relasjonelle modellen. Denne fleksibiliteten i struktureringen av data er et sentralt fellestrekk nøkkel-verdi-lagre som Dynamo og Redis deler med kolonnefamilie-lagre som Cassandra og HBase og dokumentdatabaser som MongoDB og CouchDB. Derfor definerer Sadalage and Fowler (2013) en felles kategori for disse tre NoSQL-typene: ”Aggregatorienterte databasesystem”.

Begrepet ”aggregat” (må ikke forveksles med det matematiske verbet som betegner en operasjon på en gruppe av tupler) er lånt fra domenedrevet design og er definert som en samling sammenknyttede objekter som en datamodellør ønsker å behandle som en datamanipulasjonsenhet. Når komplekse aggregater aksesseres, gjøres det med et oppslag på én enkelt nøkkel, så får man både dataobjektet med den tilhørende nøkkelen samt eventuelle assosierte dataobjekter. Å utføre en tilsvarende lesing av to assosierte relasjoner i for eksempel MariaDB krever først oppslag i en tabell på dens nøkkelverdi, deretter enda et oppslag på en fremmednøkkel i den assosierte tabellen, altså må en JOIN-operasjon utføres.

Lesing av aggregerte dataobjekter medfører altså at man med ett enkelt oppslag får både i pose og sekk. Aggregatmodellen er også en enklere datamodell å forholde seg til for de som programmerer selve applikasjonen som behandler dataene. De enkelte aggregater, det vil si applikasjonsprogrammererens definisjon for databehandlingsenhet utgjør en naturlig enhet for replikering i en klynge av enkeltstående databasenoder. I et distribuert databasesystem gjelder det å minimalisere antall noder som kontaktes for hver spørring. Når konsepter settes sammen eksplisitt i datamodellen slik som vi ser i de fleksible dokumentstrukturene til Mongo, vet databasen hvilke dataenheter som skal aksesseres samtidig, og som derfor naturlig nok bør plasseres på én og samme node.

Sadalage and Fowler (2013) kaller relasjonelle database og grafdatabaser for **aggregat-uvitende**. Deres datamodeller betrakter ikke aggregater eller sammensatte datastrukturer i deres dataoperasjoner. Aggregat-uvitenhet er ikke nødvendigvis et dårlig designvalg, ettersom det ikke alltid er opplagt for den enkelte webapplikasjonsutvikler hvilke enhetsbegrensinger i datamodellen som er logiske, iallfall ikke før datamodellen er definert for første gang og revidert to til tre ganger i løpet av utviklingsprosessen. Den lagrede dataen

kan ha mange forskjellige brukskontekster, avhengig av applikasjonens funksjonelle krav som ofte blir forandret underveis i applikasjonens livssyklus.

En enkelt aggregatstruktur kan ikke medføre optimale spørringsytelse for alle mulige brukskontekster. Her gjelder det for utvikleren å prioritere den mest typiske leseoperasjonen tjenesten utsettes for. Hvis applikasjonen ikke har en slik primær aksess – struktur på dataobjektene kan man like godt modellere dem på et aggregat-uvitende vis. I en aggregat-uvitende modell har brukskonteksten ingen innvirkning på spørringen, fordi operasjonsenheten er én enkelt tupel i MariaDB (én enkelt data-node i en grafdatabase som Orient) uansett hvordan konseptene er satt sammen.

Transaksjoner er en velprøvd og høyt akseptert logisk modell for databehandling. Relasjonelle databaser tillater oppdatering av eller lesing av tupler i opptil flere relasjoner innen et sett med atomiske operasjoner. Det er den enkelte mengden av hendelser som heter for en transaksjon. En transaksjon avgrenser mengden av hendelser og skriver enten samtlige eller ingen endringer til disk.

Transaksjonenes egenskaper beskrives med akronymet ACID: De er atomiske, dvs at samtlige hendelser i transaksjonen blir enten gjennomført fullstendig eller ei; konsistente (eng. ”Consistent”), dvs at to transaksjoner som kjører parallellt alltid medfører det samme sluttresultatet; isolerte, det vil si holdbare i den grad transaksjonen persisteres til disk (eng. ”Durable”). Transaksjonsmodellen fremmer en spesifikk handling, `COMMIT`, som signaliserer at endringene spesifisert i den enkelte transaksjon er blitt gjort permanente.

Aggregatorienterte databasesystemer innehar ikke ACID - egenskapene på samme vis som relasjonelle databasesystemer. Imidlertid støtter de naturlig atomiske manipulasjoner på ett eneste aggregat av gangen. Operasjoner på flere aggregater må derfor handteres i kildekoden til applikasjonen, spørring for spørring, der et unntak kastes hvis én av spørringene mislykkes. Å emulere transaksjoner i enkeltaggregater inngår som en viktig faktor i hvordan aggregatene defineres i datamodellen (Sadalage and Fowler, 2013).

Fowler og Sadalage omtaler tre datamodeller som opererer med aggregater. Nøkkel-verdimodellen behandler aggregat som en ugjennomsiktig helhet (Sadalage and Fowler, 2013). Altså går det ikke an å hente deler av aggregatet ved et nøkkeloppslag. Dokumentmodellen eksponerer aggregatet til databasen, og tillater dermed delvise spørringer. I og med at dokumentmodellen også er skjemaløs går det ikke an å optimalisere spørringer på hele eller deler av aggregatet. Kolonnefamilier inndeler aggregatet i grupper, noe som tillater databasen å operere på hver av disse gruppene som en egen dataenhet, lik som attributter i tuplene i den relasjonelle modellen. Selv om kolonnefamilier gir opp full skjemaløshet, har databasen nå

mulighet til å nytte eksponeringen av attributter/kolonner til å optimalisere aksesseringer.

Semistrukturerte datamodeller, en iboende egenskap i de aggregatorienterte modellene Sadalage and Fowler (2013) presenterer, gir den som er interessert i rullerende oppgradering av distribuerte webapplikasjoner en svært fleksibel vei hva angår dataskjema-oppgradering. Sett fra databasetjenesten sitt perspektiv er det ikke noe i veien for at strukturen i "verdiene" tilknyttet nøklene i datalageret ikke er samstemte seg imellom, i motsetning til den rigide ordningen av tupler i relasjonsdatabaser som MariaDB og PostgresSQL. Det er nemlig ikke alle typer skjemaendringer som lar seg utføres i databasetjenere på rullerende vis. Et gjenstående problem med rullerende applikasjonsoppdatering er, uansett hvor strukturert applikasjonens datamodell er, at mens node etter node skiftes ut til en ny instans vil instanser av den oppgraderte applikasjonstjeneren generelt sett respondere på brukerforespørsler på en annerledes måte enn instanser av den gamle versjonen som ennå ikke er oppgradert. Dette problemet er kjent under navnet "Mixed Version Race", som er et gjennomgående tema i Dumitraş et al. (2010); Dumitraş and Narasimhan (2009).

## 2.2 Amazon Dynamo

DeCandia et al. (2007) presenterer arkitekturen til Dynamo, et høytilgjengelig, distribuert nøkkel-verdi-lager som ved nettverkspartisjoner ofrer replikakonsistens til fordel for å være mottakelig for lese - og skriveforespørsler fra diverse mikrotjenester som lever i Amazons applikasjoner. I kjernen av problemet denne distribuerte databasearkitekturen forsøker å løse ligger et sterkt krav til at enhver kunde av Amazons netthandel alltid skal kunne legge artikler i handlekurven. Handlekurven i nettbutikken, så vel som hver eneste artikkel i nettbutikken skal til enhver tid kunne interageres med. Ethvert avbrudd i denne tjenesten kan og vil medføre monetære tap enten direkte i form av utsatte handler og indirekte i form av øøkende mistro hos forbrukerne. Amazon.com er samlet sett en gigantisk, distribuert netthandelapplikasjon bestående av mange tusen nettverkskomponenter og uavhengige tjenere spredt utover mange datasentre. I denne infrastrukturen er feil som oppstår i enkeltkomponenter normen, ikke unntaket.

Amazon.com er en av verdens største netthandelplattformer. På det meste handler 10 millioner brukere på nettbutikken samtidig.

Dynamos ring av noder som lagrer data er et likemannsnettverk.

# Bibliografi

Bass, L., Clements, P., Kazman, R., 2013. Software architecture in practice.

Choi, A., 2009. Online application upgrade using edition-based redefinition. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. ACM, p. 4.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W., 2007. Dynamo: Amazon's highly available key-value store. In: SOSP'07 - Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles. pp. 205–220.

Dumitraş, T., Narasimhan, P., 2009. Why do upgrades fail and what can we do about it? toward dependable, online upgrades in enterprise system. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. Springer-Verlag New York, Inc., p. 18.

Dumitraş, T., Narasimhan, P., Tilevich, E., 2010. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In: ACM Sigplan Notices. Vol. 45. ACM, pp. 865–876.

Oliveira, F., Nagaraja, K., Bachwani, R., Bianchini, R., Martin, R. P., Nguyen, T. D., 2006. Understanding and validating database system administration. In: USENIX Annual Technical Conference, General Track. Boston, MA, pp. 213–228.

Oppenheimer, D., Ganapathi, A., Patterson, D. A., 2003. Why do internet services fail, and what can be done about it? In: USENIX symposium on internet technologies and systems. Vol. 67. Seattle, WA, pp. 11–25.

Sadalage, P., Fowler, M., 2013. Nosql distilled : a brief guide to the emerging world of polyglot persistence.

Schiller, K., 06 2011. Amazon ec2 outage highlights risks. Information Today 28 (6), 10, name - Amazon.com Inc; Copyright - Copyright Information Today, Inc. Jun 2011; Document feature - Photographs; Last updated - 2013-06-27.