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 netthandelsystemer er kunsten å minimalisere nedetid som følger av oppdatering av systemet. For mange store bedrifter som eier og administrerer slike systemer er det totalt uaktuelt å slå av 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 på levende vis, det vil si at oppgraderingen 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, eller ”skjema” som det heter i relasjonelle databaser, mens den opererer i et produksjonsmiljø.

Når et programvaresystem som opererer med mange brukere og mye trafikk oppgraderes levende befinner det i en spesiell tilstand: Versjonsmiks. I et distribuert, versjonsmikset system eksisterer to forskjellige applikasjonsinstanser parallelt på forskjellige tjenestenoder, med to forskjellige utgaver av datamodellen. I denne tilstanden kan det oppstå en spesiell type tjenestesvikt, der den oppdaterte applikasjonslogikken slår opp på et dataobjekt opprettet eller oppdatert i en tidligere versjon av applikasjonens logikk slik at data i aggregatet ikke passer den nye applikasjonslogikken.

Denne masteroppgaven setter som mål å realisere støtte for levende oppgradering av datamodeller 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. Dette verktøyet setter også som mål å dirigere tjenesteforespørsler slik at enhver applikasjonsinstans kun opererer på dataobjekter skrevet på en matchende datamodell, og således unngå systemfeil relatert til versjonsmiksing.

Abstract

Modern, commercial software systems often cater to hundreds of thousands unique users over the Internet, typically by use of HTTP - the Hypertext Transfer Protocol. It is on such large-scale systems NoSQL databases are practical to use to persist application data, as they are more capable than conventional relational databases at storing large volumes of data that are persisted rapidly and frequently on the system. NoSQL databases scale horizontally, which means that instead of upgrading a database server by replacing it with newer, more powerful hardware, a cheap server with poor hardware specifications is added to an existing cluster of similar cheap servers. Data items called ‘’aggregates” are partitioned and replicated across the members of this cluster.

One of the biggest challenges presented by the large-scale production system is updating its source code with improvements and fault repairs. For the commercial firm which owns and administrates this system, completely decommissioning it for maintenance is out of the question, as the potential income lost the service shutdown entails would be intolerable. Thus, many such systems are upgraded in a rolling fashion, which means that at most one server is turned off during upgrade at the time. Instead, individual program processes are replaced one by one for each application server that services requests.

As such the servicing of user request can be done continuously despite of ongoing maintenance operations. Installing patches this way is not without downsides. The rolling update opens for one strange anomaly: Mixed versions. In this system state, a new, upgraded instance of the server may look up a persisted data objected that was either updated or created on the old data model. The query will execute and finish, but the behaviour in the frontend following the delivery of the result will likely cause unintended consequences because of the schema mismatch between that of the tuple and the application instance submitting the query.

This thesis presents the module DBUpgradinator, a continuous migration tool written primarily for aggregate - oriented data models such as Key-Value. It is a support tool for administering live migration of aggregates without allowing queries across different data models through identifying the schema versions explicitly. The tool migrates each aggregate when it is requested by an application client. This lazy migration strategy is compatible with high availability requirements, as it enables a lazy data migration service that is suited to a rolling upgrade of the web application.

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 e-kommersiell setting.

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.

Trondheim, 27. mai 2018

Vegard Bjerkli Bugge

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

Innhold

Kodeoppføringer

Figurer

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 rette opp i ettertid.

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, må programvarearkitekter som vil gjeste sine systemer på EC2 ta høyde for den halve prosentandelen, der plattformen ikke er tilgjengelig for tjenesteleveranse.

Hvert sekund nedetid teller når det kommer til høyt trafikkerte 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

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

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 bemerkning 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. For å estimere denne verdien måles tiden fra en programvareinstans slås av til en ny startes opp, for eksempel ved en programvareoppdatering.

Alternativt kan man estimere en forventningsverdi for hvor lang tid det tar å reparere eller maskere nevnte feil, slik at tjenester kan leveres av systemet som normalt. For å estimere denne forventningsverdien måles hvor lang tid en programvareoppdatering 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 feilkilden (i over 50 prosent av problemene som ble oppgitt under intervjuene) til at disse feiltypene 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 programvareoppgradering 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 installasjonen ikke forstyrre applikasjonens leveranse av tjenester, e.g. behandling av innkommende HTTP-forespørsler.

Kunsten å holde styr på tredjeparts pakker som en programvaremodul er avhengig av, er et annet sentralt problem relatert til levende programvareoppgradering. En modul er avhengig av en pakke hvis den kaller på funksjoner fra pakken. Dette må gjøres for å oppdage om den gamle og nye versjonen har delte avhengigheter, det vil si at begge versjoner 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 håndterer avhengigheter, en form for manuelt skrevet konfigurasjonsfil der alle avhengigheter listes i form av par av unike pakkenavn og påkrevd versjon. For eksempel leser pakkehåndtereren 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 oppgradering*, 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 som regel er 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 at det distribuerte produksjonsmiljøet befinner seg i en blandet tilstand. Da kan en uoppgradert, kjørende webapplikasjon potensielt interagere med en oppgradert datamodell og vice versa. Denne kommunikasjonen 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 i en tenkt applikasjons implisitte skjema.

Oppgaven har hatt følgende overordnede mål:

1. Beskrive sammenhengen mellom kontinuerlig programvareleveranse og levende oppgradering av datamodeller
2. Modellere og implementere en modulær løsning der semistrukturerte datamodeller, også referert til som NoSQL-datamodeller, kan oppgraderes synkront med applikasjonslogikken til webapplikasjoner

1.3 Oppgavens struktur

Denne rapporten har følgende struktur. Kapittel 1 er introduksjonskapitlet, som illustrerer oppgavens problemstilllingen, og hvordan den skal løses.

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 diskuteres løsningsforslagene som både oppgraderer en applikasjons implisitte datamodell og av distribuerte databasesystem, som ble funnet i løpet av et litteratursøk høsten 2017. Denne sammenliknignen er fortrinnsvis kvalitativ. 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 presenteres 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 aggregatorienterte datamodeller*. Dette kapitlet vil presentere systemdesignet rundt en kommersiell webapplikasjon kalt ”DBUpgradinator”, som inneholder en simulert frontend 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 beskrevne krav til datamodellevolusjonsverktøyet evalueres. Under testing vil webapplikasjonen DBUpgradinator gjen-

nomgå 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 feltarbeid som kan bygge på denne analysen.

Kapittel 2

Teori

Ifølge Sadalage and Fowler (2013) kan nøkkelverdi-lagre (eng. key-value store), kolonnefamilielagre (eng. column family stores) og dokumentlagre (eng. ”document stores”) ordnes under én og samme ”art” av NoSQL-databaser: Aggregatorienterte databasesystem (eng. ”aggregate oriented databases”). Denne introduksjonen til de tre nevnte NoSQL - datamodellene forholder seg til denne klassifiseringen. Utover disse tre typene av NoSQL - modeller finnes også den graforienterte modellen, som benyttes av systemer som InfiniteGraph, OrientDB og FlockDB. Denne datamodellen vil ikke bli beskrevet i noen videre detalj. Det bør nevnes at med begrepet ”datamodell” menes den programmatiske metoden data organiseres av et DBMS, i vitenskaplig notasjon er ”metamodell” mer presist.

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.

Vi begynner med å beskrive den relasjonelle datamodellen, og hvorfor den ikke holder mål i et distribuert produksjonsmiljø der større datavolum behandles. Dernest blir den aggregatorienterte datamodellen presentert, en kategori underordnet NoSQL-paraplyen som denoterer fellesnevneren mellom nøkkelverdilagre, dokumentlagre og kolonnefamilielagre. Samtlige tre former for aggregatorientering beskrives i avsnitt 2.2.1 til 2.2.3.

2.1 Den relasjonelle datamodellen

For å forstå framveksten av NoSQL - databaser, er det hensiktsmessig å forstå forskjellen mellom dem og relasjonelle databaser. I den relasjonelle datamodellen organiseres forskjellige former for applikasjonsdata inn i relasjoner, og data tilhørende samme relasjon inndeles i atomiske, disjunkte enheter kalt *tupler*. En tuppel er en flat, endimensjonal liste

av dataverdier. Hver av disse verdiene korresponderer til nøyaktig ett attributt av relasjonen tuppelen er lagret i.

Det foreligger visse begrensninger på denne datastrukturen. Til eksempel kan ikke en enkelt tuppel nøstes inn i en annen, og hvert attributt i tuppelen har én atomisk korresponderende verdi, aldri en liste av verdier. Nå skal det sies at nyere versjoner av MariaDB støtter JSON-objekter som datatype (MariaDB, 2017). JSON-objekter er serialiserte (dvs objekter konverterte til strenger), fleksible dataenheter som kan inneholde nøstede datastrukturer.

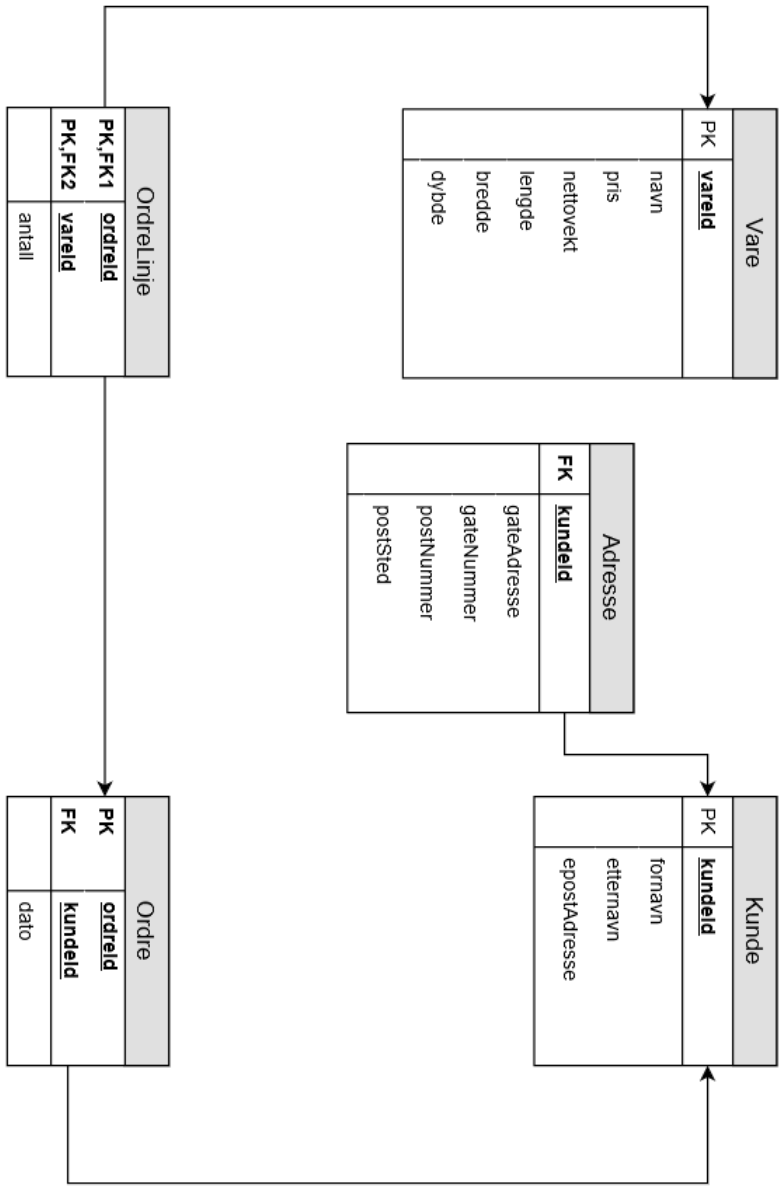
Imidlertid har ikke databasesystemet noen forståelse for de enkelte dataelementer som objektet innkapsler, det ser bare en helhetlig, ugjennomsiktlig dataenhet, nemlig verdien av ett attributt i relasjonen. Ettersom tupler er den minste, udelelige dataenheten i den relasjonelle modellen er det korrekt å fastslå at spørringer opererer med og returnerer (et helt antall) tupler for hver enkelt spørring (Sadalage and Fowler, 2013). Riktignok går det an å utvelge distinkte attributter i spørringen, også kjent som kommandoen `PROJECT` i relasjonell algebra, likefullt er den tellbare dataenheten i resultantrelasjonen tupler, også referert til som rader i kontekst av databaseapplikasjoner.

Følgelig gir slike strengt strukturerte datamodeller stor fleksibilitet for spørringene som utføres av databasesystemet. Det kan for eksempel samle sammen alle verdier for ett spesifikt numerisk attributt i én bestemt relasjon, summere disse attributtverdiene sammen og returnere resultatet som et separat attributt i resultanttuppelen. På samme vis kan relasjonsdatabasesystemet kalkulere gjennomsnittet for alle eller enkelte av tuplene i en relasjon, telle opp antallet tupler i den, finne den høyeste numeriske verdien eller finne den laveste.

Ved hjelp av JOIN-operasjoner finner man eksisterende tupler fra forskjellige relasjoner, som er forbundet via fremmednøkler. Systemet kan også utføre aggreggeringsspørringer på resultanten av JOIN-spørringen. NoSQL - databaser har ikke den samme fleksibiliteten til selv å utføre slike spørreoperasjoner: Hver spørring henter kun én dataenhet ad gangen, noe som holder især for nøkkelverdilagre. Hver spørring er logisk sett bare et oppslag på en nøkkel i en hashtabell. Eventuell aggregering der sum, gjennomsnitt eller ekstremalverdier regnes må gjøres i selve applikasjonen, etter at oppslag på **samtlige** nøkler er gjort i databasen.

2.1.1 Impedansproblemet (Impedance mismatch problem)

I en typisk netthandelapplikasjon er applikasjonens datamodell denotert i form av dataobjekter midlertidig lagret i primærminnet, mer presist sagt, i adresseområdet til èn, eller flere, av nettleserens kjørende prosesser i kundens datamaskin. Variablene i datafeltene i disse objektene kan være saå mangt: strenger, tall, referenser til andre objekter, lister bestående av ovennevnte typer. Disse dataobjektene endres i sanntid av kunden som aksesserer nettbutikken gjennom standardiserte interaksjonselementer som tekstfelt, knapper, slidere, sjekkbokser og radiobokser, gruppert sammen i dynamiske sideelementer kjent som skjema (eng. ”form”). Et praktisk eksempel på et skjema i en netthandel er handlevognen, som viser kunden hvilke vareartikler han (foreløpig) vil kjøpe, hvor mange av hver vare som ”ligger” i handlevogna, og hvor mye varene koster sammenlagt.



Figur 2.1: Diagram over tabeller i en relasjonsdatabase som registrerer ordrer fra kunder i en nett-handelapplikasjon.

Når det kommer til å skrive dataene fra web-skjemaet til disk i en relasjonsdatabase ser ting annerledes ut. Her persisteres data om varer (navn, identifikasjonsnummer, størrelsesdimensjoner, nettovekt, og enhetspris, gitt at hver enkelt vare-tuppel representerer en diskret enhet for

eksempel én bøtte maling eller én sekk med poteter) til en egen relasjon. En annen relasjon holder data om kunder, en tredje holder på informasjon om bestillinger, og for normaliseringens skyld eksisterer en egen jointabell kalt Ordrelinje som kopler sammen Ordre og Vare, som illustrert i tabell-diagrammet i 2.1. Denne uoverensstemmelsen mellom strukteren av applikasjonsdata i programminne og strukturen på de samme dataene i en relasjonsdatabase, refereres til i industrien som "the impedance mismatch" (Sadalage and Fowler, 2013).

Følgelig må applikasjonsutviklere konvertere data fra spørreresultater til den dataobjektstrukturen applikasjonen påkrever, noe som per idag ofte løses ved å innføre et seperat abstrahert lag i applikasjonens logiske arkitektur: En tredjepartsmodul kalt "object-relational mapper" (ORM). For språket Java kan man bruke Hibernate[1], JavaScript har SequelizeJS[2], og PHP-utviklere kan bruke Doctrine[3], som også er en del av webapplikasjonsrammeverket Symfony[4]. Med slike tredjepartsbibliotek følger et nytt mønster som utviklere blir nødt til å forholde seg til, hvis de skal ta det i bruk. Ettersom det relasjonelle datalaget abstraheres bort, er det tilforlatelig å glemme at applikasjonsdata faktisk blir persistert i en relasjonell database. En tilsynelatende enkel henteoperasjon på objektform kan bety to kostbare JOIN-operasjoner i databasen som kjøres hver gang spørringen utføres.

En annen innvending mot den relasjonelle datamodellen involverer hvordan den støtter skalering av arbeidslast ved økende antall brukere, økende antall datakilder, og økende spørrefrekvens databasesystemet utsettes for. Datamodellen ble etablert på 70-tallet, i en æra lenge før distribuert databeregning tok av i bedriftsmarkedet. Relasjonsdatabaser er designet med tanke på monolittiske systemarkitekturer, programvarearkitekturer hvis system kjører på én enkel datamaskin, fordelt på et sett med prosesser innad i den. For et begrenset antall datakilder/brukere er slike systemer i noen grad vertikalt skalerbare, det vil si at økt last på systemet kan løses ved å oppgradere maskinvaren. Denne metoden er innlysende nok kostbar i det lange løp ettersom maskinvare som skiftes ut ikke er brukelig for systemet lengre. Da er det billigere å skalere horisontalt, det vil si å kjøre programvaresystemet i en klynge av datamaskiner sammenkoplet over et IP - nettverk. Hver av disse datamaskinene er billige, altså består de av maskinvarekomponenter som yter dårligere enn den jevne stordatamaskin som prosesserer banktransaksjoner. Som demonstrert i følgende eksempel statuert av George (2011) skal vi se at å operere i klynget system ikke er så lett når data modelleres relasjonelt.

2.1.2 Hush

Hush er en (fiktiv) url-forkortelsestjeneste som i begynnelsen har omtrent et par tusen brukere, og vedlikeholdes og bygges med gratis tredjepartsmoduler, blant annet driftes en LAMP-tjener (Linux, Apache, MariaDB, PHP) som leverer en prototype av denne tjenesten i form av en webapplikasjon. Hush sin relasjonelle databasemodell normaliserer sine data ved å definere fire tabeller, `user`, `url`, `shorturl`, og `click` (George, 2011). De

[1] `http://hibernate.org/orm/`
[2] `http://docs.sequelizejs.com/`
[3] `http://www.doctrine-project.org/`
[4] `http://symfony.com/`

tre sistnevnte tabellene er assosiert med `user` gjennom en fremmednøkkel som refererer til nøkkelattributtet til den tabellen. I tillegg er brukertabellen og kort-URL-tabellen indeksert etter sine respektive nøkkelattributter for å gjøre oppslag på korte URLer og brukere raskere. Ved å sluse endringer inn i systemet som transaksjoner, sikrer man at de relaterte tabellene (den for URLer, korte URLer og klikk) endres sekvensielt og fullstendig uavhengig av hvor samtidig de uavhengige skriveoperajsonene forekommer slik at et strengt konsistensnivå opprettholdes tuplene imellom.

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 kalles 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.

Denne monolittiske databasearkitekturen fungerer med det gitte antall brukere. Idet tjenesten blir verdenskjent, og antallet brukere øker eksponensielt med fire tierpotenser, blir arbeidslasten for databasetjeneren etter hvert for stor å handtere alene. Den naturlige løsningen på å tekkes vekstraten i databasens arbeidslast er å innføre flere databasetjenere installert på separate datamaskiner. Når skriveoperajsoner og leseoperasjoner i et databasesystem distribueres utover en klynge tjenere, er det viktig å organisere dem slik at arbeidslasten av skrivinger og lesinger jevnfordeles metodisk slik at databasesystemets distribuerte natur ikke er synlig for applikasjonen som utfører spørringen. En vanlig organiseringsmetode er master-slave-replikering, der én master-tjener mottar alle skriveoperasjoner for å serialisere dem (George, 2011).

I historien om Hush er dette veien dets utviklere tar: Slavetjenerne får motta lesespørringer, én enkel mastertjener fordeler skriveoperasjoner blant slavene. Hush er en applikasjon der leseoperasjoner utnummerer skriveoperasjoner i antall, det hender jo oftere at noen klikker på en forkortet lenke snarere enn at noen poster en lenkeforkortelse på tjenesten. For en stakket stund fungerer denne lastfordelingen utover klyngen, men etterhvert er tilveksten av brukere såpass stor at leseoperasjonene samlet sett blir trege. Etter hvert blir også masterdatabasetjeneren som håndterer samtlige skriveoperasjoner blir en flaskehals i systemet (George, 2011).

For å øke ytelsen til leseoperasjonen installerer utviklerne av Hush et distribuert hurtiglager med det minnebaserte nøkkelverdilageret Memcached[5]. Imidlertid svekkes oppdateringskonsistensnivået til systemet ettersom dataverdier i hurtiglageret må skiftes ut etter hvert som transaksjoner behandles av mastertjeneren. For å holde tritt med den økende skrivelasten kunne man oppgradere mastertjenerens maskinvare, altså å skalere oppover.

[5] `https://www.memcached.org/`

Dennne løsningen er lite bærekraftig i lengden, ettersom det finnes et øvre fysisk tak på antallet mikrotransistorer som kan få plass innen én kvadratmillimeter mikrochip. Løsningen er også svært kostbar, fordi slavetjenernes maskinvare må nødvendigvis også oppgraderes i lengden for å holde tritt med de stadig innkommende skriveforespørslene fra mastertjeneren. På toppen av disse bekymringene går også utførelse av JOIN - operasjonene for tregt for at systemene skal kunne holde tritt med den økende frekvensen av spørringer fra applikasjonstjenerne, så man velger da å denormalisere tabellene. Det er nå kommet tydelig fram at den relasjonelle datamodellen nå er til mer bry enn den er til hjelp for Hush-utviklerne.

Av denne historien kan man oppsummere at relasjonelle databasesytemer ikke er laget for å kjøre i et distribuert miljø. Ei heller lar de seg skaleres horisontalt, altså at løsningen på økende arbeidslast er å legge til en datamaskin med billige maskinvarekomponenter i et distribuert nettverk av andre liknende datamaskiner og jevnfordele spørringene utover dem. En relasjonsdatabasetjener kan istedet skaleres vertikalt, det vil si at prosessorenheter med høyere klokkefrekvenser og minnekort og harddisker med større datakapasitet installeres i tjenermaskinen og erstatter de gamle. Ikke bare er dette en utålelig dyr løsning i lengden, men den krever også lange vedlikeholdsperioder. Hos dagens moderne skytjenester tar riktignok et bytte av tjenermaskin kort tid, på grunn av at maskinene er virtuelle, ikke fysiske. Derfor blir pausen der applikasjonen ikke kan betjene forespørsler litt kortere. For å takle lagring og behandling av stadig større og raskere datavolum, må vi se nærmere på en nyere og mer fleksibel måte å strukturere lagret data på.

2.2 Den aggregatorienterte datamodellen

Så har vi den aggregatorienterte modellen, en metamodell som tillater den enkelte systemarkitekt å selv definere kompleksiteten til strukturen til sine egne dataenheter, slik at persisterte data er tilpasset applikasjonens struktur på sine dataobjekter 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 nøkkelverdilagre 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 i kontekst av databasemodellering definert som en samling sammenknyttede objekter som en datamodellør ønsker å behandle som en enhet for datamanipulasjon og konsistenshandtering. 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 MySQL 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. Med begrepet dataobjekt

menes en serialisert, distinkt, flatt datastruktur på JSON-form. Et aggregat er til sammenlikning et dataobjekt med nøstede dataobjekter.

En aggregatmodell avgrenser den objektstrukturen til applikasjonens data som alltid skal skrives i ett, hvilket betyr at når data i et nøstet objekt endres, blir hele aggregatobjektet i seg selv omskrevet. Aggregatet utgjør dermed en naturlig enhet for replikering i et distribuert databasesystem, da hele den aggregerte objektstrukturen som programvarens forretningslogikk jobber innenfor, replikeres i sin helhet. En tuppel i en normalisert relasjon inneholder nødvendigvis ikke hele omfanget av dataobjektstrukturen som forretningslogikken opererer med, iallfall ikke uten en eller to JOIN-operasjoner.

Aggregatet utgjør også en naturlig enhet for partisjonering. En stor mengde av individuelle aggregater er fra programvaresystemet sitt sitt standpunkt aksessert fullstendig uavhengig av hverandre, derfor kan de fordeles tilfeldig, og kopieres utover et sett med uavhengig opererende databasenoder, uten at objektenes plassering får konsekvenser for applikasjonens aksessmønster - skal en klient ha tak i ett spesifikt objekt kan den i prinsippet kontaktet én spesifikk databasenode i nettverket som er kjent for å holde på dette ønskede objektet. I et relasjonelt, distribuert databasesystem innebærer partisjonering av tabeller negative konsekvenser for spørreytelsen.

Lesing av aggregerte dataobjekter medfører at man med ett enkelt oppslag på én enkel nøkkel 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, av den enkle grunn at de slipper å skrive kode for å konvertere en tilfeldig liste av flate tupler. 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 databaser 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 tuppel i MariaDB uansett hvordan konseptene er satt sammen.

Aggregatorienterte databasesystemer innehar ikke ACID - egenskapene som vi finner hos

transaksjoner i relasjonelle databasesystemer. Imidlertid støtter de naturlig atomiske manipulasjoner på ett eneste aggregat av gangen. Ved nøkkeloppslag får man hele dataobjektet den tilkoplete applikasjonen leser og manipulerer, Samtidighetskontroll ved operasjoner på flere aggregater må følgelig handteres i kildekoden til applikasjonen, spørring for spørring, der et unntak må 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).

2.2.1 Design av aggregatmodeller

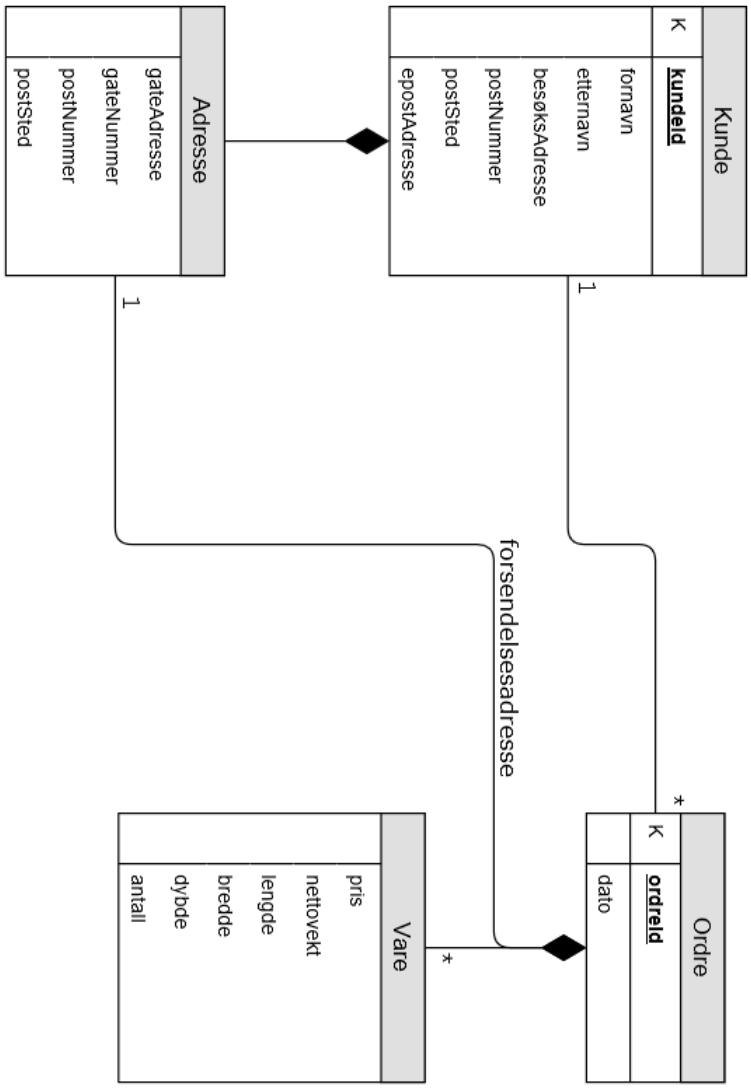
Slik kan en generisk aggregatmodell, uttrykt i UML, ekvivalent til datamodellen fra 2.1 se ut.

Denne figuren presenterer to forskjellige aggregatmodeller, `Kunde` og `Ordre`. Sadalage and Fowler (2013) demonstrerer et eksempel på en aggregatorientert modell i samme forretningsdomene, med de samme to entitetene. Forbindelsen mellom disse to denoterer en én-til-mange-assosiasjon som man kjenner til fra den relasjonelle modellen, men som ikke realiseres med fremmednøkler, iallfall ikke fremmednøkler slik man er vant til dem fra MySQL og PostgreSQL. I stedet indikerer denne assosiasjonen i diagrammet at hvert kundeaggregat har en flat liste bestående av `Ordre`-nøkler som representerer alle ordrene en kunde har opprettet i netthandelen. Likeledes kan hver enkelt ordre holde på en "fremmednøkkel" som kun applikasjonen kan tolke, og således er det i selve applikasjonslogikken av nettbutikksystemet at "JOIN" blir gjort. Komposisjonselementet indikerer bruk av nøsting i aggregatet, det vil si at et Vare-objekt til enhver tid eksisterer innenfor et aggregat, nemlig `Ordre`-entiten, og aldri som et selvstendig aggregat selv. I relasjonelle ER-modeller er slik nøsting av entiteter strengt forbudt. Aggregatorienterte modeller er således *denormaliserte*, det vil si at den enkelte aggregatmodell ikke oppfyller første normalform.

1NF-relasjoner består av flate tupler uten nøsting av relasjoner, eller som Codd (1971) definerer en unormalisert tabell: "[a] group schema which contains a repeating group schema". Med "repeating group" menes også attributter på listeform, for eksempel en liste av strenger. Videre tillater ikke Codd (1971) kryssreferanse per primærnøkkel innad i en relasjon, fordi det er en uønsket egenskap for en relasjon å ha, datamodellen blir følgelig rotete og vanskelig å lese for dens brukere.

En annen interressant detalj ved modellen presentert i 2.2 er at Adresse-objektet er inneholdt i både `Kunde`-aggregatmodellen og `Ordre`-aggregatmodellen. Semantikken bak er at hver ordre har en leveringsadresse som de tilknyttede varene leveres til. I praksis betyr dette at aggregater av både `Ordre` og `Kunde` lagrer potensielt samme adresseinfo, altså er adressedataobjekt lagret dobbelt opp i databasen, den er *duplisert*.

Ved modellering av aggregater må man avveie mellom hvor stort hvert enkelt aggregatobjekt i en applikasjon kan bli og hvor mange forskjellige aggregatentiter applikasjonen kan ha. Alternativt kunne datamodellen i 2.2 bestått av ett eneste aggregat, noe man oppnår ved å erstatte assosiasjonslinjen mellom `Kunde` og `Ordre` med en komposisjonslinje. Med to nivåer av nøstede lister av objekter innen ett aggregat, kan besvarelse av ett enkelt oppslag



Figur 2.2: Aggregatdiagram med to forskjellige entiter, modellert for den samme tjenesten fra 2.1.

på én spesifikk nøkkel potensielt medføre et så stort svar i retur at HTTP-responsen må deles opp i flere TCP-pakker, hvilket igjen er delt opp i mange IP-pakker som tar forskjellige ruter til den spørrende klienten. Store, komplekse aggregat vil innvirke på spørringens nettverksforsinkelse. Sadalage and Fowler (2013) poengterer at hvis applikasjonen alltid har bruk for å hente informasjon om ordre, samtidig som den henter informasjon om en kunde, så er det hensiktsmessig å legge ordrehistorikken til hver kunde inn under Kunde-aggregatet. Hvis applikasjonen fokuserer på kun én ordre ad gangen i dets brukervisning,

så vil todelingen av aggregater som vist i 2.2 være mer hensiktsmessig. Hvordan aggregater defineres er alt i alt opp til den enkelte datamodellør.

Fowler og Sadalage omtaler tre unike datamodeller som opererer med aggregater. Nøkkelverdimodellen behandler det enkelte 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, liksom attributter i tuplene i den relasjonelle modellen. Selv om kolonnefamilier til dels ofrer den komplette skjemaløsheten som vi ser i nøkkelverdimodellen, har databasen nå mulighet til å nytte eksponeringen av attributter/kolonner til å optimalisere aksesseringer og oppdatere separate kolonner.

2.2.2 Nøkkelverdimodellen

Nøkkelverdi-lagre er den type NoSQL-DBMS med den enkleste aggregatorienterte datamodellen. Dens hovedkarakteristikk er at hvert aggregat som lagres må være tilknyttet én unik identifikator kalt **nøkkel**, og for å hentet det lagrede aggregatet må man vite verdien til denne nøkkelen (Elmasri, 2016). En nøkkel er en helt unik streng som brukes av databasesystemet til å lokalisere raskt et assosisert dataobjekt (også kalt **verdi**) i en homogen klynge av databasenoder (Elmasri, 2016). Hvert aggregat som lagres er ugjennomsiktig, det er en stor boble av serialiserte bytes som databasesystemet ikke kan anskue. I prinsippet er nøkkelverdimodellen totalt ustrukturert, altså er det opp til selve applikasjonen hvis data lagres i nøkkelverdidatabasen å påføre dataobjektet mening og definere dets struktur (Elmasri, 2016). Fordelen med at aggregatet ikke synes i databasen er at applikasjonsutvikleren kan endre aggregatets struktur, også kjent som dets implisitte skjema, helt etter eget ønske. Man kan lagre hva man enn vil i et nøkkelverdilager så lenge man spesifiserer en nøkkel databasen bruker til å lete opp dataobjektet med, og følger en eventuell størrelsesbegrensning definert av databasens konfigurasjon (Sadalage and Fowler, 2013).

Spørringer skrives ikke i et domenespesifikt språk som for eksempel SQL. I stedet kaller applikasjonen på et sett funksjoner eksponert for den gjennom et applikasjonsprogrammeringsgrensesnitt (API). Dette APIet tilbyr hovedsaklig tre forskjellige spørringer: Lesespørring (`GET`), skrivespørring (`PUT`) og slettespørring (`DELETE`). Både oppdateringer og opprettelser av dataverdier gjøres med én og samme kommando, PUT. Enkelte nøkkelverdidatabaser kan supplere med flere funksjoner til for eksempel administrative behov. Hver enkelt spørring behandler ett helhetlig aggregat av gangen. En lesespørring på en nøkkel henter ut hele aggregatobjektet assosiert med nøkkelen. En skrivespørring som oppdaterer et dataobjekt skriver over eksisterende data assosiert med objektets nøkkel fullstendig.

Moderne, populære NoSQL-systemer avviker noe fra dette prinsippet som skiller nøkkelverdimodellen fra dokumentmodellen. I dokumentlagre går det an å definere et ID-felt, en primærnøkkel, brukt til å gjøre ID-oppslag på samme vis som i et nøkkelverdilager. Riak KV tilla-

ter applikasjonsutvikleren å legge inn metadata direkte inn i det lagrede aggregatobjektet slik at deler av aggregatet indekseres, og det samme systemet implementerer også en form for assosiasjon mellom aggregater. Videre har Riak også en søkefunksjon som kan brukes på JSON-serialiserte aggregater (Sadalage and Fowler, 2013). Redis, et annet nøkkelverdisystem, støtter lagring på og oppslag av aggregatobjekter som ikke likner på tradisjonelle JSON-objekter, men objekter i form av lister, hashede verdier og mengder.

I kraft av databasens egenskap i å støtte kun én eneste indeks, nemlig oppslagsnøkkelen til dataobjektene, er nøkkelverdimodellen forenlig med systemkrav om at datavolumet som lagres skal jevnfordeles utover en klynge av uavhengig opererende databasenoder. Å ta høyde for aggregateksponering til datalageret, direktereferanser til andre aggregat, og delvis indeksering på aggregatet, gjør oppfyllelse horisontal skalerbarhet i databasen vanskeligere. Videre er kontroll av transaksjonskonsistens (også kjent som ”Consistency” i ACID-forkortelsen) en jobb som i utgangspunktet ”outsources” til applikasjonen, fordi nøkkelverdi-lageret er prinsipielt uvitende om den bakenforliggende semantikken til det enkelte dataobjekt.

2.2.3 Dokumentmodellen

I motsetning til nøkkelverdilagre er det enkelte aggregatobjekt sin datastruktur synlig for dokumentlagre, som også kan utføre både lesespørringer og oppdateringsspørringer på spesifikke feltvariable innen de lagrede aggregater, altså bare hente ut distinkte deler av dokumenter. Dokumentlageret kan lese av disse ”selvbeskrivende dataene” (Sadalage and Fowler, 2013; Elmasri, 2016) som finnes i aggregatobjektene, og derfor er slike delvise spørringer mulig. De enkelte attributter/feltvariable innen aggregater som er interressant for en applikasjon kan også indekseres av databasen på samme måte som relasjonsdatabaser er i stand til å indeksere enkeltattributter i relasjoner. Samtidig fører dokumentlageret begrensninger på hvordan strukturen til det lagrede aggregatet kan se ut, gjennom definisjoner av tillatte datatyper og objektstrukturer (Sadalage and Fowler, 2013). På tross av at dokumentdatabasen kan anskue aggregatets struktur opererer hver enkelt spørring på ett enkelt, helhetlig dokument av gangen.

Dokumentlagere inndeler vanligvis data i samlinger (eng. ”collections”) av dokumenter (eng. ”documents”). De enkelte dokumenter er aggregatobjekter utformet i et semistrukturert format, som XML eller JSON. Dokumenter lagret innen samme dokumentsamling tillates å ha forskjellige definerte eller udefinerte verdier for enkelte variable. Denne fleksible egenskapen er ikke å finne i den relasjonelle modellen.

Spesifikt i MongoDB lagres dokumenter i BSON, et binært format som er en utvidelse av JSON-standarden med noen ekstra datatyper, optimalisert for lagring på disk. For å opprette en ny dokumentsamling har MongoDB metoden `createCollection`, som tar inn samlingens navn og en mengde innstillinger som bestemmer begrensninger for samlingen, som for eksempel det totale datavolumet og det maksimale antallet dokumenter som samlingen kan holde på. Hvert dokument har et automatisk generert felt som MongoDB bruker som primærnøkkel, med mindre applikasjonsutvikleren definerer en slik indeks selv (Elmasri, 2016).

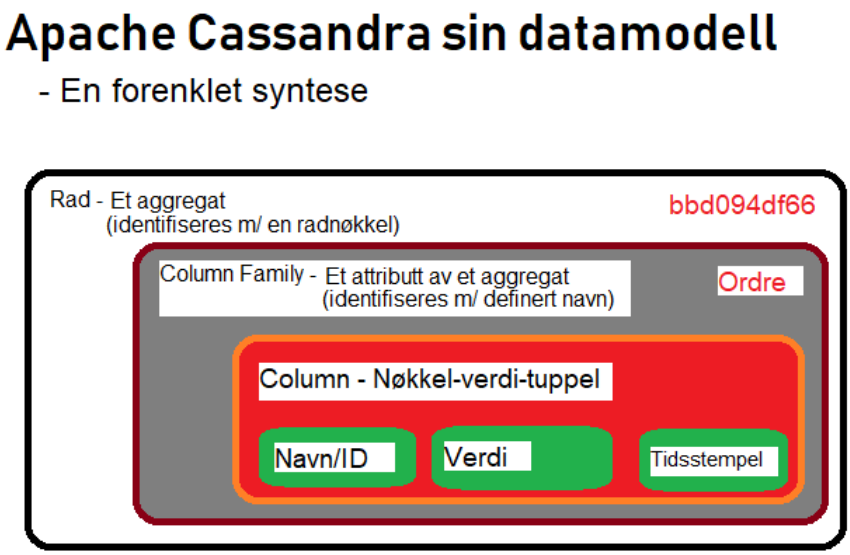
Dokumentsamlinger er ikke bundet til et skjema slik som relasjonelle databaser. Strukturen til aggregatet/dokumentet defineres av applikasjonens utviklere, og velges utifra applikasjonens krav til datamodellen, i stedet for at appliaksjonens logikk tilpasser seg en rigid, todimensjonal struktur slik vi ser i relasjoner. Følgelig blir det umulig for databasesystemet å optimalisere spørringer med basis i aggregatstrukturen, fordi den er ikke gitt før innsettingsspørringer tikker inn (i kontekst av Mongo via `insert` - funksjonen).

2.2.4 Kolonnefamiliemodellen

Googles BigTable er et databasesystem som har inspirert mange senere NoSQL-systemer, akkurat som Dynamo. BigTable er en ofte brukt lagringsteknologi hos Googles applikasjoner, deriblant Gmail (Elmasri, 2016). Apache HBase er et kolonnefamilielager tilgjengelig i åpen kildekode[6] som implementerer de sentrale teknikkene inneholdt i BigTable.

Kolonnefamilielagre kan beskrives av Elmasri (2016) som ordnede, flerdimensjonale, distribuerte ordlister, hvorav nøkkelverdilagre som Redis kan ansees å være enkeltdimensjonale, distribuerte ordlister. Den enkelte aggregat-nøkkel i et kolonnefamilielager består av flere distinkte komponenter, vanligvis et navn på samlingen av aggregater, en nøkkel som identifiserer selve aggregatet (”row key”), aggregatet i seg selv som også refereres til som en ”kolonne” og et tidsstempel (Elmasri, 2016).

[6] Tilgjengelig på følgende GitHub-repositorium: `https://github.com/apache/hbase`



Figur 2.3: Metamodell som viser forholdene mellom dataenheter i Cassandras logiske datamodell.

I et kolonnefamilelager er hvert enkelt aggregat identifisert av en todimensjonal nøkkel: Den ene komponenten er en radnøkkel og den andre er navnet på en kolonnefamile underordnet "raden", som i Cassandra er en samling aggregater (se 2.3). En kolonnefamilie er en samling nøkkel-verdi-tupler, ikke ulikt en tuppel og dens attributter i den relasjonelle datamodellen.

I eksempelet i 2.2 med aggregatmodellen måtte man i de øvrige to datamodellene velge mellom å ha èn entitet, Kunde med n Ordre underordnet, eller to altså at lesing av kundedata innebærer raskere lesing på disk, men innhenting av alle ordre må gjøres med n enkeltoppslag når listen av ordreID-er i kundens ordreliste er av størrelse n. I et kolonnefamilielager kan man i prinsippet få i pose og sekk ved å definere hvert aggregat til å representere en kunde og la samlingen av `Ordre`-entiteter underordnes som en kolonnefamilie i tillegg til å definere en annen kolonnefamilie, Profil, som inneholder kundens personlige data og leveringsadresse. Hvis applikasjonen nå er interresert i en kundes profildata, slipper den å hente alle dens ordre i tillegg.

2.3 Amazon Dynamo

Her rettes fokuset mot en berømt artikkel hvis primære fokus var å besvare Amazon sitt problem med skalering av skriveoperasjoner i et kommersielt netthandelsystem som betjener flere hundre tusen forbrukere verden over. Den hypotetiske systemarkitekturen artikkelen beskriver, Dynamo, er stamfar til mange populære NoSQL-databasesystemer lansert som åpen kildekode, deriblant Apache Cassandra, Basho Riak, Amazons DynamoDB og

Linkedins Project Voldemort. Drivkraften bak designet av dette likemannssystemet er den samme som den bak vårt ønske om å kunne migrere data levende i det distribuerte produksjonsmiljøet: Høyere tilgjengelighet for interaksjon mellom brukernes klientapplikasjoner og deres databasetjenere. Derfor er en god forståelse av problemstillingene og utfordringene Dynamo løser, samt hvordan løsningene kan implementeres, relevant for oppgavens problemstilling.

2.3.1 Bakgrunn for artikkelens arbeid

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 om 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, eller indirekte i form av økende mistro hos forbrukerne. Amazon.com er samlet sett en gigantisk, distribuert netthandelapplikasjon bestående av mange tusen nettverkskomponenter og uavhengige tjenere spredt utover mange datasentre. Feil oppstår kontinuerlig i enkeltkomponenter i dette systemet.

Amazon.com er en av verdens største e-kommersplattformer. Den består av flere tusen uavhengige tjenerkomponenter lokalisert i flere datasentre verden over, og betjener flere titalls millioner kunder samtidig ved julehøytider når besøkstrafikken er på sitt høyeste (Pepitone, 2010). Et knippe av disse tjenestene er illustrert som testimonale på økosystemets diversitet i figuren "Figure 1" i (DeCandia et al., 2007). Hvis én enkelt komponent i den avanserte tjenestearkitekturen netthandelen avhenger av for å være oppegående påkrever vedlikeholdsarbeid, går det ikke an å ta ned hele nettbutikken fullstendig for så mye som én time uten å tape en uutholdelig mengde millioner dollar i urealiserte inntekter. Derfor stilles svært høye ytelses - og pålitelighetskrav til denne e-kommersplattformen. I tillegg må dette komplekse systemet kunne skaleres horisontalt inkrementelt, altså at det distribuerte systemet kan utvides med én lagringsnode av gangen med minimal påvirkning på dets tjenesteytelsesevne.

I et stort, distribuert system vil fatale tjenerfeil, det vil si feilscenarier der en enkelt tjener følgelig blir utilgjengelig for dets brukere, inntreffe jevnlig, og med høy frekvens. Bevis: Hvis variabelen N denoterer et stort antall tjenere i et distribuert system, og sannsynligheten for at én tjener lider en fatal tjenerfeil i løpet av ett døgn estimeres til p, så er sannsynligheten for at minst én tjener i klyngen blir utilgjengelig gitt ved $1 - (1 - p)^s$. Et konkret eksempel: $p = 0.05, s = 50 => 1 - (1 - p)^s = 0.9231$. I en mellomstor klynge med 50 tjenere, der hver tjener feiler med en sannsynlighet på fem prosent i løpet av et døgn, så er sannsynligheten for at samtlige tjenere er oppegående i løpet av et helt døgn under åtte prosent. For at så lite datavolum skal være utilgjengelig til enhver tid i et distribuert databasesystem, må det fordele datalasten jevnt utover alle tjenerne, i tillegg til å bestå av mange tjenere.

Helt siden deres framvekst på 80 - tallet har applikasjonsdata i større produksjonsmiljø blitt lagret av relasjonelle databasesystemer. DeCandia et al. (2007) anser imdlertid transaksjonsbasert lagring som en suboptimal og ineffektiv løsning for sitt eget produksjonsmiljø, av hovedsaklig to årsaker:

Horisontal skalerbarhet Som vist i det tidligere nevnte hypotetiske systemet Hush (George, 2011) er tradisjonelle relasjonsdatabaser lite fleksible når det kommer til datareplikering i distribuerte databaser. DeCandia et al. (2007) bemerker også at det er vanskelig å skalere ut og partisjonere data jevnt utover lagringsnodene i distribuerte, relasjonelle databasesystemer, på tross av at de mest avanserte relasjonelle DBMSene støtter noen former for klyngeoperasjoner.

Unødvendige DBMS-funksjoner Tjenestearkitekturen til Amazon har behov for en svært begrenset mengde funksjoner fra dets datalager, og sjelden behøves det at databasen skal kunne utføre avanserte oppgaver som triggere og lagrede prosedyrer, eller kjøre komplekse spørringer aggregeringsoperasjon som summering og gjennomsnitt på enkeltverdier på enkeltattributter over flere dataobjekter. Flesteparten av de enkeltstående tjenestene som eksempelvis handterer bestselgerlister, handlevogner, kundepreferanser, salgsstatistikk, og innloggingsdata, både spør etter og persisterer dataobjekter utelukkende på enkeltnøkler. Altså er de korresponderende abstrakte relasjonsmodellene for hver av disse uavhengige tjenestene assossiasjonsløse, det vil si at de kan modelleres som én enkeltstående, kompleks entitet, med nøsting av objekter og lister.

Følgelig går Dynamo inn for en enkel spørringsmodell: Ethvert dataobjekt identifiseres med en unik (hash)nøkkel. Alle skrive - og leseoperasjoner i lagringssystemet gjøres gjennom oppslag på en slik ID. Samtlige spørringer gjøres på ett enkelt dataobjekt ad gangen slik at det ikke er behov for å innføre støtte assosiasjoner mellom ”tuplene” i datamodellen.

Dynamo sitt systemdesign ble primært laget for å kunne tekkes Amazons storskala - produksjonsmiljø med flere titalls millioner samtidig påloggede forbrukere. Det distribuerte lagringssystemet som realiserer Dynamos arkitektur er designet både for tilgjengelighet og skalerbarhet. Førstnevnte kvalitetsattributt realiseres gjennom planlegging for feilsituasjoner, og motvirkning av effektene diverse typer feil kan ha. Feil kan til eksempel oppstå i nodens maskinvare eller dens kjørende programvareprosess. Sistnevnte oppfylles ved å implementere en lastbalanseringsalgoritme som fordeler ansvaret for lagring av de enkelte dataobjektene utover nodene i lagringssystemet. Hvis én av lagringsnodene svikter og blir utilgjengelig, kan algoritmen omfordele data til de andre nodene i live.

Utviklerne bak Dynamo valgte av hensyn til tilgjengelighetskravene å gjøre replikering av dataobjekter asynkront, derav kan ikke det samme skrivekonsistensnivå som vi finner hos relasjonelle databasesystem oppfylles. Årsaken til dette tilfellet er at datalagret verken kan eller vil kontrollere hvorvidt hver enkelt spørring mottar eller opererer på dataobjektets nyeste utgave, altså tillater databasen lesing av foreldede data. Hva angår ACID - egenskapene fokuserer Dynamo på å opprettholde atomisiteten til spørreoperasjonene, på grunn av kravene DeCandia et al. (2007) stiller til spørringsmodellen. Altså lagrer Dynamo aggregatobjekter som innehar hele den persisterte tilstanden som applikasjonene til Amazon opererer på i primærminne. Av hensyn til skrive-tilgjengeligheten isoleres

ikke spørreoperasjonene, hvilket betyr at systemet tillater at skriveoperasjoner kan bli tapt, jamfør Last Write Wins - prinsippet. Av økonomiske hensyn er det også viktig at Dynamo, det distribuerte nøkkelverdilageret, kan kjøre på en infrastruktur bestående av billige datamaskiner, hvilket betyr få prosessorkjerner og begrenset datavolum både i primærminnet (RAM) og sekundærminne (platelager).

Artikkelens vitenskaplige bidrag er en vurdering av hvordan ulike algoritmer og teknikker kan kombineres til å implementere et høytilgjengelig nøkkelverdilager. Den viser at en database som ikke garanterer et strengt konsistensnivå trygt kan brukes i et produksjonsmiljø der frekvensen av innkommende spørringer er høy. Artikkelen viser også hvordan et slikt nøkkelverdilager kan konfigureres til å oppfylle strenge ytelseskrav i høytraffikerte produksjonsmiljø (DeCandia et al., 2007).

2.3.2 Om Dynamos arkitektur og Amazons implementasjon

Designet til et distribuert nøkkelverdi-lager som opererer i et stort produksjonsmiljø slik som Dynamo må nødvendigvis være ganske komplekst, fordi det er mange problemer som må løses av dette designet som en monolittisk databasearkitektur ikke har. For å tekkes strenge krav til tilgjengelighet, må systemet ha gode løsninger til lastbalansering av data, datareplikering, dataobjektversjonering, og feilhandtering.

Dynamos logiske ring av noder som lagrer data er et likemannsnettverk. Et likemannsnettverk er en distribuert programvarearkitektur der alle noder utfører de samme arbeidsoppgavene. I et likemannsnettverk har hver enkelt node kjennskap til et visst antall andre noder i nettverket. I litteraturen kalles disse kjente nodene for ”naboer” (*sitat*). En node holder rede på sine naboer i en oppslagsliste kalt ”routingtabell” (sitat). Hvis noden mottar en forespørsel etter et objekt den ikke har, slår den opp i routingtabellen for å finne en annen node å videresende forespørselen til.

I tråd med artikkelens observasjon vedrørende Amazon-applikasjonenes begrensede behov for spørrefunksjonalitet i databasen de kontakter, har Dynamo-implementasjonen beskrevet av DeCandia et al. (2007) definert et spinkelt spørre - API bestående av to funksjoner: `get(key)` og `put(key, context, object)`. `get` - funksjonen er for lesespørringer. Hver lesespørring er et oppslag på en nøkkel, *key*, som er tilknyttet et dataobjekt på binær form. Når `get` kalles, vil databasenoden som mottok spørringen, i artikkelen referert til som koordinatoren, først identifisere nodene som lagrer replikaene av det ønskede dataobjektet og kontakter alle nodene som holder på disse replikerte objektene. Det aggregatet med den nyeste versjonen returneres til applikasjonsklienten. Spørringen kan også returnere en liste av objekter hvis dataversjoner divergerer, i tillegg til tilhørende metadata - da må de divergerende aggregatene flettes sammen, og en ny versjon må deretter persisteres. Amazons implementasjon av Dynamo anskuer både nøkkelen og dataobjektet som en ugjennomsiktig streng av biter.

DeCandia et al. (2007) sin implementasjon av Dynamo bruker Last-Write-Wins-strategien som konfliktresolusjonsalgoritme, hvilket er en lettvint løsning: Ved fletting lagrer databasenoden simpelthen den oppdateringen på det vedkommende dataobjekt som har det seneste tidsstemplet eller har høyest vektor-klokkeverdi. Slik vil en skriveoperasjon gå

tapt for programvaresystemet Dynamo lagrer data for. Per nøkkelverdidatamodellen sin natur er det urimelig å forvente at det distribuerte datalageret har noen form for semantisk kjennskap til dataobjektene den tar hand om, da de for databasenodene bare er tilfeldige samlinger av binære tall. Derfor kan applikasjonsutviklere implementere sine egne konfliktresolusjonsalgoritmer som påkalles hvis samtidige skriveoperasjoner gjør at versjonshistorikken til to replikaer av et aggregat divergerer. For Amazon sine hovedsaklige bruksområder for Dynamo er ikke tapte skrivinger et problem, med unntak av handlevognsapplikasjonen kundene bruker i nettbutikken. Loggdata er ikke like kritisk som handledata, så der kan LWW-fletting trygt brukes.

`put` - funksjonen brukes både til å opprette og oppdatere lagrede dataobjekter assosiert og slått opp på nøkler. Når `put` kalles, blir først nodene som er ansvarlige for dataobjektet identifisert og kontaktet på samme måte som i `get`. I tillegg til aggregatets nøkkel, *key*, har `put` to andre argumenter, *context* og *object*. *context* - variabelen representerer metadata om dataobjektet som lagres, deriblant verdiene i dets vektorklokke. Informasjonen i *context* - variabelen brukes også til å sammenliknes med tilsvarende lagret metadata i databasen for å validere dataobjektet som sendes inn i som argument i `put` - kallet.

For å identifisere hvilken node i likemannsnettverket som skal ha ansvar for oppslaget på en nøkkel *k*, blir den hashet med MD5-algoritmen som returnerer en 128-bit streng *h* hvis verdi faller imellom to andre IDer som hver identifiserer én separat databasenode i hashringen. Den databasenoden hvis ID er nærmest, men lavere enn *h* i binærtallsystemet, er den noden som får ansvar for å levere dataobjektet med nøkkel *k* ved spørringer etter denne nøkkelen.

For konsistenskontroll bruker Dynamo en egen protokoll inspirert av quorumreplikering [7] for å replikere dataobjekter. Quorum-replikering er en datareplikeringstrategi for distribuerte datalagre som er justerbar. Et quorum er en mengde noder som tilordnes ansvaret for en spørring, til vanlig av en valgt koordinator. Bailis et al. (2014) viser gjennom simulasjoner med en probalistisk sannsynlighetsmodell og evaluering av loggdata innsamlet i store, kommersielle produksjonsmiljø at quorumreplikerte databasesystemer lar databaseadministratorer i prakis stille på konsistensnivået til samtidige spørringer ved å konfigure en tuppel bestående av følgende tre tall som indikerer antallet replikeringsoperasjoner for hver innkommende forespørsel:

- R - quorumstørrelse for leseoperasjon
- W - quorumstørrelse for skriveoperasjon
- N - Antall replikerte dataelementer for én enkelt nøkkel

I kontekst av Dynamo er dens replikeringsalgoritme ekvivalent med et strikt quorumsystem hvis R, W og N er definert slik at begge av følgende betingselser er oppfylt:

- $W > N/2$, altså at spørringskoordinatoren avventer bekreftelser på at over halvparten av de N replikaene fullbyrdet spørringen
- $R+W > N$, altså at mengden replikaer av et aggregat spørringskoordinatoren venter på før ett av disse returneres til databaseklienten overlapper med mengden be-

[7] Begrepet *quorum* er latin og betyr "beslutningsdyktig antall"

kreftelsesmeldinger fra en skrivespørring spørringskoordinatoren venter på før den returner en bekreftelsesmelding til klienten

I et strikt quromsystem oppfylles sterk replikakonsistens, enhver lesespørring på et dataobjekt får med seg dens siste oppdatering/PUT som er skrevet til disk nettopp fordi minst én node er med både i dataobjektets skrivequorum og dets lesequorum. Det må påpekes at strikte quorumsystem venter på at samtlige noder i ethvert quorum returner svar på forespørselen fra spørringskoordinatoren, det vil si at strikte quorumsystem øker nettverksforsinkelsen til hver enkelt spørring for å sikre sterk replikakonsistens, fordi koordiantoren må nødvendigvis vente på den noden i quorumet der spørringen har høyest forsinkelse i nettverket (her antas det at nettverks-I/O er langt mer tidkrevende enn I/O-operasjoner på et platelager lokalt).

Quorum-replikeringsstrategien til Dynamo avviker et ordinært quorumsystem på to fronter. For det første: Hvis Dynamo hadde brukt et strikt quorumsystem for datareplikering, ville enkelte dataobjekter ha blitt utilgjengelige for databaseklientene, hvis nettverkspartisjoner i ringen oppstår, eller hvis de enkleste feilscenarier, der bare én enkelt node rammes, inntreffer. I tillegg ville skrivespørringer være langt mindre holdbare (eng. "durable") på disk (DeCandia et al., 2007). For det andre er Dynamos quromprotokoll "sløv", det vil si at en spørringskoordinator oppnevner ikke en bestemt mengde noder som alle **må** fullføre spørringen for at resultat kan sendes tilbake til databaseklienten som kjørte databasespørringen. Fordelen med denne sløvheten, eller "fleksibiliten", er at risikoen minimaliseres for at en spørring ikke blir besvart på grunn av at én enkelt node er utilgjengelig. For en høytilgjengelig nettbutikk som Amazon.com er nettverksforsinkelse et mye større problem enn synkronisering av replikaer, derfor konfigureres quorumet gjerne slik at $R + W < N$, for eksempel $R = 1, W = 1, N = 3$.

For øvrig kan Dynamo sine kjerneegenskaper oppramses som følger:

Konsistent hashing Lastbalanseringsalgoritmen Dynamo bruker for å partisjonere data utover ringen av databasenoder, som er formen til den strukturen til likemannsnettverket, av databasenoder. Både data og nodeidentifikatorer hashes inn i ringen. En node, det vil si en tjenerprosess i ringen, lagrer eller holder rede på dataobjekter som har blitt hashet til forgjengernoden i ringen, således realiseres lastbalansering i et likemannsnettverk bestående av mange tjenere på en ryddig måte. Den konsistente hashalgoritmen oppfyller kravet om inkrementell skalerbarhet.

Vektorklokker Versjoneringsteknikk brukt til å holde styr på oppdateringshistorikken til et dataobjekt. Hver databasenode som oppdaterer et bestemt aggregat, skriver et stykke metadata, en vektor på formen (`nodeNo`, `seqNo`), der førstnevnte representerer den enkelte noden som skriver dataobjektet, mens den andre variabelen er et versjonsnummer (eventuelt et tidsstempel) som øker monotont for hver gang objektet skrives til disk. Ved å lese av denne vektoren kan man utlede endringshistorikken til hvert enkelt dataobjekt og se kausaliteten mellom forskjellige versjoner av det, altså hvilke skrivinger som er fundert på tidligere skrivinger. Hvis to samtidige oppdateringer medfører to divergerende versjoner, kan versjonshistorikken brukes til å identifisere "synderne" som utførte de uavhengige oppdateringene. Databaseklienten kan deretter finne ut hvilke dataverdier innad i de divergerende objektene som

skal beholdes under fletteprosessen.

Slurvete quorum (eng. Sloppy quorum) Lese - og skriveoperasjoner utføres på de første N ”friske” databasenodene ifølge en preferanseliste konfigurert globalt i systemet. Noder som ikke responderer på forespørsler, dvs ”syke” noder, hoppes over. Denne midlertidige økningen av medlemmer i quorumet kalles også for antientropi.

Antydet avlevering (eng. Hinted handoff) - Teknikk for hurtig handtering av innkommende skrivinger adressert til en utilgjengelig node. Når den slurvete quorumalgoritmen finner at én av adressentene i skrivequorumet er (midlertidig) utilgjengelig, så blir skrivequorumet midlertidig utvidet med en ekstra ”opp-passernode”. Denne midlertidige vakten lagrer dataobjektet helt til den frafalne noden er tilbake i ringen. Når opp-passernoden mottar nyheter (via sladderprotokollen) om at noden som dataene opprinnelig tilhører lever igjen, blir disse avlevert til sitt opprinnelige tiltenkte lager fra opp-passernode.

Merkle - trær Et Merkle-tre er et hashtre der hver enkelt løvnode holder på en hashet verdi. Foreldrenodens tilstandsverdi tilsvarer den kumulative hashverdien av dens barn. Rekursivt sett vil dette si at rotnoden holder på den sammenlagte verdien for alle løvnodene i treet. Hvert enkelt nivå i treet er følgelig sammenliknbart med korresponderende nivå i liknende tre.

Antientropi Mens slurvete quroumoperasjoner og antydede avleveringer kan brukes til å respondere på midlertidige fravær av noder, som kan forårsakes av at et unntak kastes i selve programvareprosessen til noden, benyttes Merkle - trær til å rebalansere datalasten i likemannsnettverket etter at en databasenode legges til eller blir permanent utilgjengelig. Merkle-trær er således en viktig datastruktur for å realisere antientropi-protokollen til Dynamo. Merkle-trær brukes til å synkronisere utdaterte replikaer uten å sende én enkelt melding per oppdaterte replika av hvert eneste lagrede dataobjekt. Istedet kan hver enkelt par av noder utveksle en hash-verdi som representerer den samlede hashverdien for deres respektive nøkkelrom. Når en node legges til eller forsvinner fra ringen, må nøkkelintervallene som nodene er ansvarlige for å lagre, omfordeles for å fordele datalasten jevnt. Hvis ringen har høy ”churn” - rate, det vil si at noder svært ofte kommer og går ut av ringen eller at fatale eksekveringsfeil oppstår abnormalt ofte hos nodene, vil nøkkelintervallene bli rekalkulert svært ofte, noe som hemmer ytelsen til systemet.

Sladder-basert medlemskapsprotokoll (eng. ”Gossip-based membership protocol”) // ”Gosipping”, eller sladder på godt norsk, er en masterløs, skalerbar medlemskapsprotokoll der hver enkelt node jevnlig kontakter en tilfeldig valgt nabo (”peer”) og utveksler og oppdaterer egne registrerte medlemsdata om ringen. DeCandia et al. (2007) registrerer også at denne ”explicit node join” - prosessen gjør at noder blir oppmerksomme både på nye noder og nylig utilgjengelige noder, som gjør en separat distribuert feildetektor overflødig. På grunn av protokollens asynkrone natur (hver enkelt medlemskapsendring i ringen medfører ikke en direkte oppdatering hos alle nodene samtidig utstedt fra en sentral medlemskoordinator) er medlemslistene svakt konsistente. Hensikten med en slik medlemsprotokoll er både å holde styr på hvilke databasenoder som fortsatt er del av Dynamo-ringen og å detektere at en node

er utilgjengelig, dvs at den er utsatt for feil, slik at systemet kan midlertidig persistere replikaer offeret for feilen hadde styr på hos noen andre noder med antydet avlevering.

Lesereparasjon (eng. "Read repair") I Amazons implementasjon venter den enkelte databasenode på respons fra spørringskoordinatoren, etter å ha returnert et dataobjekt til den. Hvis spørringskoordinatoren mottok replikaer som ikke er av den nyeste versjonen, kan den sende den nyeste versjonen til de respektive noder som returnerte de utdaterte dataobjektene. Prosessen bærer navnet "lesereparasjon" fordi replikaer som ikke har lagret den nyeste dataversjonen, blir reparert på opportunistisk vis av spørringskoordinatoren. Denne prosessen kjøres ad-hoc og er ikke effektiv når det kommer til å synkronisere replikaer ved endringer i medlemsmassen til Dynamoringen.

Denne teknologiske syntesen, tildels inspirert av distribuerte hashtabeller og quorumsystemer, medfører i sum et høytilgjengelig, feilrobust, lastbalansert nøkkelverdi-lager. De kvalitative lagringsegenskapene til lageret kan finjusteres ved å konfigurere quorumvektoren R, W, N. Mange av disse teknikkene er, som vi skal se i neste delkapittel, også realisert i Project Voldemort.

2.4 Project Voldemort

Project Voldemort er et distribuert nøkkelverdilager inspirert av Amazons nøkkelverdilager Dynamo, hvis arkitektur presenteres av DeCandia et al. (2007). Dette systemet replikerer og partisjonerer sine data automatisk. Forfatterne og vedlikeholderne [8] av kildekoden refererer til Voldemort som en stor distribuert, persistent, feiltolerant hash-tabell (Kreps, 2009). Hver enkelt node i det kjørende databasesystemet holder en delmengde av den totale datamengden som handteres. Flere komponenter i dette systemet, deriblant databasemotoren, plasseringsstrategien for data-tupler, og serialiseringsmetoden, er valgfrie for den enkelte programmerer. Blant annet kan man bruke lagringskomponenter som InnoDB, RocksDB (som benytter LSM-trær), Berkeley DB, eventuelt kan man lagre tupler i primærminnet. I tillegg er også nivået på konsistens av skriveoperasjoner justerbart. Prosessperspektivet til arkitekturen er master-fri, det vil si at i det distribuerte datalageret holdes det ikke valg av spørringskoordinator blant nodene. Derfor opererer de som et likemannsnettverk. Feil som oppstår ved spørringseksekvering behandles transparent.

2.4.1 Støttede operasjoner

Nøkkelverdilagre tilbyr tradisjonelt sett et svært enkelt spørregrensesnitt til applikasjoner som bygger sine datamodeller med dem. Og Voldemort er overhodet ikke annerledes i den forstand. Applikasjonprogrammerings-grensesnittet til Voldemort definerer hovedsakelig tre funksjonelle endepunkter: `get` (leseoperasjoner), `put` (skriveoperasjoner, både

[8] Kildekoden til Project Voldemort er lisensiert under Apache 2.0 - lisensen, og er tilgjengelig på følgende GitHub-repositorium: `http://github.com/voldemort/voldemort/tree/master`

til opprettelse og oppdatering av tupler), og `delete` (sletteoperasjoner). I tillegg til disse tre operasjonene som er karakteriske for de fleste nøkkelverdilagre definerer Voldemorts klient – API endepunktene `getAll`, som er en leseoperasjon på multiple nøkler, og versjonerte utgaver av `get` og `put` som til forskjell fra de ordinære funksjonene med samme navn tar inn et versjonsobjekt som input til funksjonen i tillegg til en nøkkel og en verdi, slik at tjeneren slipper å gjøre et oppslag for å finne den nyeste versjonen til det angitte objektet først. Hvis versjonen av objektet hvis nøkkel er spesifisert i funksjonskallet ikke er den nyeste vil klientprogrammet kaste et unntak, kalt `ObsoleteVersionException`, og brukeren får ingenting returnert. Som tidligere nevnt i den generelle diskusjonen om nøkkelverdilagre anser en databasenode hos Voldemort både nøkkel og det assosierte dataobjekt som en tilfeldig sekvens of bytes.

2.4.2 Voldemorts egenskaper sammenliknet med RDBMS

I forhold til relasjonelle databasehåndteringssystemer er Voldemort vesentlig bedre egnet til å innføre replikering av data, da ytelsen til både leseoperasjoner og skriveoperasjoner lar seg skalere horisontalt. Det vil si at hvis en ny node legges til det distribuerte miljøet, også benevnt i litteraturen som "databaseinstansen" (Sadalage and Fowler, 2013), så vil det ha ingen eller neglisjerbar påvirkning på ytelsen.

Et relasjonelt DBMS har den fordel over Voldemort at dets innebygde samtidighetshåndtering, som utføres ved hjelp av transaksjonsmønsteret, er strengere og derfor mer pålitelig enn Voldemort. Voldemort fokuserer heller på å holde orden på endringshistorikken til hver enkelt tuppel som lagres på den enkelte node i det distribuerte lageret. Ved hjelp av versjonering kan kopier, eller replikaer, av tupler hvis dataverdier er divergerende, rettes opp gjennom en flettingsprosess.

Moderne webapplikasjonssystemer består gjerne av forskjellige, adskilte tjenester eller applikasjonsgrensesnitt, hver av disse kan i sin tur distribuere egne data over opptil flere datasentre rundt om i verden. Slike webapplikasjoner kan ikke ta seg tid til å vente på JOIN - operasjoner mellom to entiteter eller tabeller som potensielt ligger på hver sin MySQL - tjener på hver sin datamaskin i hvert sitt datasenter på to vidt forskjellige lokasjoner på kloden. Én framgangsmåte på å skalere en relasjonell datamodell til å møte behovene til flere tusen forespørsler samtidig, er å introdusere et hurtiglager-nivå i systemarkitekturen ved hjelp av et distribuert, minnebasert cachesystem som MemCache eller Redis. Dette hurtiglageret avlaster databasen for leseoperasjoner, som vil utgjøre en flaskehals for den samlede ytelsen til den distribuerte applikasjonen. Dessverre vil ikke denne løsningen skalere for skriveoperasjoner og JOIN - operasjoner så lenge logisk konsistens for skrivinger er et krav. Voldemorts løsning, hvis tekniske detaljer vil bli diskutert i neste delkapittel, er å lempe på disse konsistenskravene.

Relasjonelle databasesystemer realiserer de assosiasjoner som er spesifisert mellom entitetene i datamodellen til applikasjonens arkitektur. Relasjonelle databasesystemer oppfyller samtidig fire egenskaper for skriveoperasjoner som gjøres i systemet gjennom transaksjonsmønsteret. Atomisitet, en garanti på at hver enkelt transaksjon enten utføres fullt og helt eller avbrytes; Konsistens: for hver enkelt utførte transaksjon etterlates databasen i en

konsistent tilstand; Isolasjon: Hver enkelt transaksjon kjøres uavhengig av hvilke andre transaksjoner som kjøres samtidig; Holdbarhet: Data som er persistert gjennom transaksjoner i systemet, vil ikke endres eller forsvinne med mindre påfølgende transaksjoner gjør det. Hver enkelt PUT-operasjon i Voldemort er atomisk så lenge den opererer på et og samme aggregat i datamodellen, det vil si den komplekse objektverdien som aksesseres med nøkkelen.

Transaksjoner gjør skriveoperasjoner i relasjonelle databaser lineariserbare, som er det strengeste konsistensnivået skriveoperasjoner kan ligge på i databasesystem, ved å tvinge skriveoperasjonen til å vente hvis den prøver å endre rader eller tabeller som allerede er reservert for en tidligere påbegynt transaksjon. Voldemort, på sin side legger seg på et mildere konsistensnivå, kalt svak konsistens (eng. ”eventual consistency”).

2.4.3 Konsistenskontroll og versjonering

I likhet med Amazon Dynamo er også Voldemort sin replikakonsistenskontroll inspirert av quorumsystemer. Quorumreplikering og Dynamos liknende replikeringsstrategi av det beskrives i 2.3.2.

På samme vis som i Dynamo tillater Voldemort samtidige skriveoperasjoner på samme nøkkel på forskjellige noder. Begrunnelsen for dette valget er at Dynamo er alltid mottakelig for skriveoperasjoner (DeCandia et al., 2007), slik at for eksempel en handlekurv alltid kan oppdateres i sanntid. Følgelig kan to eller flere replikaer for samme nøkkel være lagret på forskjellige noder, uten å være synkroniserte. Det vil si at ingen av de to replikaene har den fulle og hele oppdateringshistorikken til nøkkelen. I stedet for å preservere replikakonsistens i systemet for alle nøkler til enhver tid, går både Dynamo og Voldemort for å forsone divergerende replikaer for en og samme nøkkel når den blir `get`-et av applikasjonen, en prosess kalt ”read repair” av DeCandia et al. (2007).

Hvert dataobjekt som skrives tillegges en vektorklokkeverdi som del av dets metadata. Ved en leseforespørsel på samme nøkkel kan koordinatoren for denne leseforespørselen motta replikaer med divergerende vektorklokkeverdier, også kalt *versjoner* (Kreps, 2009). Hvis systemet er i stand til å forsone og flette oppdateringshistorikken til nøkkelen basert på vektorklokkeverdiene til replikaene, så vil spørringskoordinatoren returnere én gjeldende dataverdi tilbake til klienten som sendte leseforespørselen. Samtidig vil koordinatoren sende skrivespørringer til hver av replikanodene til nøkkelen slik at de kan oppdatere sitt replikaobjekt med den flettede verdien.

Hvis en fletteoperasjon ikke er mulig å utføre med bare vektorklokker, vil spørrekoordinatoren returnere alle de konflikterende versjonene av aggregatet til applikasjonsklienten, slik at den selv kan flette den sammen. Applikasjonen vil deretter persistere den definitive, sammenflettede versjonen av aggregatet tilbake til datalageret.

2.4.4 Serialisering av dataobjekter

Innen datateknologi er serialisering prosessen der objekter eller datastrukturer i datamaskinminne som holder på applikasjonsdata, konverteres til et format som lar seg lagres på disk, eventuelt forsendes over et nettverk. I en datamaskins interne minne kan dataobjekter som aksesseres av en og samme programvareprosess, ligge på vidt forskjellige minneadresser. Hvis disse dataobjektene skal forsendes over et I/O - grensesnitt, er det viktig å samle dem sammen og ordne dem på et vis som kan leses av en datamaskin, derav begrepet serialisering. Serialisering går som regel ut på å flatpakke en nøstet datastruktur, for eksempel et tre, et objekt, eller en matrise, til en enkelt-dimensjonal sekvens av binære sifre. Den konverterte strengen av bits blir da enten persistert til disk eller sendt til en helt annen datamaskin over et IP - nettverk. Ved mottak eller lesing fra disk, blir bit-strengen konvertert tilbake til den opprinnelige datastrukturen. Serialisering kan også brukes i eksterne prosedyrekall (RPC).

Serialisering av minnebaserte datastrukturer er en ganske vanlig oppgave i distribuerte systemer og applikasjonsprotokoller, og derfor kan dette gjøres i mange programmeringsspråk. I Java blir klasser som implementerer grensesnittet java.io.Serializable automatisk serialisert. I Python brukes modulen pickle fra standardbiblioteket til samme formål. I PHP nyttes funksjonen `serialize()` til serialisering og `unserialize()` til parsing. I JavaScript er JSON - objektet, med dets to metoder `parse()` og `stringify()` innebygd i språket. JSON - protokollen er nemlig basert på et subsett av JavaScript. Følgelig kan en hvilken som helst datastruktur formet i JSON - syntaksen oversettes til JavaScript - syntaks.

Hos Voldemort serialiseres dataobjekter før de lagres. Ved å serialisere data kan man garantere at datalageret er helt og holdent uvitende om hva det er sekvensene av 1-ere og 0-ere som blir lagret står for. Applikasjonsutviklere som benytter Voldemort som sin database, kan bruke en av mange forskjellige teknikker, eventuelt implementere sin egen. Serialisering sies å være -pluggbart- i Voldemort, slik at man står fritt til å bruke serialiseringsrammeverk som Apache Avro, Googles ProtoBuf, Apache Thrift og Javas serialiseringsgrensesnitt som tidligere nevnt (Kreps, 2009). Ved hjelp av serialisering kan komplekse datastrukturer, som lister og navngitte tupler, benyttes både som nøkler og verdi for de enkelte rader. Et siste konfigurasjonsalternativ som kan nevnes er ”string”-serialisering. Dette alternativet benyttes hvis databaseklienten mottar preformatert data på strengform, og er dermed fullstendig skjemaignorant.

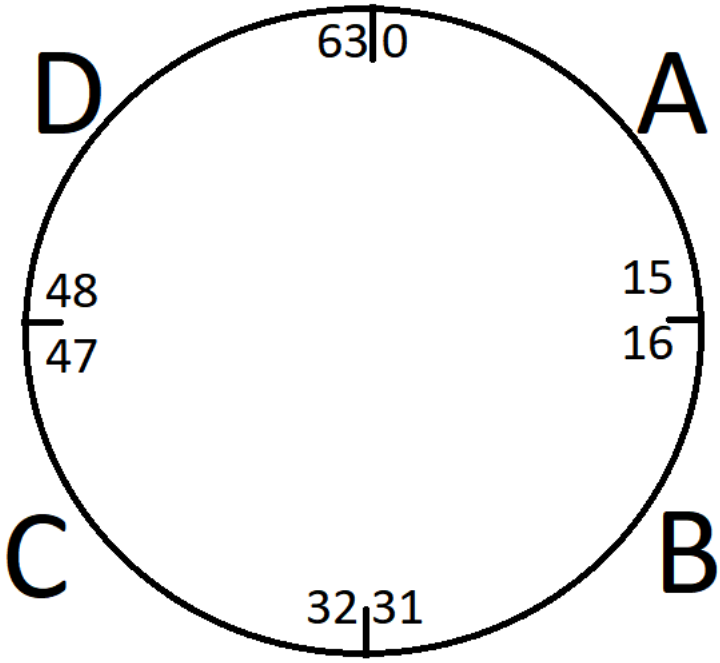
2.4.5 Lastbalansering av dataobjekter

I et klynget databasesystem er det viktig å fordele datareplikaer, slik at hver node i klyngen holder på omtrent like stort datavolum til enhver tid. For skalerbarheens skyld kan ikke én eneste node holde på samtlige dataobjekter som lages i databasen, ellers vil datautilgjengeligheten være inversproporsjonal med antallet tjenere i klyngen.

Konsistent hashing - teknikken brukes også i Voldemorts lastbalanseringsalgoritme. I likhet med Amazon Dynamo er konsistent hashing en viktig algoritme som brukes for å

oppnå jevn fordeling av nøkler. I tillegg er konsistent hashing involvert i både replikering og rebalansering av datalasten, noe en enkel hashfunksjon ikke kan gjøre. En vanlig hashalgoritme hasher data eller nøkler inn i binært rom bestående av et fast, forhåndsinnstilt antall *partisjoner*, eller bøtter. Hver av disse partisjonene handterer et forhåndsinnstilt del av det binære verdiområdet (et tall fra 0 til en stor toerpotens minus 1) til hashfunksjonen.

Når en node svikter permanent, vil datalasten bli omdistribuert til de gjenværende nodene i ringen, hvis partisjoner grenser mot de partisjonene av ringen den feilede noden hadde ansvar for, altså omadresseres aggregater tidligere lagret av en nå feilet og permanent utilgjengelig node. Noder med høyere lagringskapasitet lagrer flere datatupler fordi de har ansvar for en større brøkdel av ringen av diskrete hashverdier.



Figur 2.4: Eksempel på konsistent hashring i Project Voldemort.

Den logiske strukturen til likemannsnettverket av databasenoder i Voldemort har formen til en ring. Denne ringen består av en stor mengde diskrete posisjoner, verdiområdet til en hashfunksjon *h*. Hver node som deltar i databaseklyngen, tildeles en gruppe av disse punktene i ringen. *h* brukes til å finne ringposisjonen til en nøkkel *k*. Den noden, hvis tilordnede mengde av hashverdier inneholder *h(k)*, blir den som lagrer det assosierte dataobjektet til *k*.

Den konsistente hashalgoritmen og ringstrukturen til likemannsnettverket realiserer datareplikering ved å legge antallet påkrevde replikaer, per quorumkonfigurasjonen, på etter-

følgende noder i ringen, med sola (Elmasri, 2016). Per figur 2.4 og replikakonfigurasjonen $N = 3$ vil eksempelvis alle dataobjekter som hashes til node A, også kopieres til node B og C. Av denne figuren kan man lese at node A lagrer nøkler med hashverdi 0-15, B lagrer nøkler hvis hashverdi er mellom 16 og 31, C lagrer nøkler hvis hashverdi er mellom 32 og 47, og D lagrer nøkler hvis hashverdi er mellom 48 og 63.

Kapittel 3

Relatert arbeid

I dette kapitlet presenteres eksisterende moduler og programvare som kan brukes av programvareutviklere til å migrere data fra én datamodell til en annen under appliaksjonsoppgradering. Kapitlet vil begynne med å beskrive tilgjengelighet som kvalitetsattributt i moderne webapplikasjoner. Det vil også skissere viktigheten av levende eller rullerende oppgradering i lys av kontinuerlig leveranse. I tre påfølgende delkapitler beskrives tre forskjellige oppgraderingsrammeverk: Utgavebasert oppgradering av datamodeller, kalt EBR; KVolve, en datamigrasjonsmodul integrert i nøkkelverdisystemet Redis; Imago, et atomisk rammeverk for helhetlig webappliaksjonsoppgradering. Til sist beskrives en datamigrasjonsmodul for Apache Cassandra, som er implementert i programmeringsspråket Java, kalt **cassandra-migration-tool-java**.

3.1 Høytilgjengelige webapplikasjoner

Tilgjengelighet er i mange moderne web 2.0 - applikasjoner en kritisk kvalitet som det stilles svært strenge krav til, især systemer som brukes av flere hundre tusen mennesker verden over. Slike systemer oppfyller strenge krav til tilgjengelighet, og refereres til i litteraturen som høytilgjengelige systemer (eng. ”high-availability”). For å tilfredsstille høye tilgjengelighetskrav, må man forstå hvilke typer feil som oppstår, hvorfor de oppstår, og i hvilke omstendigheter de oppstår. Når en bestemt prosentandel i en tjenestenivåavtale refereres til som et kvantitativt mål på systemets lovede tilgjengelighet, er den ofte basert på formelen for steady-state-tilgjengelighet Dernest kan man implementere strategier for å redusere feilenes påvirkning på programvaren. Høytilgjengelige systemer lover som regel et pålitelighetsnivå på 99.9 %, det vil si inntil 8 timer nedetid om året.

3.1.1 Tilgjengelighet

Et systems tilgjengelighet er kvalitetsattributtet som beskriver hvordan systemet oppfører seg i et abnormalt operasjonsmiljø, for eksempel ved et feilscenario eller et kontrollert avbrudd. Tilgjengelighet bygger på et annet kvalitetsattributt, pålitelighet, ved å legge til reparasjonsaspektet - når et system utsettes for en feil, så vil systemet respondere på hendelsen slik at feilen ikke forårsaker nedetid (Bass et al., 2013). Kvaliteten beskriver altså mer enn bare hvor ofte systemet er tilgjengelig for å behandle tjenesteforespørsler fra dets brukere.

Bass et al. (2013) definerer et programvaresystem sin tilgjengelighet som dets evne til å maskere eller reparere feil, slik at dets sum av alle perioder med nedetid innenfor et definert tidsintervall i forhold til nevnte intervall, ikke overstiger en bestemt prosentandel. Med ”nedetid” menes en konkret, lukket tidsperiode der systemet ikke er mottakelig for kommando fra dets påtenkte brukere.

Mer presist beskrevet er steady-state-tilgjengelihet, en indikator på oppetiden til en komponent av et system eller systemet sett under ett, bestemt av to estimerte forventningsverdier: Tiden det tar før en svikt oppstår (MTBF), og reparasjonstiden (MTTR). Steady-state-tilgjengelihet er et forholdstall uttrykt matematisk som $\frac{MTBF}{MTTR+MTBF}$.

I kontekst av programvareutvikling bør denne formelen tolkes dithen at for å diskutere systemets tilgjengelighet, må sannsynlige scenarier der feil oppstår og hvordan håndtere dem identifiseres. Effektene av hver feil må estimeres. Ikke minst må man tenke hvor lang tid man må bruke på å reparere eller maskere den enkelte feil. I tillegg kan feil unngås og fjernes, eventuelt kan man velge ikke å gjøre noe som helst i det de oppstår (Bass et al., 2013).

For å øke et systems tilgjengelighet må man minimalisere effekten som forårsakes av feil. En feil (eng. ”fault”) er et fenomen som forårsaker svikt i systemet (eng- ”failure”), skriver Bass et al. (2013). En svikt er et avvik i systemets oppførsel fra dets spesifikasjoner. Implisitt i denne forklaringen av begrepet, ligger at systemsvikt er synlig for en menneskelig observatør, til eksempel en autorisert bruker. Eksistensen til en systemsvikt er altså ikke definert før den observeres.

Til vanlig telles ikke planlagt nedetid, som oppstår i forbindelse med vedlikeholdsoperasjoner som programvareoppgradering eller generell minnerensing ved restart av tjenere, i beregningen av tilgjengelighetsprosenten som typisk baseres på den tidligere nevnte formel. Vedlikehold utføres gjerne på et planlagt tidspunkt der forespørselstrafikken er på et svært lavt nivå. Forskningslitteratur dette dokument henviser til, indikerer at planlagt vedlikeholdsarbeid på et kjørende, distribuert system utgjør en egen klasse av mange forskjellige feilkilder som kan medføre nedetid, dog tradisjonelle feiltoleransemekanismer fokuserer på å motvirke, unngå, eller tillate uventede feil (Dumitraş and Narasimhan, 2009).

3.1.2 Versjonskappløp i distribuerte systemer

(Dumitraş et al., 2010) stiller følgende hypotetiske scenario om risiko forbundet med rullerende oppgradering av distribuerte kjøringsmiljøer: I en webbasert nettbankappliaksjon som komunniserer med et banksystem bestående av mange tjenere oppdages en potensielt farlig anomalie sikkerhetsmessig. I den gamle versjonen av klientprogramvaren finnes en tekstboks der brukeren skriver inn informasjon om pengeoverføring, både antall pengeenheter så vel som valutaen enhetene måles. Ergo aksepterer denne tekstboksen alle alfanumeriske tegn, hvilket betyr at nettbanken står i fare for å bli utsatt for SQL-innsprøyting gjennom web-grensesnittet.

For å motvirke denne sikkerhetsrisikoen, beslutter ingeniørne som vedlikeholder programmet at den enkelte bruker skal velge pengevalutaen med en radio-boks istedet for å skrive den inn på tekstform. Dermed vil tekstboksen kun måtte akseptere tall. Imidlertid avdekkes en annen risiko under integrasjonstesting av denne feilreparasjonen: Denne endringen i klientdelen av arkitekturen medfører at tjenerapplikasjonen nå mottar to seperate parametre istedet for den ene strengen fra tekstboksen, og mens tjenerapplikasjonen oppgraderes, vil tjenere som kjører den utdaterte applikasjonen kun lese innholdet fra tekstboksen, som i den nye versjonen klientprogrammet bare er et tall, og følgelig anta på grunn av oppførselskonfigurasjon at beløpet som sendes er oppnevnt i amerikanske dollar. Denne inkonsistensen under oppgradering kan skape betydelig trøbbel for kunder som skal overføre penger i ikke-amerikanske valutaer.

Slike scenarier omtaler (Dumitraş et al., 2010) som versjonskappløpsbetingelser ”mixed version race”. I hans artikkel ”To Upgrade Or Not To Upgrade” demonstreres en sannsynlighetsmodell for risikovurdering av rullerende programvareoppgradering med spesielt hensyn på versjonskappløpsbetingelser, tilstanden systemet har der responsen på en forespurt operasjon er forskjellig mellom to noder som kjører hver sin versjon av nodeprogramvareprosessen.

3.1.3 Dynamisk programvareoppdatering (DSU)

Hicks and Nettles (2005) introduserer et generelt, fleksibelt, og effektivt oppdateringsrammeverk for å oppdatere en kjørende programvareprosess, kalt *Dynamic Software Upgrade*. Artikkelen er en utvidet versjon fra et foredrag Michael Hicks holdt for ACM i 2001.

Én ofte benyttet tilnærming for å innføre online oppgradering, er å installere oppdaterte versjoner av programmet på reserve-tjenere som kan settes ut i produksjon umiddelbart hvis den aktive produksjonstjeneren utsettes for et feilscenario (Hicks and Nettles, 2005).

For eksempel benyttet Visa tidlig på 2000-tallet hele 21 store datamaskiner for å kjøre sitt transaksjonsprosesseringssystem (TPS). Ved oppgradering ble den enkelte instansen sin tilstandsdata bevart mens de enkelte prosessene ble byttet ut. Transaksjonsprosesseringssystemet til Visa opererte per 2001, ifølge Hicks and Nettles (2005), med et oppetidkrav på 99.5 %, men gjennomgikk tusenvis av mindre oppdateringer per år. Denne maskinbundne oppgraderingsteknikken er nødvendigvis meget dyr, og gjør utviklingsprosessen for

de som vedlikeholder programvare ekstra tungvint. Dette grunnet den nødvendige rutinen med å overføre tilstandsdata fra en gammel applikasjonstjener til en ny og nødvendigheten av å kjøpe inn og starte opp en ny maskin hver gang en oppdatering skal installeres (Hicks and Nettles, 2005).

DSU – rammeverket omgår de ekstra oppgraderingskostnadene skissert i avsnittet ovenfor. Hicks and Nettles (2005) beviser at det innehar fire forskjellige kvaliteter: Fleksibilitet (en hvilken som helst komponent i systemet kan oppdateres uten at hele systemet må slåes av); robusthet (systemet minimaliserer risikoen for systemfeil ved oppdatering); brukbarhet (oppgraderingsprosessen er enkel å forstå); ytelse (oppgraderingsprosessen skal ikke utføres på bekostning av systemet tjenesteevne).

Gyldighetsområdet til DSU – Rammeverket er ment å være programvaren på én enkelt maskin, det kan med andre ord ikke brukes på for eksempel distribuerte databaseinstanser tilhørende en webapplikasjon. Endringer i lagringsstrukturen påkrever en koordinert oppdatering av alle applikasjonsinstanser som kommuniserer mot den kjørende databasens grensesnitt.

Systemer som oppdaterer dynamisk bruker litt tid på starte opp en ny prosessnode og å overføre data fra den gamle til den nye, men kan samlet sett redusere oppgraderingstiden vis-a-vis distribuerte systemer som implementerer online-oppgradering ved hjelp av varme reserver, det vil si en annen fysisk maskin som overtar arbeidet fra tjeneren med den gamle applikasjonen.

En dynamisk oppdatering gjennomføres ved å starte den nye overtakende prosessen, fryse tilstanden til det opprinnelige programmet og overføre tilstandsdata over til den nye, blanke prosessen, og til sist fase ut den gamle til fordel for den nye. Det mest sentrale elementet i denne prosessen er en rutine for å sammenlikne semantiske forskjeller mellom de to versjonene og for å finne ut hvordan oppførselen til den nye versjonen er forskjellig fra den gamle.

Ved en dynamisk oppdatering må applikasjonsdata i den gamle, kjørende versjonen overføres til den nye, innkommende versjonen. I denne prosessen defineres en tilstandstransformasjonsfunksjon som kan transponere en tilstandsvariabel fra den gamle versjonen til en tilsvarende verdi for den nye.

Hicks and Nettles (2005) omtaler to forskjellige måter å implementere dynamisk oppdatering, eller ”dynamic patching”, som det også kalles. Den første framgangsmåten er å kompilere den nye versjonen fra den endrede kildekoden, instansiere den nye versjonen i en egen prosess på lik linje med prosessen til den gamle versjonen av programmet, og deretter beordre sistnevnte til å overføre sine tilstandsdata over til prosessen som kjører den nye versjonen. Etterpå bruker den nye versjonen tidligere nevnte transformeringsrutine for å konvertere tilsendte tilstandsdata. Så vil den nye prosessen begynne sin kjøretid med de transformerte dataene fra tidligere.

Den andre måten går ut på å lenke oppdateringen inn i den eksiterende, kjørende prosessen, transformere tilstandsdata der de er (her blir tilstanden kopiert til en annen adresse i prossessens adresserom i minne) og deretter endre koden prosessen kjører fra direkte, ved hjelp av lenken.

3.1.4 Nedetid forårsaket av vedlikeholdsarbeid

En undersøkelse fra 1998 av totalt 426 høytilgjengelige programvaresystemer avdekket at 75 % av totalt 6000 tilfeller av nedetid i profesjonelle, digitale tjenester kom av planlagt vedlikeholdsarbeid på enten maskinvare eller programvare, og disse planlagte nedetidsperiodene varte som regel dobbelt så lenge som de periodene med nedetid som kom av uventet systemsvikt (Lowell et al., 2004; Dumitras and Narasimhan, 2009). Feiltoleransemekanismer lages utelukkende for å kunne forhindre sistnevnte type nedetid. Dette har en sammenheng med at planlagt nedetid tradisjonelt ikke inngår i beregningen av total systemtilgjengelighet per år, fordi systemvedlikehold kunne gjøres på tidspunkt der systemets tjenester ikke etterspørres (for eksempel om natten). Høytilgjengelige systemer påkreves gjerne å være tilgjengelig 24 timer i døgnet, så denne betingelsen er naturligvis ikke lengre gjeldende.

Oppgradering av store informasjonssystemer er også svært dyre, grunnet større endringer i dataskjemaet eller dataformatet og/eller datamigrasjon. Oppgraderinger som innebærer komplekse endringer i datamodellen, medfører at dataene må gjennomgå tunge og langtekkelige konverteringer som kan ta flerfoldige timer for å konformere til nevnte endringer. Disse oppgraderingene er vanskelig å installere mens systemet leverer brukerne sine tjenester, uten å forstyrre tjenesteleveransen såpass mye at systemets ytelse faller ned på uakseptable nivåer. Dette betyr at tunge oppdateringer i datamodellen ofte må gjøres offline, når systemet er slått av. Som en konsekvens av dette, unnlater systemadministratorer i det lengste å endre datamodellen når applikasjonen er i et kjørende produksjonsmiljø (Dumitras and Narasimhan, 2009).

Dumitras and Narasimhan (2009) hevder at manuelt styrte oppgraderingsmetoder, som fordrer mye kunnskap om applikasjonslagets interaksjoner med datamodellen, ikke er veien å gå for å nærme seg idealet om tilnærmet døgnåpen tjenesteytelse. Med sitt helhetlige system for gjennomføring av oppgraderinger med komplekse datamodellendringer, Imago, ønsker Dumitras and Narasimhan (2009) også å unngå behovet for å holde styr på avhengighetene til den kjørende/gamle versjonen. Det er jo tross alt et NP-hardt problem.

Ved å gjennomgå oppgraderingshistorikken til Wikipedia (`http://www.wikipedia.org`) påviser Dumitras and Narasimhan (2009) de viktigste årsakene til at nedetid blir planlagt ved systemoppgradering. Wikipedia er den mest ettertraktede kilden til informasjon på Internett. Leksikonet driftes på wiki-plattformen MediaWiki, implementert i PHP, som aksesserer en distribuert infrastruktur bestående av flere hundre databasetjenere.

Dumitras and Narasimhan (2009) stiller følgende eksempel for en skjemaendring (i en relasjonell database), illustrert i figur 2 på artikkelens tredje side: Kolonne `a` i tabell `X` blir byttet ut med kolonne `b`, begge kolonner er også representert i tabell `Y` hvis attributtverdier brukes til å initialisere attributtverdiene for `b` i tabell `X` med joinbetingelsen `Y.a = X.a`. Når kommandoen `DROP COLUMN a` kalles under `ALTER TABLE X`, vil påfølgende spørringer fra uvitende tjenere, som kjører den gamle versjonen av MediaWiki, påkalle databasefeil. På motsatt vis går det ikke an for oppgraderte MediaWiki-tjenere å spørre databasetjenere som kjører den gamle datamodellen der kolonne `b` ikke er definert i tabell `X`. Dermed må systemoppgraderingen foregå i to atomiske steg, der samtidige

spørringer trygt kan komme innimellom: Først defineres kolonnen `X.b` og initialiseres med data fra `Y.b` ved hjelp av tidligere nevnte joinbetingelse. I samme steg oppgraderes applikasjonstjenerne til MediaWiki (dvs mengden av PHP-skript). Før dette steget er ferdig gjennomført, får ikke klienter lov til å sende forespørsler overhodet. Steg 2: Slett kolonnen `X.a`.

Ved online oppgradering er det meningen at klienter kan aksessere systemet samtidig som det oppdateres. Den automatiserte prosedyren må derfor ta høyde for innkommende `UPDATE`, `INSERT` og `DELETE` - spørringer og samkjøre disse forespurte endringene med kolonnetillegget i tabell `X`. I praksis betyr dette at resultatrelasjonen fra joinbetingelsen blir kalkulert på nytt for hver nye forespørsel som inntreffer. Dumitras and Narasimhan (2009) argumenterer at sjonglering med versjonmiks mellom to forskjellige lag av systemet kan fort medføre ekstra ytelsesoverhead i form av forlenget oppgraderingstid sett i forhold til om den ble utført offline.

Dumitras and Narasimhan (2009) sin studie av MediaWiki sin oppgraderingshistorikk kan oppsummeres slik: Inkompatible skjemaendringer forhindrer rullerende oppgraderinger og påbyr at oppgraderingen skjer i ett helhetlig steg. Dataavhengigheter er vanskelige å synkronisere med endringer som følger av innkommende forespørsler og kan påføre ekstra prosesseringstid for oppdateringsprosessen samt overbelastning av lagringsressurser under datakonvertering og/ellere migrasjon. Dataskjemaendringer som motstrider hverandre, slik som flettekonflikter i asynkrone distribuerte databaser, krever manuell intervensjon.

Når programvaren, eventuelt operativsystemet som kjører på hver enkelt datamaskin i et produksjonsmiljø spredt utover opptil flere datasentre på vidt forskjellige geografiske lokasjoner skal oppgraderes fra én versjon til en nyere, så har systemets administratorer flere valg. Man kan slå av alle noder i hele systemet, eventuelt hele datasentre om gangen, for deretter å installere oppdateringen på én og én node. I systemer med et moderat antall datamaskiner som tilsammen leverer en applikasjonstjeneste over et nettverk av klienter, og som alle er i ett og samme tjenerrom samlet i én klynge over et lokalt nettverk, vil denne såkalte ”stopp-verden” - strategien, som Saur et al. (2016) nevner, være en gangbar strategi som innebærer lave, tolererbare kostnader i form av tapt tjenestetid for brukerne.

3.2 Kontinuerlig leveranse

Blogginnlegget til Hauer (2015) handler om behandling av databaser i en applikasjonlivssyklus der utviklingsmetodikken ”Kontinuerlig Leveranse” (eng. ”Continous Delivery”) gjennomføres. For hver gang ny kode skal kjøres på applikasjonstjenerne må databaseskjemaet og følgelig eksisterende data oppdateres på samtlige databaseinstanser før applikasjonsoppdateringen kan selv tre i kraft.

I webapplikasjoner med mange brukere opparbeides mye lagret data over tid. Følgelig blir datamigrasjoner som følge av endringer av applikasjonslogikken også mer tidkrevende. Hvis det stilles høye tilgjengelighetskrav til webapplikasjonen, kan ikke denne migrasjonsprosessen utføres i ett, langtekkelig steg. Når denne migrasjonsprosessen er ferdig utført er det minst like tidkrevende å ”rulle tilbake” disse endringene, og det lar seg nødvendigvis

ikke alltid gjøre. Dette er særlig tilfellet i produksjonsmiljø der databasesystemet integrerer data fra flere separate uavhengige applikasjoner, for eksempel et intranett for et firma med ett eneste databasesystem som håndterer data både for salgsavdelingen og varebeholdningsavdelingen. Således anbefaler Hauer (2015) å separere datakilder i en mikrotjenestearkitektur og derfor benytte et persisteringsmønster Sadalage and Fowler (2013) refererer til som ”polyglot persistance”.

I et kontinuerlig leveransemiljø med relasjonelle datamodeller anbefaler Hauer (2015) å gjøre skjemaendringer ”backwards compatible”, altså at den nye versjonen av dataskjemaet kan brukes av applikasjonslogikken som er koplet til det gamle databaseskjemaet. Et eksempel er fjerning av et attributt fra en tuppel. Denne endringen er ikke direkte bakoverkompatibel. Derfor må denne endringen oppdeles i et sett med endringer: Først må attributtet defineres til ikke å ha en påkrevd verdi, på fagspråket kalt ”nullable”. Dernest må applikasjonslogikken på hver applikasjonstjener oppdateres, slik at attributtet ikke etterspørres. Til sist kan det ubrukte attributtet fjernes fra databaseskjemaet.

3.2.1 Skjemaendringer og datamigrasjon i NoSQL-databaser

NoSQL-datamodeller har til felles at de ikke eksplisitt deklarerer et databaseskjema, slik som man ser i Postgres eller MySQL. Ethvert aggregat i prinsippet kan ha en hvilken som helst struktur og innheholde hvilken som helst datastruktur, inklusive nøstede lister og objekter. Kort sagt, i et nøkkelverdi-lager kan man lagre det man vil per nøkkel. Skjemaendringer er ikke et vesentlig problem i et kontinuerlig leveranse-utviklingsmiljø. Når man ikke trenger å vite strukturen på dataobjektene som lagres på forhånd, blir kontinuerlig endring og publisering til produksjon av applikasjonen tilsynelatende enklere.

Som Sadalage and Fowler (2013) poengterer at enhver applikasjon som persisterer sine tilstand, gjør til enhver tid antakelser på hvordan aggregatet den får tak i per oppslagsnøkler ser ut. Applikasjonens kildekode gjengir et implisitt skjema. Hvis koden er godt nok strukturert, for eksempel ved å definere egne modell-klasser som gjør applikasjonens dataskjema eksplisitt, vil gjøre jobben med å identifisere endringer i applikasjons usynlige dataskjema lettere.

Å migrere data i et høytilgjengelig produksjonsmiljø er dessverre ikke stort lettere med NoSQL enn i relasjonsdatabaser. Migrasjon er en aktivitet som nødvendigvis vil forekomme mye oftere i en kontinuerlig leveranse-livssyklus, og er desto mer tidkrevende jo mer omfattende endringen er. Hauer (2015) demonstrerer to endringsscenarier der levende datamigrasjon må utføres:

Enkel endring i datamodellen Et eksempel på en enkel datamodellendring er å endre datatypen til attributtet ”postNummer” i `Kunde` - aggregatet fra heltall til streng, jamfør modellen i 2.2. Så lenge applikasjonen klarer å håndtere den eksisterende datatypen for attributtet, så vel som den nye, så trenger vi ikke å oppdatere data som allerede er lagret i databasesystemet, med mindre de absolutt **må** være konsistente med den nye applikasjonsversjonen.

Kompleks endring i datamodellen Herunder inkluderes omarrangering av strukturen til

> aggregatet applikasjonen bruker, som en endring av objektnøstingen, oppsplitting eller fletting av aggregater. Å tillate en kunde å registrere flere leveringsadresser istedet for bare én, jamfør 2.2, telles som et eksempel på en slik kompleks endring.

Den sistnevnte endringen er ofte en nødvendig handling for å ”endre” spørringer til databasen. I prinsippet kan man ikke ”endre” spørringer i NoSQL-modellen, det finnes bare én type lesespørring og én skrivespørring. Følgelig blir programvareutviklere nødt til å strukturere aggregatet til å passe med de funksjonelle krav som stilles til spørringer i databasen. Når de funksjonelle kravene til applikasjoner, hvis datamodeller er aggregatbaserte, endres, så vil krav til spørringsresultater endres deretter, og følgelig må datastrukturen til aggregatet som lagres, også endres. Den logiske konklusjonen, som Hauer (2015) påpeker, er at omfattende datamigrasjoner gjøres oftere hvis datamodellen er enten delvis strukturert eller totalt ustrukturert, enn i strengt strukturerte datamodeller.

Der semistrukturerte datamodeller har stor skjemafleksibilitet har strengt strukturerte datamodeller stor spørringsfleksibilitet (gitt at de er normaliserte). I SQL-databaser trenger man ikke endre databaseskjemaet, metadataene som beskriver relasjonene i databasen, for hver eneste gang de funksjonelle kravene endres. Grunnen til det er at spørringene kan endres ved hjelp av `SELECT` og `JOIN` - operandene, i tillegg til aggregeringsoperander tilbudt av spørringsgrensesnittet.

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).

I diskusjonen om hvordan semistrukturerte datamodeller kan oppgraderes uten å avbryte tjenesteleveranse i produksjonsmiljøet, er visse fakta om aggregatorienterte datamodeller og NoSQL - modeller forøvrig, relevante å kunne. Slike databaser har ingen semantisk formening om data som lagres, de bare lagres på en eller flere lagerinstanser i et distribuert system. Selve meningen til de dataobjektene håndteres og utvikles i selve applikasjonen sammen med dets kildekode. I datalagerdelen av systemets arkitektur er disse objektene ofte representert som JSON-strenger, ProtoBuf-objekter, Avro-objekter og så videre.

3.2.2 Migrasjonsverktøy for Cassandra

I migrasjonsverktøyet *cassandra-migration-tool-java*, som er et tredjepartsverktøy til Apache Cassandra for databaseadministratorer, gjøres versjoner av det implisitte skjemaet til en ap-

plikasjon eksplisitt, og lagres i en separat metadatatabell, kalt `schema_version`, i form av en identifikatorstreng og et tidsstempel (Gobec and Bozic, 2015). Verktøyet, som er åpent tilgjengelig på GitHub, kan utføre to forskjellige migrasjoner: Skjema-migrasjoner, der databaseskjemaet (tillegg og fjerning av attributter, tillegg av "tabell") endres, og data-migrasjoner (endringer av eksisterende aggregat i databasen) (Gobec and Bozic, 2015). Ved skjemamigrasjon er det, iallfall for modulens forfattere, kritisk for programmet å sikre oppdateringskonsistens i databaseklyngen hvis databaseadministratoren har behov for det.

Derfor er en konsensusprotokoll implementert der alle spørrende klienter nødvendigvis må aktivt vente på at alle databasenoder har migrert ferdige sine egne skjema før de kan sende lese - og skriveforespørsler. Konsekvensen av ikke å føre en slik konsistenskontroll på replikanivå er at applikasjonsklientene eksponeres for versjonmiksen i datalageret, noe som åpner for versjonskappløp replikaene imellom per nøkkel ved både lese - og skrive-operasjoner.

3.3 EBR

Online oppgradering av distribuerte programvaresystemer blir stadig mer nødvendig for stadig flere virksomheter. Å planlegge nedetid for hele systemer,selv om det er for en god sak, nemlig viktige vedlikeholdsoperasjoner, er logisk ekvivalent med å avbryte flyten i forretningsprosessene og følgelig tape inntekter og kundetilfredshet – med vilje vil kanskje utenforstående si. Behovet for online oppgraderinger er definitivt prekært, og en suksessfull implementasjon av automatisert oppgradering vil utvilsomt gi sterk datadrevne tertiære bedrifter et viktig konkurransefortrinn (Choi, 2009).

Utgavebasert redefinisjon (EBR) er en oppgraderingsmetode ment for å oppgradere databaseprogrammet som holder rede på datalaget til en applikasjons arkitektur, uten å forårsake nedetid for applikasjonens sluttbrukere. Denne metoden må applikasjonens utviklere ha god kjennskap til når de implementerer selve oppdateringsskriptet, ellers er det vanskelig å få oppgradert applikasjonen online uten å påføre systemsvikt (Choi, 2009).

EBR har en vært fast funksjonalitet i Oracle - databasesystemet siden versjon 11gR2, og ser ut til å være ganske populær blant store aktører med produksjonsmiljø som aksesseres av mange brukere verden over 24 timer i døgnet. Choi (2009) kommenterer at både BetFair og IFS, to store tertiære selskaper, er ivrige brukere av patching med EBR. Disse to kundene av Oracle demonstrerer at det er et meget stort marked for arkitekturer som støtter oppdatering av programvaresystem uten nedetid. Et kjapt internettsøk avslører at denne oppgraderingsmetoden er fremdeles ønsket og relevant for datadrevne applikasjoner som benytter en relasjonell datamodell. [1]

[1] En video der en ivrig EBR-bruker forklarer metodens fordeler er å finne på `https://www.youtube.com/watch?v=nRiAQgNDgoA`

3.3.1 Løsning

I versjon 11.2 av Oracle definerer Choi (2009) til i alt tre nye konsepter for å muliggjøre online oppgradering:

- Utgave (”Edition”). Hver instans av for eksempel en funksjon, SQL-setning, pakke eller view (immuterbar dataobjekt) eksisterer som en egen entitet, en utgave av nevnte konstruksjon. Slik definerer konseptet om utgaver en måte å skille variasjoner av det samme kodeobjektet fra hverandre, slik at man kan operere på en spesifikk instans av gangen

- Utgaveperspektiv (”Editioning View”). Ved hjelp av projeksjoner gjemmer utgaveperspektivet nyopprettede attributter eldre utgaver av koden ikke er ment å se fordi attributtet ikke skal eksistere i deres verdensbilde

- Triggere på tvers av utgaver (”Cross Edition Triggers”). For å synkronisere data forskjellige utgaver ikke har til felles, brukes triggere. Dataendringer som inntreffer hos tupler tilhørende eldre utgaver, propageres til kolonnene nye utgaver leser ifra, og vice versa

Meningen med utgaver er å sette opp et isolert miljø slik at mengden av kodeobjekter (instanser fra kildekoden) som trygt kan endres i tandem gjøres Slik. Å inkludere dataverdier i seg selv i dette miljøet vil åpenbart medføre mange unødige, tunge dataoperasjoner i den grad datatupler dermed må oppdateres og kopieres fram og tilbake for hver opp - eller nedgradering. Derfor behandles kolonner og tabeller som statiske elementer, åpent synlig for alle versjoner. Slik blir deling av data enklere. Semantisk sett må det gå raskt å opprette nye utgaver, men samtidig må tilstanden til koden pre-patch også være tilgjengelig for nye utgaver (Choi, 2009). Utgaver innehar derfor følgende semantikk:

1. Det finnes to typer objekter: versjonerbare (editionable) og ikke-versjonerbare, hvorav sistnevnte har den statiske egenskap at de er konstant identiske og synlige for alle utgaver (herunder inngår dataobjekter som tabeller og kolonner)

2. Når en ny utgave opprettes, vil den holde på en komplett historikk av alle versjonerbare objekter

3. Endringer av versjonerbare objekter er kun synlige for versjonen der nevnte endring ble utført

For å illustrere hvordan de tre ovennevnte konseptene fungerer sammen for å realisere en trygg oppgraderingsprosess som kan brukes til online patching, så vel som tilbakerulling av oppgraderinger, beskriver Choi (2009) et case med et eksempelskjema levert av Oracle Corp. I caset blir et attributt for telefonnummer splittet i to, en for landskode og en for telefonnnummer innad i landet med den korresponderende landskoden. I den datamodellen utvinnes to nye attributter og én ny funksjonell avhengighet fra landskode på telefonnummer. I denne patchen blir også en del SQL-prosedyrer oppdatert for å samsvare med denne attributtsplittingen. Omfanget av dette caset representerer en typisk patch for en datamodell som allerede kjører i et produksjonsmiljø.

Tilbakerulling av en oppgradering i tilfelle en svikt oppsto med den nye utgaven er såre enkelt:

1. Post-patch - utgaven blir slettet
2. Nye tabeller, samt nye kolonner i eksisterende tabeller, som ble opprettet i løpet av oppgraderingen blir gjemt bort takket være projeksjonene til utgaveperspektiv-komponenten

Ubrukte tabeller og kolonner kan dermed tas fram igjen"(Choi, 2009) hvis man forsøker å installere patchen på nytt og lykkes. Tilbakerulling av oppgraderinger har ingen effekt på tilgjengeligheten til pre-patch - utgaven.

3.3.2 Diskusjon

Choi (2009) beskriver metoden utgavebasert redefinering (eng. Edition-Based Redefinition) som er implementert i Oracles database. EBR er en patchingmetode for databaseapplikasjon-instanser slik at patching av databasen ikke påkrever planlagt nedetid av appliaksjonssystemet i sin helhet. Vitnesbyrd fra to store firma beviser at EBR fyller et sårt behov for online programvareoppgradering. Dataobjekter, som for eksempel tabeller, er inkompatible med utgavemetoden. EBR fokuserer på prosedyrer og funksjoner databasesystemet er bygget med. Med tanke på at artikkelen ble utgitt i 2009, er det tenkelig at de fleste industrier, deriblant telekom, allerede har anvendt manuell, rullerende oppgradering av sine distribuerte systemer over lang tid, og møtt på overraskende og dyre forviklinger som ikke kunne ha blitt forutsett uansett hvor erfarne databaseadministratorene er.

Som Oliveira et al. (2006) påpeker i deres oppsummering av undersøkelsene sine, er det menneskelige feil som står for majoriteten av feil som oppstår i databaser som kjører i et produksjonsmiljø. Feil som databaseadministratoren forårsaker, er som regel ikke maskerbare med for eksempel backup - teknikker eller redundansstrategier. Slike feil kan medføre blant annet at lagrede data blir fullstendig utilgjengelige, sikkerhetssårbarheter oppstår og at systemets ytelse svekkes betraktelig.

Et relevant spørsmål man kan stille denne oppgraderingsmetodologien er: Hvordan håndterer den miksede versjonstilstander? Det interessante svaret er at skopet til EBR er det relasjonelle databasesystemet applikasjonen persisterer sin datatilstand med. Versjonskappløp og miks av datalagversjoner og tjenestelagversjoner/controllerlagsversjoner er kun relevant for rammeverk som anskuer hele programvarearkitekturen og som versjonerer datamodellen separat fra selve applikasjonen, noe som faller naturlig for applikasjoner hvis datamodell opprettholdes av en relasjonell struktur.

Så kan man alltids lure på hvorfor Choi (2009) ikke stiller opp med en eksperimentell evaluering av teknologien de framstiller, all den tid designvalgene begrunnes blant annet med synet på ytelse. Kanskje oppleves skryten fra markedet eller det faktum at EBR fortsatt er en funksjonalitet av Oracle-databasen, i det vi skriver versjon 12c, som ble sluppet den 1. mars 2017, som bevis nok for bibliotekets utviklere. Dessuten er ordentlig testing av distribuerte systemer med en ordentlig distribuert arbeidslast en ganske dyr affære. Det er

trolig grenser for hvor presise, kostbare og omfattende tester Oracle tillater for ett eneste aspekt ved Oracle RDBMS.

3.4 Imago

Her følger en presentasjon av en helhetlig systemarkitektur som unngår versjonsløpsbetingelser og gjennomfører atomiske, konsistente datakonverteringsprosedyrer med minimalt behov for manuell styring. Med dette designet foreligger en vesentlig økning både i maskinberegninger (CPU), og bruk av lagringsressurser (disk). Dumitras and Narasimhan (2009) påpeker at designet kan innføre flere oppgradingsscenarier uten å påtvinge planlagt nedetid for applikasjonen som kjører med dette systemdesignet.

Dumitras and Narasimhan (2009) undersøker årsakene til planlagt nedetid ved å studere oppgraderingshistorikken til Wikipedia. Endringer i skjemaet som krever at både applikasjonen og datalaget oppgraderes i ett enhetlig, atomisk steg, fordrer vanligvis at hele programvaresystemet må slås av.

3.4.1 Løsning

Imago er bygget på tre grunnleggende målsetninger (Dumitraş and Narasimhan, 2009):

- **Isolasjon**. Avhengigheter innad i det gamle unierset/versjonen av systemet må være adskilt fra oppgraderingsprosessen
- **Udelbarhet**. Under et hvilket som helst gitt tidspunkt kan klienter som kontakter systemet under oppgradering bli betjent av enten den gamle eller den nye versjon, men aldri begge to. Dessuten må oppgraderingen skje på atomisk vis
- **Naturtro testmiljø**. Testmiljø må rekonstruere realistiske omgivelser som produksjonsmiljø trolig kan utsettes for

Dumitraş and Narasimhan (2009) sitt bidrag til å automatisering av applikasjonsoppgraderinger er et system, der oppdateringer kan utføres online, og samtidig inkludere komplekse endringer i dataformatet. I Imago betegner begrepet ”univers” en distinkt logisk mengde av tjenernoder som eksisterer i form av prosesser som kjører i fysiske datamaskiner eller prosesser i virtualiserte datamaskiner. Image definerer to slike univers, ett for det eksisterende systemet, og ett der den nye versjonen av systemet startes opp og kjører parallellt med den gamle versjonen av systemet. Baktanken med denne arkitekturen er å forhindre versjonsløpsbetingelser (”mixed version race”). Dumitraş et al. (2010) viser at versjonsløpsbetingelse kan medføre uante konsekvenser i systemets oppførsel i form av nye feil som oppstår. Disse feilene oppstår på grunn av uventet kommunikasjon mellom en oppgradert versjon av ett lag i systemet (for eksempel applikasjonslogikken) og en gammel versjon av ett annet lag, som for eksempel datalaget. Systemets arkitekter tar sjelden høyde for slike feil, gitt deres eksepsjonelle natur.

3.4.2 Diskusjon

En vesentlig antakelse som ligger til grunn for Imago er at det distribuerte systemet har ett aksesspunkt, eller API, for innkommende forespørsler fra brukere, og én eneste metode applikasjonstjenere kommuniserer med databasetjenere/datamodellen på.

Imago kan garantere fraværet av versjonsløpsbetingelser til gjengjeld for en vesentlig, dog midliertidig økning i bruk av lagringsressurser og beregningsressurser. KVolve har ennå til gode å bli implementert og testet ut på et større distribuert system (Saur et al., 2016), men følger det samme udelbarhetsprinsippet som Imago.

En svakhet med udelbar oppgradering er at oppgraderingsprosessen kan aldri termineres hvis systemet samtidig mottar mange datamanipuleringsforespørsler mens oppgraderingen kjører. Dette fordi en oppdatering øyeblikkelig fører til at alle tuplene som utgjør datamodellens nåværende tilstand, må leses på nytt. Derfor avbryter og utsetter Imago skriveforespørslene `CREATE`, `UPDATE`, og `DELETE` under oppgraderingsprosessen, enten ved å blokkere dem direkte, eller ved kun å tillate lesespørringer. For å sikre den atomiske oppgraderingen av systemet, er det nødvendig at utenforstående operasjoner ikke kan forstyrre datamigrasjoner forbundet til endring av applikasjonens oppførsel.

3.5 KVolve: Evolusjon av datamodeller uten nedetid

Artikkelen som presenteres i dette delkapitlet handler om levende oppdatering av datamodellen til høytilgjengelige webapplikasjoner uten tjenesteavbrudd eller ytelsesdemping, der et nøkkelverdi-datalager (her representert ved Redis) brukes til å persistere applikasjonens data. Selv om nøkkelverdi-datalagre er skjemaløse, påfører webapplikasjonen datamodellen gjerne en logisk aggregatstruktur, realisert med spesielle dataformat. Eksempler på slike format inkluderer protokollbufre, Avro - objekt eller JSON - dittoer (Saur et al., 2016).

Nye funksjonelle krav som spesifiseres for webapplikasjoner, medfører ofte endringer i datamodellen. Typiske endringer i aggregatet inkluderer fjerning av, gi nytt navn til eller tillegging av ett eller flere attributter. Dataformatet til aggregatet kan også bli endret fra for eksempel JSON til Apache Thrift. Sistnevnte innehar for øvrig støtte for versjonering og sporing av ”dataformatet”.

Den typiske metoden for å implementere slike oppdateringer i NoSQL - datamodellen, er å migrere dataene fra den gamle datamodellen, slå av samtlige noder i tjenesten, installere oppdateringen, konvertere dataene til å passe den nye modellen, og deretter importere dataene inn i den oppdaterte databasen. Dette medfører selvsagt nedetid i systemet, som tradisjonelt sett kan bortdefineres (jamfør kapittel 2.1), da vedlikeholdsarbeid gjøres på et tidspunkt ingen behøver systemets tjenester.

I en høytilgjengelig webapplikasjon som betjener forespørsler nærmest døgnet rundt er denne stopp-og-restart-strategien uholdbar. Det er også problematisk å konvertere data

slik at den konformerer med et nytt aggregatformat. Dette fordi migrasjon er en veldig tidkrevende prosess. Ved å opprettholde bakoverkompabilitet på dataformatet kan systemets utviklere trygt oppgradere datamodellen med en manuell, rullerende teknikk, men da vil man samtidig legge kraftige begrenser på hvordan applikasjonen videreutvikles.

3.5.1 Løsning

KVolve, hvis navn kommer fra frasen *Key-value store evolution*, er en utvidelse av det populære databasesystemet Redis. Det er implementert i C i form av et modulært bibliotek kompilert sammen med Redis. Kildekoden til dette selvstendige databasesystemet ble publisert på GitHub den 18. september 2016 [2], og benytter kildekodeversjon redis-2.8.17 av Redis i sin implementasjon. KVolve fungerer som et eget databasesystem som tilbyr støtte for levende skjemaoppgradering gjennom lat datamigrering (eng. ”lazy migration”) (Saur et al., 2016).

Systemet eksponerer aggregatformatets versjon for applikasjonen. Data konverteres ikke på ivrig vis, ved for eksempel å iterere over nøklene i databasen og endre både dem (hvis nødvendig) og deres verdier. Formatkonvertering av nøkler og verdier utføres når applikasjonen gjør oppslag på dem. KVolve sporer versjonene til de forskjellige dataobjektene ved hjelp av prefikser i nøklene. For hvert dataobjekt lagres en versjonstag (”version tag”) som i tilfeller der dataobjekt ikke er blitt migrert, kan være eldre enn dets logiske versjon (Saur et al., 2016).

Idet applikasjonen kopler til KVolve må den indikere hvilken logisk versjon av datamodellen den forventer å få. Tilkoplingen tillates hvis den oppgitte versjonen tilsvarer den nyeste KVolve har opprettet. Ved oppgradering av datamodellen må logisk nok også selve applikasjonen oppgraderes for å kunne behandle den nye versjonen. Dette kan for eksempel gjøres på manuelt, rullerende vis. For å kunne ta i bruk dette systemet, kreves ingen manuell versjonshåndtering i applikasjonen.

Med dette systemet kan utviklere definere en oppgraderingsspesifikasjon som består av en mengde transformasjonsfunksjoner, som likner på det Hicks and Nettles (2005) kaller for ”state transformation functions”, i forbindelse med dynamisk oppdatering. Michael Hicks er for øvrig en av medforfatterne av denne artikkelen. Oppgraderingsspesifikasjonen definerer hvordan eksisterende data må transformeres for å passe den nye datamodellen. Disse transformasjonsfunksjonene kompileres til en delt objekt - fil som KVolve kan lese inn ved hjelp av I/O - grensesnittet til Redis. Når datamodellen er oppdatert, blir dette delte objektet persistert til disk i Redis.

Selve datatransformasjonen følger en lat strategi, altså blir data transformert i tråd med spesifikasjonen først når den spørres etter i applikasjonen. Lat datamigrering er også en kjent strategi som kan benyttes ved rullerende oppgraderinger av datamodeller, der dataobjekter som aksesseres, konverteres til det nye aggregatformatet når spørringer inntreffer. Denne konverteringen gjøres da for hver enkelt spørring, og legger en vesentlig demper på

[2] Tilgjengelig på Github, `https://github.com/plum-umd/kvolve`

spørreytelsen til databasen. Late beregningsstrategier står sentralt i det funksjonelle programmeringsparadigmet, der de utgjør en vesentlig faktor for ytelsen til språk som Haskell, Scala og Erlang.

KVolve definerer to typer oppdateringer av formater i datamodellen: 1: Oppdatering kun av nøkkelformatet og 2: oppdatering av nøkkelformatet og verdiformatet. Ved sistnevnte påkalles transformasjonsfunksjonen. Denne funksjonen tar inn to argumenter: Én referanse til en gammel nøkkel i form av en streng, og én referanse til en mengde binærdata. Den binære strengen er dataverdien strengnøkkelen peker på i databasen. Transformasjonsfunksjonen bruker disse referansene til å endre nøkkelen og dets assosierte verdi til å samsvare med den nye versjonen av datamodellen. Figur 1 til og med 5 illustrerer et eksempel på en datamodelloppgradering (Saur et al., 2016).

3.5.2 Evaluering av løsning

Artikkelens eksperimentelle resultater kan kort oppsummeres som følger: Ved hjelp av ytelsesmålingsverktøy innebygd i Redis ble det avdekket lite ekstra kostnad i ytelse ved normale databaseoperasjoner, og versjonslagring og dataoppdatering medfører et tillegg i lagringsforbruk på inntil 15 % (128.6 MB kontra 112.1 MB). I tillegg ble en test utført der filsystemet brukt til å lagre data med, RedisFS, ble oppdatert. I denne oppdateringen utføres også navneendring på enkelte nøkler og komprimering av dataverdier. Den late datatransformasjonsstrategien utklasser stopp-og-restart-strategien i tidsbruk. Ved datamigrasjon offline var tiden 12 sekunder for samme oppdatering. Versjonskontrolleringen i KVolve medfører naturlig nok også litt ekstra minnebruk. Evalueringene ble utført på én enkel datamaskin med 24 prosessorkjerner og 32 GB RAM, med operativsystem Red Hat Enterprise Linux, versjon 6.5.

Redis-bench er et innebygd benchmarking - verktøy i Redis. Redis-bench emulerer en klientprosess som sender forespørsler til Redis-databasetjeneren. For denne ytelsestestingen av KVolve er antallet nøkler i databasen konfigurert til én million, og antallet operasjoner til fem millioner, slik at testen strekker over lang tid og mange forespørsler. Dermed får man et klarere bilde av det langsiktige ressursoverheadet som det dynamiske skjemaoppgraderingsrammeverket påfører applikasjonen, og i tillegg blir testscenariet så realistisk som mulig.

For å få stabile testresultater som eliminerer eksepsjonelle enkeltresultater har benchmarkingen blitt kjørt flere ganger. Tallene Saur et al. (2016) lister opp i tabell 1 av artikkelen er mediantider målt med en nøyaktighet på hundredels sekundet, beregnet ut ifra totalt 11 forsøk.

Ytelsespåvirkningen, målt i antall spørringer per sekund, til to forskjellige applikasjonsoppgraderinger er blitt testet. I det ene scenariet har RedisFS, et filsystem hvis metadata som inoder og kataloger og filsystemdata er lagret i Redis, blitt oppgradert fra versjon `redisfs.5` til `redisfs.7`. Tre forskjellige metoder er blitt brukt i oppgraderingene. I den ivrige metoden blir forespørselstrafikken til Redis avbrutt helt, navngivning av nøkler utføres ved behov og data migreres på manuelt vis. KVolve sin oppgraderingsmetode, der tjenesteforespørsler blir behandlet mens oppgraderingen utføres, er blitt realisert

med kodeendringsverktøyet Kitsune, som også er blitt utviklet ved universitet ved Maryland av blant annet artikkelens hovedforfatter. Den tredje metoden er den manuelle, late migrasjonsmetoden beskrevet i kapittel 12 av Sadalage and Fowler (2013).

I det andre oppdateringseksperimentet ble et sosialt medium - system implementert ved hjelp av biblioteket Amico (det italienske ordet for "venn"). I dette systemet ble Amico - komponenten oppgradert fra versjon `1.2.0` til `2.0.0`, både ved manuelle datamigrering og ved hjelp av KVolve. Med datamigrasjon på vanlig, ivrig vis var applikasjonen utilgjengelig i totalt ett minutt og 27 sekunder, mot ett sekund for KVolve (Saur et al., 2016). Figur 7 illustrerer disse ytelsesforskjellene tydelig og kvantitativt.

3.5.3 Diskusjon

Til tross for artikkelens praktiske tilnærming, er den publiserte kildekoden kun ment å være en "proof-of-concept" - implementasjon av levende skjemaoppgradering. Ved evaluering av systemet er det blitt foretatt både mikro-benchmarking og marko-benchmarking, altså har man testet oppgraderingsmekanikkens ytelsespåvirking både på databasens indre ytelse og dets innvirkning på en kjørende webapplikasjon i sin helhet.

Evalueringsseksjonen fokuserer ikke på å sammenlikne KVolve med andre systemer som organiserer skjemaendringer på dynamisk vis. Dette kan ha noe med at samhandling mellom datammodellendringer og applikasjonslogikken er en oppgave som tradisjonelt overlates til den enkelte applikasjonsvedlikeholder. Derfor er det svært få systemer KVolve kan sammenliknes med. Ett sammenlignbart system Saur et al. (2016) nevner, er Googles F1, som har en asynkron protokoll som kan legge til og fjerne tabeller, kolonner og indekser på dynamisk vis. På den måten har F1 tilgang til datamodellen under oppgradering. Her blir linearisering av oppdateringer håndtert ved hjelp av fysiske klokker levert av TrueTime API-et til Google. F1 støtter imidlertid ikke endring av ProtoBuf - formatet som lagres i dets kolonner.

De fem første figurene i artikkelen gir et godt bilde av en eksempelapplikajson som henter inspirasjon fra Sadalage and Fowler (2013) og hvordan KVolve oppfører seg med de gitte objektformater. Figur 6 og 7 illustrerer tydelig ytelsesfordelen KVolve gir sammenliknet med konvensjonell lat datamigrasjon og ivrig, rullerende oppdatering.

Et annet poeng til denne artikkelen er hvilken type problem med levende oppgradering av databaser den prøver å løse. KVolve behandler oppgradering av datamodeller, ikke hele databaseapplikasjoner, slik som Imago kan få til. Det påpekes i sistnevntes disfavør at mens den realiserer atomisk oppgradering av applikasjoner, så er det på bekostning av en vesentlig økning - om enn midlertidig - ressursbruk i form av dataduplisering og dobbelt opp av noder.

Testene referert til i forrige delkapittel er ikke gjort i et reelt distribuert system med forskjellige maskiner på forskjellige datasentre rundt om i verden, men på én enkelt datamaskin som bruker loopback-grensesnittet (localhost). Testresultatene gir altså ikke et bilde av hvordan KVolve kan yte i et moderne produksjonsmiljø, et aspekt artikkelens konklusjon gjerne har lyst til å finne ut av ved å få KVolve implementert på Redis Cluster.

Kapittel 4

Levende oppgradering av aggregatorienterte datamodeller

Her følger en skisse av en modulær programvareløsning kalt DBUpgradinator, implementert i Java. De første to delkapitler diskuterer modulens inspirasjoner fra KVolve og problemstillingens måloppnåelseskrav. I det tredje delkapitlet skisseres løsningens tre designmessige perspektiv: Dets fysiske perspektiv; det logiske perspektivet; prosessperspektivet. Dernest skildres kildekoden til modulen. Til sist blir evalueringsprosessen av oppgavens måloppnåelseskrav beskrevet, og et testprogram implementert til inntekt for evalueringen skildres. For å persistere aggregat-objekter brukes Project Voldemort.

Oppgaven implementerer en frittstående applikasjonsmodul som forsørger støtte for automatisk dataoverføring ved dataaksessering, benevnt som ”lazy data migration” av Saur et al. (2016), og samtidig tester det ut i et simulert produksjonsmiljø med en aggregatorientert datamodell.

4.1 Inspirasjoner fra KVolve

Implementasjonen av migrasjonsløsningen er i stor grad fundamentert på de samme tre sentrale karakteristikkene som KVolve har.

Versjonering av aggregat KVolve kopler sammen logiske versjonsidentifikatorer med aggregatnøkler. Forskjellige versjoner av aggregatene knyttes til forskjellige nøkkelprefiksstrenger, det vil si at at to aggregat som er skrevet innen samme skjemaversjon av applikasjonen har samme nøkkelprefiks-streng. Til eksempel er to aggregat *k:y* og *k:x* begge skrevet under skjemaversjon *k*, derfor identifiseres begge med prefikset *k:*. En applikasjon som sender en spørring til databasen indikerer hvilken skjemaversjon den vil se (Saur et al., 2016).

Oppdaterings-spesifikasjoner og aggregat-transformasjonsfunksjoner Når en datamodell skal oppgraderes installerer databaseadministratoren en programkompilert spesifikasjon som beskriver det nye versjonsprefikset som identifiserer versjonen til det neste dataskjemaet, samt en mengde aggregat-transformasjonsfunksjoner som brukes til å oppgradere visse aggregater. Enhver transformasjonsfunksjon har et prefiks i sitt navn, *p* som indikerer skjemaversjonen til aggregater funksjonen skal kalles på.

Aggregat-transformasjon ved dataaksess Aggregater lagret i KVolve blir ikke automatisk oppgradert når en ny oppdateringsspesifikasjon installeres i databasen. Når en ny versjon av applikasjonstjeneren starter vil den handtere brukerforespørsler og deretter sende GET og SET - operasjoner til databasen. Hvis en GET - operasjone treffer et aggregat som har en eldre skjemaversjon men ikke den nyeste, vil en transformasjonsfunksjon kalles for å oppgradere aggregatet slik at det kan returneres til applikasjonsinstansen med den nyeste skjemaversjonen. Hvert enkelt aggregat blir altså oppgradert ved forespørsel, og ikke på ivrig, tvungen vis som direkte respons på at en oppgradering er kommisjonert. I KVolve blir applikasjonsinstanser hvis skjemaversjon er utdatert automatisk frakoplet databasen og får ikke lov til å sende forespørsler. Således hindrer Saur et al. (2016) et mikset versjonsmiljø og sikrer konsistente skjemaversjoner.

Oppgavens programvareløsning, DBUpgradinator, er preget av noen annerledes designvalg som passer bedre til en likemannsarkitektur. Det mest vesentlige avviket er at datamodelloppgraderingsløsningen, som implementeres i denne oppgaven, tillater miksing av skjemaversjoner blant applikasjonstjenerne, slik at løsningen blir kompatibel med rullerende oppgraderingsprosedyrer. Om datamodellen må oppgraderes på grunn av endringer i applikasjonslogikken så skal ikke det gå utover tjenesteytelsen til de uoppgraderte applikasjonstjenerne her og nå. For DBUpgradinator kan vi ha persistert to aggregat, *k:x* og *j:x*, som begge i prinsippet peker på samme "dataobjekt", men for to forskjellige skjemaversjoner. Dette scenariet er ikke tillatt i KVolve.

Til grunn for denne designavgjørelsen ligger replikeringsmønsteret til Redis. Redis bruker Mester-Slave - mønsteret. Mester-Slave - replikering innebærer at én node i en databaseklynge velges til å ha vervet som klyngens mester av de andre nodene, som blir mesterens slaver. Mesternoden har ansvaret for å distribuere skriveforespørsler utover klyngen, så vel som replikering av skriveoperasjoner. Fordi mesternoden håndterer samtlige skriveoperasjoner i klyngen er replikakonsistens lettere å opprettholde enn for en likemannsarkitektur. Derfor er migrasjonsløsningen til Kvolve implementert på databasenivået fra det logiske perspektiv. Likemannsreplikerte systemer som Project Voldemort prioriterer tilgjengelighet for interaksjon med databasen over konsistente replikaer. Derfor er DBUpgradinator implementert i applikasjonsnivået sett fra det logiske perspektiv.

Et annet designmessig avvik ligger i programmeringsspråket brukt til å implementere DBUpgradinator, Java. Proof-of-conceptimplementasjonen av KVolve er skrevet i C. I et objektorientert språk som Java kan klasser brukes til å modellere oppdateringsspesifikasjonen. En klasse kalt **AbstractAggregateTransformer** realiserer både oppdateringsspesifikasjonen, ved å holde på to tilstandsvariable som indikerer navnet på en skjemaversjon som kommer før en annen, og dets transformasjonsfunksjon.

KVolve - implementasjonen til Saur et al. (2016) kan også brukes til å oppgradere applikasjonstjenere, i artikkelen brukes Redis - modulen Amico som et praktisk eksempel. I tillegg er KVolve innstilt på å vise et logisk konsistent bilde på datalget for webapplikasjonen som opererer på den, av hensyn til Dumitras and Narasimhan (2009) sine krav referert til i delkapittel 3.4.1. Når man implementerer en liknende tjeneste i en høytilgjengelig applikasjon må man være obs på at designvalgene gjort i Amazon Dynamo er gjort primært for å fremme tilgjengelighet for skriveoperasjoner.

4.2 Krav til implementasjon av levende datamigrasjonsløsning

Her følger de funksjonelle og ikke-funksjonelle krav som stilles til programvaremodulen som skal kunne migrere data fra et implisitt aggregatskjema til det neste uten stans i dataleveranser på brukernes tjenesteforespørsler.

4.2.1 Antakelser

I kraft av å demonstrere et praktisk konsept vil modulen gjøre visse antakelser om programvarearkitekturen til den jevne webapplikasjon DBUpgradinator kommer til å operere i. Enkelte av disse fordringene er gjort for å gjøre den objekt-orienterte implementasjonen av datamodellevolusjonsløsningen mindre parametrisert og derfor enklere og skrive og enklere å forstå.

For å realisere tillatelsen av versjonsmiks mellom forskjellige applikasjonstjenere i webapplikasjonen DBUpgradinator er installert i vil dets implementasjon ikke oppdatere aggregater direkte, men heller lage et nytt et med et annerledes streng-prefiks. Følgelig vil den samme ”tuppelen” lagres dobbelt opp hvis miksen av skjemaversjoner er av størrelse 2.

Her listes øvrige antakelser, begrensninger og spesifikke designvalg for prototypeimplementasjonen av DBUpgradinator, databasen den kommer til å operere på, webapplikasjonen den testes i, og produksjonsmiljøet DBUpgradinator testes i.

- I databasearkitektur modulen testes i serialiseres aggregater ’client’ side, altså er lagringsnodene ’dumme’ og backend-tjenerne ’smarte’
- Project Voldemort brukes som database i testapplikasjonen
- Serialisering (Voldemort): String, altså ser den enkelte databasetjener kun strenger og kan ikke lese av skjemaet; videre antas det at databasen bruker LWW - strategien for å løse flettetvister automatisk
- Applikasjonsstakken til Voldemort-instansen denne oppgaven tester har en fast konfigurasjon
- Nøkler i skjemaet endrer ikke type eller form

- Dataobjekter har en skjema-versjonstag brukt til å sjekke om tuppelen må oppgraderes
- Fordi modulen i prinsippet må være skjemaopplyst må den implementeres som en del av applikasjonslogikken
- Oppdateringsspesifikasjoner kan kjedes sammen, denne kjeden har Migrator-klassen styr på, altså er det Migrator-klassen som har en tilstandvariabel som peker på en lenket liste av instanser av transformasjonsobjekter
- Akkurat som med KVolve blir ikke disse transformeringsobjektene fjernet når alle aggregatene lagret på det gamle skjemaet er migrert ferdig
- Applikasjonen, skrevet i Java, holder styr på sitt implisitte skjema gjennom eksplisitt versjonering og eksplisitt deklarering av aggregatets modell ved å definere en egen Aggregate-klasse
- Webapplikasjonens utviklere programmerer på et eget lokalt utviklingsmiljø
- I serversiden er tupler lagret med nøkler på form k:x, der suffikset x indikerer skjemaversjonen. Disse dataobjektene har ingen separate versjonsfelt idet de sendes Voldemort-serveren
- Den enkelte utvikler har ikke behov for å endre nøkkelens struktur eller datatype
- Alle nøkler Voldemort mottar er på strengform, selv om nøkler i praksis også kan være en liste av binærdata, derfor støtter ikke DBUpgradinator nedring av nøkkelens skjemastruktur
- Webapplikasjonen som modulen testes i har en RESTful - arkitektur
- For ett bestemt produksjonsmiljø er det til enhver tid kun én klientinstans som kommuniserer med de andre; denne klientinstansen sender til enhver tid transformasjonsobjekter i ordentlig rekkefølge, det vil si at klienten gjør seg ferdig med å sende ett transformasjonsobjekt for én skjemaversjon til alle applikasjonstjenere før denne sender over transformasjonsobjektet som gjelder for den etterfølgende versjonen
- Når transformasjonsobjekter sendes, så sendes de i en bestemt rekkefølge, altså er ikke rekkefølgen applikasjonstjenere mottar transformatorer i tilfeldig - denne antakelsen tillater oss å spesialisere og forenkle enkelte detaljer av kildekoden som kjører hos applikasjonsinstansene
- Modulen skal kjøre i produksjonsmiljøet til en webapplikasjon skrevet i Java
- Til enhver tid oppgraderes applikasjonen kun én versjon ad gangen, det vil si at på det meste er to forskjellige utgaver av versjonen som opererer i produksjonsmiljøet på ett og samme tidspunkt

4.2.2 Funksjonelle krav

1. DBUpgradinator skal lage én ekstra databasespørring for hver nye transformasjon som påføres et aggregat
2. Ved en GET-forespørsel til aggregatet med nøkkel *x* fra en applikasjonsinstans som bruker en ny skjemaversjon *k* skal DBUpgradinator også opprette en GET-forespørsel etter det samme aggregatet men med prefikset til den foregående skjemaversjonen i nøkkelen, *j*
3. Hver enkelt applikasjonsinstans har kontroll over versjonen av dataskjemaet sitt, som identifiseres ved en hashstreng som konkatineres med IDen til hver enkelt dataelement før PUT sendes til databasen
4. DBUpgradinator vil opprette en ny tuppel i lageret hvis skjema-versjon-suffiks er annerledes og dataobjekt-ID-prefiks er likens prefikset til tuppelen som oppgraderes; således kan det distribuerte systemet kjøre en mikset tilstand mens datamigrasjonen foregår
5. Ved kjøring av DBUpgradinator: Hvis programmet mottar en spørring etter et aggregat som tilhører den nyeste skjema-versjonen (f.eks en GET på en nøkkel hvis skjemaversjon-suffiks tilsvarer versjonsstrengen til den nyeste skjemaversjonen) så vil ikke en migrasjon med transformator-klassen bli utført

4.2.3 Kvalitetskrav

1. Tilgjengelighet: Det skal være mulig å forespørre data fra databasen mens datamigrasjon er iverksatt. Dette kravet evalueres ikke på kvantitativt, men på kvalitativt grunnlag, det vil si ved å kjøre DBUpgradinator i et simulert produksjonsmiljø, det vil si en applikasjon som ikke mottar forespørsler fra klienter som kommuniserer med HTTP.
2. Ytelse: Ved rullerende, eller lat datamigrasjon, utsettes databasen for en vesentlig degradering i gjennomstrømskapasitet, målt i spørringer per sekund. Denne hemmelsen må være minimal i den implementerte migrasjonsløsningen.
3. Modularitet: Løsningen som programmeres bør interferere minst mulig med eksisterende kildekode i Project Voldemort, primært gjennom å utvikle det som et separat prosjekt.

4.3 Modeller

Her følger uformell dokumentasjon over dataevolusjonsmoculen DBUpgradinator og hvordan den skal oppføre seg i en datadreven webapplikasjon hvis dataelementer persisteres i en distribuert database. Dokumentasjonen er basert på 4+1-metamodellen Phillippe Kruchten presenterer i en artikkel fra 1995.

4.3.1 4+1 - stilen

Dokumentasjon av programvarearkitektur er en kommunikasjonskunst. Symbolene som tegnes på et lerret eller på en bildefil bærer ingen betydning eller mening i seg selv, hva de representerer er noe dokumentets forfattere og dets påtenkte publikum må være samstemte om. Videre er tydelige kommuniserende begrenset når det kommer til hvor mange konsepter de kan representere på ett og samme tidspunkt. En firkant, eller en boks, kan til eksempel ikke representere både en statisk fil med programkode i og samtidig en kjørende prosess.

Enkelte arkitekturdokumenter fokuserer for mye på ett enkelt aspekt ved datasystemet, eller tar ikke hensyn til ønsker og bekymringer til samtlige interessenter, de distinkte gruppene som leser dokumentet. Derfor foreslår Kruchten (1995) å organisere dokumentasjonen til en programvarearkitektur inn i en mengde av fire perspektiv (eng. viers) som hver for seg besvarer ett spesifikk interesseområde for applikasjonen.

Disse fire perspektivene kaller Kruchten (1995) for:

Det logiske perspektivet I kontekst av objekt-orientert programmering er dette en oversikt over objekter (teknisk sett prosesser/tråder) som eksisterer i primærminnet til datamaskinene i systemet

Prosessperspektivet Gir leseren en oversikt over detaljer angående samtidige operasjoner og synkronisering i systemets design

Det fysiske perspektivet Beskriver 1) koplingene mellom programvareelementer og maskinvareelementer i systemet og 2) systemets distribuerte naturBeskriver 1) koplingene mellom programvareelementer og maskinvareelementer i systemet og 2) systemets distribuerte natur

Utviklingsperspektivet Beskriver programvarens statiske organisering av dets kildekode i utviklermiljøet

Den primære oppgaven til det logiske perspektivet av arkitekturdokumentasjonen er å representere effektene de funksjonelle krav som stilles til datasystemet har på det. De funksjonelle krav har opprinnelse i domenet til problemet datasystemet angriper. Systemets viktigste abstrakte konstruksjoner, kalt objekter i programmets og klasser i kildekoden, kan knyttes til separate begreper inneholdt i domenet til problemet interessentene (de som leser arkitekturdokumentet) vil løse (Kruchten, 1995). Objekter utnytter egenskaper som arv/spesialisering, innkapsling og funksjonsmaskering gjennom grensesnitt for å realisere påkrevd funksjonalitet.

Logiske perspektiver inneholder gjerne klassediagrammer og sekvensdiagrammer skrevet med UML - syntaksen. Et klassediagram viser en oversikt over klassene og relasjoner mellom dem i et objekt-orientert program. Hvis applikasjonen er sterkt datadrevet, slik tilfelle gjerne er i kommersielle webapplikasjoner som netthandel-informasjonssystemer, er det hensiktsmessig å presentere et ER - eller et EER - diagram i det logiske perspektivet også.

Prosessperspektivet beskriver hvordan oppfører seg når det kjører og hvordan det kjører.

Prosessperspektivet reflekterer de ikke-funksjonelle krav, herunder kvalitetskrav, krav til systemoppførsel og begrensende krav, som stilles til datasystemet. Sentrale problemer som tas stilling til inkluderer samtidighet, feiltoleranse, distribusjon av arbeidslast, integritet, parallelt kjørende tråder og hvordan entiteter, klasser eller objekter fra det logiske perspektivet gjenspeiles i prosessperspektivet (Kruchten, 1995).

Her svarer et objekt fra det logiske perspektivet til en tråd underordnet en kjørende prosess i et operativsystem. Et høytilgjengelig system vil typisk modelleres slik at enhver slik prosess i systemet har en identisk tvilling som kan steppe inn og ta over sin partners arbeidslast skulle den svikte, også kjent som en reserveprosess, på engelsk kalt for ''spare".

Oppgaven til programvarearkitekturdokumentasjonens fysiske perspektiv er å definere sammenhengen mellom kildekode og maskinvaren (m.a.o. datamaskiner) som koden kjører på. Perspektivet er særskilt relevant for systemer der strenge, spesifikke krav stilles dets ytelse (gjennomstrømming i fore eksempel antall transaksjoner per sekund), tilgjengelighet (evnen til å handtere delvis eller fullstendig svikt) og pålitelighet (feiltoleranse).

Arkitekturen til et distribuert system består av et nettverk av uavhengige datamaskiner som prosesserer data, kalt *noder*. Derfor er det vesentlig å dokumentere hvilken rolle de abstrakte programvareelementer fra prosessperspektivet og utviklingsperspektivet spiller i forhold til de forskjellige fysiske noder (Kruchten, 1995).

Systemet har vanligvis forskjellige konfigurasjoner, en mengde parametere som bestemmer programvarens oppførsel under kjøretid. Én konfigurasjon kan gjelde for systemets testfase, en annen for når dets produksjonsmiljø, og et atter annet for vedlikeholds- eller utviklingsmiljø (Kruchten, 1995). De forskjellige konfigurasjoner illustreres ofte i hvert sitt distribusjonsdiagram, som også kan skisseres med UML-syntaks.

Utviklingsperspektivet tiltaler primært applikasjonens utviklere og vedlikeholdere, de som skal endre dets kildekode. Programmet inndeles i mindre enheter, kalt ''bibliotek" eller ''moduler", hvilket kan refereres til og benyttes av utviklere på tvers av store eller små prosjektgrupper. Disse modulene er organisert hierarkisk i forskjellige *lag*, det vil si at hver modul eksponerer et veldefinert programmeringsgrensesnitt som andre moduler organisert direkte over den kan kalle på uten å være involvert i bibliotekets kildekode.

Diagrammer innen utviklingsperspektivet viser derfor typisk import/eksport - forhold imellom forskjellige programvaremoduler, som i tur også kan samles sammen inn i pakker, som best kan beskrives som ''meta-moduler". Selv om utviklingsperspektivet ikke er komplett før samtlige programvareelementer i systemet er definert og navngitt kan det deklarere de regler som bestemmer kildekodeorganiseringens natur, deriblant eksponering av funksjoner via import/eksport av bibliotek, gruppering av moduler inn i pakker, og avgrensning av moduler. Utviklingsperspektivet er nyttig primært for ansvarsfordeling og å gjøre konkrete implementasjonsvalg som valg av språk, utviklingsverktøy og tredjepartsmoduler lettere.

Disse fire separate perspektivene kan også kombineres sammen i et femte perspektiv, kalt bruksscenarier, som oppsummerer systemets mengde av funksjonelle krav (Kruchten, 1995). Meningen bak dekomponeringen av programvarearkitekturdokumentasjonen er å belyse vesentlige interesseområder av systemet. Det er vesentlig å notere at perspektivmodellen i all hovedsak er en metamodell - en beskrivelse av sammenhengen mellom

konkrete modeller for programmet brukt i arkitekturens dokumentasjon.

Videre må det nevnes at samtlige perspektiver i metamodellen er valgfrie, enhver arkitekt står fri til å gjøre justeringer på sitt dokument i henhold til de krav som stilles systemarkitekturen som modelleres. DBUpgradinator modelleres med de tre første perspektivene i 4+1-metamodellen. Ettersom prosjekter er et enmannsarbeid er et sett av diagrammer som utgjør et eget utviklingsperspektiv ikke strengt nødvendig. Ett perspektiv som i kontekst av distribuerte applikasjoner er uunnværlig er det fysiske perspektivet.

4.3.2 Det logiske perspektiv

Figur 4.1 viser en oversikt over de ulike bestanddelene i DBUpgradinator. Figuren viser også DBUpgradinator sin logiske relasjon til REST-webapplikasjonen brukt til å teste dette migrasjonsverktøyet i. Det logiske perspektivet (eng. ”view”) beskriver også de ulike atomiske moduler webapplikasjonen består av.

Webapplikasjonen er inndelt i to nivå: Ett nivå for visningslogikken (”Frontend”) og ett for tjenerlogikken (”Backend”). Visningslogikken kjøres i et nettlesermiljø, det vil si at visningslogikken er en nettside som åpnes av en nettleser, som Microsoft Edge eller Mozilla Firefox. Nettleseren viser elementer som dikteres av kildekoden til siden nettleseren åpnet. ”Frontend” - miljøet i diagrammet markeres med en sirkel fordi det er et klientprogram, separat fra tjenerne, som kjører i en nettleser. Intensjonen med webapplikasjonen er at det er nettlesere som skal kjøre klientprogrammet. Frontendklienten oppretter og sender HTTP-forespørsler til det funksjonelle grensesnittet til applikasjonstjeneren.

Tjenerlogikken instansieres og kjøres innenfor et kjøretidssystem (eng. ”run-time system”), en spesialisert programvare som implementerer deler av kjøringsmodellen til et språk. En kjøringsmodell skisserer hvordan kode kjøres i et operativsystemet, for eksempel kan den være event-basert og tillate asynkron kjøring av kode, slik tilfelle er med kjøretidssystemet NodeJS. Fra et operativsystem sitt perspektiv anskues kjøretidssystemet som en selvstendig prosess. Det kommuniserer med operativsystemets grensesnitt på vegne av programmet det instansierer og kjører som en tråd underordnet seg selv. Kjøretidsmiljøet til Java heter for Java Runtime Environment (JRE).

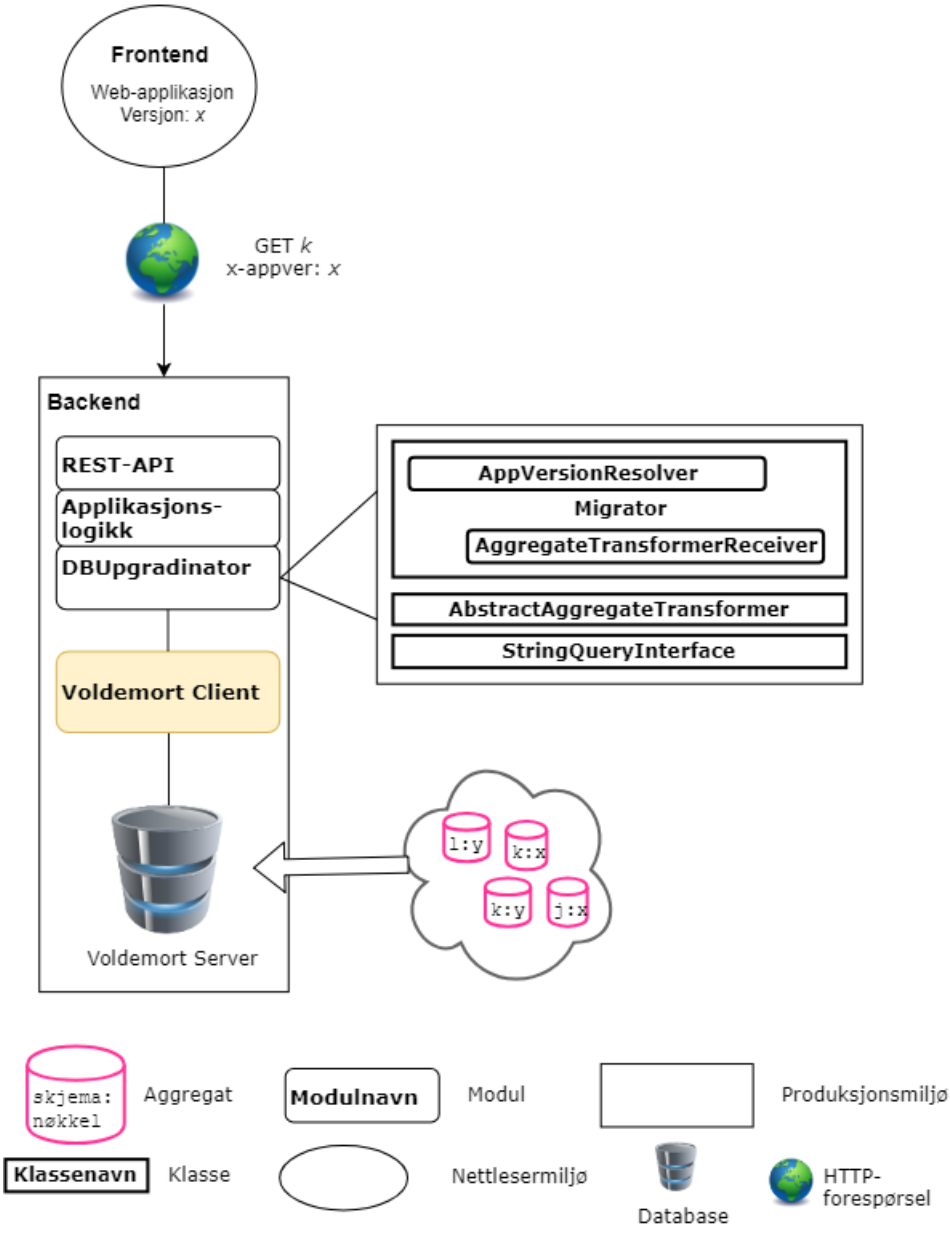
Figur 4.1 viser hvordan den logiske strukturen til webapplikasjoner, som DBUpgradinator er skrevet for, skal se ut. Applikasjonstjeneren består av følgende elementer:

REST-API er applikasjonsgrensesnittet visningslogikken kommuniserer med via HTTP. De fire typiske kommandoene grensesnittet betjener er HTTP-verbene `GET`, `PUT`, `POST` og `DELETE`.

Applikasjonslogikken til webapplikasjonen er kode som behandler HTTP-forespørsler og oppretter HTTP-responser. Den videresender både skjemanøkkel og aggregatnøkkel ned til DBUpgradinator, som handterer og besvarer spørringen.

DBUpgradinator - Programmet som migrerer hvert aggregat, asynkront fra brukerforespørsler.

VoldemortClient er klientprogrammet til Project Voldemort. Det utfører databasekall til N databasenoder. VoldemortClient er også ansvarlig for å kontrollere aggregatenes



Figur 4.1: Logisk oversikt over moduler i det typiske produksjonsmiljøet DBUpgradinator opererer i. Klassene som DBUpgradinator består av nevnes med navn.

vektorklokkeversjoner når den mottar spørringsresultater fra av hver av disse nodene før den returner et endelig spørreresultat. Ved skriveoperasjoner venter klienten på W bekreftelser før den besvarer applikasjonen. Ved leseoperasjoner venter klienten på R bekreftelser.

Voldemort Server - Én eller flere databaseprosesser som lagrer og henter data.

DBUpgradinator består av følgende elementer, som vist i boksen til høyre i figur 4.1:

Migrator er det objektet i pakken som utfører alle spørringer mot databasen, både for å betjene brukerforespørsler, og for å persistere migrerte aggregater, opprettet av **AbstractAggregateTransformer**-objektet på strengformat. Derfor er DBUpgradinator modellert som et eget lag i webapplikasjonens logiske stakk, plassert under appliaksjonslogikken og over databasegrensesnittet "VoldemortClient", i figur 4.1.

AggregateTransformerReceiver er en modul som mottar et **AbstractAggregateTransformer** - objekt via en I/O-operasjon, og holder rede på det i programminnet.

AppVersionResolver Oppgaven til denne modulen er å kople dataobjektets nøkkel i en innkommende HTTP-spørring (k) med applikasjonsversjonens nøkkel (x) på formen `k + ''':''' + x`

AbstractAggregateTransformer er et objekt som transformer aggregater som enten er mottatt fra en HTTP-forespørsel (PUT eller POST) eller mottatt fra databasen som et resultat av en GET-forespørsel. Selve transformasjonsfunksjonen er implemtert av applikasjonens utviklere eller vedlikeholdere.

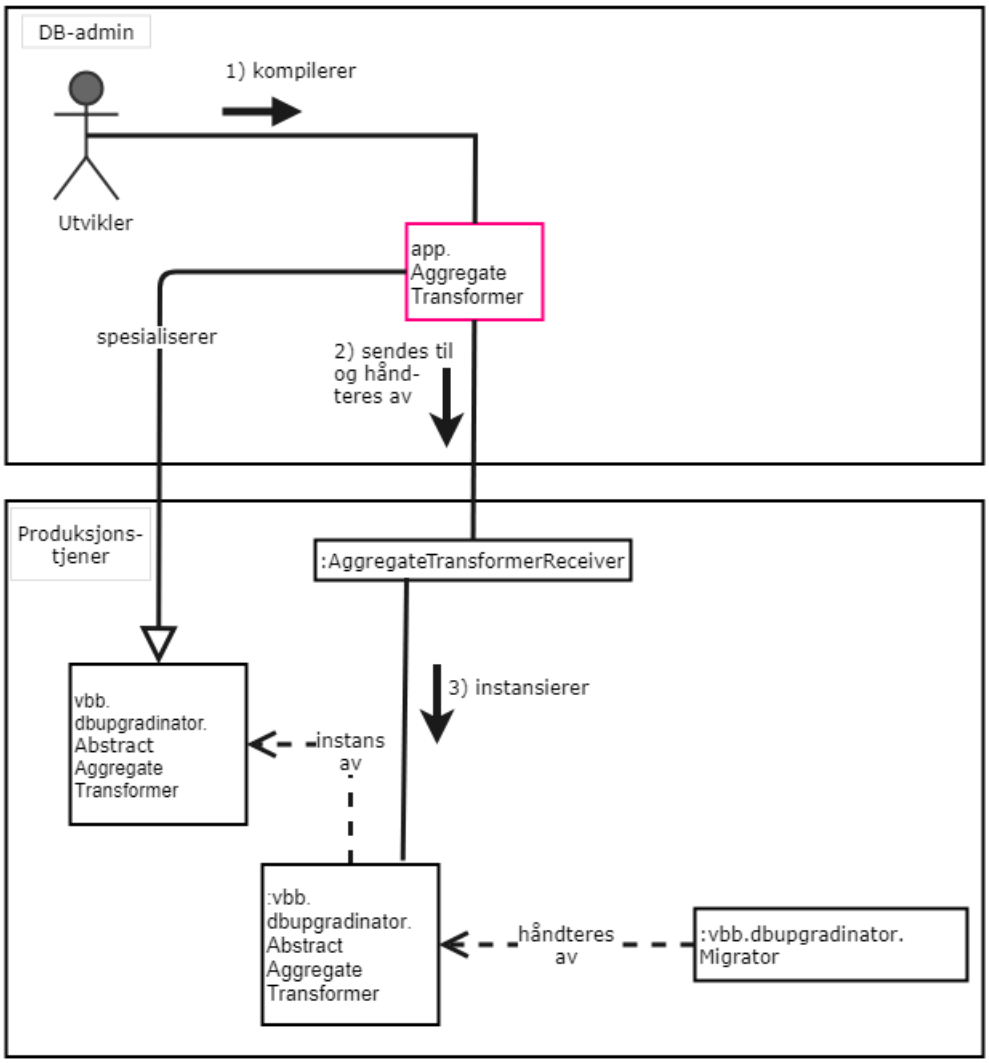
StringQueryInterface Modul som sender GET og PUT - spørringer for Migrator-objektet til "VoldemortClient".

Figur 4.2 forteller oss hvem det er som har ansvar for å skrive aggregattransformasjonsklassen, nemlig aktøren kalt "Utvikler", som har ansvar for migrasjonen av data i en databaseinstans. Akkurat som i KVolve, er det den enkelte applikasjonsutvikler som må definere aggregattransformasjonsfunksjonen fordi de både har kunnskap om domenet applikasjonen opererer i, og hvordan datamodellen til applikasjonen utvikler seg i takt med kildekodens endringer.

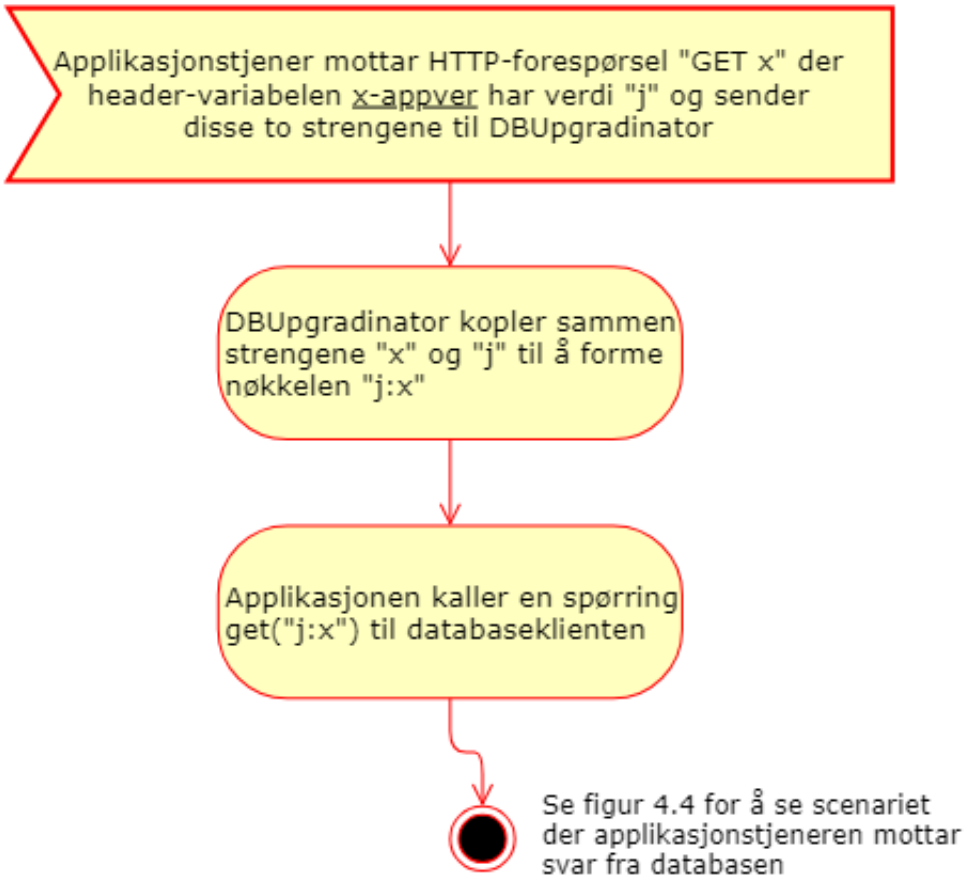
Etter å ha implementert transformasjonsklassen (markert med rosa omriss i kommunikasjonsdiagrammet) som spesialiserer **AbstractAggregateTransformer**, kompilerer utvikleren filen til en `.class` - fil og sender den til hver applikasjonstjener som kjører i systemet. AggregateTransformerReceiver-modulen leser inn alle oktettene i filen tjeneren mottar, inn en byte-matrise og skaper et objekt i minnet. Dette objektet opprettes som en instans av **AbstractAggregateTransformer**. Dette objektet blir dernest plassert i **Migrator**-instansens liste av transformatorer. I 4.2 er objektinstanser merket med kolon i navnet.

4.3.3 Prosessperspektivet

Aktivitetsdiagrammet i figur 4.3 beskriver systemets oppførsel når en GET-forespørsel fra en applikasjonstjener hvis skjemaversjon er j, mens produksjonsmiljøet er under oppgra-



Figur 4.2: Kommunikasjonsdiagram som illustrerer forholdet mellom en utvikler og en applikasjonstjener der DBUpgradinator benyttes for å migrere data levende.



Figur 4.3: Aktivitetsdiagram som illustrerer hvordan DBUpgradinator interfererer i applikasjonslogikken ved en `GET` - forespørsel før databaseklienten mottar spørringen.

dering av dataskjemaet fra versjon *j* til *k*, der *k* etterfølger *j*. Applikasjonsjeneren kaller på AppVersionResolvermodulen i DBUpgradinator for å sette sammen aggregatets nøkkel med dataskjemanøkkelen. Dernest kaller tjeneren på `get` - metoden til en StoreClient - instans. Tråden som utførte dette databasekallet vil da bli blokkert.

Aktivitetsdiagrammet i figur 4.4 beskriver systemets oppførsel når applikasjonstjenertråden mottar resultatet fra GET-forespørselen beskrevet i figur 4.3. DBUpgradinator migrerer aggregatet returnert fra GET-spørringen hvis det er behov for det. Når applikasjonen mottar et aggregat fra VoldemortClient, sender den både aggregatet og nøkkelen til DBUpgradinator.

Pakken kontrollerer om aggregatet er av den nyeste skjemaversjonen, det vil si at det ikke finnes noen transformatorobjekt hvis "nå"-versjon tilsvarer aggregatets skjemaversjon. Hvis dette er tilfelle, overlater DBUpgradinator eksekveringskontroll tilbake til applikasjonstjeneren direkte, som putter aggregatet inn i body-feltet til en HTTP-respons, som til slutt sendes tilbake til applikasjonsklienten.

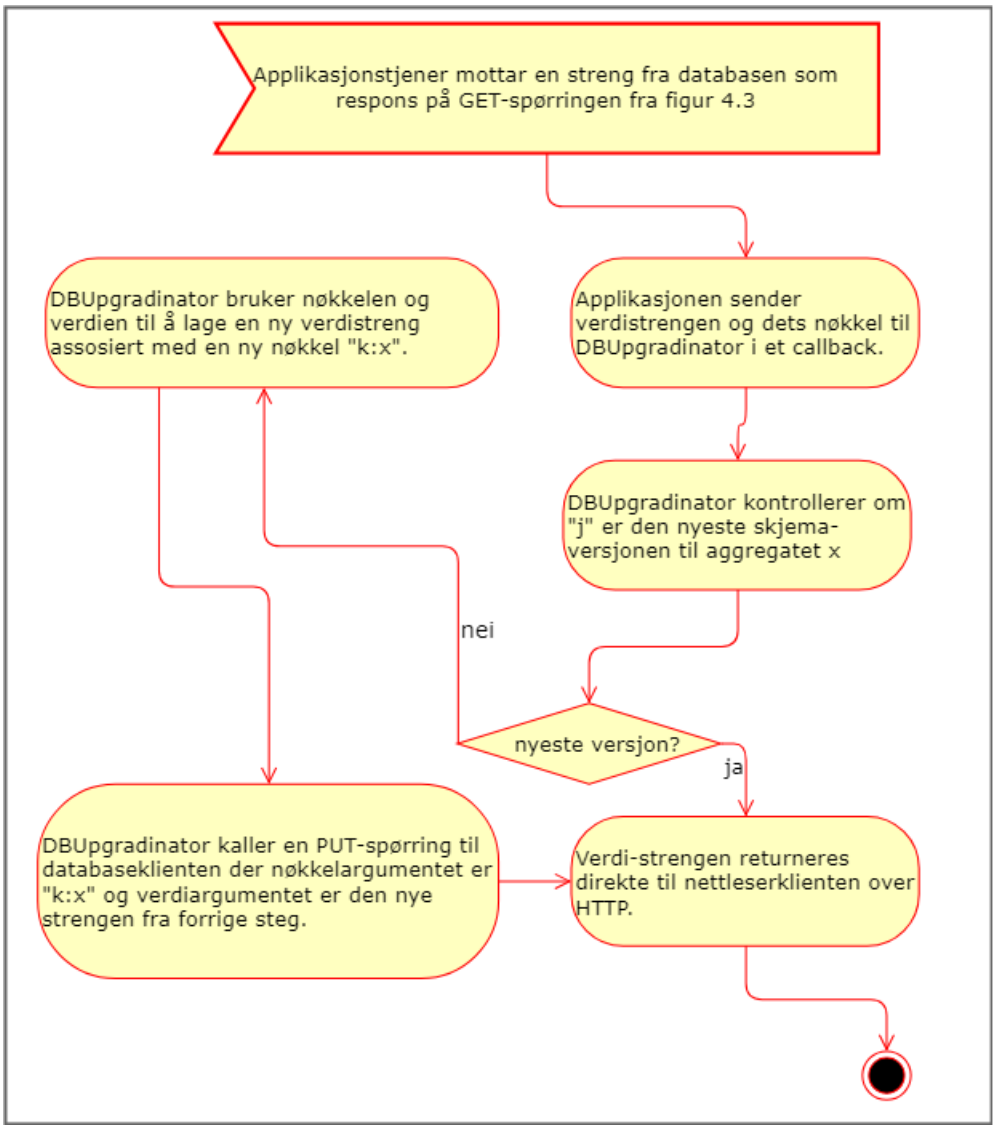
I det motsatte tilfelle sender programmet aggregatet inn i transformasjonsfunksjonen, som lager en ny streng utifra strengargumentet den fikk tilsendt. Ved hjelp av prosessen *AppVersionResolver* lager programmet en ny nøkkel der skjemaversjonsuffikset tilsvarer den strengen transformatorobjektet indikerer er navnet på det neste skjemaet. Dernest sendes en ny databasespørring der det migrerte aggregatet opprettes/oppdateres (put-kall) på den nye nøkkelen, fra en separat tråd enn den som betjener den brukerinitierte forespørselen. Med en gang databasespørringen til det migrerte aggregatet tilsendes StoreClient, returneres spørreresultatet til applikasjonstjeneren.

Aktivitetsdiagrammet i figur 4.5 beskriver webapplikasjonens oppførsel når en PUT-forespørsel fra en applikasjonstjener hvis skjemaversjon er *j*, mens produksjonsmiljøet er under oppgradering av dataskjemaet fra versjon *j* til *k*, der *k* etterfølger *j*.

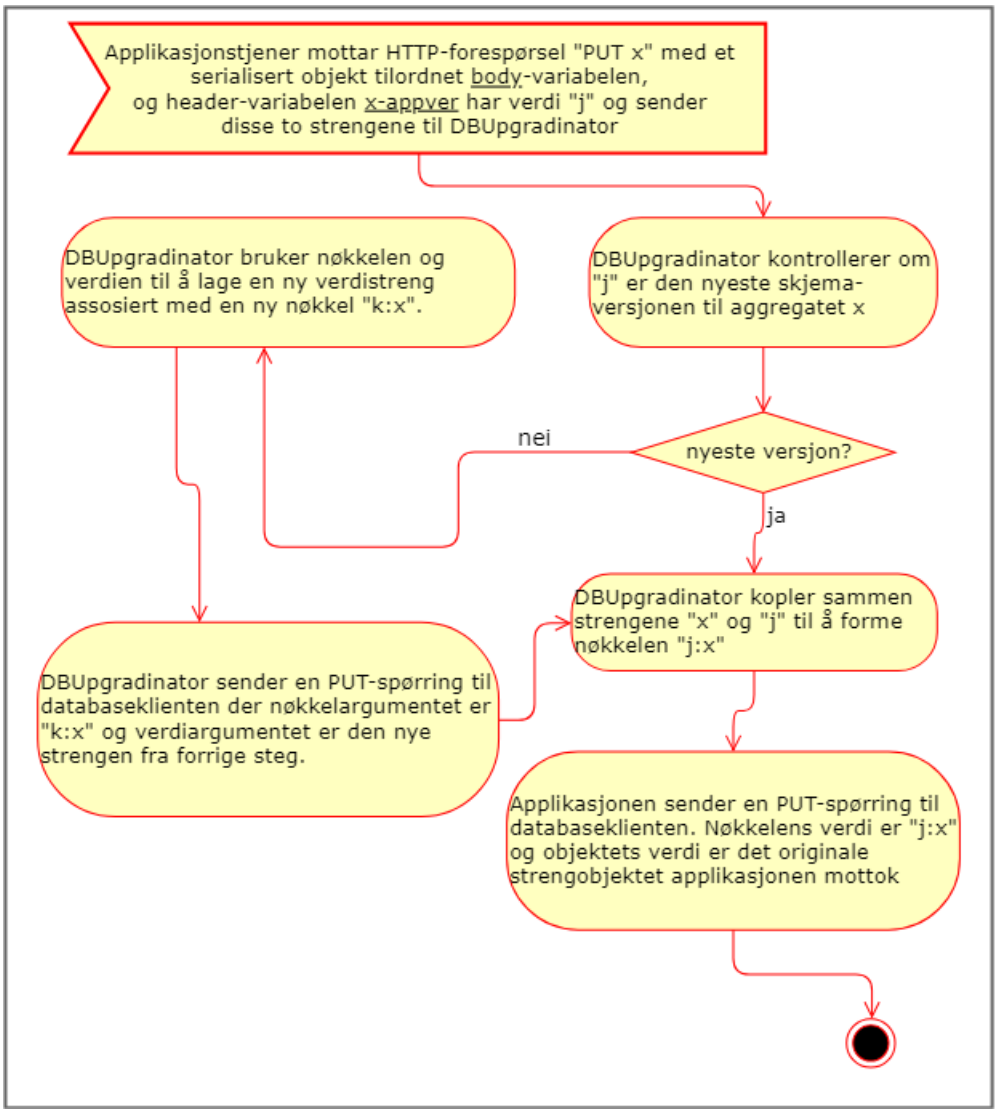
Forretningsordenen for en skriveforespørsel er omvendt av den vi så for leseforespørsler i figurene 4.3 og 4.4. Her blir DBUpgradinator kontaktet før databasespørringen sendes. Årsaken er at aggregatet ved PUT eller POST er gitt av klienten, sammen med både dets påtenkte skjema og dets statiske nøkkel. Igjen vil DBUpgradinator kontrollere skjemaversjonen, og asynkront migrere aggregatet ved å opprette en ny streng basert på det, hvis skjemanøkkelen som kommer fra forespørselen ikke er den nyeste. Etter å ha utført migrasjonen av det oppdaterte aggregatet og sendt resultatet til databaseklienten i form av et put-funksjonskall, hopper programmet tilbake inn i applikasjonslogikken, og sender et nytt put-kall til samme databaseklient.

Dette databasekallet gjøres for å oppdatere det objektet som i prinsippet er det samme aggregatet, men for det gamle skjemaet. Årsaken til at DBUpgradinator oppdaterer aggregatet for det nye skjemaet, samtidig som for det gamle, er at ved rullerende oppgradering av en klynge applikasjonstjenere har man en miks av tjenere som opererer med den gamle versjonen av tjenerprogrammet og tjenere som kjører den nye. Mengden av nye tjenerversjoner øker gradvis på bekostning av mengden av de gamle, naturlig nok, ved en rullerende applikasjonsoppdatering.

Når applikasjonen mottar en bekreftelse fra et put-kall opprettes en HTTP-respons som



Figur 4.4: Aktivitetsdiagram som illustrerer hvordan DBUpgradinator interfererer i applikasjonslogikken ved en `GET` - forespørsel etter at spørringen er ferdig.



Figur 4.5: Aktivitetsdiagram som illustrerer hvordan DBUpgradinator påvirker applikasjonens oppførsel ved en innkommende `PUT` - forespørsel.

informerer om hvorvidt skriveoperasjonen lyktes eller ikke. Responsen sendes tilbake til applikasjonsklienten over TCP/IP - forbindelsen.

4.3.4 Det fysiske perspektiv

De fire boksene i figur 4.6 refererer til fire forskjellige datasentre. Hver enkelt av de fire applikasjonsnodene gjestes av skyinfrastrukturleverandøren DigitalOcean, og alle er spredd utover fire forskjellige datasentre i London, Amsterdam (to av datasentra er lokaliserte her), og Frankfurt am Main.

I hvert datasenter betjenes én virtuell maskin, eller droplet, som DigitalOcean kaller dem. Selve maskinvaren blir beskrevet i detalj i delkapittel 5.3. Hver av de virtuelle maskinene kjører to separate programmer ("kjøretidsmiljø") som lytter på én nettverksport hver. Disse programmene er:

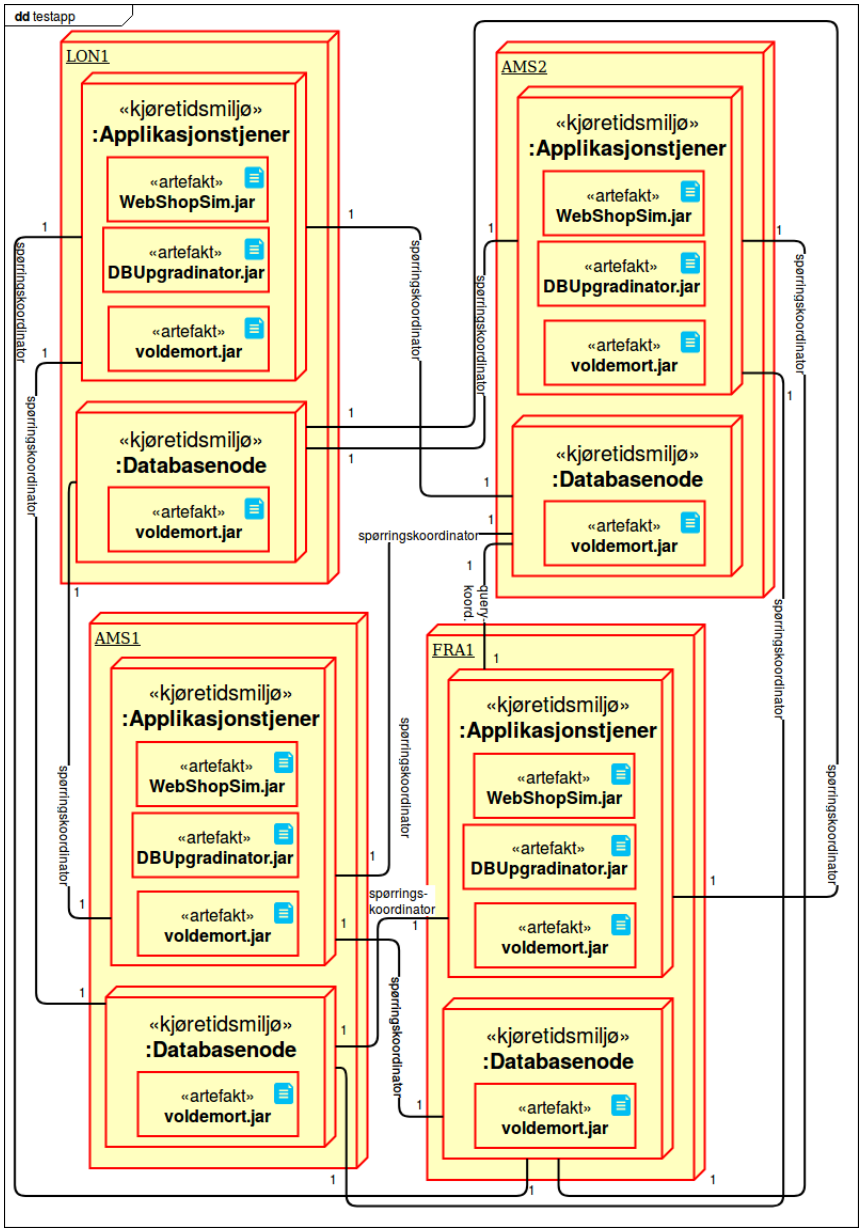
1. Webapplikasjonen, som i 4.6 benytter artefaktene WebShopSimulator.jar og DBUpgradinator.jar til å kjøre selve appliaksjonstjeneren; samtidig benytter prosessen voldemort.jar - filen for å instansiere en klient som replikerer hvert aggregat til tre databasenoder
2. Databasenoden, som lagrer og henter data på kommando fra Voldemort-klientprogrammene

Figur 4.6 viser også at hver kjørende instans av Voldemort-klienten fra applikasjonstjenerprogrammet har én replikeringsassosiasjon med hver av de tre databasenodene stasjonert på de andre virtuelle maskinene. Det er ikke nødvendigvis slik hash-strategien er konfigurert i databaseinstansen, figuren er ment å illustrere at klientprogrammet til Voldemort er del av applikasjonsprosessen, og det er den som utfører spørringer mot de enkelte databasenoder for å persistere og lese repliaker.

4.4 Implementasjon av DBUpgradinator

DBUpgradinator er skrevet i det objektorienterte programmeringsspråket Java. Java er et statisk, nominelt, og sterkt typet språk. Med nominelt menes at type-ekvivalens eller typekompabilitet baseres på eksplisitte deklarasjoner i kildekoden. I Java deklareres heltall representert ved fire bytes med nøkkelordet *int*. Med sterk typing menes at en type, enten det er en primitiv datatype, streng eller hjemmelaget klasse, alltid blir deklarert for ethvert objekt i kildekoden. At et språk er statisk typet betyr at en variabels type kontrolleres idet kildekoden kompileres, ikke under kjøring.

Som vist i figur 4.1, består DBupgradinator av tre klasser (firkant): **AbstractAggregateTransformer**, **Migrator** og **StringQueryInterface**. Migrator - klassen innehar tre forskjellige migrasjonsmetoder, og to moduler (avrundet firkant): *AppVersionResolver* og *AggregateTransformerReceiver*. I implementasjonen er disse to elementene henholdsvis representert ved funksjonene *getPersistedKey* og *aggregateTransformerReceiver*. Koden i



Figur 4.6: Deployment-diagram av webapplikasjonen DBUpgradinator testes i.

migrasjonsmetodene *getAndMigrateAggregate*, *putAndMigrateAggregate* og *postAndMigrateAggregate* er skrevet i henhold til aktivitetsdiagrammene 4.3, 4.4 og 4.5.

4.4.1 AbstractAggregateTransformer

Kodeoppføring 4.1: Klassen *AbstractAggregateTransformer*, hvis hovedansvar er å lage en ny streng ut fra et gitt strengargument i funksjonen *transformAggregate*, som programvareutvikleren selv må implementere.

```
1  package vbb.dbupgradinator;
2
3  public abstract class AbstractAggregateTransformer {
4     private String currentAppVersion;
5     private String nextAppVersion;
6
7     public AbstractAggregateTransformer(String currentAppVersion,
          String nextAppVersion) {
8        this.currentAppVersion = currentAppVersion;
9        this.nextAppVersion = nextAppVersion;
10    }
11
12    public String getNextSchemaVersion() {
13       return nextAppVersion;
14    }
15
16    public String getAppVersion() {
17       return currentAppVersion;
18    }
19
20    /**
21     * Both parameters are strings either returned from or going
           into a DB query
22     * @return String - the new aggregate based on the input
23     * @param val The value, which is a generic object
24    */
25    public abstract String transformAggregate(String val);
26 }
```

AbstractAggregateTransformer er en abstrakt klasse, som har to konstante feltvariable. Verdien på disse feltvariablene defineres når dens konstruktør kalles. Den ene tilstandsvariabelen indikerer skjemaversjonen til transformasjonsklassen mottar aggregater fra, og én som indikerer versjonen til det skjemaet som den transformerte verdien gjelder for. Klassen har én abstrakt metode, som implementeres i en subklasse. Denne subklassen blir programmert og kompilert til en binær `.class`-fil av den individuelle applikasjonsprogrammerer. Den eneste abstrakte metoden i klassen er *transformAggregate*, som påkalles når en forespørsel fra webapplikasjonens gamle versjon (som er under rullerende oppgradering) tilsendes databaseklienten, den påkalles både når klienten mottar data fra data-

lageret, etter at databaseklienten har flettet divergerende elementer; argumenter: key (fra DB), value (deserialisert), så vel som når klienten mottar data fra applikasjonen, altså fra forespørselen direkte.

Den sterke, statiske typekontrollen til Java pålegger én lei begrensning for implementasjonen av migrasjonsmodulen, angående AAT. Det har seg slik at to klasser er identiske hvis og bare hvis de leses inn av samme klasseinnlaster (eng. "class loader"), og har samme fulle navn (Lindholm and Yellin, 1999), i tillegg til å ineholde ekvivalent kildekode og like navn for alle variabler og metoder. Hvis de befinner seg i to forskjellige pakker er de ikke identiske.

Det ble opprinnelig forsøkt å implementere et eget klientprogram for DBUpgradinator som tillot utviklerne å sende komplierte `.class` - filer over TCP ved hjelp av

...Denne begrensningen handler om klassen utvikleren implementerer en subklasse for, **AbstractAggregateTransformer**. Sannsynligvis er denne klassen unødvendig, da man trolig kun trenger å bruke getMethod() - metoden fra reflection-biblioteket for å kalle på hver enkelt transformAggregate - metode.

4.4.2 Migrator

Migrator er en `public` klasse som eksponerer DBUpgradinator sin kjernefunksjonalitet til programvareutvikleren. Dens tilstand inkorpurerer aggregatets objekttype - det vil si domeneklassen til aggregatet, som sendes inn i transformasjonsfunksjonen til AbstractAggregateTransformer-klassen som verdi-argument. Transformasjonsfunksjonen blir overskrevet av den implementerte klassen. En instans av Migrator spør etter data på forespørsler tilsendt tjeneren med HTTP-protokollen, og persisterer samtidig migrerte aggregater returnert av transformasjonsfunksjonen til instansen av AbstractAggregateTransformer.

aggregateTransformerReceiver er en funksjon, som kjøres i en separat tråd idet konstruktøren til Migrator kalles, og er følgelig ikke del av en ordinær spørrings livsløp. Kodeobjektet er en funksjon som mottar AggregateTransformer-objekter og instansierer dem som instanser av den abstrakte superklassen **AbstractAggregateTransformer**. og holder rede på dem i versjonsrekkefølge i en privat liste. Funksjonen har også ansvaret for å påkalle transformasjonsmetoden til hvert objekt.

Oppgaven til funksjonen *getPersistedKey* er å kople dataobjektets nøkkel i en innkommende HTTP-spørring (k) med applikasjonsversjonens nøkkel (x) på formen `k + ":"+ x`

En instans av **Migrator** har en lenket liste av instanser av **AbstractAggregateTransformer**, som utviklerens egenskrevne klasse AggregateTransformer (markert med rosa omriss i figur 4.2) arver fra. Det er Migrator-instansen som har ansvar for å utlede skjemaversjonskausalitet med denne lenkede listen og transformere innkommende aggregater, fra databasen så vel som applikasjonen, ved behov.

Migrasjonssystemet forventer at utvikleren skriver en `.java` - fil med en klasse som spesialiserer **AbstractAggregateTransformer**.

Kodeoppføring 4.2: Migrator-klassen i DBUpgradinator

```
package vbb.dbupgradinator;

import java.lang.reflect.Constructor;
import java.net.URI;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.HashMap;
import java.util.concurrent.CompletableFuture;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Migrator {
   private static final Logger logger =
       LogManager.getLogger("DBUpgradinator");
   private final String className;
   private final HashMap<String, AbstractAggregateTransformer>
       transformers = new HashMap<>(4, (float) 0.95);

   public Migrator() {
      // Forenkling av kildekode for testingens skyld
      this.className = "UserAggregateTransformer";
      // Sets up separate process which listens actively on the
          server socket
      new Thread( this::aggregateTransformerReceiver ).start();
   }

// Se kodelisting 4.2 til og med 4.7 for resten av kildekoden til
    Migrator
// ...

}
```

Kodeoppføring 4.3: Indre klasse brukt til å definere en transformasjonsklasse som arver konstruktør fra **AbstractAggregateTransformer**.

```
private class AggregateTransformerLoader extends ClassLoader {
   AggregateTransformerLoader() { super(); }
   final Class<?> createClass(String name, byte[] b) throws
       ClassFormatError {
      return super.defineClass(name, b, 0, b.length);
   }
}
```

Kodeoppføring 4.4: Metoden *addTransformer*, og metoden *aggregateTransformerReceiver*, som kjøres i en separat tråd idet **Migrator**-konstruktøren kalles.

```
// Addition of transformer class to the HashMap
private void addTransformer(AbstractAggregateTransformer t) {
    this.transformers.put(t.getAppVersion(), t); }

// Separate process that actively listens for new classes that
    extend AAT
private void aggregateTransformerReceiver() {
   // Define ClassLoader instance
   AggregateTransformerLoader loader = new
       AggregateTransformerLoader();
   while ( this.transformers.size() < 1 ) {
      try {
         Path path = Paths.get( new URI("./classes/" +
             this.className + ".class"));
         // Use Files.exists - condition
         if (Files.exists(path)) {
            byte[] classData = Files.readAllBytes(path);
            Class<?> c = loader.createClass(this.className,
                classData);
            Constructor cons = c.getConstructor(String.class,
                String.class);
            AbstractAggregateTransformer tran =
                (AbstractAggregateTransformer) cons.newInstance(
                "x", "y" );
            // What to do with the transformer object: Add it to
                the AAT list
            this.addTransformer(tran);
         }
      } catch (Exception e) {
         this.logger.error("An error occurred in
             AggregateTransformerReceiver ", e);
      }
   }
}
```

Kodeoppføring 4.5: Funksjon som oppfyller rollen tl objektet ”AppVersionResolver” fra figur 4.1.

```
// Used by the application instance to get the persisted key in
    DB - always run before a query
private String getPersistedKey(String aggregateKey, String
    schema) {
   return aggregateKey + ":" + schema;
}
```

Kodeoppføring 4.6: Metode for håndtering av GET-spørring i Migrator.

```
public String getAndMigrateAggregate(StringQueryInterface db,
    String aggregateKey, String schema) {
   String key = this.getPersistedKey(aggregateKey, schema);
   String aggregate = db.query(key); // Blocking DB op
   AbstractAggregateTransformer spec =
       this.transformers.get(schema);
   if (null != spec) {
      String nextSchema = spec.getNextSchemaVersion();
      String nextKey = this.getPersistedKey(aggregateKey,
          nextSchema);
      CompletableFuture.supplyAsync(() -> {
         if (!aggregate.equals("")) {
            // Migrate the aggregate having the key _key to
                another with the key _nextKey using spec
            String migratedAggregate =
                spec.transformAggregate(aggregate);
            Exception fail = db.persist(nextKey,
                migratedAggregate);
            if (fail == null) { return true; }
            this.logger.error("Error during migration from " +
                key + " to " + nextKey + ":\n"+ fail.toString());
         }
         return false;
      }).thenAccept((b) -> {
         if (b) {
            this.logger.info("Migrated aggregate with key " + key
                + " to " + nextKey);
         }
      });
   }
   return aggregate;
}
```

Kodeoppføring 4.7: Metode for håndtering av POST-spørring i Migrator.

```
private static boolean logUpdateResult(String key, Exception ex) {
   if (ex == null) {
      this.logger.info("Persisted key " + key);
      return true;
   } else {
      this.logger.error("Error during persisting" + key + ": "+
          ex.toString());
      return false;
   }
}

public boolean migrateAndPostAggregate(StringQueryInterface db,
    String aggregateKey, String schema, String ag) {
   AbstractAggregateTransformer spec =
```

```
        this.transformers.get(schema);
13    String key = this.getPersistedKey(aggregateKey, schema); //
        This is the key used by the application
14    if (null != spec) {
15       String nextSchema = spec.getNextSchemaVersion();
16       String nextKey = this.getPersistedKey(aggregateKey,
             nextSchema);
17       String migratedAggregate = spec.transformAggregate(ag);
18       CompletableFuture.supplyAsync(() -> db.persist(nextKey,
             migratedAggregate)).thenAcceptAsync((fail) -> {
19          if (fail == null) {
20             this.logger.info("Migrated aggregate with key " + key
                   + " to " + nextKey);
21          } else {
22             this.logger.error("Error during migration from " +
                   key + " to " + nextKey + ":\n"+ fail.toString());
23          }
24       });
25    }
26    Exception ex = db.persist(key, ag);
27    return logUpdateResult(key, ex);
28 }
```

Kodeoppføring 4.8: Metode for håndtering av PUT-spørring i Migrator.

```
1  public boolean migrateAndPutAggregate(StringQueryInterface db,
       String aggregateKey, String schema, String ag) {
2     String key = this.getPersistedKey(aggregateKey, schema);
3     AbstractAggregateTransformer spec =
          this.transformers.get(schema);
4     if (null != spec) {
5        String nextSchema = spec.getNextSchemaVersion();
6        String nextKey = this.getPersistedKey(aggregateKey,
             nextSchema);
7        CompletableFuture.supplyAsync(() -> db.persist(nextKey,
             spec.transformAggregate(ag))).thenAccept((fail) -> {
8           if (fail == null) {
9              this.logger.info("Migrated aggregate with key " + key
                   + " to " + nextKey);
10          } else {
11             this.logger.error("Error during migration from " +
                   key + " to " + nextKey + ":\n"+ fail.toString());
12          }
13       });
14    }
15    Exception ex = db.persist(key, ag);
16    return logUpdateResult(key, ex);
17 }
```

4.4.3 StringQueryInterface

Kodeoppføring 4.9: StringQueryInterface, grensesnittet Migrator-klassen bruker til å kjøre databasespørringer med.

```
1 package vbb.dbupgradinator;
2
3 public interface StringQueryInterface {
4    public String query(String key);
5    public Exception persist(String key, String aggregate);
6 }
```

StringQueryInterface er en liten mengde med funksjoner utfører spørringer for Migrator-instansen. Kodeoppføring 4.9 viser hele dette grensensitt, som eksponerer to metoder: *get* og *put*. *put* - metoden forventes å returnere en instans av **Exception**. Utvikleren forventes å skrive en separat klasse som implementerer dette grensesnittet.

Kapittel 5

Evaluering av datamodellevolusjonsløsning

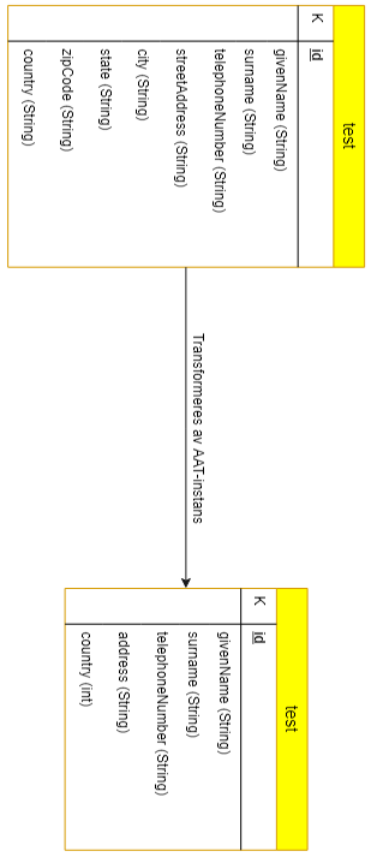
Dette kapitlet presenterer og diskuterer resultatene av testingen av datamodellevolusjonsløsningen DBUpgradinator, og sammenlikner datamodellevolusjonsløsningens kvaliteter med øvrige systemer presentert i kapittel 3. I delkapittel 5.2 vil testprosessen beskrives, herunder inkludert konfigurasjon av Voldemort-klyngeinstansen; hvordan testprogrammet WebshopSimulator er bygd opp; innsamling av testdata ved hjelp av loggeprogrammet Log4j og diverse tellere innebygd i simulasjonsprogrammet.

5.1 Testscenario

Her følger en beskrivelse av webapplikasjonen migrasjon med DBUpgradinator blir testet på. Nettbutikken ”WebShop” benytter nøkkelverdilageret Project Voldemort for å lagre et register over dens kunder. Kundene bor stort sett enten i Storbritannia eller i Australia.

5.2 WebshopSimulator

Webapplikasjonen som DBUpgradinator testes på, kalles for *WebshopSimulator*. Dens arkitektur følger både det logiske perspektivet, prosessperspektivet og den fysiske dittoen skissert i figurer 4.1, 4.2, 4.3, 4.4, 4.5, og 4.6. Testapplikasjonen etterkommer også systembegrensningene opplistet i delkapittel 4.2.2. Dette delkapitlet skisserer strukturen til testapplikasjonens kildekode, dets avhengigheter, og hvordan tjenerprogrammet blir kompilert til en kjørbar .jar - fil.



Figur 5.1: Datamodeller for aggregatene til skjemaversjon ”x” og dets etterfølger, ”y”.

5.2.1 Simulasjon av brukerforespørsler

Frontend-delen av webapplikasjonen er skrevet i Javascript, og kjøres som et separat klientprogram. Dette skriptet kjøres i kjøretidsmiljøet NodeJS. Klientprogrammet kalles for *WebAppSimulator*. Dets hovedoppgave simulerer en kontinuerlig serie med forespørsler som jevnfordeles blant applikasjonsinstansene. Programmets funksjonalitet kan i korte trekk inndeles i tre satser:

1. Programmet sender klyngen av tjenere en serie av POST-forespørsler, slik at nye aggregater lagres på hver av de tomme nodene i databasen. Totalt 8 GB i serialiserte JSON-objekter sendes applikasjonsinstansene og Voldemort-klientene. Samtidig holder en egen loggeprosess styr på ID-verdiene som genereres av tjenerne som tilsendes denne simulerte klienten
2. Etter at den første satsen er ferdig, terminerer programmet. Da må applikasjonsloggen, generert av frontendsimulatoren, transformeres til en liste av unike IDer. Til å oppnå dette formålet brukes tekstbehandlingsverktøyet Vim, en kommandolinjeapplikasjon som kan redigere alle linjer i loggen samtidig ved hjelp av regulære uttrykk og dets kraftige kommandolinjesyntaks. Dernest må klientsimulasjonsprogrammet redigeres slik at listen av aggregat-IDer blir lest inn fra korrekt filsti, og slik at funksjonen som kjører den tredje satsen blir kjørt neste gang programmet blir startet opp fra kommandolinjen
3. Programmet sender en blanding av 30 prosent PUT og 70 prosent GET - forespørsler til databasenodene, i sum én HTTP-forespørsel per ID i listen som ble oppretttet i forrige steg.

For å generere data til POST - forespørslene i steg 1 og PUT - forespørslene i steg 3, brukes en kommeaseparert seed-fil som inneholder fornavn, etternavn, postkode, post

[1]

[1] Navn - og addresser i seed-filen stammer fra gratistjenesten Fake Name Generator, som tilbyr inntil 100 000 navn og addresser i en kommaseparert fil per bestilling. URL: `https://www.fakenamegenerator.com/order.php`

Kapittel 6

Konklusjon og videre arbeid

Dette kapitlet vil oppsummere masteroppgavens formål og hvordan det har blitt oppnådd. Oppfyllelse av målene satt i kapittel 1.2 blir vurdert.

I dette prosjektet ble en programvaremodul, kalt DBUpgradinator, implementert i språket Java. Denne modulen er skrevet for automatisk å migrere persisterte aggregat i en NoSQL - database på lat vis. Modulen abstraherer bort databaselogikk og gjemmer den bak et spørringsgrensesnitt. På dette viset er den ikke låst til en spesifikk databasepakke. Modulen er testet i en Java-webapplikasjon der Project Voldemort tjener som databasehåndterer.

Testresultatene fra kapittel 5 demonstrerer at DBUpgradinator kan brukes til å migrere-re JSON-serialiserte dataobjekter til én versjon til den neste av en webapplikasjon, uten at noen noder blir slått av. Man kan derfor fastslå at programvaremodulen oppfyller de funksjonelle krav opplistet i kapittel 4.2.

Modulens største styrke er at den, gitt de antakelser presentert i kapittel 4, kan regulere versjonsmiks mellom skjemaer og applikasjonsinstanser, fordi modulens kildekode forventer at applikasjonsutviklere navngir deres implisitte skjema. Modulen benytter skjemanavn som del av nøkkelen til aggregatene. Følgelig må applikasjonsinstansen spesifisere navnet på skjemaversonen den opererer på, i tillegg til aggregatets ID for å lese dets data. Modulens kildekode viser også at hvordan støtte for levende oppdatering av en aggregatoritentert datamodell implementeres i programkode avhenger av programvarens arkitektur.

Videre påfører datamigrasjonsmodulen, gjennom Migrator - klassen, én ekstra asynkron PUT - spørring for hver spørring på et aggregat i et gammelt skjema. Denne ekstra operasjonen enten oppretter eller oppdaterer en nyere utgave av aggregatet. Den nye utgaven er en streng av tekst, utledet og opprettet av en aggregat-transformasjonsklasse. Slik kan endringer som gjøres i en gammel applikasjonsinstans også kunne leses av en ny applikasjonsinstans.

Ulempen med denne egenskapen er at skriveoperasjonen forårsaket av Migrator-instansen

kan sammenfalle med brukerinititerte transaksjoner i en oppgradert applikasjonstjener, og derav skape flettekonflikter utvikleren må ta høyde for i sin egen kildkeode. Sett fra applikasjonens perspektiv vil databaselaget i praksis yte færre brukerinitierte transaksjoner, det vil si PUT eller GET - operasjoner, per tid.

Alternativt kunne implementasjonen utføre én ekstra GET - spørring på den nye aggregatnøkkelen for å kontrollere at aggregatet ikke allerede er migrert til neste skjemaversjon. Imidlertid vil databasen utsettes for mange bomsøk når applikasjonen gjennomgår en rullerende oppgradering. Følgelig vil gjennomstrømmings-ytelsen ytterligere når hver brukerinitiert spørring påkrever to ekstra spørring når applikasjonen og dets skjema oppgraderes.

Det er få artikler å finne i indeksen til Google Scholar på levende datamigrasjon av NoSQL-baserte applikasjoner. Én årsak til dette er at datamigrasjon er en spesifikk problemstilling den enkelte applikasjonsutvikler selv må ta stilling til, og løse innenfor rammene satt av egne systemkrav. I likhet med versjonskonflikter i Dynamo/Voldemort, er det utvikleren som har kjennskap til applikasjonens domene. Derfor er det kun vedkommende som kan avgjøre hvilke data som skal bevares ved en flettekonflikt. Applikasjonens utviklere er best rustet til å vite hvordan et aggregat skal migreres fra den gamle tilstanden av datamodellen til den neste. DBUpgradinator demonstrerer i testapplikasjonen at dette er tilfelle. Denne problemstillingen har ikke vært spesielt interressant fra et akademisk synspunkt, primært fordi det allerede finnes programvaremoduler som kan migrere dataobjekter i levende tjenerinstanser, til eksempel *cassandra-migration-tool-java*.

DBUpgradinator pålegger webapplikasjonsutvikleren å skrive én separat klasse for hver oppdatering av applikasjonen. Denne øvelsen kan bli svært tidkrevende hvis programlogikken gjennomgår omfattende endringer. Disse kommandoene kan brukes til å definere transformasjonsfunksjoner på ett spesifikt skjemaformat, for eksempel JSON eller Avro. Formålet bak en slik spesialutformet syntaks er å strømlinjeforme prosessen med å definere transformasjonsfunksjoner. Dette programmet kan da erstatte den tungvinte prosessen der en programmer deifnerer en subklasse av AbstractAggregateTransformer, distribuerer en `.java` - fil der nevnte subklasse er definert til hver av applikasjonstjenerne, og kompilerer en `.class` på hver av dem. Programmet bør kunne kjøres i et skall fra enhver applikasjonsinstans der DBUpgradinator er installeres. Dette konsollgrensesnittet vil være en vesentlig forbedring fra den nåværende kildekoden. Saur et al. (2016) nedfeller et liknende mål for KVolve i sine konklusjoner.

For enkelhets skyld har implementasjonen av DBUpgradinator støtte for kun ett serialiseringsformat, JSON. I testapplikasjonen har Voldemort serialisert med `string`-formatet, det vil si at databasen ikke har noe formening om strukturen til hvert persisterte aggregat. Voldemort har et eget serialiseringsformat, basert på Apache Avro, som kan migrere data. Det ville være interressant å undersøke hvilken datamigrasjonsmetode som gjør webapplikasjonen mest tilgjengelig for betjening av brukersorespørsler under datamigrasjon.

Bibliografi

Bailis, P., Venkataraman, S., Franklin, M., Hellerstein, J., Stoica, I., August 2014. Quantifying eventual consistency with pbs. Communications of the ACM 57 (8), 93–102.

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.

Codd, E. F., 1971. Normalized data base structure: A brief tutorial. In: Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control. SIGFIDET '71. ACM, New York, NY, USA, pp. 1–17.
URL http://doi.acm.org/10.1145/1734714.1734716

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.

Dumitras, T., Narasimhan, P., 2009. No downtime for data conversions: Rethinking hot upgrades. Tech. rep., Technical Report CMU-PDL-09-106, Carnegie Mellon University.

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.

Elmasri, R., 2016. Fundamentals of database systems.

George, L., 2011. HBase: the definitive guide: random access to your planet-size data. "O'Reilly Media, Inc.".

Gobec, M., Bozic, N., May 2015. cassandra-migration-tool-java/readme.md.
URL https://github.com/smartcat-labs/cassandra-migration-tool-java/blob/develop/README.md

Hauer, P., Nov 2015. Databases as a challenge for continuous delivery.
URL https://blog.philipphauer.de/databases-challenge-continuous-delive

Hicks, M., Nettles, S., Nov. 2005. Dynamic software updating. ACM Trans. Program. Lang. Syst. 27 (6), 1049–1096.
URL http://doi.acm.org/10.1145/1108970.1108971

Kreps, J., Mar 2009. Project voldemort.
URL http://www.project-voldemort.com/voldemort/

Kruchten, P., November 1995. The 4+1 view model of architecture. Software, IEEE 12 (6), 42–50.

Lindholm, T., Yellin, F., 1999. The javatm virtual machine specification - concepts.
URL https://docs.oracle.com/javase/specs/jvms/se6/html/Concepts.doc.html

Lowell, D. E., Saito, Y., Samberg, E. J., 2004. Devirtualizable virtual machines enabling general, single-node, online maintenance. ACM SIGARCH Computer Architecture News 32 (5), 211–223.

MariaDB, 2017. Json data type.
URL https://mariadb.com/kb/en/library/json-data-type/

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.

Pepitone, J., Dec 2010. Why attackers can't take down amazon.com.
URL http://money.cnn.com/2010/12/09/technology/amazon_wikileaks_attack/

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

Saur, K., Dumitraş, T., Hicks, M., Oct 2016. Evolving nosql databases without downtime. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 166–176.

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.