



DBUpgradiator

Levende migrasjon av NoSQL-datamodeller

Vegard Bjerkli Bugge

Master i datateknologi

Innlevert: juni 2018

Hovedveileder: Svein Erik Bratsberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknologi og informatikk

Sammendrag

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 tjenestener, 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 migrasjon av delvis strukturerte datamodeller i høytligjengelige systemer uten nedetid, ved å utvikle en separat programvaremodul. 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

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 perform rolling upgrades with on the web application.

Forord

Min masteroppgave presenterer et modulært programvarebibliotek som automatiserer oppdatering av semistrukturerte datamodeller i distribuerte, aggregatororienterte 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 L^AT_EX, 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, 11. juni 2018

Vegard Bjerkli Bugge

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

Innhold

Sammendrag	i
Abstract	ii
Forord	iii
Innholdsfortegnelse	vii
Liste over tabeller	ix
Liste over figurer	xii
Liste over kodeoppføringer	xiii
Forkortelser	xiv
1 Introduksjon	1
1.1 Bakgrunn	1
1.2 Oppgavens problemstilling og mål	3
1.3 Oppgavens struktur	4
2 Teori	7
2.1 Den relasjonelle datamodellen	7
2.1.1 Impedansproblemet (Impedance mismatch problem)	8
2.1.2 Hush	10
2.2 Den aggregatororienterte datamodellen	12
2.2.1 Design av aggregatmodeller	14
2.2.2 Nøkkelverdimodellen	16
2.2.3 Dokumentmodellen	17
2.2.4 Kolonnefamiliemodellen	18
2.3 Amazon Dynamo	19

2.3.1	Bakgrunn for artikkelens arbeid	20
2.3.2	Om Dynamos arkitektur og Amazons implementasjon	22
2.4	Project Voldemort	26
2.4.1	Støttede operasjoner	27
2.4.2	Voldemorts egenskaper sammenliknet med RDBMS	27
2.4.3	Konsistenskontroll og versjonering	28
2.4.4	Serialisering av dataobjekter	29
2.4.5	Lastbalansering av dataobjekter	29
3	Relatert arbeid	33
3.1	Høytilgjengelige webapplikasjoner	33
3.1.1	Tilgjengelighet	34
3.1.2	Versjonskappløp i distribuerte systemer	34
3.1.3	Dynamisk programvareoppdatering (DSU)	35
3.1.4	Nedetid forårsaket av vedlikeholdsarbeid	37
3.2	Kontinuerlig leveranse	38
3.2.1	Skjemaendringer og datamigrasjon i NoSQL-databaser	39
3.2.2	Migrasjonsverktøy for Cassandra	40
3.3	EBR	41
3.3.1	Løsning	42
3.3.2	Diskusjon	43
3.4	Imago	44
3.4.1	Løsning	44
3.4.2	Diskusjon	45
3.5	KVolve: Evolusjon av datamodeller uten nedetid	45
3.5.1	Løsning	46
3.5.2	Evaluering av løsning	47
3.5.3	Diskusjon	48
4	Levende oppgradering av aggregatororienterte datamodeller	49
4.1	Inspirasjoner fra KVolve	49
4.2	Krav til implementasjon av levende datamigrasjonsløsning	51
4.2.1	Antakelser	51
4.2.2	Funksjonelle krav	53
4.2.3	Kvalitetskrav	53
4.3	Modeller	54
4.3.1	4+1 - stilen	54
4.3.2	Det logiske perspektiv	56
4.3.3	Prosessperspektivet	60
4.3.4	Det fysiske perspektiv	64
4.4	Implementasjon av DBUpgradinator	64
4.4.1	AbstractAggregateTransformer	66
4.4.2	Migrator	67
4.4.3	StringQueryInterface	72
5	Evaluering av evolusjonsløsning av datamodeller	73

5.1	Testscenario	73
5.2	WebshopSimulator	77
5.2.1	Simulasjon av brukerforespørsler	77
5.2.2	Programvare på tjenerne	78
5.2.3	Konfigurasjon av Voldemort-instans	80
5.2.4	Testmiljø	83
5.3	Testresultater	83
5.3.1	Test 1: Oppretting av aggregater, uten migrasjon	83
5.3.2	Test 2: Lesing, oppdatering, og oppretting av aggregater, med migrasjon	86
5.3.3	Test 3: Oppdatering av aggregater, med migrasjon	87
5.4	Diskusjon	89
5.4.1	Om flertrådede programvaresystemers blokkerende natur	89
5.4.2	Om konfigurasjonen av Voldemort-tjenerne	89
5.4.3	Erfaringer fra test 2	92
5.4.4	Erfaringer fra test 1 og 3	93
5.4.5	DBUpgradinator og KVolve	93
5.4.6	Om <code>getAndMigrateAggregate</code> - metoden i DBUpgradinator	94
5.4.7	Evaluering av kravoppfyllelse	94
6	Konklusjon og videre arbeid	97
6.1	Om DBUpgradinator	97
6.2	Akademisk interesse rundt levende datamigrasjon	98
6.3	Implementere DBUpgradinator som en valgfri komponent av Voldemort	98
6.4	Implementere et kommandolinjegransesnitt for DBUpgradinator	99
	Bibliografi	101

Tabeller

5.1	Oversikt over virtuelle datamaskiner brukt under testing av DBUpgradiator.	83
5.2	Oppsummering av frontendapplikasjonsloggen fra test 1.	84
5.3	Persisterte aggregater hos databasenodene i testmiljøet under test 1.	84
5.4	Oppsummering av frontendapplikasjonsloggen fra test 2.	86
5.5	Persisterte aggregater hos databasenodene i testmiljøet under test 2.	86
5.6	Oppsummering av frontendapplikasjonsloggen fra test 3.	87
5.7	Persisterte aggregater hos databasenodene i testmiljøet under test 3.	87

Figurer

2.1	Diagram over tabeller i en relasjonsdatabase som registrerer ordrer fra kunder i en netthandelapplikasjon.	9
2.2	Aggregatdiagram med to forskjellige entiter, modellert for den samme tjenesten fra 2.1.	15
2.3	Metamodell som viser forholdene mellom dataenheter i Cassandras logiske datamodell.	19
2.4	Eksempel på konsistent hashring i Project Voldemort.	30
4.1	Logisk oversikt over moduler i det typiske produksjonsmiljøet DBUpgradinator opererer i. Klassene som DBUpgradinator består av nevnes med navn.	57
4.2	Kommunikasjonsdiagram som illustrerer forholdet mellom en utvikler og en applikasjonstjener der DBUpgradinator benyttes for å migrere data levende.	59
4.3	Aktivitetsdiagram som illustrerer hvordan DBUpgradinator interfererer i applikasjonslogikken ved en GET - forespørsel før databaseklienten mottar spørringen.	60
4.4	Aktivitetsdiagram som illustrerer hvordan DBUpgradinator interfererer i applikasjonslogikken ved en GET - forespørsel etter at spørringen er ferdig.	61
4.5	Aktivitetsdiagram som illustrerer hvordan DBUpgradinator påvirker applikasjonens oppførsel ved en innkommende PUT - forespørsel.	63
4.6	Deployment-diagram av webapplikasjonen DBUpgradinator testes i.	65
5.1	Datamodeller for aggregatene til skjemaversjon "x" og dets etterfølger, "y".	74
5.2	Klassediagram av WebShopSimulator. Den gule, stiplede linjen markerer et skille mellom klassene til DBUpgradinator og klassene til WebShopSimulator.	79
5.3	Graf som plotter grupperte tidspunkt for persistering av data.	85
5.4	Graf som plotter grupperte tidspunkt for persistering av data (lesinger ekskludert).	88

5.5 Graf som plotter grupperte tidspunkt for persistering av oppdaterte data (skriveoperasjoner for migrasjonene er ekskludert).	90
---	----

Kodeoppføringer

4.1	Klassen <i>AbstractAggregateTransformer</i> , hvis hovedansvar er å lage en ny streng ut fra et gitt strengargument i funksjonen <i>transformAggregate</i> , som programvareutvikleren selv må implementere.	66
4.2	Migrator-klassen i DBUpgrador	67
4.3	Metodene <i>addTransformer</i> og <i>aggregateTransformerReceiver</i>	68
4.4	Indre klasse brukt til å definere en transformasjonsklasse som arver konstruktør fra AbstractAggregateTransformer	69
4.5	Funksjon som oppfyller rollen til objektet "AppVersionResolver" fra figur 4.1.	69
4.6	Metode for håndtering av GET-spørring i Migrator.	70
4.7	Metode for håndtering av PUT-spørringer i Migrator.	71
4.8	StringQueryInterface, grensesnittet Migrator-klassen bruker til å kjøre databasespørringer med.	72
5.1	Transformasjonsklasse for aggregat som holdeer data om registrerte brukere i WebShop.	76
5.2	cluster.xml - filen definerer de fire nodene i databaseklyngen til testmiljøet.	80
5.3	server.properties er filen som bestemmer oppførselen til den enkelte databasenode i databaseklyngen.	81
5.4	En STORE - fil som bestemmer hvordan data lagres i databasen.	82
5.5	De første 50 linjene i den applikasjonsloggen til tjeneren vbbmaster2018-crabbe (se tabell 5.1) fra test 2.	91

Forkortelser

<u>Forkortelse</u>	=	<u>Definisjon</u>
SQL	=	Structured Query Language
NoSQL	=	Not only SQL
DBMS	=	DataBase Management System
DDL	=	Data Definition Language
SLA	=	Service Layer Agreement
EC2	=	Elastic Compute Cloud
3-SAT	=	3-satisfiability
ReST	=	Representational State Transfer
BDB	=	Berkeley DB
TCP	=	Transmission Control Protocol

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

¹Tilgjengelig på url <https://aws.amazon.com/ec2/sla/>

medførte at mange tjenesteneder 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 interessant 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 hovedsakelige feilkilden i to av tre tilfeller; (2) operatørfeil har størst innvirkning på reparasjonstiden i to av tre internetttjenester; (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), tilgangsrrettigheter ("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 onlineoppgraderinger. 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 & 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 prosessen *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 presenteres denne definisjonen, hvorpå et sett med konkrete og oppnåelige målpunkter forbundet til definisjonen av problemstillingen også blir listet opp.

Problemstilling: Formålet med denne oppgaven er å modellere og implementere en programvaremodul, der delvis strukturerte datamodeller, også referert til som NoSQL-datamodeller, kan migreres uten å forårsake nedetid i applikasjonen. Modulen skal testes i et simulert, høytilgjengelig produksjonsmiljø for et fiktivt programvaresystem, hvis datamodell er aggregatororientert. I dette systemet gjennomføres en oppdatering som involverer endringer i applikasjonens implisitte dataskjema.

Opgaven har hatt følgende overordnede mål:

1. Modellere og implementere en modulær løsning i Java, der semistrukturerte datamodeller, også referert til som NoSQL-datamodeller, kan oppgraderes synkront med applikasjonslogikken til webapplikasjoner.
2. Teste den ovennevnte løsningen og evaluere om applikasjonens gjennomstrømming av forespørsler svekkes med 50 % eller lavere.

1.3 Oppgavens struktur

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

Kapittel 2 omtaler relevant teori om NoSQL - datamodeller, en kort innføring i databasearkitektur, distribuerte 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 - DBMS 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 applikasjonens implisitte datamodell og av distribuerte databasesystem, som ble funnet i løpet av et litteratursøk høsten 2017. Denne sammenlikningen 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 aggregatororienterte 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 binæradatokodingsbiblioteket 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 datamodell-evolusjonsverktøyet evalueres. Under testing vil webapplikasjonen DBUpgradiator gjennomgå en patch der den implisitte datamodellen sett fra applikasjonslagets perspektiv blir endret samtidig som en mengde genererte forespørsler tilsendes tjenerne, slik at en vanlig arbeidslast med spørringer simuleres.

I kapittel 6 konkluderes evalueringen og i det beskrives forslag til videre kvantitativt feltarbeid som kan bygge på denne analysen.

Kapittel 2

Teori

Ifølge Sadalage & 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: Aggregatororienterte 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 Infinite-Graph, OrientDB og FlockDB. Denne datamodellen vil ikke bli beskrevet i noen videre detalj. Det bør nevnes at med begrepet "datamodell" menes den programmatisk metode data organiseres av et DBMS, i vitenskapelig 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 aggregatororienterte datamodellen presentert, en kategori underordnet NoSQL-paraplyen som denoterer fellesnevneren mellom nøkkelverdilagre, dokumentlagre og kolonnefamilielagre. Samtlige tre former for aggregatororientering 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.

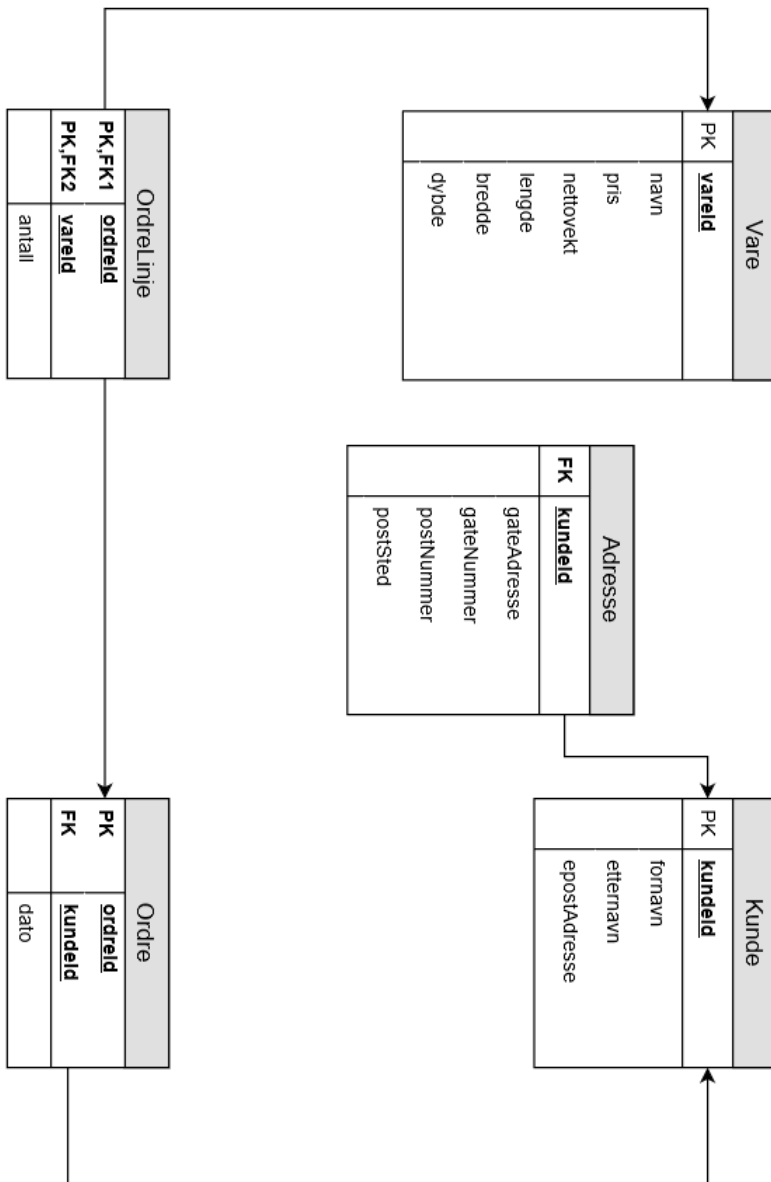
Imidlertid har ikke databasesystemet noen forståelse for de enkelte dataelementer som objektet innkapsler, det ser bare en helhetlig, ugjennomsiktig dataenhet, nemlig verdien av ett attributt i relasjonen. Ettersom tupler er den minste, udelelige dataenheten i den relasjonelle modellen er det korrekt å fastslå at spørringer opererer med og returnerer (et helt antall) tupler for hver enkelt spørring (Sadalage & 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 resultatrelasjonen 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 resultatuppelen. 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 aggregeringsspørringer på resultatant 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, maksimumsverdier eller minimumsverdier regnes ut, må gjøres i selve applikasjonen, etter at oppslag på **samtlig**e 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 én, eller flere, av nettleserens kjørende prosesser i kundens datamaskin. Variablene i datafeltene i disse objektene kan være så 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 netthandelapplikasjon.

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-tupple representerer en dis-

kret 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 jointtabell kalt *Ordrelinje* som kopler sammen *Ordre* og *Vare*, som illustrert i tabell-diagrammet i figur 2.1. Denne uoverensstemmelsen mellom strukturen av applikasjonsdata i programminne og strukturen på de samme dataene i en relasjonsdatabase, refereres til i industrien som ”the impedance mismatch” (Sadalage & Fowler, 2013).

Følgelig må applikasjonsutviklere konvertere data fra spørreresultater til den dataobjektstrukturen applikasjonen påkrever, noe som per idag ofte løses ved å innføre et separat abstrahert lag i applikasjonens logiske arkitektur: En tredjepartsmodul kalt ”object-relational mapper” (ORM). For språket Java kan man bruke Hibernate¹, JavaScript har SequelizeJS², og PHP-utviklere kan bruke Doctrine³, som også er en del av webapplikasjonsrammeverket Symfony⁴.

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 sammenkopleet 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 et 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

¹<http://hibernate.org/orm/>

²<http://docs.sequelizejs.com/>

³<http://www.doctrine-project.org/>

⁴<http://symfony.com/>

LAMP-tjener (Linux, Apache, MariaDB, PHP) som leverer en prototype av denne tjenesten i form av en webapplikasjon (George, 2011). Hush sin relasjonelle databasemodell normaliserer sine data ved å definere fire tabeller, `user`, `url`, `shorturl`, og `click` (George, 2011). De 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 skriveoperasjonene 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 avgrensar mengden av logisk relaterte 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 at en transaksjon kan ikke lese endringer gjort av andre transaksjoner som ikke er fullførte (eng. "Isolation"); holdbare, slik at i den grad transaksjonen persisteres til disk (eng. "Durable") er man sikret at alle transaksjonens endringer er trygt lagret. 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 (George, 2011).

Når skriveoperasjoner 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 stakkert 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 hurtig-

lager med det minnebaserte nøkkelverdilageret Memcached⁵. 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. Denne 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 skriveforespørlene 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 aggregatororienterte datamodellen

Så har vi den aggregatororienterte 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 & Fowler (2013) en felles kategori for disse tre NoSQL-typene: "Aggregatororienterte 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 data-

⁵<https://www.memcached.org/>

basemodellering 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 avgrensner 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 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 kontakte 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 applikasjonsprogrammerers 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 & 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.

Aggregatororienterte 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 håndteres 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 (Sadallage & Fowler, 2013).

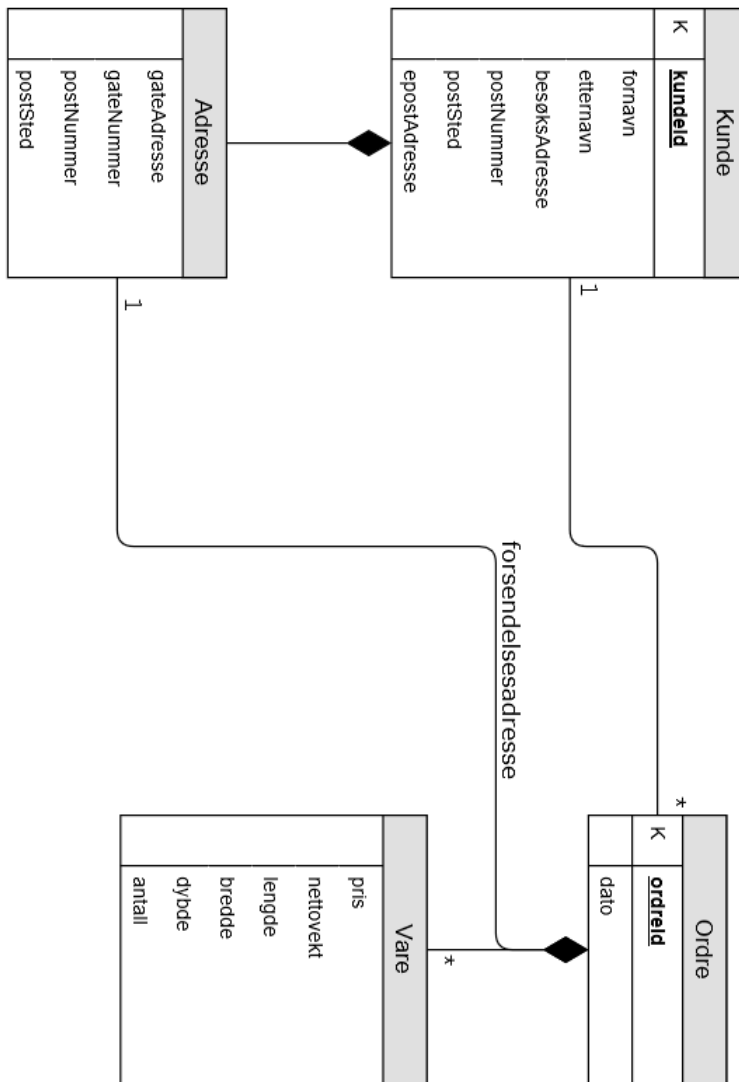
2.2.1 Design av aggregatmodeller

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. Sadallage & Fowler (2013) demonstrerer et eksempel på en aggregatororientert 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 nettbutikkssystemet at ”JOIN” blir gjort. Komposisjons-elementet 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. Aggregatororienterte 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 altså attributter på listeform, for eksempel en liste av strenger. Videre tillater ikke Codd (1971) kryssreferanse per primærnøkkel innad i en relasjon, fordi det med den egenskapen blir datamodellen rotete og vanskelig å lese for dens brukere.

En annen interessant detalj ved modellen presentert i delkapittel 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



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

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

på én spesifikk nøkkel potensielt medføre et stort svar i retur. Responsens kan bli så stor 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 & 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 & Fowler, 2013). Altså går det ikke an å hente deler av aggregatet ved et nøkkellopslag. Dokumentmodellen eksponerer aggregatet til databasen, og tillater dermed delvise spørringer.

I og med at dokumentmodellen også er skjematløs, 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 skjematløsheten 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 aggregatororienterte datamodellen. Dens hovedkarakteristikk er at hvert aggregat som lagres må være tilknyttet én unik identifikator kalt **nøkkel**, og for å hente 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 assosiert 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 syns 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 & 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 tillater 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 & 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 & Fowler, 2013; Elmasri, 2016) som finnes i aggregatobjektene, og derfor er slike delvise spørringer mulig. De enkelte attributter innen aggregater som er interessant 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 & Fowler, 2013). På tross av at dokumentdatabasen kan anskue aggregatets struktur, opererer hver enkelt spørring på ett enkelt, helhetlig dokument av gangen.

Dokumentlagre inndeles 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 dokumentssamling 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).

Dokumentssamlinger er ikke bundet til et skjema slik som relasjonelle databaser. Strukturen til aggregatet/dokumentet defineres av applikasjonsutvikleren, og velges utifra applikasjonskrav til datamodellen, i stedet for at applikasjonslogikk 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 innsettings spørringer tikker inn (i kontekst av Mongo via `insert` - funksjonen).

2.2.4 Kolonnefamilie modellen

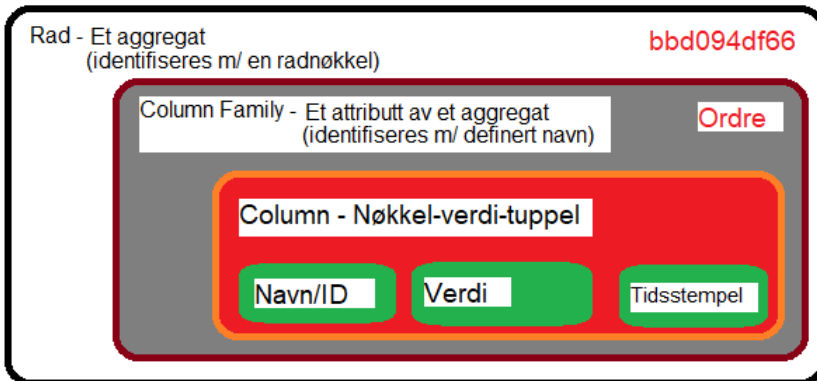
Googles BigTable er et databasesystem som har inspirert mange senere NoSQL-systemer, akkurat som Dynamo. BigTable er en ofte brukt lagringsteknologi hos Googles applikasjoner, deriblant Gmail (Elmasri, 2016). Apache HBase er et kolonnefamilielager tilgjengelig i åpen kildekode⁶ som implementerer de sentrale 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).

⁶Tilgjengelig på følgende GitHub-repositorium: <https://github.com/apache/hbase>

Apache Cassandra sin datamodell

- En forenklet syntese



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

I et kolonnefamilielager er hvert enkelt aggregat identifisert av en todimensjonal nøkkel: Den ene komponenten er en radnøkkel og den andre er navnet på en kolonnefamilie 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 én 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 enkelttoppslag 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 interressert 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 produktjonsmiljøet: 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 høyt tilgjengelig, 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-kommersplattformen. 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 testimoniale 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 påkrevder 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: $p = 0.05, s = 50 \Rightarrow 1 - (1 - p)^s = 0.9231$. 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 imidlertid 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 data-replikering 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 databasene 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 aggregasjonsoperasjon som summering og gjennomsnitt på enkeltverdier på enkeltattributter over flere dataobjekter. Flesteparten av de enkeltstående tjenestene som eksempelvis handlerer bestselgerlister, handlevogner, kundepreferanser, salgsstatistikk, og innloggingsdata, både spør etter og persisterer dataobjekter utelukkende på enkeltøkler. Altså er de korresponderende abstrakte relasjonsmodellene for hver av disse uavhengige tjenestene assosiasjonsløse, 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 applikasjonerne 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 data-maskiner, 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". En node holder rede på sine naboer i en oppslagsliste kalt "routingtabell". 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 lese-spørringer. 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 lettvinnt løsning: Ved fletting lagrer databasenoden simpelthen den oppdateringen på det vedkommende dataobjekt som har det seneste tidsstempet 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 replikarer 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 quorumreplikering⁷ 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 databaseadministratører i praksis stille på konsistensnivået til samtidige spørringer ved å konfigurere 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 betingelser er oppfylt:

⁷Begrepet *quorum* er latin og betyr ”beslutningsdyktig antall”

- $W > N/2$, altså at spørrekoordinatoren avventer bekreftelser på at over halvparten av de N replikaene fullbyrdet spørringen
- $R + W > N$, altså at mengden replikaer av et aggregat spørrekoordinatoren venter på før ett av disse returneres til databaseklienten overlapper med mengden bekref-telsesmeldinger fra en skrivespørring spørrekoordinatoren venter på før den returner en bekreftelsesmelding til klienten

I et strikt quorumsystem oppfylles sterk replikakonsistens, enhver lesespørring på et data-objekt 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 spørrekoordinatoren, det vil si at strikte quorumsystem øker nettverks-forsinkelsen 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 fron-ter. For det første: Hvis Dynamo hadde brukt et strikt quorumsystem for datareplikering, ville enkelte dataobjekter ha blitt utilgjengelige for databaseklientene, hvis nettverkspar-tisjoner i ringen oppstår, eller hvis de enkleste feilscenarier, der bare én enkelt node ram-mes, inntreffer. I tillegg ville skrivespørringer være langt mindre holdbare (eng. "durable") på disk (DeCandia et al., 2007). For det andre er Dynamos quorumprotokoll "sløv", det vil si at en spørrekoordinator oppnevner ikke en bestemt mengde noder som alle **må** fullføre spørringen for at resultat kan sendes tilbake til databaseklienten som kjørte data-basespørringen. Fordelen med denne sløvheten, eller "fleksibiliteten", er at risikoen mini-maliseres for at en spørring ikke blir besvart på grunn av at én enkelt node er utilgjen-gelig. 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 oppramsas som følger:

Konsistent hashing Lastbalanseringsalgoritmen Dynamo bruker for å partisjonere data utover ringen av databasenoder, som er formen til den strukturen til likemannsnett-verket, 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 represen-terer den enkelte noden som skriver dataobjektet, mens den andre variabelen er et versjonsnummer (eventuelt et tidsstempel) som øker monotont for hver gang objek-tet 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.

Slurvetete 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 håndtering av innkommende skrivinger adressert til en utilgjengelig node. Når den slurvetete quorumalgoritmen finner at én av adressentene i skrivequorumet er (midlertidig) utilgjengelig, så blir skrivequorumet midlertidig utvidet med en ekstra ”opp-passernode”. Denne midlertidige vekten 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 slurvetete quorumoperasjoner 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. Isteden 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” - ”Gossiping” er en mesterløs, 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 antydning av levering.

Lesereparasjon (eng. "Read repair") I Amazons implementasjon venter den enkelte databasenode på respons fra spørrekoordinatoren, etter å ha returnert et dataobjekt til den. Hvis spørrekoordinatoren 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 spørrekoordinatoren. Denne prosessen kjøres ad-hoc og er ikke effektiv når det kommer til å synkronisere replikaer ved endringer i medlemsmassen til Dynamo-ringen.

Denne teknologiske syntesen, tildels inspirert av distribuerte hashtabeller og quorumsystemer, medfører i sum et høytliggjengelig, 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 håndteres. 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 spørrekoordinator blant nodene. Derfor opererer de som et likemannsnettverk. Feil som oppstår ved spørringseksekvering behandles transparent.

⁸Kilkoden til Project Voldemort er lisensiert under Apache 2.0 - lisensen, og er tilgjengelig på følgende GitHub-repositorium: <http://github.com/voldemort/voldemort/tree/master>

2.4.1 Støttede operasjoner

Nøkkilverdilagre 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 til opprettelse og oppdatering av tupler), og `delete` (sletteoperasjoner).

I tillegg til disse tre operasjonene som er karakteriske for de fleste nøkkilverdilagre definerer Voldemorts klient – API endepunktene `getAll`, som er en leseoperasjon på multiple nøkler, og versjonerte utgaver av `get` og `put`. Til forskjell fra de ordinære funksjonene med samme navn tar disse metodene 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.

Dersom 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 nøkkilverdilagre 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 & 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 de fire ACID - egenskapene for skriveoperasjoner som gjøres i systemet gjennom transaksjonsmønsteret. Dette mønsteret sørger for at at hver enkelt gruppe av dataendringer enten utføres fullt og helt eller avbrytes, og mens endringene loggføres, kan de ikke leses av andre samtidige transaksjoner. Dette er atomisitet. 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 motakelig 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 spørrekoordinatoren 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 applikasjons-

klienten, 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.

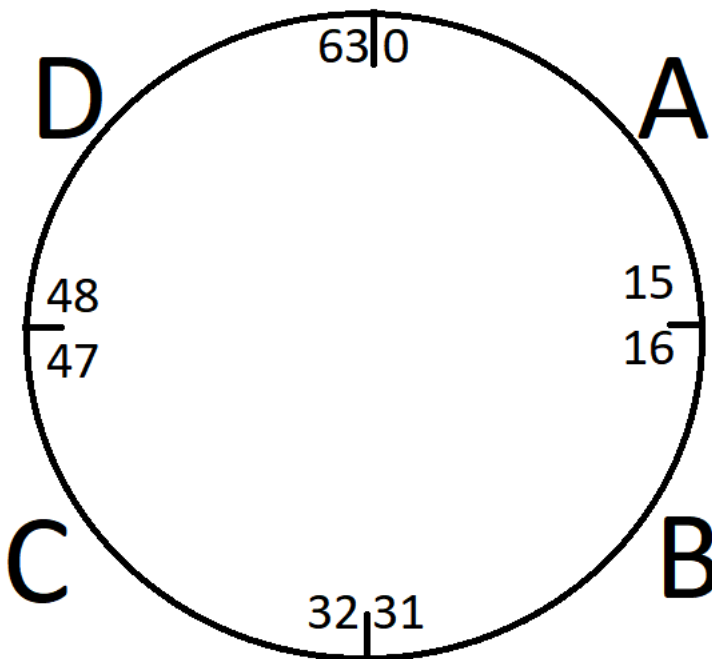
2.4.5 Lastbalansering av dataobjekter

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 bømter. Hver av disse partisjonene håndterer 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 hashing 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 data-replikering ved å legge antallet påkrevde replikaer, per quorumkonfigurasjonen, på etterfølgende 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 applikasjonsoppgradering. 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 webapplikasjonsoppgradering. 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 tjenestnivåavtale refereres til som et kvantitativt mål på systemets lovede tilgjengelighet, er den ofte basert på formelen for steady-state-tilgjengelighet. Dermed 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 abnormt 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-tilgjengelighet, 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-tilgjengelighet 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 forårsaker 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 feiltoleransmekanismer fokuserer på å motvirke, unngå, eller tillate uventede feil (Dumitraş & Narasimhan, 2009).

3.1.2 Versjonskappløp i distribuerte systemer

(Dumitraş et al., 2010) stiller følgende hypotetiske scenario om risiko forbundet med rul-

lerende oppgradering av distribuerte kjøringsmiljøer: I en webbasert nettbankapplikasjon som komunniserer med et banksystem bestående av mange tjenere oppdages en potensielt farlig anomali 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-innsprøytning 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 separate 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 & 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 & Nettles, 2005).

For eksempel benyttet Visa tidlig på 2000-tallet hele 21 store datamaskiner for å kjøre sitt transaksjonsprosesseringsystem (TPS). Ved oppgradering ble den enkelte instansen sin tilstandsdata bevart mens de enkelte prosessene ble byttet ut. Transaksjonsprosesserings-systemet til Visa opererte per 2001, ifølge Hicks & 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 & Nettles, 2005).

DSU – rammeverket omgår de ekstra oppgraderingskostnadene skissert i avsnittet ovenfor. Hicks & 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 & Nettles (2005) omtaler to forskjellige måter å implementere dynamisk oppdatering, eller ”dynamic patching”, som det også kalles. Den første framgangsmåten er å compilere den nye versjonen fra den endrede kildekode, 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 eksisterende, kjørende prosessen, transformere tilstandsdata der de er (her blir tilstanden kopiert til en annen adresse i prosessens 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 & Narasimhan, 2009). Feiltoleransmekanismer 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 & Narasimhan, 2009).

Dumitras & 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 & 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 & 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 & 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 overhead. 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 & 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 & 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

Blogginlegget til Hauer (2015) handler om behandling av databaser i en applikasjonlivssyklus der utviklingsmetodikken "Kontinuerlig Leveranse" (eng. "Continuous Delivery") gjennomføres. For hver gang ny kode skal kjøres på applikasjonstjenerne må database-skjemaet 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. Hauer (2015) anbefaler derfor å separere datakilder i en mikrotjenestearkitektur og derfor benytte et persisteringsmønster Sadalage & Fowler (2013) refererer til som ”polyglot persistence”.

I et kontinuerlig leveransmiljø 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”. Der- nest 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 inneholde 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 & 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 data-skjema lettere.

Å migrere data i et høytliggjengelig 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 aggregatororienterte modellene Sadalage & Fowler (2013) presenterer, gir den som er interessert i rullerende oppgradering av distribuerte webapplikasjoner en svært fleksibel vei hva angår dataskjema-oppgradering. Sett fra databasetjenestens 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 PostgreSQL. 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 Dumitraq et al. (2010); Dumitraq & Narasimhan (2009).

I diskusjonen om hvordan semistrukturerte datamodeller kan oppgraderes uten å avbryte tjenesteleveranse i produksjonsmiljøet, er visse fakta om aggregatororienterte 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

applikasjon eksplisitt, og lagres i en separat metadatatabell, kalt `schema_version`, i form av en identifikatorstreng og et tidsstempel (Gobec & 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 & 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 versjonsmiksen i datalageret, noe som åpner for versjonskappløp replikaene imellom per nøkkel ved både lese - og skriveoperasjoner.

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

¹En 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 telefonnummer 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 utgaveperspektivkomponenten

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 applikasjonssystemet 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, rullende 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. Versjonsskappløp 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 alltså 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 & Narasimhan (2009) påpeker at designet kan innføre flere oppgradingsscenarioer uten å påtvinge planlagt nedetid for applikasjonen som kjører med dette systemdesignet.

Dumitras & 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 (Dumitras & 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

Dumitras & 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"). Dumitras 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 midlertidig 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 høytligjengelige webapplikasjoner uten tjenesteavbrudd eller ytelsesdemping, der et nøkkelverdi-datalager (her representert ved Redis) brukes til å persistere applikasjonens data. Selv om nøkkelverdi-datalagre er skjematøse, 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 (jmfør kapittel 2.1), da vedlikeholdsarbeid gjøres på et tidspunkt ingen behøver systemets tjenester.

I en høytligjengelig 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 bakoverkompatibilitet på dataformatet kan systemets utviklere trygt oppgradere datamodellen med en manuell, rullerende teknikk, men da vil man samtidig legge kraftige begrensninger 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. Kildetoden til dette selvstendige databasesystemet ble publisert på GitHub den 18. september 2016², 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 tilsvarende 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 & 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 kompiles 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

²Tilgjengelig 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 ytelsestesting 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 ressuroverheadet 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.

Ytelsepåvirkningen, 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`. To 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.

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 ytelsespåvirkning både på databasens indre ytelse og dets innvirkning på en kjørende webapplikasjon i sin helhet.

Evalueringseksjonen fokuserer ikke på å sammenlikne KVolve med andre systemer som organiserer skjemaendringer på dynamisk vis. Dette kan ha noe med at samhandling mellom datamodellendringer 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 eksempelapplikasjon som henter inspirasjon fra Sadalage & 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.

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.

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

Oppdaterings-spesifikasjoner og aggregat-transformasjonsfunksjoner Når en datamodell skal oppgraderes installerer databaseadministratoren en programkompilert spesifisering 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 skjema-versjonen til aggregat-funksjonen skal kalles på.

Aggregat-transformasjon ved dataaksess Aggregater lagret i KVolve blir ikke automatisk oppgradert når en ny oppdateringsspesifisering 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 - operasjon treffer et aggregat som har en eldre skjema-versjon, men ikke den nyeste, vil en transformasjonsfunksjon kalles for å oppgradere aggregatet. Deretter kan det returneres til applikasjonsinstansen med den nyeste skjema-versjonen. 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 skjema-versjon er utdatert, automatisk frakoplet databasen og får ikke lov til å sende forespørsler. Slik hindrer Saur et al. (2016) et mikset versjonsmiljø og sikrer konsistente skjema-versjoner.

Opgavens 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 mikning av skjema-versjoner blant applikasjonstjenerne, slik at løsningen blir kompatibel med rulle-ende 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 skjema-versjoner. 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 database-klynge velges av de andre nodene til å ha vervet som klyngens mester, og 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-concept-implementasjonen av KVolve er skrevet i C. I et objektorientert språk som Java, kan klasser brukes til å modellere oppdateringsspesifiseringen. En klasse kalt **AbstractAggregateTransformer** inneholder en transformasjonsfunksjon, kalt *transformAggregate*. Denne klassen har to tilstandsvariabler. Den ene indikerer versjonen til skjemaet som data migreres fra. Den andre tilstandsvariablen indikerer

versjonen til skjemaet som data migreres til.

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 datalaget for webapplikasjonen som opererer på den, av hensyn til Dumitras & Narasimhan (2009) sine krav referert til i delkapittel 3.4.1. Når man implementerer en liknende tjeneste i en høytligjengelig applikasjon, må man være obs på at Amazon Dynamo er designet 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 å skrive og enklere å forstå.

KVolve sitt design ikke tillater versjonsmiks aggregater imellom. Det gjør imidlertid DBUpgradinator. Når den migrerer et aggregat oppretter den en ny nøkkel i stedet for å skrive en migrasjon til samme nøkkel. Dette betyr at dataobjekter lagres dobbelt hvis to forskjellige versjoner av webapplikasjonen er i omløp i programvaresystemet.

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.

- Project Voldemort brukes som NoSQL-database i testapplikasjonen.
- I databasearkitekturen modulen testes i, serialiseres aggregater på 'klientsiden' av Voldemorts logiske arkitektur, som er å finne i "Design" - siden til Kreps (2009) - altså er lagringsnodene "dumme" og backend-tjenerne "smarte".
- 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 som 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 tuppleen 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 som er lagret på det gamle skjemaet er migrert ferdig.
- Applikasjonen, skrevet i Java, holder styr på sitt implisitte skjema gjennom eksplisitt versjonering og eksplisitt deklarerer av aggregatets modell ved å definere en egen **Aggregate**-klasse.
- Webapplikasjonens utviklere programmerer på et eget lokalt utviklingsmiljø.
- I databasen er aggregatet lagret med nøkler på formen $k:x$, der suffikset x indikerer skjema-versjonen og prefikset k indikerer applikasjonens egendefinerte ID for aggregatet. 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.
- DBUpgradiator støtter hverken endring av nøkkelens strengstruktur eller endring av nøkkelens datatype.
- 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 skjema-versjon til alle applikasjonstjenere før denne sender over transformasjonsobjektet som gjelder for den etterfølgende versjonen.
- Transformasjonsobjekter sendes i en bestemt rekkefølge. Rekkefølgen applikasjonstjenere mottar transformatorer i er ikke tilfeldig. Denne antakelsen tillater oss å spesialisere og forenkle enkelte detaljer av kildekoden som kjører hos applikasjonstjenerne.
- 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 det to forskjellige utgaver av versjonen som opererer i produksjonsmiljøet på ett og samme tidspunkt.

4.2.2 Funksjonelle krav

1. DBUpgradiator skal kunne migrere hvert aggregat i et aggregatororientert database-system som tilhører et utdatert implisitt skjema. For å gjøre det skal DBUpgradiator opprette en ny nøkkel hvis prefiks, det vil si strengen før kolonet, er uforandret, mens suffikset er endret til navnet på det nye skjemaet. På denne nøkkelen persisteres et nytt dataobjekt som er transformert fra et annet dataobjekt fra det gamle skjemaet.
2. DBUpgradiator skal sende én ekstra skrivespørring til databasen for hver nye transformasjon som påføres et aggregat.
3. Ved en GET-forespørsel til aggregatet med applikasjons-ID k fra en applikasjonsinstans som bruker en gammel skjemaversjon x , skal DBUpgradiator sende databasen en annen GET-forespørsel på en nøkkel opprettet under det nye skjemaet, y . Dette gjøres for å kontrollere om aggregatet allerede har en korresponderende kopi i det nye skjemaet, før en eventuell migrasjon av aggregatet blir gjort.
4. Hver enkelt applikasjonsinstans vet versjonen til det implisitte dataskjemaet sitt, som identifiseres ved en streng som DBUpgradiator kopler sammen med applikasjonens egendefinerte ID og adskiller dem med kolon, før en spørring sendes til databasen.
5. For å migrere et aggregat med nøkkel $k:x$ vil DBUpgradiator opprette en ny nøkkel i databasen hvis skjemaversjon-suffiks er y (den neste skjemaversjonen) og applikasjons-ID-prefiks er uforandret.
6. Hvis DBUpgradiator mottar en spørring etter et aggregat som tilhører den nyeste skjema-versjonen (f.eks en GET på en nøkkel hvis skjemaversjon-suffiks tilsvarende versjonsstrengen til den nyeste skjemaversjonen), så vil ikke en migrasjon med transformator-klassen bli utført.

4.2.3 Kvalitetskrav

Tilgjengelighet Det skal være mulig å forespørre data fra databasen mens datamigrasjon er iverksatt. Dette kravet evalueres på et kvalitativt grunnlag. For å teste dette kravet kjøres DBUpgradiator i et simulert produksjonsmiljø, det vil si en applikasjon som mottar forespørslar fra klienter som kommuniserer med HTTP.

Ytelse Ved rullerende, eller lat datamigrasjon, utsettes databasen for en vesentlig degrading i gjennomstrømskapasitet, målt i spørringer per sekund. Denne hemmelsen må ikke være mer enn 50 % i den implementerte migrasjonsløsningen.

Modulær løsning Programmet bør kunne kjøres i de fleste ReST-orienterte webapplikasjoner skrevet i Java, uavhengig av hvilken aggregatororientert database applikasjonen lagrer data med.

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 spesifikt 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) kopleingene mellom programvareelementer og maskinvareelementer i systemet og 2) systemets distribuerte natur
Beskriver 1) kopleingene 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 arkitektur-dokumentasjonen er å representere effektene av de funksjonelle krav som stilles til datasystemet. De funksjonelle kravene har opprinnelse i domenet til problemet som datasystemet angriper. Systemets viktigste abstrakte konstruksjoner, kalt objekter i programmets og klasser i kildekode, 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 datasystemet oppfører seg ved kjøretid. 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".

Opgaven til arkitektur-dokumentasjonens 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 det stilles strenge, spesifikke krav til dets ytelse, tilgjengelighet og pålitelighet.

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

Utviklingsperspektivet taltaler primært applikasjonens utviklere og vedlikeholdere, som skal endre 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 mellom 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 deklare de regler som bestemmer organiseringen av kildekode. Eksempler på slike regler er 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

for 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 bruksscenarioer, som oppsummerer systemets mengde av funksjonelle krav (Kruchten, 1995). Meningen bak dekomponeringen av arkitektur-dokumentasjonen 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 prosjektet 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

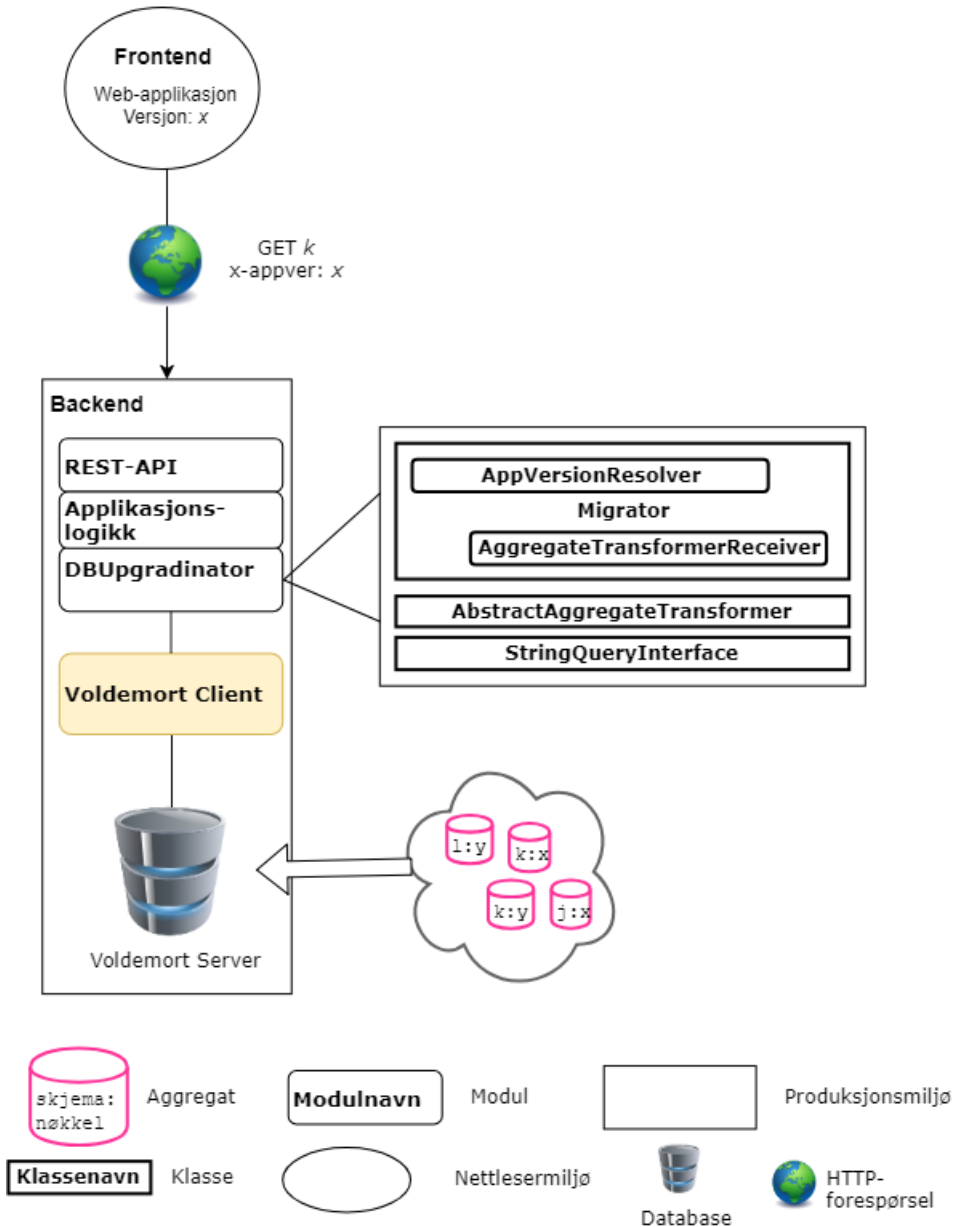
Figur 4.1 viser en oversikt over de ulike bestanddelene i DBUpgradinator. Figuren viser også DBUpgradinator sin logiske relasjon til REST-webapplikasjonen brukt til å teste dette migrasjonsverktøyet i. Det logiske perspektivet (eng. "view") beskriver også de ulike atomiske moduler webapplikasjonen består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



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

Applikasjonslogikken til webapplikasjonen er kode som behandler HTTP-forespørsler og oppretter HTTP-responser. Den videregiver både skjemanøkkel og aggregatnøkkel ned til DBUpgradinator, som håndterer 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 vektorklokkeversjoner når den mottar spørringsresultater fra 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 applikasjonslogikken 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.

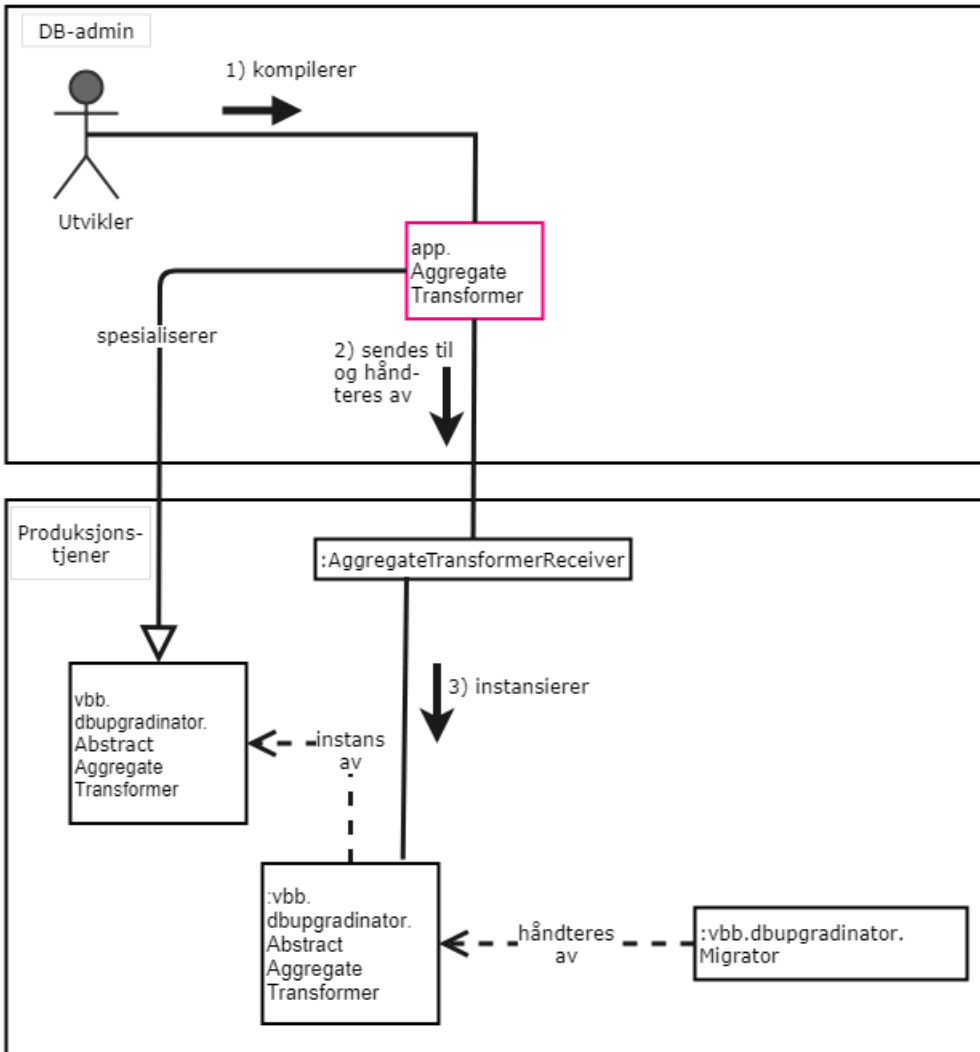
AppVersionResolver Oppgaven til denne modulen er å kople dataobjektets nøkkel i en innkommende HTTP-spørring (k) med applikasjonsversjonens nøkkel (x) på formen $k + : + x$

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 implementert 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 KVue, 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.

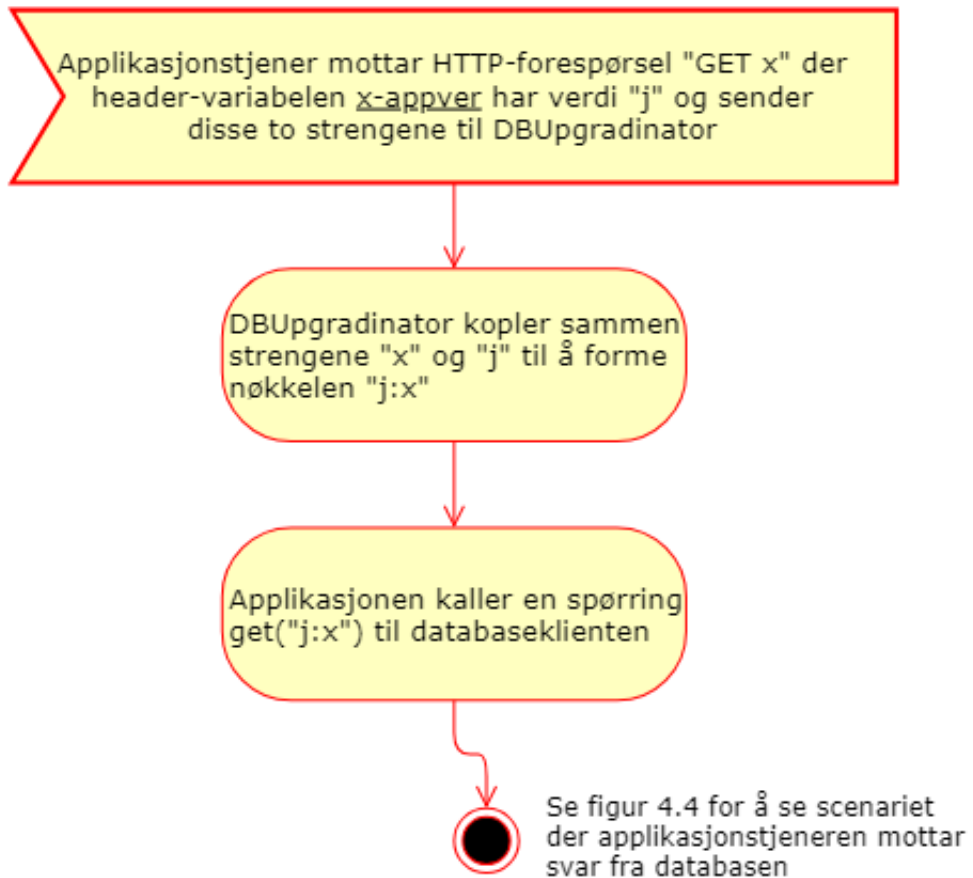
Etter å ha implementert transformasjonsklassen (markert med rosa omriss i kommunikasjonsdiagrammet) som spesialiserer **AbstractAggregateTransformer**, kompilerer utvikleren filen til en `.class` - fil og sender den til hver applikasjonstjener som kjører i systemet. AggregateTransformerReceiver-modulen leser inn alle objektene i filen tjeneren mottar, inn en byte-matrise og skaper et objekt i minnet. Dette objektet opprettes som



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

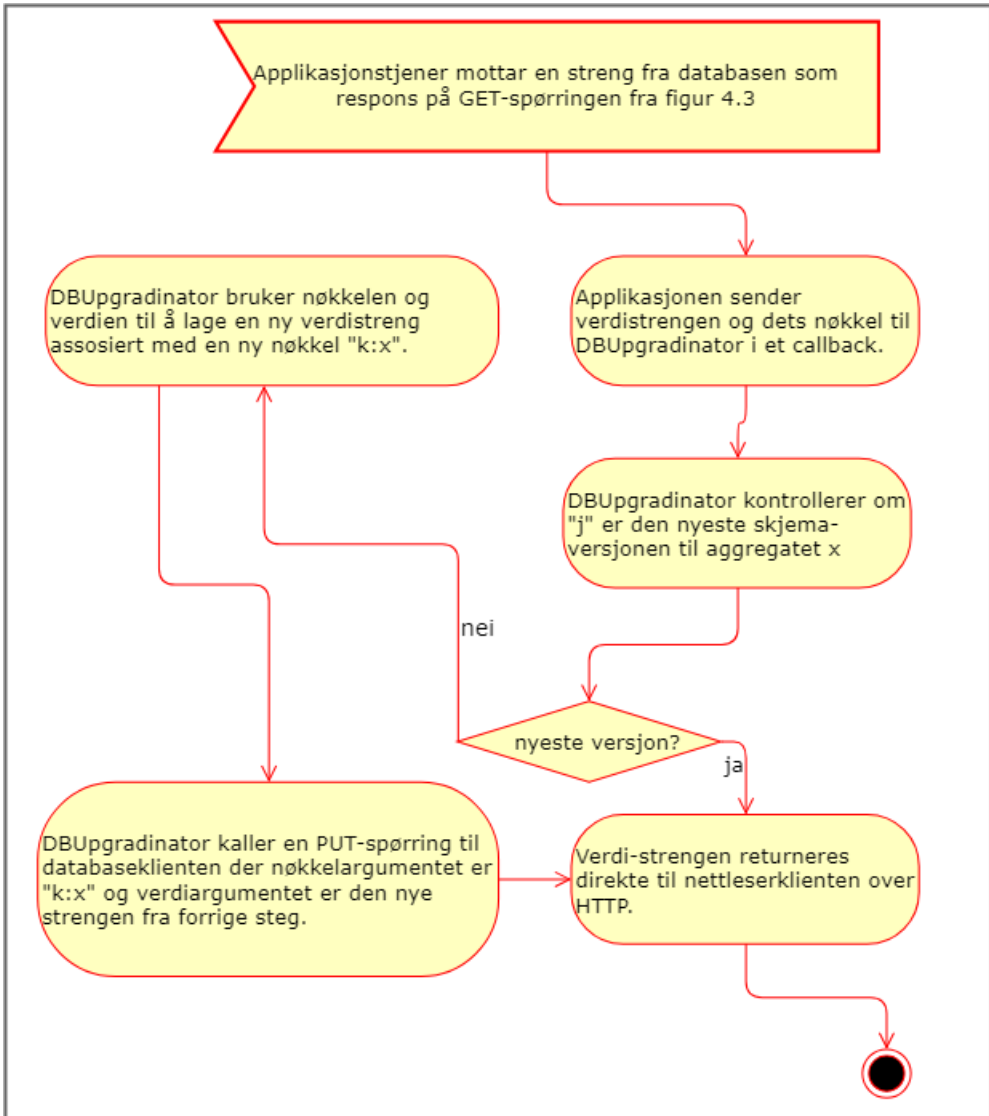
en instans av **AbstractAggregateTransformer**. Objektet blir dernest plassert i **Migrator**-instansens liste av transformatorer. I figur 4.2 er objektinstanser merket med kolon i navnet.

4.3.3 Prosessperspektivet



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

Aktivitetsdiagrammet i figur 4.3 beskriver systemets oppførsel når en GET-forespørsel fra en applikasjonstjener hvis skjema-versjon er j , mens produksjonsmiljøet er under oppgradering av dataskjemaet fra versjon j til k . Applikasjonstjeneren kaller på `AppVersionResolver`-modulen i DBUpgradiator for å sette sammen aggregatets nøkkel med dataskjemanøkkel. Dernest kaller tjeneren på `get` - metoden til en `StoreClient` - instans. Tråden som utførte dette databasekallet vil da bli blokkert.



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

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.

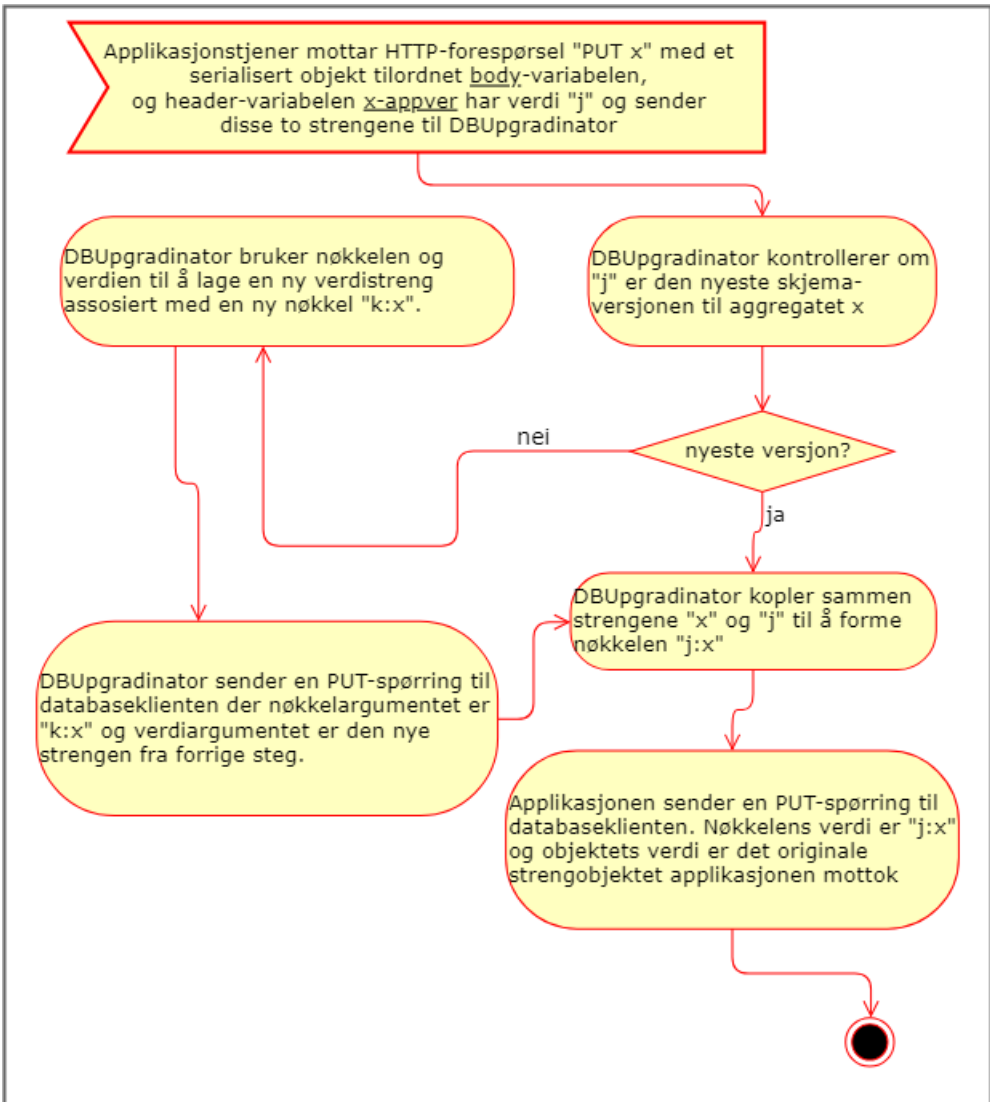
I det motsatte tilfelle sender programmet aggregatet inn i transformasjonsfunksjonen, som lager en ny streng utifra strengargumentet den fikk tilsendt. Ved hjelp av prosessen *AppVersionResolver* lager programmet en ny nøkkel der skjemaversjonsuffikset tilsvarer den strenge transformatorobjektet indikerer 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 annen tråd enn den som betjener den brukerinitierte forespørselen. Med en gang databasespørringen til det migrerte aggregatet tilsendes StoreClient, returneres spørresultatet 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 .

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 ved en rullerende applikasjonsoppdatering.

Når applikasjonen mottar en bekreftelse fra et put-kall opprettes en HTTP-respons som informerer om hvorvidt skriveoperasjonen lyktes eller ikke. Responsen sendes tilbake til applikasjonsklienten over TCP/IP - forbindelsen.



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

4.3.4 Det fysiske perspektiv

Det fysiske perspektivet beskriver testmiljøet DBUpgradinator skal testes i. De fire bokserne i figur 4.6 tilsvarer fire forskjellige virtuelle datamaskiner. Hver av de virtuelle maskinene kjører to separate programmer (kalt ”kjøretidsmiljø”) som lytter på én nettverksport hver. Disse programmene er:

1. Webapplikasjonen, som i figur 4.6 benytter artefaktene `WebShopSimulator.jar` og `DBUpgradinator.jar` til å kjøre selve applikasjonstjeneren; 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 instanser av **StoreClient**

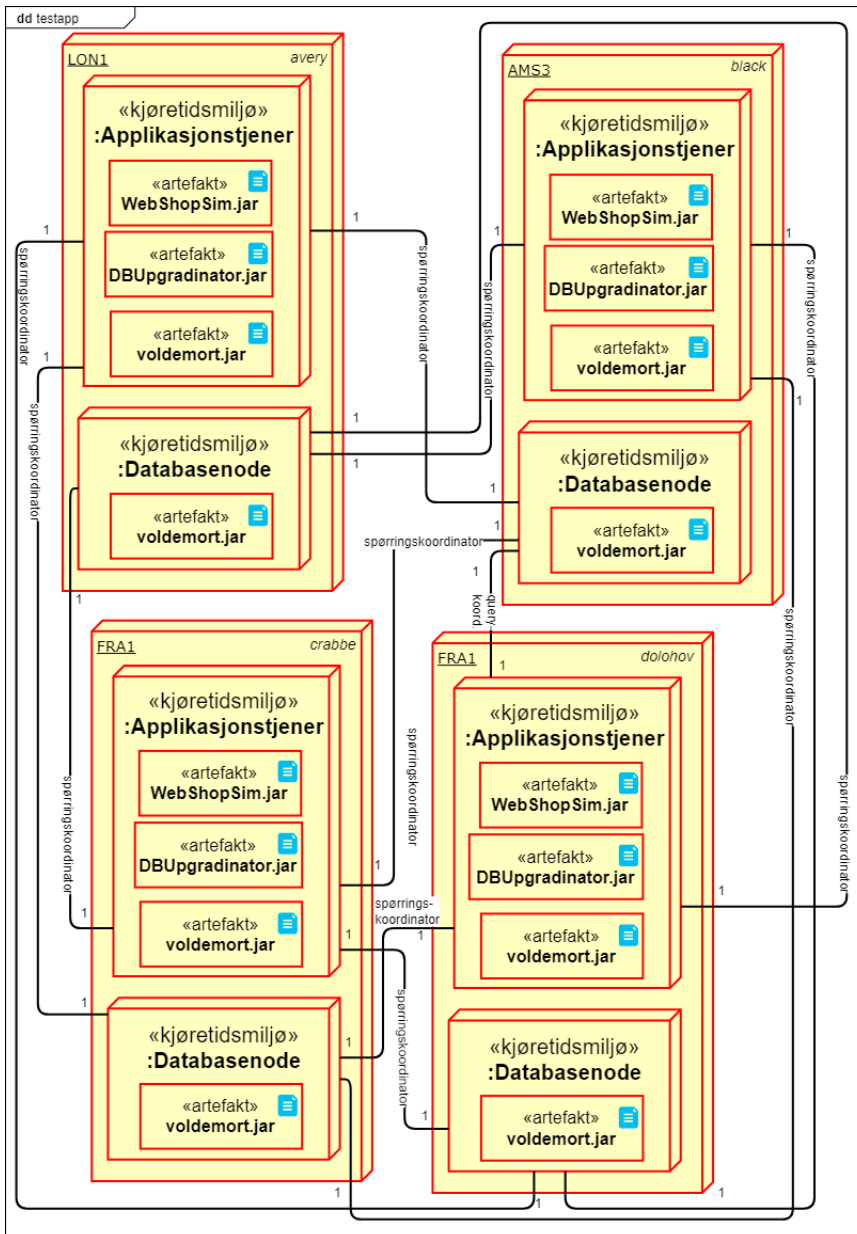
Figur 4.6 viser også at hver kjørende instans av Voldemort-klienten fra applikasjonstjenerprogrammet har én replikeringsassosiasjon (kalt ”spørrekoordinator”) med hver av de tre databasenodene stasjonert på de andre virtuelle maskinene. Det er ikke nødvendigvis slik replikeringsstrategien er konfigurert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

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 type-kompabilitet baseres på eksplisitte deklarasjoner i kildekoden. I Java deklarerer 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 deklarerert for ethvert objekt i kildekoden. Et statisk typet språk er et språk der typen til en variabel kontrolleres idet kildekoden kompiles, 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 migrasjonsmetodene *getAndMigrateAggregate*, *putAndMigrateAggregate* og *postAndMigrateAggregate* er skrevet i henhold til aktivitetsdiagrammene 4.3, 4.4 og 4.5.



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

4.4.1 AbstractAggregateTransformer

Kodeoppgøring 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 final String currentAppVersion;
5     private final String nextAppVersion;
6
7     public AbstractAggregateTransformer(String currentAppVersion,
8         String nextAppVersion) {
9         this.currentAppVersion = currentAppVersion;
10        this.nextAppVersion = nextAppVersion;
11    }
12
13    public String getNextSchemaVersion() {
14        return nextAppVersion;
15    }
16
17    public String getAppVersion() {
18        return currentAppVersion;
19    }
20
21    /**
22     * Both parameters are strings either returned from or going
23     * into a DB query
24     * @return String - the new aggregate based on the input
25     * @param val The value, which is a generic object
26     */
27    public abstract String transformAggregate(String val);
28 }
```

AbstractAggregateTransformer er en abstrakt klasse, som har to konstante feltvariable. Verdien på disse feltvariablene defineres når dens konstruktør kalles. Den ene indikerer versjonen til skjemaet transformasjonsklassen mottar aggregater fra, og den andre indikerer versjonen til skjemaet som den transformerte verdien gjelder for.

Klassen har én abstrakt metode, som implementeres i en subklasse. Denne abstrakte metoden heter *transformAggregate*, som påkalles når en forespørsel sendt fra en tjenerinstans som bruker den gamle skjema-versjonen, tilsendes databaseklienten. Oppgaven med å skrive denne subklassen overlates til applikasjonsutviklerne som benytter seg av DBUpgradinator. Etter å ha skrevet kildekoden til subklassen, må utvikleren distribuere .java - filen der subklassen er definert, til hver av applikasjonstjenerne manuelt. .java - filen må ligge underordnet samme katalog som den webapplikasjonens kildekode blir kompilert fra. Etterpå må klassen bli kompilert til en binær .class-fil.

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 & Yellin, 1999), i tillegg til å inneholde ekvivalent kildekode og like navn for alle variabler og metoder. Dersom de befinner seg i to forskjellige pakker, er de ikke identiske.

Det ble opprinnelig forsøkt å implementere et eget kommandolinjebasert program som lar utviklerne sende kompilerte `.class` - filer over TCP ved hjelp av klassene **ObjectOutputStream** og **ObjectInputStream** fra `java.io` - pakken. Ettersom den hjemmelagde aggregattransformatoren må være en spesialisering av `AbstractAggregateTransformer`, en klasse fra `DBUpgradorator` - modulen, så må artefakten `DBUpgradorator.jar` legges inn i klientprogrammets prosjektfiler som en avhengighet. En annen kopi av `DBUpgradorator.jar` er å finne blant webapplikasjonens avhengighetsfiler.

Per JVM-spesifikasjonen er klassen **AbstractAggregateTransformer** fra `.jar` - filen i klientprogrammet ikke den samme klassen som **AbstractAggregateTransformer** fra `.jar` - filen i tjenerprogrammet. Årsaken til det er at det er to forskjellige instanser av system-klasserinnlasteren som leser klassesdefinisjonene inn i hver sin instans av JVM. Derfor ble ikke forsendelse av binære Java-klasser implementert.

Den enkleste løsning på dette problemet er å bruke `getMethod()` - metoden fra *reflection*-biblioteket for å instansiere hver enkelt `transformAggregate` - metode som en egen instans av **Method** - klassen. Ved migrasjon kalles `invoke` - metoden for å kalle på `transformAggregate`. Den returnerte verdien fra `invoke` - kallet kan da "castes" til **String**, jamfør linje 26 i kodeoppføring 4.1. Denne løsningen skroter **AbstractAggregateTransformer** - klassen fullstendig.

4.4.2 Migrator

Kodeoppføring 4.2: Migrator-klassen i DBUpgradorator

```
1 package vbb.dbupgradorator;
2
3 import java.lang.reflect.Constructor;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7 import java.nio.file.StandardOpenOption;
8 import java.util.HashMap;
9 import java.text.DateFormat;
10 import java.text.SimpleDateFormat;
11 import java.util.Date;
12
13 public class Migrator {
14     private final AggregateTransformerLoader loader = new
        AggregateTransformerLoader();
```

```
15     private final DateFormat dateFormat = new
        SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
16     private final String className;
17     private final HashMap<String, AbstractAggregateTransformer>
        transformers = new HashMap<>(4, (float) 0.95);
18
19     public Migrator() {
20         this.className = "app.UserAggregateTransformer";
21         this.aggregateTransformerReceiver();
22     }
23
24     // Se kodelisting 4.3 til og med 4.8 for resten av kildekoden til
        Migrator
25     // ...
26
27 }
```

Migrator er en `public` klasse som eksponerer DBUpgradinator sin kjernefunksjonalitet til programvareutvikleren. Dens tilstand inkorporerer 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`.

En instans av **Migrator** har en ordliste (**HashMap**) 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.

I likhet med `Kvolve` kan ikke en instans av `Migrator` slås av, den er alltid på. Ei heller kan **AbstractAggregateTransformer** - objekter fjernes fra `Migrator`s transformatorordliste, fordi migrasjonsprosessen ikke er styrt på tvungent vis, slik konvensjonen er med relasjonelle databaser. Følgelig kan man konkludere at man ikke kan være sikker på når en "migrasjonsprosess" med `DBUpgradinator` er omme i praksis. Dette kommer av at man da må overvåke alle nøkler i databasen på tvers av skjemaversjonene, hvilket ville ha vært lettere hvis det hadde vært mulig å utføre delvis nøkkeloppslag, på til eksempel skjema-suffikset, i nøkkelverdilagre.

Kodeoppføring 4.3: Metodene `addTransformer` og `aggregateTransformerReceiver`.

```
1     private void addTransformer(AbstractAggregateTransformer t) {
        this.transformers.put(t.getAppVersion(), t); }
2
3     private void aggregateTransformerReceiver() {
4         if ( this.transformers.size() < 1 ) {
5             try {
6                 Path path = Paths.get( "classes/transformer/" +
```

```

        this.className.replace(".", "/") +
        ".class").toAbsolutePath();
7     if (Files.exists(path)) {
8         byte[] classData = Files.readAllBytes(path);
9         Class<?> c = loader.createClass(this.className,
10            classData);
11        Constructor cons = c.getConstructor(String.class,
12            String.class);
13        AbstractAggregateTransformer tran =
14            (AbstractAggregateTransformer) cons.newInstance(
15            "x", "y" );
16        this.addTransformer(tran);
17        log("INFO @ " + this.dateFormat.format(new Date()) +
18            " - Received aggregate transformer for schema " +
19            tran.getAppVersion() + "\n");
20    }
21    } catch (Exception e) {
22        e.printStackTrace();
23        log("ERROR @ " + this.dateFormat.format(new Date()) + "
24            " + e.toString() + "\n");
25    }
26    }
27    }

```

Funksjonen *aggregateTransformerReceiver* vises i kodeoppføring 4.3. Den påkalles hver gang en av migrasjonsfunksjonen blir kalt av applikasjonen (se oppføringer 4.6 og 4.7), og én gang i konstruktøren til **Migrator**. Metoden gjør et I/O-kall til disk for å kontrollere om en ny `.class` - fil (med et spesifikt navn) er ankommet den oppgitte katalogen.

Kodeoppføring 4.4: Indre klasse brukt til å definere en transformasjonsklasse som arver konstruktør fra **AbstractAggregateTransformer**.

```

1 private class AggregateTransformerLoader extends ClassLoader {
2     AggregateTransformerLoader() { super(); }
3     final Class<?> createClass(String name, byte[] b) throws
4         ClassFormatError {
5         return super.defineClass(name, b, 0, b.length);
6     }
7 }

```

I kodeoppføring 4.4 kan man lese kildekoden til en indre klasse, **AggregateTransformerLoader**, som er en spesialisering av **ClassLoader**. En indre klasse er en ikke-statisk klasse, deklartert innen en annen klasse. Hver **Migrator** - instans har en egen instans av denne klassen. Den eneste oppgaven instansen har er å lese inn rekkefølger av oktetter (`byte[]` - typen i Java) og transformere dem til **Class** - objekter. Hver slik instans av **Class** brukes i sin tur til å instansiere **AbstractAggregateTransformer** - objekter.

Kodeoppføring 4.5: Funksjon som oppfyller rollen til objektet "AppVersionResolver" fra figur 4.1.

```
1 private String getPersistedKey(String aggregateKey, String
    schema) {
2     return aggregateKey + ":" + schema;
3 }
```

Oppgaven til funksjonen *getPersistedKey* er å kople dataobjektets nøkkel i en innkommende HTTP-spørring (k) med applikasjonsversjonens nøkkel (x) på formen k + ":" + x. Det er en privat metode som brukes av de migrasjonsmetodene i kodeoppføringer 4.6 og 4.7. Metoden er gjort privat for klasseinstansen med vilje for å abstrahere applikasjonen bort fra databaselogikken.

I Migrator - klassen er to metoder implementert for å migrere aggregater som kommer fra henholdsvis en GET - og en PUTforespørsel. I testapplikasjonen WebShopSimulator er det DBUpgradiator som står for all håndtering av spørringskall til databaseklienten.

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

```
1 public String getAndMigrateAggregate(StringQueryInterface db,
    String aggregateKey, String schema) {
2     this.aggregateTransformerReceiver();
3     String key = this.getPersistedKey(aggregateKey, schema);
4     String aggregate = db.query(key); // Blocking DB op
5     AbstractAggregateTransformer spec =
        this.transformers.get(schema);
6     if (null != spec) {
7         String nextSchema = spec.getNextSchemaVersion();
8         String nextKey = this.getPersistedKey(aggregateKey,
            nextSchema);
9         new Thread(() -> {
10             if (!aggregate.equals("")) {
11                 String checkedAggregate = db.query(nextKey); //
                    Blocking DB op
12                 if (checkedAggregate.equals("")) {
13                     String migratedAggregate =
                        spec.transformAggregate(aggregate);
14                     Exception fail = db.persist(nextKey,
                        migratedAggregate); // Blocking DB op
15                     if (fail == null) {
16                         log("INFO @ " + this.dateFormat.format(new
                            Date()) + " - Migrated aggregate with key "
                                + key + " to " + nextKey + "\n");
17                     } else {
18                         log("ERROR @ " + this.dateFormat.format(new
                            Date()) + " - Error during migration from "
                                + key + " to " + nextKey + ":\n" +
                                    fail.toString() + "\n");
19                     }
20                 }
            }
        })
    }
```

```

21     }
22     }).start();
23 }
24 return aggregate;
25 }

```

GETmigrasjonsmetoden returnerer spørringsresultatet av brukerens forespørsel til applikasjonen. Selve migrasjonsprosessen blir utført ved å opprette én ny tråd der en lambda-funksjon blir kalt én gang.

Kodeoppføring 4.7: Metode for håndtering av PUT-spørringer i Migrator.

```

1 public boolean migrateAndPutAggregate(StringQueryInterface db,
2   String aggregateKey, String schema, String ag) {
3   this.aggregateTransformerReceiver();
4   AbstractAggregateTransformer spec =
5     this.transformers.get(schema);
6   String key = this.getPersistedKey(aggregateKey, schema); //
7     This is the key used by the application
8   if (null != spec) {
9     String nextSchema = spec.getNextSchemaVersion();
10    String nextKey = this.getPersistedKey(aggregateKey,
11      nextSchema);
12    new Thread(() -> {
13      Exception fail = db.persist(nextKey,
14        spec.transformAggregate(ag));
15      if (fail == null) {
16        log("INFO @ " + this.dateFormat.format(new Date()) +
17          " - Migrated aggregate with key " + key + " to "
18          + nextKey + "\n");
19      } else {
20        log("ERROR @ " + this.dateFormat.format(new Date()) +
21          " - Error during persisting " + nextKey + "\n" +
22          fail.toString() + "\n");
23      }
24    }).start();
25    Exception ex = db.persist(key, ag);
26    return (ex == null);
27 }

```

Den andre migrasjonsfunksjonen er noe annerledes fra *getAndMigrateAggregate*. Den returnerer en bekreftelse som indikerer hvorvidt en skrivespørring som ble utført i klassen som implementerer **StringQueryInterface** kastet et unntak eller ei. I fall *put*-kallet gjorde det, returnerer migrasjonsfunksjonene false, og true i motsatt tilfelle.

4.4.3 StringQueryInterface

Kodeoppføring 4.8: 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 som utfører spørringer for Migrator-instansen. Kodeoppføring 4.8 viser hele dette grensesnittet, som eksponerer to metoder: *query* og *persist*. *persist* - metoden forventes å returnere en instans av **Exception**. Grunnen til dette er at *persist* skal kalle *put*-metoden til en instans av **StoreClient**. I Voldemort sitt tilfelle kan *put* kaste et **ObsoleteVersionException** - unntak, som må fanges opp.

For at **Migrator** - instansen skal kunne registrere at spørringen feilet, må den motta et unntak som returverdi. Den verdien kommer til å være lik `null` hver gang spørringen ble fullbyrdet. Utvikleren forventes å skrive en separat klasse som implementerer dette grensesnittet. Det er denne klassen som må rydde opp i eventuelle flettekonflikter, det vil si hver gang *get*-kallet returnerer en liste av aggregater i stedet for bare ett.

Som en konsekvens av denne kildekoden, kan DBUpgradinators brukes i webapplikasjoner med aggregatororienterte datamodeller som bruker andre nøkkelverdi-lagre enn Project Voldemort til å persistere data.

Kapittel 5

Evaluering av evolusjonsløsning av datamodeller

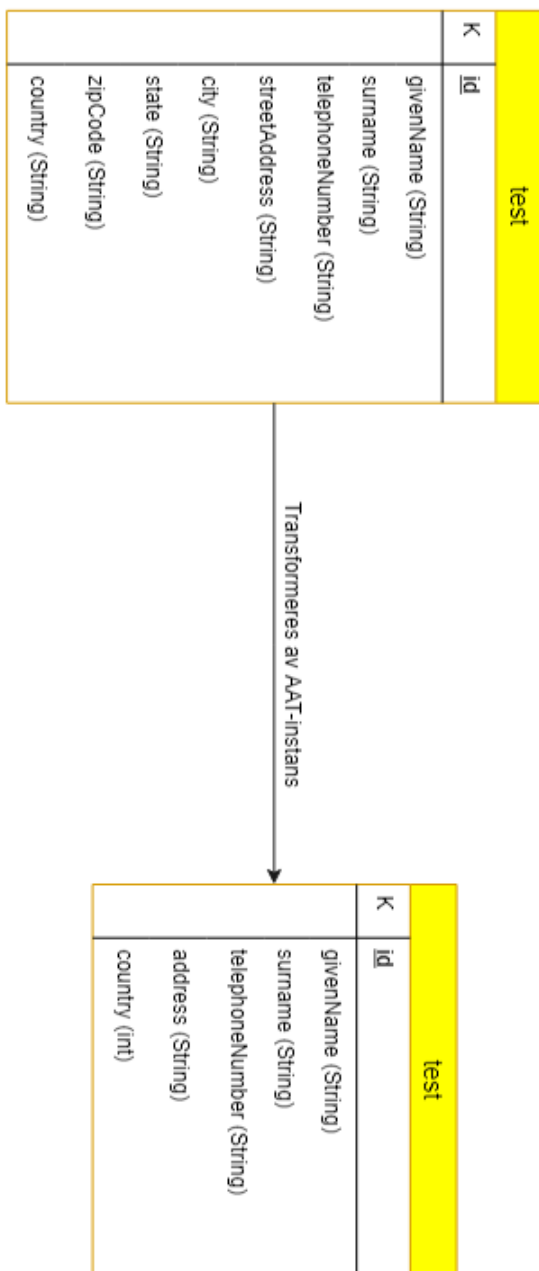
Dette kapitlet presenterer og diskuterer resultatene av testingen av evolusjonsløsningen DBUpgradiator, og sammenlikner datamodellevolusjonsløsningens kvaliteter med øvrige systemer presentert i kapittel 3. I delkapittel 5.2 vil testprosessen beskrives, herunder inkludert konfigurasjon av Voldemort-klyngeinstansen;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

5.1 Testscenario

Her følger en beskrivelse av webapplikasjonen migrasjon med DBUpgradiator blir testet på. Nettbutikken "WebShop" benytter nøkkelverdilageret Project Voldemort for å lagre et register over sine kunder. De fleste kundene av nettbutikken bor enten i Storbritannia eller i Australia. Aggregatene som lagres av Voldemort er flate, og består av strengnøkler og strengverdier.

I skjemaversjon "x", vist i venstre modell i figur 5.1, har aggregatene til WebShop følgende attributter: `givenName` (fornavn), `surName` (etternavn), `telephoneNumber` (telefonnummer), `streetAddress` (gateadresse), `city` (poststed), `state` (delstat - for australske og eventuelt amerikanske kunder), `zipCode` (postkode, hvis system varierer fra land til land), og `country` (landet til adressen).

Samtlige attributter er strenger, og deres verdier er også strenger. Dette er det enkleste alternativet for å representere tekst som inneholder mellomrom, bindestrek og parantes, hvilket er tilfelle for telefonnummeret og postkoden.



Figur 5.1: Datamodeller for aggregatene til skjemaersjon "x" og dets etterfølger, "y".

For amerikanske og australske kunder av WebShop er skjema "x" en optimal aggregatstruktur, fordi alle har en leveringsadresse bestående av både en gateadresse, poststed, delstat, postkode og land. For kunder i Storbritannia eller Irland vil delstats-feltet måtte stå tomt.

Derfor blir følgende endring gjort: Adressekomponentene `streetAddress`, `city`, `state`, `zipCode` og `country` slås sammen til ett, helhetlig adressefelt, slik at hver enkelt kunde selv kan skrive inn sin korrekte, internasjonale postadresse i en tekstboks, der hver adressekomponent står på sin egen linje. Alle de nevnte individuelle attributtene, bortsett fra én, slettes fra aggregatstrukturen.

Feltet `country` endres i stedet for å slettes. I stedet for å lagre en streng som identifiserer et land på en forkortelse, brukes i stedet en landskode, et heltall¹ man må ta i bruk hvis man skal ringe en person hvis telefoniabonnement er registrert i utlandet. Først skrives et pluss tegn ("+"), etterfulgt av landskoden og deretter personens telefonnummer. Skjemaendringen blir gjort fordi de færreste brukere skriver inn landskoden sammen med telefonnummeret sitt i et tekstfelt når de registrert. Den typiske netthandel utleder istedet brukerens landskode utifra deres oppgitte land.

Denne skjemaoppgraderingen består av to endringer som gjør at det nye skjemaet, "y", ikke er bakoverkompatibelt med skjema "x". For det første endres typen til ett attributt fra streng til heltall. Denne endringen er inkompatibel skjemaene imellom, fordi mengden gyldige strenger er uendelig større enn mengden gyldige heltall.

Den andre kompatibilitetsbrytende endringen er at ett eller flere attributter slettes. Se kodeoppføring 5.1 for kildekode som oppretter et nytt aggregat basert på aggregatet `val`, som tilhører det gamle skjemaet. Her brukes en tredjeparts modul til å forme JSON-strenger uten at metoden har noe direkte informasjon om parameteret `val`, som er en streng.

¹For enkelte selvstendige stater går det ikke an å representere deres landskode med bare et heltall, da deres kode inneholder "-". Eksempler på slike stater inkluderer Jamaica (1-876), Trinidad og Tobago (1-868) og Bahamas (1-242).

Kodeoppføring 5.1: Transformasjonsklasse for aggregat som holder data om registrerte brukere i WebShop.

```
1 package app;
2
3 import org.json.JSONObject;
4 import vbb.dbupgradinator.AbstractAggregateTransformer;
5
6 public class UserAggregateTransformer extends
    AbstractAggregateTransformer {
7
8     public UserAggregateTransformer(String current, String next) {
9         super(current, next);
10    }
11
12    private static String checkEmptiness(String d) {
13        if (d.isEmpty()) { return ""; }
14        return d + "\n";
15    }
16
17    private static int getCountryCode(String country) {
18        int phone = -1;
19        String checker = country.toUpperCase();
20        if (checker == "UK") { phone = 44; }
21        else if (checker == "AUS") { phone = 61; }
22        return phone;
23    }
24
25    public String transformAggregate(String val) {
26        JSONObject jo = new JSONObject(val);
27        String countryAbb = jo.getString("country");
28        String fullAddress =
29            checkEmptiness(jo.getString("streetAddress")) +
30            checkEmptiness(jo.getString("city")) +
31            checkEmptiness(jo.getString("state")) +
32            checkEmptiness(jo.getString("zipCode")) +
33            checkEmptiness(countryAbb);
34        jo.put("address", fullAddress);
35        jo.remove("streetAddress");
36        jo.remove("city");
37        jo.remove("state");
38        jo.remove("zipCode");
39        jo.put("country", getCountryCode(countryAbb));
40        return jo.toString();
41    }
42 }
```

5.2 WebshopSimulator

Webapplikasjonen som DBUpgradinator testes på, kalles for *WebshopSimulator*. Dens arkitektur følger både det logiske, prosessperspektivet og det fysiske perspektivet 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 kjørbare .jar - fil.

5.2.1 Simulasjon av brukerforespørsler

Et klientprogram, skrevet i Javascript, genererer og sender forespørsler til det simulerte produksjonsmiljøet DBUpgradinator testes i. Dette skriptet kjøres i kjøretidsmiljøet NodeJS, fra en norsk datamaskin i Trondheim. Klientprogrammet kalles for *WebAppSimulator*². Dets hovedoppgave er å simulere en kontinuerlig serie med forespørsler som jevnfordeles blant applikasjonsinstansene. Programmets funksjonalitet kan i korte trekk inndeles i tre satser:

1. Programmet sender klyngen av tjenerne en serie av POST-forespørsler, slik at nye aggregater lagres på hver av de tomme nodene i databasen. Den totale datamengden som sendes over TCP/IP er begrenset til under én gigabyte, slik at testingen ikke tar altfor lang tid. 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 applikasjonsloggene til hver av tjenerinstansene transformeres til én 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 applikasjonstjenerne én HTTP-forespørsel per ID i listen som ble opprettet i forrige steg. Hver forespørsel kan enten være en PUT - forespørsel eller en GET - forespørsel, og det er funksjonen `Math.random()` som bestemmer hvilket.

For å generere data til POST - forespørslene i steg 1 og PUT - forespørslene i steg 3, brukes en kommaseparert seed-fil som inneholder en tabell der hver rad har en strengverdi for fornavn, etternavn, telefonnummer, gateadresse, poststed, delstat, postkode, og land³.

²Kildekoden til simulatoren er tilgjengelig fra URL <https://github.com/vegardbb/DBUpgradinator/tree/master/demo/webappsimulator>.

³Navn - og adresser i seed-filen stammer fra gratistjenesten Fake Name Generator, som tilbyr inntil 100 000 navn og adresser i en kommaseparert fil per bestilling. URL: <https://www.fakenamegenerator.com/order.php>

Simulasjonsprogrammet leser inn seedfilen og grupperer hver av kolonnene i tabellen til et sett med lister, der hver liste er assosiert med ett av de åtte tidligere nevnte kolonnenavnene.

Ved hjelp av pakken ”node-fetch” genererer skriptet HTTP-forespørsler asynkront, det vil si at det ikke venter på svar fra applikasjonstjenerne for hver tilsendt forespørsel, før den neste sendes. En funksjon i skriptet, kalt `chooseRandom`, returnerer et pseudotilfeldig valgt element fra et liste-parameter. Denne funksjonen brukes til å velge en tilfeldig applikasjonstjener som skal motta en forespørsel, fra en liste av URL-er. Den samme funksjonen kalles på for å generere hvert attributt i et aggregat som enten skal postes eller oppdateres i databasen. Prosedyren for generering av PUT - forespørsler skiller seg ut fra generering av POST-forespørsler og med at PUT-endepunktet til WebShopSimulator sitt ReST-API krever et *id*-parameter.

Skriptet bruker et separat loggeobjekt, som instansieres ved hjelp av pakken Winston⁴. Loggeobjektet logger responsen applikasjonstjenerne sender tilbake for hver genererte forespørsel.

5.2.2 Programvare på tjenerne

Kildekoden til WebShopSimulator er åpent tilgjengelig på DBUpgradinator sitt GitHub - repositorium: <https://github.com/vegardbb/DBUpgradinator/tree/master/demo/WebshopSimulator>. Programmet er et Java - prosjekt som bruker DBUpgradinator til å oppgradere sin datamodell. Programmet er en webapplikasjon som eksponerer et ReST-API til mange nettleserklinter. Klassediagrammet i figur 5.2 viser den statiske relasjonen mellom kildekode til DBUpgradinator og kildekode til WebShopSimulator.

For å realisere ReST-APIet til den typiske webapplikasjon DBUpgradinator er ment å operere på (se også figur 4.1), ble Java-pakken Spark⁵ benyttet. Spark er et mikrorammeverk laget for at utviklere kan skrive en webapplikasjon sentrert rundt mikrotjenestemønsteret lettere og raskere. Applikasjonen har i hovedsak tre ruter i sitt grensesnitt:

GET /api/:id?schema=x Applikasjonen leser det aggregatet i databasen som har nøkkelprefiks lik *:id* og skjemasuffiks lik *x*

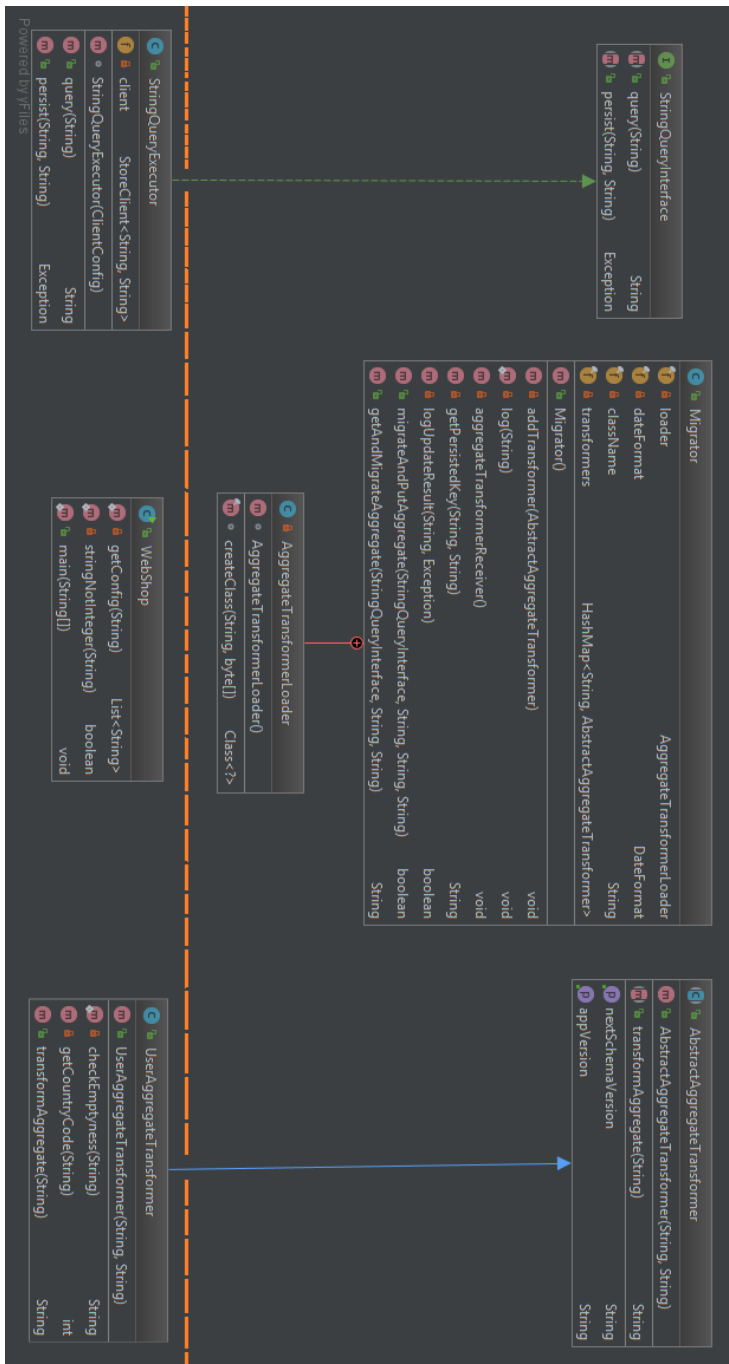
PUT /api/:id?schema=x Applikasjonen oppdaterer det aggregatet i databasen som har nøkkelprefiks lik *:id* og skjemasuffiks lik *x*

POST /api?schema=x Applikasjonen oppretter et nytt aggregat (ut i fra forespørselens ”body” - variabel) i databasen som tilhører skjemasversjon *x*

I kildekode til WebShopSimulator er det også skrevet en separat klasse, som utifra en instans av `VoldemortClientConfig` instansierer et `StoreClient` - objekt. Dette objektet koordinerer og utfører spørringer over TCP - forbindelser med hver av databasenodene i

⁴Publisert på npm. URL: <https://www.npmjs.com/package/winston>

⁵Må ikke forveksles med Apache Spark, som er et distribuert dataprosesseringsverktøy ment for stordatasystemer.



Figur 5.2: Klassediagram av WebShopSimulator. Den gule, stiplede linjen markerer et skille mellom klassene til DBUpradinator og klassene til WebShopSimulator.

Voldemort-instansen som kjører i testmiljøet. Selve klassen i WebShopSimulator implementerer grensesnittet `StringQueryInterface`, som er listet opp i kodeoppføring 4.8.

Artefaktene benevnt i 4.6 som "DBUpgradinator.jar" og "WebShopSim.jar" ble bygget i utviklingsmiljøet IntelliJ IDEA. I dette programmet kan man definere underprosjekter, eller "moduler", innen et Java-prosjekt. Enhver modul har sin egen katalog for å ha kildekode i. Avhengigheter, det vil si artefakter som inneholder klasser som importeres inn i kildekoden, registreres i en modul. Før en `.jar`-fil kan kompileres og bygges, må en artefakt defineres. IntelliJ knytter hver artefakt til en modul. Slik kan IntelliJ legge ved de `.jar`-filene artefakten trenger for å kunne fungere ordentlig.

DBUpgradinator er utstyrt med en enkel, hjemmelaget loggfunksjon. Denne loggeren påkalles hver gang et aggregat er blitt migrert til et nytt skjema, og hver gang et aggregat er blitt bekreftet skrevet til disk av databasenodene. Loggfunksjonen skriver et klokkeslett på formen `åååå-mm-dd hh:mm:ss`. Ved hjelp av loggingen kan man teste hvordan den levende datamigrasjonen påvirker en applikasjons evne til å betjene forespørsler til dens klienter.

5.2.3 Konfigurasjon av Voldemort-instans

Databaselyngens modus operandi reguleres av tre forskjellige konfigurasjonsfiler:

Kodeoppføring 5.2: `cluster.xml` - filen definerer de fire nodene i databaselyngen til testmiljøet.

```
1 <cluster>
2   <name>test</name>
3   <server>
4     <id>0</id>
5     <host>167.99.84.243</host>
6     <http-port>9630</http-port>
7     <socket-port>7666</socket-port>
8     <admin-port>7665</admin-port>
9     <partitions>0, 23, 7, 22, 10, 18, 5, 25, 35, 38, 41, 44, 45, 46,
10      48, 53</partitions>
11   </server>
12   <server>
13     <id>1</id>
14     <host>188.166.103.205</host>
15     <http-port>9063</http-port>
16     <socket-port>7670</socket-port>
17     <admin-port>7669</admin-port>
18     <partitions>15, 20, 29, 4, 30, 24, 8, 31, 32, 55, 39, 54, 42, 50,
19      37, 52</partitions>
20   </server>
21   <server>
22     <id>2</id>
23     <host>206.81.31.127</host>
24     <http-port>9306</http-port>
25     <socket-port>7674</socket-port>
26     <admin-port>7673</admin-port>
```



```

25     <partitions>19, 27, 28, 26, 2, 17, 1, 11, 47, 57, 61, 36, 62, 56,
26         40, 63</partitions>
27 </server>
28 <server>
29     <id>3</id>
30     <host>206.81.31.128</host>
31     <http-port>9360</http-port>
32     <socket-port>7678</socket-port>
33     <admin-port>7677</admin-port>
34     <partitions>3, 6, 9, 12, 13, 14, 16, 21, 51, 59, 60, 58, 34, 49,
35         33, 43</partitions>
36 </server>
37 </cluster>

```

Klyngekonfigurasjonen holder på variabler som er konsistent like for samtlige noder (det være seg kjørende databasetjenerprosesser) i Voldemort-instansen. Eksempler på slik informasjon inkluderer nodenes vertsnavn, id, og tilkoplingsport for TCP-forbindelser med **StoreClient** - instanser (Kreps, 2009). I `cluster`-konfigurasjonen defineres også alle partisjonene som er spredd utover nodene.

Fordelingen av disse partisjonene påvirker fordelingen av nøkler i den konsistente hashringen. Mens en Voldemort-instans er oppe kan disse partisjonene omfordes fritt, men de kan i sum aldri bli færre eller flere. De øvrige konfigurasjonsvariablene må være konstante gjennom hele databaseinstansens livsløp, og de må være like for alle nodene i klyngen (Kreps, 2009).

Hvis produksjonsmiljøet opererer med flere databasetjenere på samme datasenter, og samtidig kjøres på tvers av flere datasentre, går det an å definere soner innenfor klyngen. Dens funksjonelle verdi er å unngå at to replikaer for samme nøkkel havner på samme datasenter. I tilfellet med testmiljøet til WebShop hadde det vært hensiktsmessig å inndele klyngen i to soner: Nodene som er lokalisert i Frankfurt og nodene som er lokalisert utenfor Frankfurt. Dette er imidlertid ikke gjort for databaseklyngen WebShopSimulator opererer på.

Kodeoppføring 5.3: `server.properties` er filen som bestemmer oppførselen til den enkelte databasenode i databaseklyngen.

```

1 # The ID of *this* particular cluster node
2 node.id=1
3
4 max.threads=100
5
6 ##### DB options #####
7
8 socket.enable=true
9
10 # BDB
11 bdb.write.transactions=false
12 bdb.flush.transactions=false
13 bdb.cache.size=200M
14 bdb.one.env.per.store=true
15

```

```
16 #NIO connector settings.
17 enable.nio.connector=true
18
19 request.format=vp3
20 storage.configs=voldemort.store.bdb.BdbStorageConfiguration
```

En minimal tjenerkonfigurasjon inneholder én linje, og det er den som definerer id-en til noden. Dette er påkrevd, slik at noden vet hvilken nodekonfigurasjon fra `cluster.xml` som skal leses. Som vist i kodeoppføring 5.3, definerer tjenerkonfigurasjonsfilen kjøretidsegenskapene til persisteringsmotoren BDB Java, som alle de fire nodene tar ibruk for å lagre data.

Kodeoppføring 5.4: En STORE - fil som bestemmer hvordan data lagres i databasen.

```
1 <store>
2   <name>test</name>
3   <persistence>bdb</persistence>
4   <description>Distribuert datalager for testing av
5     DBUpgradinator</description>
6   <owners>vegardbbstud.ntnu.no, sveinbrantnu.no</owners>
7   <routing-strategy>consistent-routing</routing-strategy>
8   <routing>client</routing>
9   <hinted-handoff-strategy>consistent-handoff</hinted-handoff-strategy>
10  <replication-factor>3</replication-factor>
11  <required-reads>2</required-reads>
12  <required-writes>2</required-writes>
13  <key-serializer>
14    <type>string</type>
15  </key-serializer>
16  <value-serializer>
17    <type>string</type>
18    <schema-info>utf8</schema-info>
19  </value-serializer>
20 </store>
```

Quorum-konfigurasjonen bestemmes ikke i `cluster.xml`, da det noen ganger kan være ønskelig at noen aggregater skal kunne gjøres mer tilgjengelige enn andre. Quorum-parameterne R , W , og N defineres for hvert nøkkelverdilager i klyngen, altså defineres de i filen `test`, henholdsvis med nøkkelordene *required-reads*, *required-writes*, og *replication-factor*. I konfigurasjonen beskrevet av oppføring 5.4 kan man se at databasen fører en "strict quorum" - policy, jamfør kapittel 2.3.2.

`test` - filen er konfigurasjonsfilen til nøkkelverdi-lageret, hvis identifiserte navn er "test". I prinsippet kan en databaseklynge ha flere databaseinstanser, men man kan ikke kjøre spørringer på tvers av dem. Filen dikterer hvilken strategi nodene skal følge når de leverer antydede replikaaggregat til sine midlertidige adresser.

Kodeoppføring 5.4 definerer også serialiseringsmetoden nodene benytter når de lagrer aggregater mottatt fra en spørrekoordinator. I `WebshopSimulator` brukes serialiseringsmetoden "string", det vil si at ottettene noder mottar behandles som UTF8-kodet tekst. Aggregatene kodes til binært i selve applikasjonen, og er allerede kodet idet de blir be-

handlet av **StoreClient** - instansen. Databasekonfigurasjonen forblir uendret i løpet av testene som utføres.

5.2.4 Testmiljø

Deployment-diagrammet fra figur 4.6 beskriver testmiljøet DBUpgradinator skal testes i. De fire boksene refererer til fire forskjellige datasentre. Hver enkelt av de fire applikasjonsnodene gjestes av skyinfrastrukturleverandøren DigitalOcean, og alle er spredd utover tre forskjellige datasentre i London (LON1), Amsterdam (AMS3), og Frankfurt am Main (FRA1). I hvert datasenter betjenes én virtuell maskin, eller droplet, som DigitalOcean kaller dem.

Hver av de virtuelle maskinene modellert i figur 4.6 kjører samme operativsystem, versjon 8.10 av Debian GNU/Linux. Alle har hver én gigabyte med arbeidsminne. Hver av maskinene disponerer et eget SSD-lager med en kapasitet på 25 GB, og én separat, virtuell prosesseringsenhet.

Tabell 5.1: Oversikt over virtuelle datamaskiner brukt under testing av DBUpgradinator.

Vertsnavn	IP-adresse	ID	Datasenter
vbb-master2018-avery	167.99.84.243	0	LON1
vbb-master2018-black	188.166.103.205	1	AMS3
vbb-master2018-crabbe	206.81.31.127	2	FRA1
vbb-master2018-dolohov	206.81.31.128	3	FRA1

I tabell 5.1 oppgis vertsnavnene og IPv4 - addressene til hver av de virtuelle maskinene, samt hvilket datasenter hver av dem er opprettet i. Assosiasjonen mellom navnet til vertsmaskinene og datasentra er også gjengitt i figur 4.6.

5.3 Testresultater

Delkapitlet presenterer loggførte resultater fra tre tester der HTTP-forespørsler ble logget av DBUpgradinator og HTTP-responser ble logget av forespørselsgeneratoren på en datamaskin stasjonert i Trondheim.

5.3.1 Test 1: Oppretting av aggregater, uten migrasjon

I den første testen ble aggregater generert av frontendsimulatoren, beskrevet i delkapittel 5.1, for deretter å bli sendt til hver sin tilfeldig applikasjonstjener i hver sin forespørsel, hvis metode er "POST". Denne testen ble kjørt for å måle omtrent hvor mange skriveforespørsler klyngen kan håndtere samtidig, uten å migrere aggregatene som postes. Skriptet som simulerer applikasjons frontende velger pseudotilfeldig hvilken tjener som mottar

hver HTTP-forespørsel, hvis metode er "POST". "Frontenden" logger også alle responsobjektene den mottar fra applikasjonstjenerne. Responsloggen er oppsummert i tabell 5.2.

Tabell 5.2: Oppsummering av frontendapplikasjonsloggen fra test 1.

Dato	2.06.2018
Varighet	2 t; 44 min; 18 sek
Antall forespørsler	3587467
Antall 200-svar	3158592
Antall 500-svar	428875
Suksessandel	88.05 %
Feilandel	11.95 %

Denne testen ble avsluttet etter to timer og 44 minutter, av hensyn til størrelsen til loggdata som ble generert av forespørselskriptet og applikasjonstjenerne. Frontendprogrammet mottok i snitt omtrent 320 OK - responser per sekund. Litt over 11 prosent av forespørslene feilet.

Tabell 5.3: Persisterte aggregater hos databasenodene i testmiljøet under test 1.

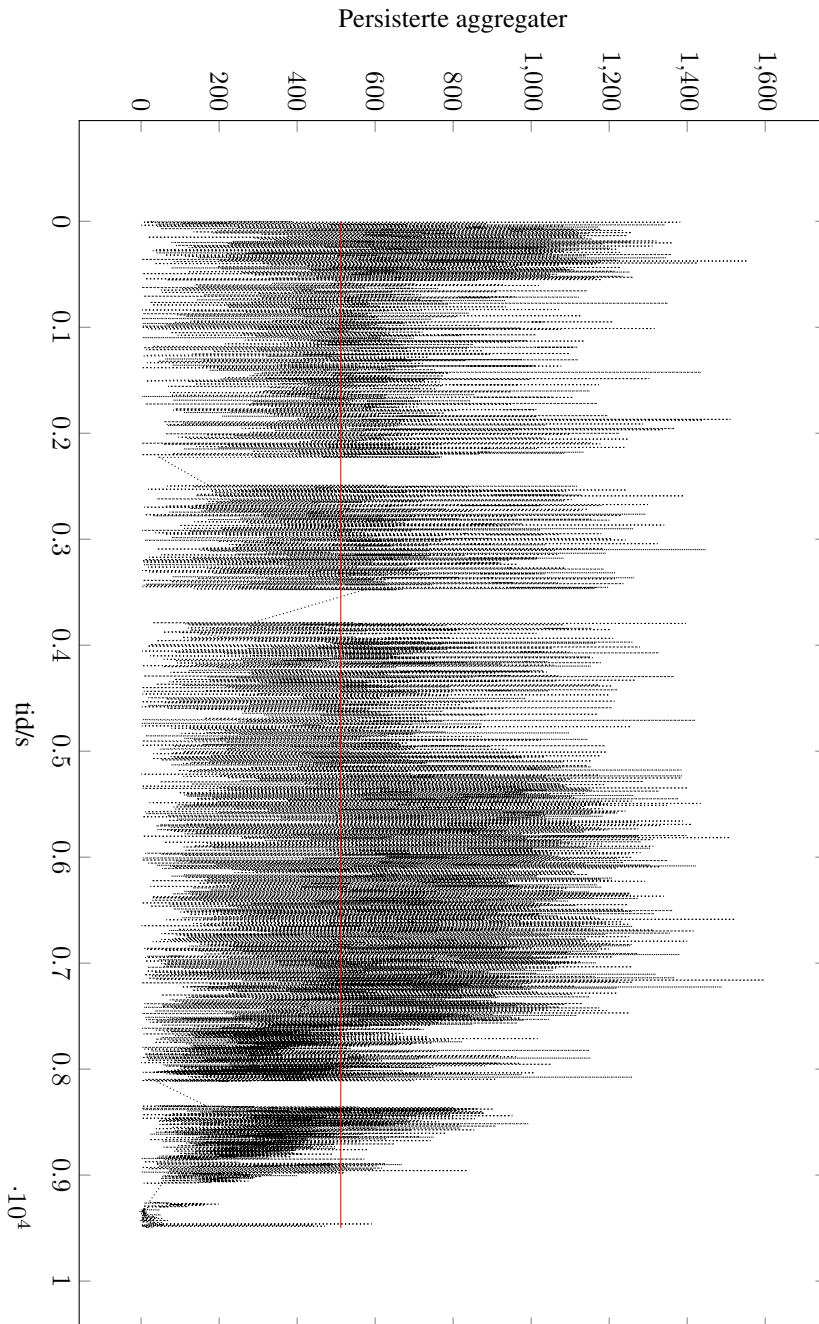
vbb-master2018-avery	791741
vbb-master2018-black	787906
vbb-master2018-crabbe	788961
vbb-master2018-dolohov	789984
Sum aggregater	3158592

Tabell 5.3 viser at antallet '200 AOK' svar oppgitt i 5.2 samsvarer med det opptalte antallet persisterte objekter, naturlig nok fordi alle forespørsler kaller på samme metode. I snitt ble 520 aggregater skrevet til disk per sekund av de fire databasenodene. Til sammenlikning har LinkedIn klart å betjene rundt 16 og et halvt tusen skriveforespørsler i sekundet på en klynge bestående av én eneste flertrådet tjener (Kreps, 2009).

Quroum-konfigurasjonen er den viktigste faktoren som bidrar til forskjellen i gjennomstrømming mellom testmiljøet og Linkedins resultater. Både replikeringsfaktoren (N), lesefaktoren (R) og skrivefaktoren (W) til klyngen beskrevet i "Performance"kapitlet er lik 1. De korresponderende innstillingene for klyngen DBUpgradinator ble testet i, er henholdsvis 3, 2, og 2.

I "Performance" - kapitlet til Project Voldemorts nettsider skriver Kreps (2009) at platelageret utgjør den ytelsesmessige flaskehalsen i en webapplikasjon. For å begrense feilraten, kan man derfor øke størrelsen på hurtiglageret i server.properties-filene. Fordelingen av antallet persisterte aggregater blant de enkelte databasenodene er for øvrig brukbart jevn.

Plottet i figur 5.3 er basert på DBUpgradinators persisteringslogg for aggregatene som ble opprettet i løpet av test 1. Loggtidspunktene er blitt gruppert sammen på sammenfallende sekund, og overført til en liste av (x,y) - tupler der x indikerer antallet sekunder siden testens start og y representerer antall aggregater persistert på tidspunkt x.



Figur 5.3: Graf som plottet grupperte tidspunkt for persistering av data.

5.3.2 Test 2: Lesing, oppdatering, og oppretting av aggregater, med migrasjon

I den andre testen forsøkte man å migrere aggregater postet under den første testen, gjennom både PUT-spørringer og GET-spørringer. Samtidig ble en stor mengde POST-spørringer, der det genererte aggregatet i forespørselens "body" følger den nye datamodellen, tilsendt klyngen. Testen var ment å simulere en migrasjonsprosess mens systemet betjente mange, hyppige forespørsler fra nye instanser av applikasjonen. På samme vis som i test 1 ble responsene fra tjenerne i produksjonsmiljøet logget til en fil.

Tabell 5.4: Oppsummering av frontendapplikasjonsloggen fra test 2.

Dato	3.06.2018
Varighet	3 min; 1 sek
Antall forespørsler	111914
Antall 200-svar	20871
Antall 500-svar	91043
Suksessandel	18.65 %
Feilandel	81.35 %

Tabell 5.5: Persisterte aggregater hos databasenodene i testmiljøet under test 2.

vbb-master2018-avery	6149
vbb-master2018-black	6286
vbb-master2018-crabbe	5822
vbb-master2018-dolohov	6233
Sum aggregater	24493
Antall migrerte	9206
Andel migrerte	37.59 %
Antall brukergenererte	15287
Andel brukergenererte	62.41 %

Denne testen ble brått avbrutt etter bare tre minutter. Årsaken til at testen ble avbrutt så tidlig var at tre av databasenodene ble permanent utilgjengelige som følge av minneoverflyt i arbeidssettet deres. I alle tre nodene ble minneoverflyten utløst i en buffer for asynkrone spørringer, hvilket var forbeholdt migrasjonsspørringer.

På dette tidspunktet brukte **Migrator**-klassen grensesnittet **CompletableFuture** til å opprette tråder som kjørte migrasjonslogikken asynkront i en separat gruppe tråder. Etter dette testavbruddet ble kildekoden til DBUpgradinator endret, slik at kallene til **CompletableFuture** og asynkrone tilbakekall erstattet med **Thread** - konstruktøren og et kall til *run*.

Tabell 5.4 viser en feilandel på åtte av ti sendte forespørsler. Dette kommer også av de fatale unntakene som ble kastet hos databasenodene. Tabell 5.5 sier at andelen migrerte aggregater DBUpgradinator registrerte er under 40 prosent. Årsaken til dette er at forespørslene som poster nye aggregat for det nye skjemaet, ikke behøver å migreres. I likhet med forri-

ge test er skriveoperasjonene nesten jevnt fordelt blant de fire tjenernodene. I snitt ble 89 aggregat persistert på de fire forskjellige databasenodene per sekund.

5.3.3 Test 3: Oppdatering av aggregater, med migrasjon

I den siste testen har en serie oppdateringsforespørsler (PUT) blitt sendt til applikasjons-tjenerne til WebShopSimulator. Forespørselsgeneratoren har lest inn et sett med IDer som den mottok fra tjenerne i test 1. For å generere et oppdatert aggregat, lagrer den et helt nytt objekt, jamfør framgangsmåten beskrevet i delkapittel 5.1.

Tabell 5.6: Oppsummering av frontendapplikasjonsloggen fra test 3.

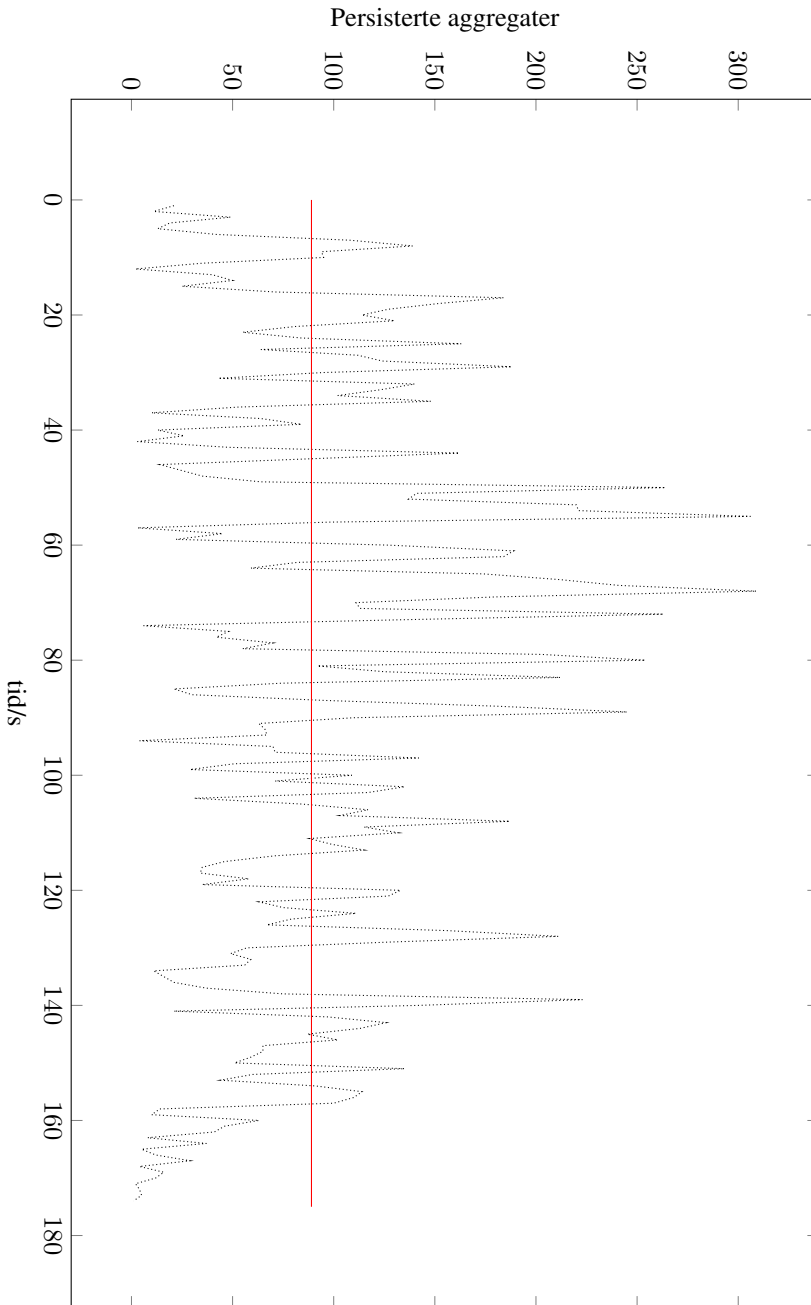
Dato	4.06.2018
Varighet	1 t; 27 min; 56 sek
Antall forespørsler	1019657
Antall 200-svar	703988
Antall 500-svar	315669
Suksessandel	69.04 %
Feilandel	30.96 %

Tabell 5.7: Persisterte aggregater hos databasenodene i testmiljøet under test 3.

vbb-master2018-avery	355203
vbb-master2018-black	363147
vbb-master2018-crabbe	368200
vbb-master2018-dolohov	342810
Sum aggregater	1429360
Antall migrerte	724831
Andel migrerte	50.71 %
Antall brukergenererte	704517
Andel brukergenererte	49.29 %

Ut ifra tabell 5.7 kan man utlede at i snitt ble 133 aggregat persistert på de fire forskjellige databasenodene per sekund. Som forventet forekommer rundt én migrasjonsspørring for hver spørring som oppdaterer et aggregat. Også i denne testen er suksessraten for forespørslene for lav for produksjonsmiljøstandarder. Kun sju av ti forespørsler ble besvart med en 200respons. Igjen er utilgjengelige databasenoder opphavet til statistikken, men av et annet slag enn hva tilfelle var i test 2.

Enhver databaseklient krever to bekræftelser på skriveoperasjoner før den returnerer en bekræftelse (skrive-ACK) til applikasjonen, og replikerer hvert aggregat til tre forskjellige noder ($R = 3$). For hver spørring applikasjonen gjør, gjør spørrekoordinatoren tre skrivespørringer. Når hver HTTP-forespørsel utløser en migrasjonsforespørsel på toppen av dette, blir arbeidslasten dobbelt så stor per aggregat. Hovedårsaken til at nodene gikk



Figur 5.4: Graf som plottes grupperte tidspunkt for persistering av data (lesinger ekskludert).

ned under denne testen, midlertidig om ikke annet, var at de ikke kunne respondere til flere TCP-forbindelser fra databaseklientene.

I alle tre plottene (figurer 5.3; 5.4; 5.5) kan man se voldsomme svinginger i antallet persisterede aggregater. Denne trenden har en programmatisk, bakenforliggende årsak. Forespørselsgeneratoren venter en fast tidsperiode for hver pulje/batch med forespørsler som tilsendes applikasjonstjenerne. Dette gjøres både av hensyn til nettverket forespørslene traverserer, og for ikke å overbelaste applikasjonstjenerne.

5.4 Diskusjon

Her følger en tolkning av testenes resultater og en kvalitativ sammenlikning med KVolve sin ytelsespåvirkning på Redis, jamfør testresultatene Saur et al. (2016) presenterer i kapittel V i sin artikkel. Før testresultatene drøftes og evalueres opp mot kvalitetskravene vil noen relevante erfaringer belyses, som ble notert da testene ble gjennomført.

5.4.1 Om flertrådede programvaresystemers blokkerende natur

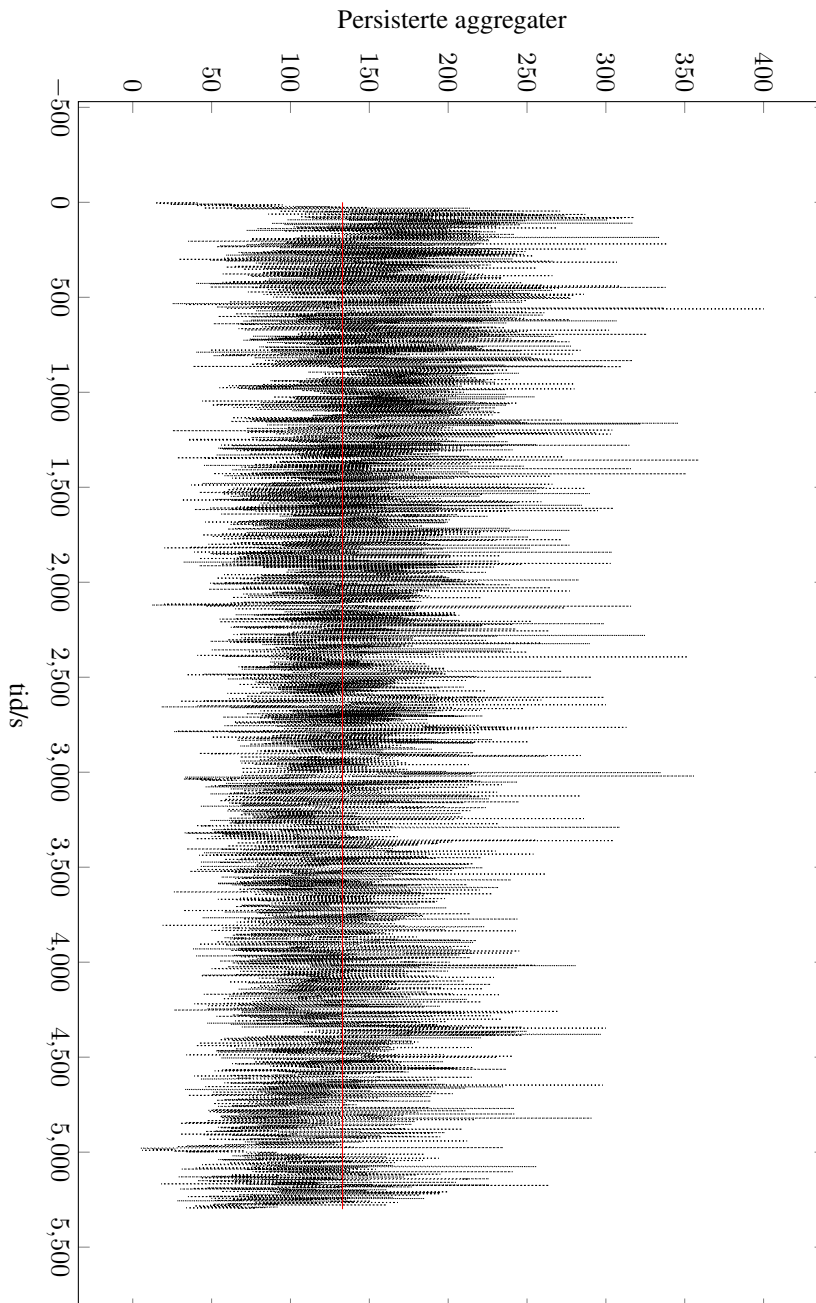
Migrator-klassen, beskrevet i delkapittel 4.4.2, skulle opprinnelig instansiere en egen tråd som konstant kjørte en while-løkke i funksjonen *aggregateTransformerReceiver* der den ventet på nye aggregattransformatorer i for mav binære filer. Denne funksjonen gjør et kall til metoden *accept* fra **ServerSocket**-klassen.

Effekten av dette metodekallet er at tråden som kjører funksjonen må aktivt vente på en innkommende dataenhet, for eksempel en matrise (byte[] i Java) av oktetter. Ved kjøretid kan denne tråden okkupere prosessoren store deler av kjøretiden. Dette ville ha resultert i en vesentlig reduksjon av systemets ytelse sammenliknet med et tilsvarende produksjonsmiljø der DBUpradator ikke hadde vært installert. Under en prøvetest viste proseshåndtøren htop at tråden som *aggregateTransformerReceiver* kjørte i brukte den ene prosessorkjernen i datamaskinen 95 % av kjøretiden.

5.4.2 Om konfigurasjonen av Voldemort-tjenerne

Hver virtuell maskin har kun én logisk prosessorkjerne og bare én gigabyte fysisk minne tilgjengelig for å oppbevare adresseområder til kjørende tråder i. Under kjøring av alle tre testene var langt ifra all minnet opptatt til enhver tid. Ifølge htop var på det meste rundt 560 MB opptatt på én av de virtuelle datamaskinene.

Minneforbruket når hverken applikasjonen eller databasenoden var oppe lå på rundt 40 MB. Det er derfor skjellig grunn til å anta at det hadde gått fint an å doble antallet tråder (max.threads) i tjenerkonfigurasjonen server.properties (se kodeoppføring 5.3), uten å bruke opp hele kapasiteten til primærminnet på de virtuelle maskinene.



Figur 5.5: Graf som plottet grupperte tidspunkt for persistering av oppdaterte data (skriveoperasjoner for migrasjonene er ekskludert).

Videre burde forespørsels - simulatoren ha sendt det samme antallet forespørsler om gangen. Ettersom testmiljøet i utgangspunktet er primært ment for å demonstrere at DBUppgradinator oppfyller sine funksjonelle krav, har denne suboptimale konfigurasjonen lite å si for konklusjonen.

Kodeoppføring 5.5: De første 50 linjene i den applikasjonsloggen til tjeneren vbbmaster2018crabbe (se tabell 5.1) fra test 2.

```

1 INFO @ 2018-06-03 19:13:06 - Received aggregate transformer for schema x
2 INFO @ 2018-06-03 19:22:36 - Persisted key a03d6b11-0adb-4790-af1b-0b74da4ff46f:y
3 INFO @ 2018-06-03 19:22:36 - Persisted key f58e6db5-fc98-40c7-b260-ce8ace6deb37:y
4 INFO @ 2018-06-03 19:22:36 - Persisted key ffc1d627-2aa8-4d5e-84cf-400d996390a2:y
5 INFO @ 2018-06-03 19:22:36 - Persisted key f0dd8e67-75f8-468c-b034-3155936476f7:y
6 INFO @ 2018-06-03 19:22:36 - Persisted key 5eff3261-8ec0-42eb-9dcc-5eb9bd9e149f:y
7 INFO @ 2018-06-03 19:22:36 - Persisted key 2e1b0b23-6d34-43ca-86bd-3ca87129c001:y
8 INFO @ 2018-06-03 19:22:36 - Persisted key e392bc7c-cf3e-4dbe-92c2-ce8ad4d36f4:y
9 INFO @ 2018-06-03 19:22:36 - Persisted key 364e5637-0f68-4799-9a07-6b0f23081401:y
10 INFO @ 2018-06-03 19:22:37 - Persisted key 4fc88c81-6dcb-47c1-96bf-b8ce7eed70f0:y
11 INFO @ 2018-06-03 19:22:37 - Persisted key 520d27df-7cf4-4756-bb1e-a7279cf462f6:y
12 INFO @ 2018-06-03 19:22:37 - Persisted key fccd3366-9d9f-4997-af6b-51104be5fee9:y
13 INFO @ 2018-06-03 19:22:37 - Persisted key 00951196-a821-4341-9430-2e0a6a758160:y
14 INFO @ 2018-06-03 19:22:36 - Persisted key 6b6c44ef-af67-4d07-bd2a-0cc4990fe05d:y
15 INFO @ 2018-06-03 19:22:37 - Persisted key 4c7b6c62-fca2-4c9c-b734-af49eda929a6:y
16 INFO @ 2018-06-03 19:22:37 - Persisted key c07ce7c9-b4c1-42d0-ae42-8da83ecc256b:y
17 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    59ae21bf-faa1-4842-9ab7-7c95f1c9331b:x to 59ae21bf-faa1-4842-9ab7-7c95f1c9331b:y
18 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    d630c921-f508-4551-9527-c9491684622b:x to d630c921-f508-4551-9527-c9491684622b:y
19 INFO @ 2018-06-03 19:22:37 - Persisted key 59ae21bf-faa1-4842-9ab7-7c95f1c9331b:x
20 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    09d042f2-d817-4760-8916-30f45b427c39:x to 09d042f2-d817-4760-8916-30f45b427c39:y
21 INFO @ 2018-06-03 19:22:37 - Persisted key 11547285-57ee-472d-b865-fi7b49a1d6f1:y
22 INFO @ 2018-06-03 19:22:37 - Persisted key 3f16dfb8-faeb-4cf6-9063-65254b995260:y
23 INFO @ 2018-06-03 19:22:37 - Persisted key 39695220-57db-4bc4-9f19-d2572331d51e:y
24 INFO @ 2018-06-03 19:22:37 - Persisted key 204160a8-16f3-4810-995f-4a77e1daf2e5:y
25 INFO @ 2018-06-03 19:22:37 - Persisted key 91b53285-db85-4fce-b465-b6aeb3bf98b:y
26 INFO @ 2018-06-03 19:22:37 - Persisted key 811bb422-0613-4534-89b8-8ef3ae6ab614:y
27 INFO @ 2018-06-03 19:22:37 - Persisted key 80103ddb-bfda-4ee9-ab20-4cdfs1c9aae3:y
28 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    1a80ecc1-9283-4c8d-8d33-7b72f4857260:x to 1a80ecc1-9283-4c8d-8d33-7b72f4857260:y
29 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    93c33378-1cbf-4d42-acb6-e3f108e585d7:x to 93c33378-1cbf-4d42-acb6-e3f108e585d7:y
30 INFO @ 2018-06-03 19:22:37 - Persisted key c5bb9d58-33e2-4a0a-b65a-23107b70101c:x
31 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    a8804429-2993-42a3-9535-c43c55820520:x to a8804429-2993-42a3-9535-c43c55820520:y
32 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    1a5bbc47-c9ee-4f4c-8527-334a6230f8f7:x to 1a5bbc47-c9ee-4f4c-8527-334a6230f8f7:y
33 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    a5364539-bada-468f-b36e-d050aeb83929:x to a5364539-bada-468f-b36e-d050aeb83929:y
34 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    a345c42e-81ab-4a79-bac2-99731cdb5461:x to a345c42e-81ab-4a79-bac2-99731cdb5461:y
35 INFO @ 2018-06-03 19:22:37 - Persisted key 1a80ecc1-9283-4c8d-8d33-7b72f4857260:x
36 INFO @ 2018-06-03 19:22:37 - Persisted key a5364539-bada-468f-b36e-d050aeb83929:x
37 INFO @ 2018-06-03 19:22:37 - Persisted key cb3ec8a0-0495-4e81-bb54-14236a0ed842:y
38 INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key
    493f88a1-33bf-4bcc-839f-102bb34489a3:x to 493f88a1-33bf-4bcc-839f-102bb34489a3:y
39 INFO @ 2018-06-03 19:22:38 - Persisted key 605e6a1f-39c8-4d25-9660-52d191e3eed5:y

```

```
40 INFO @ 2018-06-03 19:22:38 - Persisted key 0321a52a-05c5-45b7-bf02-768b471cc401:x
41 INFO @ 2018-06-03 19:22:38 - Persisted key 32e35502-7dbe-441d-b730-7550d6b5c3cd:y
42 INFO @ 2018-06-03 19:22:38 - Migrated aggregate with key
    30932a85-c8b3-4863-99fb-f70c8da6f777:x to 30932a85-c8b3-4863-99fb-f70c8da6f777:y
43 INFO @ 2018-06-03 19:22:38 - Persisted key b432c9b9-5a84-4412-8220-f5e1709e9c6b:y
44 INFO @ 2018-06-03 19:22:38 - Migrated aggregate with key
    0321a52a-05c5-45b7-bf02-768b471cc401:x to 0321a52a-05c5-45b7-bf02-768b471cc401:y
45 INFO @ 2018-06-03 19:22:38 - Migrated aggregate with key
    9507a355-f53e-4962-ac00-ef353e9c8ef3:x to 9507a355-f53e-4962-ac00-ef353e9c8ef3:y
46 INFO @ 2018-06-03 19:22:38 - Persisted key 63282b64-c582-49c3-9774-f3aabd0f518b:x
47 INFO @ 2018-06-03 19:22:38 - Persisted key c7a6c395-e269-48a5-9eff-514c0a9ca57c:y
48 INFO @ 2018-06-03 19:22:38 - Persisted key 63e937a9-2414-4af6-9086-23744f04bc09:y
49 INFO @ 2018-06-03 19:22:38 - Persisted key d01a7279-0a1d-4f77-a05d-b661b8350bb8:y
50 INFO @ 2018-06-03 19:22:38 - Persisted key 41701182-d9fc-49a6-bd41-a18848900c74:y
```

5.4.3 Erfaringer fra test 2

Man kan konkludere med at DBUpgradinators klasser fungerer slik de er ment å fungere, ved å se på DBUpgradinator sin applikasjonslogg på hver av de fire applikasjonstjenerne fra den noe korte perioden i test 2. Deler av alle fire loggene er samlet sammen i kodeoppføring 5.5. I denne loggen kan vi se på den aller første linjen at I/O - operasjonen som laster inn en aggregattransformator fra en allerede lagret fil fungerte feilfritt.

Dermed har feltvariabelen ”transformers” ett objekt i sin nøkkelverdiliste. Det objektet er en instans av **AbstractAggregateTransformer**, som inneholder en konkret implementasjon av den abstrakte metoden *transformAggregate*, som tilhører subclassen **app.-UserAggregateTransformer**. Objektets oppgave er å kopiere aggregater for skjema *x* til ekvivalente aggregater for skjema *y*.

På linje 17 i kodeoppføring 5.5 står følgende: ”INFO @ 2018-06-03 19:22:37 - Migrated aggregate with key 59ae21bf-faa1-4842-9ab7-7c95f1c9331b:x to 59ae21bf-faa1-4842-9ab7-7c95f1c9331b:y”. Denne logglinjen sier at aggregatet med id ”59ae21bf-faa1-4842-9ab7-7c95f1c9331b” har blitt migrert fra skjema *x* til skjema *y*, så fra og med det oppgitte tidspunktet eksisterer disse to aggregatene med lik UID-prefiks og ulik skjemasuffiks i databasen. Liknende logglinjer er å finne på linjer 18, 20, 28, 29, 31, 32, 33, 34, 38, 42, 44, og 45.

Vi kan også finne eksempler på suksessfulle oppdateringer av aggregater opprettet under skjema *x*. Den første registrerte oppdateringsversjonen finner vi på linje 19 i kodeoppføring 5.5. De øvrige bevisene på at oppdateringer ble skrevet på aggregater innen det gamle skjemaet, *x*, er på linjer 30, 35, 36, 40 og 46. Alle de øvrige ikke-nevnte linjene i kodeoppføring 5.5 er opprettelser av nye aggregater i det nye skjemaet. De 16 første linjene er alle POST - linjer.

5.4.4 Erfaringer fra test 1 og 3

For å anslå ytelsestapet til applikasjonen sammenliknes resultatene fra test 1 med test 3. Den gjennomsnittlige gjennomstrømmingen (eng. "throughput") for alle fire tjenerne samlet, målt i antall behandlede skriveforespørsler per sekund, ble for test 1 kalkulert til å være 512. I test 3 ble gjennomstrømmingen regnet ut til 133 forespørsler i sekundet. I forhold til frekvensplottene i figurer 5.3 og 5.5 er det kalkulerte gjennomsnittet visualisert som den røde streken trukket gjennom sentrum av "bølgen".

Statistikken tilsier at gjennomstrømmingen av forespørsler til applikasjonen under migrasjon er omtrent en fjerdedel av dens gjennomstrømming når den ikke gjennomgår datamigrasjon. Det intuitive resultat er at applikasjonen prosesserer 50 % færre forespørsler per tid, ettersom hver skriveforespørsel forårsaker en programmatisk initiert ditto på en annen nøkkel. I virkeligheten er forholdene mer komplekse. Forklaringen på dette ytelsestapet på 75 % er forankret i databasekonfigurasjonen.

I konfigurasjon 5.2 er "routing" - strategien satt til "client". Med denne innstillingen er det **StoreClient** - instansen i applikasjonens kjøretidsmiljø som hasher nøkler inn i den logiske hashringen. Det er altså databaseklienten som sender og koordinerer replikeringsforespørsler til de fire databasenodene i klyngen. Når hver oppdatering til aggregater for det gamle skjemaet også trigger en skriveforespørsel på en ny nøkkel vil det si at i alt $2 * 3 = 6$ replikeringsoperasjoner bli koordinert for hvert PUT - kall.

I retrospekt er det sannsynlig at ytelsestapet skyldes en kombinasjon av den statiske databasekonfigurasjonen som ikke tålte doblingen av spørringslasten under migrasjonstesten, og doblingen av nettverkstrafikk mellom databasenodene og applikasjonstjenerne under migrasjonstesten. At disse er årsakene til testresultatene er svært plausible, den økte feilraten i forespørselsbehandling tatt i betraktning.

Hvis DBUpgradinator hadde operert som en egen **Store** - subklasse i Project Voldemort, så ville man ha sluppet unna doblingen av antallet replikeringsforespørsler ved levende migrasjon. Man ville også ha sluppet å måtte midlertidig doble taket på antallet tråder i server.properties - konfigurasjonsfilen på hver databasenode, samt slå hver av disse nodene av og på for å få endringene i konfigurasjonen til å tre i kraft. Delkapittel 6.3 skisserer en slik oppgave i konklusjonen.

5.4.5 DBUpgradinator og KVolve

Opgavens modul er som tidligere nevnt noe annerledes enn KVolve, med tanke på databasearkitekturen hver av disse programmene er ment å operere på. KVolves databasesystem er mester-slave-replikert, og DBUpgradinator sitt databasesystem er quorum-replikert. Funksjonelt favner KVolve bredere enn DBUpgradinator, for KVolve kan oppdatere applikasjonslogikk, ved hjelp av verktøyet Kitsune. Under deres eksperimenter på KVolve, oppdaterte Saur et al. (2016) data og programvare hos i) RedisFS og ii) Amico.

Funksjonelt sett er DBUpgradinator annerledes i den grad at den oppretter nye nøkler ved datamigrasjon, og tillater gamle applikasjonsversjoner å lese og skrive til nøkler opprettet

innen det gamle dataskjemaet. Slik fasiliterer DBUpgradinator A/B - testing på ett og samme nøkkilverdilater. Dette er ikke tilfelle i KVolve, som under en oppgradering stenger alle forbindelser med tilkoblede klienter hvis versjon ikke er den nyeste (Saur et al., 2016, p. 5).

Saur et al. (2016) sine evalueringer resulterte i gjennomstrømningsrater på over 15 000 spørringer per sekund. Denne målingen ble gjort både når RedisFS og når Amico ble oppgradert. Det må påpekes at disse eksperimentene ble gjort på en lokal klynge med bare én databasenode, i likhet med Krepes (2009) sine tall for skrive - og lesegjennomstrømming (se "Performance"-siden), som også ligger på over 15 000. Siden evalueringen av DBUpgradinator ble utført i en faktisk databaseklynge med fire noder er det ikke relevant å sammenlikne resultatene presentert i forrige delkapittel med (Saur et al., 2016, p. 8) sitt antall spørringer per sekund.

5.4.6 Om `getAndMigrateAggregate` - metoden i DBUpgradinator

Metoden i kodeoppføring 4.6 er skrevet slik at den automatisk migrerer aggregater fra det gamle skjemaet såfremt aggregater med det gamle skjemaet sitt suffiks i sin ID blir forespurt. Det samme gjøres ikke i det omvendte tilfelle, når en applikasjon med den nye versjonen aksesserer samme aggregat-ID for sitt skjema. Migrator - klassens GET - metode (`getAndMigrateAggregate`) bør omformuleres til å ta høyde for aksesser fra den nye versjonen også.

I kodeoppføring 4.6 kan man telle i alt tre databasekall: Ett for å lese aggregatet i det gamle skjemaet, som er etterspurt av nettleserklienten, ett for å kontrollere at det nevnte aggregatet ikke allerede er migrert til det nye skjemaet, og ett for å skrive strengverdien til den migrerte aggregatnøkkelen. Denne migrasjonsfunksjonen er også skadelig for applikasjonens evne til å behandle mange HTTP-forespørsler fra klienter samtidig.

5.4.7 Evaluering av kravoppfyllelse

Oppsummering av funksjonelle krav. Tallene gjenspeiler listen over funksjonelle krav fra delkapittel 4.2.2:

1. Implementert og testet. DBUpgradinator kan migrere ethvert aggregat i et aggregatorientert databasesystem som tilhører et utdatert implisitt skjema.
2. Implementert og testet. Se kodeoppføringer 4.7 og 4.6.
3. Implementert. Se kodeoppføring 4.6.
4. Implementert og testet. Se kodeoppføring 4.5.
5. Implementert. Ved hjelp av denne strukturen på nøklene kan DBUpgradinator også brukes til å fasilitere A/B - testing av to forskjellige applikasjonsversjoner med to forskjellige skjemaersjoner.

6. Implementert.

Av kvalitetskravene er følgende attributter oppfylt:

Tilgjengelighet Delvis oppfylt. Testapplikasjonen WebShopSimulator er istand til å betjene mange forespørsler samtidig, men på grunn av at implementasjonen er utført på applikasjonsnivået i arkitekturen kan det ta lengre tid for applikasjonen å betjene like mange forespørsler når migrasjoner er igang.

Ytelse Ikke oppfylt. Tapet av gjennomstrømming er større enn 50 %, jamfør delkapittel 5.4.4.

Modulær løsning Oppfylt. Se kodeoppføring 4.8 og delkapittel 4.4.3.

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.

6.1 Om DBUpgradiator

I dette prosjektet ble en programvaremodul, kalt DBUpgradiator, 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 DBUpgradiator kan brukes til å migrere 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 delkapittel 4.4.2.

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 skjemaversjonen den opererer på, i tillegg til et ID-prefiks for å lese et aggregat.

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 brukerinitierte transaksjoner i en oppgradert applikasjonstjener, og derav skape flettekonflikter utvikleren må ta høyde for i sin egen kildekode. Sett fra applikasjonslagets 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 gjennomstrømmingsytelsen svekkes ytterligere når hver brukerinitiert spørring påkrever to ekstra spørringer når applikasjonen og dens skjema oppgraderes.

6.2 Akademisk interesse rundt levende datamigrasjon

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. DBUpgradorator demonstrerer i testapplikasjonen at dette er tilfelle.

Denne problemstillingen har ikke vært spesielt interessant fra et akademisk synspunkt, primært fordi det allerede finnes programvaremoduler som kan migrere dataobjekter i levende tjenerinstanser, til eksempel *cassandra-migration-tool-java*.

6.3 Implementere DBUpgradorator som en valgfri komponent av Voldemort

Et meget viktig designvalg er å bestemme hvilken del av den logiske arkitekturen migrasjon skal forekomme. I denne oppgaven har ansvaret for levende migrasjon av aggregater fra gamle skjema til nye tilfalt applikasjonsnivået. I et programvaresystem med en aggregatorientert datamodell er ikke databasesystemet klar over applikasjonens implisitte aggregatskjema, i motsetning til systemer hvis datamodell er relasjonell. I disse systemene er det databasesystemet som håndhever skjemaet. Da er det naturlig å forsøke å implementere levende datamigrasjon i applikasjonslogikken, snarere enn i databaselogikken.

Som vist i evalueringskapitlet av denne oppgaven, innebærer et slikt design ekstra nettverkstrafikk for hvert aggregat som aksesseres, enten for oppdatering eller lesing. Årsaken

til den ekstra nettverkstrafikken er at hvert migrerte aggregat også må replikeres. Når replikeringsfaktoren er 3, vil en PUT til et aggregat som samtidig migreres til neste versjon resultere i seks skriveoperasjoner i databaseklyngen.

Hvis migrasjonsmodulen hadde blitt implementert som en egen type NoSQL-tjener underordnet Project Voldemort, ville ikke tjenestetraffikken fordoblet seg hver gang systemet gjennomgikk en oppdatering. På den annen siden ville ikke den modulære oppgraderingsmodulen ha den enestående kvaliteten at den kan benyttes på andre aggregatororienterte databasesystemer enn Project Voldemort.

En passende prosjektoppgave på høsten kan være å kartlegge Voldemorts allerede implementerte støtte for datamigrasjon, så vel som resten av Voldemorts kildekode. En påfølgende masteroppgave kan da implementere en avart av DBUpgradinator i form av en egen **Store**-klasse i Voldemorts kildekode.

6.4 Implementere et kommandolinjegrensesnitt for DBUpgradinator

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 programmerer definerer en subklasse av `AbstractAggregateTransformer`, distribuerer en `.jar`-fil der nevnte subklasse er definert til hver av applikasjonstjenerne, og kompilerer en `.class`-filer på hver av dem.

Kommandolinjeprogrammet bør kunne kjøres i et skall fra enhver applikasjonsinstans der DBUpgradinator er installert. Dette konsollgrensesnittet vil være en vesentlig forbedring fra den nåværende kildekode. Saur et al. (2016) nedfeller et liknende mål for KVM i sine konklusjoner.

For enkelthets 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 allerede et eget serialiseringsformat, basert på Apache Avro, som kan migrere data. Det ville være interessant å undersøke hvilken datamigrasjonsmetode som gjør webapplikasjonen mest tilgjengelig for betjening av brukerforespørsler under datamigrasjon.

Bibliografi

- Bailis, P., Venkataraman, S., Franklin, M., Hellerstein, J., & Stoica, I. (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*, (pp.4). ACM.
- 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, (pp. 1–17)., New York, NY, USA. ACM.
- 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. Technical report, Technical Report CMU-PDL-09-106, Carnegie Mellon University.
- Dumitras, 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*, (pp.18). Springer-Verlag New York, Inc.
- Dumitras, T., Narasimhan, P., & Tilevich, E. (2010). To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *ACM Sigplan Notices*, volume 45, (pp. 865–876). ACM.
- 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. (2015). `cassandra-migration-tool-java/readme.md`.
- Hauer, P. (2015). Databases as a challenge for continuous delivery.
- Hicks, M. & Nettles, S. (2005). Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 27(6), 1049–1096.
- Kreps, J. (2009). Project Voldemort.
- Kruchten, P. (1995). The 4+1 View Model of Architecture. *Software, IEEE*, 12(6), 42–50.
- Lindholm, T. & Yellin, F. (1999). The Java™ Virtual Machine Specification - Concepts.
- 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.
- 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*, (pp. 213–228). Boston, MA.
- 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*, volume 67, (pp. 11–25). Seattle, WA.
- Pepitone, J. (2010). Why attackers can't take down amazon.com.
- Sadalage, P. & Fowler, M. (2013). Nosql distilled : a brief guide to the emerging world of polyglot persistence.
- Saur, K., Dumitraş, T., & Hicks, M. (2016). Evolving NoSQL Databases without Downtime. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (pp. 166–176).
- Schiller, K. (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.